Adapting God Class thresholds for software defect prediction: A case study

Mitja Gradišnik, Tina Beranič, Sašo Karakatič
Faculty of Electrical Engineering and Computer Science
University of Maribor
Koroška cesta 46, Maribor, Slovenia
{mitja.gradisnik | tina.beranic | saso.karakatic}
@um.si

Goran Mauša
Faculty of Engineering
University of Rijeka
Vukovarska 58, 51000 Rijeka, Croatia
goran.mausa@riteh.hr

***Abstract*— In software engineering there is an active research field of defect prediction using software metrics. While the research shows that the prediction of defects using software metrics performs well, prediction using metrics alone lacks clear refactoring capabilities. On the other hand, code smells have the ability to describe the code anomalies precisely, and suggest their refactoring. Therefore, code smells can be a much better starting position for software fault prediction.**

In this paper, we present the results of preliminary research on the ability to predict software defects with the code smell God Class. The aim of our research was to test the definition of God Class, as defined by Lanza and Marinescu in 2006, in the ability to predict defects in a case study of the open source projects JDT and PDE within the Eclipse framework. The definition of the God Class was adapted using the grid search technique, with the goal of maximizing the fault prediction ability while keeping the base of the original definition. The results show that adaption of the definition in the specific project resulted in improved fault prediction ability.

I. INTRODUCTION

Deploying a software product with a minimal amount of defects requires the usage of various quality assurance techniques as early as possible in the development life cycle, with software defect prediction being a very important one. Although the prediction approaches based on software metrics was proven to be useful [1], [2], [3], they do not provide guidelines regarding refactoring opportunities of a software entity. Contrarily, the advantage of code smells lies in a clear understanding of the detected anomalies within the entity, resulting in unambiguous directions for refactoring.

Code smells were first introduced by Martin Fowler [4] in 1999. Since then, they have attracted a lot of attention, and, therefore, many different definitions can be found. In general, a code smell indicates structural characteristics resulting from poor design decisions and implementation choices that have a negative impact on the quality of the developed software system [4], [5], [6], [7]. Fowler et al. [4] highlighted 22 bad smells that can be classified into seven groups proposed by Mntyl et al. [8].

Code smells could be detected with the help of predefined rules implemented within different tools [9] aimed for static analysis of the source code. Rules depend on software metrics and chosen threshold values, that could vary in different development environments. In the presented study, we limit work to the God Class code smell, and follow the definition provided by Lanza and Marinescu [10]. The defined rule is implemented within the PMD tool, wherein the applied software metrics and related threshold values are being taken from the same source. The aim of the presented research is to *adjust the default threshold values for the metrics ATFD, WMC and TCC, with the goal of maximizing the fault prediction ability*. The study was done using the grid search technique on the case study of the open source projects JDT and PDE within the Eclipse framework. In this research, we limited our scope only on the God Class code smell, and only on one source code dataset. Our intent was to test if the official definition of a particular code smell can be adjusted to align better with the fault prediction.

The structure of the paper consists of the following parts. The second section introduces the definition of the God Class, and investigates the connection between the God Class and the prediction of faults in the software products. The third section describes the case study on the connection between God Class in the software fault prediction in the case of the Eclipse JDT and PDE source code. The research methodology, the data used and the analysis procedures are presented herein. Following are the sections with the discussion of the results of the case study, and the conclusion that can be made based on the results of the analysis.

II. USING GOD CLASS TO PREDICT THE SOFTWARE FAULTS

A. God Class detection

Different strategies for identifying code smells exist, starting with the manual review, or by using predefined rules. The rules are based on combining different software metrics, since the evaluation of software projects based on just one software metric can offer insufficient results [10]. Lanza and Marinescu [10] defined detection strategies for detecting design flaws affecting classes and methods. As they say, the detection strategy consists of logical conditions that are based on software metrics, and can detect design fragments with specific properties [10]. Rules are defined for six code smells (1) God Class, (2) Feature Envy, (3) Data Class, (4) Brain Method, (5) Brain Class, and (6) Significant Duplication.

The code smell God Class is similar to the code smell Large Class defined by Fowler et al. [4] that fit the group Bloaters, proposed by Mntyl et al. [8]. God Class indicates a software class that tends to centralize the intelligence of the system by performing too much work, and delegating only minor details to other classes while using the data from other classes [10]. With this, the reusability and understandability of a certain class is decreased [10]. Therefore, the detection of God Class is done using three software metrics [10]:

- ATFD (Access To Foreign Data) expressing the number of foreign attributes used by a software class.
- WMC (Weighted Method per Class) is the sum of all statistical complexity of all methods in a software class.
- TCC (Tight Class Cohesion) represents a relative number of method pairs of a class that accesses in common at least one attribute of the measured class.

The God Class is a class that has a high value of metric ATFD, has a very high WMC and a low value of TCC, with respect to the used threshold values. The defined rule is presented graphically in Fig. 1, including the default threshold values [10].

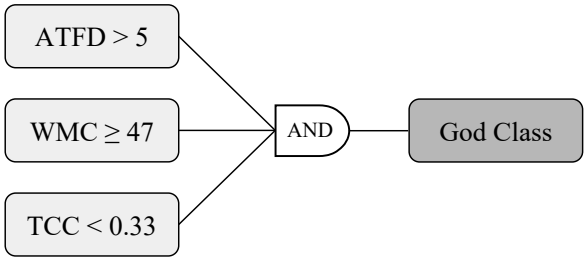

Fig. 1. The God Class detection rule with corresponding threshold values

The God Class code smell is implemented within different rules aimed for identification of code smell or for finding the refactoring opportunities. Among others, the detection of code smells can be done with JSpIRIT [11], JDeodorant [12] and PMD [13]. The latter was used in the implemented study since it follows the rules and thresholds defined by Lanza and Marinescu [10] strictly.

To increase the correctness of identification, it is crucial that reliable threshold values are used for the used software metrics. Although the PMD tool uses predefined threshold values defined by Lanza and Marinescu [10], they can be modified by the user. In the presented study, they were adjusted in order to find the combination of threshold values that had the highest ability to detect software defects.

B. Predicting software faults with God Class

Several studies have addressed the impact of God Class on the frequency of detected defects in studied software components. Despite a number of studies in this research field, the correlation between God Class and the number of detected defects remains discordant and inconclusive.

Regarding the impact of code smells on detected defects, Li and Shatnawi [14] studied the relation between Data Class, Feature Envy, God Method, God Class, Refused Bequest, and Shotgun Surgery on detected defects in three projects of the Eclipse framework, namely Eclipse Platform, Eclipse JDT, and Eclipse PDE. The authors found a positive association between God Class affected classes and the number of detected defects in the studied software.

Olbrich, Cruzes, and Sjoberg [15] conducted research on the impact of the code smells God Class and Brain class on the number of detected defects. The study was carried out on three software products, namely Apache Lucene, Apache Xerces 2 Java, and Log4J, and took into consideration the size of classes containing the code smells. Interestingly, the study results showed, that when normalizing a defect rate with respect to class size, God Class even has a negative correlation with the number of defects.

Along with Brain Class and Feature envy, the researchers Marinescu and Marinescu [16] studied the impact of the God Class code smell on the number of defects detected in three versions of Eclipse IDE (2.0, 2.1, 3.0). The study did not confirm any correlation between God Class and an increased likelihood of defect detection.

Finally, Zazworka et al. [17] observed the impact of the God Class code smell on defect-proneness in the Hadoop software package. The study confirmed a positive correlation with higher defect-proneness in the context of the source code of the Hadoop library.

III. CASE STUDY

A. Research design

The case study performed in this paper is a preliminary analysis aimed to investigate the God Class code smell. It is based on publicly available software defect prediction datasets from the Eclipse Java Development Tools (JDT) open source project.

In order to determine the coverage of the new definitions of God Class over the faulty files, we measured a couple of classification metrics – coverage accuracy and F-score of coverage. The coverage accuracy was calculated as is shown in Equation 1:

$$Accuracy = \frac{Number\ of\ faulty\ God\ class\ files}{Number\ of\ all\ files} \quad (1)$$

As the ratio between the faulty and fault free files in the JDT source is not balanced, the measure of accuracy is not advised [18], [19], as it is biased towards the majority class (in our case, the fault-free files). To tackle this, we also calculated the average F-score, which is a harmonic mean between the precision and the recall for the individual class [20], as is shown in Equation 2 bellow, and the overall F-score is calculated as the weighted average of both F-scores (Equation 3):

$$Fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (2)$$

$$Fscore_{overall} = F \cdot Fscore_F + NF \cdot Fscore_{NF} \quad (3)$$

F is the share of faulty files, $Fscore_F$ is the F-score for faulty class, and NF indicates the share of fault free files, while $Fscore_{NF}$ is the F-score for faulty free classes.

B. Datasets

For the purposes of our case study, we prepared two datasets. Both of them are based on java classes from the source code of pieces of software under the study. The first dataset contains data about the faultiness of the observed classes. The classes containing more than one detected defect are marked as faulty in the dataset. The second dataset contains metrics measurements, related to identification of God classes in the projects' source code. Based on these two datasets, we could identify God Classes in the studied software, and build defect prediction models for the observed piece of software.

The defect dataset was generated in our previous work [21] according to the systematic data collection procedure. In accordance with the existing defect dataset, we ran our experiments on the source code of two independent projects within the Eclipse IDE development process of, namely, the Eclipse JDT[1] and Eclipse PDE[2] projects. Both projects are an important component of the open-sourced Eclipse Integrated Development Environment (IDE)[3], a development tool particularly popular among Java developers. We used version 3.2 of both projects, although the selected version is somewhat obsolete from todays perspective, because its source code is still available and can be downloaded freely from the Eclipse code repositories[4].

The dataset of software metric measurements of software metrics is calculated using the PMD tool. The PMD tool [13] is able to identify God Class code smells in source code according to the definition and thresholds proposed by Lanza and Marinescu [10]. Therefore, it can also be used to retrieve all software metrics required for God Class identification. Because PMD is not designed primarily as a metrics measurement tool, some small changes in the source code of the tool downloaded from its GitHub repository[5] were required, in order to to extract the internal values of the desired metric measurements. With the use of a slightly modified PMD tool, we were able to measure values of WMC, ATDF, and TCC metrics in the source code of the observed software. In the metrics' measurement, we included only source files in the source directory of the project. Other source and test files found in the projects were omitted. Compliance of God Class definition with the definition given by Lanza and Marinescu and the availability of its source code was the main reason why we chose the PMD tool, and not one of the alternatives examined by Paiva at al [9].

C. The case study

To find the thresholds of God Class definition for optimal defect detection in the studied software, we implemented the grid search algorithm as a Java-based console application. The console application iterates through all sensible threshold values of the metrics used for God Class definition, identify God Classes in the software, and test how accurately a modified God Class definition can predict defect faulty classes in the studied software. The grid search implemented in the console application followed the brute force approach. Basically, we iterated through all sensible thresholds using three nested loops, one for each of the metrics from the definition. In order to speed up the grid search, the console applications design exploits the multi-core architecture of the CPUs by executing the program code in multiple threads.

For the purpose of our case study, the domain of thresholds used in the grid search was limited up and down by minimal and maximal values of the measured metrics obtained from the software by the PMD tool. As shown in Table I, the minimum threshold value for the WMC metric was set to 0, and the maximal value was set to 1630. Because the value of the WMC metric is an integer number, the threshold values used in the grid search were incremented by the value 1. In the case of the ATFD metric, the minimal value used in the case study was set to 0, and the maximal value to 2886. The thresholds of the ATFD software metric were also incremented by 1 for the same reason as the WMC metric thresholds. Finally, the minimal value of the TCC metric threshold was set to 0, and the maximal to 1. Because the domain of TCC metric measurements is real numbers, the thresholds used in this research were incremented by value 0.02. This value was chosen as a compromise between the accuracy of the results and the acceptability of time complexity of the grid search.

Thresholds Limits			
Metric	MIN	MAX	Increment
ATFD	0	2886	1
TCC	0.0	1.0	0.02
WMC	0	1630	1

TABLE I

TABLE OF USED METRICS AND THEIR THRESHOLDS LIMITS

For each threshold combination used in the grid search, we tested its defect prediction performance. The goal of the experiment was to find a global maximum, i.e. thresholds that lead to the most accurate defect prediction model. After setting the thresholds of the God Class definition, we marked classes in the source code of the observed software pieces either as God Class, or as an ordinary class. With this step, we got two sets of classes, smelly classes, and regular ones. By using these two class sets we tested how accurately God Classes can predict faulty classes in the observed software. Both accuracy and F-score values were calculated using predefined algorithms from the Weka library[6].

On a computer system with Intel i7 - 4790 CPU, 24 GB of RAM, and exploiting all eight available CPU cores, the grid search took less than 30 hours to calculate accuracy and

[1]https://projects.eclipse.org/projects/eclipse.jdt/

[2]https://projects.eclipse.org/projects/eclipse.pde/

[3]https://www.eclipse.org/ide/

[4]https://projects.eclipse.org/

[5]https://github.com/pmd

[6]https://www.cs.waikato.ac.nz/ml/weka/

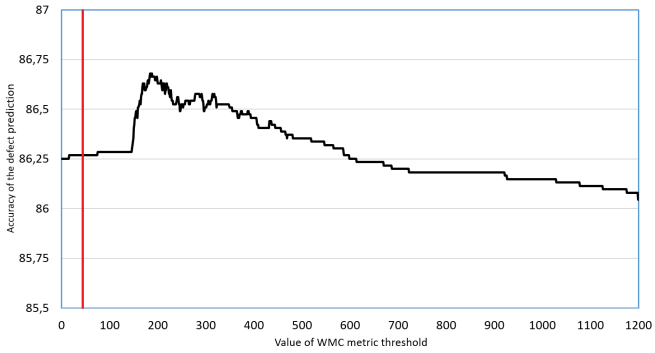

Fig. 2. Accuracy of Studied WMC Metric Thresholds

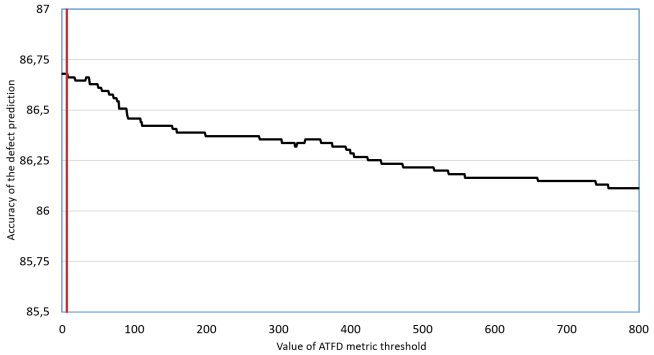

Fig. 3. Accuracy of Studied ATFD Metric Thresholds

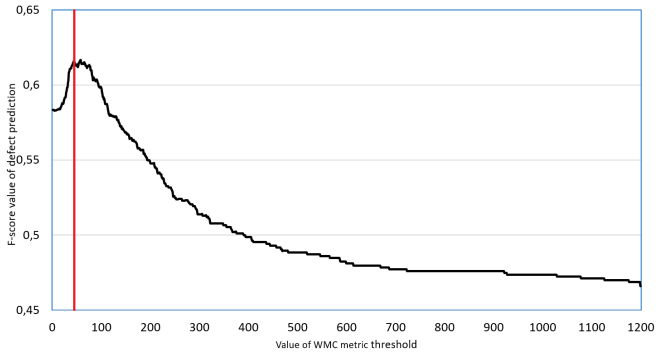

Fig. 5. F-score values of Studied WMC Metric Thresholds

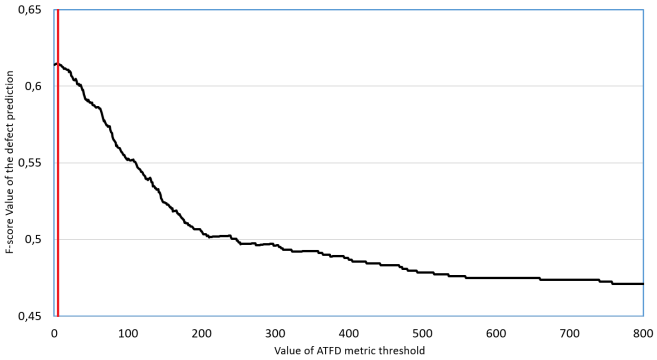

Fig. 6. F-score values of Studied ATFD Metric Thresholds

F-score values for all predicted thresholds of metrics used in the God Class definition.

D. Results of the case study

The results of both, accuracy and overall F-score are shown in the Figs. 2–7.

The output of the grid search algorithm was a dataset of data containing calculated values of accuracy and F-score parameters for each combination of thresholds of software metrics WMC, ATFD, and TCC within their limits, as defined in Table I. Consequently, the algorithm generated a vast amount of data, of which only a part is relevant for further analysis. To support further data analysis, we extracted thresholds of the studied metrics for which the maximum values of accuracy and F-score were reached. Based on the obtained maximum values of accuracy and F-score parameters, we could conclude whether better optimized software metric thresholds of God Class definition exists for purpose of the defect prediction in the projects in the experiment.

For the purpose of a deeper analysis of the behavior of the observed metric thresholds, we also extracted data of optimal accuracy and F-score values for each observed metric on the threshold interval. The analysis was performed for accuracy and F-score values separately by linear transition through the generated dataset. The data obtained in the analysis can be seen in Figs. 2-7. For clearer presentation, the interval of observed threshold interval is shortened, namely to an interval between 0 and 1200 for the WMC metric,

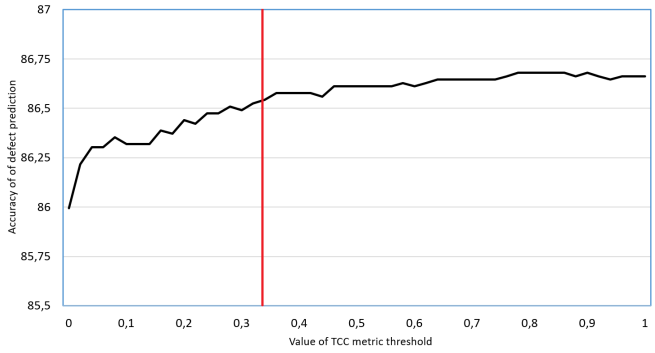

Fig. 4. Accuracy of Studied TCC Metric Thresholds

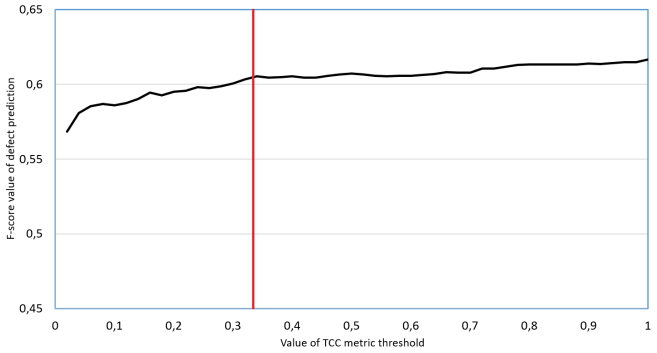

Fig. 7. F-score values of Studied TCC Metric Thresholds

and 0 and 800 for the ATFD metric. The charts in Figs. 5 and 6 for the TCC metric show the whole interval of analyzed threshold values. By shortening the interval, we could visualize important behavior at the beginning of the interval better. After the point of the cut of intervals in the charts, only a slight fall of accuracy and F-score value can be detected. The red line on all of the Figs. shows the default software metric values that are used in the definition of God Class proposed by Lanza and Marinescu.

IV. DISCUSSION

On the basis of the analysis of the data obtained by the grid search, we could answer our initial research question. The data analysis shows that the optimal thresholds of God Class definition metrics are relatively close to the values of metrics' thresholds proposed by Lanza and Marinescu [10]. Nevertheless, in the context of the studied software projects, we were able to find a more suitable metric threshold of God Class definition. With the use of these thresholds, we were able to predict faulty classes based on the identification of God Classes in the source code of the studied Eclipse JDT and Eclipse PDE projects more accurately compared to predictions made by the threshold of God Class definition metrics proposed by Lanza and Marinescu.

A. Global optimum

From the data collected by analyzing the Eclipse JDT and Eclipse PDE projects, we can discern clearly that the optimal metric thresholds of God Class definition used for the defect prediction in the observed projects is 58 for metric WMC, 5 for the ATFD metric, and 1 for the TCC metric. So, for the observed projects, the God Class definition predicts defects optimally by using the stated metrics' threshold values. The measured value of accuracy for optimum thresholds is 82.904, the measured value of F-score using the same optimal thresholds is 0.617. In comparison to the results calculated by the defect prediction using God Class definition with default threshold proposed by Lanza and Marinescu, accuracy and F-score values are slightly lower. Namely, the WMC threshold value was 47, the ATFD threshold value was 5, and the TCC threshold value was 1/3 of the reached value of accuracy, which are 81.256 and F-score 0.601. The maximum and minimum values of calculated accuracy and F-score for observed metrics is depicted in Table II. The defect prediction using the optimal threshold obtained by our grid search implementation is 1.65 pp more accurate compared to the defect prediction that uses the default threshold for God Class definition. The defect prediction performed on the studied projects using the optimal thresholds reaches a better F-score value, namely the value 0.015.

B. Thresholds of software metrics

Analysis of the deviations of the optimal thresholds of God Class definition obtained by grid search in our case study with the metrics threshold proposed by Lanza and Marinescu shows interesting differences in threshold values. The ATFD metric, our threshold matched completely with those in the generally accepted definition of God Class. Similarly, the calculated values of the WMC metric threshold were also quite close to the generally accepted values. On the contrary, in the case of the TCC metric, the optimal threshold deviates more distinctly compared to the recommended value proposed by Lanza and Marinescu.

Software Metric	Accuracy		F-score	
	MIN	MAX	MIN	MAX
ATFD	86.011	86.680	0.464	0.615
TCC	85.994	86.680	0.580	0.617
WMC	86.011	86.680	0.464	0.617

TABLE II
MINIMAL AND MAXIMAL VALUES OF ACCURACY AND F-SCORE VALUES

Figure 5 shows the chart of calculated F-score values for each value of WMC metric on the interval between 0 and 1200. In the analysis, we relied mainly on the F-score value, which represents the effectiveness of the defects' prediction algorithm balanced for both F and NF classes. The chart shows the efficiency of defect prediction of the God Class code smell by a different threshold of the WMC metric. The optimum of F-score value for the WMC metric was reached at value 58, which is relatively close to value 47 (the value is marked with the vertical line in Fig. 5), proposed by Lanza and Marinescu. Both thresholds, the threshold obtained by the grid search in our research and the generally accepted one, lie on the plateau on the interval 32 to 96, on which the defect prediction algorithm is, in the case of the studied software pieces, the most efficient, as it exceeds the F-score value of 0.6. The chart in Fig. 2 shows calculated values of accuracy value for each value of WMC metric on the interval between 0 and 1200. From the chart, it can be seen that the studied prediction algorithm is most effective by the threshold of WMC metrics at value of 200, and, after that, the accuracy is falling slowly and evenly.

The chart in Figure 6 depicts calculated F-score values for each value of the ATFD metric threshold on the interval between 0 and 800. In the chart, the depicted F-score values, which represent the effectiveness of the defects prediction algorithm balanced for both F and NF classes, reached the optimum by the value 5. The optimal value of the F-score parameter calculated by the grid search algorithm in our research completely overlaps with the ATFD metrics threshold proposed by Lanza and Marinescu. In both cases, the metric value 5 is the optimal threshold in the context of use for software fault prediction on studied pieces of software. Analysis of charts for both accuracy (Fig. 3) and F-score (Fig. 6) values of the ATFD metric, shows that software defects based on God Classes are the best predicted at low metric values. After the optimum threshold at value 5, the accuracy is falling slowly and almost evenly, while more and more relevant God Classes are left out due to the threshold of the ATFD metric being set too high.

Analysis of F-score values of the TCC metric thresholds depicted in Fig. 7 shows that the optimal value of the F-score parameter obtained by the grid search is reached at the metric

threshold value 1. The accuracy of software defect prediction of the studied Eclipse JDT and Eclipse PDE, reached the value of 0.6 at the TCC metric value of 0.33, which is in line with the thresholds in the God Class definition proposed by Lanza and Marinescu. After that point, the F-score value for the TCC metric continues to rise continuously until it reaches its optimum at value 1. From the perspective of the studied projects Eclipse JDT and Eclipse PDE, the definition of God Class with values of metric TCC around the value 1 is slightly more accurate when used in software defect prediction. Similar observations can be obtained by observing the behavior of the accuracy parameter for the TCC metric in Fig. 4. The behavior observation of accuracy and F-score values for the TCC metric raise an additional question about the contribution of the TCC metric for defect prediction. Namely, when the thresholds of the TCC metric are rising, fewer classes are marked as a God Class. When the threshold at value 1 is reached, none of the classes in the source code of observed projects is marked as a God Class. This observation demands further investigation.

V. CONCLUSION

Our case study is a preliminary case study, whose goal was to test whether there is any room for optimization of the metric thresholds of God Class definition proposed by Lanza and Marinescu. In our research, we were interested in the optimization of the God Class definition from the perspective of its use in the process of defect prediction. The data analysis shows that the optimal metric thresholds of God Class definition obtained by the grid search predicts software defects in projects under the study slightly better compared to the metric thresholds of God Class definition proposed by Lanza and Marinescu. The predictive performance of the threshold obtained by our approach is better, both from the perspective of accuracy by 1.65 pp, and by 0.0153 from the perspective of the F-score. As this is only a preliminary study, we tested our research question on a limited number of projects, namely Eclipse JDT and Eclipse PDE. The both projects are also based on java technology stack and are developed within the same open source community. Therefore, our project selection can induce potential threats to validity of our research. In order to be able to derive more reliable results, in the future, more projects written in different programming languages should be included in the replication of the case study. Our study opens new possibilities of adjusting and optimizing the process of defect prediction using approaches that are, compared to the used grid search, more time efficient. In the future, we plan to extend our research with one of the Machine Learning approaches. Further, we plan to extend our study to other types of code smells.

ACKNOWLEDGMENT

The authors acknowledge the project (An empirical comparison of machine learning based approaches for code smell detection, BI-HR/18-19-036) which was supported financially by the Slovenian Research Agency and by Croatian Ministry of Science and Education.

REFERENCES

[1] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[2] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, 2010.

[3] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] M. Hozano, A. Garcia, B. Fonseca, and E. Costa, "Are you smelling it? Investigating how similar developers detect code smells," *Information and Software Technology*, 2017.

[6] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[7] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 2015.

[8] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance*, ser. ICSM 2003, 2003.

[9] 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*, vol. 5, no. 1, Oct 2017.

[10] 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-Verlag Berlin Heidelberg, 2006.

[11] 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," in *34th International Conference of the Chilean Computer Science Society*, ser. SCCC 2015, 2015.

[12] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *IEEE International Conference on Software Maintenance*, 2007.

[13] PMD, "An extensible cross-language static code analyzer," https://pmd.github.io/, 2019.

[14] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in in the post-release object-oriented system evolution," *The Journal of Systems and Software*, 2007.

[15] S. M. Olbrich, D. S. Cruzes, and D. I. Sjberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance*, 2010.

[16] R. Marinescu and C. Marinescu, "Are the clients of flawed classes (also) defect prone?" in *IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM 2011, 2011.

[17] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Are you smelling it? Investigating how similar developers detect code smells," *Software Quality Journal*, pp. 403–426, 2014.

[18] M. Sokolova and G. Lapalme, "Performance measures in classification of human communications," in *Advances in Artificial Intelligence*. Springer, 2007, pp. 159–170.

[19] N. Japkowicz and M. Shah, *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[20] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *Journal of Machine Learning Technologies*, 2011.

[21] G. Mauša, T. Galinac Grbac, and B. Dalbelo Bašić, "A systemathic data collection procedure for software defect prediction," *Computer Science and Information Systems*, vol. 13, no. 1, pp. 173–197, 2016.