New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems

Béla Újházi[1], Rudolf Ferenc[1], Denys Poshyvanyk[2] and Tibor Gyimóthy[1]

[1]*University of Szeged, Hungary*
Department of Software Engineering
ujhazi.bela@stud.u-szeged.hu, {ferenc, gyimi}@inf.u-szeged.hu

[2]*The College of William and Mary, USA*
Computer Science Department
denys@cs.wm.edu

Abstract

The paper presents two novel conceptual metrics for measuring coupling and cohesion in software systems. Our first metric, Conceptual Coupling between Object classes (CCBO), is based on the well-known CBO coupling metric, while the other metric, Conceptual Lack of Cohesion on Methods (CLCOM5), is based on the LCOM5 cohesion metric. One advantage of the proposed conceptual metrics is that they can be computed in a simpler (and in many cases, programming language independent) way as compared to some of the structural metrics.

We empirically studied CCBO and CLCOM5 for predicting fault-proneness of classes in a large open-source system and compared these metrics with a host of existing structural and conceptual metrics for the same task. As the result, we found that the proposed conceptual metrics, when used in conjunction, can predict bugs nearly as precisely as the 58 structural metrics available in the Columbus source code quality framework and can be effectively combined with these metrics to improve bug prediction.

1. Introduction

Coupling and cohesion measures capture the degree of interaction and relationships among source code elements, such as classes, methods, and attributes in object-oriented (OO) software systems. One of the main goals behind OO analysis and design is to implement a software system where classes have high cohesion and low coupling among them. These class properties facilitate comprehension activities, testing efforts, reuse, and maintenance tasks.

A vast majority of coupling and cohesion metrics abound in the literature relies on structural information, which captures relations, such as method calls or attributes usages. These metrics have been proved useful in different tasks, such as, assessment of design quality [4, 10], impact analysis [8, 36, 41], prediction of software quality [26], and faults [17, 22, 37], identification of design patterns [2] etc. However, these structural metrics lack the ability to identify conceptual links, which, for example, specify implicit relationships encoded in identifiers and comments in source code.

In this paper we propose two new conceptual metrics, namely Conceptual Coupling between Object Classes (CCBO) and Conceptual Lack of Cohesion of Methods (CLCOM5) metrics. The proposed metrics are different from existing conceptual coupling metrics [35] and cohesion [31] metrics as they utilize different counting mechanisms inspired by peer structural cohesion and coupling metrics.

In order to evaluate the proposed metrics, we compare CCBO and CLCOM5 against a large host of existing structural and conceptual coupling metrics for predicting faults in a large open-source software system. Furthermore, we perform a comprehensive empirical evaluation of other parameters, such as, impact of pre-processing techniques. Such parameters also impact performance of other existing conceptual metrics, such as Conceptual Cohesion of Classes (C3) [31] and Conceptual Coupling among Classes (CoCC) [35]. The results of our empirical study indicate that CCBO and CLCOM5 not only can be used to build operational models for predicting fault-proneness of classes, but can also be effectively used in conjunction with other structural metrics to improve overall accuracy of bug prediction models.

Our paper warrants the following contributions:

- We define two new conceptual cohesion and coupling metrics, which are easier to compute than their structural protégé.
- We carried out an extensive empirical study of 61 software metrics, including newly proposed measures to build models for fault prediction using machine learning and logistic regression analyses.
- We empirically studied a range of parameters that can impact performance of CCBO and CLCOM5 metrics, such as, impact of corpus stemming and parameterized thresholds.
- We developed an online appendix summarizing the results of our empirical study to facilitate development and comparison of conceptual metrics and ensure reproducibility of our results.

2. Conceptual Metrics

Our approach to measuring coupling and cohesion relies on the assumption that the methods and classes of Object-Oriented systems are connected in more than one way. While the most explored and evaluated set of relations among methods and classes are based on data and control dependencies, in this work we rely on orthogonal type of relationships, known as conceptual dependencies to capture conceptual cohesion and coupling of classes.

Conceptual coupling and cohesion metrics, such as, CoCC and C3 extract, encode, and analyze the semantic information embedded in the comments and identifiers in software. Software developers utilize the comments and identifiers to represent elements of the problem or solution domain [11, 15]. Whilst conceptual cohesion [31] and coupling [35] metrics capture this information and have been proposed elsewhere in the research literature, we augment a family of conceptual metrics with two new members, namely CCBO and CLOM5.

Our metrics rely on the equivalent underlying mechanism to extract and analyze the conceptual information from the identifiers and comments in source code as previous conceptual metrics, which are based on Latent Semantic Indexing (LSI) [14]. LSI has been used before to support other source code analysis tasks such as concept location [34], identification of abstract data types [29], clone detection [40], traceability link recovery among software artifacts [1, 13, 30], software clustering [25], quality assessment [26] and software measurement [16, 31, 32, 35, 36]. For the sake of completeness we provide some of the details on the LSI in the next section.

2.1 Latent Semantic Indexing in the Nutshell

LSI is a machine-learning model that induces representations of the meaning of words by analyzing the relations among words and documents in textual corpus of data. LSI was initially developed in the context of information retrieval as a way of overcoming issues with polysemy and synonymy, which are inherent to the vector space model (VSM) [38]. The specific technique, which is used by LSI to capture vital conceptual information and tackle two aforementioned problems, is dimension reduction, which implies selecting the top dimensions from a co-occurrence term-document matrix decomposed using singular value decomposition (SVD). Consequently, LSI provides an effective mechanism to assess and evaluate similarity amid any two documents in the text corpus (*i.e.*, methods in software) in an unsupervised fashion. While the details behind SVD are rather complex and lengthy to be presented in this paper, we refer a reader to [14].

LSI relies on VSM, which is an extensively used approach for encoding documents in the corpus as numerical vectors. More specifically, VSM encodes a corpus by a term-by-document matrix whose $[i, j]^{th}$ element indicates the association between the i^{th} term and j^{th} document. In case of our particular application, a term is an identifier or a comment, and a document is a body of the method extracted from a source code file. The foundation of VSM lies in the mechanism that represents documents by its association with terms where the association is measured by term co-occurrences in the documents. There are several mechanisms to capture these associations based on term occurrences such as term frequencies (default case in our empirical evaluation) and term frequency – inverse document frequency (tf-idf). In a term-by-document matrix, a tf-idf value for $[i, j]^{th}$ element implies a statistical measure evaluating how important a word is to a document in a corpus. Formally,

$$w_{t,d} = tf_{t,d} \cdot \log N / df_t$$

where $tf_{t,d}$ is the term frequency of a document d, and df_t is the term frequency in all the documents in the corpus, whereas the N is the number of documents in the corpus. The importance of a word increases proportionally to the number of times a word appears in the document, but is offset by the number of times of that word appearing in the corpus [38].

The conceptual similarity between documents is measured via the cosine or inner product between the corresponding vectors (*i.e.*, methods), which increases if more words are shared. This underlying mechanism entirely supports the idea of measuring conceptual coupling and cohesion in software based on word matching from identifiers and comments in software.

2.2 Conceptual Cohesion & Coupling Metrics

The definitions of the new conceptual cohesion and coupling of classes builds on our previous work for measuring the conceptual cohesion [32] and coupling [36] of classes. The source code of the software system is parsed and transformed into a corpus of textual documents where each document corresponds to the implementation of a method. Aforementioned LSI technique takes the corpus as an input and creates a term-by-document matrix, which captures the dispersion and co-occurrence of terms in class methods. SVD is used next to construct a subspace, referred to as the LSI subspace. All methods from this matrix are represented as vectors in the LSI subspace. The cosine similarity between two vectors is used as a measure of conceptual similarity between two methods and is purported to determine shared conceptual information between two methods in the context of the entire software system. This mechanism to capture conceptual similarity among documents has been introduced before in Conceptual Coupling of Classes and Conceptual Cohesion of Classes measures and is also used here.

Next we define the model, CCBO, and CLOM5 measures. Some of the definitions have been presented elsewhere [35], however, we also include them for the sake of completeness.

2.3 Principal Definitions

Definition 1 (*System, Classes, Methods*). We define an OO system as a set of classes $C = \{c_1, c_2 \ldots c_n\}$ with the number of classes in the system $n = |C|$. A class has a set of methods. For each class $c \in C$, $M(c) = \{m_1, \ldots, m_t\}$ represent its set of methods, where $t = |M(c)|$ is the number of methods in a class c. The set of all the methods in the system is denoted as M(C).

An OO system C can be also viewed as a set of connected graphs $G_C = \{G_1, .., G_n\}$ with G_i representing class c_i. Each class $c_i \in C$ is also represented by a graph $G_i \in G_C$ such that $G_i = (V_i, E_i)$, where $V_i = M(c_i)$ is a set

of vertices corresponding to the methods in class c_i and $E_i \subset V_i \times V_i$ is a set of weighted edges that connect pairs of methods from the class.

Definition 2 (***C**onceptual **S**imilarity between **M**ethods*). The *conceptual similarity between methods (CSM)* $m_k \in M(C)$ and $m_j \in M(C)$, $CSM(m_k, m_j)$, is computed as the cosine amid two vectors vm_k and vm_j, representing m_k and m_j in the LSI semantic space:

$$CSM(m_k, m_j) = \frac{vm_k^T vm_j}{|vm_k|_2 \times |vm_j|_2}$$

As defined, the value of $CSM(m_k, m_j) \in [-1, 1]$, as CSM is a cosine similarity in the LSI space. In order to fulfill non-negativity property of software metrics [9], we refine CSM as the following:

$$CSM^1(m_k, m_j) = \begin{cases} CSM(m_k, m_j) & if \quad CSM(m_k, m_j) \geq 0 \\ else \quad 0 \end{cases}$$

CSM^1 has been used as a base for defining C3 [31] and CoCC [35] measures before.

Definition 3 (***P**arameterized **C**onceptual **S**imilarity*)

In our work we define conceptual cohesion and coupling metrics utilizing counting mechanisms, stemming from existing structural metrics, which are sensitive to the input information such as nodes and edges (*e.g.*, methods and attribute references). Thus, in this work we introduce a notion of a parameterized conceptual similarity, which distinguishes among significant and non-significant conceptual interactions among methods of classes.

In particular, we conjecture that it is possible to empirically derive a threshold for a given software system to distinguish between strong and weak conceptual similarities. More formally, we define parameterized CSM^P as:

$$CSM^P(m_k, m_j, t) = \begin{cases} 1 & if \quad CSM^1(m_k, m_j) \geq t \\ else \quad 0 \end{cases}$$

Of course, the particular threshold *t* depends on the specific software system. In our previous experience, the absolute value of the cosine similarity can not be used as a reliable indicator of presence or absence of conceptual relationship among pairs of methods as more comprehensive analysis of similarity distributions is required. One of the main research questions in our empirical evaluation is centered on empirically deriving such a threshold and analysis of the impact on the choice of threshold values on the resulting metrics.

2.4 Conceptual Lack of Cohesion in Classes

In this paper we define our first metric, namely CLCOM5 using CSM^P as the foundation for computing conceptual similarities among methods of classes, however, in terms of counting mechanism we rely on one of the ideas from previously defined structural metrics, namely LCOM5 [23], graph based cohesion metric. The main difference between our metric, CLCOM5 and C3, conceptual cohesion of classes metric, is that we define a parameterized version of cohesion metric using a different counting mechanism:

$$CLCOM5(c, x) = NoCC(G),$$

where *NoCC* identifies the number of connected components in the graph $G_C = (M(c), E)$, $c \in C$, $E \subseteq M(c) \times M(c)$, and $(m_k, m_j) \in E$ if $CSM^P(m_k, m_j, t) = 1$.

2.5 Conceptual Coupling between Object Classes

The definition of CCBO relies on previous definitions for CoCC metric. We provide these definitions and explain how we adjusted them in the current work.

Let $c_k \in C$ and $c_j \in C$ be two distinct ($c_k \neq c_j$) classes in the system. Each class has a set of methods $M(c_k) = \{m_{k1}, \ldots, m_{kr}\}$, where $r = |M(c_k)|$ and $M(c_j) = \{m_{j1}, \ldots, m_{jt}\}$, where $t = |M(c_j)|$. Between every pair of methods (m_k, m_j) there is a similarity measure $CSM^P(m_k, m_j)$. We can similarly define the conceptual similarity between two classes c_j and c_k, that is CSC^P , as follows:

$$CSC^P(c_k, c_j, t) = \begin{cases} 1 & if \quad CSC^1(c_k, c_j) \geq t \\ else \quad 0 \end{cases}$$

The definition ensures that the conceptual similarity between two classes is symmetrical, as $CSC(c_k, c_j) = CSC(c_j, c_k)$. In this case we use class granularity to build the corpus. This is the main difference between computing CLCOM5 and CCBO metrics. We refine the conceptual similarity for a class c as the following:

$$CCBO(c, t) = \sum_{c_k \in C, c \neq c_k} CSC^P(c, c_k, t),$$

which is the sum of the parameterized conceptual similarities between a class c and all the other classes in the system.

3. Empirical Case Study

In this section we present the design of the empirical case study aimed at comparing CLCOM5 and CCBO with other structural and conceptual coupling metrics for the task of predicting bugs in open-source software as well as identifying and analyzing various factors impacting performance of the proposed measures. The description of the study follows the *Goal-Question-Metrics* design presented in [6]. The data, which has been used to generate the results in this paper, was previously used in [36].

3.1 Definition and the Context

Our primary *goals* include comparing new conceptual metrics against existing coupling and cohesion metrics and determining whether combining the metrics can support the task of predicting bugs in large open-source software. In this empirical study the *quality focus* was on establishing orthogonality among CCBO, CLCOM5 and existing coupling and cohesion metrics and improving on accuracy of bug prediction, while the *perspective* was of a software developer analyzing a release of a software system for possible

faults. The context of this case study consists of a large open-source software system, that is, *Mozilla*, which is implemented in a mix of programming languages spanning from C/C++, Java, IDL, XML, HTML, to JavaScript. It should be noted that we analyzed only C++ classes from the source code and computed CCBO, CLCOM5 and other structural and conceptual metrics among object-oriented classes implemented in C++ only.

3.1.1. Cohesion and coupling metrics. In order to determine whether the newly proposed metrics capture new dimensions in coupling measurement, we selected 61 exiting structural and conceptual metrics for comparison, including coupling metrics (*e.g.*, CBO, RFC), cohesion metrics (*e.g.*, Coh, Coh, LCOM1, LCOM2, LCOM3), CK [12] metrics suite as well as other metrics implemented in our metrics collection tool, namely Columbus [20]. In addition to these structural metrics we also considered a conceptual cohesion metric, that is, C3 [31]. Other guiding criteria that we used to choose the metrics is availability of the results reported for these metrics elsewhere in the literature [10, 22] to facilitate systematic comparison and evaluation of the results.

3.1.2. Subject software system. For our case study we have chosen one large real-world software system. *Mozilla* [1] is an open-source Web browser ported to almost all identified software and hardware platforms. It is as large as many industrial size programs and is developed mostly in C++. We do not analyze the parts of *Mozilla* written in other programming languages, such as, C, Java, IDL, XML, HTML, etc. In our case studies, we use the source code of version 1.6 of *Mozilla*. It should be noted that we opted to work with such a system to emulate real-world settings where analyzing the source code written in such a detailed programming language as C++ introduces difficulties in addition to compiling such systems on different platforms.

3.1.3. Building and indexing text corpora. In order to compute CCBO and CLCOM5 metrics we first need to generate a corresponding corpus for the software system. To build such a corpus for *Mozilla* we extracted the textual information, *i.e.*, identifiers and comments, from the source code using method level granularity, where each document in the corpus represents a method from the software system (that is, a sequence of identifiers and comments implementing corresponding method). More specifically we extracted the following textual information: (1) comments, (2) local and attribute variable names, (3) user defined types, (4) methods names, (5) parameter lists and (6) names of the called methods. It should be noted that the comments preceding or proceeding the code have been extracted using similar heuristics to [21], which have been implemented in our Columbus reverse engineering framework. Finally, we opted for not including the names of the primitive types in the corpus and we considered those to be a part of our stop word list.

Once a corpus is built, we index it through Latent Semantic Indexing using the term-by-document co-occurrence matrix corresponding to the corpus. LSI captures important conceptual relationships (*i.e.*, couplings) among methods and classes within the corpus. After modeling the corpus using LSI, conceptual coupling and cohesion metrics can be computed (for the details on how CCBO and CLCOM5 are computed using underlying textual information refer to Section 2). The next section describes all the necessary settings for other researchers who are willing to reproduce the results of our empirical study.

3.1.4. Settings of the case study. All the structural coupling measures were computed using *Columbus* [20]. Columbus is a reverse engineering framework that contains the components for analyzing arbitrary C/C++ source code and presenting the extracted information in any desired form. In this case study, we used the compiler wrapper technology of Columbus to extract the facts from *Mozilla*'s source code. For more details on how compiler wrapping is done in Columbus refer to our previous work [19, 22].

The textual information needed to compute CCBO and CLCOM5 has been also extracted using the Columbus framework. We used a cross-platform numerical analysis and data processing library ALGLIB[2] to compute the Singular Value Decomposition, which is needed for the LSI algorithm.

Since we used method level granularity to construct the corpus for *Mozilla* we extracted all types of methods from classes in the source code, including constructors, destructors, and accessors. Comments and identifiers were extracted from the body of each method as well before and after given Columbus' heuristics. The resulting text from the source code was pre-processed using the following parameters: some of the tokens were eliminated (*e.g.*, operators, special symbols, numbers, reserved keywords of the C++ programming language, primitive data types standard library function names including standard template library); the identifier names in the source code were split into original words based on observed coding standards and naming conventions, *e.g.*, Google's C++ coding standard[3]. For instance, the following identifiers are split into words 'lack', 'of' and 'cohesion': 'LackOfCohesion', 'Lack_of_cohesion', etc. During this indexing process, LSI does not utilize a predefined vocabulary, or a predefined grammar, hence no morphological analysis or transformations were performed, such as abbreviation expansion. However, we build various corpora with and

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

[2] http://www.alglib.net

[3] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

without stemming to study its impact on the metrics in our empirical study.

3.1.5. Predicting faults using machine learning algorithms and software metrics. In order to evaluate the usefulness of our metrics we conducted a number of analyses to discover possible relationships between the values of the metrics and the number of bugs found in *Mozilla*'s classes. We employed regression analysis methods along with machine learning techniques, which are widely used to predict an unknown variable based on one or more a priori known variables. We opted to use these statistical techniques to study relationship between the metrics and faults in classes.

The logistic regression method predicts if a class is faulty or not, but does not infer a probable number of bugs in classes. We used *univariate* logistic regression analysis to examine each metric separately and *multivariate* analysis to study common effectiveness of the combinations of various metrics. In addition to regression analyses we exploited machine learning methods to predict the fault-proneness of classes. While similar studies have been done in the past on the CK metrics [22], in this paper we apply machine learning techniques on 61 conceptual and structural metrics to predict fault-proneness in software. We utilize a suite of machine learning techniques implemented in Weka[4], an open-source collection of machine learning algorithms for data mining tasks. The usage of this toolset simplifies the validation part, since these algorithms are well-documented and easy to use. In particular we use Naïve Bayes, Bayesian Logistical Regression, Bayes Net, Logistic Regression, RBF Network, Simple Logistic Regression, SMO, IB-k, Conjunctive Rule, Decision Table, ADTree and REP Tree. The Naïve Bayes is a statistical based algorithm based on probabilistic models. IB-s is a k-means clustering algorithm combined with a simple classifier. The difference between the logistic and simple logistic regression analyses are that logistic regression makes multi-nominal regression by using all of the given predictors, however, simple logistic regression may eliminate some of those. SMO is a sequential minimal optimization algorithm for training a support vector classifier. RBF network is a Gaussian radial basis function network, which is akin to artificial neural network, which uses radial basis functions as activation functions in the neurons. Some of these algorithms are rule/tree based algorithms. For instance, Conjunction Rule is rule based and it provides conjunction rules for classifying labels. Decision Table generates complex logical rules from the given learning examples and it generates rules to classify incoming labels. ADTree is an alternative decision tree algorithm, which generates rules for the cases of metric values. Finally, REPTree is a fast decision tree learner, which builds a decision/regression tree via information gain/variance and prunes it using reduced-error with back-fitting.

All the models were trained to provide binary predictions which imply that they predict if a class is prone to be faulty or not based on the values or combination of values of particular metrics. In order to estimate the performance of generated predictive models we utilized the ten-fold cross-validation technique. As for the training bug data we utilized the bug data that was gleaned and used in our previous work elsewhere [22].

3.2 Research Questions

We address the following research questions (RQ) within the context of this empirical study.

- **RQ_1**: Are the new metrics, CCBO and CLCOM5, orthogonal as compared to existing structural and conceptual coupling and cohesion metrics?
- **RQ_2**: How does stemming impact accuracy of CCBO and CLCOM5 for predicting fault-proneness of classes?
- **RQ_3**: What is the optimal threshold for CCBO and CLCOM5 for predicting fault-prone classes?
- **RQ_4**: Does combining CCBO and CLCOM5 with existing structural and conceptual cohesion and coupling metrics improve accuracy of predicting fault-prone classes?

To respond to our research questions we compare CCBO and CLCOM5 with other coupling and cohesion metrics as well as explore the impact of combining coupling metrics.

3.3 Evaluation of Metrics and Analysis

Precision, recall and *accuracy* are three widely used information retrieval metrics that were employed to measure performance of software metrics for various maintenance tasks including predicting fault-prone classes. We explain these measures in the context of *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN) in the context of fault-proneness prediction.

True Positive is a candidate class, which was predicted as faulty and contained a bug, whereas a True Negative is a candidate class predicted as non-faulty and containing no bugs. The False Positive is class that was marked as faulty by the model, but actually did not contain a fault, whereas a False Negative is a class where the model marked a class as non-faulty and it did contain a bug. *Accuracy*, *Precision* and *Recall* are defined as the following:

$$\text{Accuracy}=\frac{TP+TN}{TP+FP+TN+FN},\ \text{Prec}=\frac{TP}{TP+FP},\ \text{Recall}=\frac{TP}{TP+FN}$$

While reporting the results we also used *F-measure*, which is a harmonic mean of precision and recall.

In order to explore the principal, orthogonal dimensions captured by the coupling and cohesion measures (both conceptual and structural) we performed Principal Component Analysis (PCA) on the metrics. Applying PCA to metrics data consists of the following

[4] http://www.cs.waikato.ac.nz/ml/weka/

Table I. Ten-fold cross validation of conceptual & structural metrics with & w/o stemming for predicting faults

	Conceptual-no-stem				All-metrics-no-stem				Conceptual-with-stem				All-metrics-with-stem			
ML Algorithm	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**
Bayesian Log. Reg.	68.3	70.5	69.1	69.8	71.8	76.7	67.3	71.7	68.8	70.4	71.2	70.8	71.5	71.5	74.7	72.3
Bayes Net	67.0	64.7	83.4	72.8	70.5	72.5	71.5	72.0	67.3	65.0	83.1	73.0	70.5	72.4	71.8	72.1
Naïve Bayes	68.1	67.9	75.9	71.7	69.2	73.1	66.3	69.6	68.5	67.9	77.2	72.3	69.1	73.1	66.3	69.5
Logistic Regression	67.6	73.6	61.0	66.7	72.5	76.6	69.5	72.8	67.9	72.6	63.4	67.7	72.4	77.0	68.4	72.4
RBF Network	67.1	70.1	66.4	68.2	69.3	70.7	71.9	71.3	68.7	68.8	75.1	71.8	69.7	71.9	70.5	71.2
Simple Logistic	67.6	73.5	60.9	66.6	72.1	75.9	69.6	72.6	67.8	72.6	63.4	67.7	71.8	75.6	69.2	72.3
SMO	67.8	73.8	61.1	66.9	72.4	76.2	69.8	72.8	68.0	72.5	64.1	68.0	72.2	76	69.7	72.7
IB-k	66.6	67.9	70.3	69.1	71.2	73.1	72.5	72.8	68.8	70.5	70.9	70.7	72.7	74.5	73.8	74.2
Conjunctive Rule	65.8	79.9	47.6	59.6	69.6	81.8	55.0	65.8	64.5	73.1	52.6	61.2	69.5	82.1	54.3	65.4
Decision Table	67.8	65.8	81.9	73.0	70.3	73.6	68.5	71.0	68.1	66.5	80.7	72.9	70.5	74.3	67.9	71.0
AD Tree	68.4	65.9	84.2	73.9	70.9	72.8	72.3	72.5	68.3	65.7	84.6	74.0	71.0	74.4	69.3	71.7
REP Tree	67.3	67.8	73.2	70.4	71.2	72.6	73.6	73.1	67.6	68.5	72.3	70.3	70.6	72.2	72.6	72.4

steps: collecting the metrics data, identifying outliers, and performing PCA. We applied PCA in the similar manner as in our previous work [31, 35], including procedures for identifying outliers and rotating principal components. In general, via PCA we can recognize groups of variables (*i.e.*, metrics), which are likely to measure the same underlying dimension (*i.e.*, specific mechanism that defines, for instance, coupling or cohesion) of the object to be measured (*e.g.*, cohesion of a classes).

3.4 Case Study Results

3.4.1. RQ1 – Results of the principal component analysis. PCA was performed on 3,625 classes from *Mozilla* (that is, classes for which we could compute all the metrics) with 61 structural and conceptual metrics. While the complete results are quite lengthy to be presented in this paper, we summarize some of the results and provide the link to the complete results in the online appendix[5], which also contains the brief explanations of the metrics mentioned in the following.

The PCA resulted in 11 Principal Components (PCs) that describe 87.6% of the variance in our data set. We provide top four PCs with their interpretations:

PC_1 (27%): There are several metrics which were included in this component: cohesion metrics LCOM-LCOM5, NLMA, NLMAni and our ***CLCOM5***, size metrics NML, NMLD, NAML, NAL, NMLDpub, NMLpub, LOC, lLOC, coupling metrics NFMA, NOI and RFC, and the WMC complexity metric. These clusters of the results are consistent with previous work with some changes in the rankings of the PCs [10].

PC_2 (21%): This component was comprised of several coupling metrics RFC_1, RFC_2 and RFC_3, inheritance-based metrics AID, DIT, NOA, NMI and various size metrics, such as, NM, NMpub, NMprot, NMD, NMDpub, NMDprot, NAM.

PC_3 (7.2%): This component was described mostly by NMLDpriv, NMDpriv, NMLpriv and NMpriv metrics. As it can be seen in the results for the other RQs, these metrics' prediction performance were quite offset from the other variants of these metrics, such as, NMLDpub, NMLDprot.

PC_4 (6%): This component consisted of the structural cohesion Co_1, Co_2, Coh metrics, ***CCBO*** and C3 conceptual metrics.

In addition to PCA we also analyzed correlations among the metrics. While we pinpoint a few interesting observations in this paper, we refer the interested reader to the online appendix for the complete analysis results.

CCBO correlated with CLCOM5 with a coefficient of 0.41 and a few other structural metrics, such as Coh, CBO, RFC with a coefficient between 0.4 and 0.5. On the other hand, CLCOM5 was highly correlated with many other structural metrics such as LOC, LLOC, NOI, CBO, RFC and WMC with a correlation coefficient above 0.7. These results indicate that the new conceptual cohesion and coupling metrics are closer to structural metrics as previously defined conceptual metrics, such as C3. This result can be interpreted as a positive result as conceptual metrics are less expensive to compute compared to many structural metrics and do not depend on the specific programming language at hand as well as building specialized parsers for those languages and systems to derive the metrics.

3.4.2. RQ2 – identifying impact of stemming on CCBO and CLCOM5 metrics for predicting fault-prone classes. The conceptual metrics rely on the quality of the underlying comments and identifiers in source code as well as specific pre-processing strategies used to transform the corpus before indexing. While previous work did not look closely into this important factor, we perform close investigation of the impact of stemming on the performance of conceptual metrics and their combinations with structural metrics to identify fault-prone classes. The goal of this investigation is to identify whether stemming helps in building better models for predicting faults, which utilize conceptual metrics.

[5] http://www.cs.wm.edu/semeru/scam10-conceptual-metrics

The results of the ten-fold cross-validation of various configurations of the models using and not using stemming are presented in Table I. The first part of the table presents the results of applying several machine learning techniques for predicting bugs in *Mozilla* on the models built using three conceptual metrics (*i.e.*, CCBO, CLCOM5 and C3) without stemming. As it can be seen, the performance of these models in terms of precision (P), recall (R), accuracy (A) and F-measure (F) are quite high as compared, for instance to random classifiers. While the performance of the metrics are rather consistent across various machine leaning algorithms, we identify that the AD Tree algorithm produces the highest accuracy, recall and F-measure values (*i.e.*, 68.4%, 84.2% and 73.9% respectively), while Conjunctive rule achieves the highest precision.

It should also be noted that the results of combining new conceptual metrics (without stemming) for predicting fault-proneness is comparable to the combination of structural metrics (see Table II). Furthermore, the models based on conceptual metrics are able to outperform the models based on structural metrics in terms of recall and f-measure (*i.e.*, 84.2 vs. 73% and 73.9% vs. 72.4 respectively).

When we compare the results of combining all the structural metrics (*i.e.*, all-metrics-no-stem in Table I) against conceptual metrics without stemming (*i.e.*, conceptual-no-stem in Table I), we can observe slight improvement in the accuracy (that is, 72.5% vs. 68.4%) and precision (that is, 81.8% vs. 79.9%), while the best recall and F-measure are obtained with conceptual metrics (that is, 84.2% and 73.9%, respectively).

Table II. Ten-fold cross validation of the structural metrics for predicting fault-proneness

ML Algorithm	A	P	R	F
Bayesian Log. Reg.	70,5	71.8	73.0	72.4
Bayes Net	69.9	72.1	70.6	71.4
Naïve Bayes	69.1	73.2	66.1	69.5
Logistic Regression	72.1	76.6	68.5	72.3
RBF Network	68.9	75.0	62.1	67.9
Simple Logistic	71.7	75.2	69.9	72.3
SMO	71.4	74.7	69.9	72.2
IB-k	70.2	72.3	71.0	71.6
Conjunctive Rule	70.1	81.7	56.4	66.7
Decision Table	70.6	75.3	66.5	70.7
AD Tree	70.9	75.2	67.5	71.1
REP Tree	70.5	72.9	70.5	71.7

According to the results while applying stemming (see conceptual-with-stem in Table I), we have positive improvements in case of accuracy, recall and F-measure. Moreover, we can observe that this improvement is consistent for these parameters across different machine learning algorithms utilized. We can also observe a noticeable improvement in recall and F-measures for conceptual metrics with stemming over structural metrics.

Finally, the results for combining conceptual metrics with stemming (all-metrics-with-stem in Table I) and all the structural metrics leads to the conclusion that this combination produces the best values across all the parameters, such as, accuracy, precision, recall and F-measure (*i.e.*, 72.7%, 82.1%, 74.7% and 74.2% respectively). Likewise, the models with all the metrics and stemming outperforms the model, which is based on a combination of pure structural metrics (see all-metrics-with-stem in Table I and Table II).

Based on these results we conclude that *stemming does improve the results for predicting fault-prone classes*. According to our best knowledge, this is the first research result in the literature, which empirically confirms the positive impact of stemming on conceptual metrics given positive impact on the external software quality attribute, such as fault-proneness of classes. Assuming this result we apply stemming from now on to answer the remaining research questions.

3.4.3. RQ3 – Identifying optimal thresholds for CCBO and CLCOM5 metrics for predicting fault-proneness of classes. CCBO and CLCOM5 are parameterized metrics, which depends on the threshold *t* to identify conceptual similarities among class methods. While we used a default threshold of 0.7 to answer RQ_2, it is necessary to identify acceptable values of this parameter for the given task. We acknowledge that the process of identifying an optimal threshold could be software system specific, thus, we present the results for *Mozilla* only.

In order to search for the optimal thresholds for CCBO and CLCOM5 metrics on our dataset we computed accuracy values of the metrics across various thresholds starting from 0.05 until 0.95 with a step of 0.05. It should be noted that we used a reduced set of machine learning algorithms in this case, which corresponded to the subset of algorithms indicating a superior performance in RQ_2. According to our results (see Figure 1) it can be seen that the thresholds for CLCOM5 resulting in the accuracy of at least 64% reside in the interval [0.3, 0.95], whereas the peak performance of 68.5% in accuracy is observed in the interval of [0.7, 0.8]. These results are consistent across all the machine learning algorithms used in this situation.

On the other hand, the accuracy of CCBO is more sensitive to threshold values as compared to CLCOM5 metric. Here we observe that the accuracies of the algorithms slowly decline from 0.05 threshold. This finding is quite interesting suggesting that we should assign higher thresholds for CLCOM5 cohesion metric and lower thresholds for CCBO coupling metrics to warrant better prediction accuracy of fault-proneness.

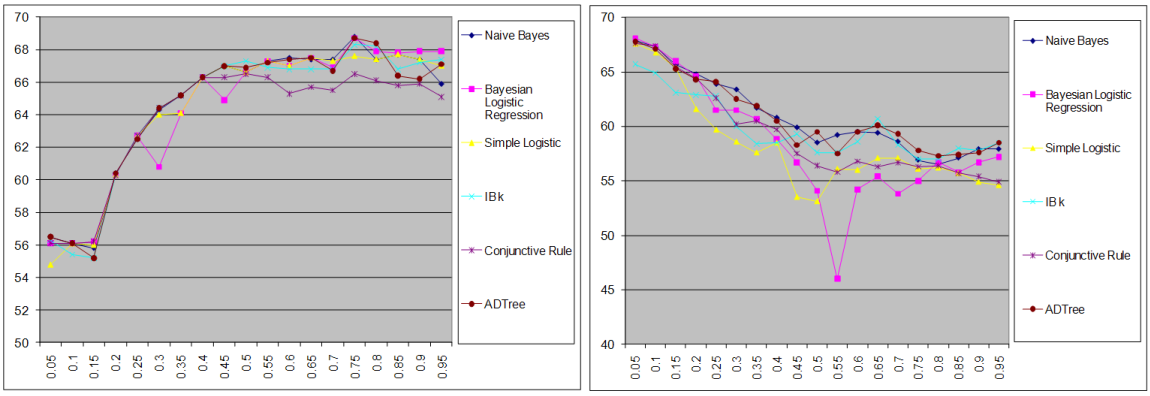

Figure 1. CLCOM5 (left) and CCBO (right) accuracies across different thresholds

While using the best thresholds (see Table V) for CLCOM5 and CCBO we observed some improvement in CLCOM5 over LCOM5 while predicting fault-prone classes in terms or accuracy (that is, 68.8% vs. 64.6%), recall (that is, 72.2% vs. 71.3%) and F-measure (that is, 70.8% vs. 68.1%). CCBO measure appears to be better at recall (that is 74.6% vs. 72.8%). Finally both conceptual measures, CLCOM5 and CCBO, outperform existing C3 measure in terms of accuracy, precision and F-measure.

3.4.4. RQ4 – Results of combining CCBO and CLCOM5 with structural and conceptual metrics for fault-proneness. Lastly, we tested if combining CCBO, CLCOM5 and structural metrics improves the performance of models for fault prediction as compared to combinations of C3 & structural metrics.

Based on the results we conclude that combining CCBO and CLCOM5 with structural metrics prediction models are more robust than combinations of existing conceptual metric C3 and structural metrics. We derive these conclusions based on the analysis of the average results of accuracy, precision and recall measures.

Table III. Combining CCBO and CLCOM5 with structural (left) and C3 and structural metrics (right)

	CCBO,CLCOM5+struct			C3+struct		
Algorithm	**A**	**P**	**R**	**A**	**P**	**R**
Bayes. Log. Reg.	71.6	74.8	70.1	71.7	73.1	73.9
Bayes Net	70.7	72.6	72.0	70.3	72.3	71.6
Naïve Bayes	69.2	73	66.5	68.9	73	65.9
Logistic Reg.	72.0	75.8	69.6	71.8	76.3	68.1
RBF Network	69.8	72.4	69.8	69.8	72.4	69.5
Simple Logistic	71.6	75.5	69.0	71.9	75.6	69.5
SMO	72.0	74	72.7	72.1	75.6	70.1
IB-k	73.0	75.4	72.9	72.3	74.7	72.2
Conjunctive Rule	69.7	81.1	56.0	69.9	80.9	56.7
Decision Table	70.4	74.1	67.9	70.4	74.1	67.9
AD Tree	71.0	72.9	72.3	70.5	74.3	67.9
REP Tree	71.1	73.4	71.3	70.6	72.6	71.6

3.4.5. Analyzing metric intervals. In addition to answering the research questions we examined the proposed metrics more closely. In particular we analyzed histograms of distributions of faulty classes across metric intervals, where *x*-axis represents metric intervals and *y*-axis shows faulty (dark grey) and non-faulty (light gray) classes (see Figure 2).

Interestingly enough, all the three conceptual metrics, existing C3 and proposed CCBO and CLCOM5 reflect our underlying hypotheses. In other words, C3 captures more faulty classes while the metrics values are low, likewise the CLCOM5 metric; whereas CCBO captures more faulty classes when the metrics values are getting higher.

3.4.6. Results for the logistic regression analysis. We also decided to examine individual performance of the metrics using *univariate* logistic regression. The set-up of this study was similar to our previous work [22, 32].

According to the results, which present top 12 performing metrics (out of 61 metrics) according to accuracy values, CCBO and CLCOM5 were not the best measures. However, CLCOM5 appears to be the best measure in the family of cohesion metrics and CCBO appears to be one of the best coupling structural metrics besides CBO, RFC and RFC3. This result further supports the usefulness of proposed metrics.

Table IV. Results of regression analysis

Metric	Acc.	Prec.	Rec.	Metric	Acc.	Prec.	Rec.
CBO	71.9	74.1	72.4	lLOC	68.9	76.4	59.8
NOI	71.4	76.8	66.1	RFC3	68.9	77.7	58.1
WMC	70.3	77.7	61.8	LOC	68.7	76.9	58.6
RFC	69.8	75.9	63.2	CLCOM5	67.5	73.5	60.9
NFMAni	69.8	75.5	63.7	CCBO	66.7	66.7	74.6
NFMA	69.3	72.9	67.2	NAML	66.7	74.7	56.4

3.5 Threats to validity

We recognize some issues that could have affected the results of the case study and may have limited our interpretations. We have demonstrated that our metrics are more similar to some of the structural metrics than to existing conceptual metrics; however, we obtained these results by analyzing classes from only one large C++ open-source system. In order to generalize the results, large-scale assessment is needed taking into account software from diverse domains, implemented in different programming languages and environments.

The conceptual measures (that is, CCBO and CLCOM5) depend on consistent and concise naming conventions for identifiers and comments. When these

Table V. Ten-fold cross validation of CLCOM5, LCOM5, CCBO and C3 for predicting faults in classes

	CLCOM5; t=0.75				LCOM5				CCBO; t=0.1				CBO				C3			
Algorithm	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**
Naïve Bayes	68.8	71.0	69.8	70.4	62.8	66.8	59.3	62.8	67.3	68.8	70.3	69.5	71.9	73.9	72.8	73.3	65.4	63.3	83.2	71.9
Bayesian Log. Reg.	68.7	70.2	71.4	70.8	61.0	69.7	46.9	56.1	67.3	68.4	71.5	69.9	71.9	74.1	72.4	73.2	55.3	54.4	97.0	69.8
Simple Logistic	67.6	73.5	61.0	66.6	64.6	65.2	71.3	68.1	66.8	66.7	74.6	70.5	71.9	74.1	72.4	73.2	55.4	54.5	97.0	69.8
IB-k	68.3	69.4	72.2	70.8	64.6	65.4	70.9	68.0	64.9	64.8	74.2	69.2	71.9	74.1	72.4	73.2	65.6	63.0	85.3	72.5
Conjunctive Rule	66.5	73.4	57.9	64.7	64.6	65.2	71.3	68.1	67.4	68.9	70.5	69.7	70.5	77.7	62.3	69.2	61.8	58.5	96.5	72.8
AD Tree	68.7	70.2	71.4	70.8	64.6	65.2	71.3	68.1	67.1	68.6	70.3	69.4	71.9	74.1	72.4	73.2	65.5	63.3	83.1	71.9

are missing, the emphasis for capturing coupling or cohesion should be placed on static or dynamic metrics.

CCBO and CLCOM5 measures, as currently defined, do not take into account polymorphism and inheritance. The measures only consider methods of a class that are implemented or overloaded in the class.

In our case study for predicting fault-proneness we used one large (*i.e.*, *Mozilla*) software system, however, to permit for generalization of the results, yet again, large-scale evaluation is needed, which should take into account several releases of software systems implemented in multiple languages (that is, not only C++ as covered in our case study).

We also observed that the machine learning algorithms did not generate the best models in every case. In other words, we did not investigate collinearity among the metrics to identify similar groups of metrics to improve the predictive power for the models. Instead, we utilized all the software metrics generated by Columbus. We will redirect our future research efforts to study this phenomenon.

Our metrics rely on parameterized conceptual similarities among methods, which assume specifying a threshold for operational measures. While we used near-optimal threshold values (as indicated via analysis of all other possible threshold values), these threshold values may vary for other software systems. Our future work will target not only identifying ranges of acceptable threshold values, but also guidelines to users of the metrics on how to identify these thresholds.

4. Related Work

Our related work can be broadly classified into two areas – conceptual cohesion and coupling metrics and predicting fault-proneness of classes.

Conceptual cohesion of classes or C3 [31] is one of the first conceptual metrics proposed in the research literature. C3, similarly to CLCOM5 and CCBO is based on the analysis of the semantic information embedded in the source code, such as identifiers and comments. C3 has been recently used in conjunction with structural cohesion metrics to predict faults in object-oriented classes [32]. CoCC [35] is a conceptual coupling metric, also based on LSI, stems from C3, however, it was defined to capture coupling among classes based on conceptual similarities among methods in different classes. CoCC has been shown to outperform structural coupling metrics for the task of impact analysis on a large open-source system [36]. Finally, WME is a conceptual cohesion metric based on Latent Dirichlet Allocation and information theory approaches [28]. This cohesion metric has been shown to capture different aspects of class cohesion and improved fault prediction for most existing cohesion metrics. While building comprehensive models for fault prediction was not at the focus of papers presenting conceptual metrics, this paper not only introduces new metrics, but also explores their role in building complete models for fault prediction.

Existing research showed that software metrics can be used as good indicators for the fault proneness of classes in OO systems [3, 5, 7, 10, 17, 22, 33, 37, 39]. More specifically, some of the existing approaches also utilized machine learning [22] and logistic regression analyses [3, 5, 7, 10, 22, 33, 39] to build metric-based models for fault prediction. Our paper is different from the previous work as it defines new conceptual metrics for class cohesion and coupling, which appear to be an improvement over the state-of-the-art. Finally, this work explores a set of machine learning techniques and regression analyses to test a number of models based on the combinations of structural and conceptual metrics along with the detailed investigation into principal factors impacting the performance of the conceptual metrics. Finally, prediction of fault-prone classes or simply bug prediction is an active area of research, which produced a number of research publications in the last decade. Besides conference and journal publications

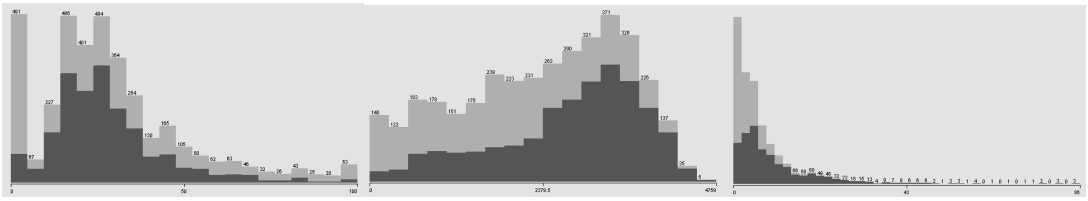

Figure 2. Distribution of (non) faulty across C3 (left), CCBO (center) and CLCOM5 (right) metric intervals

on the topic, specialized conferences were organized such as PROMISE[6] and MSR[7] with their specialized data sets for predicting fault-prone classes in software.

5. Conclusion

The paper defines novel operational measures for conceptual class cohesion and coupling measurement, which have been empirically validated. An extensive case study using machine learning techniques on metrics data indicates that the proposed measures have comparable accuracy with those defined suing structural information. Moreover, combinations of novel metrics with existing host of measures attests statistically significant improvement in the results across multiple evaluation criteria.

The paper lays the basis for the future work that makes use of conceptual information for coupling and cohesion measurement. The proposed metrics could be further extended and refined, for instance, by taking into account inheritance. Another direction is to improve the quality of the underlying textual information by applying advanced source code pre-processing techniques for splitting [18] and expanding [24, 27] compound identifiers and comments in software. Since both CCBO and CLCOM5 rely on textual (unstructured) information, we are considering including external documentation in the corpus, which should extend the context in which words are used in software to capture underlying conceptual similarities.

6. Acknowledgment

We acknowledge Malcom Gethers for preparing the online appendix. This work was supported in part by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences, the Hungarian national grants OTKA K-73688, TECH 08-A2/2-2008-0089 and GOP-1.1.2-07/1-2008-0007, and NSF CCF-1016868. Any opinions, findings and conclusions expressed herein are the authors; and do not necessarily reflect those of the sponsors.

7. References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Soft. Engineering*, vol. 28, no. 10, pp. 970 - 983., 2002

[2] Antoniol, G., Fiutem, R., and Cristoforetti, L., "Using Metrics to Identify Design Patterns in Object-Oriented Software", in Proc. of 5th IEEE METRICS'98, Bethesda, MD, pp. 23 - 34., 1998

[3] Arisholm, E., Briand, L. C., and Foyen, A., "Dynamic coupling measurement for OO software", *IEEE TSE*, vol. 30, no. 8, pp. 491-506., 2004

[4] Bansiya, J. and Davis, C. G., "A hierarchical model for object-oriented design quality assessment", *IEEE TSE*, vol. 28, no. 1, pp. 4-17., 2002

[5] Basili, V. R., Briand, L. C., and Melo, W. L., "A Validation of OO Design Metrics as Quality Indicators", *IEEE TSE*, vol. 22/10, Oct., pp. 751-761., 1996

[6] Basili, V. R., Caldiera, G., and Rombach., D. H., *The Goal Question Metric Paradigm*, John W & S, 1994.

[7] Briand, L., Melo, W., and Wust, J., "Assessing the Applicability of Fault-Proneness Models across OO Software Projects", *TSE*,vol.28/7,706-720., 2002

[8] Briand, L., Wust, J., and Louinis, H., "Using Coupling Measurement for Impact Analysis in OO Systems", in IEEE ICSM'99, pp. 475-482., 1999

[9] Briand, L. C., Daly, J., and Wüst, J., "A Unified Framework for Coupling Measurement in OO Systems", *IEEE TSE*, vol. 25/1, pp. 91-121., 1999

[10] Briand, L. C., Wüst, J., Daly, J. W., and Porter, V. D., "Exploring the relationship between design measures and software quality in object-oriented systems", *Journal of System and Software*, vol. 51, no. 3, pp. 245-273., 2000

[11] Caprile, B. and Tonella, P., "Restructuring Program Identifier Names", in Proc. of 16th IEEE ICSM'00, San Jose, California, USA, pp. 97-107., 2000

[12] Chidamber, S. R. and Kemerer, C. F., "Towards a Metrics Suite for Object Oriented Design", in Proc. of OOPSLA'91, pp. 197-211., 1991

[13] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *TOSEM*, vol. 16, no. 4, 2007.

[14] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, pp. 391-407. , 1990

[15] Deissenboeck, F. and Pizka , M., "Concise and Consistent Naming", *Software Quality Journal*, vol. 14, no. 3, pp. 261-282, 2006

[16] Dit, B., Poshyvanyk, D., and Marcus, A., "Measuring the Semantic Similarity of Comments in Bug Reports", in Proc. of 1st STSM'08, 2008

[17] El-Emam, K. and Melo, K., "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", *NRC/ERB-1064*, vol. NRC 43609, 1999.

[18] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K., "Mining Source Code to Automatically Split Identifiers for Software Analysis", in Proc. of 6th IEEE MSR, Vancouver, BC, Canada, pp. 71-80., 2009

[19] Ferenc, R., Beszedes, A., and Gyimóthy, T., "Extracting Facts with Columbus from C++ Code", in Proc. of 8th CSMR"04, March, pp. 4-8., 2004

[20] Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T., "Columbus - Reverse Engineering Tool and Schema for C++", in ICSM'02, 172-181, 2002

[21] Fluri, B., Würsch, M., Giger, E., and Gall, H., "Analyzing the co-evolution of comments and source code", *SQJ,* vol. 17/ 4, pp. 367-394., 2009

[22] Gyimóthy, T., Ferenc, R., and Siket, I., "Empirical validation of OO metrics on open source software for fault prediction", *TSE*,vol.31/10,Oct' 2005.

[23] Henderson-Sellers, B., *Software Metrics*, U. K., Prentice Hall, 1996.

[24] Hill, E., Fry, Z. P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., and Vijay-Shanker, K., "AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools", in MSR'08.

[25] Kuhn, A., Ducasse, S., and Gîrba, T., "Semantic Clustering: Identifying Topics in Source Code", *IST*, vol. 49/3, pp. 230-243, 2007

[26] Lawrie, D., Feild, H., and Binkley, D., "Leveraged Quality Assessment Using Information Retrieval Techniques", in ICPC'06, pp. 149-158., 2006

[27] Lawrie, D., Feild, H., and Binkley, D., "Extracting Meaning from Abbreviated Identifiers", in Proc. of 7th IEEE SCAM'07, Paris, France, 2007.

[28] Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimóthy, T., and Chrisochoides, N., "Modeling Class Cohesion as Mixtures of Latent Topics", in *ICSM'09*.

[29] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in ICSE'01, pp. 103-112., 2001

[30] Marcus, A. and Maletic, J. I., "Recovering Documentation-to-Source-Code Traceability Links using LSI", in ICSE'03, pp. 125-137.

[31] Marcus, A. and Poshyvanyk, D., "The Conceptual Cohesion of Classes", in Proc. of 21st IEEE ICSM'05, Budapest, Hungary, pp. 133-142., 2005

[32] Marcus, A., Poshyvanyk, D., and Ferenc, R., "Using the Conceptual Cohesion of Classes for Fault Prediction in OO Systems", *TSE*, vol. 34/2, pp. 287-300., 2008

[33] Olague, H., Etzkorn, L., Gholston, S., and Quattlebaum, S., "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes", *IEEE TSE*, vol. 33, no. 6, pp. 402-419., 2007

[34] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and IR", *TSE*, vol. 33/6, pp. 420-432., 2007

[35] Poshyvanyk, D. and Marcus, A., "The Conceptual Coupling Metrics for Object-Oriented Systems", in IEEE ICSM'06, Phil., PA, pp. 469 - 478., 2006

[36] Poshyvanyk, D., Marcus, A., Ferenc, R., and Gyimóthy, T., "Using Information Retrieval based Coupling Measures for Impact Analysis", *Empirical Software Engineering*, vol. 14, no. 1, pp. 5-32., 2009

[37] Quah, T.-S. and Thwin, M. M. T., "Application of neural networks for software quality prediction using OO metrics", in ICSM'03,pp. 116-125.

[38] Salton, G. & McGill, M., *Introduction to Modern IR*, McGraw-Hill, 1983.

[39] Subramanyam, R. and Krishnan, M. S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", *IEEE TSE*, vol. 29, no. 4, pp. 297-310., 2003

[40] Tairas, R. and Gray, J., "An Information Retrieval Process to Aid in the Analysis of Code Clones", *ESE*, vol. 14, no. 1, pp. 33-56., 2009

[41] Wilkie, F. G. and Kitchenham, B. A., "Coupling measures and change ripples in C++ application software", *JSS*, vol. 52, pp. 157-164., 2000.

[6] http://promisedata.org/

[7] http://msr.uwaterloo.ca/