ISSN:0975-9646

Implementation of Measuring Conceptual Cohesion

K. Rakesh, S. Sushumna
Computer Science Department,
GITAM Institute of Technology,
GITAM University, AP, INDIA

Abstract:Software cohesion is an important property of software system module. Cohesion is an impact of understanding reuse, understanding, and maintenance. Existing approaches to cohesion measurement in software are largely based on the structural information of the source code, such as attribute references in methods. These measures reflect particular interpretations of cohesion and try to capture different aspects of cohesion and no single cohesion metric or suite is accepted as standard measurement for cohesion. In this paper, we propose a new set of measures for the cohesion of individual classes within a software system. 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.

1. INTRODUCTION

Cohesion is a measure of the relative functional strength of a module. The cohesion of a component is a measure of the closeness of the relationships between its components. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules.

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.

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 such metrics address the conceptual and textual aspects of cohesion.

This paper presents propose a 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 Information Retrieval (IR) technique, namely, Latent Semantic Indexing (LSI), to extract, represents, and analyzes 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.

2. BACK GROUND

Lack of Cohesion in Methods (LCOM): Cohesion refers to the internal consistency within parts of the design. In other words, we could say that it refers to the concept of similarity, given as the intersection of the sets of properties of two things. This concept is applied to pairs of methods, giving the degree of similarity of methods that can be defined as the intersection of sets of attributes used by each of them.

Two methods whose degree of similarity is not zero are said to be Similar.

The degree of similarity of methods can be viewed as a major aspect of object class cohesiveness. If an object class has different methods performing different operations on the same set of instance variables, the class is cohesive.

Yet, Chidamber and Kemerer highlight that software design good-practice calls for maximizing cohesiveness:

- Lack of cohesion (low value) implies that classes should probably be split into two or more sub-classes
- Low cohesion increases complexity
- Disparateness of methods helps identify flaws in the design of classes
- A high LCOM (Lack of Cohesion of Methods) value indicates:
- That the class design resulted in attempt to achieve many different objectives (giving less predictable behavior, error-proneness, complex testing);
- A good class subdivision.

2.1 LCOM1 – Lack of Cohesion Of Methods [Henderson & Sellers]: Henderson-Sellers gave the following interpretation of LCOM:
It is the number of pairs/couples of methods having no common attribute
LCOM1 can be simply calculated by counting all the intersections between method attributes, whose result is the empty set.
Consider class C1 with methods M_1, M_2, M_3... M_n
For each two methods (M_i, M_j) the common class instance variables (class
attributes/properties) are counted:

$$\text{LCOM1} = \sum_{COMMON\ CLASS\ INSTANCE\ VARIABLE} (Mi, Mj)$$

If this sum is equal to zero ($\sum_{COMMON\ CLASS\ INSTANCE\ VARIABLE} (Mi, Mj) = 0$) than 1 is added to the result.

2.2 LCOM2 – Lack of Cohesion Of Methods [Chidamber & Kemerer]: LCOM2 is the count of the number of method pairs whose similarity is 0, minus the count of method pairs whose similarity is not zero.
LCOM2 can be simply calculated by:

- Counting all the intersections between method attributes, whose result is the empty set (P set);
- Counting all the intersections between method attributes, whose result is a non-empty set (Q set)

Consider a Class C1 With *n* methods M1, M2..., Mn

Let {Ij} = set of instance variables used by method Mi.

There are *n*such sets {I1} ... {In}

Let P = $\{(Ii, Ij) | Ii \cap Ij = \emptyset\}$ Q

$=\{(Ii, Ij) | Ii \cap Ij \neq \emptyset\}$

$$\text{LCOM2} = \begin{cases} |P| - |Q| if |P| - |Q| \\ 0 \quad OTHERWISE \end{cases}$$

2.3 LCOM3 – Lack of Cohesion Of Methods [Hitz& Montazeri]: Hitz and Montazeri proposed a different, graph-theoretic formulation of LCOM metric: it is the number of pairs/couples of methods with at least one common instance variable
LCOM3 is then defined as the number of connected components of G, that is, the number of method "clusters" operating on disjoint sets of instance variables. LCOM3 can be calculated by:

- Create a graph **G** whose vertex **V** are class methods;
- Add an edge **E** for each pair of similar methods;
- **LCOM3** is the number of connected components of **G**

$$E = \left\{ (m, n) \in V \times V \middle| \begin{matrix} \exists i \in Ix : (m\ accesses\ i) \\ \cap\ (n\ accesses\ i) \end{matrix} \right\}$$

2.4 LCOM4- Lack of Cohesion of Methods [Hitz& Montazeri]: LCOM3 presents a problem for access methods (methods which provide read or write access to an attribute). As access methods exist, other methods no longer need to directly reference these attributes. In turn, invocation of access methods represents an access to corresponding attributes.
LCOM4 is intended to give a correct representation of cohesion in such a case; the graph is changed, adding edges not only for similar methods, but also for pairs of methods in which one invokes the other. Such methods are said to be indirectly connected (where similar methods are directly connected).
LCOM4 can be calculated by:

- Create a graph **G** whose vertex **V** are class methods;
- Add an edge **E** for each pair of similar methods;
- Add an edge **E** for each pair of indirectly connected methods;
- **LCOM4** is the number of connected components of **G.**

2.4 LCOM5 – Lack of Cohesion Of Methods [Henderson & Sellers]: LCOM5 Metric provides a cohesion Metric expressed as percentage value. Therefore, it has the advantage of falling in a defined range (0 – 1, if each attribute is accessed by at least one method).
Consider a set of methods
$\{Mi\}$ (I= 1, …,*m*)
Accessing a set of attributes
{Aj} (j = 1, …,*a*)
Let the number of methods which access each attribute be μAj.
The simplest formula which gives the properties discussed above (a value of zero with full cohesion and a value of 1 for no cohesion) is:

$$\text{LCOM5} = \frac{\left[\frac{1}{a} * \sum \mu(Aj)\right] - m}{(1-m)}$$

3. FROM IR APPROACH TO CONCEPTUAL COHESION OF CLASS

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 (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 traceability link recovery between the source code and documentation, impact analysis, 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, which is an advanced IR method. Although the general approach 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, traceability link recovery between source code and documentation identification of abstract data types in legacy source code, and clone detection. In addition to other usages in software engineering, LSI has been successfully used in cognitive psychology for the measurement of textual coherence, 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): Latent Semantic Indexing is a variant of the vector-retrieval method in which the dependencies between terms are explicitly modeled and exploited to improve retrieval. One advantage of the LSI representation is that a query can retrieve a relevant document even if they have no words in common.

Most information-retrieval methods depend on exact matches between words in users' queries and words in documents. Typically, documents containing one or more query words are returned to the user. Such methods will, however, fail to retrieve relevant materials that do not share words with users' queries. One reason for this is that the standard retrieval models (e.g., Boolean, standard vector, probabilistic) treat words as if they are independent, although it is quite obvious that they are not. A central theme of LSI is that term-term interrelationships can be automatically modeled and used to improve retrieval; this is critical in cross-language retrieval since direct term matching is of little use.

LSI examines the similarity of the “contexts" in which words appear, and creates a reduced-dimension feature-space representation in which words that occur in similar contexts are near each other. That is, the method first creates a representation that captures the similarity of usage (meaning) of terms and then uses this representation for retrieval. The derived feature space reflects these inter-relationships. LSI uses a method from linear algebra, singular value decomposition (SVD), to discover the important associative relationships. It is not necessary to use any external dictionaries, thesauri, or knowledge bases to determine these word associations because they are derived from a numerical analysis of existing texts. The learned associations are specific to the domain of interest, and are derived completely automatically.

The singular-value decomposition (SVD) technique is closely related to eigenvector decomposition and factor analysis. For information retrieval and filtering applications we begin with a large term-document matrix, in much the same way as vector or Boolean methods do. This term-document matrix is decomposed into a set of k, typically 200-300, orthogonal factors from which the original matrix can be approximated by linear combination; this analysis reveals the “latent" structure in the matrix that is obscured by noise or by variability in word usage.

Figure 1 illustrates the effect of LSI on term representations using a geometric interpretation. Traditional vector methods represent documents as linear combinations of orthogonal terms, as shown in the left half of the figure, so that the angle between two documents depends on the frequency with which the same terms occur in both, without regard to any correlations among the terms. Here, Doc 3 contains Term 2, Doc 1 contains Term 1, and Doc 2 contains both terms. In contrast, LSI represents terms as continuous values on each of the k orthogonal indexing dimensions. Since the number of factors or dimensions is much smaller than the number of unique terms, terms will not be independent as depicted in the right half of Figure 1. When two terms are used in similar contexts (documents), they will have similar vectors in the reduced-dimension LSI representation. LSI partially overcomes some of the deficiencies of assuming independence of words, and provides a way of dealing with synonymy automatically without the need for a manually constructed thesaurus. (Earlier work [4, 6] presented detailed mathematical descriptions and examples of the underlying LSI/SVD method.)

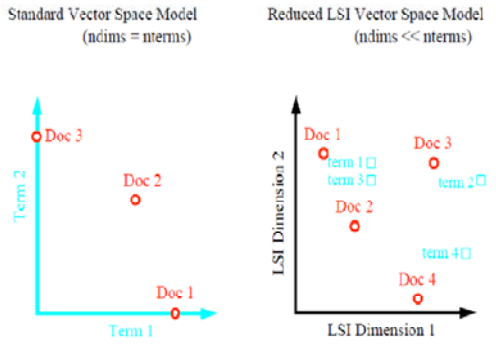

Figure 1 Term representations in the standard vector vs. reduced LSI vector models

The result of the SVD is a set of vectors representing the location of each term and document in the reduced k-dimension LSI representation. Retrieval proceeds by using the terms in a query to identify a point in the space technically, the query is located at the weighted vector sum of its constituent terms. Documents are then ranked by their similarity to the query, typically using a cosine measure of similarity. While the most common retrieval scenario involves returning documents in response to a user query, the LSI representation allows for much more flexible retrieval scenarios. Since both term and document vectors are represented in the same space, similarities between any combination of terms and documents can be easily obtained one can, for example, ask to see a term's nearest documents, a term's nearest terms, a document's nearest terms, or a document's nearest documents. We have found all of these combinations to be useful at one time or another.

New documents (or terms) can be added to the LSI representation using a procedure we call “folding in." This method assumes that the LSI space is a reasonable characterization of the important underlying dimensions of similarity, and that new items can be described in terms of the existing dimensions. Any document not used in the construction of the semantic space is located at the weighted vector sum of its constituent terms. This is exactly how queries are handled and has the desirable mathematical property that a document that is already in the space is folded in at the same location (i.e., it receives the same representation). A new term is located at the vector sum of the documents in which it occurs.

In single-language document retrieval, the LSI method has equalled or outperformed standard vector methods in almost every case, and was as much as 30% better in some cases

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 co reference, causal relationships, connectives, and signals. Existing approaches in cognitive psychology and computational linguistics for automatically measuring text coherence are based on propositional modeling. Foltz et al. 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. 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, 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. 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.
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 Conceptual Cohesion Class Measurement (IRC3M), 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).
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.
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).

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 . . . c_n). The total number of classes in the system C is n = $\left| C \right|$.
A class has a set of methods. For each class c ∈ C, M(c) = {m_1, . , .,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, . . . ,Gn), with G_i representing class c_i. Each class c_i∈ C is represented by a graph G_i∈ 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 ⊂ 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 c_i∈ C, all of the edges in $E_i$are weighted. For each edge (m_k, m_j) ∈ 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):

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

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

For each class c ∈ C, we have a maximum of N = C2n distinct edges between different nodes, where n = $\left| M(c) \right|$
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 in C is

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

Where (m_i,m_j) ∈ E, i≠ j, m_i,m_j∈ 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 ∈ C, the conceptual cohesion of c, C3(c) is defined as follows:

$$C3(c) = f(x) = \begin{cases} ACSM(c), & if ACM(c) > 0, \\ else, & 0 \end{cases}$$

Based on the above definitions, C3(c) ∈[0, 1] for all c ∈ C. If a class c ∈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.

4. EXPERIMENTAL RESULTS OF CONCEPTUAL COHESION

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, semantic metrics, and information entropy-based metrics, 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) 1, LCOM3, LCOM4, LCOM5, 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 ofa 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. Somewhat different in this class of metrics are LCOM5 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.

Other cohesion metrics exploit relationships that underline slicing. A large-scale empirical investigation of slice-based metrics indicated that the slice-based cohesion metricsprovide complementary views of cohesion to the structural metrics. Although the information used by these metrics isalso structural in nature, the mechanism used and theunderlying interpretation of cohesion set these metrics apartfrom the structural metrics group.

A small set of cohesion metrics was proposed for specifictypes of applications. Among those are cohesion metrics forknowledge-based, aspect-oriented systems, and dynamiccohesion metrics for distributed 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. Patel et al. proposed a composite cohesion metric that measures the information strength of a module. 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 ofa system based on the relationships to each other in this vectorspace. Maletic and Marcus defined file-level cohesion metric based on the same type of information that we are using for our proposed metrics here. Even though thesemetrics were not

Specifically designed for the measurement of cohesion inOO 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 cluesabout domain concepts is encoded in the source code ascomments and identifiers. Among the existing cohesion metrics for OO software, the Logical Relatedness of Methods (LORM) and the Lack of Conceptual Cohesion in Methods (LCSM) are the only ones that use this type of information tomeasure the conceptual similarity of the methods in a class.

The philosophy behind this class of metrics, into which ourwork 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 iscohesive. The difficult problem here is how conceptualrelationships can be defined and measured. LORM usesnatural language processing techniques for the analysis needed to measure the conceptual similarity of methods andrepresents a class as a semantic network. LCSM uses thesame information, indexed with LSI, and represents classes as graphs that have methods as nodes. It uses a countingmechanism similar to LCOM.

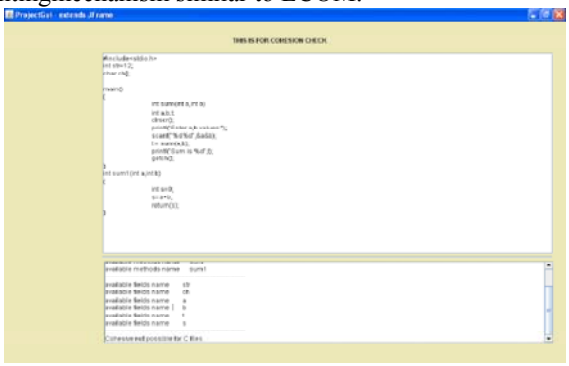

Fig 2: Screen 1

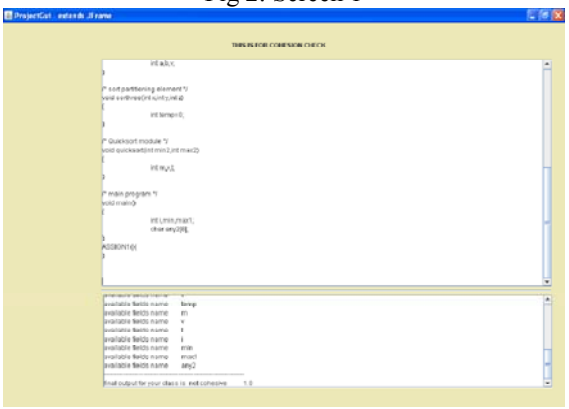

Fig 3: Screen 2

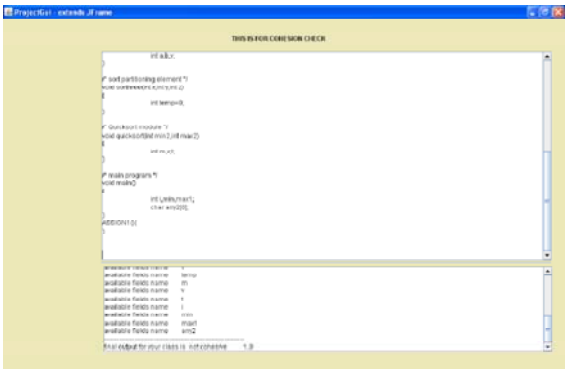

Fig 4: Screen 3

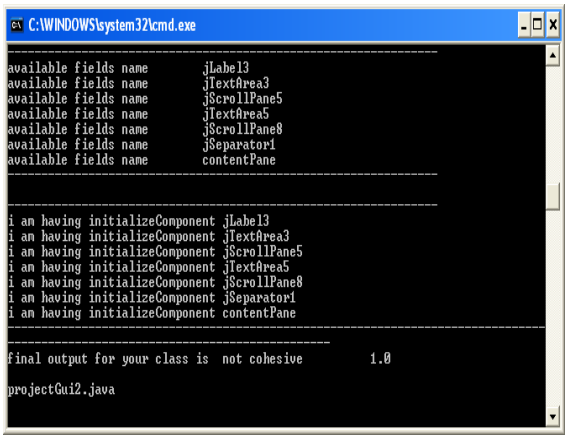

Fig 5: Screen 4

5. CONCLUSION

Object-oriented systems classes 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; 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. Latent Semantic “Indexing can be used in similar manner to measuring the coherence of natural languages.

REFERENCES

1. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing by Andrian Marcus, Jonathan I. MaleticDepartment of Computer Science Kent State University Kent Ohio 44242 330 672 9039 amarcus @ cs. ken t. edu, jmaletic @ cs. ken t. edu
2. A Comprehensive Empirical Validation of Design Measures for Object Oriented Systems by Lionel C. Briand, John Daly1, Victor Porter1, Jürgen Wüst, Fraunhofer IESE, Sauerwiesen 6, D-67661 Kaiserslautern, Germany {briand,wuest}@iese.fhg.de, Tel. +49 6301 707 251, Fax +49 6301 707 202
3. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction by TiborGyimo´ thy, Rudolf Ferenc, and Istva´nSiket
4. The Conceptual Cohesion of Classes by Andrian Marcus, Denys PoshyvanykDepartment of Computer Science Wayne State University, Detroit Michigan 48202, 313 577 5408 amarcus@wayne.edu, denys@cs.wayne.edu.
5. Working Session: Information Retrieval Based Approaches in Software Evolution by Andrian Marcus, Andrea De Lucia, Jane Huffman Hayes, Denys Poshyvanyk
6. An Information Retrieval Approach to Concept Location in Source Code by Andrian Marcus, AndreySergeyev, VáclavRajlich, Jonathan I. Maletic
7. Measuring Coupling and Cohesion In Object-Oriented Systems by Martin Hitz, Behzad MontazeriInstitutfürAngewandteInformatik und Systemanalyse, University of Vienna hitz@ifs.univie.ac.at
8. Automatic Cross-Language Information Retrieval using Latent Semantic Indexing by Michael L. Littman,Susan T. Dumais, Thomas K. Landauer , October 7, 1996
9. Property-based software engineering measurement by Lionel **C.** Briand, SandroMorasca, Member, /E€€ Computer Society, and Victor R. Basili, Fellow, /€€E
10. A Validation of Object-Oriented Design Metrics as Quality Indicators by Victor R. Basili, *Fellow, I€€€,* Lionel C. Briand, and Walcelio L. Melo, *Member, IEEE Computer Society.*
11. Measuring Coupling and Cohesion of Software Modules: An Information Theory Approach by Edward B. Allen Mississippi State University Mississippi USA, Taghi M. Khoshgoftaar, Ye Chen Florida Atlantic University Boca Raton, Florida USA.
12. Recovering Traceability Links between Code and Documentation by GiulianoAntoniol, Member, IEEE, Gerardo Canfora, Member, IEEE, Gerardo Casazza, Member, IEEE, Andrea De Lucia, Member, IEEE, and Ettore Merlo, Member, IEEE
13. Dynamic Coupling Measurement for Object-Oriented Software by Erik Arisholm, Member, IEEE, Lionel C. Briand, Member, IEEE, and AudunFøyen

AUTHORS BIOGRAPHY

K.Rakesh is pursuing M.Tech in Software Engineering from GITAM UNIVERSITY, Visakhapatnam, A.P., INDIA. My research areas include Software coupling and cohesion, cost estimation, change management.

S.Sushumna is pursuing M.Tech in Software Engineering from GITAM UNIVERSITY, Visakhapatnam, A.P., INDIA. My research areas include Software coupling and cohesion, change management.