Impact Of Code Smells On Software Fault Prediction At Class Level And Method Level

Um-E-Safia
National University of Technology, Islamabad
Islamabad, pakistan
umesafia@nutech.edu.pk

Tamim Ahmed Khan
Bahria University, Islamabad
Islamabad, Pakistan
tamim@bahria.edu.pk

2022 International Conference on Frontiers of Information Technology (FIT) | 979-8-3503-4593-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/FIT57066.2022.00066

***Abstract*—The main aim of software fault prediction is the identification of such classes and methods where faults are expecting at an early stage using some properties of the project. Early-stage prediction of software faults supports software quality assurance activities. Evaluation of code smells for anticipating software faults is basic to ensure its importance in the field of software quality. In this paper, we investigate the impact of code smells on software fault prediction at the class level and method level. Previous studies show the impact of code smells on fault prediction. However, using code smells for class level faults prediction and method level fault prediction need more concern. We use defects4j repository for the creation of datasets used in building software fault prediction model-based. We use pseudo labeling for class level prediction and bagging for method level prediction. We extract code smells from different classes and methods and then used these extracted code smells for fault prediction. We compare our prediction results with actual results and see if our prediction is correct in order to do validation.**

***Index Terms*—Code smells, Software fault prediction, Machine learning**

I. Introduction

When we talk about prediction in software engineering, it contains several approaches such as fault prediction, the effort for testing prediction, cost prediction, quality prediction, modules reusability prediction, and software security prediction. Every approach is important. However, many approaches are still in research phase because of the insufficient work done in these fields. Among all these prediction approaches, the most popular prediction approach in the research area is software fault prediction.

The role software fault prediction play is that they help in improving the quality of software and assist inspection by locating possible faults in software. Efforts are necessary to minimize software faults. However, all these efforts cost time and resources. Early fault prediction strategy is required so that it helps in the reduction of faults. It is verified that the sooner a fault is detected lesser it costs [1].

Code smells are defined as properties of source code that indicate expected faults or deeper problems [1]. At first 21 different types of code, smells were introduced [3]. Code smells are now an accepted concept that is used to refer to such design aspects and patterns, which may cause problems at later stages of software systems like development and maintenance [3], [4]. Regardless, code smells are not incorrect but instead, their essence point towards instability in design, which results in failure of the system and expected bugs in future.

in the field of Software Fault Prdiction majority of existing work used Object-Oriented and Process metrics. Commonly used metrics are "Halstead's software metrics" [5], "McCabe Cyclomatic complexity metrics" [6], and "Object-Oriented metrics" [7]–[10]. Many different modeling techniques used to predict faults. Commonly used techniques are Statistical modeling technique (univariate or multivariate logistic regression) [11]. And machine learning techniques [12]–[14].

Conventional bug prediction approaches that use above mention metrics for prediction have certain issues. For instance, it is more than obvious that if a class has an enormous line of codes, it is more error-prone. Yet this is not proved that a class or method having less line of codes has a smaller number of bugs. In this manner, some other metrics for fault prediction should use in research.

Some of the previous studies showed a significant effect of code smells for fault prediction. In literature, different types of code smell metrics have proposed and used for the fault prediction models. These metrics help in the construction of the prediction model. Fault prediction models used data gathered from such projects where faults have been identified previously [15].

However, the result of these approaches is predicting faults at class level or file level. This approach often put a considerable amount of effort on tester's shoulder to examine all classes or file until the bug is located. This approach strengthens the verity that files having many lines of code predicted as faulty.

In previous studies faults predicted at class level, no work is done for fault prediction at method level using code smells. this is a fact that faults predicted at the method level rather than at class level or file level can save time and cost. Moreover, predicting faults at method level reduce inspection effort for developer and testers. In our study, we present a methodology that will predict faults not only at class level but also at method level using code smells.

II. Related work

Initially, 21 different types of code smells were introduced [3]. A study describes the Domain Specific tailoring of Code Smell. They think the heuristics of code smell [3] is a little

979-8-3503-4593-3/22/$31.00 ©2022 IEEE
DOI 10.1109/FIT57066.2022.00066

TABLE I: List of selected code smells [3], [31]

God class	It refers to such a class that perform too much functionality and have huge lines of codes.
Shotgun surgery	It is code duplication and refer to when single change made in multiple classes.
Feature envy	It refers a method that access data of other object more than its own data.
Brain class	A class that performs too much but have strong cohesion.
Tradition breaker	Sub class should provide methods and functionality that is not related to its super class.
Brain method	A method that performs too much but have strong cohesion.
Extensive coupling	A method that communicates too much with other methods, but provider method dispersed in many classes.
Parent bequest	Refer to when child class uses only few methods inhered from parent class.
Intensive coupling	Refer to when methods are bind to each other, but provider method is only dispersed in few classes.
Long parameter list	Any method in class having more than 3 parameters.
Schizophrenic Class	When 2 or more key abstraction are captured in a single class.
Data class	Refer to when child class uses only few methods inhered from parent class.

broad, they tailor the heuristics of domain-specific of code smell to make the heuristics fit the specific domain [16].

Recently, more considerations pulled into exploring that how faults and code smells are related to each other. A study proved by giving empirical evidence that code smells are helpful in fault prediction. They used source code metrics and code smells metrics in their study and used Na¨ıve Bayes, random forest and logistic Regression techniques [17]. In a study author used code smells and community smells and compare their results for fault prediction. The results showed that community smells improve prediction model performance up to 3% in terms of AUC. While code smells improve prediction model performance up to 17% in terms of AUC [18].

In another study, the author assesses the benefaction of a proportion of the severity of code smells by adding it to existing bug prediction models dependent on both product and process metrics and looking at the consequences of the new model against the base models. They used Naive Bayes, Logistic Regression, and Decision tree techniques. As of result of this experiment by adding code smells a predictor, the accuracy of the prediction model increases [19]. In another study, the author study code smells in web applications. He extracts PHP code smells and uses them in his study. The results of this study show that code smells can help in fault prediction and it helps developers to identify faults and plan projects accordingly [20]. In another study author select 3 projects BIRT, Aspect J, and SWT. Extract code smells from them and studies how code smells are associated with bugs. His study shows that code smells and fault have a strong correlation. Lazy class, complex class, message chain, and long method have a strong correlation with faults [21].

The author in another study evaluates class level change-prone prediction power using code smells. He used Na¨ıve Bayes, Multilayer perceptron, LogitBoost, and Decision tree techniques in their study. The result of this experiment indicates that Code smell can predict class change proneness with a probability 70% superior to the existing prediction model [22]. In this study, the author proposed a model in which he used code smells from literature and designate smells. He used 97 different real projects. The results of this study showed that the model improves 5% in terms of AUC. He concludes that designated smells are a good addition and they help with code smells in prediction [23].

Another study investigates the impact of code smells and fault prediction. They used many techniques like ADTree, Na¨ıve Bayes, Logistic regression, and Multilayer perceptron. Results of this study showed that a combined model having code smells as metrics improves F-measure up to 20% [24]. Industrial system and 6 different code smell used to study the relationship between faults and code smells. It is found because of this study that "Shotgun Surgery" presence identifies with a factually critical higher likelihood of faults [26]. But, another author found in his study that there is no relation between faults and code smells and code smell does not affect the presence of faults [27]. So, in literature, the relationship between faults and code smells has not come to an agreement. A tool was built that is used to rank code smells according to severity based on 3 standards: "past component modification, important modifiability situations for the system, and importance of the sort of smell" [28].

In previous work where code smells were used for software fault prediction, the researcher creates their datasets by using CK and some other object-oriented and process metrics for code smells presence. Moreover, researchers have worked only with 4-8 types of code smells for fault prediction. However, fault prediction using code smells considering more types, needs more concern and we intend to find out if code smell would be beneficial or not.

To the best of our knowledge, all work done in the field of software fault prediction at the method level used process metrics, change metrics, and object-oriented metrics. No work is done for fault prediction at the method level using code smells. Thus, a method that can predict software faults at the method level using code smells is exceptionally wanted.

III. Methodology

The applied methodology consists of 3 parts, Preprocessing Phase, Model Development Phase, and Post-processing Phases as shown in figure 1. The main focus of the methodology is to create such a software fault prediction model using code smells (table 1) that not only predict faults at class level but also at method level. First, we clean data to remove unused code smells. We merge datasets into one after removing

unused code smells. This dataset is used to train and test the classifier. Performance evaluation metrics are used to evaluate the performance of classifier. After that we extract code smells from the case study, input these code smells into the model and our model predict faulty and non-faulty instances.

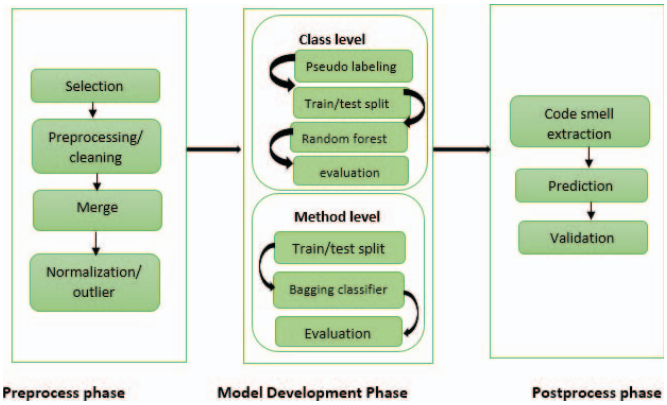


Fig. 1: Methodology

A. Preprocessing phase

The code smells we are using at the class level shown in table 1, The criteria of selecting these code smells are that these are important and most used basic code smells, tools are available to extract these code smells from the source code of the application and unlabeled dataset of these code smells are publicly available. then, in the 2nd phase, we do preprocess and cleaning of dataset. we preprocess and clean datasets to remove unused smells and to identify null values in our dataset. As all datasets have the same numbers of attributes so we merge them into one. We use discrete value of label bug, i.e., 1 and 0. 1 label depict faulty instance whereas, 0 label depict non- faulty instance.

B. Model Development phase

Second is Model development phase, we first split our clean and merge dataset into train and test split. After that we use supervised/semi-supervised learning technique to build our classifier. After building classifier, we use performance evaluation metrics and 10-folds cross validation to evaluate our classifier.

1) Class Level: We have a small amount of labeled dataset extracted from projects available at defect4j [2] and a huge amount of unlabeled dataset [28] available. To use both types of datasets in training classifiers we use semi-supervised machine learning technique named pseudo-labeling [33]. The pseudo-labeling technique utilizes a little arrangement of labeled data alongside a lot of unlabeled data to improve a model's exhibition. It first train model of labeled data. second, use that trained model to predict labels of unlabeled data. third, concatenate labeled data and pseudo-labeled data and last, retrain model again with concatenated data. To train classifier we use random forest machine learning technique. previous studies showed that random forest techniques provide better results for fault prediction as compare to other techniques [29].

2) Method Level: We extract method level dataset from defects4j [2]. this dataset is in small amount so, to increase the amount of dataset we use a technique named Bagging. Bagging classifier is a technique that divide dataset into subsets and then fit base classifier on each subset of dataset. after, through voting or average method it aggregates the results and give final prediction. Bagging not only increase the amount of data but it also improves accuracy, loss, bias, and variance and improve the performance of classifier. As a base classifier we are using random forest tree because of its effective performance for software fault prediction. Later, we evaluated our classifier using performance evaluation metrics. accuracy, precision, recall and F1-Score are used, proposed in [25].

C. PostProcessing Phase

This phase includes code smells extraction, prediction, and validation. For code smells extraction, We used iPlasma tool [32] to extract code smells from source code. we input extracted code smells to classifier and our classifier predict faulty instances shown in Fig 2. The last phase is of validation phase, in this phase we have fault information of about 30 classes/methods of projects [2], and we compare our prediction results with actual results to check how accurately we predict faulty instances.

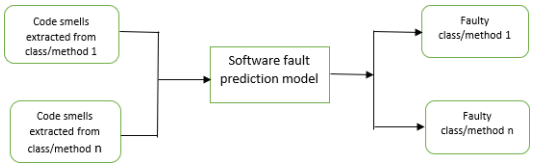


Fig. 2: Faults Prediction

IV. Implementation

We conduct experiment on defects4j projects. In order to perform experiments, we use Python for data cleansing, training and testing data, constructing models and for evaluation of final predictions.

A. For Class Level Prediction

We have 2 types of dataset primary (labeled) and secondary (unlabeled). The labeled dataset is extracted from defects4j [2]. defects4j is a repository having many java projects with complete bug information and location of bugs [30]. The unlabeled dataset is available at Qualitas Corpus [28]. The unlabeled dataset contains 21079 instances and labeled dataset contain 755 instances. We remove unused code smells from unlabeled dataset since our concern is to conduct experiment using 12 selected code smells shown in table 1. We use 0 and 1 as label for datasets. 0 is use for non-faulty and 1 is use for faulty. To predict pseudo labels for unlabeled dataset we first train our model using random forest machine learning technique. After, we concatenate both labeled and pseudo-labeled dataset and

retrain model using random forest machine learning technique. both Labeled and Unlabeled datasets combinedly have 21834 rows. In algorithn 1, we present Training Classifier (Class Level). We first select unlabeled datasets containing code smells (M= M1. Mm). we merge datasets into one. after that we select labeled dataset containing code smells. we merge datasets into one. Then we train algo using labeled dataset and predict pseudo-labels for unlabeled dataset. after predicting pseudo labels, we concatenate labeled and unlabeled datasets, split combined data into 70% train and 30% test data, and retrain model. Finally, We use accuracy, precision, recall and F-1 score to evaluate our classifier.

TABLE II: Performance Evaluation Table (Class)

Accuracy	Precision	Recall	F1 Score
99.78%	98%	97%	98%

For neutral results we use 10-fold cross validation method. We present accuracy, precision, recall and f1-scores's result of each fold in table 3.

TABLE III: 10-Folds Cross Validation (Class)

Folds	Accuracy	Precision	Recall	F1 Score
1	0.993	0.968	0.954	0.961
2	0.995	0.976	0.969	0.973
3	0.996	0.977	0.977	0.977
4	0.994	0.991	0.946	0.968
5	0.996	0.992	0.962	0.980
6	0.996	0.969	0.984	0.977
7	0.995	0.962	0.984	0.973
8	0.997	0.970	1	0.984
9	0.996	0.976	0.976	0.976
10	0.998	0.984	0.992	0.988

We use Random forest technique for the training of model. previous studies showed that random forest techniques provide better results for fault prediction as compare to other techniques. [29] we split the preprocessed dataset in train and test spilt. The ratio of train and test is 70: 30 (70 percent is of training and 20 percent is of testing). To select most optimal value for number of trees we calculate accuracy with different number of trees and select optimal number of trees Shown in Fig 3. after that, training split is used for training of data and then test split is used for testing and report precision, recall and F1 score. our results are displayed in Table 2.

For validation, we took a project Joda-time from defect4j [2]. Joda-time provides replacement for java date and time class. There is a total of 145 different classes. We extract code smells from the project using IPlasma [32]. We extract code smells of 30 classes. From 145 classes we picked only business logic classes. other classes that are platform depended are not considered. All fault reports of the project are with CSV (comma separated version) extension. code smells are extracted and become input for our fault prediction classifier. our classifier predict that among input which instance is faulty and which instance is non-faulty.

B. For Method level prediction

The labeled dataset for method level prediction is extracted from defects4j [2]. defects4j is a repository having many java projects with complete bug information and location of bugs [30]. We select 6 method level code smells to conduct our experiment shown in table 1. We use 0 and 1 as label for faulty and non-faulty data. 0 is use for non-faulty and 1 is use for faulty.we have 808 number of labeled instances. this labeled dataset is in small amount so, to increase the amount of dataset we use a Bagging. Bgging not only increase the amount of data but, it also improves accuracy, loss and bias and improve the performance of classifier shown in table 4. We create 10 bags each having 800 datapoints. After bagging our dataset is of 8000 instances. Because of its effective performance we use random forest as base classifier in bagging classifier [31]. In algorithm 2, we present Training Classifier (Method Level). First, we picked labeled datasets containing code smells. we merge datasets into one and removed unused code smells. after merge we split combined data into 70% train and 30% test data, and train model using classifier. Finally, we use accuracy, precision, recall and F-1 score to evaluate our classifier.

TABLE IV: Effect of Bagging on Bias and Loss

Classifier	Loss	Bias
Before Bagging	0.145	0.149
After Bagging	0.140	0.136

For neutral results we use 10-fold cross validation method. We present accuracy, precision, recall and f1-scores's result of each fold in table 6.

Algorithm 1 Training Supervised Model

1: **Input:** Labelled and Unlabelled datasets, having selected code smells ($M=M_1.....M_m$).
2: **Output:** Prediction Model with Performance Evaluation Metrics.
3: Take unlabeled datasets (D_{ul1}, D_{ul2}, D_{ul3} ..... ,D_{uln}.)
4: Merge into one dataset. Sum = $\Sigma_{K=1}\ D_{ulk}$
5: Remove unused code smells $M_C \leftarrow (M= M_1 ..... M_m)$
6: $D_{UL} \leftarrow$ apply data cleaning (D_{UL})
7: Gather labelled datasets (D_{l1}, D_{l2}, D_{l3}.....,D_{uln})
8: Merge into one dataset. Sum = $\Sigma_{K=1}\ D_{lk}$
9: $D_L \leftarrow$ apply data cleaning (D_L)
10: Train algo using labeled dataset D_L.
11: Predict pseudo labels for unlabeled datasets D_{UL}
12: Combine both datasets $D= D_L + D_{UL}$
13: Split the dataset into 70% train and 30% test ratio.
14: Use Random Forest Classifier for train and test of dataset with metric Mc.
15: Find Recall, Precision and F1-Score after prediction..

Random Forest technique is used to train classifier. previous studies showed that random forest techniques provide better results for fault prediction as compare to other techniques. [31]. the Cleane and preprocessed dataset is split into ratio of

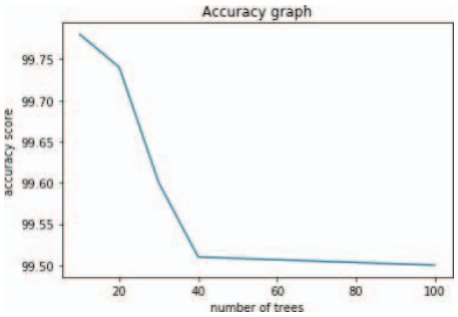


Fig. 3: Accuracy Graph (Class)

70:30 (70 percent is of training and 20 percent is of testing). To select most optimal value for number of trees we calculate accuracy with different number of trees and select optimal number of trees shown in Fig 4. after that, training split is used for training of data and then test split is used for testing and report precision, recall and F1 score. our results are displayed in Table 5.

For validation, we took a project Joda-time from defect4j [2]. Joda-time provides replacement for java date and time class. There is a total of 145 different classes. We extract code smells from the project using IPlasma [32]. We extract code smells of 30 methods of 30 classes. From 145 classes we picked only business logic classes. other classes that are platform depended are not considered. All fault reports of the project are with CSV (comma separated version) extension. code smells are extracted and become input for our fault prediction classifier. our classifier predict that among input which instance is faulty and which instance is non-faulty.

Algorithm 2 Training Supervised Model

1: **Input:** Labelled datasets, having selected code smells
2: **Output:** Prediction Model with Performance Evaluation Metrics(C_f)
3: Select labeled datasets (D_1, D_2, D_3 . . .,D_n)
4: //all have set of selcted code smells ($M= M_1$.....M_m)
5: Merge all datasets into one. Sum = $\sum_{K=1}^{n} D_k$
6: D ← Clean Dataset (D) by applying cleaing methods.
7: Split the dataset into 70% train and 30% test ratio.
8: Let N be the size of the training set.
9: for each of t iterations:
10: sample N instances with replacement from the original training set.
11: apply Bagging classifier (Cf) with random forest as base classifier on dataset (D) with code smells we have after step 3.
12: Find Recall, Precision and F1-Score after prediction.

V. Results

A. For Class level prediction

The case study we picked have 30 classes we have complete information of faulty classes of case study which shows that

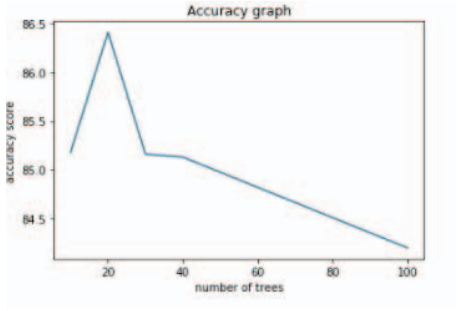


Fig. 4: Accuracy Graph (Method)

TABLE V: Performance Evaluation Table (Method)

Accuracy	Precision	Recall	F1 Score
86.41%	83%	80%	82%

TABLE VI: 10-Folds Cross Validation (Method)

Folds	Accuracy	Precision	Recall	F1 Score
1	0.94	0.836	0.816	0.813
2	0.842	0.829	0.800	0.830
3	0.848	0.830	0.810	0.930
4	0.874	0.834	0.834	0.909
5	0.851	0.843	0.843	0.904
6	0.96	0.837	0.837	0.837
7	0.872	0.831	0.829	0.829
8	0.816	0.810	0.701	0.718
9	0.813	0.838	0.73	0.73
10	0.821	0.812	0.80	0.70

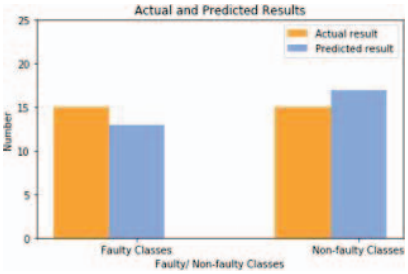
from 30 classes 15 classes and faulty class and 15 are non-faulty class. our model predict that among these 30 classes 13 classes are faulty and rest of the 17 classes are non-faulty. From Fig 5 we see that our model predict 13 actual faulty classes out of 15 classes. we use percentage of right prediction and wrong prediction to check the efficiency of our prediction. Right prediction percentage is 93.33% and wrong prediction percentage is 6.66%.

B. For Method level prediction

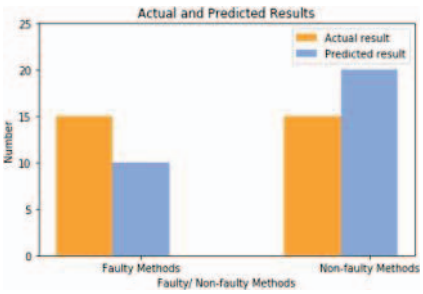
The case study we picked have 30 Methods we have complete information of faulty methods of case study which shows that from 30 methods 15 methods are faulty methods and 15 are non-faulty methods. our model predict that among these 30 methods 10 methods are faulty and rest of the 20 methods are non-faulty. From Fig 5 we see that our model predict 10 actual faulty methods out of 15 methods. we use percentage of right prediction and wrong prediction to check the efficiency of our prediction. Right prediction percentage is 83.33% and wrong prediction percentage is 16.66%.

VI. Conclusion

The role software fault prediction play is that they help in improving quality of software and assist inspection by locating possible faults in software. Efforts are necessary to minimize software faults. However, all these efforts cost time



(a) Class



(b) Method

Fig. 5: Actual and Predicted Results

and resources. It is verified that sooner a fault is detected lesser it costs. We proposed an effective and efficient model for early-stage fault prediction. The model is developed for prediction of software faults at class level and at method level using code smells. The dataset used in model building consists of different types of code smells. Labeled dataset is collected from projects available at defects4j. these datasets are used to train our classifier. code smells were extracted from the different classes of selected case syudy, which were than input to the designed classifier.The designed classifier predict faulty and non-faulty instances. We achieved satisfactory result of right prediction based on this we observe that code smells are good indicators of fault proneness. For future work we are planning to use code smells with some other pseudolabelling technique, with deep learning, and some other software metrics or process metrics with machine learning and deep learning to predict faults at the method level.

REFERENCES

[1] software bug accessed. 2018; Available from: https://en.wikipedia.org/wiki/Software_bug.

[2] Defects4J Dissection. Available from: http://program-repair.org/defects4j-dissection/#!/..

[3] Fowler, M., et al., Refactoring: improving the design of existing code, ser, in Addison Wesley object technology series. 1999, Addison-Wesley.

[4] Moha, N., et al., Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 2009. 36(1): p. 20-36.

[5] Halstead, M.H., Elements of software science. Vol. 7. 1977: Elsevier New York.

[6] McCabe, T.J., A complexity measure. IEEE Transactions on software Engineering, 1976(4): p. 308-320.

[7] Chidamber, S.R. and C.F. Kemerer, A metrics suite for object oriented design. IEEE Transactions on software engineering, 1994. 20(6): p. 476-493.

[8] Bansiya, J. and C.G. Davis, A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering, 2002. 28(1): p. 4-17.

[9] Tang, M.-H., M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. in Proceedings sixth international software metrics symposium (Cat. No. PR00403). 1999. IEEE.

[10] Martin, R., OO design quality metrics. An analysis of dependencies, 1994. 12(1): p. 151-170.

[11] Ma, W., et al., Empirical analysis of network measures for effort-aware fault-proneness prediction. Information and Software Technology, 2016. 69: p. 50-70.

[12] Menzies, T., J. Greenwald, and A. Frank, Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering, 2006. 33(1): p. 2-13.

[13] Ma, Y., L. Guo, and B. Cukic, A statistical framework for the prediction of fault-proneness, in Advances in Machine Learning Applications in Software Engineering. 2007, IGI Global. p. 237-263.

[14] Koru, A.G. and H. Liu. An investigation of the effect of module size on defect prediction using static measures. in Proceedings of the 2005 workshop on Predictor models in software engineering. 2005.

[15] Ma, W., et al. Do we have a chance to fix bugs when refactoring code smells? in 2016 International Conference on Software Analysis, Testing and Evolution (SATE). 2016. IEEE.

[16] Guo, Y., et al. Domain-specific tailoring of code smells: an empirical study. in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. 2010.

[17] Ubayawardana, G.M. and D.D. Karunaratna. Bug prediction model using code smells. in 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer). 2018. IEEE.

[18] Eken, B., et al., An empirical study on the effect of community smells on bug prediction. Software Quality Journal, 2021. 29(1): p. 159-194

[19] Palomba, F., et al., Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering, 2017. 45(2): p. 194-218.

[20] Rio, A., PHP code smells in web apps: survival and anomalies. arXiv preprint arXiv:2101.00090, 2020.

[21] Kessentini, M., Understanding the correlation between code smells and software bugs. 2019..

[22] Pritam, N., et al., Assessment of code smell for predicting class change proneness using machine learning. IEEE Access, 2019. 7: p. 37414-37425.

[23] Sotto-Mayor, B., et al., Exploring Designite for Smell-Based Defect Prediction.

[24] Catolino, G., et al., Improving change prediction models with code smell-related information. Empirical Software Engineering, 2020. 25(1): p. 49-95.

[25] Bigonha, M.A., et al., The usefulness of software metric thresholds for detection of bad smells and fault prediction. Information and Software Technology, 2019. 115: p. 79-92.

[26] Li, W. and R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of systems and software, 2007. 80(7): p. 1120-1128.

[27] Hall, T., et al., Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM), 2014. 23(4): p. 1-39.

[28] Vidal, S.A., C. Marcos, and J.A. D´ıaz-Pace, An approach to prioritize code smells for refactoring. Automated Software Engineering, 2016. 23(3): p. 501-532.

[29] Lozano, A., M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. in Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting. 2007.

[30] Olbrich, S.M., D.S. Cruzes, and D.I. Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. in 2010 IEEE International Conference on Software Maintenance. 2010. IEEE.

[31] Code Smells and their Collocations : A Large-scale Experiment on Open-source Systems. Available from: http://doi.org/10.5281/zenodo.842778.

[32] IPLASMA. Available from: http://loose.cs.upt.ro/index.php?n=Main.IPlasma.

[33] pseudo labeling. Available from: https://towardsdatascience.com/pseudo-labeling-to-dealwith-small-datasets-what-why-how-fd6f903213af