CSIT (November 2014) 2(3):191–205
DOI 10.1007/s40012-014-0057-1

ORIGINAL RESEARCH

Object oriented software metrics threshold values at quantitative acceptable risk level

Satwinder Singh · K. S. Kahlon

Received: 6 February 2014 / Accepted: 19 October 2014 / Published online: 25 November 2014
© CSI Publications 2014

Abstract The metrics can be applied by software maintenance, testing and evolution teams for a variety of purposes. Various research studies have designed metrics models for analyzing the quality of software. However, it is hard to assess the quality of software with a single metrics value. A metrics value alone is meaningless without its threshold values. In the current paper, study derives a threshold metrics value against the bad smell using risk analysis at five different levels. Three versions of Mozilla Firefox were used as a dataset to validate the study. The results show that some metrics have threshold values at various risk levels that are of practical use in predicting faulty classes. Finally one threshold value was selected from the various risk levels (for bad smell) by determining the largest area under receiver operating character curve for faulty classes at corresponding risk levels.

Keywords Object oriented metrics · Encapsulation · ROC curve · Threshold values · Risk level second keyword · More

1 Introduction

With the growth of the software industry, software evolution has become an important concern of software developers. Evolutions of software over many years often lead to obscure systems. It is hard to predict where changes have to be applied on the code and their effects on the system. This will make the maintenance and evolution of software more and more complex and expensive. In order to have efficient evolution and maintenance of code the structure and the design of a system have to be improved. To do this, first it is required to identify the areas for improvement in the code design and the problems faced because of this poorly coded design, i.e. bug origination. Manual identification of the problem of poor code design is not consistent [12]. The support of a tool is required for problem detection. Such a tool would accept input in the form of source code and produce output consisting of estimated flaws in the source code. Refactoring is one of the solutions for improving the source code. Refactoring changes the internal structure of the object oriented code without affecting the overall behavior of the system. Refactoring plays an important role in reengineering and reverse engineering, which makes the code easier for software developers to understand [22]. The process of refactoring is applied in three different stages, namely identification of the problematic area, selection of an appropriate refactoring technique and application of the refactoring technique [31]. Fowler and Beck [22] have defined the 22 bad smells that describe design problems and suggested a number of refactoring techniques that [22] can help in improving the design structure of a code.

The agile community has used the principal of refactoring as a solution to bad code smells. Fowler [22], however, did not suggest any criteria for making the decision to refactor. Software metrics have often been used to assess the quality of the software. Although a number of studies have been done to find a relationship between metrics and bad smell [8, 10, 11, 29, 30], these studies have

S. Singh (✉)
Department of Computer Science Engineering, Baba Banda Singh Bahadur Engineering College, Fatehgarh Sahib, India
e-mail: satwindercse@gmail.com

K. S. Kahlon
Department of Computer Science Engineering, Guru Nanak Dev University, Amritsar, India
e-mail: karanvkahlon@yahoo.com

not exploited the threshold values of the metrics. Threshold value is necessary to generalize the view on different software applications. Proper threshold value is an orientation for the developer in the task of refactoring. With threshold value, the developer and tester can scrutinize the classes to find candidates for refactoring on the basis of smelly classes. A class having metrics value greater than the threshold gives a sign of poor design, it needs the inspection to make a design better. In this work study has derived the metrics threshold values for identification of bad smells using a logistic regression model. The model was first proposed by Bender [13] in the field of epidemiology. This model can easily be reused to derive the threshold values with a given estimated coefficient. Moreover, it has previously been applied in the software metrics field [14]. Study has applied this model on three Firefox versions (1.5, 2.0 and 3.0) to identify the threshold metrics value for identification of bad smells in classes. The set of metrics selected for the study is DIT [2], NOC [2], RFC [2], WMC [2], LCOM [2], Co [1], CBO [2], NOA [37], NOOM [37], NOAM, PuF [23] and EncF [23]. The bad smells considered in the study are Large Method, Long Parameter List, Large Class, Temporary Fields, Shotgun Surgery, Lazy Class, Data Class, Speculative Generality, Middle Man, Feature Envy and Inappropriate Intimacy [22]. The practically effective threshold metrics value for the identification of smelly classes' was selected on the basis of high area under curve [receiver operating character (ROC)]. It was believed that a module's proneness to error is somehow associated with bad design [9, 23, 50]. So, clearly the threshold value of metrics associated with the design principles (bad smells) can also be used to predict faults that will arise in the future. The derived threshold values are validated by identifying the faulty classes for subsequent Firefox versions. Finally, study also compare result with that of a previous study [16] that also selected the fault during the development stage.

2 Literature review

Applying refactoring manually is very difficult. Numerous tools have been developed by different communities (commercial, academic and open source) to assess a bad smell from a code, including Together [17], XRefactor [18], jbuilder [19] (commercial), Eclipse [20] (open source), Columbus [21] (academic). In all these cases refactoring is applied with the interactive participation of the user. The purpose of these tools is to identify the areas of code on which refactoring should be applied.

Webster [24] gives the conceptual, coding and quality assurance view of antipatterns (which are potential causes of bad smells) in programming languages. Later on, Reil [25] defined the heuristic to assess quality of the programming language manually for object oriented design. Fowler and Beck [22] have defined the 22 bad smells, which helps in identifying the area to apply refactoring. Mantyla [12] proposed the classification of these bad smells into six groups. They all have proposed theoretical approaches to defining bad smells. They also did not define the criteria to identify them from the code. Other studies (manual and automatic) have been done to identify them from the code. A manual approach to identifying the design flaws that lead to bad smells was proposed by the Travassos [28]. They use the reading technique to identify the antipatterns; this will not be worthwhile for large industrial projects. Ciupke [29] proposed an approach to identify simple design problems which that occur frequently and locate them with the queries applied on a derived model from the source code. However, he did not address the complex design problems causing the bad smells code. Marinescu [30] has proposed a "detection strategy" for formulating metrics-based rules to check the deviation from good design principles and heuristics. In this case human intervention is required because the detection process is uncertain. Therefore, a detection process creates the uncertainty in the classes. Marinescu [30] did not give any justification for the selection of metrics, threshold and combination of metrics defined in the detection strategies. It is indispensable to know whether the metrics chosen encapsulate the design problem or not. Marinescu [31] also proposed an approach, which did not cover the uncertainty issues, quality analysts' interpretations and context of the programs. Furthermore, Marinescu had did the case studies on small projects and not on industrial projects. Rao and Ready [32] use the design change propagation probability (DCPP) matrices to detect the two antipatterns (shotgun survey and divergent change). A DCPP matrix is a N $\times$ N matrix where N represents the artifact or class. In the matrix, the value at column A, row B represents the probability that a design change in artifact A will require changes in B to preserve the overall functionality. Rao and Ready [32] do not use the threshold values in the detection of the antipatterns, but compare candidates of design defects (source code or class, they called it artifacts in their study) under certain specified conditions. They checked the specified conditions (listed in their study [32]) iteratively and correct the design defects with refactoring techniques representing the bad smell (e.g. move method for shotgun survey). Li et. al. [9] have inspected the relationship between faulty classes and a few bad smells. They have checked the relationship with three versions of Eclipse and concluded that broader studies are needed to validate the results. Current study provides such extensive studies by deriving threshold values for smelly classes (with 11 bad smells) and validate values with the probability of occurrence of faulty classes.

Many research studies have developed the metrics model or used the metrics values to predict the quality of the software both for design deviance [10, 23, 30, 38, 50] and for error proneness [4, 23, 50]. Out of these studies few studies have further focused on error proneness due to lack of design [23, 38, 50]. But few other sole research studies have been done to show the relationship between design deviance and faulty classes [9, 12].

Rosenberg [16] has designed the threshold values for some CK metrics with error data, collected in development phase. Threshold values derived in their study can be used for redesigning. Rosenberg [16] values were tested by Shatnawi et. al. [14] and they determined that it is not find it effective for predicting faulty classes as compared to their own values. Shatnawi [14] derived the threshold values for software metrics for predicting the faulty classes. They were the first one to use the approach of quantitative risk level in software metrics, which was used earlier by Bender [13] in the field of epidemiology.

As a compromise between manual and automatic technique, Dhambri et al. [33] have proposed a visual technique for identification of smelly classes. But this technique is unable to decide when a smelly class is actually a smelly and provides a futilely long list of candidates. Fully automatic technique was also proposed by Lanza et al. [33] but this does not give the threshold values for the detection of smelly classes, either, and it shows uncertainty in the detection of smelly classes which really are smelly classes. None of the above approaches, whether manual or automatic, validate their work with the bugs which is designed for smelly classes. In this paper first step is to design the threshold values for determining smelly classes and then validate it with the bug database. This shows that by improving the understandability with refactoring of a code, the probability of bugs will also be reduced. Very few studies have figured out more than five to six bad smells. Current study, however, has taken into account 11 bad smells and designed the threshold values of metrics based on these bad smells.

3 Data collection

Metrics and a bad smell database of Mozilla Firefox were collected using the Columbus Wrapper Framework tool (academic command prompt version on special request) [21]. It has earlier being used and authenticated by number of research studies [23, 26, 40, 43, 44, 50] done by developer of Columbus tool. The selection of metrics was based on criteria that it shall cover a maximum of properties of object oriented methodology. Another criterion is that the metrics should be measured or extracted by the Columbus tool. The selected set of metrics is DIT [2], NOC [2], RFC [2], WMC [2], LCOM [2], Co [1], CBO [2], NOA [37], NOOM [37], NOAM, PuF [23] and EncF [23]. The metrics are characterized as Information hiding (PuF), Encapsulation(EncF), Class Complexity (WMC), Inheritance (DIT, NOC), Class Size(NOA, NOOM, NOAM), Cohesion (LCOM, Co) and Coupling (RFC, CBO). The criteria for the selection of bad smells were the same as those for metrics. The selected set of bad smell is Large Method, Long Parameter List, Large Class, Temporary Fields, Shotgun Surgery, Lazy Class, Data Class, Speculative Generality, Middle Man, Feature Envy and Inappropriate Intimacy [22]. The Columbus tool compiled the source code and produced the metrics and bad smells for each class. Separate files of metrics and bad smells were generated.

Table 1 Bad smell-affected class count distribution in three versions

Bad smell/version	Firefox 3.0	Firefox 2.0	Firefox 1.5
Large method	775	701	534
Long parameter list	267	263	186
Large class	366	345	282
Temporary fields	123	144	136
Shotgun surgery	45	41	39
Lazy class	163	148	135
Data class	18	17	13
Speculative generality	132	168	35
Middle man	13	12	9
Feature envy	994	958	699
Inappropriate	685	445	337
Intimacy	–	–	–
No. of distinct affected classes	1,865	1,695	1,225
No. of classes in each version	4,971	4,524	3,546

3.1 Bad smell extraction

Bad smell identification in a code will help to refactor the code. Decision of refactoring the module is not simple. A number of people have done work to make a decision on refactoring [5–8] on the basis of metrics models. Research results show that there is a relationship between structural attributes (design metrics) and external quality metrics (bad smell) [7, 11, 12, 23, 50]. Bad smell findings will help the testing team to adjudge some faults which are likely to come in the future due to these bad smells [9, 10]. It also helps the manager to forecast the system for refactoring it and for long term evolution of the projects. Table 1 shows the bad smell count in the classes for three versions. Interested reader can read about the brief description of the extraction of 11 bad smells by Columbus tool from the products white paper [42]. For better statistical analysis, bad smells are divided into six categories [12]. The distribution of the effective smelly classes in first five categories is shown in Table 2.

3.2 The bug data collection

A bug database was used to validate the threshold values selected. The bug database of Firefox versions was collected from the Bugzilla [39], as it was collected by Gyimothy et al. [40]. Bugzilla includes the bugs reported in the lifetime of the Firefox project. Current study, however, only included data collection of bugs that appear in the classes extracted by the Columbus framework. Bugs that do not belong to C+ + source code classes were ignored. Each bug was collected manually and details of thereof was obtained from its patch file (attached to the bug description) to find the affected classes. The patch file contains the information about changes in the source code. This information includes how many lines were deleted from a given line number in the source code file and how many lines were inserted at a given line number in the source code file. By manually analyzing the patch file bug, the associated class was found. For each version concerned, it was looked for a class whose code in the source code overlapped with the code interval of a bug patch file. If such a class was found then the bug count was increased for that class. If one bug affected more than one class then the bug count associated with each class was incremented. The Bugzilla community divided the bugs into the following seven categories:

- Blocker: The error blocked development and/or testing work.
- Critical: The error caused crashes, loss of data, or severe memory leak.
- Major: The error caused a major loss of function.
- Normal: The error was the run of the mill bug-a non-major error.
- Minor: The error caused minor loss of function or other problems that were fixed easily.
- Trivial: The error caused a cosmetic problem like misspelled words or misaligned text.
- Enhancement: The request for enhancement.

To check the metrics threshold value for detecting faulty classes, each class was marked as faulty if at least one fault was found, and no fault, if no fault was found. To check the metrics threshold values for each severity category, each fault was categorized into one of the three aforementioned severity categories: High, Medium and Low. If one class has faults falling into more than one severity category then each fault was listed separately for that class. Table 2 shows the fault distribution of each category for three Firefox versions.

Table 2 Categorised bad smell count distribution for three Firefox versions

Bad smell categories	Bad smell	Firefox		
		Firefox 3.0	Firefox 2.0	Firefox 1.5
Bloater	Large method	878	827	626
	Long parameter list	–	–	–
	Large class	–	–	–
Object-oriented abuser	Temporary fields	113	144	136
Change preventer	Shotgun surgery	45	41	39
Dispensable	Lazy class	301	321	177
	Data class	–	–	–
	Speculative generality	–	–	–
Coupler	Middle man	1,408	1,203	875
	Feature envy	–	–	–
	Inappropriate intimacy	–	–	–

4 Research methodology

In this study the data of three versions (1.5, 2.0, and 3.0) of Mozilla Firefox were taken for analysis. Twelve metrics (DIT, NOC, CBO, RFC, WMC, NOA, NOOM, NOAM, Co, LCOM, PuF and EncF) and 11 bad smells (Large Method, Long Parameter List, Large Class, Temporary Fields, Shotgun Surgery, Lazy Class, Data Class, Speculative Generality, Middle Man, Feature Envy and Inappropriate Intimacy) databases of these three versions of Mozilla Firefox were extracted. Then, bugs from these three versions were collected from the Bugzilla database. For analysis method, regression theory was used by Bender [13] (in epidemiology) and Shatnawi [14] (in software metrics) to find the threshold values. Work started with the univariate binary logistic regresssion (UBR) analysis to find the relationship between bad smells and metrics values. Logistic regression is well suited for testing relationship between categorical variables and one or more continuous categorical and continuous predictor variables. Logistic regression was used in study because dependent variable in the study is dichotomous. For these dependent variables, scientists prefer logistic regression. Also, a logistic function takes input from negative infinity to positive infinity but gives the output in the range of 0–1. Only the significantly associated metrics in UBR analysis were considered for finding the threshold values. Significant association between metrics and bad smell in UBR was tested at a 95 % confidence level i.e. p value <0.05. Study belief is that if the threshold values of metrics using the bad smell is found successfully then the validation of these threshold values is done by predicting the faulty classes. Study validate threshold values first with the binary

analysis and then with multinomial analysis. In binary analysis, the modules is divided into two categories (faulty and no faults) using the shortlisted metrics threshold values from the above analysis. In case of multinomial analysis, faulty classes are further divided into three categories (high, medium and low error). In binary logistic regression, input is x and output function is f(x). The variable x is a measure of the total contribution of all the independent variables used in the model and is known as the logistic. The general Multivariate Logistic Regression model is as follows [27] (univariate binary regression is a special case of it where $n = 1$):

$$\pi(Y = 1|X_1, X_2, \ldots X_n) = \frac{\exp^{(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_n x_n)}}{1+\exp^{(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_n x_n)}} \quad (1)$$

where, β_0 is the intercept and $\beta_i (1 \le i \le n)$ are the regression coefficients; π is the probability of a class being smelly; Y is the dependent variable (a binary variable); $X_i (1 \le i \le n)$ are the independent variables i.e. object-oriented metrics.

In binary analysis derived threshold metrics value prediction accuracy is checked for either a class be faulty or no fault. In case of multinominal analysis, the accuracy of categorical fault (high, medium and low) is checked for prediction with the derived threshold values of each version. The threshold value which has high fault prediction accuracy is selected.

4.1 Threshold model

Logistic regression is used to select the metrics significantly associated with bad smell. Then, from this regression, an analysis threshold model was developed based on Bender's method [13] as it was used by Shatnawi [14]. Bender's proposed method for risk analysis is known as value of an acceptable risk level (VARL). It is formally defined as follows:

$$\text{VARL} = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_0}{1-p_0}\right) - \alpha\right) \quad (2)$$

where β is the metrics regression coefficient value; α is the constant coefficient value; p_0 is the probability of the class being smelly at an acceptable risk level (R.L.).

VARL values of those metrics were measured which were significant in univariate regression analysis.

4.2 Model accuracy

ROC is related to benefit analysis of decision making. It was first developed by engineers in World War II to detect the enemy object. It is also known as signal detection theory [47]. ROC analysis since then is used in different fields like medicine, biometrics, psychology and many other including the machine learning and model accuracy. First application of ROC in machine learning was tested by Spacman [48]. Spacman validated the values of ROC curve for comparing the different classification algorithms. ROC curves are two-dimensional graphs that visually depict the performance and performance trade-off of a classification model [15]. ROC curves were originally designed as tools in communication theory to visually determine optimal operating points for signal discriminators [47]. A ROC curve plots the relationship between sensitivity and specificity. Sensitivity and specificity as per the confusion matrix are defined as follows:

$$\text{Sensitivity} = \text{true positive rate (TPR)} = \frac{TP}{TP+FN}$$

$$\text{Specificity} = \text{true negative rate (TNR)} = \frac{TN}{TN+FP}$$

The area under the curve for the ROC ranges between 0 and 1. The general rule to measure the classification performance of ROC curve is as follows [27]:

1. AUC $<$ 0.5 means no discrimination classification;
2. 0.5 $<$ AUC $<$ 0.6 means poor classification;
3. 0.6 $<$ AUC $<$ 0.7 means good classification;
4. 0.7 $<$ AUC $<$ 0.8 means acceptable classification;
5. 0.8 $<$ AUC $<$ 0.9 means excellent classification;
6. AUC $<$ 0.9 means outstanding classification.

Area under curve (AUC) is the probability that classifier ranks the randomly chosen positive instance higher than a negative instance [45]. Machine learning community mostly used the ROC AUC for the model comparison to select the optimal model from suboptimal models [46]. To measure the accuracy of metrics threshold value, the ROC curve analysis was used. From the ROC graph, the area under the curve (AUC) is used to check the performance of threshold values. Threshold values are valid for error prediction model if AUC values fall in the acceptable range. AUC values that lie below this range are not acceptable and the threshold values are not valid for smelly or faulty class prediction. ROC curve analysis is used for the skewed and unbalanced data. This was suitable for Mozilla Firefox data which is skewed and unequally distributed at par with the Eclipse data used by Shatnawi [4]. If the ROC area under the curve (AUC) falls in the acceptable range, then first the corresponding metrics value for each Firefox version will be shortlisted as a threshold metrics value to predict the smelly classes. Second, the shortlisted threshold values will be used to predict the faulty classes and categorized faulty classes again by the criteria of acceptable range for AUC. Threshold values derived from the relationship between bad smells and metrics are used for faulty class prediction because earlier research studies [9, 10, 23, 50] have shown that a relationship exists between bad smells and faulty

classes. So, classes not properly checked for bad smells or improper design may result in faulty classes (over time).

5 Hypothesis of study

There are two main aims of this study: first, to derive the threshold metrics values for identification of smelly classes, and second, to use statistically significant threshold metrics derived from first context to identify the fault and its category. The following are the Null Hypothesis of study:

1. There are no practical and effective threshold values for the OO metrics that can predict smelly classes during the design.
2. A practically significant threshold metrics value that predicts the smelly classes will not predict the faulty classes or separate the classes into two modules (faulty and no-fault).
3. A practically significant threshold metrics value that predicts the smelly classes will not predict the fault categories (low, medium and high) and no fault.

6 Metrics UBR analysis

In the univariate binary logistic regression (UBR) analysis, significant metrics for predicting smelly classes were selected. Analysis was done at a 95 % (p value <0.05) confidence level. Metrics with a p value greater than 0.05 are not used for the next step of threshold analysis. Table 3 shows the UBR analysis of three versions of Mozilla Firefox.

It was noticed from the UBR analysis that all the selected metrics are significant predictors of smelly classes in all the three versions. Therefore, threshold values of all these metrics can be calculated with VARL analysis.

7 Metrics threshold values analysis

Threshold metrics values of all the above selected metrics are calculated using (2). Table 4 shows the threshold values at five risk levels (i.e. 0.05, 0.065, 0.075, 0.1, and 0.125). In some cases study did not get any useful threshold values. Such values are not mentioned in Table 4 and its corresponding AUC column is marked with N/A. There was no threshold values for LCOM and Co metrics at five supposed risk levels (R.L.) because the threshold values deduced with VARL analysis are less than the metrics minimum value or more than the maximum. Table 5 shows many metrics having potential threshold values with AUC more than 0.700. Table 5 also shows the AUC for binary fault categories of selected metrics. There is a difference in the threshold value in each version for each risk level. So, one threshold metrics value was selected for each version with the highest AUC and in the acceptable range (AUC $>$ 0.700). Analysis did not find practical threshold values for PuF, NOC, DIT, NOA (for 2.0 and 3.0 versions), NOOM, NOAM. In this case, the AUC for metrics (PuF, NOC, DIT, NOA, NOOM and NOAM) lie below the acceptable range (AUC $<$ 0.700). Shatnawi et al. [14] also did not find practical threshold values for the LCOM, NOC, DIT, NOA, NOOM and NOAM. They did not look for the Co and PuF metrics threshold values. Therefore, threshold values for CBO, RFC, WMC, NOA (for 1.5 versions) and EncF were practically useful, valid and in the acceptable range (i.e. AUC $>$ 0.7). Selected metrics with their threshold values for each version are shown in Table 6. From the table it is observed that CBO and RFC have approximately the same metrics threshold for each version. There is a difference in metrics value for EncF and WMC for each version. Table 6 also shows that the AUC is higher with

Table 3 Univariate binary regression analysis

Metrics	Firefox 3.0		Firefox 2.0		Firefox 1.5	
	B	*p* value	*B*	*p* value	*B*	*p* value
NOC	−0.166	0.000	−0.268	0.000	−0.233	0.000
DIT	0.185	0.000	0.161	0.000	0.178	0.000
CBO	0.169	0.000	0.157	0.000	0.156	0.000
RFC	0.038	0.000	0.044	0.000	0.041	0.000
WMC	0.048	0.000	0.048	0.000	0.047	0.000
LCOM	0.001	0.000	0.002	0.000	0.001	0.000
Co	0.473	0.000	0.665	0.000	0.613	0.000
PuF	−3.121	0.000	−3.486	0.000	−5.503	0.000
EncF	1.583	0.000	2.405	0.000	2.765	0.000
NOA	0.095	0.000	0.096	0.000	0.104	0.000
NOAM	0.042	0.000	0.051	0.000	0.004	0.000
NOOM	0.003	0.000	0.003	0.000	0.003	0.000

Table 4 VARL metrics threshold values of Mozilla Firefox

Metrics	Version	Area under curve (for bad smell)					Threshold values (for bad smell)				
		0.05	0.065	0.075	0.1	0.125	0.05	0.065	0.075	0.1	0.125
CBO	1.5	0.728	0.751	0.748	0.760	0.755	6	7	7	8	9
	2.0	0.662	0.709	0.731	0.743	0.743	4	5	6	7	7
	3.0	0.727	0.727	0.753	0.753	0.754	5	5	6	6	7
RFC	1.5	0.711	0.733	0.742	0.750	0.754	8	11	13	16	19
	2.0	0.650	0.706	0.715	0.728	0.735	5	8	9	12	15
	3.0	0.524	0.658	0.703	0.726	0.726	2	5	7	11	14
WMC	1.5	0.787	0.791	0.790	0.784	0.784	12	15	16	19	21
	2.0	0.777	0.777	0.778	0.777	0.769	9	11	13	16	18
	3.0	0.787	0.774	0.774	0.768	0.765	8	10	12	15	17
PuF	1.5	0.257	0.263	0.272	0.294	0.305	0.993	0.971	0.958	0.934	0.914
	2.0	N/A	N/A	N/A	0.281	0.302	–	–	–	0.982	0.951
	3.0	N/A	N/A	N/A	0.297	0.308	–	–	–	0.994	.959
EncF	1.5	N/A	N/A	N/A	0.722	0.700	–	–	–	0.030	.069
	2.0	N/A	N/A	N/A	N/A	0.700	–	–	–	–	.018
	3.0	N/A	N/A	N/A	N/A	0.700	–	–	–	–	.009
NOC	1.5	0.487	0.480	0.480	0.464	0.464	4	3	3	2	2
	2.0	0.486	0.478	0.478	0.463	0.463	4	3	3	2	2
	3.0	0.482	0.482	0.468	0.468	0.468	3	3	2	2	2
DIT	1.5	N/A	N/A	N/A	0.457	0.457	–	–	–	1	1
	2.0	N/A	N/A	N/A	N/A	0.466	–	–	–	–	–
	3.0	N/A	N/A	N/A	N/A	0.461	–	–	–	–	1
NOA	1.5	N/A	N/A	N/A	0.741	0.700	–	–	–	3	4
	2.0	N/A	N/A	N/A	N/A	0.684	–	–	–	–	3
	3.0	N/A	N/A	N/A	N/A	0.665	–	–	–	–	2
NOAM	1.5	N/A	N/A	N/A	N/A	0.677	–	–	–	–	9
	2.0	N/A	N/A	N/A	0.622	0.645	–	–	–	4	6
	3.0	N/A	N/A	N/A	0.521	0.639	–	–	–	2	5
NOOM	1.5	N/A	N/A	N/A	N/A	0.618	–	–	–	–	15

threshold values devised by the study than with threshold values designed by Shatnawi et. al. [14]. To get the more practical and useful threshold values for each selected metrics, these values were checked for their predictions of faulty classes. Positive prediction of faulty classes with these threshold values shows that a bad smell will affect the long-term evolution or maintenance of the software product. Threshold values derived from this analysis will help the testing and development teams to identify the classes for refactoring. Metrics threshold values could bring the information such as when a class has greater metrics value than the threshold it is a sign of poor design and it needs an inspection to improve the design. These values will further help to predict faulty classes, as shown in previous studies [9, 23, 50]. Various techniques [22] can be applied to improve the code design after metrics reach the threshold values.

8 Effectiveness of metrics threshold values

Proposed threshold value in Table 6 is need to be checked for the accuracy or effectiveness in predicting the faulty classes. Metrics values were evaluated by predicting the faulty classes in three releases of Firefox (1.5, 2.0 and 3.0). Metrics were evaluated with two validation criteria: binary categorization and multinomial categorization. First, the threshold metrics value tested for whether a class was faulty or not. The result of above shortlisted metrics for predicting faulty categories (binary categorisation) was shown in Table 7. Second, metrics values were used to predict the fault category (high, medium, low impact or no fault). In the second case, the shortlisted threshold value (in Table 6) was checked for each fault category against the no fault categories. Shatnawi [3] also dropped the low impact categories of faults before (combining the high and medium categories

Table 5 VARL metrics threshold values of Mozilla Firefox

Metrics	Version	Area under curve (for bad smell)					Area under curve (for faulty classes)				
		0.05	0.065	0.075	0.1	0.125	0.05	0.065	0.075	0.1	0.125
CBO	1.5	0.728	0.748	0.748	0.760	0.755	0.652	0.683	0.687	0.695	0.711
	2.0	0.662	0.709	0.731	0.743	0.743	0.651	0.692	0.713	0.731	0.731
	3.0	0.727	0.727	0.753	0.753	0.754	0.634	0.685	0.724	0.746	0.747
RFC	1.5	0.711	0.733	0.742	0.750	0.754	0.621	0.643	0.653	0.675	0.696
	2.0	0.650	0.706	0.715	0.728	0.735	N/A	0.613	0.603	0.566	0.580
	3.0	0.524	0.658	0.703	0.726	0.726	N/A	N/A	0.653	0.710	0.743
WMC	1.5	0.787	0.791	0.790	0.784	0.784	0.671	0.677	0.682	0.673	0.683
	2.0	0.777	0.777	0.778	0.777	0.769	0.640	0.632	0.588	0.589	0.595
	3.0	0.787	0.774	0.774	0.768	0.765	0.721	0.736	0.735	0.731	0.742
EncF	1.5	N/A	N/A	N/A	0.722	0.700	N/A	N/A	N/A	0.620	0.646
	2.0	N/A	N/A	N/A	N/A	0.700	N/A	N/A	N/A	N/A	0.677
	3.0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
NOA	1.5	N/A	N/A	N/A	0.741	0.700	N/A	N/A	N/A	0.674	0.672

Table 6 Selected threshold metrics value for highest AUC for bad smell identification

Metrics	Ver.	Threshold value	R.L. (p_0)	AUC	Threshold from [14]	AUC
CBO	1.5	8	0.1	0.760	9	0.743
	2.0	7	0.1	0.743	9	0.726
	3.0	7	0.125	0.754	9	0.721
RFC	1.5	19	0.125	0.754	40	0.678
	2.0	15	0.125	0.735	40	0.679
	3.0	14	0.125	0.726	40	0.653
WMC	1.5	14	0.065	0.791	20	0.781
	2.0	13	0.075	0.778	20	0.764
	3.0	8	0.05	0.787	20	0.754
EncF	1.5	0.030	0.1	0.722	N/A	N/A
	2.0	0.018	0.125	0.700	N/A	N/A
	3.0	–	–	–	N/A	N/A
NOA	1.5	3	0.1	0.741	N/A	N/A

fault) deriving the threshold metrics values for detecting faulty classes. According to Shatnawi [3], low impact category faults were not able to predict the faulty classes. So, it was expected that threshold values derived from bad smell design would predict the high or medium impact error categories. These categories included the errors due to shortcomings during development. If these threshold values predict the faulty classes or categorized faulty classes, which will indicate that controlling bad smell might successfully control errors or bugs. If AUC falls in the acceptable range, then these categorized faults can be predicted using metrics threshold values designed for the smelly classes. Table 8 shows the AUC for different fault categories. Table 8 shows that the High and Medium categories have practical threshold values with AUC $>$ 0.700. In some cases Low categories also have practical threshold values. Only the new metric EncF does not show the practical values with 3.0 versions. So exclude this value for further analysis. Finding the practical threshold values for the High and Medium categories indicates that if proposed methodology is able to predict the bad smells, then it can predict the bugs which are likely to come eventually. One threshold value is need to be selected for each of the selected metrics from Table 8. The metrics value were selected as per its efficiency in detecting the corrective faulty classes. This can be achieved by measuring the sensitivity and specificity of each metrics value against the faulty classes (Fig. 1).

There are two AUC categories, High and Medium, for each metric. These two error categories were combined into one and then recalculated the AUC for each of the selected metrics of Table 8.

Table 9 shows the result of AUC after combining the fault categories. It was noticed that after combining the fault categories, EncF for version 2.0 and 3.0 still did not have AUC in the acceptable range. But the sensitivity in these two cases is nearly 100 %. Sensitivity in all other cases is also 100 % or 1.0, indicating that these threshold values measure the faulty classes accurately for this Firefox data. So, analysis cannot select the threshold metrics value only on the basis of sensitivity of the selected metrics as Shatnawi et al. [3] did. Therefore, metrics values are selected on the basis of higher AUC: The higher the AUC, the more effective and practical is the threshold value.

On the basis of higher AUC criteria, when the threshold value of EncF derived from version 1.5 (i.e. 0.030) is applied on the 2.0 and 3.0 it is seen that the AUC is in the acceptable range. In other cases for CBO and RFC the

Table 7 AUC for fault binary categorizations of classes

Metrics	Version	Threshold value	Risk level (p_0)	Area under curve
CBO	1.5	8	0.1	0.695
	2.0	7	0.1	0.731
	3.0	7	0.125	0.747
RFC	1.5	19	0.125	0.696
	2.0	15	0.125	0.580
	3.0	14	0.125	0.743
WMC	1.5	14	0.065	0.676
	2.0	13	0.075	0.588
	3.0	8	0.05	0.721
EncF	1.5	0.030	0.1	0.620
	2.0	0.018	0.125	0.677
	3.0	–	–	–
NOA	1.5	3	0.1	0.640

Table 8 AUC for fault categorization

Metrics	Ver.	Threshold value	R.L. (p_0)	Area under curve		
				High	Medium	Low
CBO	1.5	8	0.1	0.743	0.739	0.635
	2.0	7	0.1	0.777	0.777	0.685
	3.0	7	0.1	0.787	0.787	0.744
RFC	1.5	19	0.125	0.759	0.725	0.655
	2.0	15	0.125	0.777	0.777	0.564
	3.0	14	0.125	0.806	0.806	0.754
WMC	1.5	14	0.065	0.796	0.795	0.629
	2.0	13	0.075	0.772	0.772	0.567
	3.0	8	0.05	0.744	0.744	0.718
EncF	1.5	0.030	0.125	0.706	0.706	0.716
	2.0	0.018	0.125	0.700	0.636	0.681
NOA	1.5	3	0.1	0.722	0.717	0.597

	p' (Predicted)	n' (Predicted)
p (Actual)	True Positive	False Negative
n (Actual)	False Positive	True Negative

Fig. 1 Confusion matrix

Table 9 AUC after combining high and medium categories fault

Metrics	Ver.	Threshold value	R.L. (p_0)	Combined AUC	Sensitivity
CBO	1.5	8	0.1	0.803	1.00
	2.0	7	0.1	0.777	1.00
	3.0	7	0.1	0.787	1.00
RCF	1.5	19	0.125	0.815	1.00
	2.0	15	0.125	0.777	1.00
	3.0	14	0.125	0.743	1.00
WMC	1.5	14	0.065	0.796	1.00
	2.0	13	0.075	0.772	1.00
	3.0	8	0.05	0.744	1.00
EncF	1.5	0.030	0.1	0.732	1.00
	2.0	0.018	0.125	0.643	0.889
NOA	1.5	3	0.1	0.781	1.00

Table 10 Shortlisted threshold values

Metrics	Thresh values	AUC
CBO	8	0.803
RFC	19	0.815
WMC	14	0.796
EncF	0.030	0.732
NOA	3	0.781

Table 11 Sensitivity measure for metrics threshold values

Metrics	Threshold value	Sensitivity	Rosenberg threshold	Senstivity
CBO	8	1.00	5	1.00
RFC	19	1.00	100	0.444
WMC	14	1.00	100	0.444
EncF	0.030	1.00	N/A	N/A
NOA	3	0.889	N/A	N/A

Table 12 Comparison of AUC with Rosenberg threshold values

Metrics	Proposed threshold values			Rosenberg threshold values		
	Threshold value	AUC ver. 2.0	AUC ver. 3.0	Threshold value	AUC ver. 2.0	AUC ver. 3.0
CBO	8	0.799	0.767	5	0.720	0.717
RFC	19	0.814	0.831	100	0.695	0.802
WMC	14	0.779	0.789	100	0.674	0.712
EncF	0.030	0.693	0.700	N/A	N/A	N/A
NOA	0.724	0.793	0.731	N/A	N/A	N/A

threshold values for each version is nearly the same. In the case of WMC for version 1.5 and 2.0, the values derived is nearly the same; the only difference came in the case of

version 3.0. Table 10 shows the final threshold values on the basis of area under ROC curve. Table 11 shows the sensitivity measure comparison with the Rosenberg's threshold values (applied on version 1.5). In each case sensitivity measures is more with proposed threshold values. Comparisons with other versions are shown in Table 12 and corresponding ROC curves are shown in Fig. 2.

9 Result and discussion

The threshold values derived in this study will help to identify the smelly classes and, furthermore, to predict the faulty classes. In addition, smelly classes can be improved with various refactoring techniques [22]. So, metrics threshold values in this study will help during maintenance to identify the area or class needing refactoring (which helps in better understanding [22]) and this will further reduce the maintenance cost. The result shows the practical threshold values only for CBO, RFC, WMC, EncF and NOA metrics for predicting the smelly classes in the three releases of Mozilla Firefox. On the other hand, study found that the threshold metrics values derived to predict smelly classes were not able to predict binary fault categorization of classes (i.e. class is faulty or not). But, threshold metrics values (CBO, RFC, WMC, EncF and NOA) present a more granular view by predicting the two categories of the fault i.e. high and medium impact. So, the first and third hypotheses of the current study were rejected and the second hypothesis was accepted. Study includes the discussion of results in comparison to previous studies done by Rosenberg [16]. It was noticed that with the exception of CBO, the values proposed by [16] are larger than those propose in the study. it was assumed that the probability of error occurrence is higher in the development phase than the release phase. So, it was surprising to see the difference with Rosenberg's metrics threshold values because he had collected the errors during the development phase. Current study compared its results with Rosenberg's by calculating the AUC. The comparison was conducted by applying proposed threshold values and threshold values of Rosenberg [16] over Firefox version 2.0, version 3.0 and Mozilla SeaMonkey 1.0.1 in Table 12 and Fig. 2. The AUC values with proposed threshold metrics are higher in each case than with those from [16], study validate the effectiveness of threshold value by identifying the faulty classes (after combining high and medium impact bugs) of version 2.0 and 3.0 because all the threshold values are derived from version 1.5. The results show the accuracy with $AUC > 0.70$ (acceptable range). The results were not checked with Shatnawi's [14] proposed values, which are valid only for the Eclipse system. He himself checked the validity of his metrics with other systems (i.e. Rhino and Mozilla) and found that the threshold values were not valid for these systems.

10 Threshold for other system

To generalise the proposed model, current study check the threshold effect on two other systems. One system was from within company (Mozila community i.e. SeaMonkey 1.0.1) and another was from cross company (Licq 1.3.8).[1] Bug database of SeaMonkey was collected from the Bugzilla community whereas for Licq bug database was collected from the Github[2] community. Both the system were developed in the C+ +. SeaMonkey has 3,000+ classes whereas Licq 1.3.8 is smaller system with 280 class. As it was seen from the results of various studies [51–54] that within company prediction model results outperform over the cross company results. Further, [9] and [14] in their study also validate their model in cross company projects and did not find practical threshold values. Model designed in their study were validated in cross company projects by designing the new threshold metrics for the projects. So, new metrics threshold values were designed in the study for Licq with the VARL analysis as in the case of Firefox. Significance of the traditional and proposed metrics for Licq was checked in the current study as predictor of smelly classes. Significance was checked with the UBR logistic regression, as it was done with the Firefox. VARL at five risk level was calculated for the significant metrics for Licq. Results of shortlisted metrics (CBO, RFC, WMC and EncF) on the basis of maximum AUC for predicting smelly classes is shown in Table 13. These selected values were checked for predicting smelly and faulty classes. Further, the results were also compared with the [16] designed threshold values (Table 14).

Threshold values were also tested for predicting wether the class is faulty or not in SeaMonkey and Licq. Results for the case study of SeaMonkey and Licq are tabulated in Table 15. In case of SeaMonkey, AUC was calculated after combining high and medium category faults. Due to non-availability of severity level database for Licq, only binary categorisation prediction accuracy was checked with AUC values. Result shows that there was an improvement in prediction accuracy with the proposed threshold values over the Rosenberg values. Results for SeaMonkey and Licq are similar to Firefox data as shown in Table 7. As in case of three Firefox versions, prediction probability of binary categorisation of fault is not in the acceptable range with Licq data. ROC graphs for the comparative study of threshold

[1] http://www.licq.org.

[2] https://github.com/licq-im/licq.

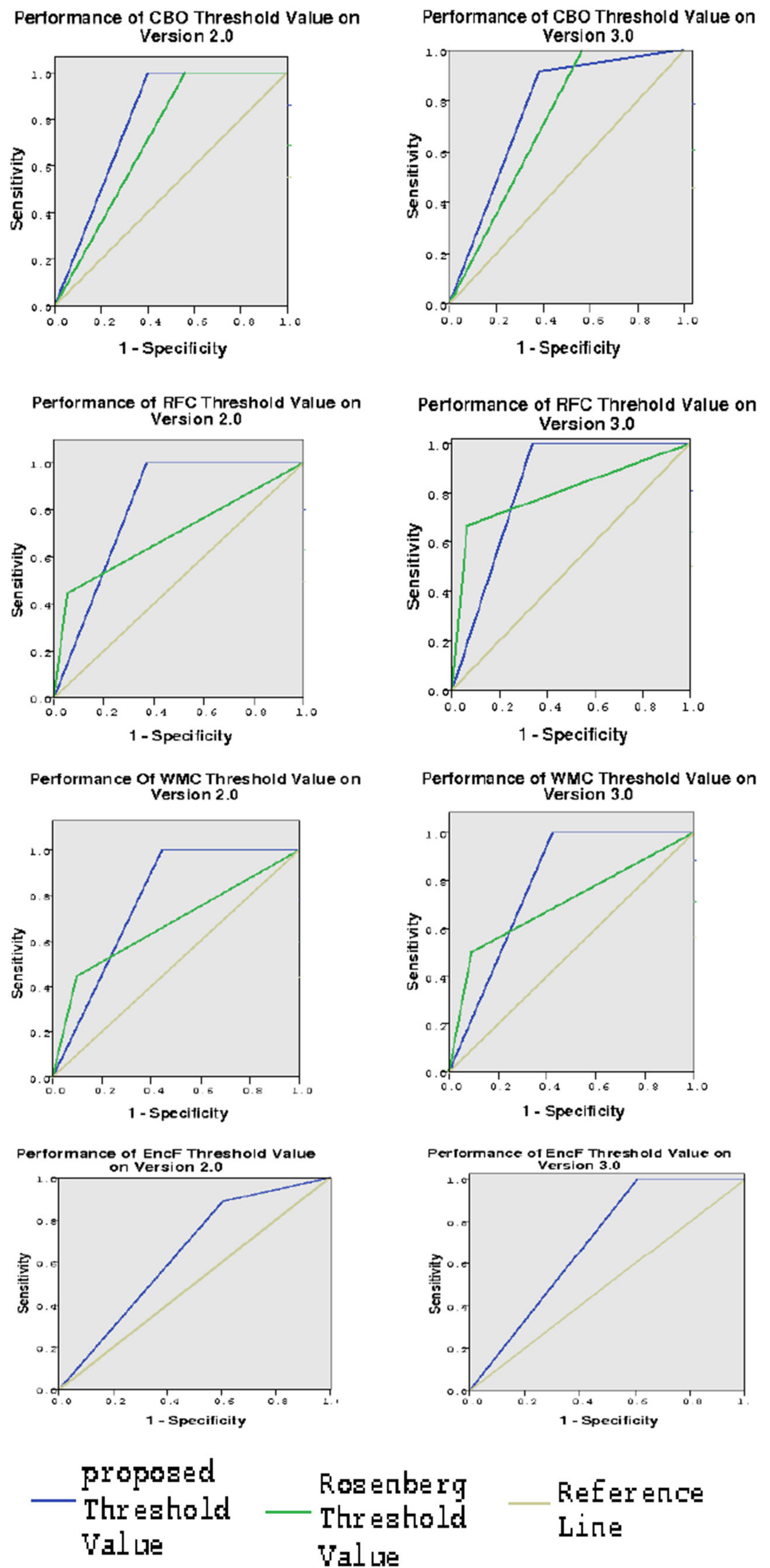

Fig. 2 ROC curves of software metrics for predicting faulty classes (after combining high and medium category faults) in ver. 2.0 and ver. 3.0

Table 13 Selected metrics threshold values of Licq 1.3.8 after VARL analysis

Metrics	Threshold value	Risk factor	AUC for smelly classes
CBO	7	0.125	0.702
RFC	4	0.125	0.570
WMC	7	0.125	0.707
EncF	0.208	0.125	0.544

values over Mozila SeaMonkey 1.0.1 and Licq are shown in Fig. 3.These results can be applied to similar types of object oriented systems, which are available as open source and contributed by many volunteer programmers.

10.1 Threats to validity

Study found a significant association between metrics and bad smell and further with different categories of faulty classes after the system were released. Some of the limitations of research are as follows:

1. Error data of Mozilla Firefox and Mozilla SeaMonkey was collected from the Bugzilla database only. Any error which was not logged was not considered. Study makes no claim about the errors that were not discovered or fixed in the life of these systems.
2. Study does not generalize these results arbitrarily to any type of error classifications. This classification was subjective and it was dependent on how the developer originally used it.

11 Conclusion

Threshold values derived for bad smells are verified with prediction of faulty classes. Tables 8 and 9 emphasized that predicted threshold values are practically significant for detecting the faulty classes. Even when applied over the subsequent and Mozilla SeaMonkey versions, these values predicted the smelly classes and faulty classes significantly. Whereas in case of Licq AUC for only smelly classes prediction is in acceptable range. The results show that refactoring of modules helps to identify the smelly area during maintenance level besides improving the understandability [22, 49]. Further, by predicting the faulty classes, refactoring may also reduce the faults after the release of the system. On the basis of these results, the present study will not suggest that refactoring always control bugs after release. Results in this study have shown the AUC in the range of 0.700 to 0.814, which is only the acceptable range, not the excellent range. These results can be used to predict the errors at the time when maintenance is ongoing. We recommend further study for these hypotheses with some other statistical techniques including power law or machine learning models. To validate this research, more experiments need to be done with different

Table 14 AUC of bad smell for Mozilla SeaMonkey 1.0.1. and Licq 1.3.8

Metrics	Proposed threshold values				Rosenberg threshold values		
	Threshold Values	AUC for SeaMonkey	Threshold values	AUC for Licq	Threshold values	AUC for SeaMonkey	AUC for Licq
CBO	8	0.766	7	0.702	5	0.720	0.616
RFC	19	0.758	4	0.570	100	0.570	0.523
WMC	14	0.795	7	0.707	100	0.643	0.541
EncF	0.030	0.739	0.208	0.544	N/A	N/A	N/A
NOA	3	0.717	N/A	N/A	N/A	N/A	N/A

Table 15 AUC of faulty classes for Mozilla SeaMonkey 1.0.1 and Licq 1.3.8

Metrics	Proposed threshold values				Rosenberg threshold values		
	Threshold values	AUC for SeaMonkey	Threshold values	AUC for Licq	Threshold values	AUC for SeaMonkey	AUC for Licq
CBO	8	0.700	7	0.669	5	0.618	606
RFC	19	0.804	4	0.618	100	0.668	523
WMC	14	0.789	7	0.674	100	0.646	528
EncF	0.030	0.652	0.208	0.610	N/A	N/A	N/A
NOA	3	0.731	N/A	N/A	N/A	N/A	N/A

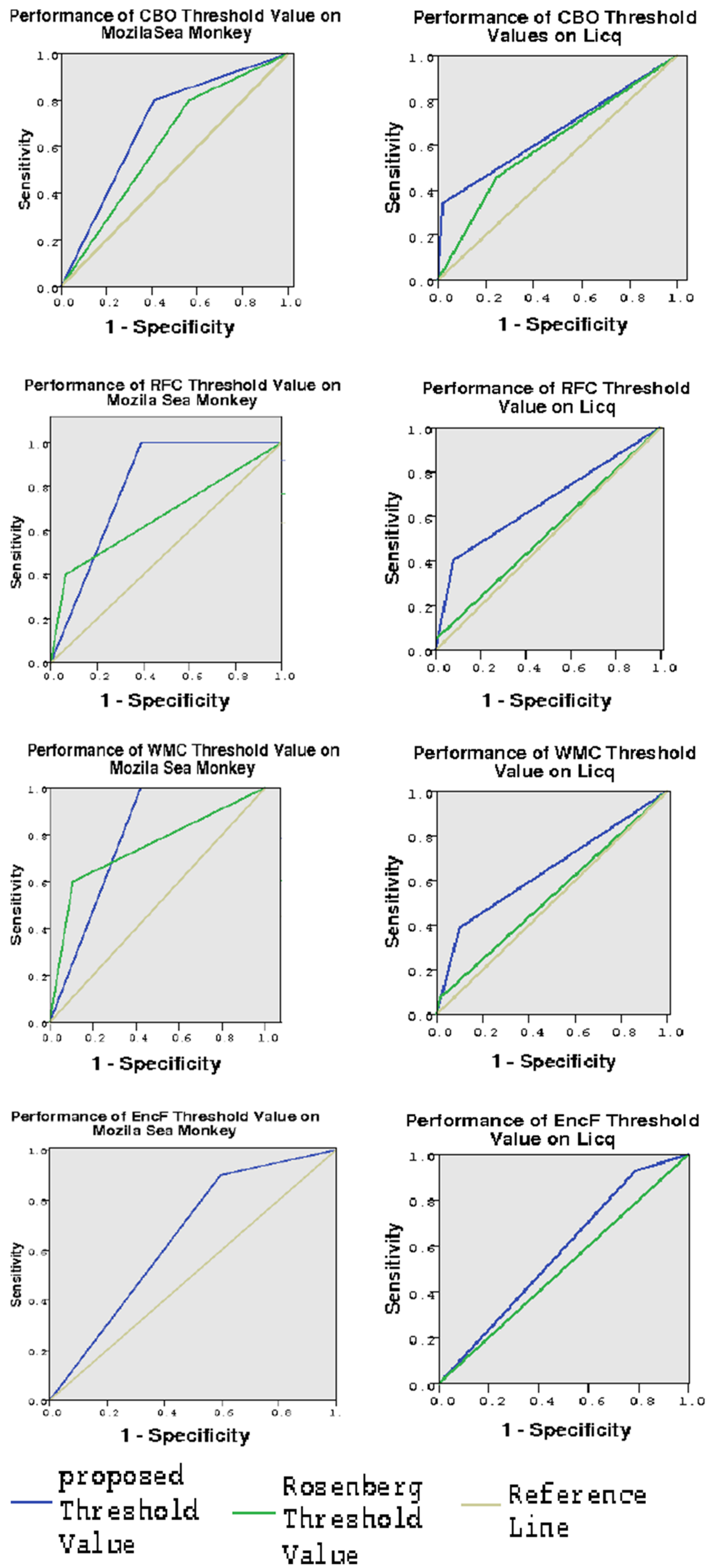

Fig. 3 ROC curves of software metrics for predicting faulty classes in Mozila SeaMonkey 1.0.1 and Licq 1.3.8

object-oriented programming languages. Moreover, this study was carried out on an open source system and needs to be done on professionally built applications.

References

1. Hitz M, Montazeri B (1995) Measuring coupling and cohesion in object oriented systems. In: International symposium on applied corporate computing, Mexico, pp 25–27
2. Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
3. Shatnawi R, Li W (2010) Finding software metrics threshold values using ROC. J Softw Maint Evol Res Pract 22:1–16
4. Shatnawi R, Li W (2008) The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. J Syst Softw 81(11):1868–1882
5. Marticorena R, Lopez C, Crespo Y (2006) Extending a taxonomy of bad code smells with metrics. WOOR'06, Nantes
6. Bravo FM (2003) A logic meta-programming framework for supporting the refactoring process. PhD thesis. Vrije Universiteit Brussel, Belgium
7. Briand L, Arisholm E, Counsell S, Houdek F, Thevenod-Fosse P (1999) Empirical studies of object-oriented artifacts. Methods, and processes: state of the art and future directions. Empir Softw Eng 4(4):387–404
8. Simon F, Steinbruckner F, Lewerentz C (2001) Metrics based refactoring. In Proceedings of the fifth European conference on software maintenance and reengineering (CSMR'01). IEEE CS, Los Alamitos, CA, pp 30–36
9. Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128
10. Richard CG (2003) Software remodeling: improving design and implementation quality using audits. Metrics and refactoring in Borland Together Control Centre, a Borland white paper
11. Marinescu R (2001) Detecting design flaws via metrics in object-oriented systems. In Proceedings of the TOOLS, USA 39, Santa Barbara, USA
12. Mäntylä MV, Lassenius C (2006) Subjective evaluation of software evolvability using code smells: an empirical study. Empir Softw Eng 11(3):395–431
13. Bender R (1999) Quantitative risk assessment in epidemiological studies investigating threshold effects. Biometr J 41(3):305–319
14. Shatnawi R (2010) A quantitative investigation of the acceptable risk levels of OO metrics in open-source systems. IEEE Trans Softw Eng 36(2):216–225
15. Fawcett T (2004) ROC graphs: notes and practical considerations for researchers. Machine learning, pp 31–35
16. Rosenberg L (1997) Metrics for object-oriented environment. Proceedings of the EFAITP/AIE third annual software metrics conference
17. Together tool: http://www.borland.com/us/products/together/
18. XRefactor tool: http://www.xref.sk/xrefactory/main.html
19. jbuilder tool: http://www.embarcadero.com/products/jbuilder
20. Eclipse http://www.eclipse.org
21. Columbus framework: http://www.frontendart.com
22. Martin F (2000) Refactoring: improving the design of existing code. Addison–Wisely, Reading, MA
23. Singh S, Kahlon KS (2011) Effectiveness of encapsulation and OO metrics to refactor code and identify error prone classes using bad smells. ACM SIGSOFT SEN 36(5):1–11
24. Webster BF (1995) Pitfalls of object oriented development, 1st edn. M and T books, New York
25. Riel AJ (1996) Object-oriented design heuristics. Addison–Wesley, Boston, MA
26. Ferenc R, Besezedes A, Gyimothy T (2004) Extracting fault with columbus from C+ code. In: Proceedings of 8th European conference on software maintenance and reengineering, Tampere, Finland, March 24–26. IEEE Computer Society
27. Hosmer D, Lemshow S (2000) Applied logistic regression, 2nd edn. Wiley-Interscience, New York
28. Travassos G, Shull F, Fredericks M, Basili VR (1999) Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proceedings of the 14th conference on object-oriented programming, systems, languages, and applications. ACM, pp 47–56
29. Ciupke O (1999) Automatic detection of design problems in object-oriented reengineering. In: Firesmith D (ed) Proceeding of 30th conference on technology of object-oriented languages and systems. IEEE Computer Society, Los Alamitos, CA, pp 18–32
30. Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th international conference on software maintenance. IEEE Computer Society, Los Alamitos, CA, pp 350–359
31. Munro MJ (2005) Product metrics for automatic identification of "bad smell" design problems in Java source-code. In: Lanubile F, Seaman C (eds) Proceedings of the 11th international software metrics symposium. IEEE Computer Society, Los Alamitos, CA
32. Rao AA, Reddy KN (2008) Detecting bad smells in object oriented design using design change propagation probability matrix. In: Proceedings of the international multiconference of engineers and computer scientists
33. Dhambri K, Sahraoui H, Poulin P (2008) Visual detection of design anomalies. Proceedings of the 12th European conference on software maintenance and reengineering, Tampere, Finland. IEEE Computer Society, Los Alamitos, CA, pp 279–283
34. Lanza M, Marinescu R (2006) Object-oriented metrics in practice. Springer, Berlin
35. Casais E (1998) Reengineering object oriented legacy systems. J Obj Orient Program 3(4):233–301
36. Coleman D, Ash D, Lowther B, Oman PW (1994) Using metrics to evaluate software system maintainability. IEEE Comput Pract 27(8):44–49
37. Li W, Henry S (1993) Object-oriented metrics that predict maintainability. J Syst Softw 23(2):111–122
38. Subramanyam R, Krishnan MS (2003) Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Trans Soft Eng 29(4):297–310
39. Bugzila: https://bugzilla.mozilla.org/
40. Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of OO metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910
41. Briand LC, Daly JW, Wust J (1998) A unified framework for cohesion measurement in object oriented systems. Empir Softw Eng 3(1):65–117
42. Rule package description for FrontEndART monitor (2009). http://www.frontendart.com/sites/default/files/BSM.pdf
43. Vidacs L, Ferenc R, Besezedes A (2004) Columbus schema for C/C+ preprocessing. In: 8th European conference on software maintenance and reengineering (CSMR 2004), Tampere, Finland, March 24–26. IEEE Computer Society, Los Alamitos, CA, pp 75–84
44. Ferenc R, Besezedes A (2002) Data exchange with the Columbus schema for C+. In: 6th European conference on software maintenance and reengineering (CSMR 2002), Budapest, Hungary, March 11–13. IEEE Computer Society, Los Alamitos, CA, pp 59–66
45. Fawcett T (2006) An introduction to ROC analysis. Pattern Recognit Lett 27:861–874

46. Hanley JA, McNeil BJ (1963) A method of comparing the areas under ROC curves derived from the same cases. Radiology 148(3):839–843
47. Green DM, Swets JM (1966) Signal detection theory and psychophysics. Wiley, New York
48. Spackman KA (1989) Signal detection theory: valuable tools for evaluating inductive learning. In: 6th international workshop on machine learning, Canada, pp 160–163
49. Moser R, Abrahamsson P, Pedrycz W, Sillitti A, Succi G (2008) A case study on the impact of refactoring on quality and productivity in an agile team. In: Balancing agility and formalism in software enineering. LNCS (5082). Springer, Heidelberg, pp 252–266
50. Singh S, Kahlon KS (2012) Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. ACM SIGSOFT SEN 37(2):1–11
51. Zimmermann T, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of ESEC/SIGSOFT FSE, New York, USA pp 91–100
52. Burak Turhan, Tim Menzies, Ayse Bener, Justin Distefano (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578
53. Turhan B, Tosun A, Bener A (2011) Empirical evaluation of mixed-project defect prediction models. In: 37th EUROMICRO conference on software engineering and advanced applications (SEAA), pp 396–403
54. Menzies T, Butcher A, Marcus A, Zimmermann T, Cok D (2011) Local vs. global models for effort estimation and defect prediction. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering (ASE'11). IEEE Computer Society, Los Alamitos, CA, pp 343–351