Improving Cohesion Measurement for Object Oriented Systems in Java

Anees Thankachan
M. Tech Student
Department of Computer Science & Engineering
Mar Baselios College of Engineering & Technology, Nalanchira, Tvm-15

Abstract

High cohesion in classes is an essential property of Object Oriented (OO) software as it positively impacts understanding, reuse, and maintenance. Existing approaches to measure cohesion are largely based on using the structural information from the source code, such as attribute references in methods. The measure, named the Conceptual Cohesion of Classes (C3), for the cohesion of classes is based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. To retrieve the unstructured information from the source code Latent Semantic Indexing is used. C3 is inspired by the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics. This measurement is used to construct models that predict software faults. This paper aims at developing a tool named “C-CUBE “to compute the conceptual cohesion of OO classes.

Keywords: C-CUBE, C3, OO Classes

I. INTRODUCTION

Aim of Software Engineering is to develop high quality software. During software design phase, itself internal software quality attributes such as coupling, cohesion and complexity are considered. 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. Many different OO cohesion metrics have been proposed which try to capture different aspects of cohesion or reflect a particular interpretation of cohesion.

The measures and metrics for cohesion in the software are useful in different tasks, including the assessment of design quality, productivity, reuse effort, prediction of software quality, fault prediction, modularization of software, and identification of reusable 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 trade-off is that such measures deal with information that can be automatically extracted from software and be 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.

A measure for class cohesion is the Conceptual Cohesion of Classes (C3) [1], 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.

C3 [1] 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, an Information Retrieval (IR) technique [7], namely, Latent Semantic Indexing (LSI), can be used to extract, represent, and analyze the textual information from the
source code. The 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 easily understood, 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.

Conceptual Cohesion of Classes (C3) [1], 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.

This paper aims at developing a tool named C CUBE which implements conceptual cohesion methodology to calculate the cohesion value of classes in software. The user can upload the project to the C CUBE, he can view the class cohesion, overall project cohesion and also it calculating the cohesion of derived classes.

II. RELATED WORKS

There are different approaches to measure cohesion in OO systems. Based on the underlying information to measure cohesion one can distinguish C K metrics, Semantic Entropy Metrics, Knowledge based metrics and Latent semantic analysis.

A. A Metrics Suite for Object Oriented Design

These metrics are based on measurement theory and also reflect the viewpoints of experienced OO software developers. The structural metric suite introduced here - Chidamber and Kemerer (CK)[2] metric suite is completely related to the structural aspects of the source code. The Chidamber & Kemerer metrics suite originally consists of 6 metrics calculated for each class. They are Weighted Methods per Class (WMC)

WMC is simply the method count of a class. Keep WMC down. A high WMC has been found to lead to more faults.

1) Depth of Inheritance Tree (DIT)

DIT is the maximum inheritance path from the class to the root class. The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex.

2) Number of Children (NOC)

NOC equals the number of immediate child classes derived from a base class.

3) Coupling between Object classes (CBO)

CBO is the number of classes to which a class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

4) Response for a Class (RFC)

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class.

CK metric suite is efficient in predicting structural cohesion and is widely used in many current Software Engineering practices. CK metric suites are particularly designed for the structural aspects of class cohesion. They do not provide a better understanding of whether a class is conceptually cohesive.

B. A Semantic Entropy Metrics:

This is a semantically based metric for object oriented systems, called Semantic Class Definition Entropy (SCDE) [6] metrics. SCDE metrics measure the complexity of classes in OO software, using domain related information from class documentation (comments and identifiers). In order to retrieve unstructured (comments and identifiers) information from the source code, SCDE uses an information retrieval technique called PATRicia (Program Analysis Tool for Reuse). PATRicia systems identify reusable components in OO software. The module of PATRicia system that handles program understanding and information extraction is called CHRiS (Conceptual Hierarchy for Reuse including Semantics). When analysing comments CHRiS first parses a sentence using a natural language parser, and then uses its inference engine to semantically process the various parses. In the case of identifiers CHRiS uses empirical information on common formats for variables and function identifiers to syntactically tag the sub keywords.

The SCDE metric is defined as,

\[
\text{SCDE} = - \sum_{i=1}^{N_1} f_i \log_b (\frac{f_i}{N_1})
\]

where \( N_1 \) is the number of non-unique domain related concepts or keyword; \( f_i \) is the frequency of occurrence of \( i \)th domain related concepts of keywords. The validation of SCDE metric as domain - task complexity metric alone would allow the separate analysis of domain - task complexity and code complexity. SCDE metric fails to provide a clear idea about the conceptual class cohesion even though it can be extended for it.

PATRicia method of information retrieval fails when polysemy or synonymy exists in the unstructured information of the source code. An efficient information retrieval method is essential for the SCDE metric to extend it for showing conceptual cohesion of classes.
C. Semantic Metrics Object Oriented Design:

This is a suite of semantically derived object oriented metrics, which provides a more direct mapping from the metric to its associated quality factor than is possible using syntactic metrics. These semantically-derived metrics are calculated using knowledge-based, program understanding and natural language processing techniques. Authors propose a metric called Logical Relatedness of Method (LORM) [5] to measure cohesion of a class in OO software. Most desirable cohesion is the model cohesion in which class represents a single semantically meaningful concept. To determine whether a class represents a single semantically meaningful concept, the LORM metrics measures the conceptual relatedness of methods of the class as determined by the understanding of the class methods represented by a semantic network of conceptual graphs.

In order to extract the semantics in the source code (comments and identifiers), PATRicia system is used as in the case of SCDE metrics. To determine whether a particular class or class hierarchy is useful, and therefore potentially reusable in a given domain the PATRicia system performs natural language understanding of comments and identifiers drawn from the object oriented code. LORM is defined as the ratio of total number of relations in the class to total number of possible relations in the class. Other metrics in this suite are LORM2, LORM3 etc. Some quantities such as key class identification, class overlap and documentation quality can be measured with these metrics, whereas measuring these quantities with syntactic or structural metrics is difficult.

LORM is an efficient method for measuring conceptual cohesion but information retrieval technique used here (PATRicia) lacks efficiency. So, a much stronger information retrieval method is essential.

D. A Knowledge-Based Cohesion Metric for Object-Oriented Software

This presents Percentage of Shared Ideas (PSI) [3], a metric for measuring the semantic cohesion of a class in object-oriented software. PSI uses information in a knowledge base to quantify the cohesiveness of a class’s task in the problem domain, allowing a clearer view of cohesion than code syntax provides. Furthermore, this metric is independent of code structure and could be calculated before implementation, providing clues to design flaws earlier in the software development cycle, when changes are less expensive. Authors created a tool called semMet to compute semantic metrics from the source code of software systems. SemMet incorporates PATRicia (Program Analysis Tool for Reuse) system, a mature program understanding engine. Program understanding approaches can be divided into three categories: algorithmic, transformational, and knowledge-based. From there, knowledge-based approaches can be divided into three categories: graph- parsing, heuristic, and using informal tokens. Informal token program understanding approaches include the DESIRE (DESign Information Recovery Environment) system, and the PATRicia system. The PATRicia system is a knowledge-based approach that incorporates a hybrid of heuristics and informal token use.

PSI is defined as the ratio of number of concepts or keywords shared by at least two member functions of a class to the number of concepts or keywords belonging to any member function in the class. PSI is empirically and theoretically valid, and that it matches well with experts’ views of cohesion, performing better than any version of LCOM (CK metric suite).

PSI does not rely on the structure of code; it could be calculated in the design phase, yielding results before implementation. PSI uses PATRicia and DESIRE systems for information retrieval which are less efficient in dealing with polysemy and synonymy, so a better Information retrieval technique is essential.

E. Indexing by Latent Semantic Analysis:

This is a new approach to automatic indexing and retrieval called Latent Semantic Indexing (LSI) [4]. It is designed to overcome a fundamental problem that plagues existing retrieval techniques that try to match words of queries with words of documents. The problem is that users want to retrieve on the basis of conceptual content, and individual words provide unreliable evidence about the conceptual topic or meaning of a document. LSI addresses this problem. Latent Semantic Analysis arose from the problem of how to find relevant documents from search words. The fundamental difficulty arises when we compare words to find relevant documents, because what we really want to do is compare the meanings or concepts behind the words. LSA attempts to solve this problem by mapping both words and documents into a concept space and doing the comparison in this space. LSI uses a technique called Singular Value Decomposition (SVD) by which a term-document matrix is divided into three other matrices: a term matrix, a singular matrix and a document matrix. Initial term-document matrix contains terms on one axis and documents on other. A cell in the matrix denoted occurrence of a particular term in document. After taking the dot product of the three matrices we will get vector space for the document. LSI overcomes two of the most problematic constraints of Boolean keyword queries: multiple words that have similar meanings (synonymy) and words that have more than one meaning (polysemy). Synonymy is often the cause of mismatches in the vocabulary used by the authors of documents and the users of information retrieval systems. As a result, Boolean or keyword queries often return irrelevant results and miss information that is relevant.

LSI is also used to perform automated document categorization. In fact, several experiments have demonstrated that there are a number of correlations between the way LSI and human’s process categorize text.
III. METHODOLOGY

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. The C-CUBE contains two phases, an information retrieval phase and metrics calculation phase. Information retrieval phase includes extraction of unstructured data from source code (comments and identifiers). In order to extract and analyze the unstructured information from the source code, Latent Semantic Indexing (LSI) has to be used. 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. 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. The metrics calculation phase includes the actual calculation of C3 metrics based on the output received from the Information Retrieval phase. In order to compute the C3 metric, a graph-based system representation similar to that used to compute other cohesion metrics has to be used. 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 \( n = |C| \). A class has a set of methods. For each class \( c \in C \), \( M(c) = \{ m_1, m_2, ..., m_k \} \) is the set of methods of class \( c \). Extract comments and identifiers from the methods and produce a text corpus. From this generate a matrix, calculate conceptual similarity between the methods and calculate the cohesion value of classes.

A. Flow Diagram of C Cube Methodology:

![Flow Diagram](image)
B. Module Descriptions:

The following modules are involved in this project
1) User Management
2) C3 Evaluation
3) Method Extraction
4) Corpus Creation
5) C3 Computation

1) User Management:
The administrator will maintain the application. Administrator has got the overall control of the application and its users. Administrator can add a new user, delete a user. Administrator can login into administrator’s page and can manipulate and modify the application.
- He/ she is responsible for and takes care of
  - Starting Server
  - Logging in - Entering the username and password.
  - Editing Personal information
  - Creating User Accounts
  - Deleting User Accounts
  - Calculating and analyzing C3 for classes or software systems.
  - Logging out

2) Normal User:
Normal users are ordinary users of the system who can use the system for calculating and analyzing cohesion of classes and software systems.
- He / she is responsible for
  - Logging in.
  - Modifying personal information.
  - Calculating and analyzing C3 for classes or software systems
  - Logging out.

3) C3 Evaluation:
In this paper information retrieval approach is adopted for measuring class cohesion. The philosophy behind this type of cohesion measurement is that a cohesive class is a crisp implementation of a problem or solution domain concept. In our project we make use of text coherence using LSI to measure class cohesion. This approach considers methods as elements of the source code that can be units of text for computing text coherence based on which the cohesion can be measured.

4) Method Extraction:
The first process in the calculation of C3 is the extraction of relevant methods from the class under consideration. Methods such as constructors, destructors, and assessors must be excluded since these methods may artificially increase or decrease the cohesion of a class. We extract only those methods from our class which will give us the most significant measure of class cohesion.

5) Corpus Creation:
This module deals with the extraction of text corpus for LSI indexing. The source code of each method we have extracted will be preprocessed and parsed to produce a text corpus. Comments and identifiers from each method are extracted and processed to produce the desired text corpus. In this process it is assumed that the developers used meaningful naming and commenting rules. Meaningful naming convention involves giving real world names to identifiers which are related to the domain concept implemented by a particular class. Thus a text corpus will be produced from each relevant method which we have extracted from the class under consideration. Here from each method we extract identifiers and comments. Identifiers are words and we can process them as such. Comments (starting with the symbol “/** & **/”) usually appear as sentences. So we extract each word from comments. For this we use String Tokenizer available in java.lang package. We can also use the split () method of the class String in Java. Thus at the end of this process we will be having an array of words extracted from each method in the class. Next, cutting all those extraneous words extracted from each method, leaving only content words likely to have semantic meaning. Here is one recipe for generating the list of all content words from a text.
- Make a complete list of all the words that appear anywhere in the text extracted from a particular method.
- Discard articles, prepositions, and conjunctions
- Discard common verbs (know, see, do, be)
- Discard pronouns
- Discard common adjectives (big, late, high)
- Discard frilly words (therefore, thus, however, albeit, etc.)
- Discard any words that appear in every method
- Discard any words that appear in only one method
Thus we will create text corpus for each method in the class. Finally we prepare a list of content words which contains content words from the entire text corpus.

6) C3 Computation:
This module is the stalwart of the project and it involves the calculation of class cohesion based on the LSI indexing of the text corpus we have produced. The first step involved in this process is to compute the similarities between each pair of methods under consideration. For this purpose, we create an equivalent semantic space by indexing the text corpus produced from each method. Thus the similarity between each pair of methods termed as CSM (Conceptual Similarity between Methods) is used to compute the Conceptual Cohesion of a class under consideration. The conceptual similarity between methods m and n CSM (m, n) is computed as the cosine between the vectors corresponding to m and n in the semantic space constructed by LSI indexing. The Average Conceptual Similarity of Methods in a class (ACSM) is defined as the average of CSM between all pairs of methods extracted from the class under consideration. If the ACSM of a class is less than zero, then the class is not at all cohesive. If the ACSM is greater than zero then cohesion of the class is equal to that value. The minimum cohesion of a class is zero and the maximum cohesion is one. The cohesion of an object oriented system will be the average of cohesion of all the class in the software system.

7) LSI Indexing:
Using our list of content words and methods, we can now generate a term-method matrix. Term Method Matrix is a large matrix with methods listed along the horizontal axis and the content words listed along the vertical axis. For each content word in our list, we go across the appropriate row and put a “1” in the column for any method where that word appears. If the word does not appear, we will put a “0” in that column.

Table - 3.1
Term Method Matrix

<table>
<thead>
<tr>
<th>METHODS</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>W2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>W3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can improve our result significantly by applying weighting for values in each non zero word/method pair. We use weighting because of two commonsense reasons. Content words that appear several times in a method are probably more meaningful than content words that appear just once. Infrequently used words are likely to be more interesting than common words. The first of these insights applies to individual methods, and we refer to it as local weighting. Words that appear multiple times in a method are given a greater local weight than words that appear once. We use a formula called logarithmic local weighting to generate our actual value.

Local Weighting Function: Term Frequency $l_{ij} = tf_{ij}$, the number of occurrences of term i in method j
Global Weighting Function: $g_i = gf_i / df_i$, where $gf_i$ is the total number of times term i occurs in the whole collection of methods, and $df_i$ is the number of methods in which term i occurs.

These two values multiplied together - local weight, and global weight, factor - determine the actual numerical value that appears in each non-zero position of our term/method matrix.

Finally we apply the SVD (Single Value Decomposition) algorithm on the term / method matrix. In order to apply the SVD algorithm we use JAMA (Java Matrix Package). The Matrix class in JAMA provides us with the facility for running 5 fundamental decompositions of which one is SVD.

At the end of running the SVD algorithm, we will get a matrix which will be the semantic space matrix that will be used for calculating CSM (Conceptual Similarity between Methods).

8) Calculation of CSM:
The conceptual similarity between two methods $m_k$ and $m_l$ that is CSM ($m_k, m_l$) is computed as the cosine between the vectors corresponding to $m_k$ and $m_l$ in the semantic space constructed by LSI

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

Where $vm_k$ and $vm_l$ are the vectors corresponding to $m_k$ and $m_l$ in the semantic space constructed by LSI.

9) Calculation of ACSM:
The ACSM (Average Conceptual Similarities between Methods) for a class C is calculated as
Calculation of $C_3$:

For any class $c$, the conceptual cohesion of the class $C_3(c)$ is calculated as

$$C_3(c) = \begin{cases} 
ACSM(c), & \text{if } ACSM(c) > 0, \\
0, & \text{else,}
\end{cases}$$

Thus if a class $c$ is cohesive, then $C_3(c)$ should be closer to 1.

### IV. Tools for System Implementation

#### A. J2EE:

The system is implemented in a J2EE application and is a web based system. The application is based on MVC application architecture and is conforming to the modern n-tier application pattern. The system front end is normal web browser (Internet explorer). This constitutes the “thin client” tier. The browser connects to a web server and the web server makes use of a Java application server to invoke the JSP components (the “web tier”). These JSP components realize presentation logic for the system. The JSP components in turn call Java Bean components. The Java Bean components are responsible for implementing the business logic of the application. This is the “business logic tier”. Finally the data persistence is done using an RDBMS system (MYSQL) comprising the “data tier”.

#### B. JAVA APIs:

A Java API is a collection of library routines which consists of set of classes and interfaces that performs predefined programming tasks. In Java APIs, classes and interfaces are packaged in packages. All these classes are written in java programming language and run on the JVM.

In our C CUBE we use three third party APIs. They are as follows

1) **QDox**:

QDox is a high speed, small footprint parser for extracting class/interface/method definitions from source file. In our project we have used QDox for extracting classes from a directory tree, extracting identifiers from a class, extracting methods from a class and extracting parameters from a method. Sample code is given below.

2) **JAMA**:

JAMA provides user-level classes for constructing and manipulating real, dense matrices. It is meant to provide sufficient functionality for routine problems, packaged in a way that is natural and understandable to non-experts. In our project we have used JAMA to generate the LSI space as a matrix and to calculate the singular value decomposition for the above matrix

3) **NetCDF**:

In this NetCDF used to calculate the Conceptual similarity between methods CSM as the sine of the angle between the vectors represented by the two methods. We have used the class MVector defined in NetCDF for the calculation of CSM.

### V. Empirical Results

![Fig. 2: Login Section](image-url)
VI. CONCLUSION

Object-oriented system classes in different programming languages contain identifiers and comments which reflect assessment of design quality, productivity, reuse effort, prediction of software quality, fault prediction, modularization of software, and identification of reusable components from the domain of the software system. This information can be used to measure the cohesion of software. This paper uses the conceptual cohesion of classes methodology which captures new and complementary dimensions of cohesion compared to a host of existing structural metrics. Conceptual cohesion is a highly advantageous, most relevant aspect of software cohesion which was concealed due to overwhelming usage of structural cohesion. Current industrial settings need a powerful software cohesion metrics that can save software developers as well as software testers bug identification time and debugging time. Most of the structural metrics used today will not incorporate an important aspect of object oriented concept- objects interrelate real world scenarios, which can be solved using Conceptual Cohesion of Classes (C3) metrics.

REFERENCES