International Journal of Engineering & Technology, 7 (2.27) (2018) 161-166

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Design of testing framework for code smell detection (OOPS) using BFO algorithm

Pratiksha Sharma [1] *, Er. Arshpreet Kaur [2]

[1] M. Tech Scholar, department of Computer Science Engineering, Chandigarh University, Gharuan, Mohali, Punjab India
[2] Assistant professor, department of Computer Science Engineering, Chandigarh University, Gharuan, Mohali, Punjab India
**Corresponding author E-mail: pratikshasharma21192@gmail.com*

Abstract

Detection of bad smells refers to any indication in the program code of a execution that perhaps designate a issue, maintain the software and software evolution. Code Smell detection is a main challenging for software developers and their informal classification direct to the designing of various smell detection methods and software tools. It appraises 4 code smell detection tool in software like as a in Fusion, JDeodorant, PMD and Jspirit. In this research proposes a method for detection the bad code smells in software is called as code smell. Bad smell detection in software, OOSMs are used to identify the Source Code whereby Plug-in were implemented for code detection in which position of program initial code the bad smell appeared so that software refactoring can then acquire position. Classified the code smell, as a type of codes: long method, PIH, LPL, LC, SS and GOD class etc. Detection of the code smell and as a result applying the correct detection phases when require is significant to enhance the Quality of the code or program. The various tool has been proposed for detection of the code smell each one featured by particular properties. The main objective of this research work described our proposed method on using various tools for code smell detection. We find the major differences between them and dissimilar consequences we attained. The major drawback of current research work is that it focuses on one particular language which makes them restricted to one kind of programs only. These tools fail to detect the smelly code if any kind of change in environment is encountered. The base paper compares the most popular code smell detection tools on basis of various factors like accuracy, False Positive Rate etc. which gives a clear picture of functionality these tools possess. In this paper, a unique technique is designed to identify CSs. For this purpose, various object-oriented programming (OOPs)-based-metrics with their maintainability index are used. Further, code refactoring and optimization technique are applied to obtain low maintainability Index. Finally, the proposed scheme is evaluated to achieve satisfactory results. The results of the BFOA test defined that the lazy class caused framework defects in DLS, DR, and SE. However, the LPL caused no framework defects what so ever. The consequences of the connection rules test searched that the LCCS (Lazy Class Code Smell) caused structured defects in DE and DLS, which corresponded to the consequences of the BFOA test. In this research work, a proposed method is designed to verify the code smell. For this purpose, different OOPs based Software Metrics with their MI (Maintainability Index) are utilized. Further Code refactoring and optimization method id applied to attained the less maintainability index and evaluated to achieved satisfactory results.

Keywords: *Software Metrics; Code Smell Detection; BFOA Method; God Class and Lazy Class.*

1. Introduction

In this modern era, countless detection strategies are present that helps to evaluate the bad programming code, particularly focused on the variations. More than that, a good source of determines the awareness about the software as well as its properties. The selection of tool is a difficult concept and always an automatic tool is preferred as best tool. Code smell is one of them. Code smell is organized structural characteristic of software which trained software as powerful and harder. Refactoring software chosen by the code smell to change and ascend the quality of code. In simple words, smell is a kind of a disease and refactoring referred as heal to disease [1]. Code smell is term which preferred to classify the weakness of any kind of programming code. It depends on the OOD concepts (object oriented design). Fowler and Beck discovered these terms. The generated category of code smell is God class smell. There is further classes came under the God class smell. Further, Fowler and Beck introduced several classes that are 24 classes or more than it [2]. Figure 1 shows the three main categories of Code Smells in which further categorizations can be made to classify Code Smells.

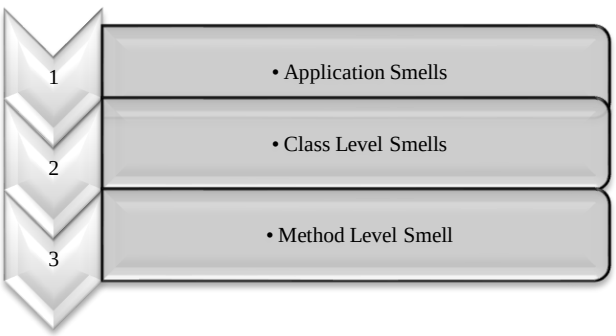

Fig. 1: Categories of Code Smell.

A. Application Smell: These kinds of smells are basically formed duplicate data which is same as the original one, they are more

complicated. The structural design enforced to become complex [3].

B. Class-Level Smells: It includes the large classes, refused bequests, lazy data, down casting, orphan variable which consists of extreme use of literals. In this, data clumped due to the movement of variable in the different parts of the code [4].

C. Method-Level Smells: The smells involve because of several parameters required to read together. The process becomes harder for call and functioning. The smells created due to the large return of data and long lines of coding [5], [6].

From all these categories of smells there are so many others smells as: Class of GOD, feature Vector, God function, long function, primal passion, long list of parameters. Other are OO (object oriented) abusers as switch-statements, temporary-fields, refused inheritance. Change preventers and dispensable, encapsulations, couplers are also categories of code smell.

The quantitative measurement of software parameters. In this research paper, it main concentrate on initial source code parameter's as considered to in the given table 1.

Table 1: Oosm (Object Orientation Software Metrics) [21]

Notation Name	Full Forms	Level Of The Software Metric
Dit	Depth Of In-Heritance Tree	Group/ Class
Par	No Of Parameters	Function Or Method
Nom	No. Of Methods	Class
Nof	No. Of Attributes	Class

Software design error is shortcoming happening from the wrong instructions in programming executions. It makes a mal method in the smell detection system, like as a slow executing, enhancing the source code, difficulty in fixing and wrong program output.

Bad Smell Detection Tool is designed as software plug-in and is competent of software detecting sub-sequent bad code smell as defined in table 2, by utilizing pre-defined value of software parameters which were explained from existing works as defined in given table 2:

Table 2: Defining the Approx. Threshold Assigned for Code Smell and Metric in Software [22]

Code Smell Names	Level Of Code Smell	Metrics And Threshold Values
Large Class	Group Or Class	Nom > 20.0 || Nof >9.0
Long Method	Function Or Method	Moc>50.0
Long Parameter List	Function Or Method	Par>5.0
Switch Statement	Function Or Method	Vg>10 .0
Parallel Inheritance	Group Or Class	Dit > 3.0 || Nsc > 4.0
Data Class	Group Or Class	Wmc > 50.0 ||Lcom >.8.0
Lazy Class	Group Or Class	(Nom<5.0 && Nof <5) || Dit < 2.0

Where parameter name is LCO is Lack of Cohesions of method, VG full form is Mc Cabe Cycloramic Complexity, MLOC full form Method Line of code , WMC full forms Weighted Methods per class.

1.1. Tools for code smell

The quality of code is necessity for the software maintenance. Poor and rich quality of code affects problems which increment the fault rate and make the run time more time consuming. The tools used for enhance the quality of code and other tools are mainly preferred for the reengineering and maintenance activities. The tools are described below:

a) Check Style: check style tool of version 5.4.1 used which is of eclipse plug in standalone. The tool is preferred to write code in the java language and there is no automatic refactoring and it is linked to the code.

b) DECOR: This tool automatically detects and allowed specifications of programming code. All codes are generated through the custom languages. The version of this tool is 1.0 2009 and its type is standalone particularly used in java as analyzed language. Refactoring is same as check style tool. Although no code links given.

c) Iplasma: This tool mostly preferred for the C++ coding and its version 6.1 2009 used There is no refactoring and link to code. It support to the model extraction and referred for the high level coding.

d) inFusion: infusion tool works like iPlasma there is only difference of versions and it support both C++ and Java coding. At a time, it detects around 20 designs which consist of: duplicate data, classes, Data Class and God Class. There are heavy coupled classes generated.

e) JDeodorant: this tool type is eclipse plug in which supported to the java analyzed coding with applicable for both refactoring and link to code. The main purpose is to detect and verify the feature envy, God Class, long techniques and to approachable to checking of variants. Its version is 4.0.4 2010.

f) PMD: This tool analyzed to support the Java language to identify its possible errors and it fixed the errors which are dead codes, empty try, switch statements and unusual variables.

g) Strench Blosom: this is the most approaches tool particularly referred for the visualized environment. Strench blosom gives more fastest and high level of smells. Its type is eclipse plug in also refer to java programming and it does not require refactoring while the process goes on [1].

Table 3: Smells Detect by Tools [23]

Code Smell	Tool Used
God Class	JDeodarant, InFusion and iPlasma
Large Class	Stench Blossom and PMD
Long Parameter List	iPlasma , PMD, DECOR
Refused Parent Bequest	InFusion, iPlasma, DECOR
Speculative Generality	iPlasma, DECORE and Benchmark

Each code smell table 3 experimented the detection tools that could detect the code smell.

1.2. Applications of code smell

The major applications of code smell are given below:

a) Mobile Applications: From few years ago, mobile applications were downloaded in enormous ways and there is coding in the mobile application, therefore code smell introduced in the terms of mobile applications. It contains limited number of resources.
For example, CPU, network and battery. These are event driven and reactive in nature [27]

b) Desktop Applications: Code smell occurred in the desktop applications. The desktop application required number of resources whereas Mobile applications were limited on resources.

c) Android Applications: From recent world of technology, near about 75% of mobile consists of android applications. More attention of code smell seen in the android applications. It contains huge updates and higher frequency as compare with other applications such as desktop applications. The J2SE API, Java FX, other APIs are not always seen in the coding of android. Graphical user interface of android is publicized by XML. [28]

d) Real-world applications: Code smell is introduced more than mobile, android and desktop applications. It can be used in the real world application in the automated security systems [7].

2. Related Work

Paiva, T., et al., [8] proposed work related to the introduction of code smell and in this paper the software matrices used for the detection of wrong codes was described. Further the code detection methods were shown. A code smell was like a disease in the coding of a programming language which effects on the maintenance and evolution of the code. The detection of the code smells was a difficult problem due to the multiple classes of code and its different tools. This paper was all about the detection tools related to code smells which are as below:

i) JDeodrant
ii) PMD
iii) JSpIRIT

All these tools were applicable on the software systems such as mobile media and health watch, particularly referred to check the accuracy. This experiment was done on each tool mainly to check the accuracy. Further secondary study done for the evolution of CS (code smell) basically focused on the target systems and evaluates code smell seen presence from the creation of class. Mansoor, U., et al., [9] researched on the multi objective code smell with the use of both good and bad designs in the programming language. Code smell involved set of detection rule which applied on the code. These rules were key systems that characterized smells using combination of metrics. A multi objective problem of code smell was detected. For the detection code designs and code smells were used to create the detection set of rules. After that, to optimize the performance of multi objective code smells, MOGP (multi objective genetic programming). These increased the detection of code smell and decrement the code designs. It was proposed with open source system and as a result the most of the basic kinds smell codes were detected through 87% of precision and approximately 90% of recall. Therefore, it was better than other detection tools. Liu, X., et al., [10] research work related to the detection tools which was referred to detect the smell codes automatically in software system. Detection tools, metrics and threshold were described in this paper. Code smells expelled the quality of code. Actually, code smell either a bugs or make system run exceptionally. The motive was just to halt the software system performance and maintenance. A successful tried method introduced which was DT (code smell detection tool). DT successfully detects more than 10 kinds of code smell and in the future time, it will become more usable and accurate, further become capable to find out various other code smells [29].

Palomba, F., et al., [11] proposed a detection tool which was lightweight and applicable to detect the android specific code smells. This paper make mention of the android code smells, detection tools and empirical study. In the evolution and software maintenance , various kinds of modification takes place which changed its processing and generate several kinds of faults or errors. Code smell was a kind of all these errors. Mostly in the android system, there was regular updation which causes errors and changes. To fix it, mobile applications support different detection tools and not applicable for the developers in a limited area. Although android was all above these tools. To overcome the errors and faults, an automatic tool namely ADOCTOR. ADOCTOR capable to detects more than 15 android code smells. The empirical study was also done. Haque, M. S., et al., [12] proposed work described the causes, impacts and detection techniques used in code smells. A survey of code smell was done in this research work. This paper consists of introduction of code smell, software engineering. CS was the anomalies produced in the implementation, structure and maintenance of software development life cycle. A survey of previously research work done on the 22 kinds of code smell. The taxonomy of code smells was explained which was collection of Blosters, Object Oriented Abusers, Change Preventions and Couplers. The impact and causes of CS was deeply explained in this paper. Di Nucci, D., et al., [13] in this world of technology, the software system growth incremented continuously and updation in the source code. Due to the updation, there were certain errors, bugs introduced automatically at the time of class generation. These errors anomalies called CS as code smell. The proposed research implies a machine learning method to detect the CS in the source code of any kind of software. Different kinds of dataset configuration replicated. Machine learning methods to detect the code smells was faced certain limitations which should be minimized with further research work [25-26].

Possible Outcomes and Problems Code smell detection and refactoring is a worm area in research work field. This approach is used to improve our code quality and behavior. Current approach uses Declarative Meta Programming. There are some plug-in which used to perform detection and refactoring on selected lines of code. Issue behind plug-in that these are not reliable in all conditions and all types of coding environment. Like if you are using java plug-in than it works with java code only. They works with small piece of code so it is not necessary that line of code or module have limited Lines of Code all time. So when we works with programming modules than it create issue to detection and refactoring bad smells from a large code. A graphical view is missing of clustering of various modules while performing detection and refactoring [14].

Smell detection tools are currently not useful to review which sections of code required to be enhanced. As a result to this there might be a possibility of higher cost in terms of time as well as accuracy. The proposed system can result in set of smelly codes along with the more optimized final code output which can be termed as clean code or well managed code. The approach used to extract the features tends to result in various modules of classes with one or more smelly codes. The optimization algorithm provides the system a high accuracy rate with lesser number or recalls [15].

Code smell detection is not having any formal definition so it is hard to explain the boundaries for smell detection tools. Each tool have their own limitations based on environment and design of the code. The proposed system could is limited with assumption of having a defined set of facts and rules which may lead to false positive rate which can be reduced with the use of more precise set of rules [16], [17].

2. Gaps In research

The major drawback of current research work is that it focuses on one particular language which makes them restricted to one kind of programs only. And these tools fail to detect the smelly code if any kind of change in environment is encountered. The base paper compares the most popular code smell detection tools on basis of various factors like accuracy, False Positive Rate etc. which gives a clear picture of functionality these tools possess. The novelty of our approach lies in motive to develop a tool which in not bounded by the language barriers and to compare it with the other tools available for detection of smelly code. Research gaps could be listed as:

- Most tools are available in form of plug-in which make them dependent on other software.
- Tools are constraint to a particular language [18], [19].
- As the software evolves detection of smelly code become more and more difficult.
- With the advancement of software the accuracy rate of smell detection tools reduces

3. Proposed methodology

Lots of researchers worked on detection of bad smells from the developed software systems. This process is used to improve the quality of software systems and easy to maintain. Various software metrics are used to evaluate the performance of detection tool. In all the existing systems the problems are still faced during the testing because the testing on different coding modules is hard to evaluate and generate the effective code modules. Figure 2 shows the methodology we follow in our work to solve this problem.

Here user can upload any project of java or .net language and the system will analyze the functional modules of the software to apply set of rules defined to detect the smelly codes and extract all the possible factors needed for further processes.

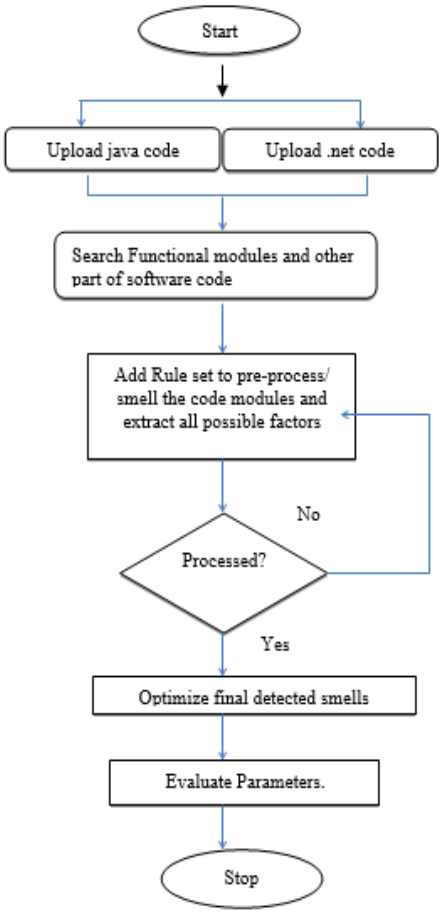

Fig. 2: Methodology Flow Chart.

Our work is headed in path to detect the smelly codes with larger accuracy and precision after considering the design of the software. The main objectives are-

"To improve code quality and lower risk analysis by identifying bad smells using refactoring techniques and metrics".

- To study and analyze various existing techniques for code smell detection with Object-Oriented Metrics.
- To design a code smell detection framework using OOPs.
- To evaluate the proposed scheme with respect to various performance metrics.
- To compare the proposed scheme with existing state of art technique.

As per growing need of software with low maintenance overhead and high evaluation rate it become necessary to detect the smelly code. The proposed method focus on development of a tool which can give better results for OOPs based code. The methodology shown in Fig 2 also solves the problem of testing advance software which is made in more than one coding language. The proposed system is using BFO Algorithm to get the optimized results for each type of code module. Use of BFO to analyze the all codes with same environment and set of rules makes the proposed method more accurate and precise.

4. Results and discussion

The result explain in this research aims to develop a code smell detection tool which can work with OOPs languages and generate results in higher accuracy, precision along with less false positive rate by use of facts and rules with Bacteria Foraging Optimization technique. The research also aims to develop such a tool which can test codes in more than one language and can be used as standalone software which makes it independent of any particular language.

Accuracy factor is very important while system is working with different environments. Here figure 3 shows average accuracy after various test cases and compared with proposed architecture. The performance of proposed architecture is shows better results than existing detection technique. Also the calculations show stable graph for all the performed test cases. This signifies that the results are more accurate when the software evolves and various versions of same software are launched with time.

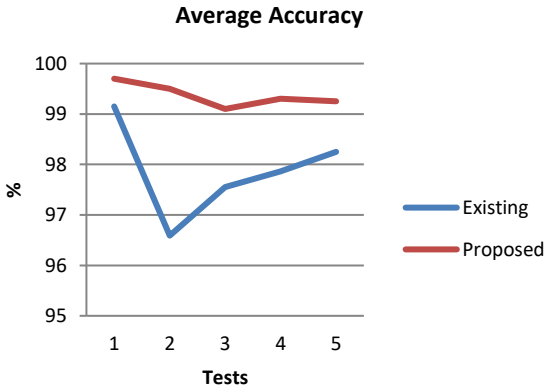

Fig. 3: Average Accuracy Factor.

The table 4 shows various smell detection support by the tools. The performance of proposed architecture is also better in this table. Here the proposed architecture perform a comparison between various different smells, the results shows that there are more than others as compared in this table. Proposed architecture shows the more detection facilities as compared to the other existing tools.

Table 4: Smell Detection Support by Code Smell Tools

Code Smells	Infusion	Jdeodorant	Pmd	Jspirit	Proposed
God Class	✓	✓	✓	✓	✓
Bad Switch Structure	✗	✗	✗	✗	✓
God Method	✓	✓	✓	✓	✓
Feature Envy	✗	✓	✗	✗	✗
Temporary Field	✗	✗	✗	✗	✓
Unused Catch Block	✗	✗	✗	✗	✓

Table 5: Languages Supports by Code Smell Tools

Tool	Version	Type	Languages	Refactoring	Export	Detection Technique
Infusion	1.8.6 2015	Standalone	Java, C, C++	✗	✓	Software Metrics
Jdeodorant	5.0.0 2015	Eclipse Plugin	Java	✓	✓	Refactoring
Pmd	5.3.0 2015	Eclipse Plugin	Java, C, C++	✗	✗	Software Metrics
Jspirit	1.0.0 2014	Eclipse Plugin	Java	✗	✗	Software Metrics

Proposed	1.0.0 2018	Standalone	Java, C++, .Net	✗	✓	Software Metrics, Fact/Rule, Optimization

Table 6: Average Recall and Precision Various Tools

Code Smells	Infusion		Jdeodorant		Jspirit		Pmd		Proposed	
	R	P	R	P	R	P	R	P	R	P
God Class	9.0	33.0	58.0	28.0	17.0	67.0	17.0	78.0	40.0	94.0
Bad Switch Structure	-	-	-	-	-	-	-	-	43.0	96.0
God Method	26.0	100.0	50.0	35.0	36.0	93.0	26.0	100.0	97.0	98.30
Feature Envy	-	-	48.0	13.0	-	-	-	-	-	-
Temporary Field	-	-	-	-	-	-	-	-	57.0	96.70
Unused Catch Block	-	-	-	-	-	-	-	-	68.0	95.80

The Table 5 shows comparison between various code smell detection tools. Here the two types of tools are compared. These are plug-in and standalone. Other factors are like detection technique and language support is also shown in the table. All these factors shows the proposed tool having better features as compared to the other existing tools.

The precision rate is used to check the system performance. Figure 4 shows the results calculated after performing various tests to note the precision rate for the uploaded projects. The system shows higher precision rate as compared in the precision and recall table. It shows the stable detection rate for this parameter in all the cases.

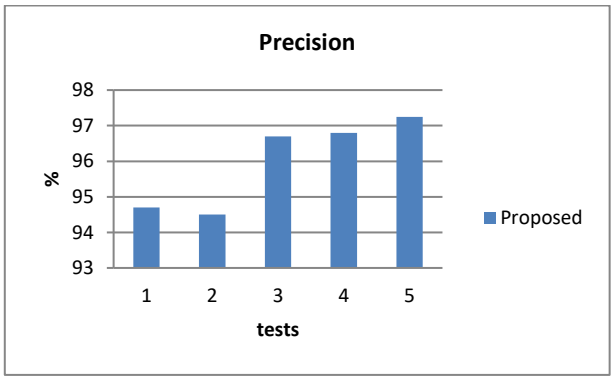

Fig. 4: Average Precision Rate.

The recall rate is used to check the system performance. Here the system performed various tests to note the recall rate for the uploaded projects. Figure 5 shows a graph for the recall rate of proposed system after each release of software version. The system shows better recall rate as compared in the precision and recall table. It shows the stable detection rate for this parameter in all the cases.

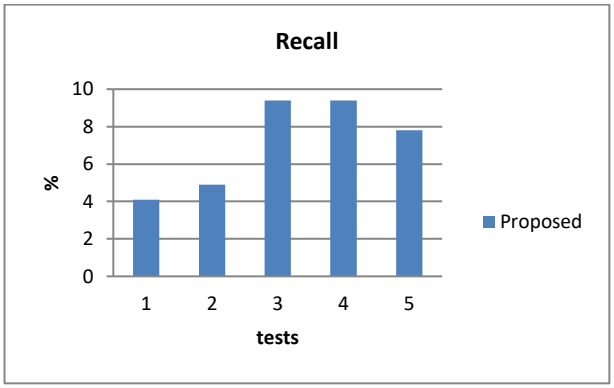

Fig. 5: Average Recall Rate.

The accuracy is overall calculations which provide detection of bad smells from the different coding modules and the string analysis in different modules along with their relationships with the other parts of the software systems. This parameter is directly connected with the detection of false sample and true samples from the overall analysis report. On the behalf of those detections the system predicts all the performance parameters and system accuracy. The lowest rate of false sample detection from the uploaded modules is enhancing the overall accuracy of the system. All the detected samples are tracked by the system to predict the working efficiency of the system.

5. Conclusion

In the conclusion, to define that to compare the detection tools are very problematic and in some possessions also using them are not very urgent and informal. Different code smells are detected in the source code using GUI (Graphical user Interface) application implemented. The evaluated a OOPs metrics defines the value of each software metrics in their respective code smells detection on the coding. The main objective of this research work was not to calculate the smell tools but to explained implemented method in using and draw the issues in the evaluation job.

i) An initial experiment study on the consequence of code smell on Software metric effort in an illegal industrial setting. It is utilized various linear regression analysis in which all of the program smells were the study in the similar program. CMs are the mainly ordinary bad designs associated with bad smells practices, which lead to deep study problems in maintaining the software.

ii) Software products that surround code smells could be not easy to maintain.

In this research work, it implements a software tool for detecting the wrong code smells, which used the threat concept. The verification concept, it implemented an automatic risk-based code smell detection tool. It utilized to recognize issues in a C# case study.

To calculate the performance metrics are Precision, Recall, and Accuracy and compared with the existing detection tool and algorithms.

Future work, it can implement a crossbreed method (GA+PSO) and designed based research to consumed less memory to detect the code smell. To performance could be optimized through some other methods and refactoring of BCSs for some other parameters of code. The proposed method can also improve the performance metrics and add some other bad smells in the code.

References

[1] Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology, 11(2), 5-1.

[2] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., and Shaw, M. (2010, September). Building empirical support for automated code smell detection. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (p. 8). ACM.

[3] Mika Mantyla. (2003). Bad smells in software-a taxonomy and an empirical study. Helsinki University of Technology (2003).

[4] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design aws. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. IEEE, 350–359.

[5] Matthew James Munro. 2005. Product metrics for automatic identi_cation of"bad smell" design problems in java source-code. In Software Metrics, 2005.11th IEEE International Symposium. IEEE, 15–15.

[6] Pietrzak, B., and Walter, B. (2006, June). Leveraging code smell detection with inter-smell relations. In International Conference on Extreme Programming and Agile Processes in Software Engineering (pp. 75-84). Springer, Berlin, Heidelberg.

[7] Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., and Jensen, C. (2016, May). Understanding code smells in android applications. In Proceedings of the International Workshop on Mobile Software Engineering and Systems (pp. 225-234). ACM.

[8] Paiva, T., Damasceno, A., Figueiredo, E., & Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. Journal of Software Engineering Research and Development, 5(1), 7.

[9] Mansoor, U., Kessentini, M., Maxim, B. R., and Deb, K. (2017). Multi-objective code-smells detection using good and bad design examples. Software Quality Journal, 25(2), 529-552.

[10] Liu, X., and Zhang, C. (2017). DT: a detection tool to automatically detect code smell in software project. Advances in Computer Science Research, 71.

[11] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2017, February). Lightweight detection of Android-specific code smells: The aDoctor project. In Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on (pp. 487-491). IEEE.

[12] Haque, M. S., Carver, J., and Atkison, T. (2018, March). Causes, impacts, and detection approaches of code smell: a survey. In Proceedings of the ACMSE 2018 Conference (p. 25). ACM.

[13] Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., and De Lucia, A. (2018, February). Detecting code smells using machine learning techniques: are we there yet?. In 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and NEgative Results (RENE) Track. Institute of Electrical and Electronics Engineers (IEEE).

[14] Ito, Y., Hazeyama, A., Morimoto, Y., Kaminaga, H., Nakamura, S., & Miyadera, Y. (2015). A Method for Detecting Bad Smells and ITS Application to Software Engineering Education. *International Journal of Software Innovation (IJSI)*, *3*(2), 13-23.

[15] Danphitsanuphan, P., & Suwantada, T. (2012, May). Code smell detecting tool and code smell-structure bug relationship. In *Engineering and Technology (S-CET), 2012 Spring*

[16] Sreenu, K., & Rao, D. J. Performance-Detection of Bad Smells in Code for Refactoring Methods.

[17] Jyothi, V. E., Srikanth, K., & Rao, K. N. (2012). Effective Implementation of Agile Practices-Object Oriented Metrics tool to Improve Software Quality. *International Journal of Software Engineering & Applications*, *3*(4), 13.

[18] Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on* (pp. 30-38). IEEE.

[19] Meananeatra, P., Rongviriyapanish, S., & Apiwattanapong, T. (2011, May). Using software metrics to select refactoring for long method bad smell. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on* (pp. 492-495). IEEE.

[20] Henderson Sellers, B., Object-Oriented Metrics: Measures of Complexity. Prentice Hall, 1996.

[21] Williams, Laurie, Dright Ho, and Sarah Heckman. (2005). Software Metrics in Eclipse [Online]. Avaliable: http://agile.csc.ncsu.edu/SEMaterials/tutorials/metrics/.

[22] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011, March). An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* (pp. 450-457). IEEE.

[23] Paiva, T., Damasceno, A., Figueiredo, E., & Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, *5*(1), 7.

[24] Abdelmoez, W., Kosba, E., & Iesa, A. F, "Risk-based code smells detection tool," In the International Conference on Computing Technology and Information Management (ICCTIM) (p. 148). Society of Digital Information and Wireless Communication, January 2012.

[25] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino & Mika V. Mantyla, "Code Smell Detection: Towards a Machine Learning-Based Approach," IEEE International Confrence, September 2013.

[26] Karaboga, D., & Basturk, B., "On the performance of artificial bee colony (ABC) algorithm," Applied soft computing, 8(1), 687-697, 2008.

[27] Ayman Madi, O.K. Zein and Seifedine Kadry, "On the Improvement of Cyclomatic Complexity Metric," International Journal of Software Engineering and Its Applications Vol.7, No.2, March, 2013

[28] Ki H. Kang and Jina Kang, "Do External Knowledge Sourcing Modes Matter for Service Innovation? Empirical Evidence from South Korean Service Firms," J prod Innov M nag 2014.

[29] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo and Cláudio Sant Anna, "On the evaluation of code smells and detection tools," Journal of Software Engineering Research and Development, Springer 2017.