

Examining the Bug Prediction Capabilities of Primitive Obsession Metrics

Edit Pengő(✉)

University of Szeged, Dugonics square 13, Szeged 6720, Hungary
pengoe@inf.u-szeged.hu

Abstract. Bug prediction is an approach that helps make bug detection more automated during software development. Based on a bug dataset a prediction model is built to locate future bugs. Bug datasets contain information about previous defects in the code, process metrics, or source code metrics, etc. As code smells can indicate potential flaws in the source code, they can be used for bug prediction as well.

In our previous work, we introduced several source code metrics to detect and describe the occurrence of Primitive Obsession in Java. This paper is a further study on three of the Primitive Obsession metrics. We integrated them into an existing, source code metrics-based bug dataset, and studied the effectiveness of the prediction built upon it. We performed a 10 fold cross-validation on the whole dataset and a cross-project validation as well. We compared the new models with the results of the original dataset. While the cross-validation showed no significant change, in the case of the cross-project validation, we have found that the amount of improvement exceeded the amount of deterioration by 5%. Furthermore, the variance added to the dataset was confirmed by correlation and PCA calculations.

Keywords: Bug prediction · Code smells · Primitive obsession · Static analysis · Refactoring

1 Introduction

Finding and fixing bugs is a very important task during software development to ensure the quality of the product. Software defects are very costly. The later

This work was partially supported by grant 2018-1.2.1-NKP-2018-00004 Security Enhancing Technologies for the IoT" funded by the Hungarian National Research, Development and Innovation Office. It was also supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB).

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12955, pp. 185–200, 2021.
https://doi.org/10.1007/978-3-030-87007-2_14

they are discovered, the more expensive they are [3], therefore, there is considerable research interest in making the bug detection process more automated and efficient. Software fault prediction or, in other words, bug prediction is a possible approach to this effort. Considering this, it is no surprise, that there is a huge amount of literature on refining bug prediction [33,34,38]. In this paper, we also aim to find improvements with the help of our previously introduced Primitive Obsession metrics [25].

Primitve Obsession is a previously little-studied code smell [37]. As a code smell it can be a useful indicator for coding flaws [12], therefore, we introduced six metrics for its detection in our previous work [25]. The current paper examines the bug prediction ability of three of these metrics. We expanded an existing, source code metrics-based bug dataset [6] and studied the impact of the alteration. We performed a 10 fold cross-validation on the whole dataset and a cross-project validation as well. We compared the new models with the results of the original dataset and analyzed the results. In addition, a Principal Component Analysis (PCA) [28] was performed to examine the dimensionality of the new metrics.

The paper is organised as follows. Section 2 provides the background and the related work for our paper. Section 3 is an overview of the Primitive Obsession metrics on which we based our research. The three research questions and their answers are presented in Sect. 4. Section 5 describes the threats to validity. We conclude our work in Sect. 6.

2 Background

The following subsections introduce the major domains of our research, such as bug prediction and Primitive Obsession.

2.1 Bug Prediction and Bug Datasets

Bug prediction is a method that supports the quality assurance during software development. It is a cost-effective approach compared to software testing and reviews. Menzies et al. report in their study [21] that the probability of detecting a bug by fault prediction may be higher than detecting it by the review process currently used in industrial environments.

Bug prediction can be used to estimate the number of remaining bugs in the codebase or to identify the bug-prone parts of the code. Historic data can help to build models to predict the severity of bug reports [30] or to estimate the reliability of a system [2,29]. Another possible use is to mine software defect associations [26]. What we call a defect association for example is if a type A and type B defect occur in the code, then type C is likely to occur as well. Our research examines the type of bug prediction that helps identify potentially buggy code parts.

Bug prediction can be based on statistical, expert-driven, or machine learning models [4]. This paper focuses on machine learning-based bug prediction.

We used a unified bug dataset, which was introduced by Ferenc et al. [6]. Bug datasets contain historical data, such as previous defects, process metrics, or source code metrics [14], on which the prediction data is built. The dataset of Ferenc contains data from five publicly available bug datasets in a unified format: the PROMISE dataset [17], the Eclipse Bug Dataset [39], the Bug Prediction Dataset [5], the Bugcatchers Bug Dataset [15], and the GitHub Bug Dataset [32]. They contain class and/or file level source code metrics and information about the previous defects. We only utilized the PROMISE, the GitHub, and the Bug Prediction datasets, as only these contain class-level metrics. There are 66 projects in the three datasets, the smallest only contains 6 classes while the largest contains 5908. To reduce noise, we decided to exclude those systems that contain less than 100 classes. Thus, a total of 58 systems remained containing a total of 45519 classes.

2.2 Code Smells

Code smells that were introduced in Fowler's 1999 book [7], demonstrate the violation of fundamental design principles, therefore, their presence can indicate underlying flaws in the source code. There are numerous studies about the impact of code smells on code quality. Moonen and Yamashita demonstrated in an empirical study [22] that some of the code smell definitions are useful to evaluate the maintainability of a project. In another study [35] they investigated how the interactions of multiple smells affect maintainability. With the help of co-located smells they identified several artefacts that need to be prioritized during refactoring. Khomh et al. [19] found a relation between code smells and class change-proneness, which can support quality assurance and focus testing activities. In contrast to these studies, Sjøberg et al. investigated the connection between code smells and maintenance effort [31], and found that the work of changing code with smells was not significantly higher than the work of changing code without smells. The 12 examined smells have limited impact on maintenance effort in contrast to code size and the number of changes.

The results of these studies suggest that although code smells are not silver bullets for defect prediction, they can still provide valuable insights into some important maintainability factors, especially when more of them are combined. Palomba et al. [24] built a bug prediction model using code smell information as well. They combined the severity of the code smells involved (God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, Message Chains) with existing bug prediction models. They found that the accuracy of the models increases by adding the code smell information as a predictor.

2.3 PCA and Correlation

Principal Component Analysis (PCA) is a widely used data reduction technique [28]. Since this is one of the best-known dimension reduction methods, we selected it to examine the dimensionality of PO metrics. PCA reduces the dimensions of a large data set – in which the variables are interrelated – while retaining

the variance of the data as much as possible. An orthogonal transformation is used to calculate the so-called principal components, which are a set of linearly uncorrelated variables calculated from the possibly correlated variables of the data. The first principal component holds the greatest possible variance, each subsequent component will have the greatest variance remaining. PCA can be used for dimensionality reduction in large datasets, and it can uncover patterns in the data as well [1].

In this research, we used PCA to examine the variance represented by the new metrics. We also calculated the correlation of the metrics to study the magnitude and direction of their linear relationships [11].

2.4 Defining Primitive Obsession

The following description of Primitive Obsession is taken from our previous work [25] for clarity. In most programming languages, there are two categories of data types: primitive and complex. Primitive types are the most basic data types that a language provides, e.g. *boolean*, *integer*, *char*, etc. Complex types like *classes* and *structs* are made up of other existing data types e.g. primitive types and complex types. The advantage of complex types is that semantic knowledge can be connected to the data with them. For example, three integers alone do not mean anything, but if we include them in a class called 3DPoint, it is immediately clear what they represent. This is a more readable and maintainable solution that also supports encapsulation, which is one of the main principles of object-oriented programming.

Primitive Obsession can be categorized as a bloater-type code smell [20]. It means that the code relies too much on the use of primitive data types. Its occurrence is a symptom of chaotic, overgrown code parts. The programmer does not create small classes e.g. value objects to perform small tasks, therefore, the data is scattered among primitive types.

Figure 1 shows several occurrences of Primitive Obsession. There are two class constants at Lines 3 and 4. `ENGINEER` and `SALESMAN` can be considered as type codes. Type codes are a set of integer or string variables that usually have an understandable name, and they are employed to simulate types. In this case, they symbolize the different types of employees. Although type codes are preferred in projects, they are considered Primitive Obsession. Not only do they violate the object oriented paradigm, but they can cause *hidden dependencies* [36] as well. A refactoring option is if developers employ *State* or *Strategy* pattern [10].

The next Primitive Obsession example is shown in Line 6. Consider the parameter list of the `work` function. The first three of its parameters are integers, whilst the fourth is a string. Parameter lists like this can be considered a kind of Primitive Obsession, especially if they appear repeatedly in the code. It is worth noting here that, strictly speaking, a string does not qualify as a primitive type, but logically it is practical to include them in the definition. The parameter list of the `work` function can be refactored by introducing a parameter object. Lines 8 and 9 also show the symptoms of Primitive Obsession. Such value checks

```
1  class Employee
2  {
3    static const int ENGINEER = 1;
4    static const int SALESMAN = 2;
5
6    public void work(int from, int to, int numberOfBreaks,
         String task)
7    {
8      if(task == null
9          || task.length() == 0) {
10     /*...*/
11     }
12   }
13 }
```

Fig. 1. Sample code containing three primitive obsessions

should usually be encapsulated. More Primitive Obsession examples are available in Steven A. Lowe's GitHub project[1]. It shows step-by-step how to refactor a project heavily infected by Primitive Obsession.

2.5 Study of Primitive Obsession

Primitive Obsession is a code smell that has received little research attention in previous years. Only a few studies paid attention to this type of smell. Zhang et al. performed a systematic literature review about code smells [37]. They examined 309 articles between 2000 and 2009, 39 of which were analyzed in detail. They studied which were the most frequently researched code smells. The results showed that Primitive Obsession was among the unpopular smells. They only found 5 corresponding papers, although these papers examined all 22 of Fowler's bad smells. The more recent systematic literature review [13] of Gupta et al. came to a similar conclusion. They investigated 60 research papers between 1999 and 2016, and found that four of Fowler's bad smells – including Primitive Obsession – had no detection method in any of the papers.

The example in the previous section highlighted that it is hard to define Primitive Obsession with one exact formula. It has various aspects. The challenging definition can be the reason for the lack of articles and detection methods. We only found one thesis [27] that proposed an automated tool, JSmell, for detecting Primitive Obsession based on the number of primitive data types declared in a class. Sadly, no comparison or in-depth study was possible as we were unable to get hold of JSmell.

Driven by the lack of literature about Primitive Obsession we decided to study it. We introduced several metrics in our previous work [25] to describe the aspects of Primitive Obsession. These metrics are described in Sect. 3.

[1] https://github.com/stevenalowe/kata-2-tinytypes.

3 Primitive Obsession Metrics

This section describes the Primitive Obsession metrics on which we based the research of this paper. There were 6 metrics introduced in our previous papers[9, 25]. In this paper, we used only three of them: the MPC, the SFP, and the $SFP - SCU$ metrics. Their definitions are taken from the previous article[25], so that this article can be understood on its own. These metrics were originally defined as class-level metrics. Collectively, we will refer to these metrics as PO metrics.

3.1 Method Parameter Clones

First, we present the Method Parameter Clones (MPC) metric which is a class-level metric. If a class is infected by Primitive Obsession, several method parameters will appear repetitively in the parameter lists of multiple methods. These parameters have the same type and the same name because logically they correspond to the same data. These are the parameters that could be extracted to a value object instead of being used separately. The MPC metric highlights this aspect. The following steps are performed for each class in a project:

1. Initially, the MPC value for a class is set to zero.
2. For each method parameter in the class, create a $(type, name)$ pair.
3. Select only those pairs where the type is in the $PrimitiveTypes$ set.
4. Increment MPC value by one for every $(type, name)$ pair that appears at least three times.

The reason that only three or more repetitions are counted is *The Rule of Three* [7]. This means that two instances of similar code do not require refactoring, but when similar code is used three times, it needs to be corrected.

3.2 Static Final Primitives

Another class-level metric is the Static Final Primitives (SFP) metric. It captures one more essential aspect of the Primitive Obsession bad smell. The SFP metric quantifies the usage of type codes. To check whether a variable can be a candidate for type code the Static Final Primitive function is used which is described in Formula 1.

$$\begin{aligned} SFP(V) \; := & \; isClassLevel(V) \\ & \wedge isStatic(V) \\ & \wedge isFinal(V) \\ & \wedge isUpperCase(V) \\ & \wedge typeOf(V) \in PrimitiveTypes \end{aligned} \tag{1}$$

The SFP function marks a variable (V) if it has all the following properties, thus it might serve as a type code:

- V is a class-level variable,
- V is static (e.g. has a `static` modifier in Java)
- V can be assigned only once (e.g. has a `final` modifier in Java),
- the name of V contains only upper case characters, numbers, and underscores,
- and the type of V is included in the previously defined $PrimitiveTypes$ set (i.e. primitive).

Other than that, SFP returns false for the investigated variable. With this function, it is possible to filter out static final primitive variable usages in methods.

Formula 2 defines the Static Final Variable Usage (SFPU) function. There are three parameters: M_c and V denote the function and variable under investigation, respectively, while F is a filter function. In the formula, U_V denotes one usage of variable V. The $U_V \in M_c$ relation means that the U_V variable usage (access or modification) is performed in the M_c method. By applying the F filter function it is possible to select a subset of the U_V variable usages. The $SFPU$ function gathers every access and modification statement for variable V in the M_c method, for which both the SFP function and the F filter function return true.

$$SFPU(M_c, V, F) := \{U_V \mid U_V \in M_c \wedge SFP(V) \wedge F(U_V)\} \tag{2}$$

In the following paragraph, we present a specific application of the $SFPU$ function. Type code usages will most possibly appear in branching structures, particularly in `switch-case` statements. To capture this aspect, let us supply an F filter function to count the number of static final primitive variables that are used as case labels in `switch` statements. The F filter function will determine for a given U_V variable usage if it is a case label. By using the $SFPU$ and this F filter function we constructed a class-level metric named Static Final Primitive - Switch Case Usage (SFP-SCU). The calculation is done for each class to see how many times its SFP variables appear as case labels globally in the project.

4 Evaluation

We formulated three research questions to examine the effect on bug prediction and the dimensionality of the PO metrics. In this section, we enumerate and answer these questions.

4.1 RQ1: How Does Adding the PO Metrics Affect the Prediction Capability of the Original Dataset?

Our first attempt was to merge the PO metrics into the original dataset. We calculated the previously introduced class-level PO metrics for the 58 Java systems. The measurements were performed by extending the OpenStaticAnalyser (OSA) [23], which is an open source static analyser tool developed by the University of Szeged. Only those classes were kept in the dataset that were present

in the original dataset as well, i.e. the classes that have bug information. We performed a 10-fold cross-validation training with Weka [8] using the J48 algorithm on the original and on the extended dataset. Although many classifiers were applied for bug prediction [16,18], Ferenc [6] used the J48 algorithm for the evaluation of the original dataset, therefore, we decided to utilize the same algorithm with its default parameters as well.

The results of the two datasets are compared in Table 1. The first column shows the values of the original dataset, while the second column shows the values of the extended dataset. In the rows, we see the most important attributes that characterize the performance of the J48 classifier: the precision[2], the recall[3], the F-measure[4], and the AUC[5]. It can be seen that the numbers show stagnation or a slight decrease. These changes are not significant. It causes changes of similar magnitude if the random seed is changed in the 10 fold cross-validation.

Table 1. The comparison of the results of the original and extended data set for J48 classifier

	Original	Extended
Precision for the not bugged class	0.876	0.876
Precision for the bugged class	0.537	0.537
Recall for the not bugged class	0.919	0.919
Recall for the bugged class	0.421	0.419
F-measure of the not bugged class	0.897	0.897
F-measure of the bugged class	0.472	0.470

We concluded that adding the PO metrics to the original dataset does not improve the overall prediction ability. Seeing this result, we decided to examine whether there is an improvement in cross-project validation. We analyse this in the next RQ.

4.2 RQ2: Does Including PO Metrics Increase the Cross-Project Bug Prediction Capability of the Metric Set?

With this RQ, we wanted to examine if there is a project on which a better model could be built using PO metrics. A project-wise evaluation was performed

[2] The precision can be calculated as the number of true positive elements divided by the sum of the true and false positive elements. It describes what portion of the identifications was actually correct.

[3] It describes what portion of the actual relevant elements was identified correctly.

[4] F-measure is the harmonic mean of precision and recall.

[5] The ROC curve displays the performance of a classification model at all classification threshold. The Area under the ROC curve (AUC) means the area under this curve. It is an aggregated measure of the performance across every classification thresholds.

meaning that we trained a model on each project and evaluated it on every other project. Out of the projects that were included in the database more than once, we used the ones with the highest version number. We found that the results for the different versions of the same systems are quite similar, therefore, they would have too much weight in the cross-project validation. A total of 29 systems were used for the cross-project validation.

The F-measure difference between the evaluations performed on the original and the expanded dataset is shown in Table 2. Here, we used the Weighted Average F-measure[6] to compactly present the change of the F-measure values for both classes. The examination was performed with the J48 algorithm as well. The first column on the left shows which system the teaching was done on and the column labels indicate which system the trained model was evaluated on. The second column contains the ID we gave to the system. These IDs are used in the column headers. The third column, called Class, shows how many classes are in the system, while the fourth column indicates the percentage of bugs. A colourmap is applied to these columns: in the third column, large class numbers are highlighted in orange, while in the fourth column orange colour represents a low bug ratio and blue colour represents a high bug ratio. We sorted the systems in such a way that the bug rate increases from top to bottom and from left to right. The green-coloured cells indicate a significant improvement, while values written in green denote slight improvements (less than 0.05). Due to the subsequent rounding, the value of 0.05 may appear in the table with both notations. For cells with a red background and values written in red, the same rule applies but in the negative direction. They indicate a slight or a significant decrease (larger than 0.05 in absolute value) in the F-measures of the expanded dataset.

It can be seen that if there were a disproportionate number of non-bugged classes (MCT, Neo4J) or bugged classes (Log4j 1.2, Xalan 2.7) in a system, then the models will be unusable on other systems. This is the explanation for the empty cells. The models are too fitted to one of the classes, therefore, no items are classified to the other class and the F-measure cannot be calculated. Where there is a 0.00 in the cell, the change in the F-measure is so insignificant that it does not appear in the rounded value. Significant F-measure reductions can be seen in the right half of the table. This can be explained by the fact that the models were built on systems with a low bug count, while in the last few columns the bug count of the systems is high, therefore, the models cannot function well.

Most of the significant improvements were achieved with the Xerces 2.0 model. It has a lower bug ratio than the Xalan 2.7 system, but still has plenty of examples to learn and has 1.8 times more classes for the J48 algorithm. The model of the mcMMO project has also been significantly improved, at least for

[6] Weka calculates the Weighted Avg. F-Measure with the following formula for an n and y class: $Weighted\ Avg.\ F-measure = \frac{F-measure(n)\cdot NumOfInstances(x)+F-measure(y)\cdot NumOfInstances(y)}{NumOfInstances(n)+NumOfInstances(y)}$, where $NumOfInstances(n), NumOfInstances(y)$ correspond to the number of instances in the given class.

projects with similarly low bug ratios. Out of the 841 cells containing F-measure changes, 350 cells hold zero value and there are 126 blank ones (including the diagonal elements). The F-measure is reduced in 170 cases and improved in 195 cases. There are 45 significant F-measure increases in contrast to the 24 significant F-measure decreases.

Overall, the addition of PO metrics brings more improvement than deterioration in the case of cross-project validation, but it is important to choose the right system to build the model on. Further research is needed for the Xerces 2.0 and the mcMMO systems.

4.3 RQ3: Do the Three PO Metrics Add Valuable Information to the Dataset?

We calculated the correlation matrix[7] of the metrics and used Principal Component Analysis (PCA) to show the dimensionality of the PO metrics.

Table 3 only shows those parts of the correlation matrix which are relevant for the PO metrics. For better readability, the table is divided into two parts. We used a colourmap to help better understand the correlations between metrics. The greater the positive correlation between two metrics the greener the cell is. Values close to 1 are labeled with pure green colour. A negative correlation would be marked in red, but there are no values close to –1 in the tables. If the value is zero or close to zero (meaning that the correlation is insignificant) the cells have a white background. It can be seen in the tables that there is no significant correlation, neither within the PO metrics, nor between the PO metrics and the other metrics.

In the following paragraphs, the results of the PCA will be discussed. The bug information was omitted for the calculations, but the other attributes were retained. Figure 2 depicts the cumulative variability for the principal components. It can be seen that the curve is gradual, i.e., a high number of principal components are required to maintain much of the variance. To cover 99% of the variance in the dataset, 33 principal components are required, which is more than half of the original attributes. Even 95% requires 20 factors.

To further investigate the significance of PO metrics we computed the so-called factor loadings. A loading represents how much each original attribute contributes to the corresponding principal component. Table 4 shows the loadings for the first 20 principal components. Cells with an absolute value of 0.7 or greater were coloured with dark green. The absolute value of the bright green fields is between 0.5 and 0.7. Since there were very few outstanding values, only the rows where they occurred are shown in the table. To illustrate how small the loading values in the table are, we took the absolute value of the table and calculated the average and the median values. The average is 0,086, while the median is 0,06. It is clear that the 20 factors are made up of many metrics, and only a few of them have a prominent role in the composition of a factor. The PO

[7] Correlation shows the degree to which a pair of variables are linearly related. We used Pearson correlation for our experiment.

Table 2. Change in weighted average F-measure values for the cross project validation

Name	ID	Class	Bug	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
mct	1	1 887	**0.5**		0.00	0.00	0.00				0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00			0.00	0.00		0.00	0.00
neo4j	2	5 899	**1.0**																													
jedit4.3	3	439	**2.5**	0.00	0.00		0.00		0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00									
ceylon	4	1 610	**4.2**	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
antrl	5	479	**4.4**	−0.01	0.00	0.01	0.00		0.00	−0.01	0.00	0.00	0.01	0.02	0.00	0.01	0.00	0.00	0.00	0.00	0.00	−0.01	−0.01	0.00	0.01	−0.07	0.05	0.00	−0.05	0.07	−0.04	0.01	−0.11
junit	6	731	**4.8**	0.01	0.00	−0.03	0.00	−0.01		0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.00	−0.01	−0.02	−0.09	0.00	0.00	0.05	0.04	−0.05	−0.06	0.02	−0.09	−0.09	−0.04	−0.24
titan	7	1 468	**6.5**	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
hazelcast	8	3 412	**11.0**	0.00	0.01	0.00	−0.02	0.01	−0.01	0.00		0.00	−0.02	0.00	0.01	0.01	−0.01	0.00	0.01	−0.03	−0.03	0.00	−0.02	−0.02	0.04	−0.01	0.01	−0.02	0.02	−0.04	0.09	0.18
elasticsearch	9	5 908	**11.5**	0.00	0.00	−0.02	0.01	0.00	0.00	0.00	0.00		0.02	0.01	0.03	0.01	0.01	0.00	−0.01	−0.02	0.00	0.03	−0.01	0.03	−0.01	0.06	0.01	−0.03	−0.01	−0.02	−0.02	0.01
mapdb	10	331	**12.1**	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.01	0.01		0.01	0.00	0.01	0.01	0.01	0.00	−0.01	0.02	0.01	0.01	0.00	−0.01	−0.02	−0.01	−0.02	−0.11	−0.07	0.01	0.01
ivy2.0	11	294	**12.6**	0.00	0.00	−0.04	0.00	0.00		0.00	0.00	0.00	−0.01		0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.01	0.01	−0.01	0.01	0.01	0.05	0.02	0.04	0.04	0.00
oryx	12	533	**13.9**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pdeu	13	1 491	**14.0**	−0.01	0.00	−0.01	0.00	−0.01	0.00	0.00	0.00	−0.01	−0.01	0.00	−0.01		−0.01	0.00	0.02	−0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.02	−0.01	0.03
myly	14	1 405	**14.9**	0.00	−0.01	0.00	−0.01	−0.01	−0.01	0.00	0.00	−0.01	−0.02	−0.03	0.00	−0.01		−0.01	0.00	0.01	−0.02	−0.01	−0.01	−0.04	0.00	0.00	0.01	−0.02	0.00	−0.05	0.04	−0.02
orientdb	15	1 847	**15.2**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	−0.01	0.00		0.00	−0.01	−0.01	0.00	0.00	−0.03	0.04	−0.02	0.00	−0.02	0.00	0.01	0.00	−0.03
broadleaf	16	1 593	18,3	0.00	0.00	0.00	0.00	−0.01	0.00	0.00	0.01	0.00	−0.02	0.01	−0.01	0.01	0.00	−0.01		0.02	−0.01	0.01	−0.01	0.04	0.00	0.00	0.00	−0.04	−0.05	0.03	−0.02	−0.07
mcMMO	17	301	18,9	0.03	0.03	0.06	0.05	0.04	0.01	0.04	0.05	0.05	0.05	0.02	0.01	0.07	0.04	0.05	0.02		0.06	0.03	0.02	0.09	0.02	−0.04	0.00	−0.07	−0.01	−0.13	−0.11	−0.18
Eclipse JDT	18	997	20,7	−0.01	0.01	−0.01	0.01	0.03	0.01	0.01	0.00	0.00	0.00	0.01	0.00	−0.01	−0.01	−0.02	−0.01	0.06		0.00	0.06	−0.01	0.01	0.03	−0.01	−0.01	0.03	0.02	0.02	0.03
camel1.6	19	795	21,4	0.00	0.00	−0.03	0.00	0.02	−0.01	0.01	0.00	0.01	0.01	−0.02	0.00	0.01	0.00	0.00	−0.01	−0.03	−0.03		0.00	−0.04	−0.06	−0.09	0.02	0.01	0.08	0.04	0.06	0.14
netty	20	1 143	23,7	0.00	0.00	0.00	0.00	−0.01	0.00	0.00	−0.01	−0.01	0.00	0.00	−0.02	0.00	−0.01	0.01	0.00	0.00	0.00	−0.01		0.02	0.06	0.00	−0.01	−0.01	−0.02	−0.01	−0.02	0.01
ant1.7	21	681	24,2	0.01	0.00	0.05	0.01	0.01	0.00	0.00	0.00	0.01	0.01	−0.02	−0.01	−0.01	0.00	0.01	0.00	−0.02	−0.01	0.00	0.00		0.00	−0.02	−0.04	−0.02	−0.02	−0.02	0.00	0.00
velocity1.6	22	188	35,1	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−0.03	0.00	0.01	0.01	0.00	−0.04	−0.03	0.01	0.01		0.01	0.04	0.02	0.00	0.03	0.02	0.08
synapse1.2	23	228	36,8	−0.01	−0.01	−0.03	−0.02	−0.01	0.00	−0.02	−0.01	0.00	−0.05	−0.01	−0.01	0.00	−0.01	−0.02	0.00	0.00	−0.05	−0.01	−0.03	−0.01	−0.01		−0.05	0.02	0.04	−0.07	0.00	−0.02
equinox	24	319	39,5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lucene2.4	25	320	60,3	0.02	0.00	0.02	0.02	−0.01	0.00	−0.01	0.00	−0.02	0.01	−0.03	0.00	−0.04	0.00	−0.01	0.01	0.00	−0.13	−0.02	0.02	0.00	0.00	0.01	0.07		0.00	0.04	0.00	0.02
poi3.0	26	414	66,7	0.01	0.01	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.02	0.02	0.00	0.00	0.01	0.01	0.01	0.02	−0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00		0.01	0.00	0.00
xerces2.0	27	342	88,6	0.25	0.28	0.21	0.34	0.09	0.26	0.39	0.22	0.16	0.24	0.18	0.40	0.24	0.28	0.16	0.12	0.28	0.12	0.09	0.17	0.12	0.14	0.07	0.09	0.02	−0.16		−0.10	−0.01
log4j1.2	28	191	91,6																													
xalan2.7	29	848	98,7		0.00					0.00	0.00	0.00											0.00					0.00		0.00		

Table 3. Correlation between the metrics containing only the columns that are relevant for the PO metrics

	MPC	SFP-SCU	SFP
MPC	1.00	0.09	0.29
SFP-SCU	0.09	1.00	0.23
SFP	0.29	0.23	1.00
CLOC	0.40	0.17	0.45
NOD	0.02	0.03	0.01
TNPA	0.04	0.04	0.17
NLPM	0.43	0.04	0.27
TNLPA	0.02	0.05	0.25
NLPA	0.01	0.05	0.24
TNOS	0.41	0.14	0.44
TLLOC	0.47	0.15	0.51
LLOC	0.48	0.15	0.52
CLC	0.01	0.00	0.03
WMC	0.44	0.14	0.42
CCL	0.17	0.06	0.24
CC	0.02	0.00	0.04
TNA	0.08	0.07	0.21
NOC	0.02	0.01	0.00
PDA	0.38	0.05	0.24
TLOC	0.48	0.17	0.55
CCO	0.13	0.07	0.19
TNS	0.10	0.02	0.10
NA	0.07	0.07	0.20
NLA	0.10	0.11	0.34
DLOC	0.40	0.15	0.31
NPA	0.03	0.04	0.16
NII	0.23	0.10	0.11
NLS	0.18	0.03	0.11
LLDC	0.22	0.04	0.27
TNLS	0.18	0.03	0.12
CI	0.15	0.05	0.21
TCLOC	0.40	0.17	0.45

	MPC	SFP-SCU	SFP
NLM	0.48	0.10	0.35
TNPM	0.18	0.01	0.16
NG	0.14	0.01	0.13
NLG	0.32	0.05	0.23
TNLA	0.11	0.11	0.35
NOA	0.02	−0.01	0.02
TNM	0.18	0.06	0.19
TCD	0.07	0.03	0.07
AD	0.10	0.04	0.11
PUA	0.23	0.00	0.14
LDC	0.22	0.04	0.26
NL	0.23	0.09	0.27
CBO	0.25	0.07	0.31
NS	0.09	0.01	0.07
NOP	0.03	−0.01	0.02
CLLC	0.02	0.00	0.04
LOC	0.49	0.17	0.55
DIT	−0.01	−0.01	0.01
NPM	0.19	0.01	0.15
TNLM	0.45	0.11	0.35
NOI	0.33	0.07	0.35
LCOM5	0.18	0.01	0.07
TNLPM	0.41	0.04	0.28
NM	0.21	0.05	0.19
CD	0.07	0.03	0.07
NOS	0.42	0.14	0.44
TNLG	0.30	0.06	0.23
CBOI	0.17	0.06	0.09
RFC	0.45	0.10	0.40
TNG	0.13	0.03	0.14
NLE	0.24	0.08	0.26
bugs	0.12	0.04	0.18

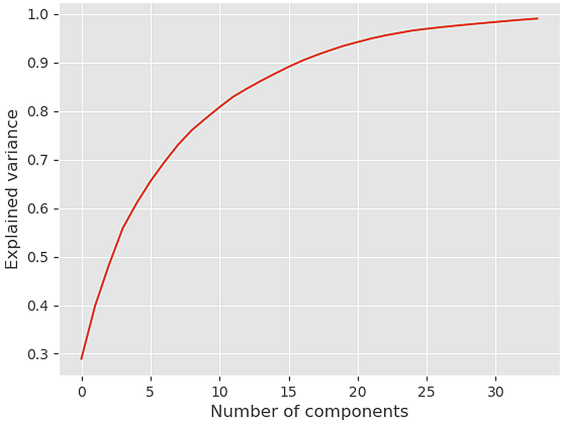

Fig. 2. Variability of the principal components

metrics contribute significantly to components 13, 18, 19. That is, using PCA analysis, we concluded that PO metrics contributed notably to the variance of the dataset.

Table 4. Factor loadings of the first 20 factors showing only the significant values

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
MPC	0.12	0.04	−0.04	−0.07	0.03	−0.02	−0.07	−0.06	0	0.16	0.04	0.12	0.03	0	−0.09	−0.09	0.53	−0.13	0.73	−0.11
SFP-SCU	0.03	0.03	0.03	−0.03	−0.02	−0.01	−0.02	−0.03	0.08	0.01	0.12	0.29	0.06	0.82	0.1	−0.18	−0.21	0.11	0.02	−0.31
SFP	0.12	0.07	0.08	−0.03	0	−0.01	0	−0.11	0.14	0.04	0.08	0.16	0.09	0.24	0.2	0.06	0.14	−0.26	−0.09	0.79
NII	0.08	−0.01	−0.03	−0.14	0.07	0	−0.21	0.33	0.08	0	0.03	0.05	−0.01	0.13	−0.55	0.05	−0.03	0.02	−0.03	0.15
TNLG	0.14	−0.09	−0.08	−0.11	0.14	0.09	0.07	0.01	−0.07	−0.02	0.07	−0.03	0.51	−0.09	0.06	−0.06	−0.17	0.03	0.08	0

5 Threats to Validity

The results are based on only one learning algorithm. Further studies with other classifiers are needed for the full picture. It is possible that for different learning algorithms, such as Random Forest or KNN, the extended dataset would yield different results.

The bug prediction database we used is widely known and accepted, however, it is relatively old and does not have a system newer than Java 8. Although we discarded projects that did not have a sufficient number of classes, there were still some systems in which the bug ratio was too low or too high to build a usable model. For real applications, a bug rate above 90% is unlikely. Our results also confirmed the idea that these systems are not effective for bug prediction. Additional databases should be tested for comparison.

The calculation of the PO metrics is currently only implemented for one programming language, Java. In order to fully evaluate their usefulness, it is

necessary to adapt them to other languages as well, e.g. C ++, JavaScript. However, in addition to the implementation tasks, building a bug database for these languages would also be a challenge.

6 Conclusion and Future Work

In this paper, three Primitive Obsession metrics were investigated: the Method Parameter Clones (MPC), the Static Final Primitives (SFP), and the Static Final Primitive - Switch Case Usage ($SFP - SCU$) metric. We examined the usefulness of these new metrics by integrating them into an existing bug prediction dataset and checking how they change the bug prediction ability of the dataset. RQ1 showed that adding them to the entire dataset did not result in an improvement, however, in the case of cross-project bug prediction they significantly contributed to the improvement of multiple prediction models. Even though they also caused a decrease in the predictive ability of some models, the number of improvements is larger than the number of declines by an extent of 25. That is, the PO metrics bring improvement in 28,5% of the cases, while the amount of deterioration is 23%. For the Xerces 2.0 system, the improvement was remarkably high on systems that have a low bug count. Adding PO metrics brings improvement in 90% of cases and a decrease in only 7% of cases.

With RQ3 we found that the PO metrics indeed contribute to the variance of the dataset. They did not correlate with the previous source code metrics. It can be seen that PO metrics are a new, useful approach for describing and measuring the source code.

In the future, we would like to expand the study with other learning algorithms and datasets. We want to take a closer look at the case of different versions of the same system. We want to study what systems can be used well for model building to stably improve bug prediction capability using PO metrics.

References

1. Becker, B., Mooney, C.: Categorizing compiler error messages with principal component analysis. In: Proceedings of the 12th China-Europe International Symposium on Software Engineering Education (CEISEE 2016), pp. 1–8 (2016). https://researchrepository.ucd.ie/handle/10197/7889
2. Behera, R.K., Rath, S.K., Misra, S., Leon, M., Adewumi, A.: Machine learning approach for reliability assessment of open source software. In: Misra, S., et al. (eds.) ICCSA 2019. LNCS, vol. 11622, pp. 472–482. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24305-0_35
3. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001). https://doi.org/10.1109/2.962984
4. Catal, C., Diri, B.: A systematic review of software fault prediction studies. Expert Syst. Appl. **36**(4), 7346–7354 (2009). https://doi.org/10.1016/j.eswa.2008.10.027. http://www.sciencedirect.com/science/article/pii/S0957417408007215
5. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 31–41 (2010). https://doi.org/10.1109/MSR.2010.5463279

6. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java and its assessment regarding metrics and bug prediction. Softw. Qual. J. (2020). https://doi.org/10.1007/s11219-020-09515-0
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
8. Frank, E., Hall, M.A., Witten, I.H.: Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques, Fourth Edition. Morgan Kaufmann (2016)
9. Gál, P., Pengő, E.: Primitive enthusiasm: a road to primitive obsession. In: The 11th Conference of PhD Students in Computer Science, pp. 134–137. University of Szeged (2018)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co. Inc., USA (1995)
11. Godfrey, K.: Correlation methods. In: IFAC Proceedings, vol. 12, pp. 527–534 (1979). https://doi.org/10.1016/S1474-6670(17)53974-9. Tutorials presented at the 5th IFAC Symposium on Identification and System Parameter Estimation, Darmstadt, Germany, September
12. Gupta, A., Suri, B., Kumar, V., Misra, S., Blazauskas, T., Damasevicius, R.: Software code smell prediction model using shannon, rényi and tsallis entropies. Entropy **20**, 372 (2018). https://doi.org/10.3390/e20050372
13. Gupta, A., Suri, B., Misra, S.: A systematic literature review: code bad smells in java source code. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10408, pp. 665–682. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62404-4_49
14. Gyimóthy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**(10), 897–910 (2005). https://doi.org/10.1109/TSE.2005.112
15. Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. **23**(4) (2014). https://doi.org/10.1145/2629648
16. Jiang, Y., Cukic, B., Ma, Y.: Techniques for evaluating fault prediction models. Empirical Softw. Engg. **13**(5), 561–595 (2008). https://doi.org/10.1007/s10664-008-9079-3
17. Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. PROMISE 2010. ACM (2010). https://doi.org/10.1145/1868328.1868342
18. Kaur, A., Kaur, I.: An empirical evaluation of classification algorithms for fault prediction in open source projects. J. King Saud Univ. Comput. Inf. Sci. **30** (2016). https://doi.org/10.1016/j.jksuci.2016.04.002
19. Khomh, F., Di Penta, M., Gueheneuc, Y.: An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th Working Conference on Reverse Engineering, pp. 75–84 (2009). https://doi.org/10.1109/WCRE.2009.28
20. Mäntylä, M.V., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of the International Conference on Software Maintenance. ICSM, pp. 381–384. IEEE (2003). https://doi.org/10.1109/ICSM.2003.1235447
21. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. **17**, 375–407 (2010). https://doi.org/10.1007/s10515-010-0069-5

22. Moonen, L., Yamashita, A.: Do code smells reflect important maintainability aspects? In: Proceedings of the 2012 IEEE International Conference on Software Maintenance. ICSM, pp. 306–315. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405287
23. Open Static Analyser GitHub Page: https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer
24. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng. **45**(2), 194–218 (2019). https://doi.org/10.1109/TSE.2017.2770122
25. Pengő, E., Gál, P.: Grasping primitive enthusiasm - approaching primitive obsession in steps. In: Proceedings of the 13th International Conference on Software Technologies. ICSOFT, pp. 389–396. INSTICC, SciTePress (2018). https://doi.org/10.5220/0006918804230430
26. Song, Q., Shepperd, M., Cartwright, M., Mair, C.: Software defect association mining and defect correction effort prediction. IEEE Trans. Softw. Eng. **32**(2), 69–82 (2006). https://doi.org/10.1109/TSE.2006.1599417
27. Roperia, N.: JSmell: a bad smell detection tool for java systems. Master's thesis, Maharishi Dayanand University (2009)
28. Shlens, J.: A tutorial on principal component analysis (2014)
29. Shukla, S., Behera, R.K., Misra, S., Rath, S.K.: Software reliability assessment using deep learning technique. In: Chakraverty, S., Goel, A., Misra, S. (eds.) Towards Extensible and Adaptable Methods in Computing, pp. 57–68. Springer, Singapore (2018). https://doi.org/10.1007/978-981-13-2348-5_5
30. Singh, V., Misra, S., Sharma, M.: Bug severity assessment in cross project context and identifying training candidates. J. Inf. Knowl. Manage. **16**, 1750005 (2017). https://doi.org/10.1142/S0219649217500058
31. Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T.: Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. **39**(8), 1144–1156 (2013). https://doi.org/10.1109/TSE.2012.89
32. Tóth, Z., Gyimesi, P., Ferenc, R.: A public bug database of github projects and its application in bug prediction. In: Gervasi, O., et al. (eds.) ICCSA 2016. LNCS, vol. 9789, pp. 625–638. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42089-9_44
33. Wahono, R.S.: A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. J. Softw. Eng. **1**, 1–16 (2015). https://doi.org/10.3923/JSE.2007.1.12
34. Weyuker, E., Ostrand, T., Bell, R.: Comparing the effectiveness of several modeling methods for fault prediction. Empirical Softw. Eng. **15**, 277–295 (2010). https://doi.org/10.1007/s10664-009-9111-2
35. Yamashita, A., Moonen, L.: Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 682–691 (2013). https://doi.org/10.1109/ICSE.2013.6606614
36. Yu, Z., Rajlich, V.: Hidden dependencies in program comprehension and change propagation. In: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, pp. 293–299 (2001). https://doi.org/10.1109/WPC.2001.921739
37. Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maintenance Evol. **23**(3), 179–202 (2011). https://doi.org/10.1002/smr.521

38. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B.: Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 91–100. ESEC/FSE 2009, Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1595696.1595713
39. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, p. 9. PROMISE 2007, IEEE (2007). https://doi.org/10.1109/PROMISE.2007.10