Utilization of Method Graphs to Measure Cohesion in Object Oriented Software

Atakan Aral*, Tolga Ovatman†
Department of Computer Engineering
Istanbul Technical University
Istanbul,Turkey
*aralat@itu.edu.tr,†ovatman@itu.edu.tr

***Abstract*—The primarily used technique in measuring cohesion in an object oriented software is analyzing the relations between methods and data members of the software classes. However, extraction of such relations is rather problematic since the usage of data members can vary heavily in program code. Object code analysis is a widely used technique to solve this problem which requires the compilation of the source code. In this paper, we analyze the relations between various method graphs (including call graphs) which can be easily obtained by static source code analysis to obtain software-wide cohesion measures. We perform our analysis to discover the relationship between a widely used cohesion metric, namely LCOM, and the relation among method graphs. This way, we provide cohesion measures that can be extracted rapidly and easily compared to conventional techniques. Proposed measures can be used to obtain information about overall degree of cohesion which in turn can let the designer infer about the quality concerns like modularity and reusability of an object oriented software. Our results show that using different kind of relations in such analysis provides significantly different results in evaluating software-wide average class cohesion. Using graphs involving cooperating methods provide the most correlated results with the LCOM metric.**

***Keywords*-Object Oriented Metrics; Software Cohesion; Static Code Analysis**

I. Introduction

In software quality literature, the term cohesion is used to represent the harmony of certain software components in performing responsibilities of a software module. Cohesion is a rather vaguely defined term which can be used to qualitatively represent a diverse set of different properties (e.g. logical cohesion, functional cohesion) of object oriented software. This situation makes quantitative analysis of cohesion harder to perform; usually ordinal scales are proposed to measure cohesion [1]. Widely accepted cohesion metrics in the literature measure lack of cohesion for a particular class by examining common data usage of its methods.

From a practical perspective, if cohesion is measured using a conventional technique, parsing the program code to obtain methods and data members (variables) separately is unavoidable. The detected class members are then put through a dependency analysis and the quantification is performed by examining the usage relation between them. This examination involves a detailed process to detect variable usage since a variable can be used in a number of different indirect ways. To precisely detect such usage relations, object code analysis is often used. This kind of analysis requires the compilation of the code and makes the process cumbersome. It also prevents evaluating incomplete/uncompilable code during the development phase. Deriving a pure source code based measure which is also an indicator of conventional cohesion measures can be very useful.

In this study, we conduct graph based analysis of object oriented software to derive a measure that mimics overall characteristics of a version of the widely used Lack of Cohesion in Methods (LCOM) metric [2]. We have used an Abstract Syntax Tree (AST) based Java parser to extract four separate graphs depending on different method call types that emerge in object oriented software (e.g. method call graph, cooperating methods that are called together). Later, we have used graph clustering to group together related methods of the software. The clusters in each graph indicate the strongly related methods when the relations among methods are represented regarding a different perspective. Finally, we compared the clustering similarity value between each graph couple to spot a correlation between the obtained values and the average/ median LCOM value of the software's classes.

Our results show that the similarity measure between certain graph couples actually correlate to LCOM. The couples that include method call graph in common show stronger correlation to average LCOM where similarity values that indicate fragmentation among class methods are not correlated to LCOM at all.

As a result of our study, we present two main contributions: i) a software-wide cohesion measure that can be used to reason about the general cohesive quality of an object oriented software and ii) a cohesion measurement technique based purely on static analysis (such as source code parsing and graph clustering) which is highly correlated to a conventional cohesion metric (LCOM). Statical measurement of cohesion, likewise other cohesion metrics, can be used as an indicator for reusability, maintainability and understandability of software.

Numerous cohesion metrics were proposed in the literature including counting based measures as well as graph based measures. Counting based measures, and structural

metrics in particular, is the most studied class of metrics which aims to measure attribute referencing and sharing between the methods of a class [3]. The number of methods in a class that share attributes or call other methods of the same class is considered to be correlated to the cohesiveness of that class. Notable structural metrics are suggested in the studies including [2], [1], [4], [5], [6], [7], and [8]. These metrics mainly differ among themselves on how they define method relationships, represent the system or count.

Although most counting based measures also use graphs either for representation or for basic calculations, some other approaches particularly implement graph theory based algorithms. Proposed metrics in [9], [10] and [11], represent methods and attributes of a class by the vertices of a graph which still requires extraction of the attributes of the class. A recent method was suggested to detect candidate classes for refactoring, e.g., splitting [12]. They represent classes with weighted graphs with methods as the vertices and used Max-flow Min-cut algorithm to split the graph into its subgraphs. Our approach examines the relationship among graphs involving different kinds of relationships rather than focusing on a single graph.

All of the graph based metrics defined above use 'method call' relation and enrich it by adding attribute based properties to measure cohesion of a single class. These properties also appear in counting based metrics. Our intention is to eliminate the cumbersome attribute extraction process, so we basically enrich the method call graph using other types of method based graphs and provide a cohesion measure correlated to LCOM.

In the following two sections, we explain the method based graphs we have extracted and the clustering/similarity measurement techniques we have used correspondingly. In Section IV, we present our measurement results and in Section V, we discuss the results. We conclude the paper in the last section and present future work.

II. Extraction Process of Method Graphs

Relations among all the the attributes and methods (including attribute-attribute, attribute-method and method-method relations) in a software comes forward when reasoning about the concept of cohesion. However, for the case of static analysis, it is more reliable to use only method-method relations since the detection of attribute usage is much harder to perform in practice because of indirect variable usages. As mentioned earlier, to overcome this problem 'object code'/'byte code' analysis which requires code compilation, is frequently used. On the other hand, method-method relations, though having their own difficulties, are more straightforward to capture and more importantly very suitable to use in graph based analysis.

For capturing different kinds of relations between method calls that exist in an object oriented software, we first need to extract all the method calls from program code. We use an open source Abstract Syntax Tree (AST) based parser called *javaParser*[1] to collect the method calls from a group of Java projects. By traversing inside the static structure of object oriented software one can obtain many different relations among various programming elements (e.g. classes, attributes, methods, etc.).

We have extracted four different inter-method relations to be used in our analysis which we believe represent different properties in terms of coupling among methods. Our cohesion analysis is based on quantifying the difference between those different couplings. The rationale behind this measure comes from the viewpoint in which the lack of cohesion is defined as the amount of behavioral deviations among the software elements working together towards a common goal. In our analysis, we extract four different method graphs using the relations defined below and cluster those graphs to extract coupled groups of methods.

- R_{CM} : Cooperating Methods relation involves the method couples that were called together from another host method. The host can be any method inside the software and it is not included in the relation. However, all the possible method call couples inside the host is added to the relation set. The clusters inside the relation graph represent the method groups that were commonly called inside the whole software. It is important to realize that transitive couples are also included in the clusters. For instance, if methods m_a and m_b are coupled inside a host method and m_b and m_c are coupled inside a different host method it is possible that m_a,m_b and m_c will be in the same cluster.
- R_{MC} : Method Call relation is a very conventional relation which is frequently used in many different purposes. Hosts of relation R_{CM} is included to the graph of this relation. Regarding the case above in the first item, this situation decreases the possibility of m_a,m_b and m_c being grouped together during the clustering process.
- R_{IC} : Internal Call relation is actually a subset of Method Call relation since it only involves the method calls inside the same class. A cluster inside the graph of this relation is a subset of all methods inside a class. Clustering of this relation's graph actually represent the possible fragmentation of each class' behavior.
- R_{ML} : Method Layout relation involves a couple for each method that resides in the same class. Clusters in this relation's graph simply represent the classes of the software. The difference between the clusterings of R_{ML} and R_{IC} is a candidate to represent the cumulative lack of cohesion for a software's classes.

We use the graphs of four relations defined above (represented by G_R) and present our cohesion measure in more detail in latter sections.

[1] https://code.google.com/p/javaparser/

III. Clustering and Similarity Measurement

To obtain a software-wide cohesion measure, four graphs generated from the source code are clustered to observe related groups of methods. Then, the similarities among these clusterings are measured to provide a point of comparison for the suggested graphs. In this section, we present details of the algorithms, tools and measures used for graph clustering and cluster similarity measurement. In summary, Java based JUNG framework is used for graph based and clustering operations while a statistical environment called R is used for similarity measurement between different clusterings (adjusted Rand index computation).

A. JUNG Framework and Weak Component Clustering

JUNG (Java Universal Network/Graph) framework is an open source software library that allows analysis and visualization of graphs. It also includes implementations of common algorithms from graph theory along with other fields [13]. In our work, modeling capabilities of JUNG are used to represent the method graphs in a consistent and simple way. Software projects with few classes and methods are visualized with JUNG during the development phase in order to observe the effects of various experiments. However, the most notable assistance of JUNG to our analysis is through the graph clustering algorithms it provides.

Among the techniques considered for clustering method graphs, three algorithms that perform best in our case are Edge Betweenness Clustering, Voltage Clustering and Weak Component Clustering. Edge betweenness is defined as the number of shortest paths between all pairs of vertices in the graph that run along an edge. Since the edges between clusters have high edge betweenness, removing the edges with highest betweenness will split the graph into tightly connected subgraphs [14]. It was not possible to use this technique for clustering method graphs since it does not scale to graphs with thousands of vertices especially due to the requirement of recalculating all edge betweenness values after removal of an edge.

Voltage clustering, on the other hand, is a linear-time algorithm in which the graph is represented as an electric circuit. The assumption is that when voltage is applied to the different parts of the circuit (graph), voltage on a resistor (edge) states which cluster it belongs to [15]. A drawback of voltage clustering is the requirement of foreknown number of clusters. In our case, number of clusters may vary over a wide range depending on the size of the software project which restrain us from using the voltage clusterer.

Finally, in weak component clustering, a weak component is defined as a maximal subgraph in which all vertex pairs in it have a path between them. Aim of the algorithm is to extract such components from the graph as clusters. The algorithm is very simple and work well only when obvious boundaries among clusters exist. Its running time is $O(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Actually, four method graphs are quite appropriate for such an approach since they are quite sparse and nearly already clustered.

As a result, weak component clustering is chosen as the best choice for our analysis since it is scalable and does not require number of clusters beforehand. One should note that, the decision about the clustering algorithm does not noticeably effect our analysis results as long as the algorithm extracts decent sets of 'communities' from the method graphs.

B. R Environment and Adjusted Rand Index

R is an open source language and environment for statistical computing and graphics that can be easily extended via packages [16]. One of such packages is *mclust* which is used for normal mixture modeling via expectation maximization, model-based clustering, classification, and density estimation [17]. Although mclust is not used for clustering of the method graphs, we benefited from its implementation of adjusted Rand index.

Rand index is used to measure the similarity between two clusterings [18]. Let us assume that, for two clusterings C_1 and C_2, a represents the number of method pairs that are in the same class in both clusterings, b represents the number of method pairs that are in the same class in C_1 but not in C_2, c represents the number of method pairs that are in the same class in C_2 but not in C_1, and finally d represents the number of method pairs that are in different classes in both clusterings. Then, $a+d$ is the number of agreements and $b+c$ is the number of disagreements between C_1 and C_2. For n methods, the Rand index of two method clusterings is defined as follows.

$$r = \frac{a+d}{a+b+c+d} = \frac{s+d}{\binom{n}{2}} \tag{1}$$

This is the rate of agreements of two clusterings within all possible method pairs. If C_1 and C_2 match perfectly, Rand index is 1. However, Rand index is not equal to 0 or any other constant value for two random clusterings. To address this problem, a corrected for chance version of Rand index that has an expected value of 0 called adjusted Rand index is suggested [19]. It can yield a value between -1 and +1 and can be calculated as follows.

$$r_{adj} = \frac{\binom{n}{2}(a+d) - e}{\binom{n}{2}^2 - e} \tag{2}$$

where e is defined as:

$$e = (a+b)(a+c) + (c+d)(b+d) \tag{3}$$

An adjusted Rand index implementation in mclust library is used in our work with the purpose of comparing the clustering similarities of four method graphs explained in Section II. These similarities indicate the degree to which the same methods are conjunctive in both graphs.

Table I
RAND INDICES AND LCOM VALUES OBTAINED FOR 14 OPEN SOURCE SOFTWARE SYSTEMS

PROJECT	RAND INDICES						LCOM HS	
	G_{CM}–G_{MC}	G_{CM}–G_{IC}	G_{CM}–G_{ML}	G_{MC}–G_{IC}	G_{MC}–G_{ML}	G_{IC}–G_{ML}	Median	Average
borg	0.183	0.012	0.017	0.004	0.003	0.438	0.520	0.460
clojure	0.210	0.077	0.057	0.019	0.009	0.484	0.100	0.360
freemind	0.181	0.010	0.020	0.003	0.003	0.291	0.595	0.455
freeplane	0.234	0.006	0.009	0.002	0.002	0.360	0.395	0.400
jabref	0.191	0.005	0.011	0.001	0.002	0.325	0.340	0.376
javaParser	0.325	0.507	0.476	0.480	0.466	0.952	0.570	0.466
jedit	0.334	0.124	0.074	0.068	0.036	0.570	0.600	0.532
junit	0.057	0.018	0.021	0.002	0.004	0.212	0.000	0.309
logisim	0.171	0.004	0.005	0.001	0.001	0.294	0.480	0.410
pdfsam	0.374	0.007	0.045	0.005	0.011	0.110	0.500	0.429
StickyRoyale	0.414	0.364	0.088	0.675	0.234	0.278	0.875	0.612
tuxguitar	0.245	0.007	0.010	0.003	0.002	0.542	0.330	0.397
wurfl	0.109	0.049	0.055	0.007	0.010	0.249	0.000	0.156
ytd2	0.308	0.181	-0.033	0.513	-0.080	0.274	0.500	0.473

IV. CORRELATION ANALYSIS OF SIMILARITIES

One of the aims of this work is to observe how method graph similarities computed in Section III vary for different software distributions. With this purpose, we calculate correlation of Rand indices obtained for various open source software using Pearson's coefficient.

Another aim is to examine the correlation between the average cohesion per class and the method graph similarities. We implement widely used Lack of Cohesion in Methods (LCOM) metric originally suggested in [2]. Original LCOM is not normalized and does not guarantee that a class with LCOM = 0 is cohesive. Thus, a revised version of the original LCOM metric, LCOM HS [5] is used for our analysis. LCOM HS considers that cohesion is directly proportional to the number of data members in a class.

Different from method graph similarities, LCOM values are not extracted statically from compiled code. Another difference is that, we have an LCOM value for each class in the software. To be able to examine the correlation between class-level LCOM and software-level Rand indices, we use the average and median LCOM of a software. LCOM values of the inner classes are not taken into consideration in average and median calculations.

V. RESULTS AND DISCUSSION

This section provides the results of our analysis on a set of real-world software systems. We also discuss these results commenting on their meaning and possible causes.

A. Dataset

14 open-source Java projects are fetched from popular source code repositories GitHub[2] and SourceForge[3]. Most downloaded projects of nontrivial but manageable sizes are preferred. Number of methods contained by chosen software vary between 41 and 7175.

[2]https://github.com/

[3]http://sourceforge.net/

B. Similarity of Clusterings

Method graphs extracted from the source codes of the software projects are clustered as described in Section III. Table I demonstrates the Rand indices calculated between all possible pairs of these clusterings for each Java project. In addition, last 2 columns of the table are the median and average LCOM HS values for all classes within a project. Below, we explain the meaning of all columns of the table in order.

Column G_{CM}–G_{MC} represents the clustering similarity between the cooperating methods graph and the method call graph. A high value in this column represents that methods calling each other are usually called together from another method body as well. We obtained higher values for this pair than most other similarities. For instance, when we compare graph G_{CM} to the internal call graph G_{IC} instead of the complete one G_{MC}, similarity coefficients decrease dramatically.

Third column, G_{CM}–G_{ML}, is the similarity between the cooperating methods graph and the method layout graph. If more methods of a class are called from the same scope, this value should increase. Our results indicate this is not the case for the tested software projects.

Internal call graph is a subgraph of the complete call graph, so their similarity, column G_{MC}–G_{IC}, reflects rate of edges that are common to both graphs (i.e. internal method calls) to the number of the edges that are only in complete call graph (i.e. intra-class method calls). We observe values from the widest range (which is $[0.001, 0.675]$) for this type of similarity. Likewise, when method call graph is compared

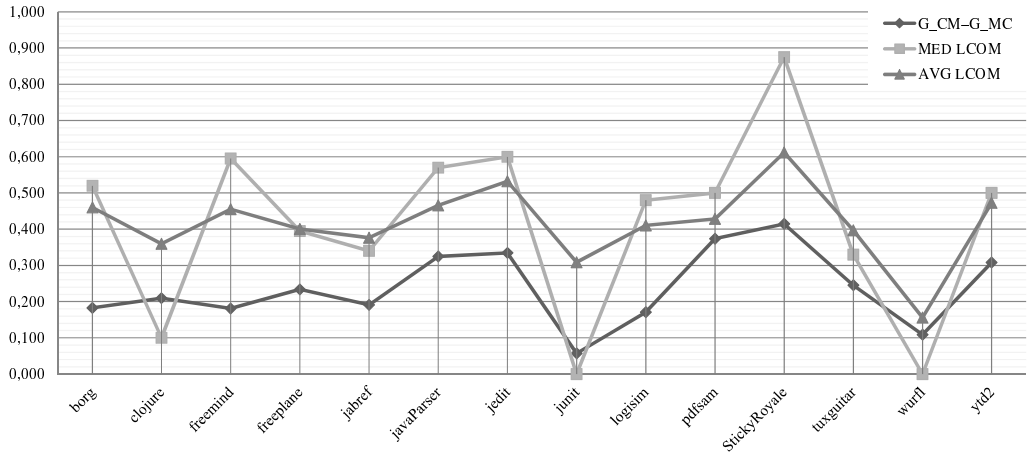

Figure 1. Average / Median LCOM HS Values and the Similarity Coefficient (Rand Index) of the Method Graphs G_{CM} and G_{MC}

with the method layout graph in column G_{MC}–G_{ML}, we obtained the lowest similarities. This may be interpreted as that, there are enough intra-class method calls creating different clusters than the class layouts.

Last similarity column, G_{IC}–G_{ML} may be considered as a type of method fragmentation within classes since it is the similarity between the internal call graph and the method layout graph. If the internal call graph for a single class contains multiple connected components, then they will be clustered separately and clustering similarity will reduce. As expected, this is the column with the highest Rand index values since the graph G_{IC} is merely a subgraph of G_{ML}.

Finally, last two columns contain the average and median LCOM HS values of all classes in each project. It should be noted that, lower values in these columns indicate more cohesive classes.

C. Correlation of Similarities

With the aim of observing the relationships among Rand indices and LCOM values, we computed correlation coefficients over different projects for each pair as described in section IV. Table II provides these pairwise coefficients where correlation values greater than 0.5 are emphasized.

Table II
CORRELATION COEFFICIENTS BETWEEN RAND INDICES AND LCOM

	$\frac{G_{CM}}{G_{MC}}$	$\frac{G_{CM}}{G_{IC}}$	$\frac{G_{CM}}{G_{ML}}$	$\frac{G_{MC}}{G_{IC}}$	$\frac{G_{MC}}{G_{ML}}$	$\frac{G_{IC}}{G_{ML}}$
G_{CM}–G_{IC}	**0.585**					
G_{CM}–G_{ML}	0.320	**0.808**				
G_{MC}–G_{IC}	**0.644**	**0.876**	0.445			
G_{MC}–G_{ML}	0.430	**0.882**	**0.938**	**0.602**		
G_{IC}–G_{ML}	0.225	**0.593**	**0.778**	0.245	**0.677**	
MED LCOM	**0.767**	0.474	0.211	**0.572**	0.393	0.168
AVG LCOM	**0.770**	0.476	0.168	**0.580**	0.393	0.233

According to the results, G_{CM}–G_{MC} is the most correlated pair to LCOM metric with a remarkable coefficient around 0.77. This relation points that software projects with highly cohesive classes do not contain many method pairs that both call each other and commonly called. We can claim that, if two methods which call each other, are also called sequentially in the body of another method, overall cohesion of the software decreases. It is possible to measure cohesion quite accurately using the similarity between the method call graph and the commonly called methods graph. This outcome makes sense since calling two methods that call each other is an example of unstructured code and it is expected to encounter low cohesion for such code. Fig. 1 demonstrates the change of the similarity G_{CM}–G_{MC} together with average and median LCOM HS over 14 software projects.

Fig. 2 visualizes correlations in Table II with a weighted, undirected graph where the vertices are the LCOM values and Rand indices, and the edges are correlations between them. Only the correlations with coefficients greater than 0.5 are taken into consideration. Line thickness of an edge is proportional to the correlation coefficient between the vertices it is incident to. Two clusters are visible in this graph representation: i) G_{CM}–G_{MC} and LCOM, ii) all the other

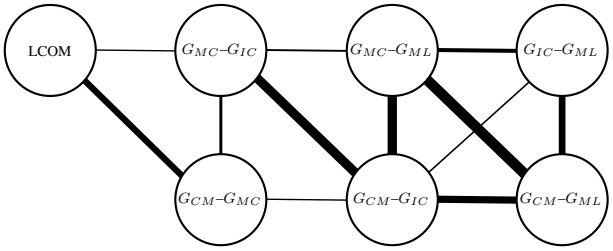

Figure 2. Graphical Representation of the Correlations

graph couples. It is clear that, all the Rand indices are highly correlated inter se except the G_{CM}–G_{MC}. We suspect that, these represent some software property or measure other than cohesion.

Though not as strong as the first one, another interesting correlation is between G_{MC}–G_{IC} and LCOM. As explained before, high values G_{MC}–G_{IC} similarity indicates high number of within class method calls in comparison to the intra-class ones. This correlation may be interpreted as that, method calls are rather between different classes in cohesive software projects. Method fragmentation, on the other hand, does not seem to be significantly related with cohesion. Correlation coefficient between G_{IC}–G_{ML} and LCOM is around 0.39.

VI. Conclusions and Future Work

In this paper, we proposed a novel technique which allows to measure software-wide cohesion. Our technique can be applied by using static source code analysis which eliminates the extra effort spent to discover the relationships between class attributes and other members. We have analyzed four different method-to-method relations' graph clusterings and detected the most suitable couple as a measure for software-wide cohesion. Our results show that measuring the difference of groupings formed by method call relation and cooperating method relation produces the most correlated results to the conventional LCOM metric.

As a future work, we plan to extend our studies by adding weights to the graphs that were used during our analysis. Moreover, we plan to study on mathematical basis behind the discovered relation between LCOM and the proposed measure by using relations and graph theory. Another interesting direction might be to study on the quality attributes represented by the non-correlating relation couples that were analyzed in our study. Results in the previous section show that the graphs except G_{CM}–G_{MC} couple pose a high correlation amongst them, so it is highly possible that they are related with a certain quality attribute of object oriented software.

References

[1] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," in *Proceedings of Symposium on Software reusability*. ACM, 1995, pp. 259–262.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[3] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 287 –300, 2008.

[4] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring the coupling and cohesion of an object-oriented program based on information flow," in *Proceedings of International Conference on Software Quality*, 1995, pp. 81–90.

[5] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[6] L. Briand, J. Daly, and J. Wst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, pp. 65–117, 1998.

[7] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang, "A novel approach to measuring class cohesion based on dependence analysis," in *Proceedings of International Conference on Software Maintenance*, 2002, pp. 377 – 384.

[8] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 2, pp. 123–149, 2006.

[9] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of International Symposium on Applied Corporate Computing*, 1995.

[10] X. Yang, "Research on class cohesion measures," Ph.D. dissertation, MS Thesis, Department of Computer Science and Engineering, Southeast University, 2002.

[11] J. A. Dallal, "Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics," *Information and Software Technology*, vol. 54, no. 4, pp. 396 – 416, 2012.

[12] G. Bavota, A. D. Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397 – 414, 2011.

[13] J. O'Madadhain, D. Fisher, S. White, and Y. Boey, "Jung: Java universal network/graph framework," *University of California, Irvine, California*, 2003.

[14] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[15] F. Wu and B. Huberman, "Finding communities in linear time: a physics approach," *The European Physical Journal B - Condensed Matter and Complex Systems*, vol. 38, no. 2, pp. 331–338, 2004.

[16] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2012. [Online]. Available: http://www.R-project.org/

[17] C. Fraley, A. E. Raftery, T. B. Murphy, and L. Scrucca, "mclust version 4 for r: Normal mixture modeling for model-based clustering, classification, and density estimation," Department of Statistics, University of Washington, Tech. Rep., 2012.

[18] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

[19] L. Hubert and P. Arabie, "Comparing partitions," *Journal of classification*, vol. 2, no. 1, pp. 193–218, 1985.