Reducing Search Space in Code Smell Detection using Change History Information

Abu Rafe Md Jamil
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0722@iit.du.ac.bd

Asadullah Hill Galib
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0712@iit.du.ac.bd

Md Nurul Ahad Tawhid
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
tawhid@iit.du.ac.bd

Nadia Nahar
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
nadia@iit.du.ac.bd

***Abstract*—Detecting code smell in large-scale projects is a critical aspect of software maintenance. Typical code smell detection approaches search code smell in all source files, and this process continues in multiple phases of the development lifecycle. That process may computationally complex for real-life large-scale projects due to the vast size of the search space. In this study, a simple search space reduction approach is proposed for code smell detection based on a novel software evolution metric of change history information. The proposed approach is evaluated on 11 popular and large-scale projects from GitHub using code smells dataset of four code smells - Blob, Feature Envy, Divergent Change, Parallel Inheritance. Primarily, these four code smells are selected to explore the applicability of the proposed search space reduction approach. The results have shown that the proposed metric significantly reduces the search space while detecting a sound percentage of the actual code smell. It is also analyzed that this approach performs considerably better in detecting Blob, Feature Envy, and Divergent Change, while depicting relatively poor performance for Parallel Inheritance. In the future, other common code smells and more large-scale projects will be analyzed using this approach.**

***Index Terms*—software maintenance, software evolution, code smells, search space reduction, change history**

I. Introduction

A code smell is an indicator of the surface and normally leads to a deeper problem in the network [1]. Code smells are symptoms of poor design and choice of implementation. In some cases, developers under pressure might trigger these symptoms, such as installing urgent patches or simply making sub-optimal choices. Code smells block comprehensibility and may increase change and vulnerability [2]–[4]. Thereby, these smells must be carefully detected and monitored, and refactoring actions should be planned and carried out to deal with those whenever necessary. Otherwise, it would be problematic to system.

The first two authors contributed equally to this work. (Corresponding author: A. Jamil and A. Galib)

Several techniques of code smell detection have been proposed. Most of the literature deals with smell detection based on static analysis [5]–[10] and change history information [11]–[14]. Both types of detection techniques are computationally complex for large-scale projects. Mostly, these techniques analyze all files of a system for code smell detection. For a large-scale project, applying the techniques to all files and multiple phases of the project would be computationally complex.

To address this issue and speed up the detection of code smell, a novel search space reduction approach is proposed in this study. It presents an approach to speed up the detection of code smells in software systems, by reducing the number of files of the system that need to be considered by the smell detector. Here, a new metric based on change history information - NCPC (Number of Changes Per Commit) is introduced for reducing the search space. To the best of the authors' knowledge, it is the first work on search space reduction for effective code smell detection. This study seeks to answer the following two research questions (RQ) analytically in reducing the search space for code smell detection:

- **RQ1:** How to reduce the search space using change history information?
- **RQ2:** To what extent the whole search space reduction approach works well in detecting code smell?

This study is based on the hypothesis that if any file of a software project undergoes several changes through commits, then it may contain one or more code smells. Especially that smells which are intrinsically related to change history information, may be deduced from evaluating change history information. This hypothesis is supported by several prior works [2]–[4], [10], [15] Here, the hypothesis is tested on four code smells: Blob, Feature Envy, Divergent Change, and Parallel Inheritance using 11 popular and large-scale GitHub projects and code smells dataset [16]. Blob class or God class is a class containing a large number of attributes and methods. Feature Envy is a class that uses methods of another class excessively. Divergent Change is when many changes are made to a single class. Parallel Inheritance occurs where an inheritance tree relies on a separate inheritance tree by construction and has a special arrangement where one subclass of contingent inheritance depends on one subclass of another. The four code smells are chosen based on the availability of such kind of dataset and to examine the feasibility of the approach. Here, Blob and Divergent Change smells are closely related to the change-proneness of software artifacts according to their signs, symptoms, and reasons for the problem. The fact is also hinted by Olbrich et al. [4], [15]. In this work, the search space is reduced using a new metric called NCPC. The usefulness of the metric is evaluated by how well it can reduce the search space. Afterward, the effectiveness of the search space reduction is evaluated based on what percentage of the actual code smells are belonging to the reduced search space.

Experimental results show that the proposed approach can significantly reduce the search space while maintaining the percentage of detected code smell. This approach performs well for the smells - Blob, Feature Envy, and Divergent Change, except for the Parallel Inheritance code smell. A rationale for this variability raises intuitive discussion and direction.

The remainder of this paper is arranged according to the following. Section 2 describes the Methodology of this approach concisely. Section 3 describes the Experiments with defining the experiment setup and dataset. Section 4 presents the Result Analysis based on the research questions. Section 5 draws a Discussion regarding the result and the approach. Section 6 denotes the Threats to Validity of this study. Section 7 describes the Related Work. Section 8 presents the Conclusion that is drawn from the results and explains what the research findings lead to.

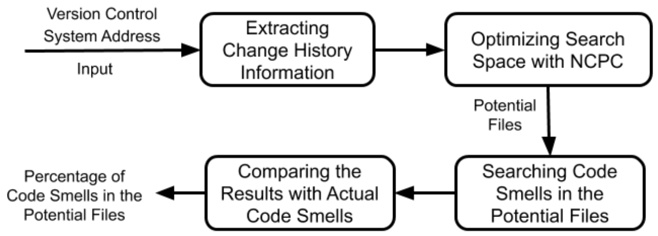
Version Control System Address
Input
Extracting Change History Information
Optimizing Search Space with NCPC
Potential Files
Searching Code Smells in the Potential Files
Comparing the Results with Actual Code Smells
Percentage of Code Smells in the Potential Files

Fig. 1. The proposed approach.

II. Methodology

This study is guided by a hypothesis based on the change proneness of a source-code file. The change history of the source-code file is extracted and analyzed according to the hypothesis. Afterward, for reducing the search space, a novel metric is introduced. For evaluation purposes, an evaluation approach is also defined. Figure 1 depicts an overview of the proposed approach.

A. Hypothesis

The hypothesis behind the research is as follows: *If a source-code file is change-prone, then the possibility of introducing code smell belonging to that file will be increased.*

That hypothesis is backed by some existing studies as well. Khomh et al. [2], [10] analyzed that the classes with code smells are more change-prone than others. Palomba et al. [3] also confirmed similar observations. In their study, it is found that classes affected by code smells have a statistically significant higher change-proneness. Olbrich et al. [4], [15] repeatedly found that classes that are infected with a God Class code smell get changed significantly more often.

TABLE I
Projects Considered in this Study

Project	git snapshot	Java files	Blob	Feature Envy	Divergent change	Parallel Inheritance
Apache Tomcat	398ca7ee	1494	5	2	1	9
Apache Cassandra	4f9e551	613	2	28	3	3
Apache Derby	562a9252	2422	9	0	0	0
Apache Commons Codec	c6c8ae7a	85	1	0	0	0
Apache Ant	da641025	1217	7	8	0	1
Apache Commons Lang	4af8bf41	236	3	1	1	6
Apache Commons IO	c8cb451c	200	2	1	0	1
Apache Commons Logging	d821ed3e	61	2	0	0	2
Android sdk	6feca9ac	1337	10	1	2	9
Google Guava	e8959ed0	1178	1	0	0	0
Eclipse Core	0eb04df7	4665	4	3	1	8

B. Change History Information Extractor

A change information extractor takes the version control repository as input. Different types of version control systems can be used, such as SVN, CVS, or git. The git is considered in this study and all the projects'change history are extracted from the git version control system. The extractor mines the versioning system log to extract the change history information. The versioning system logs can report code changes at the file level of granularity. In this study, any kind of source code file modifications and deletions are considered as changes. Newline insertions are not regarded as changes because insertions are very common in the development of a system which might not be a discerning characteristic in reducing search space for code smell detection. The extractor gives the following information as output-

- Changed filename
- Number of commits that it has changed
- Number of changes in the commits

This information is needed to define a threshold for reducing search space. The *Changed filename* is required to uniquely identify every single file.

C. Search Space Reduction Metric: NCPC

To reduce the search space and quantify the change-proneness, a novel and simple metric associated with the change history is proposed - Number of Changes per Commit

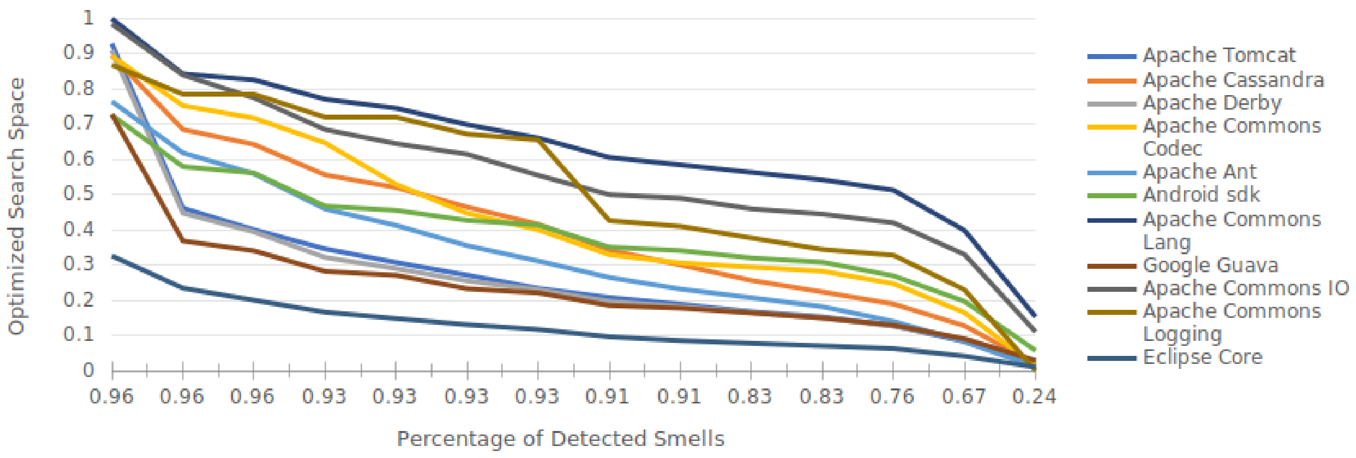

Fig. 2. Percentage of detected smells and optimized search space for Blob

(NCPC). NCPC can be calculated for a single file using the equation 1.

$$NCPC = \frac{number\ of\ changes\ in\ commits}{number\ of\ commits\ that\ file\ changed} \quad (1)$$

Here, the *number of changes in commits* denotes the total number of changes of a file in all commits. The *number of commits that file changed* denotes the total number of commits in which the file is changed. Also, line-based modifications and deletions are considered as changes in this study.

To speed up the reduction of search space, the NCPC metric is kept simple and straightforward. Considering the number of changes or size of the files might be misleading as it grows with the increase of the system and it varies context to context. But, changes per commit would be more reasonable as it denotes the variability of the code segment in a single commit. The rationale behind the idea is that many design flaws, requirement inconsistency, developer's lack of experience, or other factors may lead to frequent changes in the source code in a single commit. Consequently, code smell would also be introduced due to those pitfalls.

The metric NCPC is used as a threshold for removing files from the search space which are less change-prone. The files which are more change-prone are selected as the candidate files as those files may incite code smells. The higher the value of the Number of Changes per Commit (NCPC) for a file, the higher the change-prone it is. With the increase of NCPC value, the search space is reduced simultaneously as NCPC value induces constraints on the search space selection. But, here is a drawback. If the search space is reduced exceedingly using NCPC, then the percentage of code smells in the reduced search space will also be minimized. So, the trade-off between search space and detected code smell needs to be maintained. An insight on this trade-off depicts in the result analysis section.

D. Evaluation Approach

Palomba et al. [16] contributed an open dataset of several code smells. There are 243 instances of five types of code smells identified in 20 open-source software projects. The smells are - Blob, Feature Envy, Divergent Change, Parallel Inheritance, and Shotgun Surgery. In this study, the candidate files derived by using the NCPC metric are matched with the instances of the code smells dataset. If the file that contains an instance of code smells dataset belongs to the candidate files list, it will imply that the reduced search space also contains the smelly file. Otherwise, the smell is out of the reduced search space.

To evaluate the approach, the code smell belonging to the reduced search space with respect to the total number of code smells in the whole project is regarded as - *percentage of detected smells* and calculated using the equation 2.

$$percentage\ of\ detected\ smells = \frac{smells\ in\ the\ candidate\ files}{number\ of\ total\ smells} \quad (2)$$

Besides, it is also considered that how well the search space is reduced using NCPC. The reduced search space can be is calculated using the equation 3.

$$reduced\ search\ space = \frac{number\ of\ candidate\ files}{number\ of\ total\ files} \quad (3)$$

Evaluating the reduced search space would conducive to the first research question. The percentage of detected smells would help to analyze the second research question.

III. Experiments

In the experiments, a publicly available dataset is used. Besides, an experimental setup is defined briefly.

A. Dataset

In this study, 11 open source projects are used. The details of the projects are described in Table I. For validation purposes, the public code smells dataset provided by Palomba et al. [16] is used.

TABLE II
SEARCH SPACE ACCORDING TO NCPC

NCPC	Tomcat	Cassandra	Derby	Commons Codec	Ant	Android SDK	Commons Lang	Google Guava	Commons IO	Commons Logging	Eclipse Core
1	0.93	0.90	0.91	0.89	0.76	0.73	1.00	0.73	0.99	0.87	0.33
1.5	0.46	0.69	0.45	0.75	0.62	0.58	0.84	0.37	0.84	0.79	0.23
1.75	0.40	0.64	0.39	0.72	0.56	0.56	0.83	0.34	0.78	0.79	0.20
2.00	0.35	0.56	0.32	0.65	0.46	0.47	0.77	0.28	0.69	0.72	0.17
2.25	0.31	0.52	0.29	0.53	0.41	0.46	0.75	0.27	0.65	0.72	0.15
2.5	0.27	0.46	0.25	0.45	0.35	0.43	0.70	0.23	0.62	0.67	0.13
2.75	0.23	0.42	0.23	0.40	0.31	0.41	0.66	0.22	0.56	0.66	0.12
3.00	0.21	0.34	0.20	0.33	0.26	0.35	0.61	0.19	0.50	0.43	0.10
3.25	0.19	0.30	0.18	0.31	0.23	0.34	0.58	0.18	0.49	0.41	0.09
3.50	0.17	0.26	0.16	0.29	0.21	0.32	0.56	0.16	0.46	0.38	0.08
3.75	0.15	0.22	0.15	0.28	0.18	0.31	0.54	0.15	0.45	0.34	0.07

IV. RESULT ANALYSIS

According to the experiments, the results are thoroughly analyzed to answer the two research questions. Regarding the first research question, the search space reduction using the NCPC is examined. Regarding the second research question, the percentage of detected smells are evaluated to figure out the applicability of the proposed approach.

A. *Search Space Reduction (RQ1)*

For different NCPC values, the corresponding *reduced search space* for different projects are denoted in Table II. According to the table, the search space is significantly reduced with respect to different NCPC values. For example, in the project - Tomcat, when the NCPC is 1, the search space is 93%. Whereas, if the NCPC is increased to 3, the search space is reduced to 21%. The search space ranges from 0% to 100% for different NCPC values. So it can be concluded that the change history information metric - NCPC metric can be used for controlling and reducing the search space.

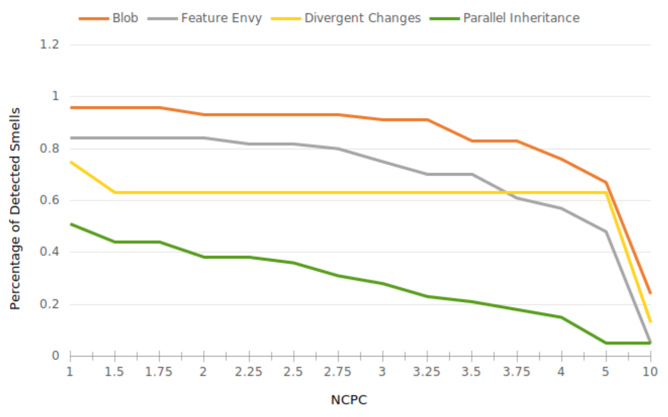

Fig. 3. Percentage of Detected Smells according to NCPC

B. *Percentage of Detected Smells (RQ2)*

For the NCPC values for different projects, the percentage of detected smells are depicted in Table III. For instance, considering the Apache Tomcat project, from 19% search space (NCPC =3.25), 91% Blob, 70% Feature Envy, 63% Divergent Change, and 28% Parallel Inheritance smells can be found.

From Figure 2, a relationship between the reduced search space and the percentage of detected smells for Blob in all projects can be inferred. As the search space decreases (NCPC values increase), the drop in the percentage of detected smells is not significant enough for a while. According to Table III, the percentage of detected smells remains steadily high (96% to 67%) concerning the increasing NCPC values (1 to 5) as well as the decreasing of the search space. But, for the NCPC value of 10, the percentage of detected smells drops significantly from 67% to 24%. Figure 2 and Figure 3 also depicted that phenomenon for all code smells: an immediate drop in the percentage of detected smells for the higher value of NCPC.

TABLE III
ACCURACY OF CODE SMELL DETECTION

NCPC	Blob	Feature Envy	Divergent Change	Parallel Inheritance
1.0	0.96	0.84	0.75	0.51
1.5	0.96	0.84	0.63	0.44
1.75	0.96	0.84	0.63	0.44
2.0	0.93	0.84	0.63	0.38
2.25	0.93	0.82	0.63	0.38
2.50	0.93	0.82	0.63	0.38
2.75	0.93	0.80	0.63	0.36
3.0	0.91	0.75	0.63	0.31
3.25	0.91	0.70	0.63	0.28
3.5	0.83	0.70	0.63	0.26
3.75	0.83	0.61	0.63	0.23
4.0	0.76	0.57	0.63	0.15
5.0	0.67	0.48	0.63	0.05
10.0	0.24	0.05	0.13	0.05

Search space reduction using NCPC while maintaining the percentage of detected smells is depicted in Figure 3 and Table III for different code smells. The figures show that the percentage of detected smells is high enough for Blob, Feature Envy, and Divergent Change and decreases slowly regarding the increasing values of the NCPC. However, for Parallel Inheritance, the percentage of detected smells is relatively low and decreases notably. This result infers that the search space reduction approach of this study is well-suited for Blob, Feature Envy, and Divergent Change while performing inadequately for Parallel Inheritance.

V. Discussion

According to the result analysis, the proposed approach performs well for Blob, Divergent Change, and Feature Envy except for Parallel Inheritance. As the change-proneness of artifacts is considered as the basis of this approach, the respective performance might be related to the change-proneness nature of the smell.

Blob class contains a large number of attributes and methods and it is hard to maintain and increase the difficulty to modify the software. So, it is quite reasonable that the change-proneness of this class might be high due to its size, functionality, and difficulty to maintain.

In the case of Divergent Change, it is an *intrinsically historical* smell [11]. Divergent Change is when many changes are made to a single class. According to its signs, symptoms, and rationale, it is also a highly change-proneness smell. This fact supports its relatively better performance.

For Feature Envy, as the smelly class is more interested in another class, the change-proneness of this smelly class might be high due to the inappropriate distribution of methods and attributes between the two classes. However, this rationale for its high change-proneness is arguable as the change-proneness of Feature Envy is neither analyzed in the literature nor explicitly derived from the definition and characteristics of it.

Parallel Inheritance occurs when an inheritance tree depends on another inheritance tree by composition, and these maintain a special relationship where one subclass of a dependent inheritance must depend on one a particular subclass of another inheritance. Parallel Inheritance is a kind of smell in which the detection process requires the entire system. As the approach of this study is based on single file change-proneness, it is clear that the approach performs poorly for this smell.

So, the change-proneness of code smell plays a crucial role in the effectiveness of the proposed approach. It would perform better for more change-prone smells rather than other smells. Also, smells dealing with the entire system cannot be analyzed properly using the proposed approach.

VI. Threats To Validity

The code smells dataset provided by the Palomba et al. [16] is used in this study for validation purposes. Therefore, the dataset and their threats validity are also inherited by this work.

In this study, all the parameters, such as different values of NCPC, time-frames of the projects used for validation are subject to internal threats to validity.

The external validity of this study can be called into question regarding generalizability as only 11 projects are used for evaluation. Moreover, only 4 code smells are investigated while overlooking other smells. So, the lack of projects and code smells considered in this study raise threats to external validity.

VII. Related Work

The issue of search space reduction for code smell detection is overlooked in the existing literature. Most of the code smell detection schemes rely on static analysis of source code. There are a few works concerning change history information.

In HIST [11], the authors used change-history information, such as classes' change history, methods' change history, etc. for detecting code smells. They defined heuristics to detect the various types of code smells, while the heuristic parameter calibration process varies smell to smell. They worked on five code smells - Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. They defined a historical detector for each smell by using association rule discovery or by analyzing classes/ methods co-changed with the suspected smell. Finally, HIST outperforms static code analysis for "intrinsically historical" smells such as Divergent Change, Shotgun Surgery, Parallel Inheritance, and performs as similar as static code analysis for Blob and Feature Envy smells.

Ratiu et al. [12] proposed an approach for detecting smell based on evolutionary information of problematic code components. They defined God Class and Data Class detection strategy by measuring the stability of classes and measuring the persistence of a design flaw. They performed multiple code analysis measurements of design problems during the propagation of code components. In another work, Lozano et al. [13] used historical information to assess the impact of code smells on software maintenance. They suggested that by using historical data, one can group bad smells according to their evolution, analyze evolution measurements of bad smells, relate flaws with later bugs.

Rao et al. [14] suggested a Design Change Propagation Probability (DCCP) matrix for a given design, represented different possible values of DCPP matrix as different conditions, and checked for the conditions satisfied by a given DCPP matrix and correlating these conditions with bad smells. They focused on Shotgun Surgery and Divergent Change code smells.

Apart from the change history base study, other works mostly focused on static analysis for various code smell detection. Static program analysis is the process by which the software is evaluated without execution [17]. Here, the analysis is carried out on some source-code versions. It is the easiest way to analyze code because it gives a thorough overview of the software quality and can identify several common coding problems [18]. In detecting code smell, several works [5]–[10] employed static analysis conclusively. However, all of these works deal with the full search space in detecting code smells.

In essence, to the best of the authors' knowledge, no prior work deals with the issue of search space reduction for code smell detection. Most of the works concerned about how to detect code smell and ignored the question: specifically where to search for code smell.

VIII. Conclusion

This study proposes a strategy to narrow down the candidate files for code smell detection. The strategy uses the number of changes in the files over the number of commits as the score. The paper also reports the evaluation of the effectiveness of the strategy. The result shows that the proposed approach not only reduces the search space but also maintains the percentage of detected smells regarding the actual smells. The proposed approach performs better for all considered smells except for Parallel Inheritance.

The most promising implication of reducing search space for code smell detection is in project management. By search space reduction, cost, time, and effort for detecting code smell in large projects can be reduced significantly. Considerable reduction of search space while costing a minimal loss of undetected code smells might be feasible for better project management.

In future work, more change history information can be considered. Besides, the implications of the proposed metric will be investigated in other contexts. For instance, this metric might be useful for evaluating the performance of the developers and for predicting the lifetime of individual projects as well as the whole project.

References

[1] Fowler, Martin. ”Refactoring: Improving the design of existing code.” 11th European Conference. Jyväskylä, Finland. 1997.

[2] Khomh, Foutse, Massimiliano Di Penta, and Yann-Gael Gueheneuc. ”An exploratory study of the impact of code smells on software change-proneness.” 2009 16th Working Conference on Reverse Engineering. IEEE, 2009.

[3] Palomba, Fabio, et al. ”On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation.” Empirical Software Engineering 23.3 (2018): 1188-1221.

[4] Olbrich, Steffen, et al. ”The evolution and impact of code smells: A case study of two open source systems.” 2009 3rd international symposium on empirical software engineering and measurement. IEEE, 2009.

[5] Travassos, Guilherme, et al. ”Detecting defects in object-oriented designs: using reading techniques to increase software quality.” ACM Sigplan Notices 34.10 (1999): 47-56.

[6] Simon, Frank, Frank Steinbruckner, and Claus Lewerentz. ”Metrics based refactoring.” Proceedings fifth european conference on software maintenance and reengineering. IEEE, 2001.

[7] Marinescu, R., 2004, September. Detection strategies: Metrics-based rules for detecting design flaws. In 20th IEEE International Conference on Software Maintenance, 2004, (pp. 350-359). IEEE.

[8] Lanza, Michele, and Radu Marinescu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science Business Media, 2007.

[9] Munro, Matthew James. ”Product metrics for automatic identification of” bad smell” design problems in java source-code.” 11th IEEE International Software Metrics Symposium (METRICS’05). IEEE, 2005.

[10] Khomh, Foutse, et al. ”A bayesian approach for the detection of code and design smells.” 2009 Ninth International Conference on Quality Software. IEEE, 2009.

[11] Palomba, Fabio, et al. ”Detecting bad smells in source code using change history information.” 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013.

[12] Rapu, D., et al. ”Using history information to improve design flaws detection.” Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.. IEEE, 2004.

[13] Lozano, Angela, Michel Wermelinger, and Bashar Nuseibeh. ”Assessing the impact of bad smells using historical information.” Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting. 2007.

[14] Rao, A. Ananda, and K. Narendar Reddy. ”Detecting bad smells in object oriented design using design change propagation probability matrix 1.” (2007).

[15] Olbrich, Steffen M., Daniela S. Cruzes, and Dag IK Sjøberg. ”Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems.” 2010 IEEE International Conference on Software Maintenance. IEEE, 2010.

[16] Palomba, Fabio, et al. ”Landfill: An open dataset of code smells with public evaluation.” 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 2015.

[17] Gomes, Ivo, et al. ”An overview on the static code analysis approach in software development.” Faculdade de Engenharia da Universidade do Porto, Portugal (2009).

[18] Tahmid, Ahmad, et al. ”Code sniffer: a risk based smell detection framework to enhance code quality using static code analysis.” International Journal of Software Engineering, Technology and Applications 2.1 (2017): 41-63.