Code Quality Metrics Derived from Software Design

Omar Masmali
Department of Computer Science
The University of Texas
El Paso, Texas USA
email: oamasmali@miners.utep.edu

Omar Badreddin
Department of Computer Science
The University of Texas
El Paso, Texas USA
email: obbadreddin@utep.edu

***Abstract*-Code smells are assumed to indicate bad design that can cause an unsustainable system. Many studies have tailored fixed threshold values for code smell metrics. However, these threshold values have ignored the fact that every system is unique, and it cannot be dynamically evolved throughout the codebase life cycle. This paper presents a novel approach that formulates dynamic code quality metrics with thresholds that are derived from software design. The first step in this approach is to measure the complexity of the design. Many researchers had developed many complexity metrics to measure the level of complexity in software models. Most of these metrics are limited and focus on counting the number of elements in each design, ignoring the unique characteristics of these elements and their interactions. In this study, we also propose a new methodology to measure the complexity of any software design. This measurement approach is based on evaluating each element in any class diagram by assigning a complexity rate. Finally, we propose a methodology to evaluate the effectiveness of this approach.**

Keywords - code quality; model-driven engineering; software quality metrics; UML class diagram; software design.

I. INTRODUCTION

An important goal of software engineering is to deliver software systems that can be sustainably maintained for an extended period of time. Software sustainability is a systematic challenge facing many communities, including professional software developers, open source communities and the research and scientific communities. It is estimated that half of software engineers' time and efforts are consumed performing avoidable maintenance activities. Current software code quality metrics that reply to code smells and technical debt suffer from key fundamental limitations. First, current methods are reactive in nature, as they are dependent on the emergence of adverse symptoms. Generally, such methods promote code refactoring to address deficiencies but provide little upfront guidance to avoid or minimize the emergence of such deficiencies. Moreover, current metrics are insensitive to diverse technologies, platforms and software contexts. This is a significant limitation, particularly at this period when software platforms, middlewares and contexts are in rapid flux. In addition, quality quantifications are not sufficiently fluid to adapt to changing software priorities and context throughout the software life cycle.

This paper presents a methodology to define code quality metrics with thresholds that are derived from software design. This ensures alignment between the intentional specification of software design characteristics and its implementation. This approach means that metrics can evolve as the codebase design evolves throughout the software lifecycle. Moreover, this approach means that each code module will have its own unique quality metrics that are tailored to its unique context. Also, in this paper, we introduce new complexity metrics for software designs. Many researchers had developed some complexity metrics to measure the level of complexity in software models [26]-[30]. Most of these metrics are limited in scope and focus on counting the number of elements in each design, overlooking the unique characteristics of these elements and their interactions. In this study, we propose a new methodology to measure the complexity of any software design. This measurement approach is based on evaluating each element in any class diagram by assigning a complexity rate.

The rest of the paper is structured as follows. In Section II, we describe the problem of current code quality metrics, then, we cover some related works. In Section IV, we present our proposed approach, and in Section V, we show the expected contribution of this work. In Section VI, we present the current status of our approach, and finally, we conclude our work in Section VII.

II. PROBLEM

Current code quality quantification methodologies adopt metrics with rigid thresholds. These methodologies do not adequately consider variations in development technologies and the architectural roles of various code and design elements. For example, one of the code quality metrics is large class code smell [1], defined as any class with more than 1000 lines of code. As software development platforms advance, managing a class with 1000 lines of code may no longer be detrimental to codebase quality. Similarly, high-performance computing platforms may require classes that are significantly larger in size to maximize performance. Moreover, long-living software systems may require significantly lower thresholds to accommodate the codebase as it evolves over an extended period of time.

To illustrate the current situation, consider the following simplified the Unified Modeling Language UML class diagram shown in Figure 1. The class diagram lists a data-heavy class (Class D), a computational heavy class (Class E) and some associations between classes. A software engineer who develops an implementation for this design, while following the design closely, will inevitably create code that suffers from significantly low sustainability quantification. For example, because Class D is data-heavy, its size, in terms of lines of code, will be very small, resulting in Lazy class code smell [14]. Similarly, Class C is designed to access many methods and attributes in other classes (it participates in five associations). The code analysis of Class C returns God class code smell [15].

 ISBN: 978-1-61208-827-3

Contemporary code analysis approaches that uncover code smells are agnostic to the intentions of the software designer, as demonstrated in the example above. Traditional analysis does not consider to what extent the implementation is aligned with the design. The identified code smells are frequently not an indication of unsustainable code but are, rather, a direct result of the intentional design specifications.

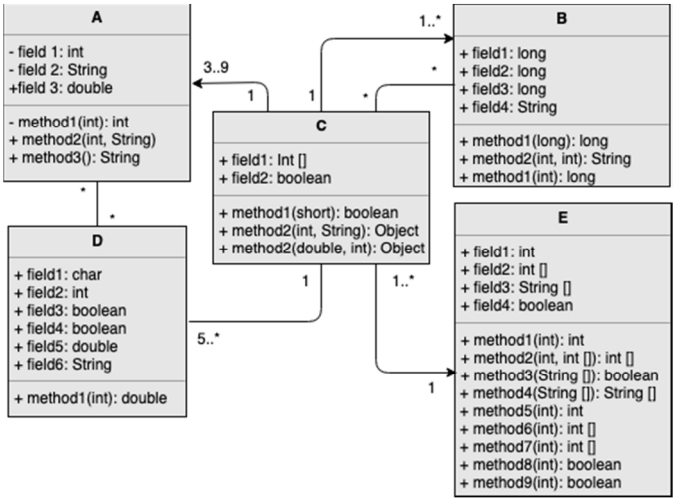

Figure 1. UML class diagram example.

Class D is Lazy because it is designed to host data and perform few computations. Class C is Large and has access to many external entities because it is designed as a root element in the design. Recommended code refactoring to remove the code smells will unavoidably suggest refactorings that are difficult to implement without violating the design specification. Therefore, we argue that such metrics with rigid thresholds are too rigid and are ineffective in characterizing codebase qualities.

III. RELATED WORK

This section will cover some related works in code quality metrics and design complexity metrics.

A. Code Quality Metrics

It has been argued that identifying appropriate code quality metrics and their thresholds is challenging, many have proposed using experience as a primary source for metric definition [21]- [23]. Code metrics are too sensitive to context and that metrics appropriate to one project are not adequate predictors for another. Aniche et al. investigated the effect of architecture on code metrics [4], proposing Software Architecture Tailored Thresholds (SATT), an approach that detects whether an architectural role is considerably different from others in the system in terms of code metrics and provides a specific threshold for that role. Our work presented in this paper is similar in the sense that it aims to improve the accuracy of code metric thresholds. However, while the SATT approach derives a unique threshold only if the architectural role of the module is deemed to be significantly different, our approach derives unique thresholds even in cases where the architectural role may only be slightly different. Gil and Lalouche demonstrated this phenomenon by applying both statistical and visual analyses of code metrics [2]. Fortunately, they demonstrate that context dependency can be neutralized by applying a Log Normal Standardization (LNS) technique. In a similar study, Zhang et al. showed that code metrics are dependent on six factors, namely, application domain, programming language, age, lifespan, the number of changes and the number of downloads [3].

Oliveria et al. proposed a method that extracts relative thresholds from benchmark data, and they evaluated their method in the Qualitas Corpus, finding that the extracted thresholds represent an interesting balance between real and idealized design rules [12]. Furthermore, Kapova et al. presented an initial set of code metrics to evaluate the maintainability that can be applied to different relational transformations, which play important roles when considering architecture refinement transformations [13]. The authors demonstrated the use of these metrics on a set of reference transformations to show their application in real-world settings and to help software architects judge the maintainability of their model transformations. Based on these judgments, software architects can take corrective actions (like refactorings or code-reviews) whenever they identify a decay in the maintainability of their transformations.

B. Design Complexity Metrics

Many different metrics for the class diagram has been developed to help software developers to analyze complexity and maintainability in the early phase of software lifecycle. One of them is developed by Peter, in [26], to analyze the complexity of architecture by using metric tree. He used UML diagram as an input to find some key indicators. He developed metrics to predict class's fault-proneness and to provide quality measurements. M. Genero discussed two groups of metrics to measure the complexity of class diagrams [30]. Kang et al. proposed weighted class dependence graphs to present a structure complexity measure for the UML class diagram by calculating classes and relationships between them [28]. They are using the entropy distance to measure the complexity of the class diagram. Use stochastic variables x and y to denote the output and input edges weight of each node. Doraisamy et al. proposed a model metric to be a guideline for software project managers in order to control and monitor software [25].

Moreover, a class diagram metrics proposed by Marchesi metrics to measure the complexity by balancing the responsibilities among packages and classes, and of cohesion and coupling among system entities [31]. The metrics are Design Size in Classes (DSC), Number of Hierarchies (NOH), Average Number of Ancestors (ANA), Direct Class Coupling (DCC), Cohesion Among Method of Class (CAM), and Measure of Aggregation (MOA). Chidamber and Kemerer proposed some metrics, only three of them for measuring the UML class diagram [27][29] which are Number of Children (NOC), Depth of Inheritance Tree (DIT), and Weighted methods per Class (WMC). Concas in his work focuses on investigating process complexity [5]. He defines process

 ISBN: 978-1-61208-827-3

complexity as the degree to which a business process is difficult to analyze, understand or explain. claims that the only way to analyze the process complexity is by using the process control-flow complexity metrics. Ma et al. [32] proposed a hierarchical metrics set in terms of coupling and cohesion for large-scale Object-Oriented (OO) software systems. They analyzed the proposed approach on a sample of 13 open-source OO software systems to empirically validate the set. Fourati et al. [33] propose an approach that identifies anti-patterns in UML designs through the use of existing and newly defined quality metrics that examines the structural and behavioral information through the class and sequence diagrams. It is illustrated through five of some well-known anti-patterns: Blob, Lava Flow, Functional Decomposition, Poltergeists, and Swiss Army Knife. Kim and Boldyreff suggested a software metrics that can be applied to the elements of UML modelling [24]. The proposed UML metrics are based on the metamodel scheme and divided into four categories of metrics which are model, class, message, and use case metrics.

IV. PROPOSED SOLUTION

Our proposed approach derives code quality metrics and their dynamic threshold values from software designs. Beginning, our approach focuses on design elements pertaining to data types, their complexities, frequencies, and the estimated complexity of the operations of such data. Then, from the estimated class and method complexity, We quantified fuzzy quality metrics to measure two of the bad code smells, which are large class and long method.

A. Complexity Metrics

The approach involves assigning complexity rate [20] values to each attribute, method and association within the class as shown in TABLE I. We assign a complexity rate (Co_{Rate}) to an attribute's visibility ($V_{Att.}$) and type ($T_{Att.}$) to estimate the attribute's complexity (Att_{comp}). Each complexity rate (Co_{Rate}) can be primitive, simple, or complex. Then we estimate method complexity ($Method_{comp}$) by summing the complexity of the method's visibility ($V_{Meth.}$), the return type ($R_{Meth.}$) and the total parameters list ($P_{Meth.}$). Further, We estimate the association complexity ($Asso_{comp}$) by adding all incoming ($IN_{As.}$) and outgoing ($OUT_{As.}$) association links. Finally, by summing all attributes, methods and association complexities, we can estimate the class complexity ($Class_{comp}$), which we use to quantify code quality factors, such as expected lines of code for any class ($ELOC(Class)$).

The following formulas describe the proposed approach. Formula 1 estimates the complexity of the attributes, as derived from the UML class diagram shown above. Formula 2 estimates method complexity based on the complexity of the parameters and return types. Formula 3 estimates the complexity of the association for each class. Formula 4 uses the previous calculations to estimate the class complexity. The following describes the quantification approach in greater detail.

$$Att_{comp} = (V_{Att.} * Co_{Rate}) + (T_{Att.} * Co_{Rate}) \quad (1)$$

where (Att_{comp}) is attribute complexity, ($V_{Att.}$) attribute visibility, ($T_{Att.}$) attribute type and (Co_{Rate}) the complexity rate.

TABLE I. CLASSIFICATION OF THE COMPLEXITY RATE

Element	Scope	Name	Classification	Examples	Rating
Attributes	Visibility	Att_{Vis}	Primitive	Private	1
			Simple	Protected, Package	2
			Complex	Public	3
	Type	Att_{type}	Primitive	int, char, boolean	1
			Simple	float, long, double, str	2
			Complex	array, struct, tuple, date, time, list, map	3
			Derived	object, array of complex types	4
Methods	Parameters	$P_{Meth.}$	Primitive	int, char, boolean	1
			Simple	float, long, double, str	2
			Complex	array, struct, tuple, date, time, list	3
			Derived	object, array of complex types, map	4
	Return Type	$R_{Meth.}$	Primitive	int, char, boolean, void	1
			Simple	float, long, double, str	2
			Complex	array, struct, tuple, date, time, list	3
			Derived	object, array of complex types, map	4
	Visibility	$V_{Meth.}$	Primitive	Private	1
			Simple	Protected, Package	2
			Complex	Public	3
Association	Incoming	$IN_{As.}$	Primitive	1 to many	1
			Simple	many to many, 1 to 1	2
			Complex	all others (such as n .. m to many, etc..)	3
	Outgoing	$OUT_{As.}$	Primitive	1 to many	1
			Simple	many to many, 1 to 1	2
			Complex	all others	3

$$Method_{comp} = (V_{Meth.} * Co_{Rate}) + (R_{Meth.} * Co_{Rate}) + \left(\sum_{i=1}^{n} (P_{Meth.} * Co_{Rate})\right) \quad (2)$$

Here, ($Method_{comp}$) is method complexity, ($V_{Meth.}$) is method visibility, ($R_{Meth.}$) is method return type and ($\sum_{i=1}^{n}(P_{Meth.} * Co_{Rate})$) is the complexity rate for all parameters in the method.

$$Asso_{comp} = \left(\sum_{i=1}^{n}(IN_{As.} * Co_{Rate})\right) + \left(\sum_{i=1}^{n}(OUT_{As.} * Co_{Rate})\right) \quad (3)$$

($Asso_{comp}$) is the association complexity, ($\sum_{i=1}^{n}(IN_{As.} * Co_{Rate})$) is the complexity for all incoming associations to the class and ($\sum_{i=1}^{n}(OUT_{As.} * Co_{Rate})$) is the complexity for all outgoing associations.

$$Class_{comp} = \left(\sum_{i=1}^{n} Att_{comp}\right) + \left(\sum_{i=1}^{n} Method_{comp}\right) + Asso_{comp} \quad (4)$$

The class complexity ($Class_{comp}$) can be calculated by summing the complexity of all class attributes $\left(\sum_{i=1}^{n} Att_{comp}\right)$, the complexity of all methods in the class $\left(\sum_{i=1}^{n} Method_{comp}\right)$ and the class association complexity ($Asso_{comp}$).

Class complexity ($Class_{comp}$) and method complexity ($Method_{comp}$) can be used to estimate the expected lines of code for the class, or any method within the class, by multiplying them by the class factor (F_{Class}) or method factor (F_{Mehtod}). Both the class factor and method factor will be estimated empirically as part of the planned research activities.

B. Fuzzy Metrics

The fuzzy quality metrics are a new methodology to measure two of the bad code smells, which are large class and long method. This methodology is based on measuring the difference between the actual and expected values of the lines of code for the class and method. To demonstrate this concept, we illustrate a fuzzy metric for the large class and long method code metrics [20].

$$FuzzyMetric(Class) = LOC(class) - ELOC(class) \quad (5)$$

$$ELOC(Class) = Class_{comp} * Class(LOC_{factor}) \quad (6)$$

$$Class(LOC_{factor}) = \frac{LOC(Total) * LOC(Average)}{Class_{comp}(Total) * Class_{comp}(Average)} \quad (7)$$

Where $ELOC(class)$ is the expected size in terms of lines of code, $LOC(Total)$ is the total LOC for all classes, and $LOC(Average)$ is the average of LOC for all classes. Similarly, the fuzzy metric for method is defined as follows:

$$FuzzyMetric(Method) = LOC(Method) - ELOC(Method) \quad (8)$$

$$ELOC(Method) = Method_{comp} * Method(LOC_{factor}) \quad (9)$$

$$Method(LOC_{factor}) = Method_{comp}(Average) \quad (10)$$

V. Expected Contributions

The expected contribution of this work is to present a new methodology for estimating software code quality. We expect that design-driven code quality metrics will improve the maintainability and sustainability of software systems by considering the variations in development technologies and the architectural roles of various code and design elements. This ensures that the derived metrics are uniquely tailored to the software under development and the derived metrics can dynamically evolve throughout the codebase life cycle. Another contribution in this paper is to introduce new complexity metrics for software designs. The approach is based on evaluating every element in each software design by assigning a relative complexity rate. The complexity rate can be either primitive, simple, or complex. As such, the complexity of a system can be estimated by summing the complexity values of all elements within the system.

VI. Current Status

This work has been formulated and submitted to different conferences. Overall, the plan came over the following phases. Phase 1: Define the complexity metrics for software design and evaluate it theoretically and empirically. Four conference papers have been submitted based on the first phase. One of those papers has been accepted at the 20th IEEE International Conference on Software Quality, Reliability, and Security. Two other papers were accepted at the Future Technologies Conference 2020 [34][35]. The fourth paper is under review at the Software Quality Days conference 2021. Phase 2: formulate and evaluate the fuzzy quality metrics. In this phase, three conference papers have been submitted to some venuses. The first one has been accepted and presented at the International Conference on Model-Driven Engineering and Software Development MODELSWARD 2020 [20]. The other two paper are under review at the International Conference on Computer Science and Software Engineering CASCON 2020, and 20th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2020. Phase 3: Completing, submitting, and defending the dissertation.

The preliminary results pertaining to class-level complexity and code fuzzy smell are as follows. In class complexity we have applied the proposed approach on code repositories obtained from opensource projects. The selection criteria considered code repositories that are most active on GitHub [16]. We included, among others, codebases developed by Google [17], Microsoft [18] and the National Security Agency [19]. We compared the results from our quantification metrics to the actual metrics derived from the codebase analysis (Figure 2). High correlation with 84%, would suggest that our metrics accurately characterize codebase quality. In the near future, we plan to compare correlation values obtained from this approach to those obtained from applying traditional code quality metrics.

 ISBN: 978-1-61208-827-3

In fuzzy code smells we attempted an extensive empirical evaluation of fuzzy code smell approach using expert reviews of large corpuses [37] of smells in open source repositories by comparing our metics, and a wide range of static code analysis tools (PMD [38], infusion [39], JDeodorant [40], and JSpIRIT [41]), against the expert reviews data sets. The results for the precision and recall show that fuzzy smell method aligned significantly better with expert's data sets than contemporary code analysis tools as shown in Figure 3.

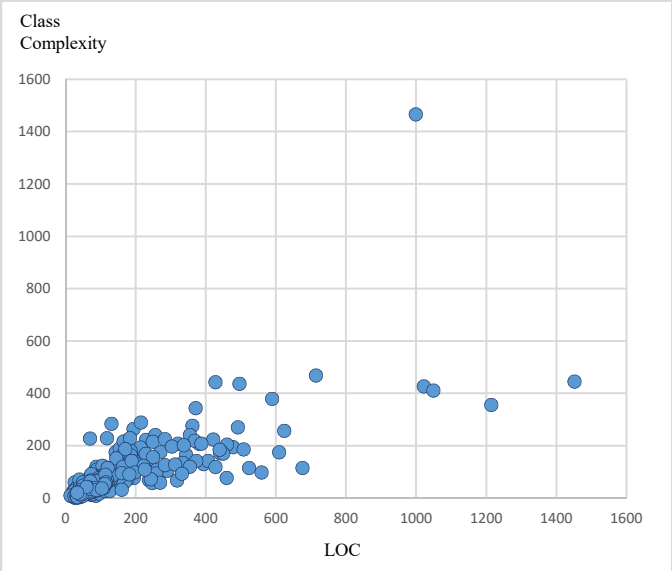

Figure 2. Correlation between class complexity and LOC.

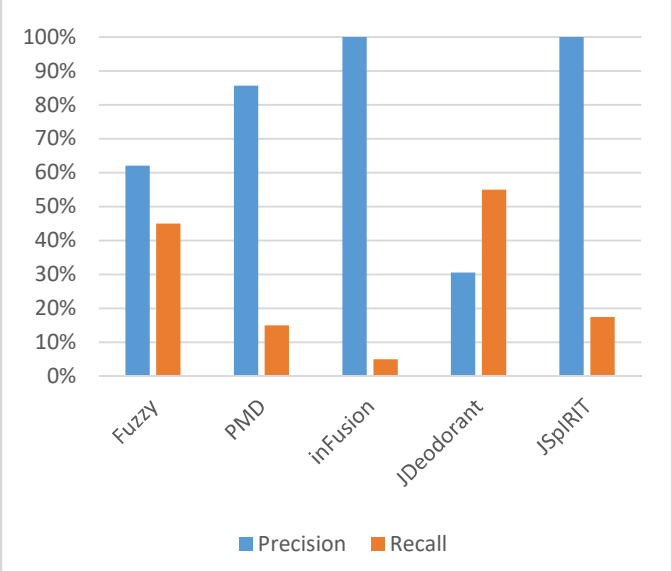

Figure 3. Precision and recall of large class code smell.

Finally, we calculated the F1 score, which is the harmonic mean of precision and recall. The F1 score is used because in many studies, the F-measure is the ultimate measure of performance of a classifier [36]. After calculating the F1 score for all the approaches, we found that the best performance for detecting a bad large class smell is our approach. Figure 4 shows that the accuracy of the fuzzy metric is the highest with 55%. The second highest is PMD with 39%, then JDeodorant with 33%, and after that JSpIRIT with 27%. The lowest accuracy is found for inFusion with only 7%. Moreover, for detecting a bad long method smell, we found that our approach is the best as well.

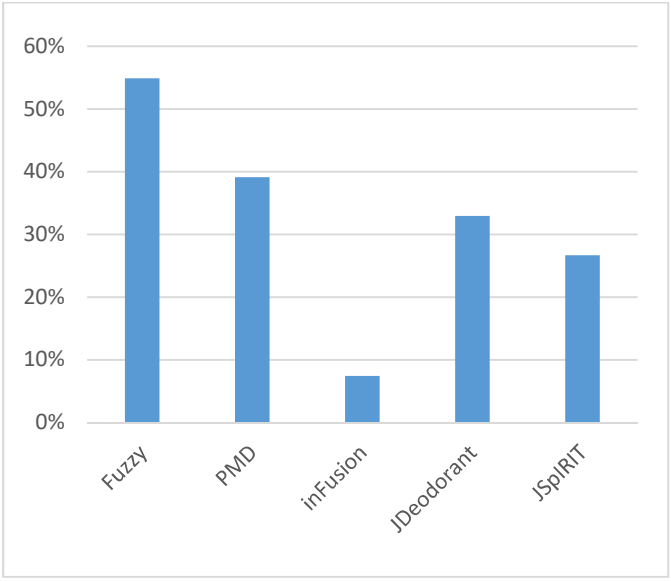

Figure 4. The total F1 score for classes of each tool.

VII. Conlusion and Future Work

In this paper, we presented a new approach that defines code quality metrics with thresholds that are derived from software design. This ensures alignment between the intentional specification of software design characteristics and their implementation. This approach means that metrics can evolves as the codebase design evolves throughout the software lifecycle. Moreover, this approach means that each code module will have its own unique quality metrics that are tailored to its unique context. Our approach started with measuring the complexity of each class and method in the system. We then estimated the expected size for each class and method by using the complexity measurement that we calculated from the first step. The last step is to calculate the fuzzy code smell based on the difference between the actual and expected size of each class and method.

The research plan in future work is to evaluate the proposed metrics theoretically and empirically by using the following methodologies: (1) Theoretical evaluation of the complexity metrics by using Weyuker's nine properties model. (2) Evaluate whether the metrics derived from software designs provide a better characterization of codebase quality and sustainability than alternate traditional metrics. (3) Quantify thresholds for the fuzzy code smells derived from the software design. (4) Compare our new fuzzy code smells with code smells resulting from code smells detection tools for different codebases.

 ISBN: 978-1-61208-827-3

REFERENCES

[1] R. Oliveira et al., “Identifying code smells with collaborative practices: A controlled experiment,” presented at X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). Brazil, 2016.

[2] J. Y. Gil and G. Lalouche, “When do software complexity metrics mean nothing? When examined out of context,” Journal of Object Technology, vol. 15, no. 1, pp. 1-25. 2016.

[3] F. Zhang, A. Mockus, Y. Zou, F. Khomh and A. E. Hassan, “How does context affect the distribution of software maintainability metrics?” In 2013 IEEE International Conference on Software Maintenance, pp. 350-359.

[4] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, “SATT: Tailoring code metric thresholds for different software architectures,” presented at IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). Raleigh, NC, USA, 2016.

[5] G. Concas, M. Marchesi, S. Pinna and N. Serra, “Power-laws in a large object-oriented software system,” IEEE Transactions on Software Engineering, vol. 33, no. 10, pp. 687-708, 2007.

[6] R. Wheeldon and S. Counsell, “Power law distributions in class relationships,” In Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003, pp. 45-54.

[7] Y. Yao, S. Huang, Z. Ren and X. Liu. “Scale-free property in large scale object-oriented software and its significance on software engineering,” In 2009 Second International Conference on Information and Computing Science, vol. 3, 2009, pp. 401-404.

[8] I. Herraiz, D. M. German and A. E. Hassan, “On the distribution of source code file sizes,” In ICSOFT (2), 2011, pp. 5-14.

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

[10] D. Coleman, B. Lowther and P. Oman, “The application of software maintainability models in industrial software systems,” Journal of Systems and Software, vol. 29, no. 1, pp. 3-16, 1995.

[11] B. A. Nejmeh, “Npath: A measure of execution path complexity and its applications,” Communications of the ACM, vol. 31, no. 2, p. 188, 1988.

[12] P. Oliveira, M. T. Valente and F. P. Lima, “Extracting relative thresholds for source code metrics,” IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Belgium, 2014.

[13] L. Kapova, T. Goldschmidt, S. Becker and J. Henss, “Evaluating maintainability with code metrics for model-to-model transformations,” International Conference on the Quality of Software Architecture, 2010.

[14] D. Taibi, A. Janes, and V. Lenarduzzi, “How developers perceive smells in source code: A replicated study,” Information and Software Technology Journal, Vol. 92, pp. 223-235, December 2017.

[15] F. A. Fontana, V. Ferme, and M. Zanoni, “Towards assessing software architecture quality by exploiting code smell relations,” IEEE/ACM 2nd International Workshop on Software Architecture and Metrics. pp. 1-7, Italy, 2015.

[16] https://github.com/ , Accessed Feb. 2019

[17] https://github.com/google , Accessed Nov. 2019

[18] https://github.com/microsoft , Accessed Nov. 2019

[19] https://github.com/nationalsecurityagency , Accessed Nov. 2019

[20] O. Masmali and O. Badreddin. "Towards a Model-based Fuzzy Software Quality Metrics." In MODELSWARD, pp. 139-148. 2020.

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

[22] D. Coleman, B. Lowther, and P. Oman, “The application of software maintainability models in industrial software systems,” Journal of Systems and Software, vol. 29, no. 1, 1995.

[23] B. A. Nejmeh, and W. Riddle, "The PERFECT approach to experience-based process evolution." In Advances in computers, vol. 66, pp. 173-238. Elsevier, 2006.

[24] H. Kim, and C. Boldyreff, “Developing software metrics applicable to UML models,” Proc. of the 6th ECOOP Workshop on Quantitative Approaches in Object-oriented engineering, Malaga, Spain, June 2002.

[25] M. Doraisamy, S. bin Ibrahim, and M. N. Mahrin, "Metric based software project performance monitoring model", Proceedings of the IEEE International Conference on Open Systems (ICOS), August 2015.

[26] P. In, S. Kim, and M. Barry, “UML-based object-oriented metrics for architecture complexity analysis”. Department of computer science, Texas A&M University, 2003.

[27] M. Manso, M. Genero, and M. Piattini, “No-redundant metrics for UML class diagram structural complexity”. Lecture Notes on Computer Science, 2681, 2003, pp.127-142.

[28] D. Kang et al., “A structural complexity measure for UML class diagrams.” In International Conference on Computational Science 2004 (ICCS 2004), Krakow Poland, June 2004, pp. 431-435, 2004.

[29] J. Bansiya, and C. G. Davis, “A Hierarchical Model for Object-Oriented Design Quality Assessment.” IEEE Trans. on Software Engineering, 28, 1, 2002, pp. 4-17.

[30] M. Genero, M. Piattini, and C. Calero, "A survey of Metrics for UML Class Diagrams". Journal of Object Technology, 4 (9). p. 59-92. 2005.

[31] M. Marchesi, “OOA metrics for the unified modeling languages.” In Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), Palazzo degli Affari, Italy, March 1998, pp. 67-73, 1998.

[32] Ma, Yu-Tao, K. He, B. Li, J. Liu, and X. Zhou, "A hybrid set of complexity metrics for large-scale object-oriented software systems." Journal of Computer Science and Technology 25, no. 6, pp. 1184-1201, 2010.

[33] R. Fourati, N. Bouassida, and H. B. Abdallah. "A metric-based approach for anti-pattern detection in UML designs." In Computer and Information Science 2011, pp. 17-33. Springer, Berlin, Heidelberg, 2011.

[34] O. Masmali and O. Badreddin, "Code Complexity Metrics Derived from Software Design: Framework and Theoretical Evaluation", In Proceedings of the Future Technologies Conference (FTC 2020), Canada, Vancouver, 2020.

[35] O. Masmali and O. Badreddin, "Theoretically Validated Complexity Metrics for UML State Machines Diagram", In Proceedings of the Future Technologies Conference (FTC 2020), Canada, Vancouver, 2020.

[36] G. Forman, "An extensive empirical study of feature selection metrics for text classification." Journal of machine learning research 3, 2003, pp 1289-1305.

[37] T. Paiva, A. Damasceno, E. Figueiredo and C. Sant Anna, "On the evaluation of code smells and detection tools," Journal of Software Engineering Research and Development, Springer 2017.

[38] PMD. Accessed Feb. 2020. Avalibale at: https://pmd.github.io/

[39] inFuction. Accessed Feb. 2020. Avalibale at: http://loose.upt.ro/iplasma/

[40] JDeodorant. Accessed Feb. 2020. Avalibale at: https://github.com/tsantalis/JDeodorant

[41] JSpIRIT. Accessed Feb. 2020. Available at: https://sites.google.com/site/santiagoavidal/projects/jspirit

 ISBN: 978-1-61208-827-3