Effectiveness of Encapsulation and Object-oriented Metrics to Refactor Code and Identify Error Prone Classes using Bad Smells

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

To assist maintenance and evolution teams, work needs to be done at the onset of software development. One such facilitation is refactoring the code, making it easier to read, understand and maintain. Refactoring is done by identifying bad smell areas in the code. In this paper, based on empirical analysis, we develop a metrics model to identify smelly 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 verify our model, we examine the open source Firefox system, which has a strong industrial usage. The results show that proposed metrics model for bad smell can predict faulty classes with high accuracy, but in the case of the categorized model not all categories of bad smells can adequately identified the faulty and smelly classes. Due to certain limitations of our study more experiments are required to generalize the results of bad smell and faulty class identification in software code.

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

In the world of imperative and object-oriented languages, software measurement, also known as software metrics has been used for many years to provide developers with additional information about their programs. Such information can give programmers important indications about where bad code may affect the system. Software metrics are invaluable in simplifying the testing process by focusing the programmer's attention on utilitarian parts of the program. Information on these parts certainly helps reduce the programmer's effort, and may provide great overall benefit and help in easing the process of validating the software. Software metrics have been the subject of research over the last three decades, as they play a crucial role in making managerial decisions during the software lifecycle. Software metrics come in many guises, as described by Fenton [5], but metrics design is essentially an attempt to measure or predict an attribute (internal or external) of some product, process, or resource.

Earlier software metrics were successfully used to check the maintainability of software [14] [37] [44]. 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.

Writing clean and understandable code is a very difficult task due to changes in rhythm (written by one person and read by some other person). To make it more understandable, it is necessary to refactor, which makes the code more readable and cleaner. Programming in this way is all about saying what you want to say. The agile community has used this principal of refactoring as a solution to remove bad code smells [22]. Fowler and Beck, however, do not suggest any criteria for making the decision to refactor. In contrast, Graddy [24] provided threshold values for some program metrics. 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. Further, metrics model for bad smells is validated by identifying the faulty classes.

Programming language researchers and academics generally use encapsulation to restrict access to some of the object's components individually (attributes and methods). Although the Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF) can to a certain extent reflect the degree of information hiding, these metrics are not sufficient. They focus on the method and attribute levels and are coarsely granular and incomplete. A finely granular information hiding metric at the class level is basically required. Further design of encapsulation metrics at the class level is one of the areas on which we focus. We attempt to formulate the measurement thereof as per the definition and validate it empirically by regression analysis. A discussion on this is presented in Section 2.

In this study we attempt to find aspects of metrics that assist in finding the bad smells in code. First, we investigate whether software metrics can predict bad smells in code. Secondly, we focus on whether software metrics can predict bad smell probability under a 6-level categorisation defined by Mäntylä [38]. Then, to validate the metrics models for bad smells, one model is applied to same and subsequent version to ascertain the smelly and faulty classes of that particular versions. An empirical study has been carried out on open source Mozilla Firefox code. In parallel, we also analysed the usability and validity of encapsulation and information hiding metrics to judge the smelly and faulty classes.

2. DIFFERENCE BETWEEN INFORMATION HIDING, COHESION, AND ENCAPSULATION

The MHF and AHF metrics are considered by the MOOD team as measures of encapsulation [1, 2]. This, however, does not explain the concept of encapsulation appropriately [10, 27, 42]. Information hiding and encapsulation are not synonymous; information hiding is only one part of the encapsulation concept and is defined as "*Every module is characterized by its knowledge of a design decision which it hides from all others*" or simply that it defines the visibility of methods and

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

attributes of a class to other code [10]. Encapsulation can be thought of as the aggregate of two different but related attributes, namely *privacy* and *unity*. The Chambers 20th Century Dictionary definition reflects this view:

encapsulate *enclose in a capsule: capture the essence of, describe succinctly but sufficiently.*

The *privacy* aspect satisfies only the first part of this definition. Privacy is concerned with the visibility of the attributes and methods of a class to other classes in the system [42].
unity *oneness (...) a single whole: the arrangement of all the parts to one purpose or effect.*

This *unity* definition represents the degree of oneness that a single class represents and satisfies the second part of the definition of 'encapsulate'. Encapsulation is measured by unity and privacy (i.e., data privacy) of a class.

A class that has only private data members may not necessarily be unified. On the other hand, a fully unified class may contain only visible (public) data. MHF and AHF are measures of the visibility of the properties of a class. They are not measures of encapsulation. Although the AHF metric indirectly contributes to such a measure, it is doubtful if the MHF measure serves an equivalent purpose [42]. According to Khan *et al.* [33], encapsulation has a separate layer between the external interface (representing privacy) and the internal implementation (representing the unity of data members).

Since information hiding and encapsulation are two different concepts [10, 42], separate metrics are required to measure them both. Although MHF and AHF are considered to be measures of encapsulation [1], they only measure the visibility of class member functions and data members separately at the system level.

According to the C&K-metric suite, encapsulation is measured by Lack of Cohesion of Metrics (LCOM) [12]. However, this only partially measures the encapsulation [41], since it only measures the unity of the class members. Further, Hitz and Montazeri [29] [30] having comprehensively investigated the shortcomings of the LCOM metrics, proposed a new framework for cohesion metrics. According to the above definition this covers only the *unity* part of the encapsulation. Many improvements have been proposed for these cohesion metrics. In Section 4 we discuss in detail the improvements to the cohesion metrics and select the most suitable ones to measure the unity or cohesion part of the encapsulation.

From the above discussion we can conclude that encapsulation is the measure of unity and data visibility of attributes and methods. In this paper we measure and validate encapsulation by including both of these parameters (unity and visibility) to complete the definition of encapsulation.

3. INFORMATION HIDING

Information hiding is defined by Harrison *et al*. [27] in terms of the visibility of methods and/or attributes to other code. The MOOD metrics [1, 2] for encapsulation (or information hiding) i.e., AHF & MHF, were designed to function at the system level. Thus in this respect, fine granular metrics are required, and hence the design of information hiding metrics are proposed at the class level (to measure the information hiding at class level), as this is a property of an object. The proposed metrics for information hiding also consider the inherited members of a class.

3.1 Public Factor (PuF): This proposed metric calculates the composite visibility scope of methods and attributes at the class level. The range of the public factor is between 0 and 1, where PuF=0 denotes that there are no methods and attributes with public scope, and PuF= 1 denotes that there are no methods and attributes defined with private scope. The visibility of any member with protected scope is counted as:

$$V_i = DC(C_i)/(TC-1)$$

where $DC(C_i)$ denotes the descendants of the *current class* C_i, *TC* denotes the total number of classes, and V_i=1 for public members and V_i=0 for private members.

$$PuF = \frac{\sum_{i=1}^{TAC}(Pu(A_i)*V_i) + \sum_{i=1}^{TMC}(Pu(M_i)*V_i)}{\sum_{i=1}^{TAC}A_{ti} + \sum_{i=1}^{TMC}M_{ti}}$$

$Pu(A_i)$= Public Attributes in Class i including inherited ones
$Pu(M_i)$= Public Methods in Class i including inherited ones
V_i= Visibility Factor
TMC & TAC =Total Methods & Attributes in class i
A_{ti} & M_{ti} =Total Attribute & Methods in class i

3.2 Private Factor (PrF): This factor measures the composite invisibility scope of methods and attributes at the class level and ranges between 0 and 1. PrF=0 if there are no methods and attributes with private scope, while PrF= 1 if there are no methods and attributes defined with public scope.

$$PrF = \frac{\sum_{i=1}^{TAC}(Pr(A_i)*(1-V_i)) + \sum_{i=1}^{TMC}(Pr(M_i)*(1-V_i))}{\sum_{i=1}^{TAC}A_{ti} + \sum_{i=1}^{TMC}M_{ti}}$$

$Pr(A_i)$= Private attributes in class i
$Pr(M_i)$= Private methods in class i
V_i= Visibility factor
TMC & TAC =Total methods & attributes in class i
A_{ti} & M_{ti} = Total attributes & methods in class i

4. COHESION METRICS

It is believed that higher cohesion results in better quality software. According to Briand *et al.* [7] various decisions need to be made to select the proper cohesion metrics. All these decisions are independent of each other. To select cohesion metrics, numerous studies have been done on OO software cohesion [6, 7, 9, 13, 28, 29, 45]. There are only a few cohesion measures that fulfil the cohesion properties [9], namely Tightly Class Cohesion (TCC) [32], Loosely Class Cohesion (LCC) [32], Cohesion (Coh) [7] and Lack of Cohesion of Method 5 (LCOM5). These metrics are in the same class as principal component analysis (i.e., in PC2) [19] and measure the same type of cohesion. Bieman and Kang [32] found redundancy in TCC and LCC values, due to the presence of constructors in calculating their values. Since the constructor initializes most of the variables in the class, there is a direct relationship between them. Moreover, any method in the class also references at least one attribute in the class (not necessarily the same attribute of the class) [7]. Bieman and Kang's [32] cohesion metric study has limited usefulness, and although the results are contrary to what they were expecting (i.e., the relationship between cohesion and reuse), they did not give any reason for the contradiction.

LCOM [12], LCOM1 [37], LCOM2 [13], LCOM3 [37], and LCOM4 [29] measure cohesion based on a pair of methods that refer to attributes directly, whereas LCOM5 [28] is a set of methods of a class referenced to attributes of a class. LCOM and LCOM1-4 are not normalised. Normalisation is required to obtain the maximum and minimum values for comparing different types of system. LCOM3 and LCOM4 both consider indirect connections between two methods, which is not the case with LCOM1 and LCOM2 [7]. Further LCOM4 addresses the accessed method problem (the presence of which decreases the cohesion), which is not addressed by LCOM1-3[7]. The definition of LCOM5 is based on the concept that the more sharing of instance variables by the methods in a class should indicate a greater single purpose of the class. This is defined as follows:

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

$$LCOM5 = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{(1-m)}$$

a = no. of attributes or instance variables
μ(Aj) = number of methods that access attribute Aj
m = no. of methods in the class
μ(Aj) is summed over all the attributes j = 1- n

LCOM5 [28] is normalized in the range 0 to 1 under the assumption that each attribute of a class is referenced by at least one method. But during maintenance activities some attributes may not be referenced and we must be prepared for that. In this case, the LCOM5 value will range between 0 and 2. LCOM5 takes its maximum value if there are no references to attributes at all. In this case, the value of LCOM5 is *m*/(*m - 1)*, where $m=|M_I(c)|$ is the number of methods of *c*. This is an anomaly in LCOM5. Briand *et al.* [7] remove this anomaly by defining a new metric Coh in terms of inheritance. This is also known as a variation of LCOM5 and is formally defined as:

$$Coh = (\sum_{j=1}^{a}\mu(A_j)) / (m.a)$$

This cohesion measure is normalized properly and ranges between 0 and

Furthermore it is not an inverse of cohesion like the other LCOM

measures.

5. ENCAPSULATION METRICS

According to its definition, encapsulation is composed of two concepts, namely data visibility and integrity or unity of the class. Class data visibility also includes the information it inherits from its parent classes. So while calculating the encapsulation protected and public data members of the inherited classes are added (in accordance with the visibility factor of the class) in our proposed metrics. The encapsulation factor (EncF) is composed of two sub factors (one for data visibility i.e., the Private Factor and the other for integrity i.e., Cohesion). The value of EncF ranges between 0 and 1. The optimal point for Cohesion (Coh) and PrF, expressed by the graph in Fig. 1, is the value [1, 1]. This shows that the ideal value for the Private Factor and Cohesion is 1, which means that all data should only be available for the class.

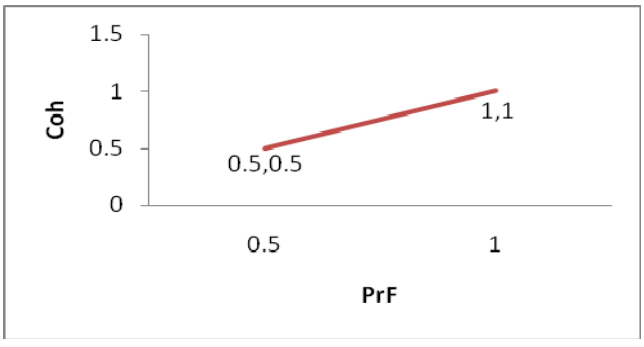

Figure 1. Plot of Coh and PrF to calculate EncF. When both Coh and PrF values are 1, the ideal condition is reached with EncF = 1.

In the graphical representation (Fig. 1) the x-axis denotes the Private Factor, while the y-axis represents cohesion in the same range, that is, [0 to 1]. The ideal value for the EncF of the class is 1, which is achieved when both Coh and PrF have values of 1. This condition represents 100% encapsulation.

The encapsulation factor is calculated using PrF as the data visibility and Coh as the unity or integrity factor of the class. Mathematically it is calculated by finding the distance between a point and the point (1,1). The distance between two points on a plane can be calculated using the distance formula:

$$Z^2=(X_2-X_1)^2+(Y_2-Y_1)^2.$$

Hence, the equation derived from this after normalization to obtain the EncF in the range 0 to 1 is given below:

$$EncF = 1 - \frac{\sqrt{(1-PrF)^2 + (1-Coh)^2}}{\sqrt{2}}$$

Next we investigate the role of this metric in identifying bad smells and faults in classes using statistical analysis.

6. BAD SMELLS IN CODE

Identifying bad smells in code helps to refactor the code. Refactoring is a process that improves maintainability, understandability, and readability of software projects. This further helps in facilitating evolution of the projects. The decision to refactor a module is not a simple one. A number of studies have been carried out on decision making for refactoring [8, 20, 23, 39, 40] and maintainability [37] based on software metrics. According to [25] 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. Making a judgment on the design without solid information is analogous to attempting to improve the performance without profiling the data. Collection and analysis of metric data is time consuming. Furthermore, it is not foolproof and interpretation of the metric results is difficult [25]. 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. Running a code review in parallel with the metric analysis helps in following best practices in coding standards for object-oriented concepts and further helps in unit testing.

Research results show that there is a relationship between structural attributes (design metrics) and external quality metrics (bad smell and class error tendencies) [4, 8,11,17,18, 26, 43]. Studies have also been done to find a relation between bad smells or anti-patterns and class error [16, 34, 36]. With the availability of bad smell information, the testing team can focus on classes for consideration. In this paper we set out to find the association between bad smells and software metrics. Previously not many studies have focused on this. Researchers [25, 38] 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 [16, 20, 38], where different developers have different views on the same module and the same bad smells. Metrics together with a code review help in making the decision to refactor, but an automated and easy to use environment is necessary to do the work. Automated tools help to identify smelly classes in a fraction of the time and with great accuracy. Such tools complement the work of the unit testing team. Finding bad smells helps the testing team to locate potential faults due to these bad smells [25, 36]. It also helps the manager determine whether the system needs refactoring and to plan accordingly. Thus, this is more beneficial if faulty class information is available before launching the software. Refactoring can be done by finding smelly classes. Which parameter 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 categories shown in Table 1[38].

The interested reader can find detailed definitions of each bad smell in [22] [46]. 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, 11 bad smells have been picked from

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

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 [31] 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.

7. DATA COLLECTION

Two Mozilla Firefox 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) [31]. The interested user can read about these metrics in detail in [12, 13, 28]. Selection of the metric criteria depends on the factor that it should cover for 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. For the collection of bugs, Bugzilla[1] database has been used.

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 Preventers	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 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	This category 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.

[1]https:// www.bugzilla.mozilla.org

Table 2. Categorised Bad Smell Distributions for Firefox Versions

Bad Smell Category	Bad Smell	Firefox 2.0	Firefox 3.0
Bloater	Large Method Long Parameter List Large Class	827	878
OO Abuser	Temporary Field	144	113
Change Preventer	Shotgun Survey	41	45
Dispensable	Lazy Class Data Class Speculative Generality	321	301
Coupler	Middle Man Feature Envy Inappropriate Intimacy	1203	1408

8. STATISTICAL METHOD

Regression analysis was used to analyse the results of the collected data. Linear regression was used to collect the set of independent software metric variables. Then, for selected independent variables, logistic regression [15] 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 another 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).

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 [35]. 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 [15]:

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

where β_0 is the intercept and β_1, β_2, β_3, *and* β_n are regression coefficients of x_1, x_2, x_3 *and* x_n, respectively.

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

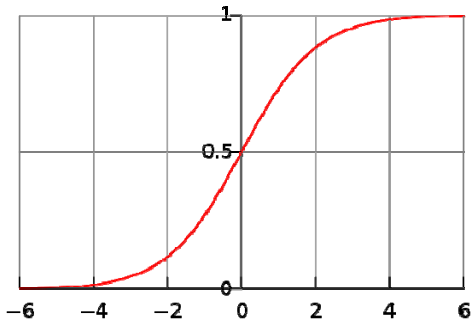

Figure 2. Logistic Curve (x- and y-axes represent the input and output parameters, respectively)

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 binary which check whether the class was smelly or not. 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 $f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \ldots + \beta_n x_n$ 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 statistical analysis was started with the following set of metrics covering each property of object-oriented programming: coupling (CBO) [4], polymorphism (RFC) [4], cohesion (Lack of Cohesion of Method (LCOM) [4] and (LCOM4)[30]), information hiding (PuF), encapsulat33ion (EncF), inheritance (NOC, DIT)[4], and abstraction (WMC) [4]. During the statistical analysis some of the metrics were dropped depending on the conditions of the statistical method.

9. EVALUATION OF METRICS

Descriptive statistics of the collected data i.e., Mozilla Firefox version 2.0 and 3.0 is shown in Tables 3. From Tables 3 it can be seen that the standard deviation of PuF and EncF is low because the values lie between 0 and 1. Apart from this, DIT has a low standard deviation because only a small number of classes were inherited. LCOM has zero values in almost 25% of the classes in all releases according to the data considered in [4, 9]. Basili [4] and Briand [9] noticed a certain deficiency in the definition of the cohesion metrics and as a result, Shatnawi [43] dropped the LCOM from his study. But now a new definition of cohesion (i.e., LCOM4) is available, which does not suffer from these deficiencies. Both LCOM and LCOM4 measure the different types of cohesion as reported by Etzkorn [19] (empirically) and as discussed earlier in Section 4 (theoretically).

Table 3. Descriptive Statistic Analysis

Metrics	Firefox Version 2.0				Firefox Version 3.0			
	Mean	Std Dev.	Min	Max	Mean	Std Dev.	Min	Max
NOC	0.97	16	0	1198	1.08	16.404	0	1132
DIT	1.97	1.926	0	12	2.14	2.02	0	11
LCOM	254.6	2264.38	0	115440	225.8	1451.15	0	55612
LCOM4	4.31	9.24	0	131	4.7	10.061	0	104
WMC	36.8	125.523	0	5301	40.04	116.305	0	3294
PuF	0.83	0.184	0	1	0.86	0.18	0	1
EncF	0.18	0.183	0	1	0.157	0.172	0	1
CBO	9.26	13.975	0	225	10.37	14.204	0	194
RFC	25.2	50.049	0	855	26.91	48.999	0	769
NOD	2.46	46.362	0	3856	2.84	50.825	0	3484

LCOM4 was included in this study along with LCOM. Inheritance (NOD, DIT & NOC), Cohesion (LCOM4), Coupling (CBO) and Polymorphism (RFC) values are same in both versions whereas Complexity value i.e. WMC is different in both the case.

9.1 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 4. Univariate Binary Regression Analysis

Metrics	Firefox 2.0		Firefox 3.0	
	B	p-value	B	p-value
NOC	-.268	.000	-.166	.000
NOD	-.020	.000	-.009	**.054**
DIT	.161	.000	.185	.000
CBO	.157	.000	.169	.000
RFC	.044	.000	.038	.000
WMC	.048	.000	.048	.000
LCOM	.002	.000	.001	.000
LCOM4	.087	.000	.051	.000
PuF	-3.486	.000	-3.121	.000
EncF	2.405	.000	1.583	.000

Table 4 presents the UBR analysis showing the association between bad smells and metrics. The metrics are significantly associated with a smelly

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

class if the *p-value* is less than 0.05. In Table 4 it can be seen that almost all the metrics are significantly associated with bad smells of the class.

No trend was found, which shows that some of the metrics are not associated with bad smells. 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 5 that the NOD metric is not associated with most of the categorised bad smells. EncF and PuF metrics are

Table 5. Univariate Multinominal Regression Analysis

Metrics	Category	Firefox 2.0		Firefox 3.0	
		B	p-value	*B*	p-value
NOC	Bloater	-.250	.000	-.157	.000
	Object-Oriented Abuser	-.375	.002	-.276	.017
	Change Preventer	.000	**.947**	.001	**.839**
	Dispenser	-.488	.000	-.085	.034
	Coupler	-.185	.000	-.095	.000
NOD	Bloater	-.015	.017	-.007	**.060**
	Object-Oriented Abuser	-.049	**.140**	-.018	**.272**
	Change Preventer	.001	**.668**	.001	**.350**
	Dispenser	-.256	.000	-.019	**.082**
	Coupler	-.014	.007	-.006	.035
DIT	Bloater	.256	.020	.246	.000
	Object-Oriented Abuser	.216	.041	.261	.000
	Change Preventer	-.374	**.140**	-.250	.031
	Dispenser	-.049	.039	-.045	**.219**
	Coupler	.163	.019	.196	.000
LCOM	Bloater	.003	.000	.002	.000
	Object-Oriented Abuser	.003	.000	.002	.000
	Change Preventer	.003	.000	.002	.000
	Dispenser	.002	.000	.002	.000
	Coupler	.003	.000	.002	.000
LCOM 4	Bloater	.136	.008	.086	.000
	Object-Oriented Abuser	.130	.010	.089	.000
	Change Preventer	.146	.012	.092	.000
	Dispenser	.045	.015	-.007	**.681**
	Coupler	.129	.008	.077	.000
WMC	Bloater	.053	.000	.053	.000
	Object-Oriented Abuser	.053	.000	.053	.000
	Change Preventer	.053	.000	.053	.000
	Dispenser	.039	.000	.042	.000
	Coupler	.052	.000	.051	.000
CBO	Bloater	.188	.000	.201	.000
	Object-Oriented Abuser	.185	.000	.201	.000
	Change Preventer	.181	.000	.190	.000
	Dispenser	.075	.000	.092	.000
	Coupler	.173	.000	.186	.000
RFC	Bloater	.054	.000	.049	.000
	Object-Oriented Abuser	.054	.000	.049	.000
	Change Preventer	.056	.000	.050	.000
	Dispenser	.021	.000	.018	.000
	Coupler	.050	.000	.045	.000
PuF	Bloater	-4.68	.000	-4.27	.000
	Object-Oriented Abuser	-4.17	.000	-4.10	.000
	Change Preventer	-3.35	.000	-3.31	.000
	Dispenser	-1.41	.000	-2.16	.000
	Coupler	-3.71	.000	-3.20	.000
EncF	Bloater	3.341	.000	2.363	.000
	Object-Oriented Abuser	1.199	.036	1.253	.022
	Change Preventer	3.523	.000	2.843	.000
	Dispenser	-.950	.040	-.561	**.176**
	Coupler	2.422	.000	1.374	.000

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 6 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 4 and 5). This shows that NOC values for VIF and tolerance are in the acceptable range. After dropping NOD we omitted RFC. In Table 6 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.

Table 6. Collinearity Statistics

Metric	Before Dropping Metrics				After Dropping Metrics			
	Tolerance		VIF		Tolerance		VIF	
	Ver 2.0	Ver 3.0	Ver 2.0	Ver 3.0	Ver 2.0	Ver 3.0	Ver 2.0	Ver 3.0
NOC	.029	.047	34.85	21.28	.998	.998	1.00	1.00
NOD	.029	.047	34.85	21.28	N/A	N/A	N/A	N/A
DIT	.723	.666	1.38	1.50	.726	.668	1.38	1.49
CBO	.170	.174	5.88	5.76	.281	.297	3.55	3.36
RFC	.084	.084	11.93	11.86	N/A	N/A	N/A	N/A
WMC	.129	.162	7.76	6.17	.173	.213	5.77	4.69
LCOM	.297	.340	3.37	2.94	.303	.357	3.30	2.81
LCOM4	.467	.398	2.14	2.51	.631	.551	1.58	1.81
PuF	.387	.362	2.58	2.76	.389	.362	2.57	2.76
EncF	.417	.378	2.39	2.64	.417	.378	2.39	2.64

9.2 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 [15] 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 7 shows the MLR model of metrics with the forward stepwise procedure. From Table 7 it can be seen that the new metrics (PuF and EncF) have a significant association with bad smells in predicting smelly classes. Table 8 gives the p-values for the goodness-of-fit test for the Likelihood Ratio and shows that the selected model is significant at 95% of confidence as a whole, rather than for individual metrics.

Table 7. Multilogistic Regression Model

	Firefox 2.0		Firefox 3.0	
Metrics	B	Sig.	B	Sig.
DIT	-.147	.000	N/A	N/A
CBO	.065	.000	.093	.000
LCOM	-.002	.000	.000	.013
LCOM4	.025	.000	-.050	.000
WMC	.049	.000	.044	.000
PuF	-2.087	.000	-2.362	.000
EncF	-1.657	.000	-1.867	.000
Constant	.148	.690	.304	.393

Table 8. Model Fitness Test

	Firefox 2.0	Firefox 3.0
Likelihood Ratio	.000	.000

9.3 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

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

model for each version of Firefox is presented in Table 9. From this table it can be seen that for version 3.0. DIT is not a significant predictor for the Bloater and OO Abuser categories 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 categories in each version. In other words, we can conclude that 4 out of the 5 smells categories 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 10 at the 95% level of confidence, where it was found that the metrics MMLR models in Table 9 are statistically significant.

Table 9. MMLR Analysis Results

Bad Smell Classification		Firefox Version 2.0		Firefox Version 3.0	
		B	Sig.	B	Sig
Bloater	Intercept	-1.316	.005	-.860	.055
	DIT	-.113	.000	-.034	**.235**
	CBO	.090	.000	.126	.000
	LCOM	-.002	.000	.000	.000
	LCOM4	.035	.000	-.042	.000
	WMC	.055	.000	.047	.000
	PuF	-2.549	.000	-3.007	.000
	EncF	-.533	**.326**	-1.041	.040
Object-Oriented Abuser	Intercept	1.419	.065	-.249	.763
	DIT	-.152	.003	-.037	**.467**
	CBO	.085	.000	.113	.000
	LCOM	-.002	.000	.000	.000
	LCOM4	.007	**.572**	-.042	.000
	WMC	.054	.000	.048	.000
	PuF	-5.765	.000	-4.940	.000
	EncF	-9.012	.000	-5.612	.000
Change Preventer	Intercept	-7.550	.000	-5.854	.000
	DIT	-1.122	.000	-.788	.000
	CBO	.070	.000	.085	.000
	LCOM	-.002	.000	.000	.000
	LCOM4	.122	.000	.029	**.087**
	WMC	.057	.000	.049	.000
	PuF	1.880	**.198**	.458	**.732**
	EncF	3.020	**.069**	1.766	**.241**
Dispenser	Intercept	1.606	.003	2.377	.000
	DIT	-.193	.000	-.143	.001
	CBO	.010	**.455**	.034	.011
	LCOM	-.002	.000	.000	.015
	LCOM4	-.015	**.318**	-.101	.000
	WMC	.046	.000	.044	.000
	PuF	-3.752	.000	-4.737	.000
	EncF	-6.041	.000	-6.523	.000
Coupler	Intercept	-.019	.962	.493	.207
	DIT	-.186	.000	-.055	.027
	CBO	.074	.000	.108	.000
	LCOM	-.002	.000	.000	.000
	LCOM4	.029	.000	-.053	.000
	WMC	.054	.000	.047	.000
	PuF	-2.446	.000	-2.873	.000
	EncF	-2.483	.000	-3.144	.000

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 9.

Table 10. Goodness of Fit Test for MMLR

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

10. METRICS MODEL ACCURACY

The accuracy of the proposed models was evaluated with ROC. ROC curves are two-dimensional graphs that visually depict the performance and performance trade-off of a classification model [21]. They were originally designed as tools in communication theory to visually determine optimal operating points for signal discriminators [21]. 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 4) 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.

Table 11. Accuracy of MLR Models for Bugs and Bad Smell Detection

MLR Models	Area under curve according to model in Table 7		Area under curve without PuF & Encf metrics	
	For faulty Classes	For Smelly Classes	For faulty Classes	For Smelly Classes
For Version 2.0	.843	.864	.838	.857
After applying Version 2.0 on 3.0	.837	.862	.830	.859
For version 3.0	.843	.860	.838	.857

Figure 3 and Table 11 show that the area under the curve can be interpreted as a measure of the performance of the discrimination model. Thus these results are acceptable. Separate MLR models with (Table 7) 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 area under the curve increased. After applying one version over the other we get the encouraging result which shows the ROC area under the curve in good classification area i.e. AUC >.80

In the case of the MMLR metrics model, the ROC area under the curve (AUC) predicted a well performing discrimination model for the Bloater and Object Oriented Abuser categories when the proposed model was applied to the same version 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 more area under the curve, than without including these metrics. Separate MMLR models for excluding the new metrics (PuF and EncF) were developed to calculate the AUC. In case of smelly class detection, the AUC with the inclusion of the new metrics is always greater for the 2nd, 4th and 5th categories. These categories include the major bad smells of object-oriented properties like abstraction in the 2nd category, unwanted code and complexity in the 4th category, and coupling and encapsulation in the 5th bad smell category. In other categories too AUC is greater with the new metrics inclusion than without it. This shows that the MMLR model is more acceptable than the MLR model.

In case of faulty class detection, for 1st and 2nd categories the AUC increases between 1% to 30% by including the new metrics. Table 12 shows that with new metrics inclusion, model for Bloater and OO Abuser bad smells categories can detect the faulty classes very well. From Table 12 it was seen that Version 2.0 model is more stable than

version 3.0 as it detect comparatively more or equally accurate faulty and smelly classes from version 3.0.

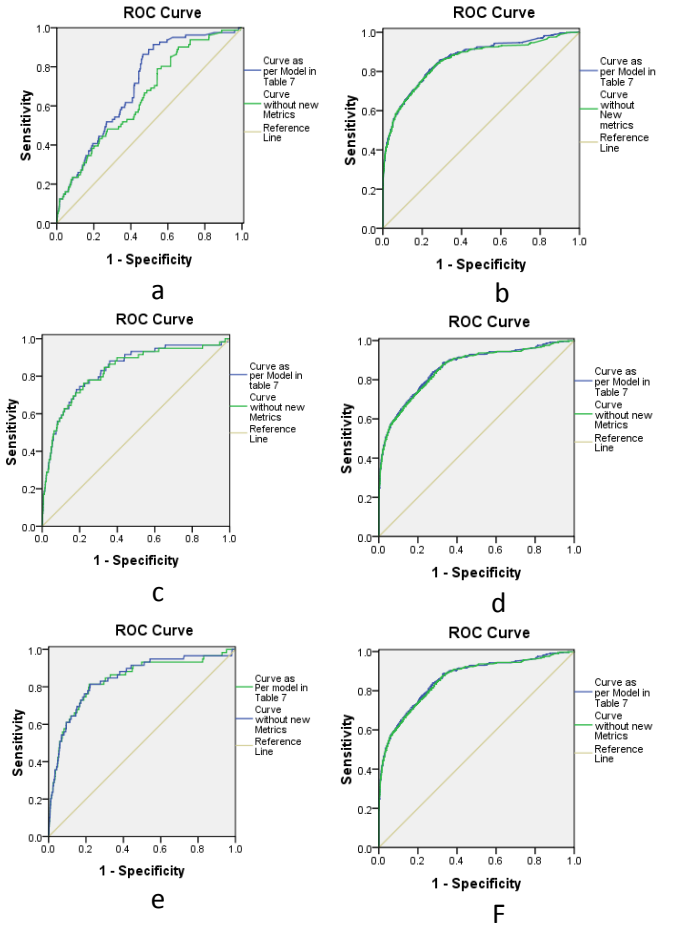

Figure 3. ROC curves for the MLR models. a) to identify faulty classes of ver. 2.0 after applying version 2.0 on 2.0, b) to identify smelly classes of ver. 2.0 after applying version 2.0 on 2.0 c) ROC Curve to identify faulty classes of 3.0 after applying version 2.0 on 3.0, d) ROC curve to identify smelly classes of 3.0 after applying version 2.0 on 3.0

The blue line shows the model accuracy as per model proposed in Table 7, while the green line shows the model accuracy without inclusion of the new metrics (PuF & EncF). The MLR model for the green line was developed after omitting the two new metrics.

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

Figure 4. Format of Confusion Matrix

11. DISCUSSION OF RESULTS

Refactoring is a process that helps in software evolution, maintenance and better understanding. The metrics model proposed over here provides facilitation to developers and managers about refactoring software. These models can also be used by the testing team to analyse the code before release. We carried out a statistical analysis on two versions of the open source Firefox software using the Columbus Framework tool. The metrics model for smelly classes proposed in this paper validates its usability to identify the smelly and faulty classes in same and subsequent version.

Empirical results obtained in this work indicate that there is a significant relationship between the metrics and bad smells and further bad smell and faulty classes. Association of metrics was observed across (using MLR) and with two bad smell categories (Bloater and Change preventer) in MMLR for smelly classes.

Table 12. Accuracy of MMLR Models to Detect Smelly and Faulty Classes

MMLR Models		Area under curve according to model in Table 9		Area under curve without PuF and Encf Metrics	
		Faulty Classes	Smelly Classes	Faulty Classes	Smelly Classes
Applying Version 3.0 to 3.0	Bloater	.813	.807	.640	.673
	OO Abuser	.837	.733	.644	.578
	Change Preventer	.273	.766	.462	.863
	Dispensable	.152	.649	.199	.583
	Coupler	.448	.623	.537	.572
Applying Version 2.0 to 2.0	Bloater	.742	.816	.731	.832
	OO Abuser	.704	.525	.492	.292
	Change Preventer	.224	.788	.222	.561
	Dispensable	.328	.650	.276	.613
	Coupler	.212	.583	.259	.314
Applying Version 2.0 to 3.0	Bloater	.830	.816	.849	.836
	OO Abuser	.546	.538	.184	.313
	Change Preventer	.473	.802	.145	.607
	Dispensable	.151	.639	.149	.595
	Coupler	.443	.725	.148	.299

Further, two bad smell categories (Bloater and OO Abuser) in MMLR has shown relation to analyse faulty classes. An empirical study of the new metrics (PuF and EncF) shows their vital role in the MMLR than the MLR analysis. In Bloater and OO Abuser categorical metrics models (with EncF and PuF metrics), the AUC for faulty class detection is in acceptable range (AUC > 0.70) when applied to the same version of metrics models. On the other hand, the other three categories show sufficient discrimination (at some places) when the predicted model is applied to the same and subsequent versions either for smelly class or for faulty class identification. From the empirical analysis it is clear that the new metrics (PuF and EncF) play a crucial role by pushing the curve to an acceptable discrimination area (towards the top left corner).

The categorisation model proposed in this paper can predict smelly classes more accurately in all cases of Bloater and Change Preventer categories. However, in other categories the smelly classes predicted by the MMLR model with new metrics are more favourable, but inconsistent. Another trend shows the critical role of the PuF and EncF metrics in almost all the categories, but especially in the OO Abuser, Dispensable, and Coupler categories. The new metrics measure the information hiding and encapsulation properties of object-oriented languages, with the same smells detected by the OO Abuser and Coupler categories, whereas the Dispensable category covers general design errors in object-oriented systems.

Only one system was analysed, namely Firefox, which is an open source system. This system is managed by a centralized team at Mozilla. These

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

results for software refactoring can be generalized to the same type of system or to different products of Mozilla. Refactoring results will assist the different software process life cycle teams (evolution, maintenance, and testing) in evaluating the software from different angles (using the categories model).

12. CONCLUSION

The purpose of this paper is to evaluate the role of metrics in measuring code quality with respect to software refactoring. We also evaluated two new metrics (PuF and EncF) and their role in identifying smelly and faulty classes for refactoring. First, we designed a binary metrics model and then a multinominal categorisation model to investigate the role of metrics (traditional and new) in identifying smelly classes. The results show that the binary regression model predicts smelly and faulty classes very well, whereas few categories were correctly identified with the multinominal model. The AUC for the binary model of smelly class detection is between 85% and 88%, but for the categorisation model it varies between 52% and 84%. With the inclusion of the PuF and EncF metrics, the AUC for the categorisation model varied from 5% to 58% (measuring the OO Abuser, Dispensable, and Coupler categories). This shows the significant role of the new metrics for diagnosing smelly classes. The new metrics role was also found to be critical in identification of faulty classes. From the above discussion, we conclude that Bloater and Change Preventer categorised smells were identified well with the metrics model. In addition, Bloater and OO Abuser categories metrics model (with EncF and PuF) can determine the faulty classes well. Also, we cannot ignore the role of new metrics in pin pointing the smelly classes under OO Abuser, Dispenser and Coupler categories.

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 those smelly classes which are identified by the Columbus Framework. However, these results will ease the work of project teams during development. The bad smell categorisation used in the study was defined subjectively [38]. Using another form of categorization may cause different results. The results may also differ if all the defined smells are considered. 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. REFERENCES

[1]Abreau, F.B., M. Goulão, R. Esteves, Toward the design quality evaluation of object-orientated software systems, Proc. 5th Int. Conf. On Software Quality, 1995.

[2]Abreau, F.B., W. Melo, Evaluating the impact of object-orientated design on software quality, Proc. 3rd International Software Metrics Symposium (METRICS'96), IEEE, Berlin, Germany, March, 1996.

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

[4]Basili, V.L., Briand, L., Melo, W.L., A validation of object-oriented metrics as quality indicators. IEEE Transactions on Software Engineering, 1996, 22(10), pp. 751–761.

[5]Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J et al. Manifesto for agile software development 2001. Available from http:// agilemanifesto.org/

[6]Bieman, J., Kang, B.K., Measuring Design Level cohesion, IEEE Transactions on software engineering, 1998,24(2), 111-124.

[7]Briand, L. C., Daly, J.W., Wust, J., (1998) A Unified Framework for Cohesion Measurement in Object Oriented Systems, Empirical Software Engineering Journal, 1998,3(1), 65-117.

[8]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.

[9]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.

[10]Cao Y., Zhu, Q., Improved Metrics for Encapsulation Based on Information Hiding, The 9th International Conference for Young Computer Scientists, 2008, 1(1), 742-747.

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

[12]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

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

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

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

[16]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.

[17]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.

[18]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.

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

[20]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.

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

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

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

[24]Grady RB, Successfully applying software metrics. IEEE Computer Vol 27, No. 9, pp. 18–25

[25]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.

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994

[26]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.

[27]Harrison, R., Counsell, S.J., Nithi, R.V., An Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Transactions on Software Engineering, 1998, 24, 491-496.

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

[29]Hitz, M. and Montazeri, B., Measuring Coupling and Cohesion in Object Oriented systems, International Symposium on Applied Corporate computing, Monterey, Mexico, 1995, 25-27.

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

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

[32]J.M. Bieman, J.M., Kang, B., Cohesion and Reuse in an Object-Oriented System, ACM System Symposium on Software Reusability, 1995, 259-262.

[33]Khan, R.A., Metric Based Testability Model for Object Oriented Design (MTMOOD) SIGSOFT Software Engineering Notes, 2009, 34(2), 1-6.

[34]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

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

[36]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

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

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

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

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

[41]Mayer T., Hall, T., A Critical Analysis of Current OO Design Metrics, Software Quality Journal, 1999, 8(2), 97-110.

[42]Mayer, T., Hall, T., Measuring OO Systems: A Critical Analysis of the MOOD Metrics, Proceedings of Technology of Object-Oriented Languages and Systems, Nancy, 1999, 108-117.

[43]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.

[44]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

[45]Tsui, F., Bonja, C., Duggins, S. and Karam, O., An Ordinal Metric for Intra-Method Class Cohesion, Proceedings of IADIS Applied Computing Conference, Algarve, Portugal, April 2008.

[46] Rule Package Description for FrontEndART Monitor. 2009:1-4 available at http://www.frontendart.com/sites/default/files/BSM.pdf

Appendix

Multivariate Logistic Regression Analysis Results (without PuF & EncF Metrics)

Metrics	Firefox 3.0		Firefox 2.0	
	B	Sig.	B	Sig.
DIT	N/A	N/A	-.118	.000
CBO	.092	.000	.062	.000
LCOM	.000	.018	-.002	.000
LCOM4	-.047	.000	.024	.000
WMC	.045	.000	.050	.000
Constant	-2.056	.000	-1.963	.000

Multivariate Multinominal Regression Analysis Results (without PuF & EncF Metrics)

Bad Smell Classification		Firefox Version 3.0		Firefox Version 2.0	
		B	Sig.	B	Sig.
Bloater	Intercept	-3.514	.000	-3.472	.000
	DIT	-.011	.678	-.086	.006
	CBO	.122	.000	.088	.000
	LCOM	.000	.000	-.002	.000
	LCOM4	-.042	.000	.029	.000
	WMC	.046	.000	.053	.000
Object-Oriented Abuser	Intercept	-5.395	.000	-4.910	.000
	DIT	.006	.905	-.104	.052
	CBO	.110	.000	.082	.000
	LCOM	.000	.000	-.002	.000
	LCOM4	-.037	.000	.014	.281
	WMC	.047	.000	.052	.000
Change Preventer	Intercept	-5.032	.000	-5.219	.000
	DIT	-.764	.000	-1.055	.000
	CBO	.079	.000	.063	.000
	LCOM	.000	.000	-.002	.000
	LCOM4	.026	.104	.110	.000
	WMC	.046	.000	.053	.000
Dispenser	Intercept	-2.717	.000	-2.489	.000
	DIT	-.052	.203	-.107	.012
	CBO	.023	.087	.002	.875
	LCOM	.000	.107	-.001	.000
	LCOM4	-.095	.000	-.012	.410
	WMC	.042	.000	.039	.000
Coupler	Intercept	-2.548	.000	-2.556	.000
	DIT	-.010	.674	-.141	.000
	CBO	.105	.000	.072	.000
	LCOM	.000	.000	-.002	.000
	LCOM4	-.049	.000	.029	.000
	WMC	.045	.000	.052	.000

DOI: 10.1145/2020976.2020994 http://doi.acm.org/10.1145/2020976.2020994