IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-5, May 2019 E-ISSN: 2347-2693

Generating Code-Smell Prediction Rules Using Decision Tree Algorithm and Software Metrics

Mohammad Y. Mhawish[1*], **Manjari Gupta**[2]

[1,2] DST-CIMS, Banaras Hindu University, Varanasi, India

Corresponding Author: bniyaseen@gmail.com

DOI: https://doi.org/10.26438/ijcse/v7i5.4148 | Available online at: www.ijcseonline.org

Accepted: 12/May/2019, Published: 31/May/2019

Abstract— Code smells identified by Fowler [1] is as symptoms of possible code or design problems. Code smells have adverse affecting the quality of the software system by making software challenging to understand and consequently increasing the efforts to maintenance and evolution. The detection of code smells is the way to improve software quality by recovering code smells and perform the refactoring processes. In this paper, we propose a code- smells detection approach based on a decision tree algorithm and software metrics. The datasets we used to train the models are built by reforming the datasets used by Arcelli Fontana et al. work [2]. We use two feature selection methods based on a genetic algorithm to select the most essential features in each dataset. Moreover, we use the grid search algorithm to tuning the decision tree hyperparameters. We extract a set of detection conditions using decision tree models, that are considered as prediction rules to detect each code smell in our binary-class datasets.

Keywords— code smells, code smells detection, Feature selection, decision tree, prediction rules.

I. INTRODUCTION

In software development, there are functional and non-functional quality requirements that the developers have to follow to assure the software quality[3]. Developers, for many reasons, focus on pure functional requirements and neglect the non-functional requirements, e.g., modifiability, maintainability, evolution and extensibility, testability, understandability, and reusability[4]. The lack of these requirements in software led to the low of software quality and increased complexity and efforts for the maintenance and evolution of the software due to the weakness of the design rules. Thus, one should follow the design principles, e.g., encapsulation, modularity, and data abstraction.

Software maintenance is the modification applied to the software in the propose of adapting or modifying the environment of the software. The increase of maintenance effort is typically caused by the weakening of software design and lousy implementation style[5]. The understanding and the comprehension of the software by increasing the understandability of source code and design is fundamental in the activity of the maintenance and the evolution process of software systems. One field of software engineering that supports the maintenance and evolution of software activity is reverse engineering. Chikofsky[6] defines reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction."

Code-smells is defined as a term to describe the characteristics in software that may indicate a design problem. Code smells term was introduced by Fowler and Beck [1]. They presented informal definitions of 22 code-smells in the software system.

Several studies have investigated the disadvantage of code-smells in software system[7][8][9][10][11]. The studies shown that the existing code smells in the software implies to increase the effort of maintenance and evolution activities of software. Van Emden et al.[12] proposed the adverse impact of code smells on the software, and they presented a methodology to reduce this impact. Moreover, in [13] the authors found that the existence of code-smells in the software was affected directly to the software quality. Olbrich et al.[14] show that one of the reasons for performance degeneration of software is the presence of code smells in the source code. Tufano et al.[15] explored the reasons for the existence of the code smells in source code and they evaluated many projects to get back on the impact of code smells on the quality of software. Yamashita and Moonen[9] evaluated the adverse impact of code smells in the software maintenance, and they conducted the validation of their work by expert observations of the software systems that were placed for maintenance.

All these studies found the relationships between the code-smells and the increase of the risk of faults and failures of the software system and the negative impact of code-smells on the quality of the software and they recommended for applying the refactoring in order to clean software system from code-smells. The Recovery of code-smells in the source code is the way to improve the maintenance and evolution activities of software. The term of code-smell detection was proposed by Fowler [1]. They identified 22 code-smells, and they associated these code-smells with the refactoring transformations that have to be applied to improve the structure of software. Many approaches have been proposed for detection code-smells; they used different techniques and attempted with different code-smells. Several approaches [16][17][18] used the manual detection methods based on Fowler's [1] identification of code-smells. On the other hand several approaches used the automatic detection techniques for example Metrics based code-smell detection [19] [20], Heuristic-based code-smell detection: [21] and Machine learning-based code-smell detection [2], [22]–[24].

In the existing approaches, there are many limitations, including there no fair in the detection to detect all code-smells and they focused on some smells and neglected the rest of the code-smells. Moreover, many approaches used the same software to experiment, but they generated different results. This difference in the result due to some reasons, including there, is no standard definition for code-smells or they used different threshold values for metrics that were used to decide if the instances are smell or no.

In this paper, four code smells are selected to apply the experiments of our approach. The chosen code smells are God Class, Data Class, Feature Envy, and Long Method. This work aimed to increase the understanding of the relationship between software metrics and code smell prediction by defining the threshold values of software metrics produced by decision tree prediction rules. In this experiment, the decision tree algorithm is trained by tuning most of the parameters that may impact the accuracy by optimizing their values.

The remainder of this paper is structured as follows. Section 2 provides the related work. In section 3, we provide solution approach and research framework. Section 4 presents the results of the conducted experiments. Finally, in Section 5 we present the conclusion of our work.

II. RELATED WORK

In the literature many approaches have been proposed for detecting code smells in software systems.
Marinescu [25] proposed a smells detection approach based on software metrics. In this approach he defined detection strategy; it is a mechanism that combines software metrics using the logical operators (OR/AND). Abílio et al. [26] proposed a detection methodology based on software metrics. They investigated the software systems using their methodology and discussed the presence of Shotgun Surgery, God Class, and God Method smells in FOP language called AHEAD. Vidal et al. [19] proposed a smell detection approach to detect ten code smells including the popular smells, Feature Envy, God Method, and God Class. Based on their approach, they produced an Eclipse plugin for detecting code smells called JSpIRIT4.

Moha et al. [21] introduced a method for specifying and detecting smells using DSL (a Domain-Specific Language) called DÉCOR. DÉCOR track for smells by the sequence of steps starting with Description analysis, Specification, Processing, Detection, and Validation. Suryanarayana et al. [27] proposed a rule-based approach for smells detection. The smells detected by defining a set of rules that check if the class is referring to its subclasses. Rules are combined with metrics to detect code smells. Baudry et al. [28] proposed an approach to detect anti-patterns at the design level rather than the implementation level by using UML extension method. Langelier et al. [29] proposed visualization approach to detect code smells. Their visualization technique is combined with automatic analysis along with human expertise. They have introduced a framework that is specified for the quality analysis of large-scale systems. They used 3D Box for classes representation. Murphy-Hill et al. [30] proposed code-smells detector called Stench Blossom. Their approach firstly gives an interactive ambient visualization designed to provide the user a quick high-level overview of the code smells in their system, and then the users can deeply investigate the characteristics of the source code items that are affected by smells through a list of details that justify and explain the code smell presence. Carneiro et al. [31] have proposed a compound apprehension from different views to detect the bad smells in the source code. The multiple perspectives approach improves four categories of source code views with concern features, namely: concern's package class method structure, concern's inheritance wise structure, concern dependency, and concern dependency weight. They conducted an exploratory study to evaluate the extent to which visual views support code smell detection.

Kreimer[32] proposed a detection approach to detect two code smells based on a decision tree model. The approach was tested on the two small-scale software (WEKA package and IYC system). They found that the use of prediction models in code smells detection is effective.

Amorin et al. [33] confirmed Kreimer's finding by testing his decision tree model over medium -scale system. Khomh et al.[34], [35] proposed an approach using Bayesian Belief

Networks to detect three code smells from open-source software.

Fontana et al. [22] [2][24] [23] proposed several effective approaches in code smells detection. They built four datasets for four code smells by analyzing 74 software [36]. They conducted their experiments using 16 machine learning models. In [24], they proposed work on the classification of the severity of code smells. The degree of severity provided an assessment for refactoring by categorizing the code smells as harmfully degree, from the perspective that the high severity needs more efforts to maintainability.

We observed that the conventional code smells detection techniques have a limitation due to the ambiguity in the code smells definition and the selection of software metrics that indicate the occurrence of smells in the source code. Moreover, the quality of the rules that are used as an indication of the existence of smells might be non-exhaustive and inaccurate and is the main reason for the false positive result. In the metrics-based approaches, we observed that the difference of software metrics used and the selection threshold values of these metrics may cause a variation on the results because there is no standard benchmark for threshold values for metrics used due to the difference in the software domain and size. In machine learning based approaches, we recognized that the accuracy of these techniques relies on the training dataset quality. It may not enough build big size datasets that include all size and domain of the software, but should be taken into account many factors including, the manner of the dataset is built with balancing examples and preprocessing classification steps.

III. SOLUTION APPROACH

In this approach, we build code-smells prediction approach to generate the code smells prediction rules based on software metrics and decision tree algorithm. Software metrics have played a key role in measuring software quality by understanding the characteristics of the source code in software systems. Metrics capture the static information of source code such as the number of classes, methods, and parameters, and measure the coupling, and Cohesion between objects in the system. The list of steps we followed to build the code smells prediction model is as follows:
Initially, we prepared datasets by creating a new form of datasets — further, we are applying feature selection techniques to reduce the dimension of the dataset and select the most impact metrics. Next, we are training the decision tree algorithm by applying parameter optimization techniques. Finally, we evaluate the performance metrics and generating the prediction rules for each code smell.

A. Datasets And Dataset Representation

In this paper, four code smells were selected to apply the experiments of our approach. We created our datasets based on datasets that were published by Arcelli Fontana et al. [2]. We have adapted the original datasets to increase the realism factors of existence of the code smells side to side with other code-smells in the source code.

The selected code smells belong to class level (God Class, Data Class) and method level(Feature Envy, Long Method) [37][1]. Each instance in dataset represented by a software metrics vector, the instance consists of 61 software metrics in the class-level code smells, and 82 software metrics in the method-level code smells.

B. Feature Selection

Feature selection is a technique aimed to find the most influence features in the dataset by removing the redundant features to increase the performance and to increase the knowledge of the software metrics that play a significant role in code smell prediction.

In this paper, we applied two feature selection methods based on the Genetic algorithm. The first feature selection method is genetic algorithm based on Naïve Bayes as a fitness function, and the second method is genetic algorithm based on CFS (correlated features to the target class) [38] as a fitness function as shown in figure 1.
For both feature selection methods, we set the Genetic algorithm operators and parameters as a following:

- population size: 20 individuals
- maximum number of generations: 100 iterations
- selection method: tournament selection method
- cross-over method: one-point crossover
- cross over probability: 0.95
- mutation probability: 0.1

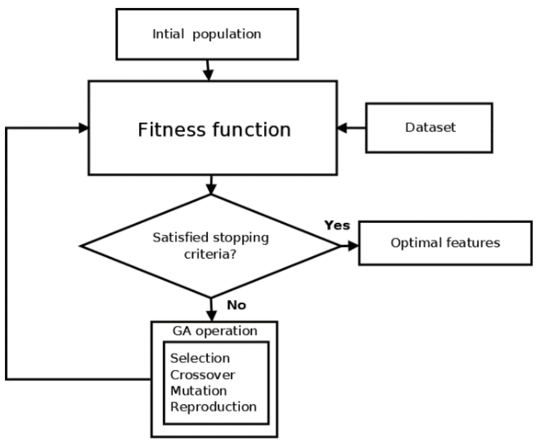

Figure 1 proposed feature selection method

*C. **Decision Tree Algorithm***

Tree-based algorithm considered one of the most accurate and commonly used supervised learning methods. Tree-based algorithms map the nonlinear relationships among the instance features and the target classes in the dataset, moreover, the algorithms provide classification rules that can ease the interpretation of the classification models[39].

The decision tree algorithm is a supervised learning algorithm that consists of nodes that are splitting based on splitting rules for each specific feature[40]. In the decision tree, the data is passed from root to leaves, and the feature value is separated based on splitting rules in each node according to predictor class until it reaches the stopping criteria. The hyperparameters of the decision tree algorithm, maximal depth and split criterion are optimized using parameter optimization techniques[41].

*D. **Parameter Optimization***

In the machine learning algorithms, some hyper-parameters need to be tuned to ensure the improvement of the performance accuracy of the algorithms. In this paper, we used a grid-search algorithm based on parameter optimization technique. In this paper, the Grid search algorithm is used to find the combination of the optimal values for parameters of the decision tree algorithm in order to optimize parameters on each dataset. The grid search algorithm is an optimization method, aims to find the optimal values for a set of parameters in a machine learning algorithm. It is based on exhaustively searches for the combination of parameters that returned the best performance value in the prediction model[42].

To performed the optimization, we identified a set of values that have to test for each parameter. For nominal parameters, we have defined the set nominal value, and for continuous parameters, we discretized the parameter values by setting the value range and number of steps for each parameter. The tested values are assigned within upper and lower bounds of range based on the specified steps assigned for each parameter as shown in Table 1.

Table 1 Tuning parameters and the specified steps assigned for each parameter

Parameter	From	To	Step
maximal depth	1	20	5
Apply pre-pruning	True or False		
Apply pruning	True or False		
Minimal size for split	1	10	10
Minimal gain	0	1	10
Criterion	gain ratio, information gain, Gini index, or accuracy		

E. Validation Methodology

For measuring the effectiveness of each experiment, we considered four performance parameters such as precision, recall, f-measure, and accuracy.

F-measure is defined as the harmonic mean of precision and recall, while the Precision is the positive-classified instances which are positive. The recall is the real-positive instances classified as positive. F-measure is a way of having a single number combining the two measures calculated using the following formulas.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F} - \text{measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \%$$

Accuracy is one of the performance measures for classification. It is the percentage of correctly classified instances in the positive and negative class and is calculated as follows:

$$\text{Accuracy (AC)} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

IV. EXPERIMENTAL RESULTS AND DISCUSSIONS

This study aimed to select the most relevant features that is not only for examining the impact of feature selection in improving the accuracy of the model, but also for comprehending the software metrics that play a significant role in the code-smells prediction process and producing the decision rules regarding to identify the code smells form software systems using software metrics. Also, this experiment aimed to increase the understanding of the relationship between software metrics and code smell prediction by defining the threshold values of software metrics produced by decision tree prediction rules.

Tables 2 and 3 showed the sets of features selected by feature selection methods for code-smell datasets. In the God class and Data class smell datasets, the GA -Naïve Bayes and GA-CFS methods selected 9 and 7 features respectively. In the long method dataset, both feature selection methods selected 8 features. Also, in feature envy dataset, 11and 7 features are selected by GA -Naïve Bayes and GA- CFS methods respectively.

Table 2 GA -Naïve Bayes feature selection methods results

Dataset	GA - Naïve Bayes	Selected metrics by GA -Naïve Bayes
Data class	9	AMW, CFNAMM, DIT, LCOM5, LOC_project, TCC, WOC, num_not_final_not_static_attributes, number_constructor_DefaultConstructor_methods
God class	9	CFNAMM, FANOUT, LOC, NMO, NOCS_project, NOII, NOM_project, WMC, number_constructor_DefaultConstructor_methods
long method	8	AMW, ATFD_method, CDISP_method, LOC_method, MAXNESTING_method, NIM, NOAV_method, NOCS_project
Feature envy	11	ATFD_method, CFNAMM, FANOUT, FDP_method, LAA_method, NMO, NOI_project NOII, NOM_project, NOPK_project, TCC

Table 3 GA- CFS feature selection methods results

Dataset	GA-CFS	Selected metrics by GA - CFS
Data class	7	AMW, NIM, RFC, NOAM, WMC, WOC, number_private_visibility_methods
God class	7	ATFD, CFNAMM, LOC, LOCNAMM, WMC, WMCNAMM
long method	8	CDISP_method, CYCLO_method, LOC_method, MAXNESTING_method, NOAV_method, NOLV_method, num_final_not_static_attributes, WMC
Feature envy	7	ATFD_method, ATFD, CDISP_method, FDP_method, LAA_method, LOC_method, NOMNAMM_package

As shown in table 4 The GA_CFS method scored the best accuracy in predicting the Data class smells by 98.05% and 98.54% F-measure. The parameters of the decision tree algorithm are tuned using the grid search algorithm and the best accuracy is achieved by setting the following parameter values:

- maximal depth = 5
- criterion = gain ratio
- apply pruning = true
- minimal size for split = 8
- minimal gain = 0.0
- apply pre-pruning = true

Table 4 Performance accuracy evaluation results

Code smell	GA -Naïve Bayes		GA- CFS	
	Accuracy	F-score	Accuracy	F-score
Data class	96.83%	97.00%	**98.05%**	98.54%
God class	97.32%	97.97%	**97.56%**	98.08%
Long method	92.96%	93.96%	**94.31%**	95.16%
Feature envy	**98.38%**	98.69%	98.11%	98.43%

The prediction rules that are extracted from the decision tree model to predict Data class is as follows:

```
WOC > 0.356
|   AMW > 1.310: no smell
|   AMW ≤ 1.310
|   |   NOAM > 3.500
|   |   |   NIM > 19.698: no smell
|   |   |   NIM ≤ 19.698: DATA class
|   |   NOAM ≤ 3.500: no smell
WOC ≤ 0.356
|   NOAM > 2.500
|   |   RFC > 43: no smell
|   |   RFC ≤ 43: DATA class
|   NOAM ≤ 2.500: no smell
```

The rules detected the data code smells when achieved the following conditions:

```
When WOC ≤ 0.356 && NOAM > 2.500 && RFC ≤
43
```

These conditions predicted 129 out of 140 data class smells, almost 92% of the number of data class smells in the dataset. These combinations of rules detect the data class if the weight of the class metric is less or equal 0.356 and the class has more than two accessor methods (getter and setter) with the response of the class metric value less than 43. On the other hand, when the class has less than two accessor methods, 44 no-smell classes were detected by these rules. Also, if the first and second rules are satisfied and the response of the class metric was more than 43, then no-smell classes are detected.

```
WOC > 0.356 && AMW ≤ 1.310 && NOAM >
3.500 && NIM ≤ 19.698
```

The second combination of rules that detected the data class is if the weight of the class metric is greater than 0.356 and the class have more than three accessor methods (getter and setter) with a number of inherited methods less than 19. That combination of rules detected 10 out of 140 data class smells.

In the God class smell, the GA-CSF feature selection method scored the best accuracy of 97.56% and with F-measure value 98.08%. The parameters of the decision tree algorithm that achieved this accuracy are tuned as follows:

- maximal depth = 5

- criterion = accuracy
- apply pruning = true
- minimal size for split = 1
- minimal gain = 0.2
- apply pre-pruning= true

The prediction rules that are extracted from the decision tree model to predict God class are as follows:

```
WMCNAMM > 47.500: GOD class
WMCNAMM ≤ 47.500
|   LOCNAMM > 415: GOD class
|   LOCNAMM ≤ 415: no smell
```

From the prediction rules, it is shown that only one condition is applied to detect the god class: WMCNAMM > 47.500. This condition detected 137 out of 140 God class smells in the dataset. The "Weighted methods count of not accessor or mutator methods" metric (WMCNAMM) considers the measure of the complexity of the method defined in the class excludes the accessor and mutator methods. On the other hand, the combination of rules to detect the no-smell classes is when WMCNAMM ≤ 47.500 and if the number of lines of code excluding accessor or mutator methods is (LOCNAMM) ≤ 415.

In the long method smell, the GA-CSF feature selection method scored the best accuracy of 94.31% and with F-measure value 95.16%. The parameters of the decision tree algorithm that achieved this accuracy are tuned as follows:

- maximal depth = 20
- criterion = accuracy
- apply pruning = false
- minimal size for split = 1
- minimal gain = 0.0
- apply pre-pruning= false

Prediction rules that are extracted from the decision tree model to predict long method smell as follows:

```
LOC_method > 79.500
|   CYCLO_method > 7.500: long method
|   CYCLO_method ≤ 7.500: no smell
LOC_method ≤ 79.500: no smell
```

The rules detected the long method smells when achieved the following conditions:

```
LOC_method > 79.500 && CYCLO_method >
7.500
```

These conditions predicted 140 out of 140 long method smells, that means 100% of the number of Long Method code smells in the dataset. The rules detect the smells if the number of lines of code for the method is greater than 79 and Cyclomatic Complexity for a method is greater than 7.5. The Cyclomatic Complexity is the maximum number of linearly independent paths in a method.

In the feature envy smell, the GA -Naïve Bayes feature selection method scored the best accuracy of 98.38% and with F-measure value 98.69%. The parameters of the decision tree algorithm that achieved this accuracy are tuned as follows:

- maximal depth = 9
- criterion = information gain
- apply pruning = false
- minimal size for split = 1
- minimal gain = 0.0
- apply pre-pruning = true

The prediction rules that are extracted from the decision tree model to predict feature envy smells as follows:

```
ATFD_method > 4.500
|   LAA_method > 0.323
|   |   ATFD_method > 8.500: feature envy
|   |   ATFD_method ≤ 8.500
|   |   |   NOPK_project > 30: feature
envy
|   |   |   NOPK_project ≤ 30
|   |   |   |   CFNAMM > 66: feature envy
|   |   |   |   CFNAMM ≤ 66: no smell
|   LAA_method ≤ 0.323: feature envy
ATFD_method ≤ 4.500
|   ATFD_method > 2.500
|   |   LAA_method > 0.450: no smell
|   |   LAA_method ≤ 0.450: feature envy
|   ATFD_method ≤ 2.500: no smell
```

The rules detected the long method smells when achieved the following conditions:

```
ATFD_method > 4.500 && LAA_method ≤ 0.323

ATFD_method > 4.500 && LAA_method > 0.323
&& ATFD_method > 8.500

ATFD_method > 4.500 && LAA_method > 0.323
&& ATFD_method ≤ 8.500 && NOPK_project >
30
```

The first combination of rules shows that if the method accessed to foreign data is greater than 4.5 and the locality of attribute accesses is equal or less than 0.323, these conditions predicted 113 out of 140 feature envy code smells in the dataset.

The second combination of rules that detects 16 out of 140 feature envy code smells is when the method accessed to foreign data is greater than 8.5, and the locality of attribute accesses is greater than 0.323.

The third combination of rules is when the no of method accessed to foreign data is greater than 4.5 and equal or less than 8.5, if the locality of attribute accesses is greater than 0.323 and the total number of packages in the system is more than 30. In this case, the rules detected 6 out of 140 smells in the dataset.

V. CONCLUSION

In this paper, we proposed an approach based on machine learning and software metrics to detect code smells from software systems and to find the metrics that play critical roles in the detection process. We have trained a decision tree model in binary-class datasets. We used two feature selection methods to select the influenced metrics in each code smell dataset. We applied our experiments using four code smells. This work aimed to increase the understanding of the relationship between software metrics and code smell prediction by defining the threshold values of software metrics produced by decision tree prediction rules, and we presented a set of conditions to detect each smell. We showed in the result that GA_CFS method scored the best accuracy in predicting the Data Class, God Class, and Long Method smells by 98.05%, 97.56%, and 94.31% respectively, and in the long method smell the GA -Naïve Bayes feature selection method scored the best accuracy of 98.38%.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.

[3] I. C. S. S. E. S. Committee and I.-S. S. Board, “IEEE recommended practice for software requirements specifications,” 1998.

[4] L. Chung and J. C. S. do Prado Leite, “On non-functional requirements in software engineering,” in *Conceptual modeling: Foundations and applications*, Springer, 2009, pp. 363–379.

[5] P. Rai, A. Pradhan, M. Pradhan, A. Chettri, and B. Limboo, “Comparative Study on Various Techniques Used in Examination System : A Survey,” vol. 7, no. 2, pp. 24–28, 2019.

[6] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: A taxonomy,” *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, 1990.

[7] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” *Proc. - Int. Conf. Softw. Eng.*, pp. 682–691, 2013.

[8] A. Yamashita and S. Counsell, “Code smells as system-level indicators of maintainability: An empirical study,” *J. Syst. Softw.*, vol. 86, no. 10, pp. 2639–2653, 2013.

[9] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 306–315, 2012.

[10] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, “Quantifying the effect of code smells on maintenance effort,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[11] D. SAHIN, M. KESSENTINI, S. BECHIKH, and K. DEB, “Code-Smells Detection as a Bi-Level Problem,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, 2014.

[12] E. Van Emden and L. Moonen, “Java quality assurance by detecting code smells,” *Proc. - Work. Conf. Reverse Eng. WCRE*, vol. 2002-Janua, no. November, pp. 97–106, 2002.

[13] M. Influential, P. Award, and E. Van Emden, “Assuring Software Quality by Code Smell Detection.”

[14] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoøberg, “Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, 2010.

[15] M. Tufano *et al.*, “When and why your code starts to smell bad,” *Int. Conf. Softw. Eng. ICSE*, vol. 1, pp. 403–414, 2015.

[16] O. Ciupke, “Automatic detection of design problems in object-oriented reengineering,” in *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings*, 1999, pp. 18–32.

[17] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” in *ACM Sigplan Notices*, 1999, vol. 34, no. 10, pp. 47–56.

[18] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, “A comprehensive approach for the development of modular software architecture description languages,” *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 2, pp. 199–245, 2005.

[19] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, “JSpIRIT: A flexible tool for the analysis of code smells,” *Proc. - Int. Conf. Chil. Comput. Sci. Soc. SCCC*, vol. 2016-Febru, 2016.

[20] R. Marinescu, “Measurement and quality in object-oriented design,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, vol. 2005, pp. 701–704, 2005.

[21] N. Moha and Y. Guéhéneuc, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[22] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, “Code smell detection: Towards a machine learning-based approach,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 396–399, 2013.

[23] U. Azadi, F. A. Fontana, and M. Zanoni, “Machine learning based code smell detection through WekaNose,” *Proc. 40th Int. Conf. Softw. Eng. Companion Proceeedings - ICSE ’18*, no. June, pp. 288–289, 2018.

[24] F. Arcelli Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowledge-Based Syst.*, vol. 128, pp. 43–58, 2017.

[25] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004, pp. 350–359.

[26] R. Ab’ilio, J. Padilha, E. Figueiredo, and H. Costa, “Detecting Code Smells in Software Product Lines--An Exploratory Study,” in *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, 2015, pp. 433–438.

[27] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[28] B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel, “Measuring and improving design patterns testability,” in

Proceedings of Metrics Symposium 2003, 2003.

[29] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 214–223.

[30] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization*, 2010, pp. 5–14.

[31] G. de F. Carneiro *et al.*, "Identifying code smells with multiple concern views," in *Software Engineering (SBES), 2010 Brazilian Symposium on*, 2010, pp. 128–137.

[32] J. Kreimer, "Adaptive detection of design flaws," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 4, pp. 117–136, 2005.

[33] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, 2015, pp. 261–269.

[34] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*, 2009, pp. 305–314.

[35] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, 2011.

[36] E. Tempero *et al.*, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010, pp. 336–345.

[37] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[38] M. A. Hall, "Correlation-based feature subset selection for machine learning," *Thesis Submitt. Partial fulfillment Requir. degree Dr. Philos. Univ. Waikato*, 1998.

[39] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, 1991.

[40] L. Rokach and O. Z. Maimon, *Data mining with decision trees: theory and applications*, vol. 69. World scientific, 2008.

[41] V. K. Gujare and P. Malviya, "Big Data Clustering Using Data Mining Technique," vol. 5, no. 2, pp. 9–13, 2017.

[42] C. J. L. Chih Wei Hsu, Chih Chung Chang, "A Practical Guide to Support Vector Classification," *BJU Int.*, vol. 101, no. 1, pp. 1396–400, 2008.

AUTHORS PROFILE

Mr. Mohammad Mhawish Bachelor of Computer Science from Al- Balqa' Applied University in 2009 and Master of Computer Science from Banaras Hindu University in year 2015. He is currently pursuing Ph.D in Computer Science from Banaras Hindu University.

Dr. Manjari Gupta is currently working as Associate Professor in the DST-CIMS, Banaras Hindu University, Varanasi, India. He is currently working in the area of Software Engineering.