Can Lexicon Bad Smells improve fault prediction?

Surafel Lemma Abebe
and Paolo Tonella
Fondazione Bruno Kessler (FBK)
38050 Povo, Trento - ITALY
surafel@fbk.eu,
tonella@fbk.eu

Venera Arnaoudova, Giuliano Antoniol
and Yann-Gaël Guéhéneuc
École Polytechnique de Montréal
Montréal, (Québec) Canada, H3T 1J4
venera.arnaoudova@polymtl.ca,
antoniol@ieee.org,
yann-gael.gueheneuc@polymtl.ca

***Abstract*—In software development, early identification of fault-prone classes can save a considerable amount of resources. In the literature, source code structural metrics have been widely investigated as one of the factors that can be used to identify faulty classes. Structural metrics measure code complexity, one aspect of the source code quality. Complexity might affect program understanding and hence increase the likelihood of inserting errors in a class. Besides the structural metrics, we believe that the quality of the identifiers used in the code may also affect program understanding and thus increase the likelihood of error insertion.**

In this study, we measure the quality of identifiers using the number of Lexicon Bad Smells (LBS) they contain. We investigate whether using LBS in addition to structural metrics improves fault prediction. To conduct the investigation, we assess the prediction capability of a model while using i) only structural metrics, and ii) structural metrics and LBS. The results on three open source systems, ArgoUML, Rhino, and Eclipse, indicate that there is an improvement in the majority of the cases.

I. Introduction

The cost of identifying and fixing faults in a system already in production may be extremely high. To avoid such costs, developers spend a large portion of the system development time on testing, to identify faulty classes prior to release. To assist developers in this respect, various studies have been conducted in the research community measuring the quality of the source code using structural metrics [1], [2], [3], [4], process metrics [5], [6] or previous faults [7], [8]. Structural metrics are a lightweight alternative and they have been shown to have good performance for fault prediction [9].

The Chidamber and Kemerer object-oriented metrics suit (CK metrics) [10] is widely used as a representative of structural metrics. The underlying idea for using these metrics is that if the code is complex, it will be also difficult to understand and maintain; hence, it is susceptible to the introduction of faults. The CK metrics are based on information about the *structure* of the source code. Besides the structural complexity, other researchers have shown the importance of source code identifiers [11], [12], [13]. We concur with those works and believe that the lexicon used in naming identifiers has an impact on the understandability of the code. To measure the linguistic quality of identifiers we use the catalog of Lexicon Bad Smells (LBS) defined by Abebe *et al.* [14]. LBS are potential identifier construction problems that can compromise the quality of the identifier and hence hinder program understanding. The advantage of LBS with respect to other measures is that LBS are easier to understand, interpret and eventually avoid or fix.

Several factors contribute to the faultiness of a class. Structural complexity of a source code is one of the factors which is widely studied to predict fault prone classes. Another factor which we believe contributes to the faultiness of classes is LBS. LBS address the quality of the source code from the lexicon point of view. Hence, we conjecture that adding such information to the structural metrics used in fault proneness prediction will improve the prediction. In this study, we investigate if this conjecture holds or not. Prior to such investigation, as a sanity check we have assessed whether LBS add any new information with respect to the CK metrics; results are positive. To conduct the prediction, we first identify the best model that can be obtained with the CK metrics and then we investigate whether adding LBS to the CK metrics improves the prediction. The results indicate that there is an improvement in the majority of the cases. Following the results, we have also carried out a study to identify those LBS that contribute the most to the improvement of the prediction.

In Section II, we give background information on the structural metrics and LBS. The related works are discussed in Section III. In the next two sections, Sections IV and V, we describe the steps involved in fault prediction and the case study conducted, respectively. The threats to the validity of our study are presented in Section VI. Finally, conclusion of the study and future works are addressed in Section VII.

II. Background

A. Structural metrics

Table I shows the structural metrics considered by Kpodjedo *et al.* [15]. The list consists of the set of well-known CK metrics [10], two metrics measuring the lack of cohesion in methods (LCOM2 and LCOM5) defined by Briand *et al.* [16], and two metrics counting the number of declared attributes and methods [17].

B. Lexicon bad smells

Besides the structural complexity of the source code, understanding a system depends on the quality of lexicon used in identifier construction [11]. Good quality identifiers contribute to the understanding of the software, hence making it less

TABLE I
LIST OF CONSIDERED STRUCTURAL METRICS.

Acronym	Description
CBO [10]	Coupling between objects
DIT [10]	Depth of Inheritance Tree
LCOM1 [10]	Lack of COhesion in Methods 1
LCOM2 [16]	Lack of COhesion in Methods 2
LCOM5 [16]	Lack of COhesion in Methods 5
LOC [10]	Line Of Code
NAD [17]	Number of Attributes Declared
NMD [17]	Number of Methods Declared
NOC [10]	Number Of Children
RFC [10]	Response For a Class
WMC [10]	Weighted Methods per Class

susceptible to the introduction of faults [13]. Anomalies that reduce the quality of identifier names are called lexicon bad smells (LBS) [14]. Lexicon bad smells can usually be fixed through renaming. A list of such smells are defined and made available online[1] by Abebe et. al. [14]. In our study we have used all the LBS currently available online. Below we present a summary of the LBS with an example. The details are available online.

a) ***Extreme contraction****:* refers to extremely short terms used in identifiers due to an excessive word contraction, abbreviation, or acronym. An example of such identifier is *aSz* (*a*=array, *sz*=size). This rule does not apply to prefixes introduced due to the naming conventions adopted in the system (e.g., *m_* is a prefix used in the Hungarian notation to mark attributes of a class), common programming and domain terms (e.g., *msg*, *SQL*, etc.), and short dictionary words (e.g., *on*, *it*, etc.).

b) ***Inconsistent identifier use****:* refers to two or more identifiers that refer to a concept in an inconsistent way. Operationally, an identifier is considered inconsistent when it is contained in another identifier of the same type (e.g., another class/method/attribute name), which is found in the same container entity (e.g., package, class). An example is given in Figure 1.

```
class Documents {
   private String absolute_path;
   private String relative_path;
   private String path;//path is inconsistent
}
```

Fig. 1. Example: Inconsistent identifier use LBS

c) ***Meaningless terms****:* refers to metasyntactic identifier names like *foo* and *bar*.

d) ***Misspelling****:* refers to misspelled words in an identifier.

e) ***Odd grammatical structure****:* refers to identifiers constructed using inappropriate grammatical structure for the specific kind of software entity they represents (e.g., a class name contains a verb, method names do not start with a verb, etc.). Figure 2 shows an example of class and method identifier names that are grammatically incorrect.

[1]http://selab.fbk.eu/LexiconBadSmellWiki/

```
class Compute //compute is a verb {
   public void addition();//addition is a noun
}
```

Fig. 2. Example: odd grammatical structure LBS

f) ***Overloaded identifiers****:* refers to identifiers that include more than one semantics and hence multiple responsibilities of the respective software entities they represent (e.g., a method name contains two verbs). The method name *create_export_list()*, for example, could refer to two tasks: creating and exporting a list.

g) ***Useless type indication****:* refers to identifiers that provide redundant information about their type. For example, the attribute name *nameString* in the attribute declaration *String nameString* gives redundant information about its type. This rule does not apply for a static attribute used to realize the singleton design pattern, which usually has the same name as the class, and individual characters or groups of characters used to denote the type of the variable, if these are prescribed in the adopted naming conventions (i.e., in the Hungarian notation, *i* is a prefix used in identifiers of integer type).

h) ***Whole-part****:* refers to a term used to name a concept that appears also in the name of its properties or operations. Figure 3 shows the ambiguous and redundant use of the concept *account*. Exceptions to this rule are a static attribute, used to realize the singleton design pattern and constructor methods, as they have the same name as the class.

```
class Account {
   int account; //Ambiguous use
   void computeAccount();
   //Account is redundant information
}
```

Fig. 3. Example: whole-part LBS

i) ***Synonyms and similar identifiers****:* refers to synonym or similar terms used to construct the identifiers representing different entities declared in the same container, such that differentiating between their responsibilities is difficult. An example of this LBS is the use of the synonym terms *copy* and *replica* in identifiers *idCopy* and *idReplica*.

j) ***Terms in wrong context****:* refers to using terms that pertain to the domain of another container (e.g., package). This indicates that the entity named by such terms may be misplaced. For example, in Figure 4 the class *TypeDetector* is wrongly placed in package *collections* or incorrectly named as all the other classes that refer to detector are in package *detectors*.

Example:

```
package collections;
class IntArray;
class TypeDetector ;
package detectors;
class MuonDetector ;
class PhosDetector ;
class HLTDetector ;
```

Fig. 4. Example: terms in wrong context LBS

*k) **No hyponymy/hypernymy in class hierarchies**:* refers to an identifier representing a child class in an inheritance hierarchy but is not hyponym of the identifier of its parent class. An example of such LBS is a class named *Violin* that extends the class *Mammal*. This violation is hard to assess when a class identifier is constructed from more than one term or contains abbreviations, contractions, or acronyms.

*l) **Identifier construction rules**:* refers to identifiers that do not follow a standard naming convention adopted in the system, prescribing the use of proper prefixes, suffixes, and term separators. In a system that adopts the Hungarian notation, for example, an attribute that does not start with one of the prefixes defined for the attributes (e.g., *m_*) is considered to have this LBS.

III. Related Work

Different approaches have investigated the prediction of fault proneness using various measures, such as structural metrics, applying various models, and considering also linguistic information. In the following paragraphs we discuss those that closely relate to our work and only the details that are relevant to the comparison.

Structural information: Structural information has been widely used in the literature for fault prediction. Basili *et al.* [1] were the first to empirically investigate the ability of CK metrics to predict fault proneness in medium size C++ systems. Later Gyimóthy *et al.* [18] used the CK metrics for predicting faults in Mozilla. Zhou and Leung [2] used the CK metrics for predicting high and low severity faults. Zimmermann *et al.* [3] predict faults in Eclipse at different levels of abstraction. Metrics at class level include NOF (Number Of Fields) and NOM (Number Of Methods). Kpodjedo *et al.* [15] used a superset of the CK metrics and DEM, a set of design evolution metrics. We adopted the superset of metrics used in the latter work as a base set of structural metrics.

Linguistic information: Previous work has investigated the relation between linguistic information and fault proneness. In [19] and [20] the authors have shown that the conceptual measures of coupling and cohesion (Cocc and C3) capture new dimensions not captured by the corresponding families of structural metrics, *i.e.*, structural coupling and cohesion respectively. Binkley *et al.* [21] predict the number of faults with QALP, LOC, and SLOC. QALP measures the similarity between method's code and comment. Arnaoudova *et al.* [22] show that HEHCC helps LOC to explain fault proneness. HEHCC measures the physical and conceptual dispersion of identifier terms across different entities. We share with those works the conjecture that linguistic information is important and capture a new dimension with respect to other types of metrics. We use lexicon bad smells in addition to the base set of structural metrics.

Techniques and Prediction models: Poshyvanyk and Marcus [19], [20] used Principal Component Analysis (PCA) to show that Cocc and C3 capture different dimensions with respect to the compared structural metrics. Logistic Regression is mainly used when the explanatory variable is a dichotomous variable [2], [3], [4], [15], [18], [20], [22]. For Weyuker *et al.* random forest was one of the two models that performed best [8]. Random Forest was also used by [2], [4], [23]. Elish *et al.* [24] showed that within the context of four NASA datasets, Support Vector Machine (SVM) performs better when compared to eight other models, among which logistic regression and random forest. In this study we have used PCA, Logistic Regression, Random Forest, and SVM to analyze, model, and answer our research questions.

Prediction configuration: We share with Kamei *et al.* [23] the prediction configuration, *i.e.*, both predicting fault proneness within the same release using cross validation and predicting fault proneness for the next release. Bell *et al.* [25] use consecutive releases to train the model and predict on the next release. D'Ambros *et al.* [9] collect bi-weekly snapshots up to the prediction release x and use this data to train the models. The authors predict post release defects for x.

Evaluation metrics: Accuracy, Correctness, Completeness, and F-measure are common measures for evaluation of prediction models and have been used in previous works [2], [3], [18], [20]. Hassan [6] used the absolute error to evaluate the models. Mende and Koshke [4] proposed P_{opt} as an effort-sensitive evaluation metric. Kamei *et al.* [23] used P_{opt} to evaluate their prediction models. Weyuker *et al.* [8], [25] used the fault-percentile-average. We considered all of the described measures for the evaluation of the models as well as Matthew's Correlation Coefficient (MCC), which is used in medicine.

IV. Fault Prediction

In this section we describe the theory behind each step of our approach.

A. PCA

Principal Component Analysis (PCA) is a technique that uses solutions from linear algebra to project a set of possibly correlated variables into a space of orthogonal Principal Components (PC), or eigen vectors, where each PC is a linear combination of the original variables. PCA is used to reveal hidden patterns that cannot be seen in the original space and to reduce the number of dimensions. When using PCA it is a common practice to select a subset of the principal components and discard those that explain only a small percentage of the variance. For each principal component, PCA reports the coefficients of the attributes on the corresponding eigen vector. Those coefficients are interpreted as the importance of the attribute on the PC.

B. Prediction models

In the subsequent paragraphs we provide a brief overview of the three models we considered.

Logistic Regression: The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\ldots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\ldots+C_n\cdot X_n}}$$

where X_i are the characteristics describing the source code classes, *i.e.*, the independent variables, and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In a logistic regression model, the dependent variable π is commonly a dichotomous variable, and thus, assumes only two values $\{0, 1\}$, *i.e.*, it states whether a class is faulty (1) or not (0). The closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the entity contains a fault. The C_i are the estimated regression coefficients, the higher the absolute value, the higher the contribution of the corresponding independent variable.

Random Forest: Random Forest [26] averages the predictions of a number of tree predictors where each tree is fully grown and is based on independently sampled values. The large number of trees avoids over fitting. Random Forest is known to be robust to noise and to correlated variables.

Support Vector Machine: SVM is a machine learning technique that tries to maximize the margin of the hyperplane separating different classifications. Some of the advantages of SVM include the possibility to model linear and non-linear relations between variables and its robustness to outliers.

C. Evaluation metrics

In the literature, various evaluation metrics are used to evaluate the prediction capability of independent variables and to compare prediction models [2], [3], [4], [6], [8]. We have categorized these metrics into three groups: *rank*, *classification*, and *error* metrics. Below we present the details of each category.

1) Rank: Rank metrics sort the classes based on the value of the dependent variable assigned to each class. Then a cumulative measure is computed using the actual values of the dependent variable over the ranked classes to assess the model and/or the independent variables. In our study, we have considered two types of rank metrics: P_{opt} and FPA (Fault Percentile Average).

a) P_{opt}: is an extension of the Cost Effective (CE) measure defined in [27]. P_{opt} takes into account the costs associated with testing or reviewing a module and the actual distribution of faults, by benchmarking against a theoretically possible optimal model [4]. It is calculated as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the optimal and the predicted cumulative lift charts. The cumulative lift chart of the *optimal* curve is built using the actual defect density of classes sorted in decreasing order of the defect density (and increasing lines of code, in case of ties). The cumulative lift chart of the *predicted* curve is built like the optimal curve, but with classes sorted in decreasing order of fault prediction score.

b) FPA: is obtained from the percentage of faults contained in the top $m\%$ of classes predicted to be faulty. It is defined as the average, over all values of m, of such percentage [8], [25]. On classes listed in increasing order of predicted numbers of faults, FPA is computed as:

$$\frac{1}{NK}\sum_{k=1}^{K}(k * n_k)$$

where N is total number of actual faults in a system containing K classes, n_k is the actual number of faults in the class ranked k [8].

In our study, however, we predict the probability of fault proneness of a class instead of the number of faults. Hence, we have adapted the metrics by using the predicted probability of fault proneness to sort the classes, and 0 and 1 are used as a replacement of the number of defects. 1 is used when a class is actually faulty; 0 otherwise.

2) Classification: Predicting fault proneness of a class is a classification problem. Hence, in various studies the confusion matrix (shown in Table II) is used to evaluate models and analyze the prediction capability of the independent variables. From the confusion matrix the following measures are computed to conduct the evaluation.

a) Accuracy (A): measures how accurately both the actual faulty and non-faulty classes are classified as faulty and non-faulty by the predictor. It is computed as the ratio of the number of classes that are correctly predicted as faulty and non-faulty to the total number of classes $A = (TP + TN)/(TP + TN + FP + FN)$. A score of 1 indicates that the model used for the prediction has classified all classes as faulty and non-faulty correctly.

b) Correctness (P): is the precision of a predictor in identifying the faulty classes as faulty. It is computed as the ratio of classes which are correctly predicted as faulty to the total number of classes which are predicted to be faulty $P = TP/(TP + FP)$. A prediction model is considered very precise if all the classes predicted as faulty are actually faulty, i.e. if $P = 1$.

c) Completeness (R): is the recall of a predictor. It tells how many of the actually faulty classes are predicted as faulty. Completeness is computed as the ratio of the number of classes which are correctly predicted as faulty to the total number of classes which are actually faulty in the system $R = TP/(TP + FN)$.

d) F-measure (F): is a measure used to combine the above two inversely related classification metrics, correctness, and completeness. F-measure is computed as the harmonic mean of correctness and completeness ($F = (2 * P * R)/(P + R)$).

e) Matthew's Correlation Coefficient (MCC): is a measure commonly used in the bioinformatics community to evaluate the quality of a classifier [28]. It is a measure which is quite robust in the presence of unbalanced data. MCC is computed as:

$$\frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The value of MCC ranges from -1 to 1. -1 indicates a complete disagreement while 1 indicates the opposite.

3) Error: In the last category of the evaluation metric types, we have *absolute error (E)*. Absolute error is a measure based on the number of faults incorrectly predicted or missed:

$$E = \sum_{k=1}^{K} |\hat{y_k} - y_k|^2$$

TABLE II
PREDICTION CONFUSION MATRIX (TP=TRUE POSITIVE, TN=TRUE NEGATIVE, FP=FALSE POSITIVE, FN=FALSE NEGATIVE)

		Actual	
		Faulty	**Not faulty**
Predicted	**Faulty**	TP	FP
	Not faulty	FN	TN

where $\hat{y_k}$ is the predicted number of faults in class k and y_k the actual number of faults [6]. As we are interested in the fault proneness of a class and not in the number of faults it contains, we use 0 and 1, as a replacement of the number of faults. 1 is used when a class is actually faulty/predicted to be faulty and 0 otherwise. Unlike the other evaluation metrics, for absolute error a value closer to 0 indicates better prediction capability.

V. CASE STUDY

A. Research questions

Structural metrics measure different aspects of the code that can be used to predict fault proneness of a class. In this study, we conjecture that the quality of identifiers has also an impact on the fault proneness of a class, besides the structural metrics. To prove this conjecture, we have formulated the following three research questions:

- RQ1: Do LBS bring new information with respect to structural metrics?
- RQ2: Do LBS improve fault prediction?
- RQ3: Which LBS help more to explain faults?

In the first research question, RQ1, we have investigated if LBS measure the same aspects of the code as structural metrics or not. To carry out this investigation, following Marcus et. al. [20], we have used PCA. PCA aggregates the metrics into few orthogonal components called principal components (PC). We use the information captured in the PCs to analyze and answer RQ1. In particular, we analyze the following two aspects of the PCs: i) the number of times an LBS contributes to at least one retained PC, and ii) the number of times an LBS is the major contributor to at least one retained PC.

In RQ2 we have, then, investigated if our conjecture holds by assessing LBS' contribution, in addition to the structural metrics, in improving the prediction capability of a model. To assess LBS' contribution, we have carried out predictions using as independent variables, on the one hand, only structural metrics, and on the other hand, structural metrics plus LBS. The capability of prediction is then evaluated using the evaluation metrics described in Section IV-C. We then compare the results using the achieved net improvements and the average delta percentage. Prior to the comparison of the two sets of independent variables, we compare and select the best model in predicting fault prone classes using only the CK metrics.

The last research question, RQ3, is focused on identifying those LBS that contribute the most to the prediction of fault prone classes. To answer this research question, we use the weights assigned to each LBS by the model and we compute the median rank of each LBS.

B. Experimental setting

In the following we describe the experimental setting of our study, starting with the dependent and independent variables and then continuing with the setting for each research question.

1) Variables: For building the prediction models we considered the following dependent and independent variables:

Dependent variable: As dependent variable we use HASB, a dichotomous variable indicating whether a class is faulty or not. The associated experimental data have been previously published by Khomh *et al.* [29].

Independent variables: The overall set of independent variables consists of the CK set of metrics as considered by Kpodjedo *et al.* [15] and the LBS defined by Abebe *et al.* [14]. The set of CK metrics has been calculated using the POM framework [30]. To identify LBS, we have used a suite of tools called *LBSDetectors* [2] which have been developed for use in previous studies [14], [31]. The tool implements heuristics to automatically detect and report LBS in class, attribute, and method identifier names.

2) RQ1: To select a subset of the PC we used a threshold of 95% (similar to [20]). That is, we retained the components that explain up to 95% of the variance. For each principal component, we apply a 10% relative threshold to decide which attributes contribute to the component and we rank the attributes of each PC based of their importance (weight). If LBS bring new information with respect to structural metrics then LBS will be kept in the retained principal components and will give major contributions to them. To answer this research question we analyze two aspects: i) the number of times an LBS contributes to at least one retained PC, and ii) the number of times an LBS is the major contributor of at least one retained PC.

3) RQ2: Here we describe the particular settings of each model. All computations are performed using R[3].

Logistic Regression Model: We used the Generalized Linear Model (package stats) glm (family=binomial("logit")). We perform backward variable elimination and predict using the retained variables.

Random Forest: We use the function randomForest (package randomForest) with the number of trees being 500 as did Weyuker *et al.* [8].

Support Vector Machine: We used the Support Vector Machine model (package e1071) svm (kernel="radial"). Elish *et al.* [24] used the same kernel, which showed good performance.

Common settings: The following settings are common for all models: As Gyimóthy *et al.* [18] we standardize all metrics before performing the calculations (*i.e.*, zero mean and unit variance). Like in Kamei *et al.* [23], for each type of

[2]http://selab.fbk.eu/LexiconBadSmellWiki/
[3]http://www.r-project.org/

model, we predict faulty classes in two configurations: within the same version and for the next version. Prediction within the same version represents scenarios in which there is no prior record of buggy classes and new systems while the latter represents scenarios in which a system's evolution is well documented. When predicting within the same version, we use 10-fold cross validation. For each configuration we build two models: one where the independent variables are the CK set of metrics alone and the second where the independent variables are CK and LBS.

4) RQ3: To decide which LBS best help for fault prediction we rank the attributes based on the their importance in the best model selected in RQ2. We then calculate the median rank across the versions of the system and select the top three LBS separately for each subject system.

C. Subjects

For our case study, we have considered three open source systems written in Java, ArgoUML [4], Eclipse [5], and Rhino [6]. ArgoUML is a UML modeling tool which includes support for all standard UML 1.4 diagrams while Eclipse is an IDE which supports different languages. In this study we have used the IDE for Java. Rhino is a Java implementation of JavaScript. The summary of the versions of the systems used in our study are shown in Table III.

TABLE III
SUMMARY OF THE SYSTEMS

System	Version	LOC	Classes	
			Total	Defective
ArgoUML	0.10.1	154442	863	49
	0.12	171746	946	47
	0.14	182627	1227	93
	0.16	185335	1185	152
	0.18.1	196505	1249	52
	0.20	186055	1333	127
Eclipse	1.0	1049434	4596	96
	2.0	1471858	5985	163
	2.1.1	1735010	6748	98
	2.1.2	1737345	6750	78
	2.1.3	1740487	6754	149
Rhino	1.4R3	43791	94	66
	1.5R1	68086	124	22
	1.5R3	86937	166	98
	1.5R4	92398	180	35
	1.5R5	92687	181	39
	1.6R1	102511	178	37
	1.6R4	102974	180	138
	1.6R5	79144	124	37

D. Results

1) RQ1: Table IV shows the percentage of the analyzed versions that retained the specific LBS in at least one PC. In Table V we show the percentage of the analyzed versions where each LBS was ranked first. Table VI shows the weight and ranking (in parentheses) of the attributes for ArgoUML v0.16 after the relative threshold is applied.

[4]http://argouml.tigris.org/
[5]http://www.eclipse.org/
[6]http://www.mozilla.org/rhino/

ArgoUML: For all versions of ArgoUML we retained between 11 and 13 principal components that explain at least 95% of the variance. Two LBS attributes were kept in at least one PC in all versions and those are: *inconsistent terms* and *useless types*. Between them, *useless types* was the major contributor of at least one PC in all versions.

Rhino: The number of components that explain at least 95% of the variance for Rhino is the same as for ArgoUML. Five LBS attributes were kept in at least one PC in all versions and those are: *inconsistent terms*, *synonym similar*, *odd grammatical structure*, *overloaded identifiers*, and *meaningless*. As in ArgoUML, one LBS attribute was present as a major contributor in all versions and this is *overloaded identifiers*.

Eclipse: The number of retained PC is between 13 and 14. The six LBS that are present in all versions are: *inconsistent terms*, *odd grammatical structure*, *extreme contraction*, *overloaded identifiers*, *useless types*, and *meaningless*. The majority of them (four) are ranked first: *inconsistent terms*, *extreme contraction*, *overloaded identifiers*, and *meaningless*.

Overall: All LBS were present in more than 50% of the analyzed systems. *inconsistent terms* was present in at least one dimension in all analyzed versions meaning that it is the major LBS attribute that helps to explain a new variability dimension. Another different variability dimension in most cases seems to be captured by *overloaded identifiers* and *useless types*.

2) RQ2: For each evaluation metric, Table VII shows the average values scored by the corresponding model on all types of prediction (same and next version). CK metrics are used to build the prediction models. The values in bold are the best values of the three models considered for the given metrics. For all the systems, SVM scores first for the majority of the evaluation metrics. Hence, we have based our investigation of LBS' contribution to the improvement of fault prediction on SVM.

Table VIII shows the number of versions in which CK plus LBS metrics improve, decrease or keep the prediction unchanged, when compared to CK metrics alone. The last two columns show the net improvement within/across versions and the average delta percentage of LBS plus CK metrics over CK alone for the various evaluation metrics. Positive values of net improvements, for all types of evaluation metrics, indicate that in the majority of the versions CK plus LBS are better predictors than CK alone, while negative values indicate the opposite. A zero net improvement means that both sets of independent variables were found better than the other in an equal number of versions or that they are equal in all versions. For all evaluation metrics except *absolute error*, the same is true for the average delta percentage, which is computed on the average values over all versions of the corresponding system. For *absolute error*, a negative value means that there is a reduction in the amount of error and hence indicates an improvement while the opposite holds for positive values of *absolute error*.

The predictions using CK plus LBS metrics have outperformed those of CK alone in most of the versions of the three

TABLE IV
LBS RETAINED IN THE PRINCIPAL COMPONENTS.

System	Misspelling	Inconsistent Terms	Synonym similar	Odd grammatical structure	Extreme contraction	Overloaded identifiers	Identifier construction	Useless types	Meaningless terms
Eclipse	0.0%	100.0%	40.0%	100.0%	100.0%	100.0%	80.0%	100.0%	100.0%
ArgoUML	66.7%	100.0%	50.0%	66.7%	66.7%	83.3%	83.3%	100.0%	16.7%
Rhino	87.5%	100.0%	100.0%	100.0%	75.0%	100.0%	62.5%	87.5%	100.0%
All	57.9%	100.0%	68.4%	89.5%	78.9%	94.7%	73.7%	94.7%	73.7%

TABLE V
LBS RANKED FIRST IN THE RETAINED PRINCIPAL COMPONENTS.

System	Misspelling	Inconsistent Terms	Synonym similar	Odd grammatical structure	Extreme contraction	Overloaded identifiers	Identifier construction	Useless types	Meaningless terms
Eclipse	0.0%	100.0%	20.0%	20.0%	100.0%	100.0%	80.0%	80.0%	100.0%
ArgoUML	50.0%	83.3%	0.0%	16.7%	33.3%	83.3%	66.7%	100.0%	16.7%
Rhino	0.0%	87.5%	50.0%	0.0%	50.0%	100.0%	62.5%	62.5%	87.5%
Overall	15.8%	89.5%	26.3%	10.5%	57.9%	94.7%	68.4%	78.9%	68.4%

TABLE VI
DETAILED RESULTS OF PCA FOR ARGOUML V0.16.

PC	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
Cumulative proportion	40.9%	51.8%	59.88%	65.54%	71.06%	76.2%	81.02%	85.29%	89.33%	92.91%	95.53%
CBO	**0.275(9)**	0.203	0.35	0.0741	0.0176	0.0994	0.0954	0.11	0.0853	0.0607	0.0953
DIT	0.0311	0.0551	0.123	0.13	**0.772(1)**	0.46	0.0998	0.338	0.0457	0.0665	0.107
LCOM1	**0.281(7)**	0.36	0.0835	0.0277	0.00268	0.0387	0.0503	0.0812	0.184	0.28	0.0641
LCOM2	**0.278(8)**	0.366	0.0879	0.0272	0.00323	0.0382	0.0507	0.0807	0.188	0.282	0.0685
LCOM5	0.111	0.307	0.00385	0.0976	0.206	0.269	0.0478	**0.753(1)**	0.35	0.16	0.0615
LOC	**0.29(5)**	0.15	0.367	0.0276	0.0123	0.0217	0.0729	0.165	0.0668	0.0341	0.134
NAD	0.21	0.101	**0.442(1)**	0.0404	0.0119	0.0138	0.0763	0.0568	0.368	0.0434	**0.604(1)**
NMD	**0.338(1)**	0.0988	0.0846	0.0403	0.0618	0.00984	0.0373	0.0147	0.0764	0.0998	0.108
NOC	0.0205	0.107	0.0854	0.386	0.428	**0.774(1)**	0.00912	0.113	0.132	0.0118	0.0585
RFC	**0.296(4)**	0.176	0.274	0.0458	0.0453	0.00323	0.0397	0.0971	0.0342	0.0189	0.0197
WMC	**0.318(2)**	0.12	0.286	0.0338	0.0116	0.0181	0.0958	0.0996	0.0384	0.0614	0.131
misspelling	0.24	0.201	0.187	0.207	0.255	0.0979	0.0973	0.0529	0.0174	0.211	**0.571(2)**
inconsistent terms	0.205	0.246	0.147	0.0116	0.0484	0.0543	0.383	0.29	0.189	**0.6(1)**	0.178
synonym similar	**0.288(6)**	0.314	0.102	0.00884	0.00461	0.000561	0.155	0.0958	0.0317	0.154	0.241
odd grammatical structure	**0.305(3)**	0.0148	0.28	0.013	0.048	0.0274	0.173	0.0301	0.0203	0.0892	0.152
extreme contraction	0.0772	0.253	0.266	**0.592(1)**	0.276	0.212	0.0624	0.166	0.336	0.288	0.15
overloaded identifiers	0.144	0.0224	0.00102	0.236	0.161	0.00659	**0.802(1)**	0.0575	0.00447	0.467	0.0241
identifier construction	0.14	**0.416(1)**	0.271	0.266	0.0227	0.143	0.0104	0.0399	0.41	0.139	0.299
useless types	0.0413	0.248	0.236	**0.539(2)**	0.0042	0.153	0.304	0.318	**0.561(1)**	0.186	0.00662

systems, when considering both within and across version prediction. For ArgoUML, the prediction on the same versions using CK and LBS together has improved in at least 4 of the 6 versions considered, according to the different evaluation metrics. For Eclipse the improvement observed in all versions is consistently reported by all evaluation metrics. Figure 5 shows the average values of all versions of Eclipse for the evaluation metrics. We observe an important improvement for all metrics except for *accuracy* where the improvement is minor. The evaluation metrics result for Rhino shows that there is improvement in at least half of the versions considered (4 out of 8). The distributions of the evaluation metrics for all systems are shown in Figure 7.

When predicting on the next version, CK plus LBS have been found to be good predictors in the majority of Eclipse's and Rhino's versions by some evaluation metrics; according to other evaluation metrics they are the same as CK alone. Figure 6 contrasts the predictions of the two models for Eclipse. For ArgoUML, negative net improvement values are observed in three of the evaluation metrics while the other three show that there is a net improvement in at least 3 out of the 5 versions predicted. Overall, in both types of predictions, within and across versions, CK plus LBS are better than CK alone in the majority of the versions. This result is confirmed by almost all average delta percentage values shown next to each net improvement. The average delta percentage decreased only in 7 out of the 36 metrics computed for the three systems. Hence, we can answer RQ2 affirmatively.

3) RQ3: Table IX shows the ranked LBS according to their contribution for SVM. Within brackets the median rank across versions is also indicated. The following observations can be made across the different systems: *synonym similar* is in the top five most important LBS for all systems. *inconsistent terms* and *overloaded identifiers* are in the top three for two of the systems. *Inconsistent terms* and *synonym similar* have a median rank at most 11. Finally, *whole part* does not seem to be important for fault prediction.

We also observe that some LBS tend to have a specific contribution for particular systems. For instance, *extreme contraction* is ranked first among all LBS for Eclipse, while *misspelling* is ranked second for Rhino.

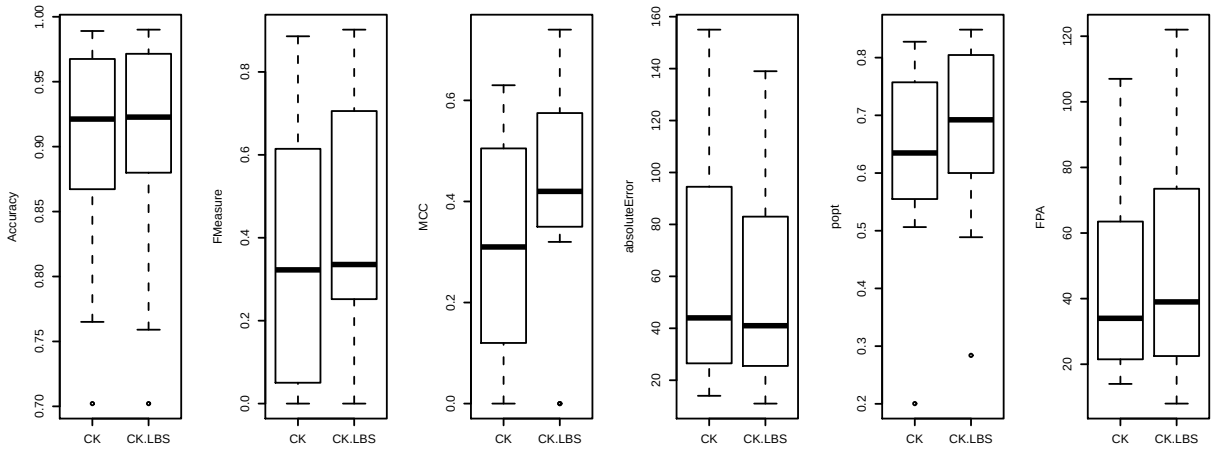

Fig. 7. All systems: Evaluation metrics for same version prediction.

TABLE VII
AVERAGE VALUES OF EACH MODEL WHILE USING THE CK METRICS AS INDEPENDENT VARIABLE

System	Category	Metric	LRM	RF	SVM
ArgoUML	**Rank**	P_{opt}	0.468	0.505	**0.603**
		FPA	38.9	4.91	**45.8**
	Error	E	91.6	88.7	**86.8**
	Classifi.	A	0.922	0.925	**0.927**
		F	0.0797	**0.199**	0.0812
		MCC	0.0991	**0.199**	0.12
Eclipse	**Rank**	P_{opt}	0.458	0.521	**0.637**
		FPA	**67**	0.444	60.8
	Error	E	124	127	**118**
	Classifi.	A	0.98	0.98	**0.981**
		F	0.0101	**0.0985**	0.0439
		MCC	0.0167	**0.139**	0.104
Rhino	**Rank**	P_{opt}	0.528	0.535	**0.568**
		FPA	21.3	16.3	**21.4**
	Error	E	42.8	42.8	**41.2**
	Classifi.	A	0.71	0.71	**0.717**
		F	0.552	0.538	**0.579**
		MCC	0.346	0.336	**0.375**

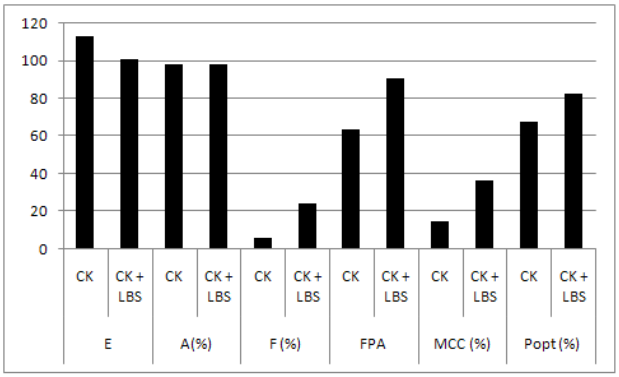

Fig. 5. Eclipse: Average of the evaluation metrics for same version prediction.

TABLE VIII
CK AND CK + LBS PREDICTION CAPABILITY COMPARISON USING SVM

Systems	Predi. version	Category	Metric	Imp.	Dec.	Equal	Net imp.	Avg. delta %
ArgoUML	**Same**	**Error**	E	5	0	1	5	-8.94
		Rank	P_{opt}	5	1	0	4	-5.878
			FPA	5	1	0	4	1.917
		Classifi.	A	5	0	1	5	0.6738
			F	5	0	1	5	81.51
			MCC	5	0	1	5	56.72
	Next	**Error**	E	1	3	1	-2	3.165
		Rank	P_{opt}	2	3	0	-1	4.405
			FPA	4	1	0	3	9.948
		Classifi.	A	1	3	1	-2	-0.2805
			F	4	0	1	4	100
			MCC	4	0	1	4	-1400
Eclipse	**Same**	**Error**	E	5	0	0	5	-11.11
		Rank	P_{opt}	5	0	0	5	22.91
			FPA	5	0	0	5	43.53
		Classifi.	A	5	0	0	5	0.2176
			F	5	0	0	5	314.6
			MCC	5	0	0	5	140.8
	Next	**Error**	E	2	2	0	0	1.212
		Rank	P_{opt}	2	2	0	0	-3.067
			FPA	2	1	1	1	0.8696
		Classifi.	A	2	2	0	0	-0.02364
			F	3	1	0	2	234.3
			MCC	3	1	0	2	200
Rhino	**Same**	**Error**	E	6	1	1	5	-11.27
		Rank	P_{opt}	6	2	0	4	3.233
			FPA	6	0	2	6	3.518
		Classifi.	A	6	1	1	5	2.343
			F	7	0	1	7	8.521
			MCC	6	1	1	5	15.24
	Next	**Error**	E	2	1	2	1	-0.6042
		Rank	P_{opt}	3	2	0	1	3.085
			FPA	4	0	1	4	8.861
		Classifi.	A	2	1	2	1	0.4126
			F	3	0	2	3	3.925
			MCC	3	0	2	3	10.53

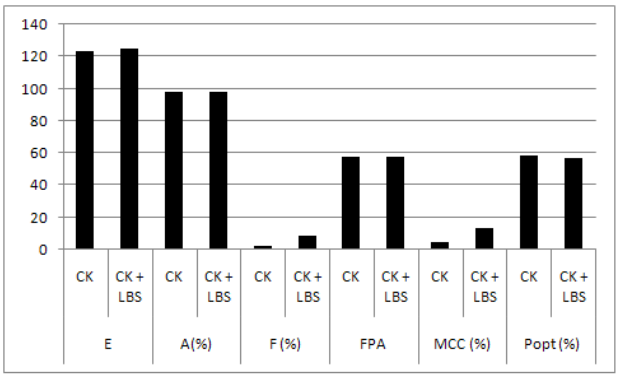

Fig. 6. Eclipse: Average of the evaluation metrics for next version prediction.

TABLE IX
RANKED LBS ACCORDING TO SVM.

ArgoUML	**Rhino**	**Eclipse**
Synonym similar (4)	Odd grammatical structure (6.5)	Extreme contraction (3)
Inconsistent terms (6.5)	Misspelling (7.5)	Overloaded identifiers (4)
Overloaded identifiers (8.5)	Inconsistent terms (10)	Identifier construction (4)
Identifier construction (9.5)	Synonym similar (11)	Useless types (7)
Odd grammatical Structure (10)	Meaningless terms (12)	Synonym similar (8)
Misspelling (10.5)	Identifier construction (12.5)	Odd grammatical structure (8)
Useless types (13)	Extreme contraction (13)	Meaningless terms (10)
Extreme contraction (15.5)	Overloaded identifiers (14)	Inconsistent terms (11)
Meaningless terms (20)	Useless types (17.5)	misspelling (14)
Whole part (20)	Whole part (20)	Whole part (20)

E. Discussion

PCA shows that the majority of LBS (all but three) are major contributors in at least one dimension for more than 50% of the analyzed versions. The strongest percentages are obtained by *inconsistent terms*, *overloaded identifiers*, and *useless types*. The weakest percentages across versions appear to be *odd grammatical structure*, *misspelling*, and *synonym similar*.

We have analyzed three types of prediction models to identify the best model which works with the CK metrics. Of the analyzed models SVM is found to be the best in the majority of the cases (see Table VII). Hence, we have used this model to assess the contribution of LBS to the CK metrics in predicting fault prone classes. The results shown in Table VIII indicate that adding LBS to CK improves the predicting capability of the models. The improvement is observed on almost all types of evaluation metrics used for the two types of predictions used in the three systems, within and across version. This result is also confirmed by the average delta percentage. Of the two types of predictions, the predictions conducted on the same versions using LBS plus CK metrics have shown improvement in more versions than observed in predictions on the next version. For example, in Eclipse LBS plus CK metrics improved the prediction in all versions (5 of 5), while across versions the improvement is observed in at most half of the versions (2 of 4). The difference can be observed by comparing Figures 5 and 6.

The results of RQ3 show that for fault prediction *synonym similar* is in the top five most important LBS. Our findings are consistent with previous research on program identifiers that suggest that identifiers using synonyms lack conciseness and consistency [32]. Overall, *synonym similar*, *inconsistent terms*, and *overloaded identifiers* seem to be in general the most important LBS for fault prediction. We also observe that other LBS are important but specific to the systems, *e.g.*, *extreme contraction* for Eclipse and *misspelling* for Rhino.

VI. THREATS TO VALIDITY

Our study uses the CK metrics considered in [15] and others as a baseline to investigate the contribution of LBS in predicting fault proneness of a class. In the literature, however, there are other metrics which are proposed to achieve the same goal. In our future work, we plan to investigate if LBS are complementary also to these metrics.

To identify LBS, we have used a suite of tools that implement general heuristics that can be configured to accommodate some variability. The three systems considered in our study are developed in different environments and hence are influenced by their respective environments. Using one general configuration for all the systems might affect the results. To handle this threat, we manually explored their documentations, when available, and configured the detectors accordingly.

Different evaluation metrics assess different aspects of prediction models and hence might give different results. To see if our results are consistent across different evaluation metrics, we based our evaluation on selected evaluation metrics which assess different aspects and have been commonly used in recent studies.

The prediction results depend on the used models and their configurations. We used default configurations or configurations used in other studies. Further tuning of the parameters however could change the rankings of the models. The best model from RQ2, SVM, was used with default parameters. Di Martino *et al.* [33] suggest the use of genetic algorithms to select the parameters for further improvement of the results.

In our study, we have considered only three Java systems which limits its generalizability. However, these systems have been selected from different domains and with different size to limit this threat. Besides, they are real world open source programs which are actively evolving.

VII. CONCLUSION AND FUTURE WORK

In this study, we have investigated whether the identifier quality contributes to the fault prediction approaches that use source code structural metrics. We measure the quality of an identifier in terms of the LBS it contains. The results show that in the majority of the cases using LBS with the structural metrics (CK) improves fault prediction. To assess

the improvement, we have used different evaluation metrics that address different aspects of the prediction. The assessment shows that the improvement is consistent in almost all types of evaluation metrics.

Among all LBS, the most important ones are *overloaded identifiers* and *inconsistent terms*. In the majority of the systems, these are the major contributors of at least one retained principal component; they are also the most important contributors for fault prediction. Moreover, for fault prediction *synonym similar* is always among the top five most important LBS. On the other hand, we believe that other LBS, not included in this list, should not be deemed irrelevant, as they become important for specific systems, *e.g.*, *extreme contraction* and *misspelling*.

In our future work, we will investigate whether LBS provide additional contributions to other types of metrics that are used to predict the fault proneness of a class. We also plan to validate our results on more systems, possibly written in different programming languages.

ACKNOWLEDGMENT

The authors are grateful to Khomh *et al.* [29], for sharing their data.

REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[2] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[3] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. IEEE CS Press, 2007.

[4] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. ACM Press, 2009.

[5] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 2005, pp. 284–292.

[6] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE CS Press, 2009, pp. 78–88.

[7] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE CS Press, 2007, pp. 489–498.

[8] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Empirical Software Engineering*, vol. 15, no. 3, pp. 277–295, 2010.

[9] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*. IEEE CS Press, 2010, pp. 31–41.

[10] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[11] F. Deissenbock and M. Pizka, "Concise and consistent naming," in *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE CS Press, 2005.

[12] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE CS Press, 2008, pp. 113–122.

[13] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2009, pp. 31–35.

[14] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Proceedings of Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2009, pp. 95–99.

[15] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Software Engineering*, vol. 16, no. 1, pp. 141–175, 2011.

[16] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[17] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[18] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction." *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[19] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2006, pp. 469–478.

[20] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[21] D. Binkley, H. Feild, D. Lawrie, and M. Pighin, "Software fault prediction using language processing," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 99 –110.

[22] V. Arnaoudova, L. M. Eshkevari, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "Physical and conceptual identifier dispersion: Measures and relation to fault proneness," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2010.

[23] Y. Kamei, S. Matsumoto, A. Monden, K. ichi Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2010, pp. 1–10.

[24] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[25] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. ACM Press, 2011.

[26] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[27] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *Proceedings of the International Symposium on Software Reliability (ISSRE)*. IEEE CS Press, 2007, pp. 215–224.

[28] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

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

[30] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Proceedings of Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2004, pp. 172–181.

[31] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE CS Press, 2011, pp. 125–134.

[32] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE CS Press, 2006, pp. 139–148.

[33] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro, "A genetic algorithm to configure support vector machines for predicting fault-prone components," in *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer-Verlag, 2011, pp. 247–261.