Code Smell Aware – Issue Prediction Model

S.Narasimhulu, CH.Lawrence Dheeraj, Dr.Madhu B.K

***Abstract*— Issues are the reasons for poor design. Previously we assess the impact of smells on code quality and it indicates their harmful impact on maintainability. In this paper we collect previous detections on issue-proneness to construct a specialized issue prediction model for code smell classes. Mainly focus on the involvement of a measure the severity of the code smells by adding it to the existing issue prediction model product based process based metrics, and comparing the results of the new model. The proposed model with the one of alternative approach which impacts metrics about the previous data of code smells in files. Identify that one proposed works usually better. However we observed the complementarities between the set of issues and smelly classes properly classified by the two models. On the basis of this result we assess a smell aware combined issue prediction model. We make obvious how such model classifies issue-prone code components with the harmonic mean of precision and recall.**

***Index Terms*— issue, smelly classes, poor design, metrics, severity**

I. INTRODUCTION

In the real-world situation, software systems change every day to be adapted to new requirements or to be fixed with regard to discovered ISSUES. They require of gathering strict deadlines does not always allow developers to manage the difficulty of such changes in an effective way. Indeed, often the development actions are performed in an disruptive manner, and have the effect to corrode the original blueprint of the system by introducing technical debts This incident is extensively known as software aging Some researchers deliberate the observable fact in terms of entropy while others defined the supposed bad code smells (shortly “code smells” or simply “smells”), i.e., recurring cases of poor blueprint choices occurring as a outcome of aging, or when the software is not properly designed from the beginning. Long or complex classes (e.g.,Blob), poorly structured code (e.g., Spaghetti Code), or long Message Chains used to develop a certain feature are only few examples of code smells that can probably affect a software system .

In addition approaches for the automatic detection of code smells in source code the research community committed a lot of effort in studying the code smell lifecycle as well as in given that evidence of the harmful effects of the presence of design flaws on Non-functional attributes of the source code. On the one hand, empirical studies have been conducted to understand when and why code smells come out what their evolution is and longevity in software projects, and to what extent they are relevant for programmers. On the other hand, a number of studies showed the harmful effects of code smells on software understand capability and maintainability Recently, Khomh et al. and Palomba et al. Have also empirically established that classes affected by design problems are more level to contain issues in the future. Even though this study showed the potential importance of code smells in the context of issue prediction, these observations have been only incompletely explored by the research community. A prior work by Taba et al. defined the first issue prediction model that includes code smell information. In exacting, they defined three metrics, coined as antipattern metrics, based on the history

Of code smells in files and able to quantify the average number of antipatterns, the complexity of changes involving antipatterns and their recurrence length. Then, a issue prediction model exploiting antipattern measures in addition structural metrics was devised and evaluated, showing that the performances of issue prediction models can increase up to 12.5% when considering design flaws. In our beginning study, we conjectured that taking into account the harshness of a design problem affecting a source code element in a issue prediction model can

1) We expand the empirical validation of the smell intensity-including (from now on, simply intensity including) issue prediction model by allowing for a set of 45 releases of 14 software projects. This allows to significantly increasing the generalizability of the achieved outcome.
2) In addition evaluating the involvement of the intensity index in the context of a structural-based issue prediction model, we expand our study to consider three more baseline models, all of them relying on process metrics. Particularly, we tested the involvement of the intensity index in the Basic Code Change Model devised by Hassan, the programmer-based Model proposed by Ostrand et al., and the Developer Changes Based Model defined by Di Nucci et al.
3) We execute an empirical comparison of the performances achieved by our model and by the model suggested by Taba et al.
4) We devise and discuss the results of a smell-aware issue prediction model, built by combining product, process, and smell-related information.
5) We provide a comprehensive duplication package including all the raw data and operational data sets of our study.

II. RELATED WORK

Although the main contribution of this paper spans in the field of issue prediction, the work is built upon previous knowledge in the field of bad code smell detection and management. For this reason, in this Section we provide an overview of the related literature in the context of both issue prediction and code smells

S.Narasimhulu,Assistant Professor,Dept of CSE,S.V College ofEngineering,Tirupati,AP,India,9550891430
CH.Lawrence Dheeraj,Assistant Professor,Dept of CSE,S.V College ofEngineering,Tirupati,AP,India,9121074996
Dr.Madhu B.K,Professor,Deptof CSE, ,Vidya vikas Institute of Engineering & Technology,Mysore,India,9742696669

III. A SPECIALIZED ISSUE PREDICTION MODEL FOR SMELLY CLASSES

Previous work has proposed the use of structural quality metrics to predict the issue-proneness of code com-ponents. The underlying idea behind these prediction models is that
The presence of issues can be predicted by analyzing the quality of source code. However, none of them take into account the presence and the severity of well-known indicators of design flaws, i.e., code smells, affecting the Source code. In this paper, we explicitly consider this information. Indeed, we believe that a more clear description and characterization of the severity of design problems affecting a source code instance can help a machine learner in distinguishing those compo-nents having higher Probability to be subject of issues in the future. To this aim, once the set of code compo-nents affected by code smells have been detected, we build a prediction model that, in addition to relying on structural metrics, also includes the information about the severity of design problems computed using the intensity index defined by Arcelli Fontana et al.

Specifically, the index is computed by JCodeOdor1, a code smell detector which relies on detection strategies ap-plied on metrics. The tool is able to detect, filter and prioritize instances of six types of code smells

God Class: A large class implementing different responsibilities;

Data Class: A class whose only purpose is holding data;

Brain Method: A large method that implements more than one function;

Shotgun Surgery: A class where every change trig-gers many little changes to several other classes;

Dispersed Coupling: A class having too many relationships with other classes;

Message Chains: A method containing a long chain of method calls.

The intensity index is an estimation of the severity of a code smell, and its value is defined as a real number in the range [1,10]. In particular, given the set of classes composing the software system that a developer wants to evaluate, JCode Odor adopts the following two steps to compute the intensity of code smells In the first step the tool aims at detecting code smells in the system given as input, relying on the detection strategies reported in Table 1. Each detection strategy is a logical composition of pred-icates, and each predicate is based on an operator that compares a metric with a threshold Our detection strategies are similar to those defined by Lanza and Marinescu, which adopted the metrics reported in Table 2 to detect the six code smells described above. More specifically, Lanza and Marinescu observed that code smells often exhibit

(i) low cohesion and high coupling, (ii) high complexity, and (iii) extensive access to the data of foreign classes: for this reason, our approach considers (i) cohesion (i.e., TCC) and coupling (i.e., CC, CDISP, CINT, CM, FANOUT), (ii) complexity (i.e., CYCLO, MaMCL, MAXNESTING, MeMCL, NMCS, WMCNAMM, WOC), and (iii) data access (i.e., ATFD and ATLD) metrics. Furthermore, the approach also computes size-related metrics such as LOC, LOCNAMM, NOAM, NOLV, NOMNAMM, and NOPA. To ease the comprehension of the detection approach, Table 2 reports the full metric names and definitions, while Table 3 describes the rationale behind the use of each predicate of the detection strategies. Moreover, in Table 4 we provide data on the distribution of the metrics used for code smell detection on the dataset exploited in this paper (more details on the systems and their selection are provided in Section 4).

Following the detection rules, a code component is detected as smelly if one of the logical propositions shown in Table 1 is true, namely if the actual metrics of the code component exceed the threshold values composing a detection strategy. It is important to note that the thresholds used by the tool have been experientially calibrated on 74 systems of the Qualitas Corpus dataset and are derived from the sta-tistical distribution of the metrics contained in the dataset . For metrics representing ratios defined in the range [0,1] (e.g., the Tight Class Cohesion), threshold values are fixed to 0.25, 0.33, 0.5, 0.66 and 0.75. For all other metrics, they are associated to percentile values on the metric distribution. For sake of completeness, we report in Table 5 all the threshold values associated to each of the detected code smells. The thresholds are also mapped by the tool onto a nominal value, i.e., VERY-LOW, LOW, MEAN, HIGH, VERY-HIGH, to ease their interpre-tation.

If a code component is detected as a code smell, the actual value of a given metric used for the detection will exceed the threshold value, and it will correspond to a percentile value on the met-ric distribution placed between the threshold and the maximum observed value of the metric in the system under analysis. The placement of the actual metric value in that range represents the “exceeding amount” of a metric with respect to the defined Threshold.
. To compute z, i.e., the normalized value, the following formula is applied:

$$z = \left[\frac{x - \min(x)}{\max(x) - \min(x)} \right] \frac{10}{} \quad (1)$$

Where min and max are the minimum and maxi-mum values observed in the distribution. This step allows having the “exceeding amount” of each metric in the same scale. To have a unique value representing the intensity of the code smell affecting the class, the mean of the normalized “exceeding amounts” is computed.

TABLE 2: Metrics used for Code Smells Detection

Short Name	Long Name
ATFD	Access To Foreign Data
ATLD	Access To Local Data
CC	Changing Classes
CDISP	Coupling Dispersion
CINT	Coupling Intensity
CM	Changing Methods
CYCLO	McCabe Cyclomatic Complexity
FANOUT	
LOC	Lines Of Code
LOCNAMM	Lines of Code Without Accessor or
	Mutator Methods
MaMCL	Maximum Message Chain Length
MAXNESTING	Maximum Nesting Level
MeMCL	Mean Message Chain Length
NMCS	Number of Message Chain Statements
NOAM	Number Of Accessor Methods
NOLV	Number Of Local Variables
NOMNAM	Number of Not Access or Mutator
	Methods
NOPA	Number Of Public Attributes
TCC	Tight Class Cohesion

IV. Prediction Model Construction

To answer our research questions, we needed to instant-ate the prediction model presented in Section 3 to define the basic predictors, (ii) the code smell detection process, and (iii) the machine learning technique to use for classifying issuegy instances

A. *basic predictors*

To this aim, we firstly set up a issue prediction model composed of structural predictors, and in particular the 20 quality metrics exploited by Jureczko et al. [30]. The model is characterized by a mix of size metrics (e.g., Lines of Code), coupling metrics (e.g., Coupling between Object Classes, cohesion metrics (e.g., Lack of Co-hesion of Methods, and complexity metrics (e.g., McCabe Complexity. In this case, the choice of the baseline was guided by the will to investigate whether the use of a single additional structural metric representing the intensity of code smells is able to add useful information in a prediction model already characterized by structural predictors, as well as by the set of code metrics used for the computation of the intensity index. It is important to note that this model might be affected by multi-co linearity, which occurs when two or more independent variables are highly correlated and can be predicted one from the other. Recent work,

B. *Code Smell Prediction*

The Bayesian technique proposed by Khomh et al. assigns a probability that a certain class is affected by the God Class code smell, while it has not been defined for other smell types. For this reason, we relied on the detection performed by JCodeOdor because

(i) It has been experientially validated demonstrating good performances in detecting code smells
(ii) It detects all the code smells considered in the EXPERIENTIAL study. Finally, it computes the value of the intensity index on the detected code smells

V. INVESTIGATION OF THE RESULTS

In the following we discuss the results, aiming at pro-viding an answer to our research questions. To avoid redundancies, we discuss the first two research questions together.

A. *The performances of the projected model*

Before describing the results related to the addition of the intensity index in the different prediction models considered, it is worth reporting the output of the feature selection process aimed at avoiding multi-collinearity by removing irrelevant features from the structural model. In particular, for each considered project we discovered a recurrent pattern in the pairs of metrics highly corre-lated:

1) Weighted Method per Class (WMC) and Response for a Class (RFC);
2) Coupling Between Objects (CBO) and Afferent Cou-plings (CA);
3) Lack of Cohesion of Methods (LCOM) and Lack of Cohesion of Methods 3 (LCOM3);
4) Maximum Cyclomatic Complexity (MAX(CC)) and Average Cyclomatic Complexity (AVG(CC));

According to the results achieved using the vif func-tion , we removed the RFC, CA, LCOM, and MAX(CC) metrics. Therefore, the resulting structural model is composed of 16 metrics.

VI. THREATS TO VALIDITY

Threats to construct validity are related to the relationship between theory and observation. Above all, we relied on JCode Odor for detecting code smells.
The intensity index computed by the tool derives by a set of code metrics characterizing cohesion, coupling, complexity, size, and data access of classes. A first problem threatening our observations might be the redundancy of such metrics. To verify the validity of the intensity computation

VII. CONCLUSION AND FUTURE WORK

In this paper, we evaluated to what level the addition of the potency key in existing state-of-the-art issue prediction models is useful to increase the performances of the baseline models. Specifically, we firstly set up four baseline prediction models then, we compared the performances of such models with and without the addition of the potency key, in order to control the actual contribution of the severity of code smells. Moreover, we also compared the models mentioned above with the same baseline models.
When compared with the models built using the antipattern metrics, we observed that the models including the obtain an accuracy up to 16% higher. More notably, we observed interesting com-plementarities between the set of issuegy and smelly classes correctly classified by the two different configurations of models.
In the second step of our analyses, we quantified the actual gain provided by the potency key with respect to the other metrics composing the models, confirming the high predictive power of the potency key over all the baseline models.
Based on these results, we built a smell-aware predic-tion model which combines product, process, and smell-related information. The performances of this new model outperform the ones of all the other experimented mod-els, confirming

once again the usefulness of considering code smells in issue prediction.

The potency key helps selective issue-prone code elements affected by code smells in issue

Prediction models based on product metrics, process metrics, and a combination of the two. Our results also suggest that the potency of code smells is helpful in all these cases, and cannot be replaced by a simple indicator of the presence or absence of a code smell. the practical applicability of the proposed smell-aware issue prediction model.

As for future work, we firstly plan to further analyze how the potency key impacts the performances of issue prediction models, by performing a fine-grained analysis into the role of each smell type independently on the pre-diction power. Furthermore, since our study has focused on global issue prediction, future effort will be devoted to the analysis of the contribution of smell-related information in the context of local-learning issue prediction models

REFERENCES

[1] Fabio Palomba Marco Zanoni,, Francesca Arcelli Fontana Andrea De Lucia Rocco Oliveto Toward a Smell-aware Issue Prediction Model-IEEE Transactions on software engineering vol 45 no.2 Feb,2019.

[2] W. Cunningham, "The WyCash portfolio management system," OOPS Messenger, vol. 4, no. 2, pp. 29–30, 1993.

[3] D. L. Parnas, "Software aging," in Proceedings of the 16th Inter-national Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994., 1994, pp. 279–287.

[4] W. Harrison, "An entropy-based measure of software complex-ity," IEEE Trans. Softw. Eng., vol. 18, no. 11, pp. 1025–1029, Nov. 1992.[Online].Available:http://dx.doi.org/10.1109/32.177371

[5] A. E. Hassan, "Predicting faults using the complexity of code changes," in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, May 2009, pp. 78–88.

[6] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley, 1999.

[7] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in Software Maintenance (ICSM), 2013 29th IEEE International Conference on, Sept 2013, pp. 396–399.

[8] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lu-cia, "Methodbook: Recommending move method refactorings via relational topic models," IEEE Transactions on Software En-gineering, vol. 40, no. 7, pp. 671–694, July 2014.

[9] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, 2010.

[10] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lu-cia, "A textual-based technique for smell detection," in Proceed-ings of the 24th International Conference on Program Comprehension (ICPC 2016). Austin, USA: IEEE, 2016, p. to appear.

[11] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 462–489, May 2015.

[12] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347–367, 2009.

[13] 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 (and whether the smells go away)," IEEE Transactions on Software Engineering, p. to appear., 2017.

[14] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in Proceedings of the International Workshop on Refactoring Tools. ACM, 2011, pp. 33–36.

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in Int'l Conf. Quality of Information and Communications Technology (QUATIC). IEEE, 2010, pp. 106–115.

[16] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in Proceedings of the International workshop on Principles of Software Evolution (IWPSE). ACM, 2007, pp. 31–34.

[17] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR). IEEE, 2004, pp. 223–232.

[18] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2014, pp. 101–110.

[19] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in Proceedings of the Working Conference on Reverse Engineering (WCRE). IEEE, 2013, pp. 242–251.

[20] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An EXPERIENTIAL study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany. IEEE Computer Society, 2011, pp. 181–190.

[21] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," IEEE Trans. Software Eng., vol. 39, no. 8, pp. 1144–1156, 2013.

[22] A. F. Yamashita and L. Moonen, "Do code smells reflect impor-tant maintainability aspects?" in Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, 2012, pp. 306–315.

[23] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An EXPERIENTIAL study," in Proceedings of the International Conference on Software Engineer-ing (ICSE). IEEE, 2013, pp. 682–691.

[24] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in Proceedings of the Working Conference on Reverse Engineering (WCRE). IEEE, 2009, pp. 75–84.

[25] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," EXPERIENTIAL Software Engineering, vol. 17, no. 3, pp. 243–275, 2012.

S.Narasimhulu,Assistant Professor, DeDept of CSE, Sri Venkateswara college of Engineering, Tirupati, AP, India. His Interesting area is Software Engineering, Software Testing

CH.Lawrence Dheeraj, Assistant Pr Professor, Dept of CSE, Sri Venkateswara college of Engineering, Tirupati, AP, India. Hi His Interesting area is Software Engineering, Software Testing

Dr.Madhu B.K, Professor, Dept of CSE, Vidya Vikas Institute of Engineering and Technology, M Mysore, India. Member in ACM, ISTE, CSI. His Research area In Software Engineering and Software T Testing.