Effectiveness of Refactoring Metrics Model to Identify Smelly and Error Prone Classes in Open Source Software

Satwinder Singh
Asstt. Professor,
Dept. Computer Science Engineering & Info. Tech.,
B.B.S.B. Engg. College, Fatehgarh Sahib-140407
satwindercse@gmail.com

K.S. Kahlon,
Professor,
Dept. Computer Science Engineering,
Guru Nanak Dev University, AMRITSAR-143001
karanvkahlon@yahoo.com

ABSTRACT

In order to improve software maintainability, possible improvement efforts must be made measurable. One such effort is refactoring the code which makes the code easier to read, understand and maintain. It is done by identifying the bad smell area in the code. This paper presents the results of an empirical study to develop a metrics model to identify the smelly classes. In addition, this metrics model is validated by identifying the smelly and error prone classes. The role of two new metrics (encapsulation and information hiding) is also investigated for identifying smelly and faulty classes in software code. This paper first presents a binary statistical analysis of the relationship between metrics and bad smells, the results of which show a significant relationship. Then, the metrics model (with significant metrics shortlisted from the binary analysis) for bad smell categorization (divided into five categories) is developed. To develop the model, three releases of the open source Mozila Firefox system are examined and the model is validated on one version of Mozila Sea Monkey, which has a strong industrial usage. The results show that metrics can predict smelly and faulty classes with high accuracy, but in the case of the categorized model, not all categories of bad smells can adequately be identified. Further, few categorised models can predict the faulty classes. Based on these results, we recommend more training for our model.

Categories and Subject Descriptors

D.2.8. [Software Engineering]: Metrics- *Performance measure, Product metrics*

General Terms

Measurement, Design and Performance

Keywords:

Refactoring, Encapsulation, Information Hiding, Evolution, Bad smells, Empirical Analysis

1. INTRODUCTION

Evolutions of software over many years often lead to obscure system. It is hard to predict where changes have to be applied on the code and their affect on the system. This leads to maintenance and makes the evolution of system more and more complex and expensive. For efficient evolution and maintenance of code, the structure and design of a system has to be improved. To do this, first we have to identify the problematic areas and then improvement needs to be applied. This is one phase of the lifecycle of software reengineering [10]. Bad smell is one of the problems in the code which is addressed in this paper. Refactoring is one of the solutions proposed by agile community [15] to solve this problematic area. Refactoring changes the internal structure of the OO code without affecting the overall behavior of the system. Refactoring plays an important role in reengineering and reverse engineering which makes the code easier to understand for software developers. Further, we need a support of the automatic tools to identify the problematic area. In software industry, a tool uses the metrics approach to report the problematic and improvement area. Object Oriented metrics are used by software manger for different purposes to serve their needs. One of the purposes served by OO metrics is to identify the refactoring area in a code. Fowler[15] has defined the 22 bad smells that describe design problems. Fowler and Beck, however, do not suggest any criteria for making the decision to refactor. In this paper, we try to define criteria by predicting the association between internal and external metrics. Design metrics on the internal metrics side and code quality (bad smells) on the external metrics side are studied to determine the metrics model for refactoring or maintenance.

Earlier, software metrics were successfully used to check the maintainability of software [7, 26, 31]. The maintainability index is also helpful in software evaluation, where software is released in sequential order. Software that is released in sequential order must be easily understandable and readable so that the evolution team can easily work with it.

In this study, we attempt to find aspects of metrics that assist in finding the faulty classes and bad smells in code. First, we investigate whether software metrics can predict bad smells and fault in code. Secondly, we focus on whether software metrics can predict bad smell and faulty class probability under a 6-level categorisation defined by Mäntylä [27]. An empirical study has been carried out on open source Mozilla Firefox code. To generalise the metrics model, we validate it by applying metrics models of Mozila Firefox on Mozila Sea Monkey to identify the smelly and faulty classes in it. In parallel, we also analyse the usability and validity of encapsulation and information hiding metrics.

2. BAD SMELLS IN CODE

Identifying bad smells in code helps to refactor the code. This further helps in facilitating evolution of the projects. A number of studies (based on software metrics) have been carried out on decision making for refactoring [2, 14, 17, 28, 29]. According to [18] metrics for reviewing or refactoring code are not well defined. So, there is a need for certain external attributes to refactor the code for better review, understandability, maintainability, and evolution of the software. Metrics can help in guiding this discussion by providing solid information regarding certain aspects of object-oriented properties. Collection and analysis of metric data is time consuming. Furthermore, it is not foolproof and interpretation of the metric results is difficult [18]. So, using the source code metric together with a code review is the best practice. Code metrics provide an objective overview, while a code review (bad smell) provides qualitative information about the evolvability of the code.

Research results show that there is a relationship between structural attributes (design metrics) and external quality metrics (bad smell and class error tendencies) [1,2,4,11,12,19,30]. Studies have also been done to find a relation between bad smells or anti-patterns and class errors[9, 23, 25]. With the availability of bad smell information, the testing team

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

can focus on classes for consideration. In this paper we set out to find the association between bad smells and software metrics. Not many studies had focused on earlier research. Researchers [18, 27] were of the opinion that certain automated tools were required to find bad smells (which is used in this paper), because humans are liable to make errors during the analysis. Human errors in subjective evaluation have been shown in [9,14, 27], where different developers have different views on the same module and the same bad smells. Refactoring can be done by finding smelly classes. The parameter that is required to be adjusted or taken care of, can be determined through the code review and metric values. For better statistical results of our empirical analysis, bad smells are divided into the following categories [27].

Table 1. Bad Smell Categorisation

S. No.	Smell Category	Bad Smells	Definition
1.	Bloaters	1. Long Method 2. Large Class 3. Primitive Obsession 4. Long Parameter List 5. Data Clumps	Bloater smells represent something that has grown so large that it cannot be handled effectively.
2.	Object-Oriented (OO) Abusers	1. Switch Statements 2. Temporary Field 3. Refused Bequest 4. Alternative Classes with Different Interfaces 5. Parallel Inheritance Hierarchies	This represents cases where the solution does not fully exploit the possibilities of large object-oriented design.
3.	Change Preventer	1. Divergent Change 2. Shotgun Surgery	Preventers are smells that hinder changes for further development of software.
4.	Dispensable	1. Lazy class 2. Data class 3. Duplicate Code 4. Speculative Generality	Dispensable smells have one thing in common, that, they all represent something unnecessary that should be removed from source code.
5.	Couplers	1. Message Chains 2. Middle Man 3. Feature Envy 4. Inappropriate Intimacy	Couplers deals with data communication and encapsulation bad smells. It also represents high coupling, which is contrary to object-oriented design, which emphasize minimal coupling between objects.
6.	Others	1. Incomplete Library Class 2. Comments	This class contains the two remaining smells that do not fit into other categories.

Table 2. Categorised Bad Smell Distributions for Firefox Versions

Bad Smell Category	Bad Smell	Ver. 1.5	Ver. 2.0	Ver. 3.0
Bloater	Large Method Long Parameter List Data Clumps Large Class	626	827	878
Object-Oriented Abuser	Temporary Field	136	144	113
Change Preventer	Shotgun Survey	39	41	45
Dispensable	Lazy Class Data Class Speculative Generality	177	321	301
Coupler	Middle Man Feature Envy Inappropriate Intimacy	875	1203	1408

The interested reader can find detailed definitions of each bad smell in [16]. In the statistical analysis approach, we provide a metrics model for each bad smell category. So, at least one bad smell has been selected to analyse each category. In total, 12 bad smells have been picked from the first five categories. The sixth category was not considered because comments are considered to be significant for maintenance activity and furthermore, the Columbus tool [22] was also not able to measure the Incomplete Library Class smell. The distribution of the effective smelly classes in each category is shown in Table 2. This table also lists the bad smells considered in this study.

2. DATA COLLECTION

Three Mozilla Firefox and one Mozila Sea Monkey open source systems were used to validate our study. A metric and bad smell database was collected by making use of the Columbus Wrapper Framework tool (academic command prompt version on special request) [22]. The interested user can read about these metrics in detail from [5, 6, 20, 32]. Selection of the metric criteria depends on the factor that it should cover each object-oriented property and be able to measure using the Columbus tool.

Each class is smelly if there is at least one type of bad smell or null if no bad smell is identified. After identifying smelly classes, the bad smell classes were categorized according to the bad smell taxonomy in Table 1. If a class has more than one type of bad smell then the class is listed each time under the particular category group. Faulty class information or data for each version of Mozila Firefox and Sea Monkey has been collected from Bugzilla[1] database.

3. STATISTICAL METHOD

Regression analysis was used to analyse the results of the collected data. For selected independent variables (collected by Multicollinearity Analysis), logistic regression [8] was used to analyse the results of the association between the bad smell and the software metrics. Thereafter, the metrics model from one version was applied to Mozila Sea-monkey 1.0.1 to check the validity of the model using the area under the ROC curve. In parallel with this validity analysis, the area under the ROC curve was also analysed with and without the Encapsulation Factor (EncF) and Public Factor (PuF).

[1] www.bugzila.org

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

Independent variables (metrics) were chosen by passing them through a test for multicollinearity. Multicollinearity of the metrics was removed by Variance Inflation Factor (VIF) analysis. The limit for VIF and tolerance is VIF <10 and tolerance level >.10, respectively [24]. VIF and tolerance are calculated as follows:

Tolerance=1-R_j^2 VIF=1/(1-R_j^2) where R_j^2 is a regression coefficient

After selecting the independent metrics from the VIF analysis, the empirical model was built with the use of logistic regression, which was used because the dependent variable is dichotomous. For this type of dependent variables the scientific community prefers logistic regression. Also, the logistic function takes input from negative infinity to positive infinity, but gives the output in the range 0 to 1 (Figure 2). In logistic regression the input is *z* and the output function is *f(z)*. The variable *z*, known as the logit, is a measure of the total contribution of all the independent variables used in the model and is defined as follows [8]:

$$Z=\beta_0+\beta_1x_1+\beta_2x_2+\beta_3x_3+\ldots+\beta_nx_n \ ,$$

where β_0 is the intercept and $\beta_1, \beta_2, \beta_3,$ *and* β_n are regression coefficients of x_1, x_2, x_3 *and* x_n, respectively.

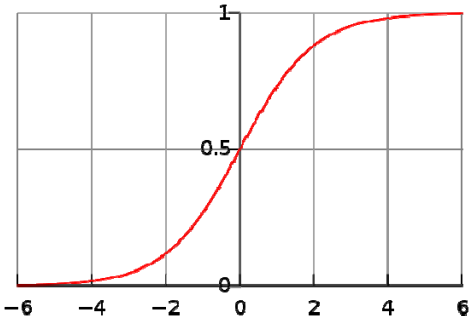

Fig. 2. Logistic Curve (x- and y-axes represent the input and output parameters, respectively)

Two types of dependent variables were considered: a binary variable and a categorical variable indicating the category of bad smell according to the taxonomy in Table 1. Multinomial Logistic Regression (MLR) was used to find the relation between smelly classes and the metrics, while the Multivariate Multinomial Regression (MMLR) model was used to find the relation between each type of metric and the different taxonomy levels of bad smells.

The significant independent metrics were considered for the multivariate prediction model. Significant metrics were selected from a Univariate Binary Regression (UBR) and Univariate Multinominal Regression (UMR), respectively, for MLR and MMLR analyses. UBR was used to examine the relation between the metric and bad smell, whereas UMR was used to examine the relation between the metric and each category of bad smell. MLR was used to predict bad smell probability within the class and MMLR to predict bad smell probability within each bad smell category. The dependent variable in the MLR model was whether the class was smelly. In the MMLR model the dependent variable was the categorical division of smelly classes into bad smells according to Table 1. We used '*None*' if the class was not smelly and as the reference base.

The general MLR model is as follows (UBR is a special case with n = 1):

$$\pi(Y = 1 \mid X_1, X_2, \ldots, X_n) = \frac{e^{f(x)}}{1+e^{f(x)}}$$

where $f(x) = \beta_0+\beta_1x_1+\beta_2x_2+\beta_3x_3+\ldots+\beta_nx_n$ is the logit function; π is the probability of a class being faulty; Y is the dependent variable (a binary variable); X_i ($1 <= i <= n$) are the independent variables, which are the OO metrics investigated in this study; and β_i($0 <= i <= n$) are the regression coefficients from maximizing the log-likelihood.

The general MMLR model is as follows (UMR is a special case with n=1):

$$\pi(Y = j \mid X_1, X_2, \ldots, X_n) = \frac{e^{f_j(x)}}{\sum_{k=0}^{m} e^{f_j(x)}}$$

where the vector $\beta_0 = 0$, $f_0(x) = 0$, $f(x) = \beta_{j0}+\beta_{j1}x_1+\beta_{j2}x_2+\beta_{j3}x_3+\ldots+\beta_{jn}x_n$ is the logit function for category j; m is the number of categories; π is the probability of error in category j; Y is the dependent variable, which is a categorical variable for bad smell; the X_i ($1 <= i <= n$) are the independent variables, which are the OO metrics investigated in this study; and the β_j ($0 <= j <= n$) are the regression coefficients from maximizing the log-likelihood.

The odds ratio (O.R.) associated with each regression coefficient value can also be obtained with logistic regression analysis. It is defined as the probability of the outcome of events occurring divided by the probability of the events not occurring. In general, the "odds ratio" is one set of odds divided by another. The odds ratio is defined as the relative amount by which the odds of the output increases (O.R. > 1.0) or decreases (O.R. < 1.0) when the value of the predictor variable (here the metric variable) is increased by 1.0 units.

The statistical analysis was started with the following set of metrics covering each property of object-oriented programming: coupling (CBO) [1], polymorphism (RFC) [1], cohesion (Lack of Cohesion of Method (LCOM) [1] and (LCOM4)[21]), information hiding (PuF) [32], encapsulation (EncF) [32], inheritance (NOC, DIT)[1], and abstraction (WMC) [1]. During the statistical analysis some of the metrics were dropped depending on the conditions of the statistical method.

4. EVALUATION OF METRICS

Descriptive statistics of the collected data i.e., Mozilla Firefox versions 3.0, 2.0, and 1.5 are shown in Tables 3 - 5.

Table 3. Descriptive Statistical Analysis of Firefox Version 3.0

		NOC	DIT	LCOM	LCOM 4	WMC	PuF	EncF	CBO	RFC	NOD
Mean		1.08	2.14	225.85	4.70	40.04	.86	.157	10.37	26.91	2.84
Std. Dev		16.40	2.02	1451.15	10.06	116.30	.18	.172	14.20	48.99	50.82
Min		0	0	0	0	0	.00	0	0	0	0
Max		1132	11	55612	104	3294	1.00	1.00	194	769	3484
Percentiles	25	.00	1.00	.00	1.00	1.00	.00	.29	3.00	4.00	.00
	50	.00	2.00	3.00	2.00	8.00	.11	.37	5.00	11.00	.00
	75	1.00	3.00	41.00	4.00	34.00	.26	.48	12.00	27.00	1.00
	90	2.00	5.00	282.80	8.00	94.00	.40	.57	25.00	66.00	2.00

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

Table 4. Descriptive Statistic Analysis of Firefox Version 2.0

		NOC	DIT	LCOM	LCOM 4	WMC	PuF	EncF	CBO	RFC	NOD
Mean		.97	1.97	254.69	4.31	36.88	.83	.18	9.26	25.20	2.46
Std. Dev		16.00	1.92	2264.38	9.240	125.52	.184	.183	13.97	50.05	46.36
Min		0	0	0	0	0	.00	0	0	0	0
Max		1198	12	115440	131	5301	1	1	225	855	3856
Percentiles	25	.00	1.00	.00	1.00	2.00	.75	.00	2.00	3.00	.00
	50	.00	2.00	3.00	2.00	7.00	.86	.13	5.00	10.00	.00
	75	1.00	3.00	35.00	4.00	28.00	1.00	.29	11.00	24.00	1.00
	90	1.00	4.00	289.00	8.00	85.00	1.00	.45	22.00	63.00	2.00

-

Table 5. Descriptive Statistic Analysis of Firefox Version 1.5

		NOC	DIT	LCOM	LCOM 4	WMC	PuF	EncF	CBO	RFC	NOD
Mean		1.07	1.99	202.16	4.30	40.30	.861	.150	10.55	27.0	2.51
Std. Dev		16.24	1.80	1298.3	8.333	112.74	.174	.1603	14.03	47.3	48.37
Min		0	0	0	0	0	.0	.00	0	0	0
Max		1074	10	54619	101	3233	1.0	.901	182	764	3204
Percentiles	25	.00	1.00	.00	1.00	1.00	.786	.000	3.00	4.00	.00
	50	.00	2.00	4.00	2.00	9.00	.919	.121	5.00	12.00	.00
	75	1.00	3.00	44.00	5.00	36.00	1.00	.250	13.00	29.00	1.00
	90	2.00	4.00	268.00	8.00	97.00	1.00	.369	24.00	64.0	3.00

Table 6. Univariate Binary Regression Analysis

Metrics	Firefox 1.5		Firefox 2.0		Firefox 3.0	
	B	p-value	B	p-value	B	p-value
NOC	-.233	.000	-.268	.000	-.166	.000
NOD	-.012	.018	-.020	.000	-.009	**.054**
DIT	.178	.000	.161	.000	.185	.000
CBO	.156	.000	.157	.000	.169	.000
RFC	.041	.000	.044	.000	.038	.000
WMC	.047	.000	.048	.000	.048	.000
LCOM	.001	.000	.002	.000	.001	.000
LCOM4	.060	.000	.087	.000	.051	.000
PuF	-5.503	.000	-3.486	.000	-3.121	.000
EncF	2.765	.000	2.405	.000	1.583	.000

4.1 Univariate Analysis of Metrics

In this section Binary (UBR) and Multinominal regression (UMR) analyses are carried out to determine the relationship between the set of metrics (NOC, NOD, DIT, LCOM, LCOM5, CBO, RFC, PuF, EncF) and bad smells. UBR and UMR analyses are used to shortlist the metrics on the basis of the significance level of their association.

Table 6 presents the UBR analysis showing the association between bad smells and metrics. The metrics are significantly associated with a smelly class if the *p-value* is less than 0.05. In Table 6 it can be seen that almost all the metrics are significantly associated with the bad smells of the class.

No trend was found, which shows that some of the metrics are not associated with bad smells, except for NOD in version 3.0. This shows that all have some role to play in the identification of a bad smell in the class. Next, multinomial regression analysis was done based on the categorisation of bad smells.

It can be seen from Table 7 that the NOD metric is not associated with most of the categorised bad smells. EncF and PuF metrics are found to be helpful in identifying almost all the bad smells. This shows the significant role of these metrics for bad smell identification in the classes. Next, independent metrics were picked by removing the collinearity from the metrics set. A collinearity test was done by calculating the virtual inflation factor (VIF) and tolerance level parameter. The acceptable range for VIF is <10 and for tolerance is 0.1. From Table 8 it can be seen that NOC, NOD, RFC, and WMC have VIF values greater than 10. So, we first omitted NOD (because of the high VIF value and also as it does not play a significant role in the identification of smelly classes as seen in Tables 6 and 7). This shows that NOC values for VIF and tolerance are in the acceptable range. After dropping NOD we omitted RFC. In Table 9 all the metrics now have VIF and tolerance values less than 10 and greater than 0.1, respectively. This leads to the required set of independent variables, that is, NOC, DIT, CBO, WMC, LCOM, LCOM4, PuF and EncF. This set of metrics was used to determine the MLR and MMLR models.

4.2 Bad Smell Multivariate Prediction Model

After obtaining the independent metric variables, we used MLR and MMLR analyses to propose the metrics model for bad smell identification. According to Hosmer-Lemshow [8] collinearity analysis alone cannot find the optimum set of independent variables. To obtain the optimum set, we have to apply a forward stepwise selection procedure for the MLR and MMLR models. Table 10 shows the MLR model of metrics with the forward stepwise procedure. From Table 10 it can be seen that the new metrics (PuF and EncF) have a significant association with bad smells in predicting smelly classes. Only the DIT metric for version 3.0 does not have a significant association with bad smells. Next, the model was tested with two fitness tests at a 95% confidence level. Table 11 gives the p-values for the goodness-of-fit test for the Likelihood Ratio and Hosmer-Lemshow and shows that the selected model is significant as a whole, rather than for individual metrics.

4.3 Multivariate Prediction Model Based on Bad Smell Categorisation

Using MMLR analysis we obtained a metrics model based on the categorization in Table 1. The model was proposed for the independent metrics set with the forward stepwise selection procedure. The proposed model for each version of Firefox is presented in Table 12. From this table it can be seen that for version 3. DIT is not a significant predictor for the Bloater and OO Abuser categories, which differs from the results in Table 10 for MLR analysis. The new metrics PuF and EncF are significant predictors of smelly classes when considering Bloater, OO Abuser, Dispenser, and Coupler smells. These categories include all bad smells related to object-oriented properties. Further it can be seen that EncF and PuF are not significant predictors of Change Preventer bad smells in versions 3.0 and 2.0. In other words, we can conclude that 12 out of the 20 smells were identified at a significant level using this proposed model. In all other cases we could not obtain a generalized view to ignore the metrics, because in most cases, the metrics significantly predicted the smelly classes. Model accuracy and significance was checked in Table 13 at the 95% level of confidence,

where it was found that the metrics MMLR models presented in Table 12 are statistically significant.

Table 7. Univariate Multinominal Regression Analysis

Metrics	Category	Firefox 3.0			Firefox 2.0			Firefox 1.5		
		B	p-value	O.R.	*B*	p-value	O.R.	*B*	p-value	O.R.
NOC	Bloater	-.157	.000	.855	-.250	.000	.779	-.176	.000	.839
	Object-Oriented Abuser	-.276	.017	.759	-.375	.002	.687	-.508	.000	.602
	Change Preventer	.001	**.839**	1.001	.000	**.947**	1.000	.001	**.909**	1.001
	Dispenser	-.085	.034	.919	-.488	.000	.614	-.630	.000	.533
	Coupler	-.095	.000	.909	-.185	.000	.831	-.173	.000	.841
NOD	Bloater	-.007	**.060**	.993	-.015	.017	.985	-.009	**.122**	.991
	Object-Oriented Abuser	-.018	**.272**	.982	-.049	**.140**	.952	-.158	.020	.854
	Change Preventer	.001	**.350**	1.001	.001	**.668**	1.001	.001	**.611**	1.001
	Dispenser	-.019	**.082**	.981	-.256	.000	.774	-.447	.000	.640
	Coupler	-.006	.035	.994	-.014	.007	.986	-.008	**.094**	.992
DIT	Bloater	.246	.000	1.279	.256	.020	1.292	.261	.000	1.298
	Object-Oriented Abuser	.261	.000	1.298	.216	.041	1.241	.181	.000	1.198
	Change Preventer	-.250	.031	.779	-.374	**.140**	.688	-.477	.002	.621
	Dispenser	-.045	**.219**	.956	-.049	.039	.952	-.051	**.295**	.950
	Coupler	.196	.000	1.217	.163	.019	1.177	.171	.000	1.187
LCOM	Bloater	.002	.000	1.002	.003	.000	1.003	.002	.000	1.002
	Object-Oriented Abuser	.002	.000	1.002	.003	.000	1.003	.002	.000	1.002
	Change Preventer	.002	.000	1.002	.003	.000	1.003	.002	.000	1.002
	Dispenser	.002	.000	1.002	.002	.000	1.002	.002	.000	1.002
	Coupler	.002	.000	1.002	.003	.000	1.003	.002	.000	1.002
LCOM 4	Bloater	.086	.000	1.089	.136	.008	1.146	.100	.000	1.105
	Object-Oriented Abuser	.089	.000	1.093	.130	.010	1.139	.086	.000	1.090
	Change Preventer	.092	.000	1.096	.146	.012	1.157	.108	.000	1.114
	Dispenser	-.007	**.681**	.993	.045	.015	1.046	-.040	**.135**	.960
	Coupler	.077	.000	1.080	.129	.008	1.138	.096	.000	1.101
WMC	Bloater	.053	.000	1.054	.053	.000	1.054	.049	.000	1.050
	Object-Oriented Abuser	.053	.000	1.054	.053	.000	1.054	.048	.000	1.049
	Change Preventer	.053	.000	1.054	.053	.000	1.054	.049	.000	1.050
	Dispenser	.042	.000	1.043	.039	.000	1.040	.039	.000	1.040
	Coupler	.051	.000	1.053	.052	.000	1.053	.048	.000	1.049
CBO	Bloater	.201	.000	1.223	.188	.000	1.207	.176	.000	1.193
	Object-Oriented Abuser	.201	.000	1.222	.185	.000	1.203	.164	.000	1.178
	Change Preventer	.190	.000	1.209	.181	.000	1.198	.165	.000	1.179
	Dispenser	.092	.000	1.096	.075	.000	1.078	.072	.000	1.074
	Coupler	.186	.000	1.204	.173	.000	1.189	.163	.000	1.177
RFC	Bloater	.049	.000	1.050	.054	.000	1.055	.048	.000	1.049
	Object-Oriented Abuser	.049	.000	1.050	.054	.000	1.055	.046	.000	1.047
	Change Preventer	.050	.000	1.051	.056	.000	1.057	.049	.000	1.050
	Dispenser	.018	.000	1.018	.021	.000	1.022	.020	.000	1.020
	Coupler	.045	.000	1.046	.050	.000	1.052	.045	.000	1.046
PuF	Bloater	-4.27	.000	.014	-4.68	.000	.009	-6.75	.000	.001
	Object-Oriented Abuser	-4.10	.000	.016	-4.17	.000	.015	-5.81	.000	.003
	Change Preventer	-3.31	.000	.036	-3.35	.000	.035	-5.04	.000	.006
	Dispenser	-2.16	.000	.114	-1.41	.000	.243	-3.61	.000	.027
	Coupler	-3.20	.000	.040	-3.71	.000	.024	-5.69	.000	.003
EncF	Bloater	2.363	.000	10.624	3.341	.000	28.258	3.566	.000	35.381
	Object-Oriented Abuser	1.253	.022	3.502	1.199	.036	3.317	.973	**.126**	2.646
	Change Preventer	2.843	.000	17.160	3.523	.000	33.874	4.27	.000	71.590
	Dispenser	-.561	**.176**	.571	-.950	.040	.387	-1.50	.025	.223
	Coupler	1.374	.000	3.950	2.422	.000	11.264	2.699	.000	14.862

Almost all the proposed models include the same set of metrics (DIT, CBO, LCOM, LCOM4, WMC, PuF, and EncF). All insignificant values in each category are demarcated in bold in Table 12. A selection of metrics was found to be significant in each case as shown in Table 13.

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

5. METRICS MODEL ACCURACY

The accuracy of the proposed models was evaluated with ROC. The ROC curves are two-dimensional graphs that visually depict the performance and performance trade-off of a classification model [15].

Table 8. Collinearity Statistics

Metric	Firefox 3.0		Firefox 2.0		Firefox 1.5	
	Tolerance	VIF	Tolerance	VIF	Tolerance	VIF
NOC	.047	21.28	.029	34.85	.034	29.02
NOD	.047	21.28	.029	34.85	.034	29.02
DIT	.666	1.50	.723	1.38	.737	1.36
CBO	.174	5.76	.170	5.88	.168	5.95
RFC	.084	11.86	.084	11.93	.081	12.41
WMC	.162	6.17	.129	7.76	.120	8.32
LCOM	.340	2.94	.297	3.37	.271	3.69
LCOM4	.398	2.51	.467	2.14	.439	2.28
PuF	.362	2.76	.387	2.58	.373	2.68
EncF	.378	2.64	.417	2.39	.409	2.45

Table 10. Multilogistic Regression Model

	Firefox 3.0			Firefox 2.0			Firefox 1.5		
Metrics	B	Sig.	Exp(B)	B	Sig.	Exp(B)	B	Sig.	Exp(B)
DIT	N/A	N/A	N/A	-.147	.000	.864	-.128	.000	.880
CBO	.093	.000	1.097	.065	.000	1.068	.071	.000	1.074
LCOM	.000	.013	.999	-.002	.000	.998	-.002	.000	.998
LCOM4	-.050	.000	.951	.025	.000	1.025	.017	.020	1.017
WMC	.044	.000	1.045	.049	.000	1.050	.046	.000	1.047
PuF	-2.362	.000	.094	-2.087	.000	.124	-4.456	.000	.012
EncF	-1.867	.000	.155	-1.657	.000	.191	-2.676	.000	.069
Constant	.304	.393	1.355	.148	.690	1.160	2.212	.000	9.132

Table 9. Collinearity Statistics (dropping NOD & RFC)

Metric	Firefox 3.0		Firefox 2.0		Firefox 1.5	
	Tolerance	VIF	Tolerance	VIF	Tolerance	VIF
NOC	.998	1.00	.998	1.00	.998	1.00
NOD	N/A	N/A	N/A	N/A	N/A	N/A
DIT	.668	1.49	.726	1.38	.743	1.34
CBO	.297	3.36	.281	3.55	.284	3.51
RFC	N/A	N/A	N/A	N/A	N/A	N/A
WMC	.213	4.69	.173	5.77	.165	6.07
LCOM	.357	2.81	.303	3.30	.273	3.66
LCOM4	.551	1.81	.631	1.58	.633	1.58
PuF	.362	2.76	.389	2.57	.375	2.45
EncF	.378	2.64	.417	2.39	.409	2.66

They were originally designed as tools in communication theory to visually determine optimal operating points for signal discriminators [15]. The ROC curve is a plot of TPR (True Positive Rate) against FPR (False Positive Rate), the definitions of which according to the confusion matrix (Figure 3) are as follows:

$$TPR=TP/(TP+FN);\ FPR=FP/(TN+FP)$$

ROC identifies the number of regions of interest. The classification model is mapped on a diagonal line, which produces as many false positive responses as it produces true positive responses. The diagonal line represents a random performance classifier model, while the bottom left represents a few false positive errors. The top of the graph represents a good true positive rate. Classifiers that fall to the right of the random performance line have worse performance than the random classifier and produce more false positives than the true positive rate. Situated above the random performance line is a well performing classifier, with the top left corner denoting perfect classification, i.e., a 100% true positive and 0% false positive rate.

Figure 4 and Table 14 show that the Area under Curve (AUC) can be interpreted as a measure of the performance of the discrimination model. Separate MLR models with (Table 10) and without (see Appendix) PuF & EncF metrics were developed to draw separate ROC curves. It was observed that with the inclusion of the new metrics in the proposed model of MLR, the AUC increased. From Table 14 it can be seen that on applying higher versions, the AUC decreased in case of identification of smelly classes. This was not expected, because as a system evolves it should become more stable by removing the previous discrepancies. This concludes that previous smelly classes were not taken care of while evolving the software, or alternatively, the new classes have a greater bad smell (i.e., the percentage increase in the bad smell count is more than the percentage increase in the class count in the new version (Table 2). New metrics also shows significant discrimination for identification of faulty classes with AUC > .80.

Table 11. Model Fitness Test

	Firefox 3.0	Firefox 2.0	Firefox 1.5
Hosmer-Lemshow	.000	.000	.000
Likelihood Ratio	.000	.000	.000

In the case of the MMLR metrics model for bad smell identification, the ROC AUC predicted a well performing discrimination model for the Bloater and Change Preventer categories when the proposed models were applied to the Mozila Sea Monkey 1.0.1 including all metrics. In almost all cases of the MMLR model, inclusion of the new metrics (PuF and EncF) in the proposed model yielded a greater area under the curve, than without including these metrics. The area under the curve with the inclusion of the new metrics is always greater for each case in contrary to our previous study [32] where the 2nd, 4th and 5th categories had shown the higher area under the curve. The area under the curve increases between 5% to 55% with the inclusion of new metrics as per Table 15. From Table 15 and Fig. 5, it can be seen that when a higher version model is applied over Mozila Sea-Monkey, the area under the curve increases in contrary to MLR models. This shows that the MMLR model is more acceptable than the MLR model. The AUC in case of faulty class detection show improvements in Bloater, OOA and Coupler

categories. Where as in our previous study, [32] only Bloater and OOA show improvement with inclusion of new metrics.

Table 12. MMLR Analysis Results

Bad Smell Classification		Firefox Version 3.0			Firefox Version 2.0			Firefox Version 1.5		
		B	Sig.	O.R.	B	Sig.	O.R.	B	Sig.	O.R.
Bloater	Intercept	-.860	.055		-1.316	.005		1.189	.078	
	DIT	-.034	**.235**	.967	-.113	.000	.893	-.098	.005	.907
	CBO	.126	.000	1.134	.090	.000	1.095	.092	.000	1.097
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.010	.998
	LCOM4	-.042	.000	.959	.035	.000	1.036	.021	.000	1.021
	WMC	.047	.000	1.049	.055	.000	1.056	.051	.000	1.053
	PuF	-3.007	.000	.049	-2.549	.000	.078	-5.155	.005	.006
	EncF	-1.041	.040	.353	-.533	**.326**	.587	-1.883	.000	.152
Object-Oriented Abuser	Intercept	-.249	.763		1.419	.065		5.868	.003	
	DIT	-.037	**.467**	.964	-.152	.003	.859	-.153	.000	.858
	CBO	.113	.000	1.119	.085	.000	1.089	.084	.000	1.088
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	**.231**	.998
	LCOM4	-.042	.000	.959	.007	**.572**	1.007	-.020	.000	.980
	WMC	.048	.000	1.049	.054	.000	1.055	.049	.000	1.050
	PuF	-4.940	.000	.007	-5.765	.000	.003	-9.851	.000	5.3E-
	EncF	-5.612	.000	.004	-9.012	.000	.000	-11.99	.000	6.2E-
Change Preventer	Intercept	-5.854	.000		-7.550	.000		-7.208	.000	
	DIT	-.788	.000	.455	-1.122	.000	.326	-1.234	.000	.291
	CBO	.085	.000	1.089	.070	.000	1.073	.072	.000	1.075
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.000	.998
	LCOM4	.029	**.087**	1.029	.122	.000	1.130	.114	.000	1.121
	WMC	.049	.000	1.050	.057	.000	1.059	.054	**.310**	1.056
	PuF	.458	**.732**	1.581	1.880	**.198**	6.551	1.759	.045	5.809
	EncF	1.766	**.241**	5.847	3.020	**.069**	20.50	3.409	.000	30.23
Dispenser	Intercept	2.377	.000		1.606	.003		7.430	.000	
	DIT	-.143	.001	.867	-.193	.000	.824	-.209	.028	.811
	CBO	.034	.011	1.034	.010	**.455**	1.010	.036	.000	1.037
	LCOM	.000	.015	1.000	-.002	.000	.998	-.001	.001	.999
	LCOM4	-.101	.000	.904	-.015	**.318**	.985	-.108	.000	.897
	WMC	.044	.000	1.045	.046	.000	1.047	.039	.000	1.040
	PuF	-4.737	.000	.009	-3.752	.000	.023	-9.894	.000	5.0E-
	EncF	-6.523	.000	.001	-6.041	.000	.002	-	.000	9.5E-
Coupler	Intercept	.493	.207		-.019	.962		2.423	.000	
	DIT	-.055	.027	.947	-.186	.000	.830	-.177	.000	.838
	CBO	.108	.000	1.114	.074	.000	1.077	.076	.000	1.079
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.007	.998
	LCOM4	-.053	.000	.949	.029	.000	1.030	.020	.000	1.020
	WMC	.047	.000	1.048	.054	.000	1.056	.051	.000	1.052
	PuF	-2.873	.000	.057	-2.446	.000	.087	-5.063	.000	.006
	EncF	-3.144	.000	.043	-2.483	.000	.084	-3.839	.000	.022

Table 13. Goodness of Fit Test for MMLR models

Likelihood Ratio Test	Firefox 3.0	Firefox 2.0	Firefox 1.5
p-value	.000	.000	.000

Table 14. Accuracy of MLR Models

MLR Models	Area under curve according to model in Table 10		Area under curve without PuF & Encf metrics	
	For Smelly Classes	For Faulty Classes	For Smelly Classes	For Faulty Classes
Applying Firefox 1.5 to Mozila 1.0.1	.887	.865	.873	.848
Applying Firefox 2.0 to Mozila 1.0.1	.886	.844	.872	.842
Applying Firefox 3.0 to Mozila 1.0.1	.874	.884	.862	.875

Table 15. Accuracy of MMLR Models

MMLR Models		Area under curve according to model in Table 12		Area under curve without PuF and Encf Metrics	
		For Smelly Classes	For Faulty Classes	For Smelly Classes	For Faulty Classes
Applying Firefox 3.0 to Mozila 1.0.1	Bloater	.825	.767	.685	.641
	OO Abuser	.698	.400	.543	.637
	Change Preventer	.875	.493	.703	.505
	Dispensable	.691	.139	.592	.162
	Coupler	.586	.293	.533	.190
Applying Firefox 2.0 to Mozila 1.0.1	Bloater	.810	.809	.834	.865
	OO Abuser	.586	.412	.371	.146
	Change Preventer	.870	.494	.570	.118
	Dispensable	.654	.115	.620	.122
	Coupler	.626	.294	.318	.149
Applying Firefox 1.5 to Mozila 1.0.1	Bloater	.789	.877	.685	.641
	OO Abuser	.573	.894	.543	.637
	Change Preventer	.874	.200	.870	.505
	Dispensable	.695	.115	.592	.162
	Coupler	.627	.293	.533	.190

Predicted	Observed	
	True	False
Positive	True Positive (TP)	False Positive (FP)
Negative	True Negative (TN)	False Negative (FN)

Fig. 3. Format of Confusion Matrix

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

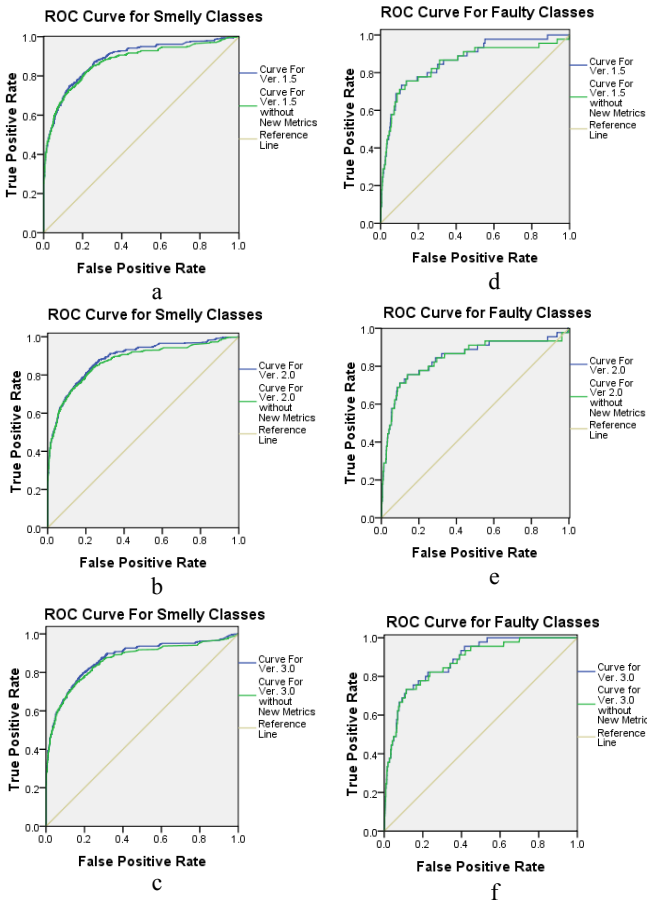

Fig. 4. ROC curves for the MLR model after applying Firefox model over Sea-Monkey. a-c) ROC curves for identifying smelly classes d-f) ROC curves for identifying faulty classes

6. DISCUSSION OF RESULTS

Bad smell identification will help in refactoring the code. It is refactoring which helps in better understandability and readability of code. The metrics model proposed over here will provide guidance to developers and managers about refactoring software. These models can also be used by the testing team to analyse the code before release. However, previous studies [9, 14, 27] tried to correlate metrics and bad smells using subjective manual evaluation. We used an automatic tool for bad smell evaluation, which produced a more encouraging and consistence result for bad smell identification with the metrics model in comparison to manual evaluations. The metrics model proposed in this paper also facilitates to control the error rate during development, where previously we identified the relationship between bad smells and error proneness [32].

Empirical results obtained in this work indicate that there is a significant relationship between the metrics and bad smells. Association of metrics was observed across (using MLR) and within almost all the bad smell categories in MMLR. An empirical study of the new metrics (PuF and EncF) shows encouraging results in the MMLR than the MLR analysis. On applying each of the three versions of the Mozila Firefox over the Mozila Sea Monkey, Mozila Firefox Version 3.0 metrics model seemed to be more favourable.

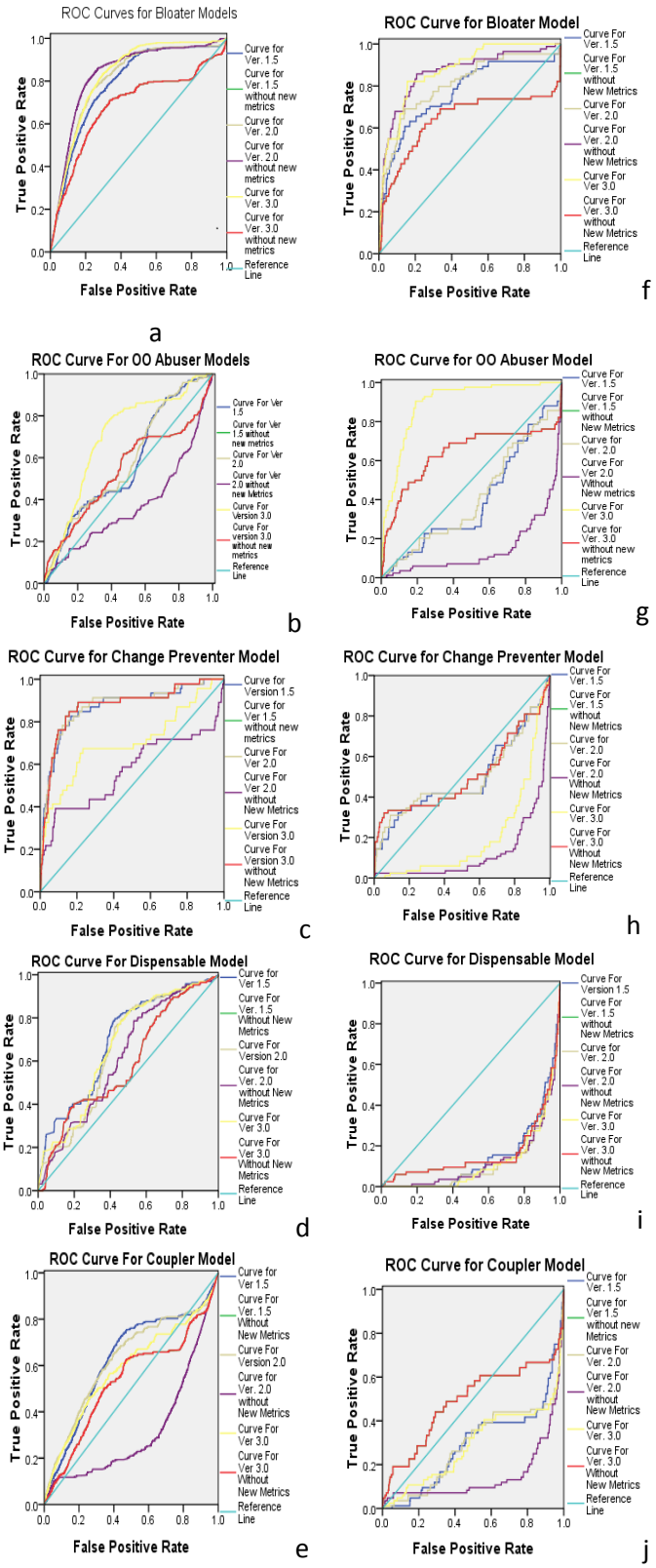

Fig. 5. ROC curves for the MMLR model after applying Firefox model over Sea- Monkey. a-e) ROC curves for identifying smelly classes f-j) ROC curves for identifying faulty classes.

MMLR analysis determines refactoring at the category level, which is more useful than the ability to predict whether refactoring is required or not (as in MLR). From the empirical analysis, it is clear that the new metrics (PuF and EncF) play a vital role by pushing the curve to an acceptable discrimination area (towards the top left corner).

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157

In each bad smell category of MMLR model for version 3.0, the area under the curve for ROC is spanning more area with inclusion of new metrics (PuF and EncF), which is contrary to our previous study [32]. In our previous study [32], model of version 2.0 was found to be more favourable for predicting the smelly and faulty classes in Firefox system.

Alike to our previous study [32], the categorisation metrics models of Bloater and Change Preventer categories can predict smelly classes more accurately (AUC > 0.70). Also, in these two categories the new metrics (PuF and EncF) play an important role for bad smell identification which is unlike to our previous work. In case of other three categories (OO Abuser, Dispensable, and Coupler) as per our previous findings in [32], new metrics play important role for identification of these three categories consistently. Unlike to our previous results [32] for identification of faulty classes, only one category model identified the faulty classes successfully in Mozila Sea Monkey i.e. Bloater category.
As per our previous recommendations in [32] (i.e. refactoring model to be applied on different Mozila products) and on applications of these, we conclude with favourable results. For more generalized results, the refactoring model needs to be applied on the software product of different business houses and on different programming languages.

7. CONCLUSION

In this research study, we designed and evaluated the refactoring metrics model for identification of smelly and faulty classes. Role of two new metrics (PuF and EncF) for identification of smelly and faulty classes with refactoring metrics models was studied. First, we designed a binary metrics model and then a multinominal categorisation model. Three Metrics model for refactoring was designed with three versions of Mozila Firefox system and evaluation of these three models was done on one version of Mozila Sea-Monkey. The results show that the binary regression model predicts smelly and faulty classes very well, whereas three categories of smells were correctly identified with the multinominal model. This is antithetical to our previous study [32] where two categories were identified correctly in case of smelly class detection. With the inclusion of the PuF and EncF metrics, the area under the curve for the categorisation model increased from 5% to 49%. This shows the significant role of the new metrics. The metrics models with new metrics were also found to be critical in identification of faulty classes, not for the system for which it was designed, but also for the third system (Mozila Sea Monkey). This is an enhancement to various previous studies [18, 19, 23, 25, 30 and 32] done on the same issue. These results can be improved by increasing the training and testing of the model. From the above discussion, we can conclude that some of the categorised smells were identified well with the metrics model, but not all. Further, two of the categorised metrics model can pinpoint the faulty classes. To validate this research, more experiments need to be done with different object-oriented programming languages. Moreover, this study was carried out on an open source system and needs to be done on professionally built applications. A limitation of this study is that it selected only 12 out of 22 bad smells which are identified by the Columbus Framework. The results may differ if other set of smells are considered. Using another form of categorisation than [27] may also cause different results. We do not recommend these results for a system that has already been developed. However, these results will ease the work of project teams during development and maintenance for better understanding and readability. We are in the process of extending our work to identify the association between different types of projects depending on the language and categories.

[13]Etzkorn L. H. et al., A comparison of cohesion metrics for object-oriented systems. Information and Software Technology., 2004,46(10), 677-687.

ACKNOWLEDGEMENT

The authors wish to thank the Frontend Arts researchers for providing the Columbus framework tool for the analysis, and especially Mr. Peter Siket for his assistance. We would also like to thank Prof. Mohamad Javed from the Statistics Department at Punjab Agricultural University for assisting us with statistical problems. Lots of thank to Prof. Gurleen Sidhu from CSE Department at BBSBEC, Fatehgarh Sahib for editing our English mistake. Thanks to the anonymous reviewers for their constructive review and comments.

8. REFERENCES:

[1]Bansiya J, David CG, A hierarchical model for object-oriented design quality. IEEE Transactions on software engineering, 2002, 28, pp. 4–17

[2]Briand, L., Arisholm, E., Counsell S., Houdek, F. and Thevenod-Fosse, P., Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Direction, Empirical Software Engineering, 1999, 4(4), 387-404.

[3]Briand, L.C., Wuest, J., Daly, J.W., Porter, D.V., Exploring the relationship between design measures and software quality in object oriented systems. Journal of Systems and Software 2000, 51(3), 245–273.

[4]Cartwright, M., Shepperd, M., An empirical investigation of an object-oriented software system. IEEE Transactions on Software Engineering, 2000, 26(7), 786–796.

[5]Chidamber S.R., Kemerer C.F., Towards a metrics suite for object oriented design, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA ’91), 1991, 197–21

[6]Chidamber, S.R., Kemerer, C.F., A Metric Suite for Object-Oriented design, IEEE Transactions on Software Engineering, June 1994, 20(6), 476-493.

[7]Coleman D, Ash D, Lowther B, Oman PW, Using metrics to evaluate software system maintainability. IEEE Computing Practices, 1994, 27(8), 44–49.

[8] D. Hosmer and S. Lemeshow, Applied Logistic Regression, second ed. John Wiley and Sons, 2000.

[9]Dhambri, K., Sahraoui, H., Poulin. P., Visual detection of design anomalies. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, IEEE CS, Tampere, Finland, April 2008, 279–283.

[10] Eduardo Casais, Reengineering object Oriented legacy systems Journal of Object Oriented Programming, pages 45-52, January 1998.

[11]Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N., The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering, 2001, 27(7), 630–648.

[12]Emam, K.E., Melo, Walcelio, Machado, Javam, The prediction of faulty classes using object-oriented design metrics. The Journal of Systems and Software, 2001, 56, 63–75.

[14]F. Simon, F, Steinbruckner, F., Lewerentz. C., Metrics based refactoring. In Proceedings of the Fifth European Conference on

Software Maintenance and Reengineering (CSMR'01) IEEE CS Press, 2001, pp 30.

[15]Fawcett, T., ROC graphs: Notes and practical considerations for researchers. Machine Learning, 2004, pp. 31

[16]Fowler, Martin, Refactoring: Improving the Design of Existing Code. Addison-Wisely, 2000.

[17]Francisca Munoz Bravo, A Logic Meta-Programming Framework for Supporting the Refactoring Process. PhD thesis, Vrije Universiteit Brussel, Belgium, 2003.

[18]Gronback Richard C., Software Remodeling : Improving Design and Implementation Quality Using audits , metrics and refactoring in Borland Together Control Centre, A Borland White Paper, January, 2003.

[19]Gyimothy, T., Ferenc, R., Siket, I., Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering, 2005, 31(10), 897–910.

[20]Henderson-Sellers, B., Object-Oriented Metrics: Measures of complexity, Prentice Hall Upper Saddle River, New Jersey, 1996.

[21]Hitz, M., Montazeri, B., Chidamber and Kemerer's metrics suite: A measurement perspective, IEEE Transactions on Software Engineering, 1996, 22(4), 267-271.

[22]http:// www.frontendart.com

[23]Khomh F, Penta MD. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness. Available at: www.ptidej.net/downloads/experiments/emse10/TR.pdf. Accessed 23 December 2010

[24]Kutner, Nachtsheim, Neter, Applied Linear Regression Models, 4th edition, McGraw-Hill Irwin, 2004.

[25]Li W, Shatnawi R., An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software. 2007, 80(7), 1120-1128. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0164121206002780

[26]Li, W. and Henry, S., Object-Oriented Metrics that Predict Maintainability, Journal of Systems and Software, 1993,23(2), 111-122.

[27]Mäntylä MV, Lassenius C. Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering., 2006, 11(3), 395-431.

[28]Marinescu, R., Detecting design flaws via metrics in object-oriented systems. In Proceedings of the TOOLS, USA 39, Santa Barbara, USA, 2001.

[29]Marticorena, R., Lopez C., Crespo Y., Extending a Taxonomy of Bad Code Smells with Metrics, WOOR'06, Nantes, 2006.

[30]Shatnawi R, Li W., The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of Systems and Software., 2008,81(11),1868-1882.

[31]Subramanyam R, Krishnan MS, Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Transactions on Software Engineering, 2003, 29(4), 297–310

[32] Singh Satwinder, Kahlon K.S., Effectiveness Of Encapsulation And Object-Oriented Metrics To Refactor Code And Identify Error Prone Classes Using Bad Smells. ACM SIGSOFT SEN, Sept 2011, Vol 36 (5)

APPENDIX: MLR and MMLR analysis results without new metrics (PuF and EncF)

MULTIVARIATE LOGISTIC REGRESSION Analysis Results (without PuF & EncF Metrics)

Metrics	Firefox 3.0			Firefox 2.0			Firefox 1.5		
	B	Sig.	O.R.	B	Sig.	O.R.	B	Sig.	O.R.
DIT	N/A	N/A	N/A	-.118	.000	.889	-.092	.002	.912
CBO	.092	.000	1.09	.062	.000	1.064	.067	.000	1.070
LCOM	.000	.018	.999	-.002	.000	.998	-.002	.000	.998
LCOM4	-.047	.000	.954	.024	.000	1.024	.016	.020	1.016
WMC	.045	.000	1.04	.050	.000	1.051	.049	.000	1.051
Const	-2.056	.000	.128	-1.963	.000	.140	-2.19	.000	.111

MULTIVARIATE MULTINOMINAL REGRESSION Analysis Results (without PuF & EncF Metrics)

Bad Smell Classification		Firefox Version 3.0			Firefox Version 2.0			Firefox Version 1.5		
		B	Sig.	O.R.	B	Sig.	O.R.	B	Sig.	O.R.
Bloater	Intercept	-3.514	.000		-3.472	.000		-3.544	.000	
	DIT	-.011	.678	.989	-.086	.006	.918	-.056	.098	.946
	CBO	.122	.000	1.12	.088	.000	1.092	.089	.000	1.093
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.000	.998
	LCOM4	-.042	.000	.959	.029	.000	1.029	.016	.051	1.016
	WMC	.046	.000	1.04	.053	.000	1.054	.050	.000	1.052
Object-Oriented Abuser	Intercept	-5.395	.000		-4.910	.000		-4.470	.000	
	DIT	.006	.905	1.00	-.104	.052	.902	-.077	.147	.926
	CBO	.110	.000	1.11	.082	.000	1.086	.080	.000	1.083
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.000	.998
	LCOM4	-.037	.000	.963	.014	.281	1.014	-.012	.457	.988
	WMC	.047	.000	1.04	.052	.000	1.053	.048	.000	1.049
Change Preventer	Intercept	-5.032	.000		-5.219	.000		-4.917	.000	
	DIT	-.764	.000	.466	-1.055	.000	.348	-1.131	.000	.323
	CBO	.079	.000	1.08	.063	.000	1.065	.062	.000	1.064
	LCOM	.000	.000	1.00	-.002	.000	.998	-.002	.000	.998
	LCOM4	.026	.104	1.02	.110	.000	1.116	.100	.000	1.105
	WMC	.046	.000	1.04	.053	.000	1.055	.051	.000	1.052
Dispenser	Intercept	-2.717	.000		-2.489	.000		-2.748	.000	
	DIT	-.052	.203	.949	-.107	.012	.898	-.074	.166	.929
	CBO	.023	.087	1.02	.002	.875	1.002	.025	.118	1.025
	LCOM	.000	.107	1.00	-.001	.000	.999	.000	.000	.999
	LCOM4	-.095	.000	.909	-.012	.410	.988	-.105	.002	.901
	WMC	.042	.000	1.04	.039	.000	1.040	.033	.000	1.034
Coupler	Intercept	-2.548	.000		-2.556	.000		-2.680	.000	
	DIT	-.010	.674	.990	-.141	.000	.868	-.114	.000	.892
	CBO	.105	.000	1.11	.072	.000	1.075	.074	.000	1.077
	LCOM	.000	.000	.999	-.002	.000	.998	-.002	.000	.998
	LCOM4	-.049	.000	.952	.029	.000	1.029	.020	.007	1.020
	WMC	.045	.000	1.04	.052	.000	1.053	.050	.000	1.051

DOI: 10.1145/2108144.2108157 http://doi.acm.org/10.1145/2108144.2108157