A Novel Approach for Cohesion Measurement to Improve the Quality of Object Oriented System

[1]N.Rajkumar & [2]C.Viji

[1]Associate Professor, Department of Computer Science and Engineering, SVS College of Engineering, Coimbatore, Tamil Nadu, India.

[2]Assistant Professor, Department of Electronics & Communication Engineering, SVS College of Engineering, Coimbatore, Tamil Nadu, India.

Email: [1]nrajkumar84@gmail.com, [2]vijisvs2012@gmail.com

***Abstract*—software quality measurements have directed to extensive research into software metrics and the development of software metric tools. Develop components which are reusable is a best practice in industry standard. To develop reusable components reduce the relation between each component as possible. Hence, to preserve high quality software, developers need to maintain low-coupled and highly cohesive design. Coupling and cohesion metrics lack proper and standardized definitions and thus for each metric there is more than one clarification. This paper introduces our view of measurement of cohesion for Java projects. *High* cohesion is desirable property in software systems to achieve *reusability* and *maintainability*. In this paper we are measures for cohesion in *Object-Oriented (OO) software* reflect particular interpretations of cohesion and capture different aspects of it. In existing approaches the cohesion is calculate from the structural information such as method *attributes* and *references*. 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 a broad set of existing metrics and uses them to construct models that predict software faults.**

Keywords—cohesion, textual coherence, fault prediction, Latent Semantic Indexing, Software Quality.

I. 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. 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. 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, 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 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 to capture different aspects of cohesion, only a few 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 co reference, 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. 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 and java reflection for retrieving structural relationship. 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.

II. RELATED WORK: COHESION MEASUREMENT IN OO SYSTEMS

There are several approaches to cohesion measurement in OO systems. Many of the existing metrics are adapted from similar cohesion measures for non-OO systems, while some of the metrics are specific to OO software. Based on the fundamental information used to measure the cohesion of a class, one can distinguish structural metrics [8], [11], [20], slice-based metrics, semantic metrics information entropy-based metrics [1], metrics based on data mining, and metrics for specific types of applications like aspect-oriented, knowledge-based, and distributed systems. The class of structural metrics is the most investigated category of cohesion metrics and includes lack of cohesion in methods.

ICH (information-flow-based cohesion), LCOM1, LCOM3, LCOM4, LCOM5, C_O (connectivity), TCC (tight class cohesion) [8], LCC (loose class cohesion) [8], NHD (normalized Hamming distance), etc. Cohesion is dependents on the number of methods pairs that share parameter or methods. The structural metrics are based on relationships among methods and variables. A broad overview of graph theory-based cohesion metrics. Somewhat different in this class of metrics are LCOM5 and Cohesion, which consider that cohesion is depends on number of instance variables in a class that shares the properties with in the class defined a combined framework for cohesion measurement in OO systems [11] which organizes and deliberates all of these metrics. A small set of cohesion metrics was proposed for specific types of applications. 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 proposed a composite cohesion metric which measures the strength of information in a module. This measure is vector based representation and refers the frequencies of occurrences of data type in a module. The approach measures the cohesion of individual subprograms based on the relationships between each other in this vector space. A file-level cohesion metric based on type of information that we are using for our proposed metrics here. The developer of a software system must consider a class as a set of task that estimated the concept from the problem domain implemented by the class as different set of attributes, methods communications. Information that contributes traces about domain concepts is embedded in the source code such as comments and identifiers.

III. AN INFORMATION RETRIEVAL APPROACH TO CLASS COHESION MEASUREMENT

Developers are attempts to control complexity, High cohesion and low coupling among classes are design principles directed to reducing the system complexity. We are trying to measure in our approach. The source code of a software system contains unstructured and structured information. The structured informations are parsed, while the unstructured information such as comments and identifiers is defined for understanding purpose, it won't executed. Our approach is based on the unstructured information as well as structured information which is embedded in the source code. We use LSI, it is an advanced IR method. We have broad experience in using LSI for other software engineering problems like concept and feature location, traceability link recovery between source code and documentation, identification of abstract data types in legacy source code and clone detection.

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 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. LSI relies on an SVD of a matrix (word _ 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.

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.

B. MEASURING TEXT COHERENCE WITH LATENT SEMANTIC INDEXING

There are many aspects that contribute to coherence, including co reference, causal relationships, connectives, and signals. LSI can be applied as an automated method that produces coherence predictions similar to propositional modelling. 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 entire program. Coherence predictions have typically been performed in all level in which a set of propositions all contained within the working memory are compared or connected to each other. 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 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. The overall coherence of a text is measured as the average of all similarity measures between successive sentences.

C. FROM TEXTUAL COHERENCE TO SOFTWARE COHESION

We adapt LSI-based coherence measurement mechanism to measure cohesion in OO system. 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. 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. 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. In software, we interpret a cohesive class as implementing one concept 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 Conceptual Cohesion Class Measurement, 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.

1. **Corpus creation.**

The source code is pre-processed 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.

2. **Corpus indexing**

LSI is used to index the corpus and create an equivalent semantic space.

3. **Computing conceptual similarities.**

Conceptual similarities are computed between each pair of methods.

4. **Computing C3.**

Our source code parser component is based on the Java Model. We use the cosine between vectors in the LSI space to compute conceptual relations.

D. THE CONCEPTUAL COHESION OF CLASSES

In order to define and compute the C3 metric,

$wC3=Mk*Mj /[Mk]_2*[Mj]_2$

K=1, 2, 3……n; n: up to n comments

j=2, 3…..n; n: up to n comments

Definition 1

(CONCEPTUAL SIMILARITY BETWEEN METHODS-(CSM)).

For every class c_i 2 C, all of the edges in E_i 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(m_k *;mj),* is computed as the cosine between the vectors corresponding to *mk* and *mj* in the semantic space constructed by the IR method.

CSM (mk, mj) = vmkt vmj / |vmk|2 |vmj|2

Where vm_k and vmj are the vectors corresponding to the mk; mj 2 M (Ci) methods, T denotes the transpose, and vmkj 2 is the length of the vector. For each class c 2 C, we have a maximum of N = C2 distinct edges between different nodes, where n = |Mc)|. 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.

Definition 2
(AVERAGE CONCEPTUAL SIMILARITY OF METHODS IN A CLASS (ACSM)).
The ACSM c € C is
ACSM(c) = 1/N x ∑*CSM (mi ,mj)*
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.

Table: 1 Conceptual Similarity between the Methods in the example 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(Example) = 0:913.

Definition *3 (c3).*
For a class c € C, the conceptual cohesion of, C3(c) is defined as follows
C3(c) = {ACSM(c), if ACSM(c)>0,
Based on the above definitions, C3 (c) € [0, 1] V c € 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, the CSM for each pair of methods is close to one).

Java Reflection
Java Reflection is quite powerful and can be very useful. Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.
Lcom5
lcom=((((1/a)*Mu)-m)/1-m);
Mu-count for fields ,
m-methods length, a-fields length,
lcom5=Math.*sqrt*(lcom*lcom);

IV Cohesion measuring Framework

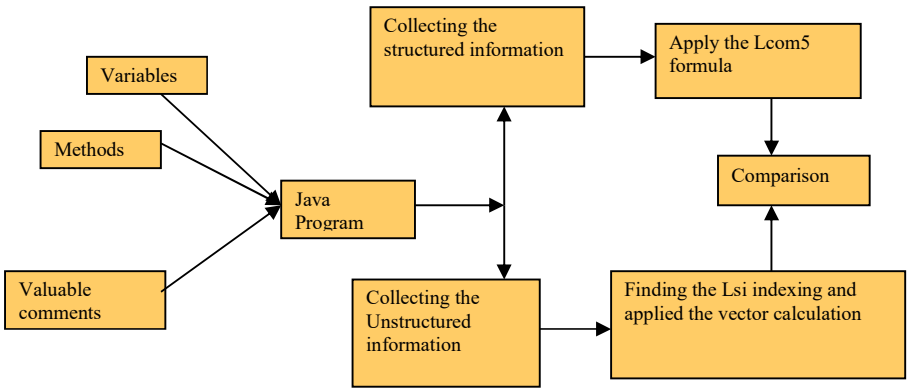

Fig:1 Data Flow Diagram
Our proposed system describes LCOM5 and C3 measure in more detail. First we develop the LCOM5 formula and find out the C3 measure and compare with structure and unstructured data. If the result is equal to one means, the class is less cohesive according to the structured information. Then we are going to retrieve the index terms based on that comments which are present in all the methods. Comments are useful information according to the software engineer.

In concept oriented analysis we are taking the comments. Based on the comments we are going to measure the class is cohesive or not.

Step-1:In this Phase system retrieve structured information like identifiers, (Example Variables).Invocation of declared methods and declared constructors using Java Reflection and it retrieve valid comments.
Step -2: In this Phase system search the declared variables among all the classes. Because the main theme of the declaring class variable is, it should be used in all methods. So that the declared variables are found among all the methods.
Step -3: In this Phase system apply the LCOM5 (Lack of cohesion in methods) formula. If the result is equal to one means, the class is less cohesive according to the structured information.
Step -4: Here we are going to retrieve the index terms based on that comments which are present in all the methods. Comments are useful information according to the software engineer. In concept oriented analysis we are taking the comments. Based on the comments we are going to measure the class is cohesive or not.
Step -5: In this Phase system examine the index terms among the comments which are present in all the comments.
Step -6: In this Phase system apply the conceptual similarity formula. Based on the result we can say the class is cohesive or less cohesive according to concept oriented.
Step -7:In this Phase we are going to compare the two results. Based on the results we can say that cohesion according to structure oriented and unstructured oriented.

IV. CONCLUSION & FUTURE WORK

In conclusion, the approach developed in this paper provides a way to measure metrics for cohesion at class level. The approach is developed into code only for Java projects. In future, we aim to develop the product for other common languages like C#,C++ etc. Also, computation of cohesion at higher levels, package level will be done to show how modules of projects are dependent on each other. The concepts of polymorphism will be taken into consideration for future. 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 are planning user studies 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. 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. Highly cohesive classes need to have a design that ensures a strong coupling among its methods and a coherent internal description.

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, 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," software Reusability, 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.
[11] L.C. Briand, J.W. Daly, and J. Wu¨ st, "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. Wu¨ st, 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. Kemmerer, "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. Kemmerer, "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. Kemmerer, "A Metrics Suite for Object- Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476- 493, June 1994.