ISSN 2047-3338

Conceptual Cohesion of Classes in Object Oriented Systems

S. Megha Chandrika[1], E. Suresh Babu[2] and N. Srikanth[3]
[1]SCIENT Institute of Technology, India
[2]Samskruti Engineering College, India
[3]Nishitha College of Engineering and Technology (JNTUH), India

***Abstract*— Cohesion measures in Object-oriented software reflect particular interpretations, High cohesion positively impacts understanding, reuse and maintenance. This paper proposes a new measure based on analysis of the unstructured information embedded in the source code, such as comments and identifiers, we have the existing applications based on using the only the structural information from the source code, attribute references in methods to measure cohesion. The new measure named the Conceptual cohesion of classes is the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics, 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 design concepts and novel measure captures different aspects of class cohesion compared to any of the existing cohesion measures.**

***Index Terms*– Cohesion, Types Cohesion and Design Concepts**

I. INTRODUCTION

SOFTWARE cohesion can be defined as a measure of the degree to which elements of a module belong together [5]. Cohesion is also regarded from a functional 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, while very intuitive, are quite vague and make cohesion measurement a difficult task, leaving too much room for interpretation. 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 among them. These class properties facilitate comprehension, testing, reusability, maintainability, etc.

The concept of software cohesion has its roots in the 1970's when Stevens et al. [7] started looking at inter-module metrics for procedural software. Yourdon and Constantine later categorized cohesion on a seven point ordinal scale from functional at one end to coincidental at the other [9]. Since then, various attempts in the object-oriented community have been made to capture cohesion through software metrics [3, 4, 5]. The best known and most investigated of these metrics is the Lack of Cohesion in Methods of a class (LCOM) proposed by Chidamber and Kemerer (C&K) [4]. The LCOM metric rates a class as cohesive if every method uses every instance variable; at the other extreme, a class whose methods use disjoint instance variables is considered Uncohesive.

This paper is organized as follows. Section 2 presents the Cohesion using conceptual classes in object oriented systems, Section 3 presents the Related Design concepts using conceptual cohesion of classes in Object oriented systems, Section 4 Presents the our proposed system Using Conceptual cohesion of classes in object oriented systems compares our study with other works on the subject. Section 5 concludes the paper by presenting lessons learned and future work.

II. COHESION USING CONCEPTUAL CLASSES IN OBJECT ORIENTED SYSTEMS

A. What is Cohesion?

Cohesion is a measure of how well the lines of source code within a module work together to provide a specific piece of functionality. In object-oriented programming, the degree to which a method implements a single function; methods that implement a single function are described as having high cohesion.

B. Types of Cohesion

1). Method Cohesion: What has been stated in the realm of coupling also holds true for cohesion. Since methods equal modules to a very high degree both bracket pieces of code implementing some functionality. We adopt the various degrees of classical cohesion [11], [12] to describe method cohesion. In contrast to coupling we do not even have to change the various notions of classical cohesion considerably. In the following seven degrees of cohesion classical cohesion adapted for method cohesion are summarized from worst to best:

- *Coincident:* The elements of a method have nothing in common besides being within the same method
- *Logical:* The elements with similar functionality, such as input/output handling and error handling are collected in one method
- *Temporal:* The elements of a method have logical cohesion and are performed at the same time.
- *Procedural:* The elements of methods are connected by some control flow.
- *Communicational:* The elements of a method are connected by some control flow and operate on the same set of data
- *Sequential:* The elements of method have communicational cohesion and are connected by a sequential control flow
- *Functional:* The elements of a method have sequential cohesion and all elements contribute to a single task of the problem domain. Functional cohesion is the best form of method cohesion since it fully supports the principle of locality and thus minimizes maintenance efforts.

For the discussion of class cohesion and inheritance cohesion we assume that all methods have functional cohesion. The reason is that in order to determine class/inheritance cohesion we have to investigate the relationship between methods and instance variables. Low cohesive methods which access most of the instance variables could fake a high degree of class/inheritance cohesion

2) .Class Cohesion: Class cohesion describes the binding of the elements define with in the same object class, not considering inherited instance variables and inherited methods. Since ignoring inheritance an object class resembles an abstract data type and since the cohesion of abstract data types has been analyzed in detail by Embley and Woodfield in [14] we build our classification of various degrees of class cohesion on that of [14] and redefine their definitions according to the idiosyncracy of object-oriented systems.

Abstract data types in procedure-oriented systems provide functionality to other abstract data types or to modules which are not abstract data types. In contrast, code in object-oriented systems is in general a method bound to a class. Thus for procedure-oriented systems with abstract data types we have to argue which functionality we factor out to abstract data types whereas in object-oriented systems we have to consider which methods are assigned to which classes.

A further crucial difference between abstract data types in the notion of Embley and Woodfield and classes is implied by the concept of object identity. Whereas a single abstract data type can export different domains an object class describes exactly one set of objects where each object is uniquely identified by some system-defined object identifier. Depending on the cohesiveness of a class its objects represent a single, semantic meaningful data abstraction or several, more or less related data abstractions. In the following we discuss the various degrees of class cohesion from worst, i.e., lowest to best i.e., highest Separable.

The cohesion of a class is rated separable if its objects represent multiple unrelated data abstractions combined in one object. This is often the case if the instance variables and methods of a class can be partitioned into two or more sets such that no method of one set uses instance variables or invokes methods of a different set. In particular the cohesion of an object class is rated separable if there is a method which does neither access any instance variable nor invokes any method of the class or there is an instance variable which is not referenced by any of the class methods. A class with separable cohesion should be split into several classes each representing a single data abstraction, i.e., a single semantic concept.

Example: Consider the object class Employee as defined:

```
Class EMPLOYEE {
    …
    (IntcomputeCompany Revenue (SET<PROJECT *)*p);
    …
};
```

The method compute Company Revenue takes all projects of a company as input parameter and computes the accumulated revenue of that company. It neither accesses any instance variables of EMPLOYEE nor does it invoke any other method of EMPLOYEE. Thus the cohesion of EMPLOYEE is separable strength. To improve its cohesion the method computeCompany Revenue should be factored out into a different object class, e.g., into class COMPANY.

III. COHESION AND DESIGN QUALITY IN OBJECT ORIENTED SYSTEM

Object oriented system is a good design for imperatives to building a quality. For this, quantification of the design property is required. Several software metrics have been developed to assess and control the design phase and its products. One of the most vital criteria in Object Oriented design is cohesion. A module is said to have a strong cohesion if it closely characterized with one task of the problem domain, and all its components contribute to this single task. Cohesion was introduced by Yourdon and Constantine as "how tightly bound or related the internal elements of a module are to one another". According to design quality, cohesion is an attribute, not of any code, but of a design that can be utilized to forecast reusability, maintainability, and changeability.

A. Cohesion and Cohesion Metrics

A class is cohesive if it cannot be partitioned into two or more sets defined as follows Each set contains instance variables and methods. Methods of one set do not access variables of another set either directly or indirectly. By way of defining cohesion metrics, many authors have effectually defined class cohesion. So far as the Object Oriented model is concerned, almost all of the cohesion metrics are influenced by the LCOM metric that is defined by Chidamber and Kemerer. According to them, "if an object class has different methods performing different operations on the same set of instance variables, the class is cohesive". The LCOM (Lack of Cohesion in Methods) defined by them is the result gained from deducting the number of pairs of methods in a class having no common attributes from the number of pairs of methods in a class sharing at least one attribute. If the value reached in this calculation is in the negative, the metric is set to zero. This is one metric for assessing cohesion. Likewise, Li and Henry defined LCOM as the number of disjoint sets of methods accessing similar instance variables.

Hitz and Montazeri reaffirm Li's definition of LCOM based on the graph theory which defines LCOM as the number of connected components of a graph. A graph consists of vertices and edges. Vertices represent methods. There is an edge between 2 vertices if the corresponding methods access the same instance variable. Hitz and Montazeri propose to divide a class into smaller, more cohesive classes, if LCOM > 1.

B. Design Quality

1). Abstraction: Abstraction is an OOP concept. It provides a facility to hide some unimportant information and provide us some information which is important for the client programmers.

eg., If we consider a car which has lot of parts such as wheels steering DVD player etc.

We need to know how to use it. We need not to know, what is the structure of all these parts to buy and drive a car?

eg., is Television. The Television has lot of properties and behaviors' like height width display On, display Off etc. and also it has chips and internal wires which enables the television's functions.

But for working the Television we do not need to know these internal things.

2). Architecture: The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them. The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

C. Modularity

Modularity refers to breaking down software into different parts. These parts have different names depending on your programming paradigm (for example, we talk about modules in imperative programming and objects in object oriented programming). By breaking the project down into pieces, it's (i) easier to both FIX (you can isolate problems easier) and (ii) allows you to REUSE the pieces.

D. Refinement

In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specification terminates when all instructions are expressed in terms of any underlying computer or programming language.

Patterns are a way to describe some best practices used in designing software applications. A pattern describes a solution to a recurring design problem. The design patterns are broken down into three subsections: Creational, Structural, and Behavioral patterns.

E. Patterns

Creational patterns are used to create objects in an application. Patterns like Factory Method are used to defer the instantiation of an object to inherited sub classes while Composite pattern allows for a recursive, tree structure of containers and elements.

Structural patterns are used to design the structure of modules in an application. For example, adapter can be used to modify an existing module to work with a developing module. The bridge pattern has a similar use. The composite pattern can also be considered a structural pattern because of the tree structure that is created.

Behavioral patterns describe how objects communicate with each other. The observer pattern is used to notify many classes of a change in the application. The mediator pattern can be used to augment communication between classes, without all of the classes knowing about each other.

F. Information Hiding

In computer science, information hiding is the principle of segregation of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

The term *encapsulation* is often used interchangeably with information hiding. Not all agree on the distinctions between the two though; one may think of information hiding as being the principle and encapsulation being the technique. A software module hides information by encapsulating the information into a module or other construct which presents an interface. A common use of information hiding is to hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program.

In object- oriented programming, information hiding (by way of nesting of types) reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

G. Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. Refactoring is used to improve code quality, reliability, and maintainability throughout the software lifecycle. Code design and code quality are enhanced with refactoring. Refactoring also increases developer productivity and increases code reuse.

For example, if two methods use a similar piece of code, the common code can be refactored into another method that the two parent methods can then call.

IV. CONCEPTUAL COHESION OF CLASSES PROPOSED APPROACH IN OBJECT ORIENTED SYSTEMS

A. Process of LCOM and C3

The following proposed system describes LCOM and C3 measure in more detail. There is no existing system exit using cohesion, first we develop the LCOM formula and find out the C3 measure and compare with structure and unstructured data

In this paper, we introduce the concept of how to find out cohesion in object oriented system. To identify the cohesion in oops First we need to calculate the LCOM5 and C3 measure then compare LCOM5 and C3 measure with Structure and Unstructured data. In this module we are going to take the structured information like identifiers, (Example Variables). Invocation of declared methods and declared constructors, here the Java program should be well compiled and it should be valid comments.

In this module deals we are going to search the declared variables among all the classes. Because the main theme of the declaring class variable is, it sh used in all methods. So that the declared variables are found among all the methods. In this module we are going to 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. 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.

In this module we are going to check the index terms among the comments which are present in all the comments. In this module we are going to apply the conceptual similarity formula. Based on the result we can say the class is cohesive or less cohesive according to concept oriented. In this module we are going to compare the two results. Based on the results we can say that cohesion according to structure oriented and unstructured oriented.

B. Formula for LCOM and C3 Measures

LCOM5 was defined by Henderson-Sellers (1996). It predominantly looks at the number of methods that access each of the set of attributes or data, specifically only the instance variables. Thus, LCOM5 does not deal with data to data interactions and the non-instance variables. It focuses on instance variables to method interactions. For LCOM5 having a value of 0 is considered perfect cohesion.

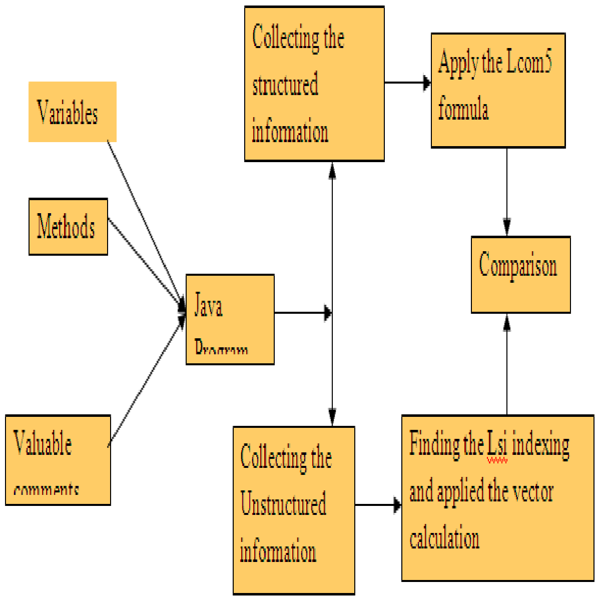

Variables
Methods
Valuable comments
Java Program
Collecting the structured information
Apply the Lcom5 formula
Comparison
Collecting the Unstructured information
Finding the Lsi indexing and applied the vector calculation

Fig. 1. This idea of the figure is taken from Ref no. [10]

Formula for LCOM5

*Lcom = ((((1/a)*Mu)-m)/deno);*

Mu-count for fields, m-

Methods length, a-fields

Length, deno=1-m;

*Lcom51 = lcom*lcom;*

Lcom5 = Math.sqrt(lcom51);

V. EXPERIMENTAL RESULTS OF 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, semantic metrics, 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, Co (connectivity), LCOM5, Coh, TCC (tight class cohesion), LCC (loose class cohesion), 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 is given by Zhou et al. 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.

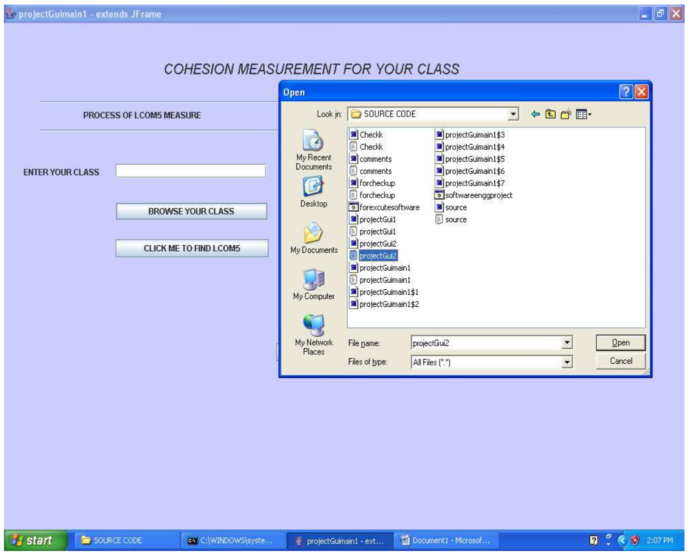
Other cohesion metrics exploit relationships that underline slicing. A large-scale empirical investigation of slice-based metrics indicated that the slice-based cohesion metrics provide complementary views of cohesion to the structural metrics. Although the information used by these metrics is also structural 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, aspect-oriented systems, and dynamic cohesion 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 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. 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)] and the Lack of Conceptual Cohesion in Methods (LCSM) 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.

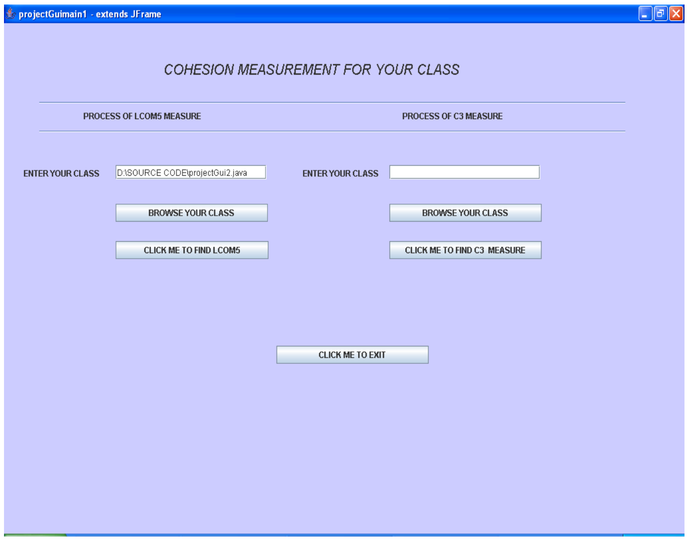


Look in: SOURCE CODE

Checkk
Checkk
comments
comments
forcheckup
forcheckup
forexcutesoftware
projectGui1
projectGui1
projectGui2
projectGui2
projectGuimain1
projectGuimain1
projectGuimain1$1
projectGuimain1$2
projectGuimain1$3
projectGuimain1$4
projectGuimain1$5
projectGuimain1$6
projectGuimain1$7
softwareenggproject
source
source

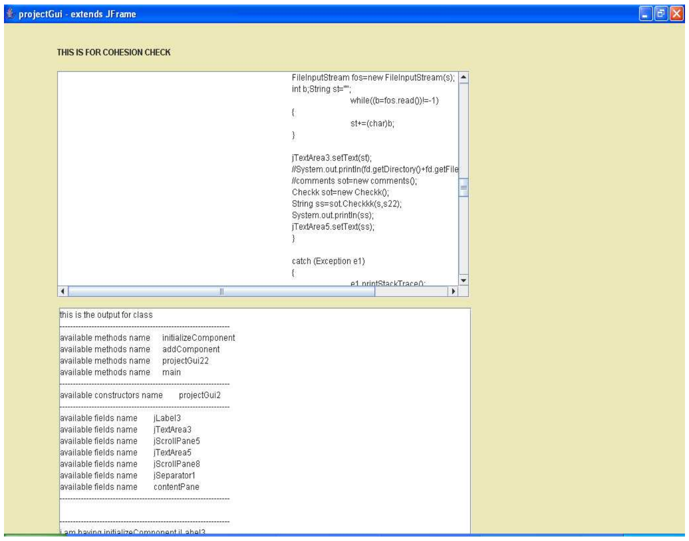
File name: projectGui2
Files of type: All Files (*.*)

Fig. 2. Screen 1



ENTER YOUR CLASS D:\SOURCE CODE\projectGui2.java

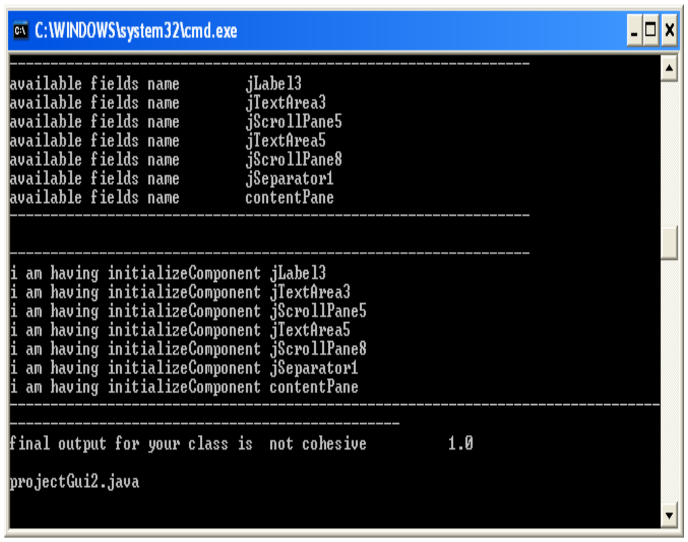
Fig. 3. Screen 2



```
FileInputStream fos=new FileInputStream(s);
int b;String st="";
                    while((b=fos.read())!=-1)
{
                    st+=(char)b;
}

jTextArea3.setText(st);
//System.out.println(fd.getDirectory()+fd.getFile
//comments sot=new comments();
Checkk sot=new Checkk();
String ss=sot.Checkkk(s,s22);
System.out.println(ss);
jTextArea5.setText(ss);
}

catch (Exception e1)
{
                    e1.printStackTrace();
```

```
this is the output for class
------------------------------------------------------------
available methods name    initializeComponent
available methods name    addComponent
available methods name    projectGui22
available methods name    main
------------------------------------------------------------
available constructors name      projectGui2
------------------------------------------------------------
available fields name      jLabel3
available fields name      jTextArea3
available fields name      jScrollPane5
available fields name      jTextArea5
available fields name      jScrollPane8
available fields name      jSeparator1
available fields name      contentPane
------------------------------------------------------------

------------------------------------------------------------
i am having initializeComponent jLabel3
```

Fig. 4. Screen 3



```
----------------------------------------------------------------------
available fields name        jLabel3
available fields name        jTextArea3
available fields name        jScrollPane5
available fields name        jTextArea5
available fields name        jScrollPane8
available fields name        jSeparator1
available fields name        contentPane
----------------------------------------------------------------------

----------------------------------------------------------------------
i am having initializeComponent jLabel3
i am having initializeComponent jTextArea3
i am having initializeComponent jScrollPane5
i am having initializeComponent jTextArea5
i am having initializeComponent jScrollPane8
i am having initializeComponent jSeparator1
i am having initializeComponent contentPane
---------------------------------------------------------------------------------------
-----------------------------------------------------
final output for your class is  not cohesive         1.0

projectGui2.java
```

Fig. 5. Screen 4

VI. 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]. Anquetil, N. and Lethbridge, T., "Assessing the Relevanceof Identifier Names in a Legacy Software System", in Proceedings of Annual IBM Centers for Advanced Studies Conference (CASCON'98), December 1998, pp. 213-222.

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

[3]. J. Bansiya, L. Etzkorn, C. Davis and W. Li. A class cohesion metric for object-oriented designs. Journal of Object-Oriented Programming (January), pages 47-52, 1999.

[4]. S. R. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. IEEE Transactions on Software Engineering, 20(6): 467-493, 1994.

[5]. S. Counsell, E. Mendes and S. Swift, Comprehension of Object-oriented Software Cohesion: the empirical quagmire Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002). Paris, France, pages 33-42, 2002.

[6]. Cho, E. S., Kim, C. J., Kim, D. D., and Rhew, S. Y., "Static and dynamic metrics for effective object clustering", in Proceedings of Asia Pacific International Conference on Software Engineering, 1998, pp. 78 - 85.

[7]. W. P. Stevens, G. J. Myers and L. L Constantine Structured Design. IBM Systems Journal, 13(2): 115-139, 1974.

[8]. A. Weinand, E. Gamma and R. Marty. ET++ - an object-oriented application framework in C++,. Proceedings of Object-oriented Programming Systems, Languages and Applications (OOPSLA), San Diego, USA, pages 46-57, 1988.

[9]. E. Yourdon and L. Constantine, Structured Design, Prentice Hall, 1979. Proceedings

[10]. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems, IEEE Transactions on Software Engineering Vol 34, No. 2, March/April 2008

[11]. W. Stevens G. Myres and L.Constantine. "Structured Design." In IBM Systems Journal .vol.13.pp.115 139.1974.

[12]. E. Yourdon and L.L Constantine. Structured Design . Prentice Hall.1979.

[13]. "Coherency of Classes to Measure the Quality of Object Oriented Design an Empirical Analysis", M.V.VIJAYA SARADHI1, B.R.SASTRY.

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

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

[16]. 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.

[17]. 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.

[18]. 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.

[19]. 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

[20]. H. Kabaili, R.K. Keller, F. Lustman, and G.Saint-Denis, "Class Cohesion Revisited: AnEmpirical Study on Industrial Systems," Proc. Fourth Int'l ECOOP Workshop Quantitative Approaches in Object-Oriented Software Eng., pp. 29-38, 2000.

[21]. H. Kabaili, R.K. Keller, and F. Lustman,"Cohesion as Changeability Indicator in Object-Oriented Systems," Proc. Fifth European Conf. Software Maintenance and Reeng., 2001.

[22]. W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," J. Systems and Software, vol. 23, no. 2, pp. 111-122, 1993.

[23]. M. Linton, P.R. Calder, and J.M. Vlissides,"InterViews: A C++ Graphical Interface Toolkit," Technical Report CSL-TR-88-358, Stanford Univ., 1988,ftp://interviews.stanford.edu/pub.

[24]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design. Prentice Hall, 1991.

[25]. W. Stevens, G. Myers, and L. Constantine, "Structured Design," IBM Systems J., vol. 12, no. 2, 1974.

[26]. 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.

Ms. S. Megha Chandrika, Assistant Professor from SCIENT Institute of Technology, B.Tech Computer science from Nizam Institute of Engg & Tech (JNTUH) and M Tech Software Engineering From GuruNank Engg College (JNTUH) has 6 years of experience in Academic. Guided many UG & PG engineering students. Papers was published in National & International journals, areas of interest are Software Engineering, Data Mining, Software Testing, Compiler design, Web Applications and Unified Modeling Languages.

Mr. E. Suresh Babu, Assistant Professor from Samskruti Engg College, B.Tech from Vathslaya Institute of Scie & Tech (JNTUH) M.Tech from SKTRCE (JNTUH). His areas of interest include Data Mining, and Software Engineering, Software Testing Methodology and Network Security.

N. Srikanth, Pursuing M.Tech Software Engineering from Nishitha College of Engg & Tech (JNTUH). His areas of interest include Mobile Computing, Networks, and Software Engineering.