Abstract -- While often defined in informal ways, software cohesion reflects important properties of modules in a software system. Cohesion measurement has been used for quality assessment, fault proneness prediction, software modularization, etc. 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. The paper proposes a new set of measures for the cohesion of individual classes within an software system, based on the analysis of the semantic information embedded in the source code, such as comments and identifiers. A case study on open source software is presented, which compares the new measures with an extensive set of existing metrics. The differences and similarities among the approaches and results are discussed and analyzed.

Keywords: Fault Prediction; Object Oriented Software; Conceptual Cohesion of Classes (C3).

1. Introduction

Software cohesion can be defined as a measure of the degree to which elements of a module belong together [1]. 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. In Object-Oriented (OO) software systems, cohesion is usually measured at class level and lately, many different OO cohesion metrics have been proposed, which try to capture different aspects of cohesion, or which reflect a particular interpretation of cohesion. As of today, there is no one cohesion metric that is accepted as a standard.

There is little research [2] on assessing the differences and similarities among these metrics. Software cohesion metrics can be used for different purposes including assessment of design quality [3, 4], prediction of software quality and fault proneness [5, 6], modularization of software [7, 8], identification of reusable of components [9, 10], etc. Most of the existing OO metrics suites are based primarily on the structural aspects of source code. These measures capture the degree to which the elements of a class belong together from a structural point of view, but give no clues whether the class is cohesive from a functional point of view. While other metrics were proposed by researchers to capture other aspects of cohesion, only few such metrics address the functional aspect of cohesion [11].

Proposals of measures and metrics for cohesion abound in the literature as software cohesion metrics proved to be useful in different tasks [12], including the assessment of design quality [13], [14], productivity, design, and reuse effort, prediction of software quality, fault prediction, modularization of software, and identification of reusable of components [15]. 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. We propose a new measure for class cohesion, named the Conceptual Cohesion of Classes (C3), which captures the conceptual aspects of class cohesion, as it measures how strongly the methods of a class relate to each other conceptually.

1.1. Class Cohesion

The components of a class are the instance variables and methods defined in the class plus those that are inherited. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through inst ante variable(s) if both methods use the instance variable(s). Class cohesion is defined in terms of the relative number of connected methods in the class.
1.2. Inheritance and Cohesion

A subclass inherits methods and instance variables from its super class. We have several options for evaluating cohesion of a subclass. We can (1) include all inherited components in the subclass in our evaluation, (2) include only methods and instance variables defined in the subclass, or (3) include inherited instance variables but not inherited methods. The class cohesion measures that we develop can be applied using any one of these options.

1.3. Measuring Object Oriented Reuse

We focus on private reuse within one software system [16]. We evaluate reuse from the server perspective, since this is the best orientation for evaluating reusability [17]. We are interested in two different forms of class reuse, reuse via instantiation and reuse via inheritance.

A class is reused by being instantiated in other classes or by being inherited to them. Instantiation reuse of a class is measured as the number of classes where the class is instantiated. Inheritance reuse of a class is the number of classes which inherit the class, i.e., the number of descendants (both direct and indirect descendants).

2. Related Work

There are several different approaches to measure cohesion in software systems. Based on the underlying mechanisms used to measure the cohesion of a class one can distinguish: structural metrics [1,2], the most popular class of cohesion metrics, semantic metrics [11], information entropy-based metrics [18], slice-based metrics [19], metrics based on data mining [20], and metrics for specific types of applications like knowledge-based [21] and distributed systems [22]. The value of these goals has not been validated empirically [23]. Rather, they have been justified on the basis of intuition. The amount of reuse of the number of times that a component is reused is an indicator of reusability. Of course, other factors such as the usefulness of a component are also components of reusability.

Cohesion refers to the “relatedness” of module components. A highly cohesive component is one with one basic function. It should be difficult to split a cohesive component. Cohesion can be classified using an ordinal scale that ranges from the least desirable category coincidental cohesion to the most desirable functional cohesion [24]. To apply this cohesion model to classes in object-oriented software, we need to add a new classification, data cohesion [10].

Bieman and Ott developed a set of functional cohesion measures based on program slices [25]. These measures apply only to individual functions; their application to entire classes is not obvious. Chidamber and Kemerer developed a Lack of Cohesion in Methods (LCOM) measure for object-oriented software [26]. LCOM is effective at identifying the most non-cohesive classes, but it is not effective at distinguishing between partially cohesive classes. LCOM indicates lack of cohesion only when, compared pair wise, fewer than half of the paired methods use the same instance variables.

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 [27, 28]. Other recent approaches have addressed class cohesion by considering the relationships between the attributes and methods of a class based on dependence analysis [29]. 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. 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.

3. An Information Retrieval Approach to Class Cohesion Measurement

OO analysis and design methods decompose the problem addressed by the software system development into classes in an attempt to control complexity. High cohesion for classes and low coupling among classes are design principles aimed at reducing the system complexity. The most desirable type of cohesion for a class is model cohesion such that the class implements a single semantically meaningful concept. This is the type of cohesion that we are trying to measure in our approach.

The source code of a software system contains unstructured and (semi)structured data. The structured data is destined primarily for the parsers, while the unstructured information (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.
3.1. An example of measuring C3

To better understand the C3 metric, consider a class \( c \in C \) with five methods \( m_1, m_2, m_3, m_4, m_5 \). The conceptual similarities between the methods in the class are shown in Table 1. For the computation of ACSM we consider all pairs of different methods, thus ACSM(c) = 0.5. Since the value is positive, \( C_3(c) = ACSM(c) = 0.5 \). This particular value for C3 does not indicate high cohesion for class \( c \) nor a low one, but the CSM values from Table 1 show that \( m_1 \) and \( m_3 \), \( m_2 \) and \( m_4 \), \( m_2 \) and \( m_5 \), and \( m_4 \) and \( m_5 \) are closely related respectively (i.e., the CSM between each pair is larger than C3) and this is shown in Fig 1. As one can see in this example, CSM is not a transitive measure. Since C3 is an average measure, we could have situations when some pairs of methods are highly related and other are not and the average is around 0.5. With that in mind, we refine the C3 to measure the influence of the difference between the highly related and unrelated pairs of methods on the cohesion of the class.

### Table 1 Conceptual similarities between the methods in class c

<table>
<thead>
<tr>
<th></th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>1</td>
<td>0.21</td>
<td>0.72</td>
<td>0.33</td>
<td>0.42</td>
</tr>
<tr>
<td>m2</td>
<td>1</td>
<td>0.28</td>
<td>0.91</td>
<td>0.66</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td>1</td>
<td>0.37</td>
<td>0.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4</td>
<td>1</td>
<td>0.89</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig 1** Similarities between methods in class c

3.2. An example of measuring LCSM

Consider the same class \( c \) described in section 3.3 with \( C_3(c) = 0.5 \). For each method of the class \( c \), we compute \( M_i \) based on definition 4: \( M_1 = \{ m_3 \}, M_2 = \{ m_4, m_5 \}, M_3 = \{ m_1 \}, M_4 = \{ m_2, m_5 \}, M_5 = \{ m_2, m_4 \} \). Table 1 shows us the intersection among all pairs of sets \( M_i \cap M_j \) in class \( c \). Based on the intersection \( P = \{(M_1, M_2); (M_1, M_3); (M_1, M_4); (M_1, M_5); (M_2, M_3); (M_2, M_4); (M_3, M_4); (M_3, M_5)\} \) and \(|P| = 7\). \( Q = \{(M_2, M_4); (M_2, M_5); (M_4, M_5)\} \) and \(|Q| = 3\). Thus, LCSM(c) = 7-3 = 4.
Table 2 Intersection results for method sets

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>M2</td>
<td>ø</td>
<td>ø</td>
<td>m5</td>
<td>m4</td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td></td>
<td></td>
<td></td>
<td>m4</td>
<td></td>
</tr>
<tr>
<td>M5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The two results combined indicate a lower value for the cohesion of class c from the example. In another situation, class c’ could have had more highly related methods than in this case (i.e., four pairs) and less unrelated method pairs with the same C3(c’) value (i.e., 0.5). Assume Table 2 would indicate 6 pairs of method sets with non empty intersection and only 4 with an empty intersection. The LCSM(c’) in that case would be 0. The combined measures will indicate that c’ is more cohesive than c.

4. System Architecture

The measuring methodology for the proposed cohesion metrics is described in Figure 1. The following steps are necessary to compute the C3 and LCSM metrics:

- Preprocessing and parsing of the source code 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.
- An IR method is used to index the corpus and create an equivalent semantic space.
- Based on the IR indexing conceptual similarities are computed between each pair of methods.
- Based on the conceptual similarity measures, C3 and LCSM are computed for each class.

We implemented a tool to compute C3 and LCSM for C++ software projects in MS Visual Studio .NET, based on the above methodology. Our source code parser component is based on the “Visual C++ Object Extensibility Model”. Using project information retrieved from Visual Studio .NET, the tool retrieves parts of source code that are used to produce a corpus. The extracted comments and identifier are processed by elimination of stop words and splitting identifiers that follow predefined coding standards. The corpus is indexed by the indexing engine, which is an implementation of LSI. We use the cosine between vectors in the LSI space to compute conceptual relations.

5. Implementation

We developed a tool IR-based Conceptual Cohesion Class Measurement, which supports this methodology and automatically computes C3 for any class in a given software system. The following steps are necessary to compute the C3 metric:

5.1. Corpus creation

The source code is preprocessed and parsed to produce a text corpus. Comments and identifiers from each method are extracted and processed. A document in the corpus is created for each method in every class.
5.2. 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.

5.3. Computing C3

Based on the conceptual similarity measures, C3 is computed for each class (definitions are presented in the next section). IRC3Mis implemented as an MS Visual Studio .NET addin and computes the C3 metric for C++ software projects in Visual Studio based on the above methodology. Our source code parser component is based on the Visual C++ Object Extensibility Model. Using project information retrieved from Visual Studio .NET, the tool retrieves parts of the source code that are used to produce a corpus. For software projects that are developed outside the .NET environment, that is, Mozilla from our case study, we use external parsers and a set of our own utilities to construct the corpus. The extracted comments and identifiers are processed in the elimination of stop words and splitting identifiers that follow predefined coding standards. We use the cosine between vectors in the LSI space to compute conceptual relations. A Java version of the tool is being developed as an Eclipse plug-in.

6. Results

First, we performed the univariate logistic regression. The R2 coefficient is defined as the proportion of the total variation in the dependant variable y (the fault proneness of a class) that is explained by the regression model. The bigger the value of R2, the larger the portion of the total variance in y that is explained by the regression model and the better the dependent variable y is explained by the explanatory variables.

In order to evaluate logistic regression models based on the studied metrics and their combinations, we utilize the following quantitative characteristics: precision, correctness, and completeness. We use these measures to be consistent with previously published results [30, 31]. Note that these characteristics of the results are somewhat different from the precision and recall measures used in IR.

Precision here is used to evaluate how well the model classifies faulty and nonfaulty classes. For example, C3 used as a separate explanatory variable in the univariate logistic model classified 1,267 (667 as nonfaulty + 600 as faulty) classes correctly out of 2,042 classes for Mozilla, that is, a precision of 62.05 percent (see Table 4). The results of the univariate logistic regression indicate that the model based on C3 is better than any other model except that of LCOM3.

Correctness is used to show what percentage of the faulty predicted classes is really faulty (computed as the number of classes observed and predicted as faulty divided by the total number of classes predicted as faulty). In the case of the univariate logistic regression model based on C3, the correctness is 61.35 percent since it suggested 978 classes as containing faults, but, in fact, only 600 of those have faults. Fig 3, 4 and 5 shows the C output file, C++ output file and JAVA output file respectively.
Fig 3 C OUTPUT File

```c
#include<stdio.h>
int main()
{
    int a,b,c;
    scanf("%d,%d,%d",&a,&b,&c);
    int sum=a+b+c;
    printf("Sum is %d",sum);
    return 0;
}
```

Fig 4 C++ OUTPUT File

```cpp
int main()
{
    int a,b,c;
    float l,m,n;
    cin >> l >> m >> n;
    cout << "Sum is " << l + m + n;
    return 0;
}
```

Cohesion not possible for C files.
7. Conclusions

Classes in object-oriented systems, written in different programming languages, contain identifiers and comments which reflect concepts from the domain of the software system. This information can be used to measure the cohesion of software. To extract this information for cohesion measurement, Latent Semantic Indexing can be used in a manner similar to measuring the coherence of natural language texts. Our results show that the classes that are heavily reused via inheritance exhibit lower cohesion. We expected to find that the most reused classes would be the most cohesive ones. Studies of additional software systems are needed to confirm these results.

Future Work

C3 does not take into account polymorphism and inheritance in its current form. It only considers methods of a class that are implemented or overloaded in the class. One way in which we can extend our work to address inheritance when building a corpus is to follow the approach in, where the source code of inherited methods is included into the documents of derived classes.

References


