ISSN:0975-9646

Fault Prediction OO Systems Using the Conceptual Cohesion of Classes

Subba Rao Polamuri, S. Rama Sree, M.Rajababu

Dept of Computer Science and Engineering,
Aditya Engineering College
Surampalem, Kakinada, A.P, India

Abstract— **High cohesion is desirable property in software systems to achieve reusability and maintainability. In this project we are measures for cohesion in Object-Oriented (OO)[10] software reflect particular interpretations of cohesion and capture different aspects of it. In existing approaches the cohesion is calculate from the structural information for example method attributes and references. In conceptual cohesion of classes, i.e. in our project we are calculating the unstructured information from the source code such as comments and identifiers. Unstructured information is embedded in the source code. To retrieve the unstructured information from the source code Latent Semantic Indexing is used. 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[9]. In our project we are achieving the high cohesion and we are predicting the fault in Object –Oriented Systems. 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.**

Keywords—**Latent Semantic Indexing, image Retrieval, Cohesion, Coupling, fault Prediction.**

I. INTRODUCTION

The 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 which try capturing different aspects of cohesion or reflect a particular interpretation of cohesion.

Cohesion measures the semantic strength of relationships between components within a functional unit. Coupling[4] measures the strength of all relationships between functional units.

Proposals of measures and metrics for cohesion abound in the literature as software cohesion metrics proved to be useful in different tasks including the assessment of design quality [5][6] productivity, design, and reuse effort, prediction of software quality, fault prediction, modularization of software, and identification of reusable of components.

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 tradeoff is that such measures deal with information that can be automatically extracted from software and analysed by automated tools and ignore less structured but rich information from the software. Cohesion is usually measured on structural information extracted solely from the source code 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 metrics address the conceptual and textual aspects of cohesion.

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. 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 analyse 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.

II. Related Work

A. *Lack of Cohesion in Methods*

Cohesion is an important concept in OO programming. It indicates whether a class represents a single abstraction or multiple abstractions. The idea is that if a class represents more than one abstraction, it should be refactored into more than one class, each of which represents a single abstraction.

Despite its importance, it is difficult to establish a clear mechanism for measuring it. This is probably due to the fact that good abstractions have deep semantics and a class that is clearly cohesive when viewed from a semantic point of view may not be so when viewed from a purely symbolic point of view. As an aside, the somewhat inelegant name is due to the wish to have lower metric values representing a 'better' situation. I have selected four definitions of lack of cohesion. That of Chidamber and Kemerer, Henderson and Sellers and two proposed in this paper. Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field. When this value is negative, the metric value is set to 0.

Henderson-sellers Lack of Cohesion in Methods as follows. Let M be the of methods defined by the class, f be the set of fields defined by the class, *r(f)* be the number of methods that access fields f, where f is a member of F.

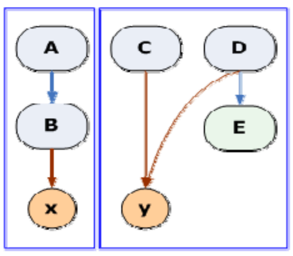
Lack of Cohesion in Methods= (<r> - |M|)/(1 - |M|)

This definition of Lack of Cohesion use Watanabe's generalization of mutual information known as Total Correlation, which determines if a group of variables exhibit redundancy or structure .Its application to the measurement of cohesion is simple. Each method in a class makes use of a subset of the fields of the class. We want to know whether the way these subsets exhibit some structure between the fields. If such a structure exists then it can be extracted into one or more other classes in order to remove or reduce the structure. We may therefore consider the use of each field by the methods as a 'random' binary variable with a certain probability of occurrence.

The rest follows naturally from the definition of Total Correlation. Although removal of structure would normally be considered a bad thing in software, the ideal cohesive scenario of 'all fields used by all methods' exhibits no structure in field usage. Pairwise Field Irrelation: Let: M be set of methods defined by the class, F be the set of fields defined by the class, M_f be the subset of M of methods that access field f, where f is a member of F. Then, the Total Field Irrelation is the mean Jaccard Distance between M_{f1} and M_{f2}, where $f_1 \neq f_2$.Note 1: I have only included methods in M if they access at least one field.

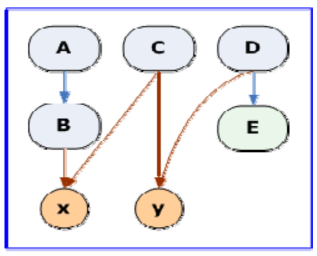
The reason for this is that methods that do not access fields are often required to be non-static for reasons of polymorphic dispatch. However, these kinds of methods skew the value of this metric in a way that is not helpful. Which methods are related? Methods a and b are related if: they both access the same class-level variable, or a calls b, or b calls a. After determining the related methods, we draw a graph linking the related methods to each other. LCOM4 equals the number of connected groups of methods.

- LCOM4=1 indicates a cohesive class, which is the "good" class.
- LCOM4>=2 indicates a problem. The class should be split into so many smaller classes.
- LCOM4=0 happens when there are no methods in a class. This is also a "bad" class.



LCOM4 = 2

The example on the left shows a class consisting of methods A through E and variables x and y. A calls B and B accesses x. Both C and D access y. D calls E, but E doesn't access any variables. This class consists of 2 unrelated components (LCOM4=2). You could split it as {A, B, x} and {C, D, E, y}.



LCOM4 = 1

In the example on the right, we made C access x to increase cohesion.

Now the class consists of a single component (LCOM4=1). It is a cohesive class.

B. *Cohesion Measures for OO Software Systems*

There are several different approaches to measure cohesion 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],semantic metrics [33], information entropy-based metrics [1], slice-based metrics], metrics based on data mining and metrics for specific types of applications like knowledge-based aspect-oriented, and distributed systems. The class of structural metrics is the most investigated category of cohesion metrics and includes lack of cohesion in methods (LCOM),1LCOM3,LCOM4, Co (connectivity) LCOM5, Coh, TCC (tight class cohesion) [8], LCC (loose class cohesion) [8], ICH (information-flow-based cohesion), NHD (normalized Hamming distance) 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 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 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. Other recent approaches have addressed class cohesion by considering the relationships between the attributes and methods of a class based on dependence analysis. 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. This measure is based on 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 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.

III. AN INCREMENTAL RETRIVEAL 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 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 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 traceability link recovery between the source code and documentation [3] 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.

A. *Overview of Latent Semantic Indexing*

LSI 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) as it generates a real-valued vector description for documents of text. Results have shown [7], 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) . As a result, LSI offers a way of assessing semantic similarity between any two samples of text in an automatic unsupervised way.

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.

B. *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 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. The following steps are necessary to compute the C3 metric

. 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.

IRC3Mis 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 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, 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

C. 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 (c1; c2 . . . cn). The total number of classes in the system C is n = |C|.

A class has a set of methods. For each class c 2 C, M(C) = (m1; . . .;mk) is the set of methods of class c.

An OO system C is represented as a set of connected graphs G_C =(G1; . . . ;Gn), withGi representing class ci. Each class ci 2 C is represented by a graph Gi € GC such that Gi=(Vi;Ei), whereVi = M(ci) is a set of vertices corresponding to the methods in class ci, and Ei $\subseteq$ VixVi is a set of weighted edges that connect pairs of methods from the class.

Definition 1 (Conceptual Similarity between Methods (CSM)). For every class ci 2 C, all of the edges in Ei are weighted. For each edge (mk,mj) €Ei, we define the weight of that edge CSM(mk,mj) as the conceptual similarity between the methods mk and mj.

The conceptual similarity between two methods mk and mj, that is, CSM(mk;mj), is computed as the cosine between the vectors corresponding to mk and mj 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 vmk and vmj are the vectors corresponding to the mk;mj 2 M(ci) methods, T denotes the transpose, and jvmkj2 is the length of the vector.

For each class c € C, we have a maximum of N = C^2_n 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 2 C is

$$ACSM(c) = \frac{1}{N} \times \sum_{i=1}^{N} CSM(m_i, m_j), \tag{2}$$

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.

where (mi;mj) € E, i ≠ j, mi;mj €M(c), and N is the number of distinct edges in G, as defined in Definition 1.

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 2 C, the conceptual cohesion of c, C3(c) is defined as follows:

$$C3(c) = \begin{cases} ACSM(c), & if \quad ACSM(c) > 0, \\ else, & 0. \end{cases} \tag{3}$$

Based on the above definitions, C3ðcÞ 2 ½0; 1_8 c 2 C. If a class c 2 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, theCSMfor 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).

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

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 C3ðcÞ will be close to zero.

IV. Assessment Of The New Cohesion Measure

Newly proposed metrics require empirical evaluations. 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.

A. *Conceptual Versus Structural Cohesion*

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.

V. 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 accessory may artificially increase or decrease the cohesion of a class. 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.

VI. 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 text.

Acknowledgment

The authors are extremely grateful to the anonymous reviewers for their very pertinent and helpful comments and suggestions, which helped in significantly improving the earlier version of this paper. Software tools are 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. Wu¨ st, "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems," Proc. Fifth IEEE Int'l Software Metrics Symp., pp. 43-53, Nov. 1998.