Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems

Andrian Marcus, *Member*, *IEEE Computer Society*,
Denys Poshyvanyk, *Student Member*, *IEEE*, and Rudolf Ferenc

Abstract—High cohesion is a desirable property of software as it positively impacts understanding, reuse, and maintenance. Currently proposed measures for cohesion in Object-Oriented (OO) software reflect particular interpretations of cohesion and capture different aspects of it. Existing approaches are largely based on using the structural information from the source code, such as attribute references, in methods to measure cohesion. This paper proposes a new measure for the cohesion of classes in OO software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. The measure, named the Conceptual Cohesion of Classes (C3), is inspired by the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics. This paper presents the principles and the technology that stand behind the C3 measure. A large case study on three open source software systems is presented which compares the new measure with an extensive set of existing metrics and uses them to construct models that predict software faults. The case study shows that the novel measure captures different aspects of class cohesion compared to any of the existing cohesion measures. In addition, combining C3 with existing structural cohesion metrics proves to be a better predictor of faulty classes when compared to different combinations of structural cohesion metrics.

Index Terms—Software cohesion, textual coherence, fault prediction, fault proneness, program comprehension, information retrieval, Latent Semantic Indexing.

1 Introduction

Software modularization, Object-Oriented (OO) decomposition in particular, is an approach for improving the organization and comprehension of source code. In order to understand OO software, software engineers need to create a well-connected representation of the classes that make up the system. Each class must be understood individually and, then, relationships among classes as well. One of the goals of the OO analysis and design is to create a system where classes have high cohesion and there is low coupling among them. These class properties facilitate comprehension, testing, reusability, maintainability, etc.

Software cohesion can be defined as a measure of the degree to which elements of a module belong together [8]. Cohesion is also regarded from a conceptual point of view. In this view, a cohesive module is a crisp abstraction of a concept or feature from the problem domain, usually described in the requirements or specifications. Such definitions, although very intuitive, are quite vague and make cohesion measurement a difficult task, leaving too much room for interpretation. In OO software systems, cohesion is usually measured at the class level and many different OO cohesion metrics have been proposed (see Section 2 for details) which try capturing different aspects of cohesion or reflect a particular interpretation of cohesion.

Proposals of measures and metrics for cohesion abound in the literature (see Section 2) as software cohesion metrics proved to be useful in different tasks [24], including the assessment of design quality [5], [13], productivity, design, and reuse effort [18], prediction of software quality, fault prediction [30], [39], [69], modularization of software [14], [54], and identification of reusable of components [32], [50].

Most approaches to cohesion measurement have automation as one of their goals as it is impractical to manually measure the cohesion of classes in large systems. The trade-off is that such measures deal with information that can be automatically extracted from software and analyzed by automated tools and ignore less structured but rich information from the software (for example, textual information). Cohesion is usually measured on structural information extracted solely from the source code (for example, attribute references in methods and method calls) that captures the degree to which the elements of a class belong together from a structural point of view. These measures give information about the way a class is built and how its instances work together to address the goals of their design. The principle behind this class of metrics is to measure the coupling between the methods of a class. Thus, they give no clues as to whether the class is cohesive from a conceptual point of view (for example, whether a class implements one or more domain concepts) nor do they give an indication about the readability and comprehensibility of the source code. Although other types of metrics were proposed by researchers (see Section 2 for details) to capture different aspects of cohesion, only a few such

- *A. Marcus and D. Poshyvanyk are with the Department of Computer Science, Wayne State University, 5143 Cass Avenue, Detroit, MI 48202. E-mail: {amarcus, denys}@cs.wayne.edu.*
- *R. Ferenc is with the Department of Software Engineering, University of Szeged, H-6720 Szeged, Hungary. E-mail: ferenc@inf.u-szeged.hu.*

Manuscript received 21 Aug. 2006; revised 22 June 2007; accepted 18 Oct. 2007; published online 1 Nov. 2007.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0200-0806.
Digital Object Identifier no. 10.1109/TSE.2007.70768.

metrics address the conceptual and textual aspects of cohesion [33], [59].

We propose a new measure for class cohesion, named the Conceptual Cohesion of Classes (C3), which captures the conceptual aspects of class cohesion, as it measures how strongly the methods of a class relate to each other conceptually. The conceptual relation between methods is based on the principle of textual coherence [52]. We interpret the implementation of methods as elements of discourse. There are many aspects of a discourse that contribute to coherence, including coreference, causal relationships, connectives, and signals. The source code is far from a natural language and many aspects of natural language discourse do not exist in the source code or need to be redefined. The rules of discourse are also different from the natural language.

C3 is based on the analysis of textual information in the source code, expressed in comments and identifiers. Once again, this part of the source code, although closer to natural language, is still different from it. Thus, using classic natural language processing methods, such as propositional analysis, is impractical or unfeasible. Hence, we use an Information Retrieval (IR) technique, namely, Latent Semantic Indexing (LSI), to extract, represent, and analyze the textual information from the source code. Our measure of cohesion can be interpreted as a measure of the textual coherence of a class within the context of the entire system.

Cohesion ultimately affects the comprehensibility of source code. For the source code to be easy to understand, it has to have a clear implementation logic (that is, design) and it has to be easy to read (that is, good language use). These two properties are captured by the structural and conceptual cohesion metrics, respectively.

This paper is organized as follows: An overview of other cohesion metrics for OO systems is presented in Section 2, emphasizing the type of information used in the computation of the metrics and the measuring mechanisms. Section 3 describes the principles and technology behind the C3 metric and formally defines the metric, giving an example as well. Section 4 presents two case studies aimed at comparing C3 with an extensive set of existing cohesion measures and assessing its ability to predict faults in the source code, in combination with the existing metrics. Finally, limitations, future work, and conclusions are presented in Sections 6 and 7.

2 Related Work: Cohesion Measures for OO Software Systems

There are several different approaches to measure cohesion in OO systems. Many of the existing metrics are adapted from similar cohesion measures for non-OO systems (we are not discussing those here), while some of the metrics are specific to OO software.

Based on the underlying information used to measure the cohesion of a class, one can distinguish structural metrics [8], [11], [20], [41], [42], [51], [78], semantic metrics [33], [59], information entropy-based metrics [1], slice-based metrics [61], [64], metrics based on data mining [62], and metrics for specific types of applications like knowledge-based [47], aspect-oriented [75], and distributed systems [21].

The class of structural metrics is the most investigated category of cohesion metrics and includes lack of cohesion in methods (LCOM),[1] LCOM3 [42], LCOM4 [42], C_o (connectivity) [42], LCOM5 [41], Coh [11], TCC (tight class cohesion) [8], LCC (loose class cohesion) [8], ICH (information-flow-based cohesion) [51], NHD (normalized Hamming distance) [22], etc.

The dominating philosophy behind this category of metrics considers class variable referencing and data sharing between methods as contributing to the degree to which the methods of a class belong together. Most structural metrics define and measure relationships among the methods of a class based on this principle. Cohesion is seen to be dependent on the number of pairs of methods that share instance or class variables one way or another. The differences among the structural metrics are based on the definition of the relationships among methods, system representation, and counting mechanism. A comprehensive overview of graph theory-based cohesion metrics is given by Zhou et al. [76]. Somewhat different in this class of metrics are LCOM5 and Coh, which consider that cohesion is directly proportional to the number of instance variables in a class that are referenced by the methods in that class. Briand et al. defined a unified framework for cohesion measurement in OO systems [11] which classifies and discusses all of these metrics.

Recently, other structural cohesion metrics have been proposed, trying to improve existing metrics by considering the effects of dependent instance variables whose values are computed from other instance variables in the class [16], [77], [78]. Other recent approaches have addressed class cohesion by considering the relationships between the attributes and methods of a class based on dependence analysis [17]. Although different from each other, all of these structural metrics capture the same aspects of cohesion, which relate to the data flow between the methods of a class.

Other cohesion metrics exploit relationships that underline slicing [61], [64]. A large-scale empirical investigation of slice-based metrics [61] indicated that the slice-based cohesion metrics provide complementary views of cohesion to the structural metrics. Although the information used by these metrics is also structural in nature, the mechanism used and the underlying interpretation of cohesion set these metrics apart from the structural metrics group.

A small set of cohesion metrics was proposed for specific types of applications. Among those are cohesion metrics for knowledge-based [47], aspect-oriented systems [75], and dynamic cohesion metrics for distributed applications [21].

From a measuring methodology point of view, two other cohesion metrics are of interest here since they are also based on an IR approach. However, IR methods are used differently there than in our approach. Patel et al. [65] proposed a composite cohesion metric that measures the information strength of a module. This measure is based on

1. This was originally introduced [19] and subsequently extended [20] by Chidamber and Kemerer. We will refer to the first version of LCOM [19] as LCOM1 and to the extended one [20] as LCOM2.

a vector representation of the frequencies of occurrences of data types in a module. The approach measures the cohesion of individual subprograms of a system based on the relationships to each other in this vector space. Maletic and Marcus [54] defined a file-level cohesion metric based on the same type of information that we are using for our proposed metrics here. Even though these metrics were not specifically designed for the measurement of cohesion in OO software, they could be extended to measure cohesion in OO systems.

The designers and the programmers of a software system often think about a class as a set of responsibilities that approximate the concept from the problem domain implemented by the class as opposed to a set of method-attribute interactions. Information that gives clues about domain concepts is encoded in the source code as comments and identifiers. Among the existing cohesion metrics for OO software, the Logical Relatedness of Methods (LORM) [31] and the Lack of Conceptual Cohesion in Methods (LCSM) [59] are the only ones that use this type of information to measure the conceptual similarity of the methods in a class. The philosophy behind this class of metrics, into which our work falls, is that a cohesive class is a crisp implementation of a problem or solution domain concept. Hence, if the methods of a class are conceptually related to each other, the class is cohesive. The difficult problem here is how conceptual relationships can be defined and measured. LORM uses natural language processing techniques for the analysis needed to measure the conceptual similarity of methods and represents a class as a semantic network. LCSM uses the same information, indexed with LSI, and represents classes as graphs that have methods as nodes. It uses a counting mechanism similar to LCOM.

3 An Information Retrieval Approach to Class Cohesion Measurement

OO analysis and design methods decompose the problem addressed by the software system development into classes in an attempt to control complexity. High cohesion for classes and low coupling among classes are design principles aimed at reducing the system complexity. The most desirable type of cohesion for a class is model cohesion [28] such that the class implements a single semantically meaningful concept. This is the type of cohesion that we are trying to measure in our approach.

The source code of a software system contains unstructured and (semi)structured data. The structured data is destined primarily for the parsers, while the unstructured information (that is, the comments and identifiers) is destined primarily to the human reader. Our approach is based on the premise that the unstructured information embedded in the source code reflects, to a reasonable degree, the concepts of the problem and solution domains of the software, as well as the computational logic of the source code. This information captures the domain semantics of the software and adds a new layer of semantic information to the source code, in addition to the programming language semantics. Existing work on concept and feature location [60], [66], traceability link recovery between the source code and documentation [3], [58], impact analysis [2], and other such tasks showed that our premise stands and this type of information extracted from the source code is very useful.

In order to extract and analyze the unstructured information from the source code, we use LSI [25], which is an advanced IR method. Although the general approach (see Section 3.3) would work with other IR methods or with more complex natural language processing techniques, we decided to use LSI here for several reasons. First, we have extensive experience in using LSI for other software engineering problems like concept and feature location [60], [66], [68], traceability link recovery between source code and documentation [58], identification of abstract data types in legacy source code [54], and clone detection [57]. In addition to other usages in software engineering, LSI has been successfully used in cognitive psychology for the measurement of textual coherence [38], which is the principle upon which we base our approach.

The remainder of this section gives a short overview of LSI and explains how it can be used to measure textual coherence based on previous work. The extension of this concept to cohesion measurement is then discussed and the formalism behind the definition of C3 is presented, together with examples.

3.1 Overview of Latent Semantic Indexing

LSI [25], [27] is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of the natural language) reflective of their usage in large bodies of text. LSI is based on a vector space model (VSM) [71] as it generates a real-valued vector description for documents of text. Results have shown [7], [49] that LSI captures significant portions of the meaning not only of individual words but also of whole passages, such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about the contexts in which a particular word appears or does not appear provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other.

LSI was originally developed in the context of IR as a way of overcoming problems with polysemy and synonymy that occurred with VSM approaches. Some words appear in the same contexts and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix (words by context) decomposed using singular value decomposition (SVD) [71]. As a result, LSI offers a way of assessing semantic similarity between any two samples of text in an automatic unsupervised way.

LSI relies on an SVD of a matrix (word $\times$ context) derived from a corpus of natural text that pertains to knowledge in the particular domain of interest. According to the mathematical formulation of LSI, the term combinations that occur less frequently in the given document collection tend to be precluded from the LSI subspace. LSI does "noise reduction," as less frequently co-occurring terms are less mutually related and, therefore, less sensible.

Similarly, the most frequent terms are also eliminated from the analysis. The formalism behind SVD is rather complex and too lengthy to be presented here. The interested reader may refer to the work of Salton and McGill [71] for details.

Once the documents are represented in the LSI subspace, the user can compute similarity measures between documents by the cosine between their corresponding vectors or by their length. These measures can be used for clustering similar documents together to identify "concepts" and "topics" in the corpus. This type of usage is typical for text analysis tasks. Uses of LSI in software engineering are presented and discussed in our previous work [56].

3.2 Measuring Text Coherence with Latent Semantic Indexing

In a language such as English, there are many aspects of a discourse that contribute to coherence, including coreference, causal relationships, connectives, and signals [40]. Existing approaches in cognitive psychology and computational linguistics for automatically measuring text coherence are based on propositional modeling. Foltz et al. [38] showed that LSI can be applied as an automated method that produces coherence predictions similar to propositional modeling.

The primary method for using LSI to make coherence predictions is to compare some unit of text to an adjoining unit of text in order to determine the degree to which the two are semantically related. These units could be sentences, paragraphs, individual words, or even whole books. This analysis can then be performed for all pairs of adjoining text units in order to characterize the overall coherence of the text. Coherence predictions have typically been performed at a propositional level in which a set of propositions all contained within the working memory are compared or connected to each other [46]. For an LSI-based coherence analysis, using sentences as the basic unit of text appears to be an appropriate corresponding level that can be easily parsed by automated methods. Sentences serve as a good level in that they represent a small set of textual information (for example, typically three to seven propositions) and thus would be approximately consistent with the amount of information that is held in short-term memory.

To measure the coherence of a text, LSI is used to compute the similarities between consecutive sentences in the text. High similarity between two consecutive sentences indicates that the two sentences are related, while low similarity indicates a break in the topic. A well-written article or book may provide coherence, even at these break points; thus, topic changes are not always marked by the lack of coherence. For example, an author may deliberately make a series of disconnected points, such as in a summary, which may not be a break in the discourse structure. The idea is that if the similarity between adjacent sentences is kept high, the reader can follow the logic and understand the text easier. As the similarity measure, as defined by LSI, is not transitive, it is possible to have nonadjacent sentences with very low similarity measure yet maintain high coherence. The overall coherence of a text is measured as the average of all similarity measures between consecutive sentences.

3.3 From Textual Coherence to Software Cohesion

We adapt the LSI-based coherence measurement mechanism to measure cohesion in OO software. One issue is the definition of documents in the corpus. For a natural language, sentences, paragraphs, and even sections are used as units of text to be indexed (that is, documents). Based on our previous experience [59], [60], [67], we consider methods as elements of the source code that can be units for indexing. Thus, the implementation of each method is converted to a document in the corpus to be indexed by LSI.

Another issue of interest lies in the extraction of relevant information from the source code. We extract all identifiers and comments from the source code. As mentioned before, we assume that developers used meaningful naming and commenting rules. One can argue that this information does not fully describe a piece of software. Although this is true, significant information about the source code is embedded in this data, as our previous work suggests [55]. More than that, analogous approaches are used in other fields such as image retrieval. For example, when searching for images on the Web with Google or other search engines, one really searches in the text surrounding these images in the Web pages [45].

Finally, Foltz's method for coherence measurement is based on measuring the similarity between adjacent elements of text (that is, sentences). The OO source code does not follow the same discourse rules as in a natural language; thus, the concept of adjacent elements of text (that is, methods) is not present here. To overcome this issue, we compute similarities between every pair of methods in a class. There is an additional argument for this change. A coherent discourse allows for changes in topic as long as these changes are rather smooth. In software, we interpret a cohesive class as implementing one concept (or a very small group of related concepts) from the software domain. With that in mind, each method of the class will refer to some aspect related to the implemented concept. Hence, methods should be related to each other conceptually.

We developed a tool **IR**-based **C**onceptual **C**ohesion **C**lass **M**easurement (IRC^{3}M), which supports this methodology and automatically computes C3 for any class in a given software system. The following steps are necessary to compute the C3 metric (the tool is also used to measure the LCSM metric [59]):

- **Corpus creation.** The source code is preprocessed and parsed to produce a text corpus. Comments and identifiers from each method are extracted and processed. A document in the corpus is created for each method in every class.
- **Corpus indexing.** LSI is used to index the corpus and create an equivalent semantic space.
- **Computing conceptual similarities.** Conceptual similarities are computed between each pair of methods.
- **Computing C3.** Based on the conceptual similarity measures, C3 is computed for each class (definitions are presented in the next section).

IRC^{3}M is implemented as an MS Visual Studio .NET add-in and computes the C3 metric for C++ software projects in

Visual Studio based on the above methodology. Our source code parser component is based on the Visual C++ Object Extensibility Model. Using project information retrieved from Visual Studio .NET, the tool retrieves parts of the source code that are used to produce a corpus. For software projects that are developed outside the .NET environment, that is, Mozilla from our case study, we use external parsers (for example, Columbus [35] and srcML [53]) and a set of our own utilities to construct the corpus. The extracted comments and identifiers are processed in a similar fashion as in what we used in previous work [60], that is, by the elimination of stop words and splitting identifiers that follow predefined coding standards. We use the cosine between vectors in the LSI space to compute conceptual relations. A Java version of the tool is being developed as an Eclipse plug-in.

3.4 The Conceptual Cohesion of Classes

In order to define and compute the C3 metric, we introduce a graph-based system representation similar to those used to compute other cohesion metrics.

We consider an OO system as a set of classes $C = \{c_1, c_2 \ldots c_n\}$. The total number of classes in the system C is $n = |C|$.

A class has a set of methods. For each class $c \in C$, $M(c) = \{m_1, \ldots, m_k\}$ is the set of methods of class c.

An OO system C is represented as a set of connected graphs $G_C = \{G_1, \ldots, G_n\}$, with G_i representing class c_i. Each class $c_i \in C$ is 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 $E_i \subset V_i x V_i$ is a set of weighted edges that connect pairs of methods from the class.

Definition 1 (Conceptual Similarity between Methods (CSM)). *For every class $c_i \in C$, all of the edges in E_i are weighted. For each edge $(m_k, m_j) \in E_i$, we define the weight of that edge $CSM(m_k, m_j)$ as the conceptual similarity between the methods m_k and m_j.*

The conceptual similarity between two methods m_k and m_j, that is, $CSM(m_k, m_j)$, is computed as the cosine between the vectors corresponding to m_k and m_j in the semantic space constructed by the IR method (in this case LSI):

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

where vm_k and vm_j are the vectors corresponding to the $m_k, m_j \in M(c_i)$ methods, T denotes the transpose, and $|vm_k|_2$ is the length of the vector.

For each class $c \in C$, we have a maximum of $N = C_n^2$ distinct edges between different nodes, where $n = |M(c)|$.

With this system representation, we define a set of measures that approximate the cohesion of a class in an OO software system by measuring the degree to which the methods in a class are related conceptually.

Defintion 2 (Average Conceptual Similarity of Methods in a class (ACSM)). *The ACSM $c \in C$ is*

$$ACSM(c) = \frac{1}{N} \times \sum_{i=1}^{N} CSM(m_i, m_j),$$

where $(m_i, m_j) \in E$, $i \neq j$, $m_i, m_j \in M(c)$, and N is the number of distinct edges in G, as defined in Definition 1.

TABLE 1
Conceptual Similarities between the Methods in the MySecMan Class

		m1	m2	m3	m4
m1	CanCreateWrapper	1	0.971	0.968	0.889
m2	CanCreateInstance		1	0.995	0.828
m3	CanGetService			1	0.827
m4	CanAccess				1

C3(MySecMan) = 0.913.

In our view, ACSM(c) defines the degree to which methods of a class belong together conceptually and, thus, it can be used as a basis for computing the C3.

Definition 3 (C3). *For a class $c \in C$, the conceptual cohesion of c, C3(c) is defined as follows:*

$$C3(c) = \begin{cases} ACSM(c), & if \quad ACSM(c) > 0, \\ else, & 0. \end{cases}$$

Based on the above definitions, $C3(c) \in [0, 1] \forall c \in C$. If a class $c \in C$ is cohesive, then C3(c) should be closer to one, meaning that all methods in the class are strongly related conceptually with each other (that is, the CSM for each pair of methods is close to one). In this case, the class most likely implements a single concept or a very small group of related concepts (related in the context of the software system).

If the methods inside the class have low conceptual similarity values among each other (CSM close to or less than zero), then the methods most likely participate in the implementation of different concepts and $C3(c)$ will be close to zero.

3.5 Measuring C3 Using LSI: An Example from Mozilla

To better understand the C3 metric, let us consider an example, that is, the MySecMan (my security manager) class from Mozilla 1.6 with six methods: SetMode, MySecMan, CanCreateWrapper, CanCreateInstance, CanGetService, and CanAccess. This class is one of the classes implementing the XPCSecurityManager interface, which is responsible for the interaction between Mozilla and its embedded JavaScript engine SpiderMonday. In particular, this class is responsible for enforcing the security mechanisms that all downloaded JavaScript programs are subject to. MySecMan keeps the information about the script that currently executes within SpiderMonkey and, whenever a script tries accessing information outside the engine, MySecMan grants or denies the access based on the sandbox policy, the same-origin policy, or the signed-script policy.

For simplicity and easier understanding, in computing the C3 metric, we exclude in this example two methods—the accessor SetMode and the constructor MySecMan—as these methods contain little semantic information. The conceptual similarities between the methods in the class are shown in Table 1. For the computation of ACSM, we consider all pairs of different methods; thus, $ACSM(c) = 0.913$. Since the value is positive, $C3(c) = ACSM(c) = 0.913$. The C3 value

TABLE 2
Partial Co-Occurrence Matrix for the MySecMan Class

	Wrapper	Context	Pending	Exception	Error	Failure	Security	Policy
CanCreateWrapper	1	4	1	2	1	1	1	1
CanCreateInstance	0	4	1	2	1	1	1	0
CanGetService	0	4	1	2	1	1	1	0
CanAccess	0	10	1	2	1	1	1	2

indicates a very high cohesion for the MySecMan class as the methods in this class are closely related to each other.

To better understand the mechanism behind LSI, Table 2 presents a partial co-occurrence matrix, with some important terms from the source code, relevant to this class. This type of matrix, constructed for the entire software, is subjected to SVD during indexing. The term frequencies are often adjusted to more complex measures [25], [27]: We use in our implementation the term frequency-inverse document frequency (*tf-idf*) measure [71]. One can see that these four methods share several terms such as, *context*, *pending*, *exception*, *error*, *failure*, and *security*.

The MySecMan class implements a well-defined concept from the domain of the Mozilla browser: It manages access policies for JavaScript programs within the JavaScript engine. It consists of the set of methods that are highly related conceptually as they have no other responsibilities; hence the high cohesion value. Other structural cohesion metrics also show that MySecMan is a highly cohesive class (for example, $\mathrm{LCOM1} = 20$, $\mathrm{LCOM2} = 4$, $\mathrm{LCOM3} = 3$, $\mathrm{LCOM4} = 2$, $\mathrm{LCOM5} = 0.71$, $\mathrm{ICH} = 2$, $\mathrm{TCC} = 0.5$, $\mathrm{LCC} = 0.78$, and $\mathrm{Coh} = 0.37$).

4 Assessment of the New Cohesion Measure

Newly proposed metrics require empirical evaluations [11], [12]. We present the results of two case studies aimed at comparing and combining C3 with a set of existing cohesion measures. Sections 4.1 and 4.2 describe the objectives and the design of the case studies. In the subsequent sections, quantitative results are presented and explained for each case study separately.

4.1 Objectives and Methodology

In order to evaluate our measure, we conducted two case studies. The goal of the first case study was to determine whether the C3 measure captures *additional dimensions* of cohesion measurement when compared to existing structural cohesion measures. Our hypothesis is that, given the nature of the information and counting mechanism employed by C3, it should capture different aspects of class cohesion than existing structural measures.

Existing research showed that cohesion measures can be used as good indicators for the fault proneness of classes in OO systems [30], [39], [69]. In the second case study, C3 is compared with existing metrics and combinations of C3 with existing cohesion metrics are also compared with combinations of structural metrics (with each other) to assess whether they provide *better results in predicting faults in classes* or not. Our assumption is that combining C3 with other structural cohesion metrics should be a more complete indicator of cohesion (given that they capture different aspects of it); hence, it is a better indicator of fault proneness than combinations of structural metrics alone.

In summary, the case studies address the following research questions:

- Does C3 capture aspects of class cohesion that are not captured by other structural cohesion metrics?
- Does the combination of structural cohesion metrics with C3 provide better results in predicting faults in classes than the combinations of structural metrics?

4.2 Design of the Case Studies

We followed recommendations in state-of-the-art work on case studies [37], [74] to design our two studies. We used several open source systems of different sizes.

4.2.1 Software Systems and Metrics

We chose three open source software systems from different domains, developed mostly in C++: TortoiseCVS v.1.8.21, WinMerge v.2.0.2, and Mozilla v.1.6. TortoiseCVS is an extension of the Microsoft Windows Explorer that makes the use of concurrent versioning system (CVS) convenient and easy. WinMerge is a tool for visual differencing and merging for both files and directories. Mozilla is an open source Web browser ported on almost all known software and hardware platforms. It is large enough to represent a real-world software system. The source codes for Tortoise CVS and WinMerge were downloaded from http://sourceforge.net, while the source code for Mozilla is obtained from www.mozilla.org.

We selected the following structural cohesion metrics to compare with C3: LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Coh, ICH, TCC, and LCC. Our choice of metrics is not random since these structural metrics were extensively studied and compared to each other and to other metrics in previously published studies [5], [11], [13], [15], [34], [73]. The guiding criterion that we used in choosing the metrics for our case study is the availability of results reported for these metrics in the literature in order to facilitate comparison and evaluation with our results. For the definitions, explanations, and further references on these measures, refer to Section 2. We computed all of these metrics for 2,151 classes from the three open source systems.

4.2.2 Settings of the Case Studies

All of the structural metrics are collected using Columbus [35] and the conceptual cohesion metrics are computed with our tool $\mathrm{IRC^3M}$. $\mathrm{IRC^3M}$ can be used with several settings for the underlying LSI-based analysis. In these case studies, we used the following ones.

TABLE 3
Rotated Components: Loading of the PCs

	PC_1	PC_2	PC_3	PC_4	PC_5	PC_6
Proportion	29.6	20.91	10.12	10.04	17.0	8.56
Cumulative	29.6	50.51	60.63	70.67	87.67	96.24
C3	-0.061	-0.037	-0.017	**0.996**	-0.043	0.008
LCOM1	**0.922**	-0.001	0.052	-0.032	0.317	-0.012
LCOM2	**0.914**	-0.018	0.044	-0.029	0.331	0.004
LCOM3	**0.609**	-0.129	0.052	-0.048	**0.736**	-0.138
LCOM4	0.206	-0.196	-0.001	-0.036	**0.937**	-0.102
LCOM5	0.084	0.032	**0.995**	-0.017	0.018	-0.040
ICH	**0.914**	0.056	0.066	-0.057	-0.065	-0.144
TCC	-0.023	**0.933**	-0.033	-0.002	-0.116	0.283
LCC	0.045	**0.966**	0.079	-0.050	-0.136	0.095
Coh	-0.118	0.476	-0.061	0.012	-0.176	**0.846**

For constructing the corpora, we extracted all types of methods from classes in the source code, including constructors, destructors, and accessors. Comments and identifiers were extracted from each method. The resulting text was processed as follows: Some tokens are eliminated (for example, operators, special symbols, some numbers, keywords of the programming language, and standard library function names). The identifier names in the source code were split into parts based on known coding standards. For example, all of the following identifiers were broken into the words "split" and "identifiers": "split_identifiers," "Split_identifiers," "SplitIdentifiers," etc. The original form of each identifier is preserved in the documents. Since we do not consider n-grams, the order of the words is not of importance. It is essential to note that LSI in this process does not use a predefined vocabulary or a predefined grammar; therefore, no morphological analysis or transformations are required. Some researchers use word stemming; however, this is an optional step.

Based on our previous experience with LSI on similar size corpora [58], [66], we used a 300 factor reduction. We defined a corpus with 637 documents and 1,915 terms for Tortoise CVS, one with 522 documents and 1,738 terms for WinMerge, and one with 48,823 documents and 64,979 terms for Mozilla (which we used in both case studies).

4.3 First Case Study: Principal Component Analysis of the Metric Data

Briand et al. [13] proposed a methodology for analyzing software engineering data in order to make an experiment repeatable and the results comparable. The methodology consists of three steps: collecting the data, identifying outliers, and performing the Principal Component Analysis (PCA).

In order to understand the underlying orthogonal dimensions captured by the cohesion measures, we performed PCA on all the metrics measured for all three software systems. PCA is a useful technique that has been used in a number of case studies to show that new measures capture new dimensions in cohesion measurement [15]. As the results of our analysis can be impacted by the outliers, they were removed [13], [15]. To identify outliers in the data, we utilized the $T^2 \max$ procedure based on the Mahalanobis distance [43].

After outliers were eliminated, we performed PCA, which was used in our case to identify groups of variables (that is, metrics), which likely measure the same underlying dimension (that is, the mechanism that defines cohesion) of the object to be measured (that is, the cohesion of a class). In order to identify these variables and interpret the principal components (PCs), we consider the rotated components, which is a technique where PCs are subjected to an orthogonal rotation. Thus, the resulting rotated components show clearer patterns of loading for the variables. In order to perform this rotation, we used the rotation technique known as "varimax" [44].

4.3.1 *Results*

PCA revealed six PCs that describe 96 percent of the variance in our data set. The loadings of every PC are presented in Table 3, where we marked important coefficients for each PC in boldface. In addition, for every PC, we provide the proportion for that PC in terms of the variance of the data set, which is explained by that PC and also the cumulative variance. The overall purpose of PCA is to identify factors that explain as much of the variation with as few factors as possible.

We interpret the loadings determined for every PC as follows:

PC_1 (29.6 percent): LCOM1, LCOM2, LCOM3, and ICH. These metrics count the number of pairs of methods that share instance variables. Another commonality among LCOM1-LCOM3 is that these measures are not normalized and they do not have upper bounds (for example, the nsHTMLEditor class from Mozilla has a value of 55,907 for LCOM2).

PC_2 (20.91 percent): TCC and LCC. These are among the measures that are computed as the ratio of method pairs with shared instance variables, also considering indirect sharing of instance variables by method invocations. Noticeably, the measures are also normalized.

PC_3 (10.12 percent): LCOM5. This is a normalized cohesion measure with upper and lower bounds. However, it is also an inverse cohesion measure, which ranges between 0 (maximum cohesion) and 2 (minimum cohesion). The metric is dependent on instance variable usage, counting the number of interactions between instance variables and methods.

PC_4 (10.04 percent): C3. This is a conceptual cohesion metric that measures the cohesion of a class in the context of the complete software system based on the usage of the terms shared between pairs of methods in the class, assuming that there is an underlying or latent structure in word usage for the software system for which a document set (that is, corpus) is constructed.

PC_5 (17.0 percent): LCOM3 and LCOM4. These metrics count common attribute usage within a class. LCOM4 additionally accounts for method invocations that do not seriously affect the distribution of the measure.

PC_6 (8.56 percent): Coh. This is a normalized measure which counts individual references to attributes by methods.

The PCA results show that C3 defines a dimension of its own: C3 is the only major factor in PC4. These results statistically support our hypothesis that the C3 cohesion measure captures different aspects of what is considered to be a cohesion measurement of the class, as defined by all the metrics measured in the case study.

The PCA results reinforce the work previously done by other researchers. Chae et al. [15], Briand et al. [10], and Etzkorn et al. [34] also used PCA over different data sets by using similar collections of structural cohesion metrics to those presented in this work. The results of this study are closer to those of Briand et al. [10] and Etzkorn et al. [34] as, in each case, the first two PCs are the same. All studies have LCOM1-LCOM3 measures in PC_1 and TCC-LCC in PC_2.

4.4 Second Case Study: Predicting Faults in Classes

The first case study showed that C3 captures different aspects of cohesion from the structural metrics against which we analyzed it. Given our interpretation of cohesion, we believe that a combination of structural and conceptual metrics is a more complete cohesion indicator than any combination of structural metrics since the combination of structural metrics still captures only the structural properties of cohesion, while the combination of structural and conceptual metrics might capture orthogonal yet complementary properties of class cohesion. In other words, structural cohesion indicates whether a class is built cohesively and conceptual cohesion indicates whether a class is written coherently (as a function of the identifier names and comments). One use of cohesion metrics in software engineering is to predict faults in classes [30], [39], [69]. The focus here is to analyze the extent to which each of the cohesion measures used in the case study can be used to predict faults and to see whether the combination of any structural cohesion measure with C3 outperforms the combinations of structural cohesion measures for identifying fault-prone classes.

This case study is performed in a similar fashion as in [39]. We used Bugzilla (http://bugzilla.mozilla.org/), collected the bugs between two versions of Mozilla (that is, 1.6 and 1.7), and correlated each bug with specific classes. Details on how we mined the bugs can be found in our previous work [39].

4.4.1 Analyses

We employed regression analysis methods to discover the possible relationships between values of collected metrics and the fault proneness of those classes. These methods have been widely used to study the relationships between the metrics and fault or change proneness of classes [4], [6], [9], [13], [39], [63], [72]. In order to analyze our data, we chose univariate and multivariate logistic regression analysis methods, which predict if a class is faulty or not.

In the logistic regression, the unknown variable, called the *dependent variable*, can take only two different values. Therefore, we preliminarily classified all classes into two categories: classes that contain at least one bug and those without bugs. The known variables, called the explanatory variables, are the cohesion metrics in our case. Given that cohesion metrics change over different ranges, they were standardized, that is, each metric has a zero mean and unit variance:

$$\pi(X_1, X_2) = \frac{e^{C_0 + C_1 \times X_{i1} + C_2 \times X_{i2}}}{1 + e^{C_0 + C_1 \times X_{i1} + C_2 \times X_{i2}}}.$$

The multivariate logistic regression model is based on the relationship equation, where X_is are the explanatory variables (in our study, we considered only pairs of metrics and, therefore, only two explanatory variables) and π is the probability that a fault was found in a class during the validation procedure. The coefficients C_1 and C_2 are the estimated regression coefficients. The larger the absolute value of the coefficient, the stronger the impact (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability of a fault being detected in a class. Univariate logistic regression is a special case of multivariate regression where only one exploratory variable X_1 is used.

Since logistic regression is a commonly used statistical method, it will not be explained here in detail. For more details on regression analyses and their applications on using metrics to detect fault prone classes, the reader is referred to our previous work [6], [13], [39], [72].

In the logistic regression analysis, we used univariate and multivariate regressions. In our case, the univariate regression analysis is used to analyze the effect of each metric separately, whereas multivariate regression is used to analyze the effect of the combination of metrics on the final results to see whether combinations of C3 with structural cohesion metrics can improve detecting fault-prone classes as compared to combinations of only structural metrics alone. All analyses in this case study are applied with the same settings as those described by Gyimóthy et al. [39].

For the logistic multivariate analysis, we build models for predicting faults in classes based on all possible combinations of pairs of cohesion metrics used in this case study (that is, 45 different pairs of metrics comprising 10 unique cohesion metrics). We study all of the resulting models based on pairs of cohesion metrics to obtain more insights on whether the models where one of the exploratory variables is C3 are superior or not to those models where the two exploratory variables are structural cohesion measures. The parameters (constant C_0 and

TABLE 4
Results of the Univariate Logistic Regression (Sorted by the $\mathbf{R}^2$ Values)

Metrics	Precision	Prec. Rank	Correct-ness	Corr. Rank	Complete-ness	Compl. Rank	R^2 Values	C_0	C_1
LCOM1	61.90	4	74.39	2	60.95	5	0.109	-0.41	0.0012
LCOM3	62.59	1	70.55	4	64.15	4	0.107	-0.71	0.061
LCOM2	62.05	3	75.93	1	59.16	7	0.106	-0.38	0.0013
LCOM4	59.75	7	66.36	5	54.85	8	0.079	-0.60	0.0725
C3	***62.05***	***2***	***61.35***	***6***	***73.13***	***3***	***0.073***	***2.22***	***-4.11***
ICH	60.92	6	73.52	3	53.80	9	0.069	-0.30	0.008
Coh	61.21	5	59.33	7	80.18	1	0.032	0.35	-1.96
LCOM5	56.56	8	54.48	8	77.83	2	0.006	-0.38	0.481
TCC	51.81	9	50.60	9	59.61	6	0.01	0.14	-0.799
LCC	50.73	10	49.47	10	42.64	10	0.002	0.04	-0.292

TABLE 5
Results of the Multivariate Logistic Regression for the Top 10 Pairs of Cohesion Metrics with the Largest $\mathbf{R}^2$ Values (Sorted by $\mathbf{R}^2$ Values)

Model	Precision	Prec. Rank	Correct-ness	Corr. Rank	Complete-ness	Compl. Rank	R^2 Values	C_0	C_1	C_2
C3+LCOM3	66.20	1	68.47	26	76.26	6	0.160	1.384	-3.783	0.060
C3+LCOM1	65.23	2	68.23	27	74.27	10	0.154	1.536	-3.471	0.001
C3+LCOM2	64.88	5	67.54	29	73.49	11	0.151	1.572	-3.486	0.001
C3+LCOM4	64.98	4	66.20	30	75.61	7	0.141	1.594	-4.054	0.078
C3+ICH	63.71	6	64.74	34	74.60	9	0.119	1,710	-3.597	0.006
LCOM4+ICH	63.32	9	72.87	16	65.39	15	0.119	-0.717	0.0581	0.006
LCOM3+ICH	63.46	7	72.61	17	65.32	16	0.118	-0.703	0.0484	0.003
LCOM1+LCOM3	63.27	10	74.16	12	64.12	21	0.116	-0.611	0.0006	0.034
LCOM1+LCOM4	61.90	28	73.05	15	61.41	32	0.114	-0.553	0.0008	0.030
LCOM1+Coh	62.34	21	72.44	18	64.28	18	0.113	-0.208	0.0011	-0.816

TABLE 6
Results of the Multivariate Logistic Regression for the Remaining Pairs of Metrics Which Contain C3 (Sorted by $\mathbf{R}^2$ Values)

Model	Precision	Prec. Rank	Correct-ness	Corr. Rank	Complete-ness	Compl. Rank	R^2 Values	R^2 Rank	C_0	C_1	C_2
C3+Coh	65.03	3	64.22	35	79.43	4	0.105	28	2.709	-4.220	-2.086
C3+TCC	63.41	8	62.72	38	75.38	8	0.085	29	2.507	-4.225	-0.956
C3+LCOM5	63.22	11	62.47	39	76.59	5	0.079	34	1.878	-4.129	0.503
C3+LCC	63.17	12	62.80	37	72.64	12	0.078	37	2.440	-4.230	-0.495

coefficients C_1 and C_2) of all instantiated models are provided in Tables 4, 5, and 6, respectively.

4.4.2 Results

First, we performed the univariate logistic regression (see the results in Table 4). The R^2 coefficient is defined as the proportion of the total variation in the dependant variable y (the fault proneness of a class) that is explained by the regression model. The bigger the value of R^2, the larger the portion of the total variance in y that is explained by the regression model and the better the dependent variable y is explained by the explanatory variables.

In order to evaluate logistic regression models based on the studied metrics and their combinations, we utilize the following quantitative characteristics: *precision*, *correctness*, and *completeness*. We use these measures to be consistent with previously published results [6], [39]. Note that these characteristics of the results are somewhat different from the *precision* and *recall* measures used in IR.

Precision here is used to evaluate how well the model classifies faulty and nonfaulty classes. For example, C3 used as a separate explanatory variable in the univariate logistic model classified 1,267 (667 as nonfaulty + 600 as faulty) classes correctly out of 2,042 classes for Mozilla, that is, a precision of 62.05 percent (see Table 4). The results of the univariate logistic regression indicate that the model based on C3 is better than any other model except that of LCOM3.

Correctness is used to show what percentage of the faulty predicted classes is really faulty (computed as the number of classes observed and predicted as faulty divided by the total number of classes predicted as faulty). In the case of the univariate logistic regression model based on C3, the correctness is 61.35 percent since it suggested 978 classes as containing faults, but, in fact, only 600 of those have faults.

The other important characteristic used to evaluate our models is *completeness* or what percentage of the total number of faults can be captured by the model. Completeness is defined as the number of faults in classes predicted as faulty divided by the total number of faults in all classes. In our example, with the model based on C3, the total number of faults in all classes is 823 + 2,240, whereas in classes that are predicted as faulty, 2,240 faults are present. Thus, the completeness value of the model based on C3 is $2,240/(823 + 2,240) = 73.13\%$. Notice that logistic regression predicts if a class has at least one bug or none.

The results of the univariate logistic regression (see Table 4) allow us to draw the conclusions that, if we use each of the 10 metrics as a separate indicator of fault proneness, LCC is the least significant and C3 has the second largest *precision*, third largest *completeness*, fifth largest R^2 value, and sixth largest *correctness*. These results are not surprising as C3 captures only certain aspects of cohesion, whereas faults may be caused by other issues affecting cohesion which are not captured by C3 alone. Nonetheless, C3 ranks better than many of the cohesion metrics that we analyzed.

Our assumption is that C3 complements existing structural metrics, so, in order to investigate whether combining C3 with structural cohesion measures can improve the detection of fault-prone classes, we applied multivariate logistic regression analysis (see Table 5). We built 45 models based on all combinations of the pairs of cohesion metrics. Table 5 presents the top 10 models based on the largest $\mathbf{R^2}$ values. Based on the results, C3 appears in the first five combined models. In order to compare the $\mathbf{R^2}$ values of these combinations, we performed the Wilcoxon signed-rank test. The results revealed that, out of those five pairs, C3 appears in three statistically significant combinations as the $\mathbf{R^2}$ values for C3+LCOM3 and C3+LCOM4 and the $\mathbf{R^2}$ values for C3+LCOM1 and C3+LCOM2, respectively, are not different.

In addition, all of the models that contain C3 (see Tables 5 and 6) are among the top 12 models in terms of precision and completeness values. This result supports the idea that C3 is indeed *complementary* to the structural metrics, at least from the point of view of fault prediction.

In addition, note that the three models with the best precision are C3+LCOM3, C3+LCOM1, and C3+Coh. Any of these models has better precision values than the best model based on a single metric in Table 4. Similarly, the models with the best values for correctness and completeness outperform the relevant models based on a single metric.

Overall, the results indicate that C3 is a useful indicator of an external property of classes in OO systems, that is, the fault proneness of classes. Based on the results of the regression analyses, we can conclude that C3 is a valuable complement in a number of combinations with other structural cohesion metrics. More importantly, the results support our assumption that the combination of C3 with other cohesion metrics allows us to build superior models for detecting fault prone classes based on cohesion metrics.

4.5 Threats to Validity

Several issues affect the results of the case studies and limit our interpretations and generalizations of the results. The first case study showed that our metric captures new dimensions in cohesion measurement; however, we obtained these results by analyzing classes from only three open source applications written primarily in C++, even though one of the systems (that is, Mozilla) represents a real-life application. In order to generalize the results, a large-scale evaluation similar to the one presented by Succi et al. [73] is required which should take into account software systems from different domains written in different programming languages and of varying class sizes in those systems [29]. In the case study, we compare our measures with existing structural cohesion measures which could be computed with available tools. The results could be somewhat different if we considered semantic metrics or those based on information-theory approaches.

One issue may affect the internal validity of the second case study: Cohesion is not the only factor affecting the fault proneness of classes. To build complete models for fault prediction, other factors would have to be considered. However, this is out of the scope of this paper as the purpose of analyzing fault prediction was to see if the combination of C3 with structural metrics brings any improvements.

5 Conceptual versus Structural Cohesion

Henderson-Sellers [41] noted that: "It is after all possible to have a class with high internal, syntactic cohesion but little semantic cohesion." To gain more insight into how our metric differs from some of the structural ones, we manually analyzed classes from Mozilla and WinMerge for which the structural and conceptual metrics disagree (that is, high structural cohesion with low conceptual cohesion and vice versa). We selected classes based on high LCOM2 values (that is, indicating low cohesion) and high C3 values and vice versa. We considered a value for an LCOM2 and C3 high if it is in the top 15 percent and low if it is in the bottom 15 percent, respectively. Based on this criteria, we identified 25 classes in Mozilla that have low structural cohesion (based on LCOM2) and high conceptual cohesion (based on C3) and 61 classes in the opposite category. In WinMerge, we identified four classes that have low structural cohesion and high conceptual cohesion and nine classes in the opposite category. We provide a few examples of such classes (see Table 7), although we do not claim that all such cases in the two open source software systems that we analyzed follow these patterns.

5.1.1 *Analyzing Classes from WinMerge*

The analysis of the classes in Table 7 yields very interesting results. For example, the IVSSItem class is a wrapper class that does not have data members: It only has methods that wrap the implementation for the OLE automation on the client side. High values for LCOM2 in methods are easily explained in this case. The intersection of every pair of methods in this class is null because the class does not contain any attributes. For the IVSSItem class that has 33 methods, $\mathrm{LCOM2} = 528$. The high C3 value is also understandable as the implementation of every method contains invocations of the InvokeHelper method of the derived class COleDispatchDriver and a similar subset of identifier names for local variables. Wrappers tend to group together methods that are conceptually similar. The IVSSDatabase and IVSSItemOld classes follow the same pattern since they also implement wrappers for the

TABLE 7
Classes Analyzed from WinMerge and Mozilla

WinMerge class	C3	LCOM2	Mozilla class	C3	LCOM2
IVSSItem	0.64	528			
IVSSDatabase	0.635	136	nsXIContext	0.314	1
IVSSItemOld	0.632	465	txFormatNumberFunctionCall	0.306	1
BCMenuData	0.434	0	nsAbCardProperty	0.810	8907
CDirDoc	0.294	0	nsProfile	0.650	1768
RescanSuppress	0.392	1	nsPrintSettings	0.656	5402

COleDispatchDriver interface. In conclusion, in these situations (that is, wrappers), it seems that C3 can give more clues on cohesion than LCOM2.

From the other group of investigated classes, BCMenu Data is a class that implements a "property container" for menu items that are drawn using a style like "Office XP." It is a small class, with a set of accessor functions. LCOM2 is 0, meaning that the number of intersecting sets is more than the number of nonintersecting ones. Close examination of the class supports the fact that the class represents a single meaningful abstraction; however, values of C3 do not capture this fact due to the large number of unique identifier names used in these accessors. As mentioned above, accessor methods, just like constructors, may significantly influence the measurement of C3 and such situations warrant the use of structural metrics in support of C3.

The CDirDoc class is an example of a class with concealed cohesion, which means that the class includes some attributes and methods that might create another class. Close analysis revealed that the class handles the following activities: creating and closing of a new document, representing "right-left" panel abstraction in the "view-merge" application, keeping track of updating time, status, and content, and choosing different view modes. It has only several attributes like pointer to CDirView class and a container of CMergeDoc classes. Those attributes are referenced in most methods of the class. Thus, LCOM2 for the CDirDoc is 0. On the other hand, the value of C3 for CDirDoc is 0.294, which shows low conceptual similarity of methods inside the class. Detailed analysis shows that the class implements a set of concepts that could be refactored into separate classes, implementing each one concept only. The low LCOM2 value would indicate a difficult refactoring since it may create high coupling. However, when considering the low number of attributes of the class, this may not be a major issue. Although it is hard to generalize, in situations when a class has few attributes and many methods by comparison, a low LCOM2 value and a low C3 value may indicate lack of cohesion and a need for refactoring.

The RescanSuppress class implements an abstraction of a simple lock that prevents objects of type CMergeDocs from rescanning within its lifetime unless the clear() method is called. It is a small class with three attributes and three methods: constructor, destructor, and the clear() method. Although the class represents a crisp abstraction from a user point of view, the small value for C3 can be explained due to the small number of identifiers and their intersections within method implementations of the class. This is a situation where C3 showed its limits and structural metrics are needed.

5.1.2 Analyzing Classes from Mozilla

We applied the same strategy to analyze several more classes from Mozilla (see Table 7). In this case, the nsXIContext class aggregates dialogs like "license," "welcome," and widgets like "next" and "previous" buttons, which appear during the installation process. Therefore, the main purpose of this class is to store the user interface components during the installation process. The class has a set of methods to load, get, and release resources. It also contains a proprietary implementation of the *itoa* function, which, in fact, decreases the conceptual cohesion of the class since it has low conceptual similarities with other methods in this class (*itoa* performs a very specific operation, that is, conversion from integer to string, which does not relate conceptually to loading and releasing resource operations implemented in the class). Another class from the group of classes with low conceptual and high structural cohesion is txFormatNumberFunctionCall. This is a class that derives from an abstract class FunctionCall, which, in turn, also derives from an abstract class, namely, Expr. This is a base class of the Extensible Stylesheet Language (XSL) expressions and its "evaluate()" method is responsible for evaluating and formatting numbers in XSL transformations. After analyzing the "evaluate()" method, we concluded that it has two parts: One is the part of the process of parsing the format of the string and the other one is the actual formatting of the number. Thus, we could refactor another class (for example, txFormatParseState), which would be responsible for parsing the string, whereas txFormatNumberFunctionCall would perform the actual transformation of numbers in XSL expressions. As with the class CDirDoc from WinMerge, this is another example of a class with concealed cohesion.

On the other hand, we analyzed three classes with high conceptual and low structural cohesion. The first class in this group is nsAbCardProperty, which implements a well-defined concept, "Address Book Person Card Entry." It has more than 50 different attributes that can occur in the address book (for example, m_PhoneticFirstName, m_DisplayName, and m_DefaultEmail) and it has many accessor methods. Thus, the low structural cohesion can be easily explained because of the presence of these accessor methods, which usually reference one attribute at a time. Since all of these methods are small and share many similar terms (for example *card*, *property*, *set*, *prunichar*, *attribute*, and *name*), on average, all pairs of methods have high conceptual similarities. We concluded that this class is one implementing a cohesive concept, that is, storing and accessing the information about the user in the address book entry.

Another class, nsProfile, implements the concept of a profile in the Mozilla Web browser and it is derived from two interfaces: nsIProfileInternal and nsIProfileChangeStatus. It is rather large, consisting of over a dozen methods and attributes, implemented in over 2 KLOCs. Some of the methods with self-descriptive names are LoadDefaultProfile Dir, LoadNewProfilePrefs, MigrateProfileInternals, Update 4xProfileInfo, etc. After inspecting all of the methods, we identified that they can be classified into two groups: operations on Profile and on Registry. This classification explains the high values of LCOM since these two groups of methods reference nonoverlapping attributes. Overall, we conclude that the nsProfile class implements a single concept, even though its operations may be categorized into two related groups (the second group of Registry operations relates to Profile operations in the sense that those methods are tailored toward reading and writing profile-related keys in the system registry).

The class nsPrintSettings describes the print settings for a document: print range, colors, paper size, orientation, etc. This class aggregates a lot of attributes that describe these properties. In addition, all of these attributes have accessor methods. This class looks like a property container since it does not have any operations on these attributes, which can be broadly classified into two groups: printer attributes (mPrintBGColors, mPrintBGImages, and mPrintPreview) and paper attributes (mPaperName and mPaperSizeUnit). Low structural cohesion is explained by the number of accessor methods referencing unique attributes. However, conceptually, the class implements a single concept, that is, a property holder of print settings, which is supported by the high C3 values.

6 Limitations and Future Work

The C3 metric depends on reasonable naming conventions for identifiers and relevant comments contained in the source code. When these are missing, the only hope for measuring any aspects of cohesion rests on the structural metrics.

In addition, methods such as constructors, destructors, and accessors may artificially increase or decrease the cohesion of a class [11]. Although we did not exclude them in the results presented here, our method may be extended to exclude them from the computation of the cohesion by using approaches for identifying types of method stereotypes [26].

C3 does not take into account polymorphism and inheritance in its current form. It only considers methods of a class that are implemented or overloaded in the class. One way in which we can extend our work to address inheritance when building a corpus is to follow the approach in [48], where the source code of inherited methods is included into the documents of derived classes.

Since C3 uses textual information in its definition, we are considering the inclusion of external documentation in the corpus. This will allow us to extend the context in which words are used in the software and see if there are inconsistencies between source code and external documentation.

Given our results that show that C3 and some structural metrics complement each other at least for fault prediction, more insights into the combination of conceptual and structural metrics for solving other problems are required and planned. Specifically, we intend to address crosscutting concerns, which affect the cohesion of classes. We can achieve this by extending our current and previous work on using LSI to compute the conceptual coupling among classes [67] and on the detection of conceptual clones in the source code [57].

Structural and semantic cohesion directly impacts the understandability and readability of the source code. To measure and understand this effect, user studies are needed. We are planning user studies similar to those described in [23] to test whether C3 may reflect the perception of the programmers on class cohesion.

Finally, we plan to test this approach on several releases of a system such that the models are built on a given release and faults are predicted for the next release of the system.

7 Conclusions

Classes in object-oriented systems, written in different programming languages, contain identifiers and comments which reflect concepts from the domain of the software system. This information can be used to measure the cohesion of software. To extract this information for cohesion measurement, Latent Semantic Indexing can be used in a manner similar to measuring the coherence of natural language texts.

This paper defines the conceptual cohesion of classes, which captures new and complementary dimensions of cohesion compared to a host of existing structural metrics. Principal component analysis of measurement results on three open source software systems statistically supports this fact. In addition, the combination of structural and conceptual cohesion metrics defines better models for the prediction of faults in classes than combinations of structural metrics alone. Highly cohesive classes need to have a design that ensures a strong coupling among its methods and a coherent internal description.

Acknowledgments

This work was supported in part by US National Science Foundation Grant CCF-0438970, US National Institutes of Health Grant NHGRI 1R01HG003491, the 2006 IBM Eclipse Innovation Award, and Hungarian National Grants GVOP-3.3.1.-2004-04-0024/3.0 and GVOP-3.1.1.-2004-05-0345/3.0. The authors are extremely grateful to the anonymous reviewers for their very pertinent and helpful comments and suggestions, which helped them in significantly improving the earlier version of this paper. The software tools Columbus [36] and srcML [53] were used in the production of the results presented here.

References

[1] E.B. Allen, T.M. Khoshgoftaar, and Y. Chen, "Measuring Coupling and Cohesion of Software Modules: An Information-Theory Approach," *Proc. Seventh IEEE Int'l Software Metrics Symp.*, pp. 124-134, Apr. 2001.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the Starting Impact Set of a Maintenance and Reengineering," *Proc. Fourth European Conf. Software Maintenance,* pp. 227-230, 2000.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Trans. Software Eng.,* vol. 28, no. 10, pp. 970-983, Oct. 2002.

[4] E. Arisholm, L.C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 30, no. 8, pp. 491-506, Aug. 2004.

[5] J. Bansiya and C.G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Trans. Software Eng.,* vol. 28, no. 1, pp. 4-17, Jan. 2002.

[6] V.R. Basili, L.C. Briand, and W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751-761, Oct. 1996.

[7] M.W. Berry, "Large Scale Singular Value Computations," *Int'l J. Supercomputer Applications,* vol. 6, pp. 13-49, 1992.

[8] J. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *Proc. Symp. Software Reusability,* pp. 259-262, Apr. 1995.

[9] L. Briand, W. Melo, and J. Wust, "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 706-720, July 2002.

[10] L.C. Briand, J.W. Daly, V. Porter, and J. Wüst, "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems," *Proc. Fifth IEEE Int'l Software Metrics Symp.,* pp. 43-53, Nov. 1998.

[11] L.C. Briand, J.W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.,* vol. 3, no. 1, pp. 65-117, 1998.

[12] L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurements," *IEEE Trans. Software Eng.,* vol. 22, no. 1, pp. 68-85, Jan. 1996.

[13] L.C. Briand, J. Wüst, J.W. Daly, and V.D. Porter, "Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Systems," *J. System and Software,* vol. 51, no. 3, pp. 245-273, May 2000.

[14] F. Brito e Abreu and M. Goulao, "Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded," *Proc. Fifth European Conf. Software Maintenance and Reeng.,* pp. 47-57, 2001.

[15] H.S. Chae, Y.R. Kwon, and D.H. Bae, "A Cohesion Measure for Object-Oriented Classes," *Software: Practice and Experience,* vol. 30, pp. 1405-1431, 2000.

[16] H.S. Chae, Y.R. Kwon, and D.H. Bae, "Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables," *IEEE Trans. Software Eng.,* vol. 30, no. 11, pp. 826-832, Nov. 2004.

[17] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang, "A Novel Approach to Measuring Class Cohesion Based on Dependence Analysis," *Proc. 18th IEEE Int'l Conf. Software Maintenance,* pp. 377-384, 2002.

[18] S. Chidamber, D. Darcy, and C. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 629-639, Aug. 1998.

[19] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," *Proc. Sixth ACM Conf. Object-Oriented Programming, Systems, Languages and Applications,* pp. 197-211, 1991.

[20] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.

[21] E.S. Cho, C.J. Kim, D.D. Kim, and S.Y. Rhew, "Static and Dynamic Metrics for Effective Object Clustering," *Proc. Fifth Asia-Pacific Software Eng. Conf.,* pp. 78-85, 1998.

[22] S. Counsell, S. Swift, and J. Crampton, "The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design," *ACM Trans. Software Eng. and Methodology,* vol. 15, no. 2, pp. 123-149, 2006.

[23] S. Counsell, S. Swift, and A. Tucker, "Object-Oriented Cohesion as a Surrogate of Software Comprehension: An Empirical Study," *Proc. Fifth IEEE Int'l Workshop Source Code Analysis and Manipulation,* pp. 161-172, 2005.

[24] D. Darcy and C. Kemerer, "OO Metrics in Practice," *IEEE Software,* vol. 22, no. 6, pp. 17-19, Nov./Dec. 2005.

[25] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *J. Am. Soc. Information Science,* vol. 41, pp. 391-407, 1990.

[26] N. Dragan, M.L. Collard, and J.I. Maletic, "Reverse Engineering Method Stereotypes," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 24-34, Sept. 2006.

[27] S.T. Dumais, "Improving the Retrieval of Information from External Sources," *Behavior Research Methods, Instruments, and Computers,* vol. 23, no. 2, pp. 229-236, 1991.

[28] J. Eder, G. Kappel, and M. Schreft, "Coupling and Cohesion in Object-Oriented Systems," technical report, Univ. of Klagenfurt, 1994.

[29] K. El-Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.,* vol. 27, no. 7, pp. 630-650, July 2001.

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

[31] L. Etzkorn and H. Delugach, "Towards a Semantic Metrics Suite for Object-Oriented Design," *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems,* pp. 71-80, July 2000.

[32] L.H. Etzkorn and C.G. Davis, "Automatically Identifying Reusable OO Legacy Code," *Computer,* vol. 30, no. 10, pp. 66-72, Oct. 1997.

[33] L.H. Etzkorn, S. Gholston, and W.E. Hughes, "A Semantic Entropy Metric," *J. Software Maintenance: Research and Practice,* vol. 14, no. 5, pp. 293-310, July/Aug. 2002.

[34] L.H. Etzkorn, S.E. Gholston, J.L. Fortune, C.E. Stein, D. Utley, P.A. Farrington, and G.W. Cox, "A Comparison of Cohesion Metrics for Object-Oriented Systems," *Information and Software Technology,* vol. 46, no. 10, pp. 677-687, Aug. 2004.

[35] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus: Reverse Engineering Tool and Schema for C++," *Proc. 18th IEEE Int'l Conf. Software Maintenance,* pp. 172-181, Oct. 2002.

[36] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting Facts from Open Source Software," *Proc. 20th IEEE Int'l Conf. Software Maintenance,* pp. 60-69, Sept. 2004.

[37] B. Flyvbjerg, "Five Misunderstandings about Case Study Research," *Qualitative Inquiry,* vol. 12, no. 2, pp. 219-245, 2006.

[38] P.W. Foltz, W. Kintsch, and T.K. Landauer, "The Measurement of Textual Coherence with Latent Semantic Analysis," *Discourse Processes,* vol. 25, no. 2, pp. 285-307, 1998.

[39] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 897-910, Oct. 2005.

[40] M.A.K. Halliday and R. Hasan, *Cohesion in English.* Longman, 1976.

[41] B. Henderson-Sellers, *Software Metrics.* Prentice Hall, 1996.

[42] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *Proc. Third Int'l Symp. Applied Corporate Computing,* Oct. 1995.

[43] J.D. Jobson, *Applied Multivariable Data Analysis.* Springer-Varlag, 1992.

[44] I.T. Jolliffe, *Principal Component Analysis.* Springer Verlag, 1986.

[45] M.L. Kherfi, D. Ziou, and A. Bernardi, "Image Retrieval from the World Wide Web: Issues, Techniques, and Systems," *ACM Computing Surveys,* vol. 36, no. 1, pp. 35-67, 2004.

[46] W. Kintsch, *Comprehension: A Paradigm for Cognition.* Cambridge Univ. Press, 1998.

[47] S. Kramer and H. Kaindl, "Coupling and Cohesion Metrics for Knowledge-Based Systems Using Frames and Rules," *ACM Trans. Software Eng. and Methodology,* vol. 13, no. 3, pp. 332-358, July 2004.

[48] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," *Proc. 12th IEEE Working Conf. Reverse Eng.,* pp. 133-142, Nov. 2005.

[49] T.K. Landauer and S.T. Dumais, "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge," *Psychological Rev.,* vol. 104, no. 2, pp. 211-240, 1997.

[50] J.K. Lee, S.J. Jung, S.D. Kim, W.H. Jang, and D.H. Ham, "Component Identification Method with Coupling and Cohesion," *Proc. Eighth Asia-Pacific Software Eng. Conf.,* pp. 79-86, Dec. 2001.

[51] Y.S. Lee, B.S. Liang, S.F. Wu, and F.J. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *Proc. Int'l Conf. Software Quality*, 1995.
[52] R.F. Lorch and E.J. O'Brien, *Sources of Coherence in Reading.* Erlbaum, 1995.
[53] J.I. Maletic, M.L. Collard, and A. Marcus, "Source Code Files as Structured Documents," *Proc. 10th IEEE Int'l Workshop Program Comprehension*, pp. 289-292, June 2002.
[54] J.I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," *Proc. 23rd IEEE Int'l Conf. Software Eng.*, pp. 103-112, May 2001.
[55] A. Marcus, "Semantic Driven Program Analysis," PhD dissertation, Kent State Univ., 2003.
[56] A. Marcus, A. De Lucia, J. Huffman Hayes, and D. Poshyvanyk, "Working Session: Information-Retrieval-Based Approaches in Software Evolution," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 197-199, Sept. 2006.
[57] A. Marcus and J.I. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.*, pp. 107-114, Nov. 2001.
[58] A. Marcus, J.I. Maletic, and A. Sergeyev, "Recovery of Traceability Links between Software Documentation and Source Code," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 15, no. 4, pp. 811-836, Oct. 2005.
[59] A. Marcus and D. Poshyvanyk, "The Conceptual Cohesion of Classes," *Proc. 21st IEEE Int'l Conf. Software Maintenance*, pp. 133-142, Sept. 2005.
[60] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. 11th IEEE Working Conf. Reverse Eng.*, pp. 214-223, Nov. 2004.
[61] T.M. Meyers and D. Binkley, "Slice-Based Cohesion Metrics and Software Intervention," *Proc. 11th IEEE Working Conf. Reverse Eng.*, pp. 256-265, Nov. 2004.
[62] C. Montes de Oca and D.L. Carver, "Identification of Data Cohesive Subsystems Using Data Mining Techniques," *Proc. 14th IEEE Int'l Conf. Software Maintenance,* pp. 16-23, Nov. 1998.
[63] H. Olague, L. Etzkorn, S. Gholston, and S. Quattlebaum, "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 Trans. Software Eng.*, vol. 33, no. 6, pp. 402-419, June 2007.
[64] L.M. Ott and J.J. Thuss, "Slice Based Metrics for Estimating Cohesion," *Proc. First IEEE Int'l Software Metrics Symp.*, pp. 71-81, 1993.
[65] S. Patel, W. Chu, and R. Baxter, "A Measure for Composite Module Cohesion," *Proc. 14th IEEE Int'l Conf. Software Eng.*, pp. 38-48, May 1992.
[66] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 420-432, June 2007.
[67] D. Poshyvanyk and A. Marcus, "The Conceptual Coupling Metrics for Object-Oriented Systems," *Proc. 22nd IEEE Int'l Conf. Software Maintenance*, pp. 469-478, Sept. 2006.
[68] D. Poshyvanyk and D. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," *Proc. 15th IEEE Int'l Conf. Program Comprehension,* pp. 37-48, June 2007.
[69] T.-S. Quah and M.M.T. Thwin, "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics," *Proc. 19th IEEE Int'l Conf. Software Maintenance,* pp. 116-125, Sept. 2003.
[70] M. Sahami and T.D. Heilman, "Web Mining with Search Engines: A Web-Based Kernel Function for Measuring the Similarity of Short Text Snippets," *Proc. 15th Int'l World Wide Web Conf.,* pp. 377-386, 2006.
[71] G. Salton and M. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.
[72] R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.*, vol. 29, no. 4, pp. 297-310, Apr. 2003.
[73] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, "An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite," *Empirical Software Eng.*, vol. 10, no. 1, pp. 81-104, Jan. 2005.
[74] R.K. Yin, *Applications of Case Study Research,* second ed. Sage Publications, 2003.
[75] J. Zhao and B. Xu, "Measuring Aspect Cohesion," *Proc. Seventh Int'l Conf. Fundamental Approaches to Software Eng.*, pp. 54-68, 2004.
[76] Y. Zhou, J. Lu, H. Lu, and B. Xu, "A Comparative Study of Graph Theory-Based Class Cohesion Measures," *ACM SIGSOFT Software Eng. Notes*, vol. 29, no. 2, p. 13, Mar. 2004.
[77] Y. Zhou, L. Wen, J. Wang, Y. Chen, H. Lu, and B. Xu, "DRC: A Dependence-Relationships-Based Cohesion Measure for Classes," *Proc. 10th Asia-Pacific Software Eng. Conf.*, pp. 215-223, 2003.
[78] Y. Zhou, B. Xu, J. Zhao, and H. Yang, "ICBMC: An Improved Cohesion Measure for Classes," *Proc. 18th IEEE Int'l Conf. Software Maintenance*, pp. 44-53, Oct. 2002.

Andrian Marcus received the PhD degree in computer science from Kent State University in 2003 and prior degrees from the University of Memphis and the "Babeş-Bolyai" University of Cluj, Romania. He is currently an assistant professor in the Department of Computer Science at Wayne State University, Detroit. Since 2005, he has been serving on the steering committee of the IEEE International Conference on Software Maintenance (ICSM). His research interests include software evolution, program understanding, and software visualization, in particular using information retrieval techniques to support software engineering tasks. He is the recipient of a Fulbright Junior Research Fellowship in 1997. He is a member of the IEEE Computer Society.

Denys Poshyvanyk received the MS degree in computer science from the National University of Kyiv-Mohyla Academy, Ukraine, in 2003 and the MA degree in computer science from Wayne State University, Detroit, in 2006. He is currently working toward the PhD degree in the Department of Computer Science at Wayne State University. He is a Microsoft Student Partner for Wayne State University. His research interests include software maintenance and evolution, program comprehension, source code analysis, reverse engineering, and software metrics. He is student member of the IEEE, the IEEE Computer Society, and the ACM.

Rudolf Ferenc received the PhD degree in mathematics and computer science from the University of Szeged in 2005. He was a member of the program committee of the 21st IEEE International Conference on Software Maintenance (ICSM '05), ICSM '06, and ICSM '07 and the Second International Conference on Software and Data Technologies (ICSOFT '07). He was a cochair of the Tools Track at ICSM '05. His research interests include source code analysis, modeling, measurement, quality assurance, and bug detection, as well as design pattern usage recognition and open source software development.