Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics

Safwat M. Ibrahim[1], Sameh A. Salem[1], Manal A. Ismail[1], and Mohamed Eladawy[1]

[1] Department of Electronics, Communications & Computers, Faculty of Engineering, Helwan University. Helwan, Cairo, Egypt

Abstract

The production of well-developed software reduces the cost of the software maintainability. Therefore, many software metrics have been developed to measure the quality of the software design. Measuring class cohesion is considered as one of the most important software quality measurements. Unfortunately, most of approaches that have been proposed on cohesion metrics do not consider the inherited attributes and methods in measuring class cohesion. This paper provides a novel assessment criterion for measuring the quality of a software design. In this context, inherited attributes and methods are considered in the assessment. This offers a guideline for choosing the proper Depth of Inheritance Tree (DIT) that refers to the nominated classes for refactoring. Experiments are carried out on more than 35K classes from more than 16 open-source projects using the most used cohesion metrics.

Keywords: *Class Cohesion, Metrics, Quality, Software Measurement, Refactoring, Inheritance.*

1. Introduction

Class cohesion is defined as the degree of the relatedness of the members in the class [2], [3]. Various metrics were developed to measure the similarity between the class elements. Many cohesion measurements are based on the Low-Level Design (*LLD*) information. *LLD* class cohesion metrics require to analyze the algorithms used in the class methods or the code itself (if available) in order to measure the class cohesion [1], [2]. Another approach for class cohesion measurement is based on the High-Level Design (*HLD*) information. *HLD* class cohesion metrics rely on information related to class and method interfaces [2]. Many of the proposed *LLD* cohesion metrics focus on measuring the correlation between pairs of methods in the class. Such as Chidamber and Kemerer Lack of COhesion in Methods (*LCOM1* and *LCOM2*) metrics [6], [7], Bieman and Kang Tight Class Cohesion (*TCC*) and Loose Class Cohesion (*LCC*) metrics [4], Badri et al. Lack of Cohesion in the Class-Direct (LCC_D) and Lack of Cohesion in the Class-Indirect (LCC_I) metrics [3], and Bonja and Kidanmariam Class Cohesion (*CC*) metric [5]. Generally, a pair of methods is correlated if a common attribute is used (either directly or indirectly) or via method invocation.

Alternatively, Henderson-Sellers [8] proposed *LCOM3* metric as a different approach for measuring the class cohesion by measuring the attribute-method correlation. Because the *LCOM3* metric has a drawback as it is not normalized into ranging between 0 and 1, Braind et al. [6] proposed the *Coh* metric by enhancing the *LCOM3* metric to be normalized.

Bieman and Kang [4] introduced the concept of tight and loose class cohesion; Badri et al. [3] enhanced both *TCC* and *LCC* metrics by including the methods invocation in measuring the cohesion value.

For the analysis of the class cohesion, Braind et al. [6] defined two options concerning the inherited attributes and methods:

1. Exclude the inherited attributes and methods form the analysis.
2. Include the inherited attributes and methods in the analysis.

If the inherited attributes and methods are excluded, this approach analyzes to what degree this extension represents a single semantic concept [6]. If the inherited attributes and methods are included, this approach analyzes whether the class as a whole still representing a single semantic concept [6]. The including and excluding of the inherited elements (attributes and methods) are two different aspects and both should be considered [6].

However, most of the developed cohesion metrics tools measure only cohesion on the implemented elements (attributes and methods) and do not consider the inherited elements [25], but as a design inspector, there is a necessity to study the overall class cohesion including all inherited elements. Therefore, there is a need for design quality measures that are able to examine the class cohesion with/without the inherited elements. In this paper, a Cohesion Measure Tool (CMT) is proposed to enable a

software inspector to either include or exclude the inherited elements in the assessment. In addition, it recommends the value of Depth of Inheritance Tree (DIT) which contain the most promising classes for Refactoring.

This paper is structured as follows. Section 2 describes the effect of including inherited elements in measuring the class cohesion. Section 3 illustrates the measurement process. Section 4 describes the different selected projects. Section 5 provides the experimental results and discussion. Finally, Section 6 draws conclusion.

2. Class Cohesion and Inheritance

This section analyzes the effect of including the inherited elements (attributes and methods) in the measurement of class cohesion.

2.1 Effect of Inheritance on Class Cohesion:

By including the inherited attributes and methods, the class cohesion can be increased or decreased depending on the design of the class and its parent classes. This section illustrates several cases that address the increase of the class cohesion when including the inherited elements. For example, assume class B has n2 implemented elements, and n1 inherited elements from class A, so the connections among all elements in class B could be analyzed as the connections between all nodes in a graph contains n1+n2 nodes. The connections in this graph could be classified into three categories.

- **Implemented** elements internal connections:

This category contains all connections between implemented elements.

- **Inherited** elements internal connections:

This category contains all connections between inherited elements.

- **Cross-connections** between implemented and inherited elements:

This category contains all connections between inherited and implemented elements.

Therefore, the increase of the cohesion for a child class when considering the inherited elements could be as a result of one of the above categories. The next sections discuss each category in details.

Category 1: Implemented elements internal connections

In this case, the cohesion value for the implemented elements increase as shown in Figure 1, This occurs when a class contains elements that are connected through elements defined in the parent class.

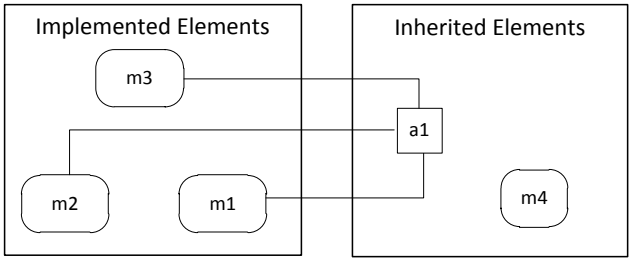

Fig. 1 Implemented elements internal connections

Figure 1 shows a class that has three implemented methods m1, m2, m3 and two inherited elements a1 and m4. As shown, the class methods (m1, m2, m3) are connected via common attribute a1. Therefore by excluding inherited elements, the cohesion value of the class will be 0, while including inherited elements, the class cohesion increases.

Category 2: Inherited elements internal connections

In this case, the cohesion increases because of the high connectivity between the inherited elements. This occurs when a class has a low cohesion value and inherits elements from a very cohesive class.

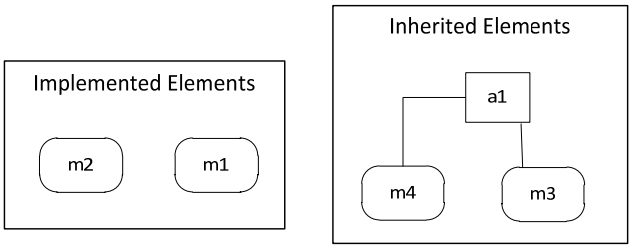

Fig. 2 Inherited elements internal connections

Figure 2 shows a class that has two implemented methods m1, m2 and three inherited elements a1, m3, and m4. As shown, the class methods (m1 and m2) are not connected, while the inherited elements are directly connected. Therefore by excluding inherited elements, the cohesion value of the class will be 0, while including inherited elements, the class cohesion increases.

Category 3: Cross-connections between implemented and inherited elements

In this case, the cohesion increases because of the connection between the inherited elements and the implemented elements, and this occurs in many scenarios:

1) Methods in the child class are used in the parent class and that happens only when the child class overrides some methods.

2) Most of the inherited elements are used in the implemented elements, and generally the connection

between the elements in the class has two types, tight connection and loose connection [4].

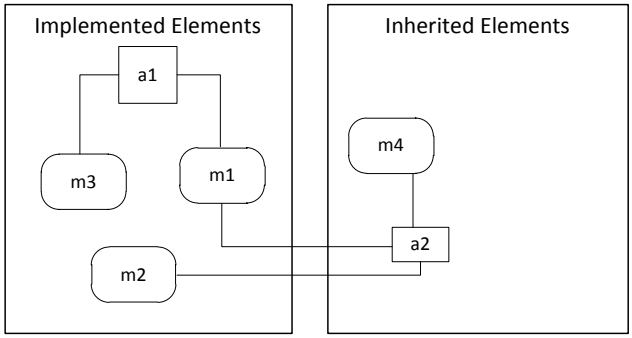

Fig. 3 Cross-connections between implemented and inherited elements

Figure 3 shows a child class that has four implemented elements a1, m1, m2, m3 and two inherited elements a2, m4. As shown, the class methods (m1, m2, m3) are connected via common implemented attribute a1 and inherited elements a2 and m4. Therefore by excluding inherited elements, the cohesion value of the class will be based on the connection between m1 and m3, while it increases when including inherited elements a2 and m4.

For example, the tight class cohesion value for the child class (shown in Figure 3) in case of including inheritance (like TCC metric) is 4/6=0.67 and the loose class cohesion value (like LCC metric) is 6/6=1.0 as pairs of methods m2, m3 and m3, m4 are loosely connected to each other.

It should be noticed that the third category is **the most descriptive category** than other two categories, for example assume class B has 5 inherited methods, and 5 implemented methods, so the internal connections (ether for implemented or inherited elements) need (4*5/2) links =10 links, but the cross-connections between inherited and implemented elements need (9*10/2)-(10+10) = 25 links.

2.2 Cohesion Measure Tool (CMT)

In this subsection, a cohesion Measure Tool (CMT) is proposed to examine the quality of object oriented software design. This measure is based on a well known cohesion metrics [13]. In this context, the CMT assesses and computes the different cohesion metrics for Java open-source code by processing the compiled code. Additionally, the CMT enables a design inspector to customize and configure the setting for each cohesion metric, as follows:

- The design inspector can choose either to exclude/include the access methods and constructors from the analysis.
- The design inspector determines either to exclude/include the inherited attributes and methods.

The CMT is developed in Java using the ASM 3.2 framework [9]. Figure 4 shows a general overview for the proposed CMT.

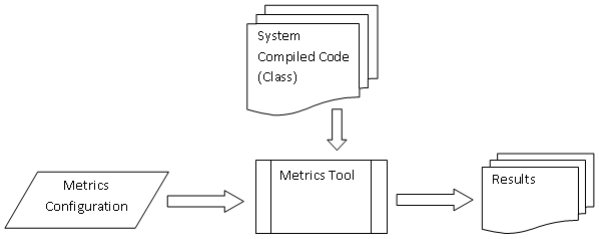

Fig. 4 Overview of the CMT

2.2.1 Configuration Details

Some researchers faced problems with some kinds of special methods. Briand et al [6] mentioned that the access methods artificially reduce the cohesion level, and it was suggested to exclude the access methods for resolving these problems [6], [14]. Bieman et al. [4] also recommended to exclude constructors to remove the impact of artificial connection by those methods [14].
Jehad Al Dallal [15] also illustrated empirically the effect of excluding special methods (like constructors and access methods) in improving the cohesion measurements. Therefore, in measuring a class cohesion, the CMT is configured to exclude the special methods to eliminate their artificial effect on cohesion measurements. Additionally, abstract classes and interfaces are also excluded.
The static methods and fields are generally quite different than instance methods and fields (in particular, static fields are often used for constants)[13], Barker et al. [13] excluded all static methods and attributes in measuring class cohesion. Therefore, the proposed CMT is configured to exclude all static methods and fields.
Badri et al. [3] recommended to exclude classes with fewer than two public methods as TCC, LCC, LCC_I, and LCC_D
[3], [4] metrics provide undefined values in these cases.

2.2.2 Overloaded and Overridden Methods

Barker et al. [13] excluded all methods with same signature and same number of parameters as these methods caused difficulties in implementation, Badri et al. [3] unified all methods with same name. The proposed

CMT adopted the same approach by unifying all overloaded methods.
The overridden methods appear if the design inspector configured the CMT to include the parent attributes and methods. Then the tool recursively loads all parent classes, and starts in the reverse direction (from the top parent class and then the children classes) when the child class has a method with the same name and signature as the declared one in the parent class. In this case the new method overrides the old existing method, so the tool replaces the old method code from the list of declared methods in the class with the new defined method in the child class.

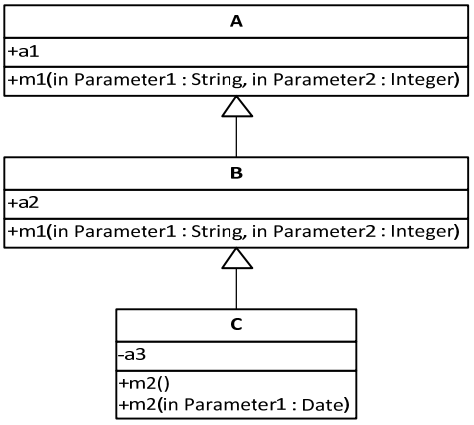

Fig. 5 Example of a simple class hierarchy

Figure 5 shows an example of a class hierarchy. It should be noted that the proposed CMT can be configured to include the inherited elements. In this case, Class C will be analyzed as follows:

- Class C contains public attributes a1 and a2 inherited from parent classes.
- Private attribute a3 declared in the class C.
- Public method m1 with code declared in class B (overridden method).
- Public Method m2 that unifies both overridden methods m2() and m2(in Parameter1: Date) that are declared in class C.

2.3 Practical Code Example

In this subsection, the effect of including the inherited attributes and method in measuring class cohesion is illustrated by analyzing a sample CategoryImmediateEditor class from the Log4j [24] project source code.

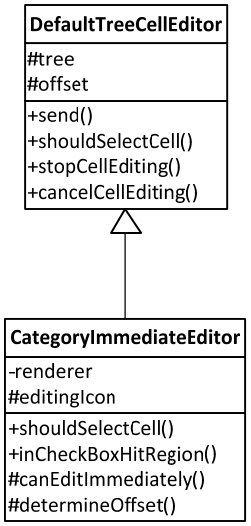

Fig. 6 CategoryImmediateEditor class diagram

As illustrated in Figure 6, The class CategoryImmediateEditor has two implemented public methods, these methods access the inherited attribute 'tree', so by excluding the inherited elements the cohesion value is 0 (lack of cohesion is 1) and by including the inherited elements, the cohesion value increased. Additionally the inherited elements (stopCellEditing, cancelCellEditing, tree ..) are connected to each other, and connected to the implemented methods via calling both 'tree' attribute and the overridden method ' shouldSelectCell'.

Table 1: Results for CategoryImmediateEditor class

Case	LCCI	LCCD	LCC	TCC	CC	Coh	LCOM3
Excluding Inheritance	1	1	1	1	1	0.88	1.167
Including Inheritance	0.00	0.36	0.00	0.36	0.87	0.87	0.92
Difference	-1.0	-0.64	-1.0	-0.64	-0.13	-0.01	-0.25

Some of the cohesion metrics are defined as cohesion value and other metrics are defined as **Lack of Cohesion** (LOC) value. Thus, in order to simplify the comparison among metrics, all metrics are measured as LOC (LOC = 1 - the measured cohesion value for the class). Table 1 illustrates the results of applying different cohesion metrics in both cases of including/ excluding of inherited elements. It could be noticed from the results that the cohesion value increased (lack of cohesion value decreased) by including of the inherited elements.

3. Measurement Process

This section describes the proposed steps to measure the effect of including the inherited elements. Additionally, the different patterns for the LOC difference are illustrated.

3.1 Experimental Procedures

In order to study the relation between Lack Of Cohesion (LOC) and Depth of Inheritance Tree (DIT) [7] for the measured classes, the CMT is applied on the different projects in the following steps:

Step 1: CMT is configured to **include** the inherited elements, then the class lack of cohesion is measured for all classes in open-source projects.

Step 2: CMT is configured to **exclude** the inherited elements, then the class lack of cohesion is measured for all classes in open-source projects.

Step3: Select classes that have defined values in both cases of including and excluding of the inherited elements. Classes with fewer than two public methods are excluded as TCC, TCC, LCC, LCC_D, and LCC_I metrics provide undefined values for these cases[3]. Additionally, classes with attributes few than one attribute are excluded as LCOM3 metric provides undefined value in this case [6].

Step4: Determine the DIT threshold. Consequently, all classes with DIT higher than the DIT threshold will be excluded.
The percentage of classes with certain DIT to the total number of classes in the project (The DIT distribution) varies from project to another. The number of classes with certain DIT in some cases are relatively very small count which may lead to inaccurate conclusion. Thus, in order to enhance the analysis of the results, the following factors are considered to determine the range of DIT that will be included:

- The maximum DIT in the project.
- The distribution of the number of classes with certain DIT.
- The total number of classes in the project.

By applying the mentioned criteria, there will be a number of excluded classes whenever the classes that have certain DIT is less than 2% of the project size.

Step5: Calculate the difference between lack of cohesion value in case of including inherited elements and in case of excluding inherited elements.

Step6: Measure the average value for the calculated difference grouped by DIT.

3.2 LOC Difference

The difference between the Lack of Cohesion (LOC) value in case of including inherited elements and in case of excluding inherited elements is calculated in the following Eq.(1):

$$LOC_{Diff(DIT)} = Avg_{Inc(DIT)} - Avg_{Exc(DIT)} \quad (1)$$

Where:
$Avg_{Inc(DIT)}$ is the average LOC measured for all classes with same DIT in case of ***including*** the inherited attributes and methods.
$Avg_{Exc(DIT)}$ is the average LOC measured for all classes with same DIT in case of ***excluding*** the inherited attributes and methods.

The $LOC_{Diff(DIT)}$ can be classified into various patterns as follows:

Pattern1: In this pattern the LOC_{Diff} is a positive value in almost all DIT range (as shown on Figure 7) which means the LOC value in case of including the inherited elements is greater than the value in the case of excluding inherited elements. (i.e., the measured cohesion value is reduced by including the inherited elements). This can be motivated as follows, either the metrics couldn't measure the reusability in the inherited classes, or the inherited elements are not well used. Consequently the cohesion value is reduced by inheritance.

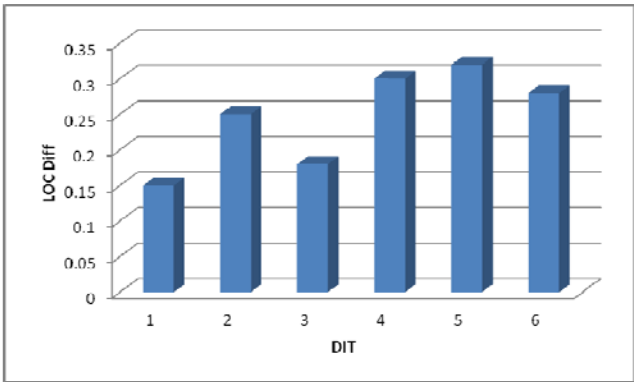

Fig. 7 Pattern1 the LOC difference is a positive value in almost all DIT range

Pattern2: In this pattern, the LOC_{Diff} decreases by DIT increase (as shown on Figure 8) which means the inherited attributes and methods are highly reused in the children classes. Therefore, on the basis of cohesion this project is

well designed as all the inherited elements are highly reused in the children classes.

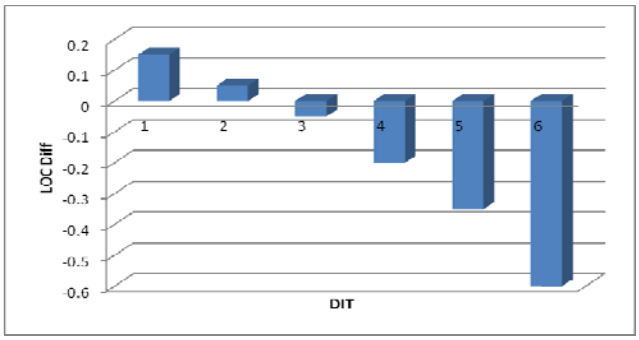

Fig. 8 Pattern2 the LOC difference decreases by DIT increase

Pattern3: In this pattern, the LOC_{Diff} is a negative value at lower DIT and step up to a positive value at DIT greater than certain threshold (as shown on Figure 9). Therefore, it should be noted that the DIT of these projects should not be increased above the threshold, or refactor the classes to higher DIT than this threshold.

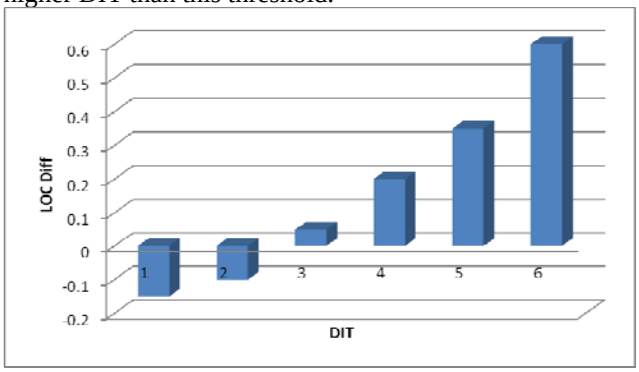

Fig. 9 Pattern3 the LOC difference steps to positive at DIT greater than certain threshold

4. Case Studies

In order to study the effect of measuring the inherited attributes and methods, CMT is applied on different Java open-source projects. These projects have been selected due to the following criteria:

- Variation in vendors (selected projects are developed in different organizations with different organizations scales).
- Variation in project categories (selected projects are distributed in different domains like games, tools, application server, development, graphics, and communications).
- Variation in scale and size (some projects are in range of a few hundred of classes, other projects are in a scale of thousands of classes).

The proposed CMT has been applied on more than 16 open-source projects, which contain more than 35K classes. Table 2 illustrates the general overview for some of the projects, including their version, category, and number of classes.

5. Results and Discussion

This section describes the experimental results obtained by applying the mentioned steps in section 3.1. Additionally, the different patterns for the LOC difference are illustrated.

5.1 Including the Inherited Attributes and Methods

In this step the CMT is configured to included the inherited attributes and methods. As mentioned in section 2.3, some of the cohesion metrics are defined

Table 2: Projects Overview

Project	***Version***	Vendor/ Author	Category	Number of Classes	Number of Classes After Excluding Interfaces and Abstract Classes
FreeMind [10]	0.9.0	Several people contributed in the development	Business & enterprise application	424	354
Azureus / Vuze [11]	4702	Vuze	Communications application	4702	3253
Jboss Application Server [12]	7.0.1	Red Hat	Application server	2707	2254
JDK [16]	1.7.0	Oracle	Software development kit	13278	9617
JSF [17]	2.1.6	Glassfish	Web application framework	918	658
JUnit [18]	4.1	Kent Beck	Testing tool	162	107
Log4j[24]	1.2.16	Apache Software Foundation	Logging tool	221	188
Google Web Toolkit [20]	2.4.0	Google	Ajax framework	4217	3120

Table 3: Results in case of Including Inherited Elements

Project	*Measure*	LCC_I	LCC_D	LCC	TCC	CC	Coh	LCOM3
JDK [16]	Mean	0.360	0.495	0.533	0.623	0.796	0.706	0.785
	Standard Deviation	0.337	0.347	0.323	0.308	0.240	0.245	0.258
Log4j [24]	Mean	0.459	0.667	0.585	0.709	0.856	0.768	0.872
	Standard Deviation	0.361	0.303	0.329	0.297	0.196	0.218	0.225
JSF [17]	Mean	0.482	0.612	0.653	0.701	0.802	0.723	0.819
	Standard Deviation	0.378	0.329	0.351	0.339	0.248	0.252	0.247
GWT [20]	Mean	0.558	0.626	0.697	0.726	0.843	0.782	0.893
	Standard Deviation	0.348	0.323	0.359	0.343	0.245	0.244	0.273
Jruby [22]	Mean	0.385	0.541	0.513	0.607	0.773	0.739	0.819
	Standard Deviation	0.361	0.324	0.360	0.327	0.263	0.260	0.272

as **Lack of Cohesion** (LOC) value. Thus, for the purpose of simplicity, all metrics are measured as LOC (LOC = 1 - the measured cohesion value for the class). Table 3 summarizes some of the obtained results by applying the CMT on different open-source projects.

It can be analyzed from the obtained results that:
The average values for the Method-Method class lack of cohesion metrics (LCC,TCC, LCC_I, and LCC_D) are lower than the average values for the Attribute-Method class lack of cohesion metrics (Coh, and LCOM3). Additionally, the standard deviation for these Method-Method metrics (LCC,TCC, LCC_I, and LCC_D) are higher than the standard deviation for the Attribute-Method metrics (Coh, and LCOM3) which reflects the difference in metrics design where the variation in Method-Method metrics measures (according to different classes with various designs) are greater than the corresponding Attribute-Method metrics measures.

5.2 LOC Difference Results

The second step (as mentioned in section 3.1) the CMT is configured to exclude the inherited attributes and methods, and lack of cohesion is measured for all the studied open-source projects. Then, the difference between the average LOC value in both cases of including and excluding inherited elements is calculated. Figures 10 to 14 illustrate the results for some projects.

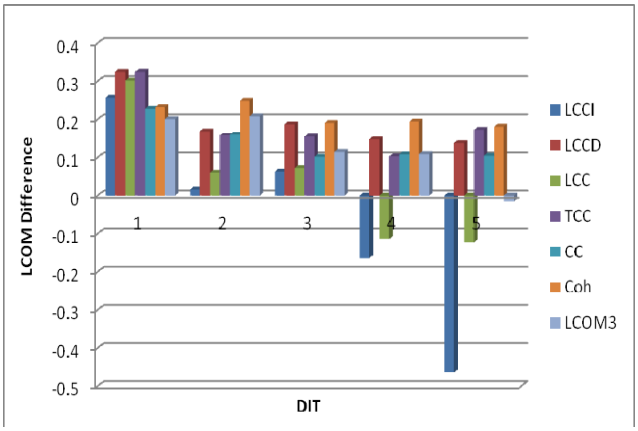

Fig. 10 Results for FreeMind project

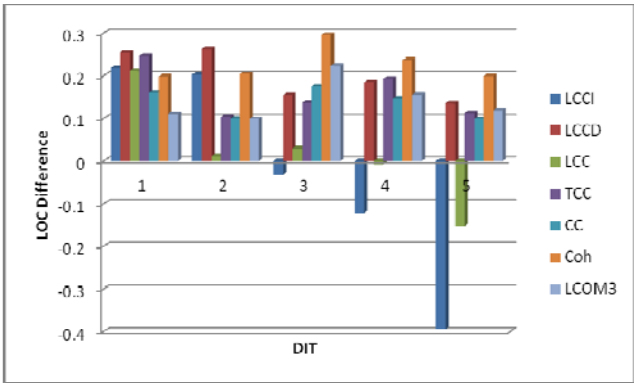

Fig. 11 Results for JHot Draw project

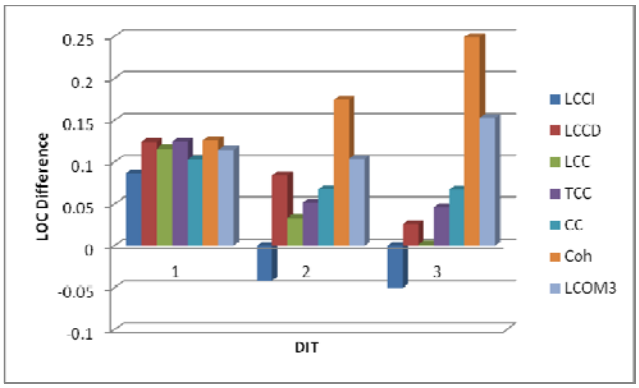

Fig. 12 Results for JSF project

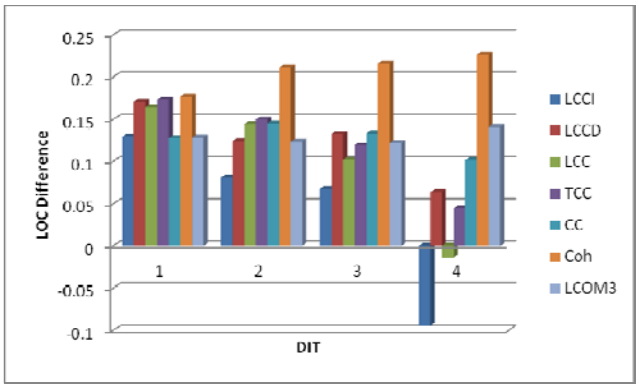

Fig. 13 Results for GWT project

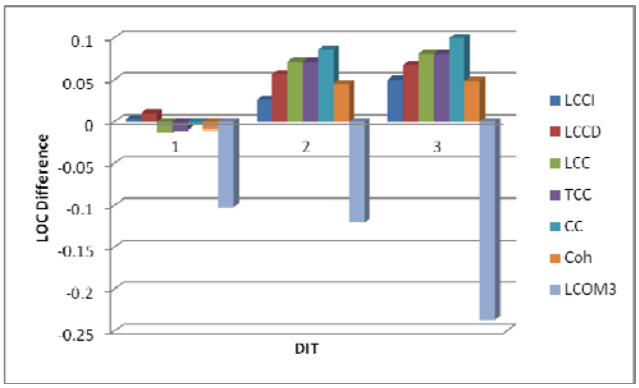

Fig. 14 Results for J2EE project

It could be noticed from Figures 10 to 14 that some of the obtained results (projects FreeMind [10], Google Web Toolkit (GWT) [20], JHot Draw [21], JRuby [22], and JSF [17]) are belong to pattern1 (the Difference is positive for all DIT range) and pattern2 (the difference decreases by DIT increase). This could be explained by studying the effect of the three categories that increase the class cohesion. As illustrated in section 2.3 the most effective category that has higher impacts on the cohesion value (in case of including the inherited elements) is the cross connection between the implements elements and the inherited elements. Consequently, cohesion value decreased in the tight method-method and attribute-method metrics as it is not frequent where each inherited element is tightly connected (used directly or indirectly) with all the implemented elements. But the implemented elements are loosely connected with inherited elements. Therefore, the loose class cohesion metrics (such as LCC and LCC_l) produce negative values when including the inherited elements.

Thus, it could be recommended empirically to use the loose cohesion metrics (like LCC and LCC_l) while evaluating the project structure, as this category of cohesion metrics detects the loosely connections between class elements.

In other cases (only two projects J2EE [23] and Log4j [24]) the difference graph is similar to pattern 3 (the DIT was negative and stepped to positive value at DIT>1), and it is recommended to revise the classes with higher DIT values.

6. Conclusions

In this paper, a novel assessment criterion based on including the inherited attributes and method has been proposed. Additionally, the effect of including the inherited attributes and methods in measuring class cohesion has been extensively discussed.
Experimental results showed that the proposed approach has a good indicator in identifying classes nominated for software refactoring, especially if the loose cohesion metrics (such as LCC and LCC_l) are used in measuring the class cohesion.

In the future, further investigations are needed to study the effect of applying other High Level Design (HLD) cohesion metrics using the proposed approach in improving the identification of classes nominated for software refactoring.

References

[1] J. Al Dallal, "Mathematical Validation of Object-Oriented Class Cohesion Metrics", International Journal of Computers, Vol. 4, No. 2, 2010, pp. 45-52.
[2] J. Al Dallal, and L. C. Briand, An Object-Oriented High-Level Design-Based Class Cohesion Metric, Simula Research Laboratory, 2009.
[3] L. Badri, and M. Badri, "A Proposal of a New Class Cohesion Criterion: An Empirical Study", Journal Of Object Technology, Vol. 3, No. 4, 2004, pp. 145-159.

[4] J. M. Bieman, and B. Kang, Cohesion and Reuse in an Object-Oriented System, In Proceedings of the 1995 Symposium on Software reusability, 1995, pp. 259-262.
[5] C. Bonja, and E. Kidanmariam, "Metrics for Class Cohesion and Similarity Between Methods", In Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 91-95.
[6] L. C. Briand, J. W. Daly, and J. K. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", Empirical Software Engineering, Vol. 3, No. 1, 1998, pp. 65-117.
[7] S. R. Chidamber, and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, pp. 476-493.
[8] B. Henderson-Sellers, Software Metrics, Hemel Hempstaed, U.K.: Prentice Hall, 1996.
[9] http://asm.ow2.org/download/index.html
[10] http://sourceforge.net/projects/freemind/
[11] http://sourceforge.net/projects/azureus/
[12] https://github.com/jbossas/jboss-as
[13] R. Barker, and E. Tempero, "A Large-Scale Empirical Comparison of Object-Oriented Cohesion Metrics", In Proceedings of the 14th Asia-Pacific Software Engineering Conference, 2007, pp. 414-421.
[14] H. Chae, Y. Kwon, and D. Bae, "A Cohesion Measure for Object-Oriented Classes", Software-Practice & Experience, Vol. 30, No. 12, 2000, pp. 1405-1431.
[15] J. Al Dallal, "Improving Object-Oriented Lack-of-Cohesion Metric by Excluding Special Methods", In Proceedings of the 10th WSEAS International Conference on Software Engineering Parallel and Distributed Systems, 2011, pp. 124-129.
[16] http://www.oracle.com/technetwork/java/javase/downloads
[17] http://java.net/projects/javaserverfaces/
[18] https://github.com/KentBeck/junit
[19] http://sourceforge.net/projects/hibernate/
[20] http://code.google.com/p/google-web-toolkit/
[21] http://sourceforge.net/projects/jhotdraw/
[22] http://jruby.org/download
[23] http://mvnrepository.com/artifact/javax/javaee-api
[24] http://logging.apache.org/log4j/
[25] R. Lincke, J. Lundberg, and W. Löwe "Comparing Software Metrics Tools", In Proceedings of the 2008 International Symposium on Software Testing and Analysis, July 20-24 2008.