View metadata, citation and similar papers at core.ac.uk
brought to you by CORE

ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20210205095410
Preliminary communication

An Analytical Study of Code Smells

Lida Bamizadeh*, Binod Kumar, Ajay Kumar, Shailaja Shirwaikar

Abstract: Software development process involves developing, building and enhancing high-quality software for specific tasks and as a consequence generates considerable amount of data. This data can be managed in a systematic manner creating knowledge repositories that can be used to competitive advantage. Lesson's learned as part of the development process can also be part of the knowledge bank and can be used to advantage in subsequent projects by developers and software practitioners. Code smells are a group of symptoms which reveal that code is not good enough and requires some actions to have a cleansed code. Software metrics help to detect code smells while refactoring methods are used for removing them. Furthermore, various tools are applicable for detecting of code smells. A Code smell repository organizes all the available knowledge in the literature about code smells and related concepts. An analytical study of code smells is presented in this paper which extracts useful, actionable and indicative knowledge.

Keywords: code smells; data mining; knowledge repository; refactoring methods; software metrics

1 INTRODUCTION

Today's software development process produces large amount of data. Lesson's learned and best practices in software development process are spread out over literature in various forms such as Code smells, design patterns, idioms etc. Organizing this knowledge into a knowledge repository, extracting insights from this data and making them available to code developers and software practitioners, can assist the software development process. Code smell is a general mechanism to distinguish structural design issues in software projects [1, 2]. Code smell term was formulated by Kent Beck when helping fowler for his refactoring book and has since become an important word in software maintenance vocabulary. Existence of code smell would not interrupt the functionality of system but it would enhance the risk of decay and reduce the software quality of system over time [3, 4]. Many software metrics are available in literature for detection of code smells [5, 6]. Moreover, there are several tools that developers can apply for automatic or semi-automatic detection of code smells in their code. Applying appropriate refactoring actions is the right way to deal with code smells. Refactoring actions can remove Code smells and optimize the quality of software design during maintenance process [7-9].

This paper presents an analytical study of code smells and its related concepts. The significance of this study is to extract some insightful information from inter relation between code smells, software metrics, refactoring actions and detection tools. This paper is organized as follows: background and related work is described in section 2. The design of code smell repository is presented in section 3. Section 4 presents application of different analytical techniques to code smell related information tables and extracting of indicative information that can further enhance the usefulness of a code smell repository. This is followed by conclusion in section 5.

2 BACKGROUND AND RELATED WORKS

In 1999, Beck and Fowler [9] found out that code smells are some indications in source code which don't prevent of its functionality but may reveals lots of problems in future. They presented 22 code smells and some refactoring actions that can be used to develop the design.

Mantyla [10] classified 22 code smells in seven categorized because of their similar features.

Mens and Tourwé [11] presented their survey on refactoring. It includes all aspects of refactoring process such as general ideas, refactoring actions, different formalism and methods, attentions and how refactoring suits the software development process. Walter and Pietrzak [12] pointed out that certain code smells such as divergent change get added as part of the maintenance phase. They proposed that multiple pieces of code need to be analyzed to detect the change. Marinescu [13] promoted the formalization of definition of code smells. He developed the detection to a broader range of code smells and a number of design principle violations. Olbrich et al. [14] demonstrated that in the existence of bad smells, performance of open source projects is degraded. They examined this bad feature for three software projects. God Class and Brain Class were selected by them for their experimental study. They observed that without normalization of size, both smells are harmful for code. In contrast, with normalization of size, outcomes are reversed. Therefore, they evolved that the size of both code smells are major factor for measuring the harmfulness of these smells. An investigation about God Class and Data Class presented by Ferme et al. [15] proves that bad smells are destructive for source code. Different filters were suggested by them to decrease or refine detection rules for code smells. Mahmood et al. [16] investigated several refactoring tools and established their purpose of usage. Also, they examined automation of tools for different code smells. Ganea et al. [17] described that code smells make considerable disadvantages in source code. They presented a tool named "InCode" that is an Eclipse plug-in. This tool is designed for Java programs and has capability of increasing the quality of source code and decreasing the code smells. Yamashita and Moonen [18] presented an empirical study about inter relation of code smells and their effect on occurrence of maintainability issues. They found out that certain inter-smell relations were connected with issues in the maintenance process and some inter-smell relations indicated

through couple artifacts. Yamashita et al. [19] had a survey for detecting a broader range of inter-smell relations. They observed that for various domains some of the code smells have same inter relation and should pay attention to them. Therefore, these inter relations can help practitioners for improving the quality of software systems.

3 BUILDING CODE SMELL REPOSITORY

An extensive literature survey was carried out to gather all the information about Code smells and the related concepts. An initial list of 22 code smells was proposed by Kent Beck and Martin Fowler [20] which has since grown with contributions from several researchers and practitioners into almost 65 code smells. With the increase in number of code smells, Mantyla [10] proposed a classification of code smells into six categories.

Software Metrics use measurable software attributes as indicators of latent software quality attributes [21-23]. Detrition of quality created by presence of code smells can be quickly detected by using one or more related software metrics [24-26]. The literature survey identified that around 49 software metrics are applicative in code smell detection. Software metrics are categorized in many ways and one such classification separates class level metrics from method level metrics.

Tool support is essential, as several code smells can go undetected while programming [27]. Tools are available for automatic or semi-automatic detection of code smells. The detection methods applied by tools are generally established on the calculation of a specific set of composite metrics using the threshold values for these metrics [28]. Numerous tools are accessible but 9 detection tools are popular to use by developers.

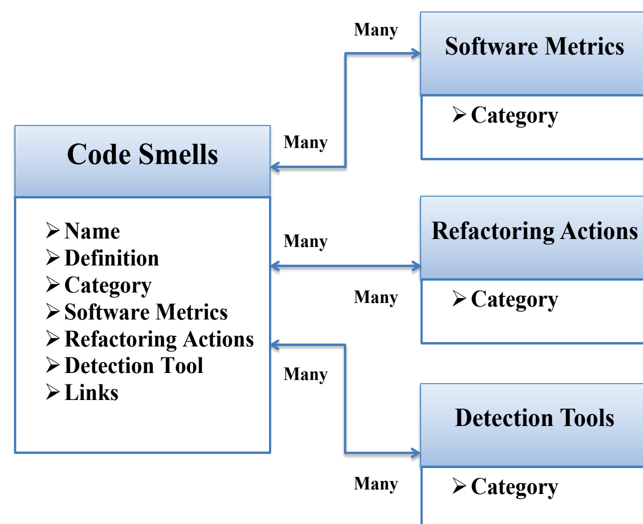

Figure 1 Code smell repository schema

Maintainability is the most important step in software development process [29]. Maintainability can be improved by use of refactoring methods. The term 'Refactoring' was presented by Opdyke [30] in his PhD thesis. Later, Fowler [9] identified that refactoring is a disciplined method for restructuring internal structures of existing source code without changing its external structures [27, 31]. There are around 87 refactoring actions that could be picked from literature, which are classified into six groups.

The schema of Code smell repository presents that code smell operates as the main object of repository which is linked to software metrics, refactoring actions and detection tools. Code smell relations with its corresponding related concepts are many to many. Each of the related concepts has its own details such as name, definition, category and etc. Fig. 1 displays Code smell repository schema. Further 'links' attribute can be used to navigate to different sources of detailed information about code smells.

The code smell repository thus constructed is available at https://serene-tundra-28026.herokuapp.com

3.1 Methodology of Building Code Smell Repository

- Selection of different kind of papers about code smells and related concepts from different journals and internet sources
- Data extraction about code smells and related concepts from the literature
- Organization of code smell knowledge

1) Designing a code smell repository template
2) Designing a code smell repository schema
3) Generating tables between code smells and each related concept for presentation of their relationship
4) Designing a web code smell application using Angular, Material Design, Node JS, Express JS and MongoDB
5) Implementing the code smell repository

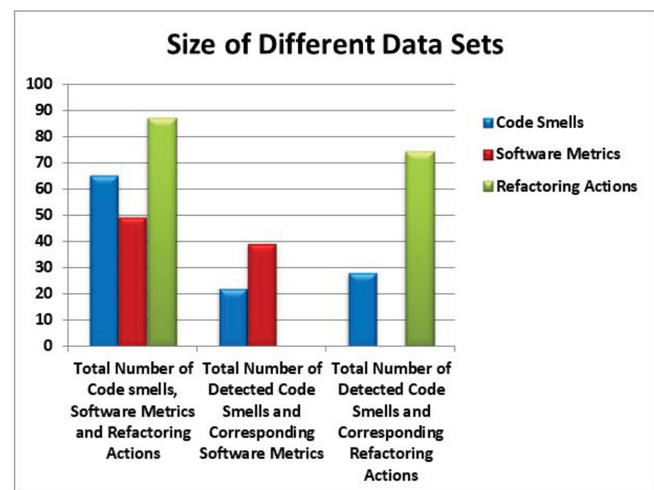

Figure 2 Size of Different Data Sets

4 ANALYTICAL STUDY OF CODE SMELLS AND ITS RELATED CONCEPTS

Data collected in the Code smell repository can be analyzed to gain useful insights into the world of code smells. Analytical study of code smells focuses on inner relation between code smells and its related concepts. Though only 22 code smells are detected by one or more out of 39 software metrics, this table capturing the link among code smells and

software metrics may be subjected to several analytical techniques to extract useful insights. Also, 28 code smells can be removed by 74 refactoring actions. Fig. 2 shows it.

4.1 Identifying Most Significant Set of Software Metrics

Many code smells can be detected by one or more metrics. As a sample, a formula for God Class detection is:

$$WMC \geq VERY_HIGH \wedge ATFD > FEW \wedge TCC < \frac{1}{3} \quad (1)$$

where *FEW* is 5 and *VERY_HIGH* is 47.

In contrast, *LOC* alone can detect Large Class [32]. One or more metrics may detect one or more code smells. Therefore, they have a many to many relationship.

Notably, most obvious metric for code smell detection is size metric. *LOC* (number of lines of code) acts as the leader of metrics in detection of code smells as it is used in detection of as many as 8 code smells.

Table 1 Significant Software Metrics and Code Smells Detected by Them

No	Metrics	Detected code smells	Total
1	LOC	God class, Brain class, Long method, Brain method, Duplicated code, Primitive obsession, Large class, Lazy class	8
2	WMC	God class, Brain class, Duplicated code, Large class, Lazy class, Data class, Refused bequest	7
3	VG	God class, Long method, Switch statements, Brain method, Conditional complexity, Duplicated code, Lazy class	7
4	NOM	Message chain, Middle man, Lazy class, Refused bequest, Large class	5
5	LCOM	Feature envy, God class, Large class, Data class	4
6	CBO	Feature envy, God class, Large class, Lazy class	4
7	DIT	Parallel inheritance hierarchy, Duplicated code, Large class, Lazy class	4

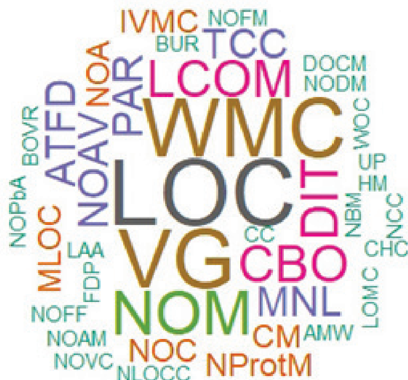

Figure 3 Significance of different software metrics in Code smell detection

Both WMC (Weighted Method Count per Class) and VG (McCabe Cyclomatic Complexity per module) are related to Cyclomatic complexity and are used in detection of 7 code smells each. Clearly, 7 metrics (LOC, WMC, VG, NOM, LCOM, CBO and DIT) present the most significant set of software metrics used in detection of a large number of code smells. Tab. 1 shows the significance of detected smells and software metrics. The wordcloud in Fig. 3 gives a visual presentation of Significance of different software metrics in detection of Code smells. Appendix A shows the used metrics abbreviation.

4.2 Identifying Representative Metric for Each Code Smell Category

Code smells are organized into 6 categories [33-36]. All code smells are not categorized in literature. A decision tree classification method is applied on this table where Code smell category acts as a class label. Classification result shows that each code smell category can have one or more representative metric. Result is showed in Tab. 2.

Table 2 Code Smell Category and Its Representative Metric

No	Code Smell Category	Representative Metric
1	Bloaters	LOC
2	Object Orientation Abusers	MNL or WMC
3	Change Preventers	CM
4	Dispensables	DIT
5	Couplers	NOM

This information can be used to advantage in predicting categories of code smell not yet categorized also in designing detection metrics for code smells from a particular category.

4.3 Identifying Association between Software Metrics

Code smell table with related metrics can be also subjected to identifying association between different metrics. The results of apriori algorithm with minimum support: 0.15 (3 instances) and minimum confidence: 0.9 generated 11 one itemsets, 13 two itemsets and 4 three itemsets and corresponding association rules. The three itemsets as given in Tab. 3 bring out most frequently occurring groups of related metrics.

Table 3 Frequency Occurring Metric Groups

No	Most frequently occurring groups
1	Lines of Code, Weighted Method Count per Class, Depth of Inheritance Tree
2	Lines of Code, Weighted Method Count per Class, Coupling Between Objects
3	Lines of Code, Weighted Method Count per Class, Tight Class Cohesion
4	Coupling Between Objects, Lack of Cohesion in Methods, Access To Foreign Data

4.4 Identifying Clustering between Code Smells and Software Metrics

The objective of clustering analysis is to recognize patterns in data and make groups based on those patterns. Thus, if two observations have similar features that mean they have the identical pattern. Consequently, they are included in the same group. Clustering is capable to shows what characteristics frequently appear together. Fig. 4 is result of clustering on detected code smells and corresponding used metrics. After cutting dendrogram at *k* =

4, one can find out the sets of code smells that appear together. These code smells have similarity as to the detection metrics used by them. Thus there is more chance of them occurring together.

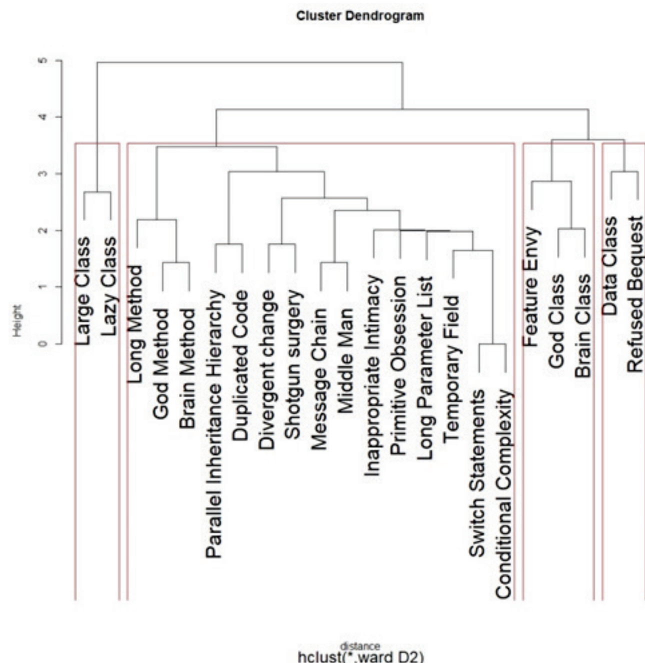

Figure 4 Dendrogram of Code Smells with Clusters

Table 4 Important Refactoring Methods

No	Metrics	Detected code smells	Total
1	Move Method	Switch Statements, Data Class, Feature Envy, Message Chains, Middle Man, Alternative Class with Different Interfaces, Shotgun Surgery, Parallel Inheritance Hierarchies, Inappropriate Intimacy	9
2	Extract Method	Long Method, Switch Statements, Comments, Data Class, Feature Envy, Message Chains, Duplicate Code	7
3	Extract Class	Inappropriate Intimacy, Duplicate Code, Temporary Field, Large Class, Data Clumps, Divergent Change, Primitive Obsession	7
4	Move Field	Feature Envy, Middle Man, Shotgun Surgery, Parallel Inheritance Hierarchies, Inappropriate Intimacy	5
5	Introduce Parameter Object	Long Method, Data Clumps, Long Parameter List, Primitive Obsession	4
6	Preserve Whole Object	Long Method, Long Parameter List, Primitive Obsession, Data Clumps	4
7	Extract Superclass	Alternative Classes with Different Interfaces, Duplicate Code, Refused Bequest, Divergent Change	4
8	Inline Class	Shotgun Surgery, Speculative Generality, Lazy Class, Dead Code	4

4.5 Inter Relation between Code Smells and Refactoring Actions

Refactoring is an important task of maintenance phase that aims at improving latent software quality attributes like understandability, flexibility, and reusability [40]. One or more refactoring actions have been suggested for eliminating of one or more code smells, so the type of correlation between them is many to many. 'Move method' is an important refactoring action that addresses the problem of as many as 9 code smells. Tab. 4 shows the most important refactoring actions that can be used in getting rid of a large set of code smells. Also, Fig. 5 represents a wordcloud of Significance of different refactoring methods in detection of Code smells.

Figure 5 Significance of different refactoring actions in Code smell detection

4.6 Identifying Association between Refactoring Actions

Code smell table with related refactoring actions can be also subjected to identifying association between different refactoring actions. The results of apriori algorithm with minimum support: 0.1 and minimum confidence: 0.9 generated 15 one item sets and 4 two item sets. The best four association rules generated with confidence 1 are as given below in Tab. 5.

Table 5 Association between Refactoring Methods

No	Most frequently occurring groups
1	Movie Field ⇒ Move Method
2	Preserve Whole Object ⇒ Introduce Parameter Object
3	Introduce Parameter Object ⇒ Preserve Whole Object
4	Collapse Hierarchy ⇒ Inline Class

This association indicates the pairs of refactoring actions that is closely linked.

5 CONCLUSION

Organization of knowledge about code smells and related concepts spread out in literature into code smell repository gives rise to tables holding useful information. Applying analytical techniques to these tables can help in improving this knowledge bank further.

Analytical study of code smells and its related concepts gives insightful knowledge about code smells to improve the software development process. Results of this paper are as follows:

- 22 code smells are detected by one or more out of 39 software metrics and 28 code smells can be removed by 74 refactoring actions

- Presenting the table of top 7 software metrics to detect the maximum number of code smells and top 8 refactoring actions to eliminate the maximum number of code smells
- Preparing a wordcloud of Significance of different software metrics and refactoring actions with respect to Code smells
- Presenting of one or more representative software metric for each code smell category by applying a decision tree classification method
- Presenting the most Frequently Occurring Groups of software metrics based on code smells and metrics relationships by applying association apriori algorithm
- Presenting the most Frequently Occurring Groups of Refactoring actions based on code smells and refactoring methods relationships by applying association apriori algorithm
- Applying the hierarchical clustering based on code smells and software metrics relationships to find the similarity of code smells by presenting of a dendrogram. This presents a new way of categorizing code smells

The code smell repository and the extracted insights can assist the developers and software practitioners.

Notice

This paper was presented at IC2ST-2021 – International Conference on Convergence of Smart Technologies. This conference was organized in Pune, India by Aspire Research Foundation, January 9-10, 2021. The paper will not be published anywhere else.

6 REFERENCES

[1] Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., & Oizumi, W. (2015, November). JSpIRIT: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)* (pp. 1-6). IEEE. https://doi.org/10.1109/SCCC.2015.7416572

[2] Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., & Jensen, C. (2016, May). Understanding code smells in Android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (pp. 225-236). IEEE. https://doi.org/10.1145/2897073.2897094

[3] Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018, March). Detecting code smells using machine learning techniques: are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 612-621). IEEE. https://doi.org/10.1109/SANER.2018.8330266

[4] Firdaus, M. F., Priyambadha, B., & Pradana, F. (2018, November). Refused Bequest Code Smells Detection on Software Design. In *2018 International Conference on Sustainable Information Engineering and Technology (SIET)* (pp. 288-291). IEEE. https://doi.org/10.1109/SIET.2018.8693156

[5] Eisty, N. U., Thiruvathukal, G. K., & Carver, J. C. (2018, October). A survey of software metric use in research software development. In *2018 IEEE 14th International Conference on e-Science (e-Science)* (pp. 212-222). IEEE. https://doi.org/10.1109/eScience.2018.00036

[6] Do Vale, G. A., & Figueiredo, E. M. L. (2015, September). A method to derive metric thresholds for software product lines. In *2015 29th Brazilian Symposium on Software Engineering* (pp. 110-119). IEEE. https://doi.org/10.1109/SBES.2015.9

[7] Khurana, G. & Jindal, S. (2013). A model to compare the degree of refactoring opportunities of three projects using a machine algorithm. *Advanced Computing, 4*(3), 17. https://doi.org/10.5121/acij.2013.4302

[8] Dhaka, G. & Singh, P. (2016, December). An empirical investigation into code smell elimination sequences for energy efficient software. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)* (pp. 349-352). IEEE. https://doi.org/10.1109/APSEC.2016.057

[9] Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[10] Mantyla, M. (2003). *Bad smells in software-a taxonomy and an empirical study*. Helsinki University of Technology.

[11] Mens, T. & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30*(2), 126-139. https://doi.org/10.1109/TSE.2004.1265817

[12] Walter, B. & Pietrzak, B. (2005, June). Multi-criteria detection of bad smells in code with UTA method. In *International Conference on Extreme Programming and Agile Processes in Software Engineering* (pp. 154-161). Springer, Berlin, Heidelberg. https://doi.org/10.1007/11499053_18

[13] Marinescu, R. (2005, September). Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 701-704). IEEE. https://doi.org/10.1109/ICSM.2005.63

[14] Olbrich, S. M., Cruzes, D. S., & Sjøberg, D. I. (2010, September). Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance* (pp. 1-10). IEEE. https://doi.org/10.1109/ICSM.2010.5609564

[15] Ferme, V., Marino, A., & Fontana, F. A. (2013). Is it a real code smell to be removed or not? In *International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference*.

[16] Mahmood, J. & Reddy, Y. R. (2014, February). Automated refactorings in Java using IntelliJ IDEA to extract and propogate constants. In *2014 IEEE International Advance Computing Conference (IACC)* (pp. 1406-1414). IEEE. https://doi.org/10.1109/IAdCC.2014.6779532

[17] Ganea, G., Verebi, I., & Marinescu, R. (2017). Continuous quality assessment with inCode. *Science of Computer Programming, 134*, 19-36. https://doi.org/10.1016/j.scico.2015.02.007

[18] Yamashita, A. & Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection?– An empirical study. *Information and Software Technology, 55*(12), 2223-2242. https://doi.org/10.1016/j.infsof.2013.08.002

[19] Yamashita, A., Zanoni, M., Fontana, F. A., & Walter, B. (2015, September). Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 121-130). IEEE. https://doi.org/10.1109/ICSM.2015.7332458

[20] Sharma, T. & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software, 138*, 158-173. https://doi.org/10.1016/j.jss.2017.12.034

[21] Sharma, M. & Singh, G. (2011). Analysis of Static and Dynamic Metrics for Productivity and Time Complexity. *International Journal of Computer Applications, 30*(1), 7-13. https://doi.org/10.5120/18036-6883

[22] Núñez-Varela, A., Perez-Gonzalez, H. G., Cuevas-Tello, J. C., & Soubervielle-Montalvo, C. (2013). A methodology for obtaining universal software code metrics. *Procedia Technology, 7*, 336-343.

https://doi.org/10.1016/j.protcy.2013.04.042
[23] Tahvildari, L. & Kontogiannis, K. (2003, March). A metric-based approach to enhance design quality through meta-pattern transformations. In *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings*. (pp. 183-192). IEEE.
[24] Sarıman, G. & Kucuksille, E. U. (2016). A novel approach to determine software security level using bayes classifier via static code metrics. *Elektronika ir Elektrotechnika, 22*(2), 73-80. https://doi.org/10.5755/j01.eie.22.2.12177
[25] Srinivasan, K. P. & Devi, T. (2014). A complete and comprehensive metrics suite for object-oriented design quality assessment. *International Journal of Software Engineering and Its Applications, 8*(2), 173-188.
[26] Kaur, S. & Maini, R. (2016). Analysis of various software metrics used to detect bad smells. *Int J Eng Sci (IJES), 5*(6), 14-20.
[27] Hamid, A., Ilyas, M., Hummayun, M., & Nawaz, A. (2013). A comparative study on code smell detection tools. *International Journal of Advanced Science and Technology, 60*, 25-32. https://doi.org/10.14257/ijast.2013.60.03
[28] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011, March). An experience report on using code smells detection tools. In *2011 IEEE fourth international conference on software testing, verification and validation workshops* (pp. 450-457). IEEE. https://doi.org/10.1109/ICSTW.2011.12
[29] Sharma, T. & Janakiram, D. (2010). Inferring design patterns using the ReP graph. *J. Object Technol., 9*(5), 95-110. https://doi.org/10.5381/jot.2010.9.5.a5
[30] Opdyke, W. F. (1992). Refactoring object-oriented frameworks.
[31] Szőke, G., Antal, G., Nagy, C., Ferenc, R., & Gyimóthy, T. (2017). Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software, 129*, 107-126. https://doi.org/10.1016/j.jss.2016.08.071
[32] Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology, 11*(2), 5-1. https://doi.org/10.5381/jot.2012.11.2.a5
[33] Roperia, N. (2009). *JSmell: A Bad Smell detection tool for Java systems*. California State University, Long Beach.
[34] Ahmed, I., Ghorashi, S., & Jensen, C. (2014). An exploration of code quality in FOSS projects. In *IFIP International Conference on Open Source Systems*, pp. 181-190. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-55128-4_26
[35] Saranya, G. (2017). *Code smell detection and prioritization of refactoring operations to enhance software maintainability*. Faculty of Science and Humanities Anna University.
[36] Refactoring ·guru· (n.d.c.). Code Smells. https://refactoring.guru/refactoring/smells/ (Accessed on 16 October 2020)

Authors' contacts:

Lida Bamizadeh, research scholar
(Corresponding author)
Department of Computer Science, Savitribai Phule Pune University,
Ganeshkhind Rd, Ganeshkhind, Pune, Maharashtra 411007, India
9503039485, lida_bamizadeh@yahoo.com

Binod Kumar
JSPM's Rajarshi Shahu College of Engineering (MCA Dept.),
Tathawade, Pimpri-Chinchwad, Maharashtra 411033, India
9665548971, binod.istar.1970@gmail.com

Ajay Kumar
JSPM Jayawant, Technical Campus,
Tathawade, Pimpri-Chinchwad, Maharashtra 411033, India
7972095030, ajay19_61@rediffmail.com

Shailaja Shirwaikar
Department of Computer Science, Savitribai Phule Pune University,
Ganeshkhind Rd, Ganeshkhind, Pune, Maharashtra 411007, India
7066046154, scshirwaikar@gmail.com

APPENDIX A

No	Metrics	Abbreviations
1	Number of Lines of Code	LOC
2	McCabe Cyclomatic Complexity per Module	VG
3	Weighted Method Count per Class	WMC
4	Depth of Inheritance Tree	DIT
5	Class Coupling	CC
6	Coupling Between Objects	CBO
7	Lack of Cohesion in Methods	LCOM
8	Number of Parameters per Method	PAR
9	Tight Class Cohesion	TCC
10	Number of Methods	NOM
11	Number of Attributes	NOA
12	Method Lines of Code	MLOC
13	Number of Children	NOC
14	Access To Foreign Data	ATFD
15	Locality of Attribute Accesses	LAA
16	Foreign Data Provider	FDP
17	Number of Accessor Methods	NOAM
18	Halstead Metric	HM
19	Number of Brain Methods	NBM
20	Maximum Nesting Level	MNL
21	Number of Accessed Variables	NOAV
22	Unused Parameters	UP
23	Weight of a Class	WOC
24	Number of Public Attributes	NOPbA
25	Number of Protected Members	NProtM
26	Base-class Usage Ratio	BUR
27	Base-class Overriding Ratio	BOvR
28	Average Method Weight	AMW
29	Number of Lines of Code in a Class	NLOCC
30	Instance Variable per Method in a Class	IVMC
31	Number of Delegate Method	NODM
32	Number of Foreign Fields	NOFF
33	Number of Foreign Methods	NOFM
34	Length of Methods Call Chain	LOMC
35	Number of Variables per Class	NOVC
36	Changing Classes	CHC
37	Changing Methods	CM
38	Dependency-Oriented Complexity Metric	DOCM
39	Number of Concerns per Component	NCC