A New Metric for Class Cohesion for Object Oriented Software

Anjana Gosain[1] and Ganga Sharma[2]
[1]University School of Information, Communication and Technology, Guru Gobind Singh Indraprastha University, India
[2]School of Engineering, G D Goenka University, India

Abstract: *Various class cohesion metrics exist in literature both at design level and source code level to assess the quality of Object Oriented (OO) software. However, the idea of cohesive interactions (or relationships) between instance variables (i.e., attributes) and methods of a class for measuring cohesion varies from one metric to another. Some authors have used instance variable usage by methods of the class to measure class cohesion while some focus on similarity of methods based on sharing of instance variables. However, researchers believe that such metrics still do not properly capture cohesiveness of classes. Therefore, measures based on different perspective on the idea of cohesive interactions should be developed. Consequently, in this paper, we propose a source code level class cohesion metric based on instance variable usage by methods. We first formalize three types of cohesive interactions and then categorize these cohesive interactions by providing them ranking and weights in order to compute our proposed measure. To determine the usefulness of the proposed measure, theoretical validation using a property based axiomatic framework has been done. For empirical validation, we have used Pearson correlation analysis and logistic regression in an experimental study conducted on 28 Java classes to determine the relationship between the proposed measure and maintenance-effort of classes. The results indicate that the proposed cohesion measure is strongly correlated with maintenance-effort and can serve as a good predictor of the same.*

Keywords: *Class cohesion, metrics, OO software, maintenance-effort, metric validation.*

Received June 18, 2017; accepted March 11, 2018
https://doi.org/10.34028/iajit/17/3/15

1. Introduction

Cohesion of a module has been defined in relation to procedural paradigm as the degree of relatedness of components of a module [9]. A module in which all its components contribute to a single logical task is said to have high cohesion [5].Cohesion plays an important role while designing of a module as it allows the measurement of the structural quality of the module [5]. Researchers have shown that it is comparatively easier to maintain a highly cohesive module [14]. Therefore one can say that highly cohesive modules are a pre-requisite of quality software. Since the past few decades, Object Oriented (OO) paradigm has become widespread in industry for the development of software. In OO paradigm, a class is the basic module, which contains methods (or functions) and instance variables (or attributes). Accordingly, cohesion of a class measures the degree of relatedness of these attributes and methods within a class. Building classes with high cohesion is an important goal for software developers as such a class depicts a single logical task and splitting such class into separate classes becomes difficult [2].

Various class cohesion metrics exist in literature which can be defined at design level as well as source code level [5, 26]. Design level cohesion metrics are based on the design information pertaining to a class and its method interfaces whereas actual code is used to compute the class cohesion metrics at source code level. Researchers have used class cohesion as a means of measuring quality of OO software [1, 3, 13, 25, 28]. Most of these metrics are based on two kinds of cohesive interactions for measuring cohesion [5] viz.

1. Instance variable (or attribute) usage by methods, i.e., these metrics are computed based on the number of attributes used/referenced by methods [5, 22].
2. Similarity of methods based on sharing of instance variables by methods i.e., these metrics count the number of method pairs that share instance variables [2, 8, 10, 15, 16, 18].

However, many researchers believe that metrics based on the above mentioned cohesive interactions do not properly reflect cohesion in many situations and provide only a restricted view of measuring cohesion [4, 5, 25]. Therefore, cohesion measures with different perspective on the idea of cohesive interactions must be developed in order to accurately measure cohesiveness.

In this paper, we propose a source code level measure for class cohesion focusing on instance variable usage by methods and having the following three types of cohesive interactions (or relationships)

between instance variable (attribute) and a method:

1. *Received* i.e., when method receives an instance variable as a parameter.
2. *Manipulated* i.e., when an instance variable is used in some computation inside the method body.
3. *Returned* i.e., when an attribute is returned as a value by a method.

These three cohesive relationships can give rise to 2^3=8 categories of attribute-method usage interaction. We use these interaction categories to build our new class cohesion metric, Low-level Attribute-Method usage Class Cohesion (LAMCC) metric. (To avoid confusion and as a matter of convenience, the words metric/measure and attribute/instance variable have been used interchangeably in this paper provided their meaning is preserved).

In order to determine the usefulness of a metric, researchers have stressed that it should be validated theoretically as well as empirically [19, 20, 27]. Theoretical validation assesses whether a metric conforms to the necessary properties of the measured concept i.e., whether it measures what it is supposed to measure [11]. On the other hand, empirical validation tests for the statistical relationships between a metric and measures of external software quality [11]. Consequently, in this paper, LAMCC has been theoretical analysed and has been found to comply with the axiomatic properties of cohesion proposed in Briand *et al*. [12] framework. For empirical validation, an experimental study consisting of Pearson Correlation analysis and logistic regression has been conducted using 28 sample Java classes to determine the relationship between LAMCC and maintenance-effort of classes. Maintaining a software is concerned with how easily a software can undergo changes in requirements initiated by the user or that arising from changes in real world [17]. Software maintenance is regarded as one of the most costly task in development process and requires time and effort [17, 20]. We have used change i.e., "number of lines of code added/deleted in maintaining a software artefact" as a measurable aspect of maintenance-effort in our experiment. The results of our experiment show a significant negative correlation between LAMCC and maintenance-effort and also LAMCC can be used as a good predictor of maintenance of classes.

The organization of the paper is as follows: section 2 provides an overview of several source code level cohesion measures proposed in literature, section 3 describes the measurement of our proposed cohesion measure LAMCC. Section 4 provides the theoretical validation while section 5 gives the details of the experimental study and discusses the obtained results. Section 6 gives an overview of threats to validity, while section 7 gives conclusion and future directions.

2. Literature Review

In the OO paradigm, Chidamber and Kemrer [15, 16] were the first to propose a class cohesion measure viz. Lack of Cohesion in Methods (LCOM). They computed LCOM [15] as the number of pairs of methods that do not share an attribute. Later, they modified the definition of LCOM and computed it by subtracting the number of pairs of methods which have at least one shared attribute from those pairs of methods that do not have a single shared attributes [16]. For cases where the metric value comes out to be negative, it is reset to zero. Since then, LCOM has got many variations defined by various researchers. The definition of LCOM by Li and Henry [28] used graphs to represent a class. In a graphical representation of class, a method is represented by a vertex and sharing of attribute is represented by an edge. Then, the number of connected components in the graph gives the LCOM value for the class. Hitz and Montazeri [23] also used the same graph theoretic approach as [28] to define LCOM except that an edge now also represents method invocations. Hendersen-Sellers [22] compute LCOM by counting the number of instance variables referenced by a method. The authors proposed that when a method references more instance variables, the cohesiveness of the class increases. These variations of LCOM have been used extensively in various empirical studies to predict fault-proneness [1, 13], maintainability [3, 28] etc.

The cohesion measures Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC) given by Bieman and Kang [8] are based on the criteria of common instance variable usage by method pairs. The authors proposed that two methods are said to be connected if they directly (or indirectly) use/refer the same instance variable. A direct usage of an instance variable *A* by a method *M* is characterized by the fact that *A* appears in the body of *M*. Whereas, an indirect usage of *A* by *M* is characterized when *A* is directly referenced by a method *M'* which is directly or indirectly called by *M*. TCC is defined as the percentage of pairs of methods that have direct connection while LCC is defined as percentage of pairs of methods that have direct or indirect connections. Bonja and Kidanmariam [10] defined similarity degree between two methods by computing the ratio of shared attributes to total number of attributes used by the methods and used it to obtain their Class Cohesion (CC) measure as the ratio of the summation of the similarity degrees between all pairs of methods to the total number of method pairs. The authors showed that CC captures more cohesiveness as compared to other cohesion metrics [10]. Fernandez and Pena [18] defined their measure Sensitive Class Cohesion Measure (SCOM) using connection intensity and a weight factor. The connection intensity between two methods *M1* and *M2* is computed as the ratio of

number of common attributes between *M1* and *M2* to the maximum number of attributes used by either *M1* or *M2*. The weight factor for a method pair is computed as the ratio of number of shared attributes in the method pair to the total number of attributes in class. Then, the summation of product of connection intensity and weight factors over all possible method pairs gives the value of SCOM. AlDallal and Briand [2] also used similarity between methods to define the metric Low-level design Similarity-based Class Cohesion (LSCC). For this, they used a binary mXn matrix called Method Attribute Reference matrix (MAR) where m represents number of methods and n represents number of attributes. The value in a given cell of MAR is 1 if the corresponding method references the corresponding attribute, otherwise it is 0. They proposed that two methods are similar if the entries in their corresponding rows in *MAR* are similar and defined this similarity as the number of entries that are similar. This similarity is then averaged for all method pairs in the class to compute LSCC. They then used LSCC, along with several other cohesion measures [5, 16, 18, 21, 22, 23, 28] in predicting fault prone classes.

Badri and Badri [5] proposed that even if two methods of a class do not share any common attribute still they may be related to each other. Instead of using similarity based criteria, the authors emphasized the use of interaction patterns between methods of a class to measure cohesion. For example, private or protected methods of a class generally do not refer to any attribute of the class. If two public methods invoke (directly or indirectly) such private or protected methods, they are said to be related. They then proposed two cohesion measure *viz* DC_D which computes the percentage of directly connected public methods pairs and DC_I which gives the percentage of methods pairs which are directly or indirectly related. They found that DC_D and DC_I capture more method pairs and concluded that these metrics are better at measuring class cohesion than their counterparts like [8,16].

Aman *et al.* [4] take into account sizes of cohesive parts while defining their measure Association Extent based Class Cohesion (AECC). They propose that the size of cohesive part gives an indication about the extent of association between a pair of methods through attribute-attribute usage or method invocation in a class. They build an association graph G_a to compute AECC as the ratio of the number of methods reachable by a method m_i in its association graph G_a to the total number of methods in the class. They further performed a correlation analysis of AECC with other class cohesion metrics and showed that AECC is a reasonable class cohesion metric.

Table 1 provides an overview of above mentioned metrics.

Table 1. Brief overview of source code level cohesion metrics.

SNO	Cohesion Metric	Definition
1.	LCOM1 [15]	It is a count of method pairs that do not share attributes.
2.	LCOM2 [16]	Given that P= method pairs that do not have any shared attributes. Q=method pairs that have at least one attribute in common. LCOM2 = $\begin{cases} P-Q if P-Q \geq 0 \\ 0 \quad otherwise \end{cases}$
3.	LCOM3 [28]	It is the number of connected components in graphical representation of a class where method is depicted as vertex and sharing of attribute is depicted as an edge.
4.	LCOM4 [23]	Same as LCOM3, except method invocations are also represented as an edge.
5.	LCOM5 [22]	LCOM5= (m*l*-*a*)/(m*l*-*l*) Where, *m*=number of methods *l*=number of attributes *a*=number of distinct attributes referenced by a method.
6.	TCC [8]	Percentage of method pairs that have direct connection. Two methods M and M' are directly connected if they access a common attribute A directly, i.e., A appears in their method body.
7.	LCC [8]	Percentage of method pairs that have direct or indirect connections. Two methods M and M' are indirectly connected if they access an attribute A indirectly.
8.	DC_D [5]	Number of directly connected method pairs. A direct connection exists between two methods M and M' if they directly call the same method M''.
9	DC_I [5]	Number of directly or transitively connected method pairs. A transitive connection exists between Two methods M and M' if they directly or transitively call the same method M''.
10.	SCOM [18]	SCOM= $\frac{2}{m(m-1)}\sum_{i=1}^{m-1}\sum_{j=i+1}^{m} C_{i,j} * \alpha_{i,j}$ Where m is the number of methods C is the connection intensity α is the weight factor
11.	CC [10]	It is based on similarity degree between two methods *m* and *n* defined as: MS(m,n) = $\frac{\lvert IV_c \rvert}{\lvert IV_t \rvert}$ where IV_c is the set of shared attributes between m and n while IV_t is the set of total number of attributes referenced by m and n. The ratio of the summation of similarity degrees of all pairs of methods to the total number of method pairs gives the value of CC.
12.	LSCC [2]	LSCC= $\begin{cases} 0 \text{ if } l=0 \text{ and } k>1, \\ 1 \; if (l>0 \; and k=0) or k=1 \\ \frac{\sum_{i=1}^{l} x_i(x_i-1)}{k(k-1)} \end{cases}$ Where k= number of methods l= number of attributes
13.	AECC [4]	AECC= $\begin{cases} max\left[\frac{\lvert R_a(m) \rvert}{\lvert M \rvert -1}\right], \lvert M \rvert > 1 \\ 0 \quad , \lvert M \rvert < 1 \end{cases}$ Where R_a is the graph reachable by method m in its association graph G_a and M is the total number of methods.

3. Cohesion Measurement

Most cohesion metrics developed thus far have focused either on instance variable (attributes) usage by methods of the class or similarity of methods based on sharing of instance variables (attributes). We have also used the attribute usage by methods of a class as our basis for defining our new class cohesion metric. However, the idea of cohesive interactions (or relationships) is interpreted from a different perspective. We propose that an instance variable and a method of a class can have three types of cohesive interactions (or relationships) viz.

1. *Received* i.e., when method receives an instance variable as a parameter.
2. *Manipulated* i.e., when an instance variable is used in some computation inside the method body.
3. *Returned* i.e., when an attribute is returned as a value by a method.

These three types of cohesive interactions are very intuitive in nature. A method of a class can be related to an attribute of the class in the way it accesses it or manipulates it or returns a value for it. The more an attribute and method are related by these cohesiveinter actions, the more will be the cohesion of the class. These cohesive interactions also emphasize the fact that when a method is not using an attribute in any of these ways then that method is doing a task which is altogether different from the goal of the class. Therefore such kinds of methods should be avoided from including inside the class. Consider a class C having A as the set of attributes and M as the set of methods. Here, M is taken as the set of normal methods i.e., those methods which are involved in the actual functionality of the class. Special methods like constructors, destructors etc. are excluded as they do not contribute much towards cohesion measurement [21]. Then, for an attribute $a_i \in A$ and method $m_j \in M$, we define our three types of cohesive relationships in the following way:

- Received R_v: (A×M) such that $a_i\ R_v\ m_j$ if $a_i \in A$ is passed as a parameter to the method $m_j \in M$.
- Manipulated M_v: (A×M) such that $a_i M_v mj$ if $a_i \in A$ is used in some computation in the method $m_j \in M$. A computation can be a mathematical computation or a function call. When a function call is made then the attributes used in the called method also become related to the calling method by an M_v cohesive relation. This way, we have also captured the transitive nature of M_v relation.
- Returned Rt_v: (A×M). such that $a_i Rt_v m_j$ if the value of $a_i \in A$ is returned by the method $m_j \in M$.

Note that $R_v \cap M_v \cap Rt_v \neq \phi$.

These three cohesive interactions or relationships can combine to form $2^3=8$ categories of instance variable usage by methods as shown in Table 2. We also propose that these categories can have different rankings from 1 to 8 (1 being highest and 8 being lowest) based on the category's importance towards measuring cohesion and therefore can be given weights. These weights of each category can be assigned based on the opinion of experienced software professionals [21]. As an example, consider category I which contains methods that do not receive an instance variable as parameter, do not manipulate any instance variable and do not produce any instance variable value. These types of methods tend to perform a functionality which is altogether different from the goal of the class. Such type of methods should not be included in a class. Hence this category has been given the lowest rank and lowest weight (refer Table 2). On the other hand, category VIII has methods which receive as well as manipulate and return some instance variables. These types of methods increase the cohesiveness of the class. Therefore the methods belonging to this category have been given highest rank and highest weight. Other categories have been provided weights in between. In order to maintain a normalized effect of these weights, for our purpose, the values have been chosen in the interval [0, 1].

3.1. Proposed Cohesion Measures

3.1.1. Method Cohesion (MC)

The cohesion of j[th] method m_j is given by

$$MC(m_j)=\frac{|R_{vj} \cup Rt_{vj} \cup M_{vj}|}{|A|} \quad (1)$$

The numerator will have the value 1 if a method m_j receives, manipulates as well as returns an instance variable. In that case cohesion of method m_j will be 1.

3.1.2. Class Cohesion (LAMCC)

Now, we define cohesion of a class *C* as the weighted average of the cohesion of all of its methods $m_j \in M$. Hence

$$LAMCC(C)=\frac{\sum_{j=1}^{|M|} w_j * MC(m_j)}{\sum_{j=1}^{|M|} w_j} \quad (2)$$

LAMCC for a class *C* will be 0 if all the methods of the class have *MC* values as 0 and it will be 1 if all the methods of the class have *MC* values as 1.

3.2. A Worked Example

Consider an example of a Stack class (Example 1) [21]. This class has four normal methods M1= push, M2=pop, M3=is Stack Empty, M4=top of Stack and two attributes A1=stck[] and A2=tos. Then, MC(M1)=2/2=1; MC(M2)=2/2=1; MC(M3)=1/2=0.5; MC(M4)=1/2=0.5; and thus LAMCC(Stack)=1*0.8+1*0.8+0.5*0.4+0.5*0.4/2.4=2/2.4=0.833.

Table 2. Cohesive relationship categories.

Category	R_v	M_v	Rt_v	Interpretation	Ranking	Weight
I	X	X	X	None of the instance variables are received/ manipulated/ returned	8	0
II	X	X	✓	None of the instance variables are received/manipulated. However some are returned.	6	0.4
III	X	✓	X	None of the instance variables are recieved/ returned. However, some are manipulated.	4	0.8
IV	X	✓	✓	None of the instance variables are received. However, some are manipulated and returned.	2	0.9
V	✓	X	X	None of the instance variables are manipulated/ returned. However, some are received.	7	0.2
VI	✓	X	✓	None of the instance variables are manipulated. However, some are received/ returned.	5	0.6
VII	✓	✓	X	Some of the instance variables are received /manipulated. However, none is returned.	3	0.9
VIII	✓	✓	✓	Some of the instance variables are received as well as returned and manipulated.	1	1

- *Example* 1: Stack Class

```
class Stack {
int stck[];
int tos;
Stack(int size){
        stck= new int[size];
        tos=-1;}
push (int item){
        if (tos==stck.length-1)
        System.out.println("Stack is full");
        else
           stck[++tos]=item;}
pop(){
        if(isStackEmpty){
                System.out.println("Stack underflow");
                return 0;}
        else
                return stck[tos--];}
int isStackEmpty(){return tos==-1;}
int topofStack(){return stck[tos-1];}
}
```

Table 3 represents a comparison of LAMCC with several variations of LCOM metric for the code presented in Example 1.

Table 3. LAMCC and LCOM.

LAMCC	LCOM1 [15]	LCOM2 [16]	LCOM3 [28]	LCOM4 [23]	LCOM5 [22]
0.833	0	0	1	1	1

4. Theoretical Validation

We have used one of the most frequently used theoretical framework given by Briand *et al*. [12] for validating our proposed measure LAMCC. The framework provides axiomatic properties for analysing different measurement concepts for software artefacts like size, length, complexity, coupling and cohesion. Below we give a brief overview of the properties defined for the concept of cohesion in this framework and consequently prove that our proposed measure LAMCC conforms to these properties.

- *Property* 1: *Non-negativity*. It states that a cohesion measure cannot have values less than 0.
- *Proof*. LAMCC is computed from the modulus of union of three types of cohesive interactions R_v, Rt_v and M_v, therefore its value cannot be negative. Hence, this property is satisfied.
- *Property* 2: *Normalization.* The value of a cohesion measure is contained in the interval [0, Max].
- *Proof*. LAMCC attains minimum value(i.e., 0) when there are no cohesive relationships and goes to a maximum value (i.e.,1) when all the cohesive relationships are present. Hence, this property is satisfied.
- *Property* 3: *Null Value*. It states that if no cohesive interactions exist, then the cohesion values of a class is equal to 0.
- *Proof*. According to the definition of LAMCC, it achieves value 0 when a class contains only the methods belonging to category I, i.e., no cohesive interactions are there. Hence, this property is satisfied.
- *Property* 4: *Monotonicity*. It states that adding a cohesive relationship to a module cannot decrease its cohesion.
- *Proof*. Adding a cohesive relation to our model implies that either $|R_v|$ or $|M_v|$ or $|Rt_v|$ increases by 1. This can have the effect that $| R_v \cup M_v \cup Rt_v |$ may increase or remain same. But it can never decrease. Consequently, LAMCC may increase or remain same. Hence this property is also satisfied.
- *Property 5: Cohesive modules*. This property states that if two unrelated modules are merged, then the cohesion of the merged module does not increase. In context of OO software, given two unrelated classes C1 and C2 which do not have common attributes and methods, the cohesion of a class C formed by merging C1 and C2 cannot exceed the maximum cohesion of the individual classes.
- *Proof*. Suppose that two unrelated classes C1 and C2 are merged form class C, then the merged class C contains all the attributes and methods of C1 and C2. This has the effect that the cohesive relations in merged class C can remain same or can decrease but can never increase because a method from one of the split class will not access an attribute fromthe other split class. Hence LAMCC(C)<=

{max(LAMCC(C1),LAMCC(C2))}. Hence this property is also satisfied.

5. Empirical Validation

In this section, we assess the empirical validity of the proposed measure LAMCC in predicting maintenance-effort of classes. Maintenance is the process that involves changing a software due to fault(corrective), need for improvement(perfective), or adapting to hardware/software environment change (adaptive) [6, 24]. Using software metrics for predicting class maintenance has been an active area of research in software engineering community [3, 6, 20]. For our purpose, we have operationalized maintenance as "the effort expended in incorporating a new/changed requirement". We call it as maintenance-effort (maint-effort) and have measured it in terms of change, i.e., number of lines of code added/deleted while incorporating the new/changed requirement". Change has been used as an indicator of maintenance-effort in various empirical studies [3, 28]. These studies have indicated that classes with more changes require more maintenance-effort than those with fewer changes. Therefore, we have also conducted our experimental study consistent with this approach.

5.1. Experimental Set-up

We selected 28 sample Java classes from various sources like [29, 30] and web (with slight coding modifications) for the experiment. Table 4 presents these classes with their corresponding LAMCC values.

Table 4. Sample java classes.

SNO	Class	LAMCC
C1	Loan	0.345
C2	BMICalculator	0.62
C3	Point	0.33
C4	Employee	0.725
C5	SwapCircle	0.28
C6	EmpBusinessLogic	0.95
C7	Calendar	0.50
C8	MessageUtil	0.392
C9	Course	0.452
C10	Graph	0.19
C11	Measures	0.22
C12	Box	0.356
C13	Stack	0.833
C14	Complex	0.93
C15	ShoppingCart	0.226
C16	Account	0.77
C17	Lottery	0.26
C18	Account	0.73
C19	Circle	0.486
C20	Manager	0.666
C21	Triangle	0.382
C22	StackOfIntegers	0.775
C23	Car	0.18
C24	Rectangle	0.52
C25	BangBean	0.266
C26	Queue	0.82
C27	Tax	0.73
C28	Hex2Dec	0.41

5.1.1. Experimental Goal

This experimental study is conducted with the goal of finding whether the proposed measure LAMCC is statistically related with maintenance-effort of classes. Accordingly, as suggested in [32], we use Goal Question Metric (GQM) [7] template for this purpose (Table 5).

Table 5. GQM [7] Template for experimental goal.

Analyze	Proposed cohesion measure
For the purpose of	evaluating
With respect to	The relationship with maintenance-effort of OO classes
From the point of view of	Researchers
In the context of	Postgraduate computer science students

5.1.2. Planning

- *Selection of context*: In our experimental study, we try to establish the fact that the proposed cohesion measure can be used as indicator of maintenance-effort of OO classes.
- *Selection of subjects*: We conducted the experimental study with 28 subjects who were in the final year of post-graduation in computer science at *USICT, GGSIPU, New Delhi*.[1] Various experimental studies in the field of software metric validation have used students as subjects [6] as researchers have always encouraged such pilot studies in academic environment [7]. The participation of the subjects was voluntary. Although the subjects were chosen for convenience (as we could not find software professionals), these subjects had requisite grasp of the concepts of OO, Software Engineering and Java and some of them also had industrial experience.
- *Selection of variables*: For our purpose, the independent variable is the OO class cohesion and the dependent variable is maintenance-effort.
- *Instrumentation*: We have used our proposed measure LAMCC to measure the independent variable (class cohesion). The dependent variable (maintenance-effort) was operationalized as "the number of lines of code added/deleted while incorporating a new/changed requirement". Many researchers have used this approach in the field of maintainability prediction [3, 28].
- *Experimental Design*. Each subject was given one sample Java class randomly. This randomization in an experimental design is said to curb any kind of bias [32].
- *Empirical Hypotheses*. In our case, we define two empirical hypotheses as follows:

$H1_0$:(Null hypothesis)-No statistically significant relationship exists between the proposed cohesion measure LAMCC and maintenance-effort of classes.
$H2_0$:(Null Hypothesis)-Classes with low LAMCC are not costly to maintain.

[1]University School of Information, Communication and Technology, Guru Gobind Singh Indraprastha University, New Delhi

$H1_1$:(Alternate hypothesis)-A statistically significant relationship exists between the proposed cohesion measure LAMCC and maintenance-effort of classes.
$H2_1$:(Alternate Hypothesis): Classes with low LAMCC are costly to maintain.

5.1.3. Operation

- *Preparation*. To prepare the subjects for the experiment, they were asked to attend to a training session in which they were made aware regarding some do's and dont's of the experiment. For e.g., they were given clear instructions on what type of conduct and behaviour they should pursue during the task, how maintenance-effort values would be recorded by them etc. Nonetheless, the instructions were carefully disseminated so that the subjects did not get any idea about the empirical hypotheses under study.

After the training session, the experimental task were given as a handout document consisting of:

a) Source code of the sample Java class source code.
b) A short summary about the functioning of the class.
c) One new/changed requirement that had to be integrated into the class functionality.

Table 6. Maintenance-effort values.

SNO	LAMCC	Maint-effort
C1	0.345	22
C2	0.62	18
C3	0.33	20
C4	0.725	13
C5	0.28	19
C6	0.95	11
C7	0.5	20
C8	0.392	25
C9	0.452	19
C10	0.19	25
C11	0.22	18
C12	0.356	16
C13	0.833	12
C14	0.93	15
C15	0.226	22
C16	0.77	16
C17	0.26	15
C18	0.73	18
C19	0.486	20
C20	0.666	16
C21	0.382	16
C22	0.775	18
C23	0.18	22
C24	0.52	18
C25	0.266	24
C26	0.82	15
C27	0.73	12
C28	0.41	20

For e.g., in the Emp Business Logic class (C6), the experimental task consisted of adding a new functionality - calculating the house tax from salary of an employee. Accordingly, the subjects reported the number of lines of code added/deleted while including this new functionality.

Execution: The experimental tasks were given as assignments to the subjects. The subjects performed the tasks at home (without any supervision) and submitted the same to us after completion within two days.

Table 6 provides the LAMCC and the collected maintenance-effort values.

5.2. Data Analysis Methodology and Results Discussion

5.2.1. Correlation Analysis

The Pearson correlation analysis was conducted to test the hypothesis $H1_0$ and $H1_1$ (refer section V). The value of the correlation coefficient r signifies the strength of relationship while the sign represents the direction of relationship between two variables. The coefficient values lie in the interval [-1, 1] where r=1 represents perfect positive correlation, r=-1 represents perfect negative correlation; and r=0 indicates absence of relationship. We have used the adjective ratings as used in [21] for interpreting other values of r (with p=0.01).
Table 7 shows the values of Pearson correlation coefficient between LAMCC and maintenance-effort. As can be inferred from the table, LAMCC has a significant negative correlation with maintenance-effort. This means higher LAMCC values indicate that a class will require less maintenance-effort to make changes to its functionality while classes with lower LAMCC values will be more prone to changes and hence will require more maintenance-effort. Therefore, we reject the null hypothesis $H1_0$ and accept the alternate hypothesis $H1_1$.

Table 7. Pearson correlation coefficient.

	Maint-effort	P
LAMCC	-0.716	<.001

5.2.2. Logistic Regression

We used univariate logistic regression analysis to test the hypothesis $H2_0$ and $H2_1$. It is a statistical technique based on maximum likelihood estimation [13]. The dependent variable in a logistic regression based prediction model is dichotomous i.e., has two values only. Therefore we used a binary variable Costly Maintained Class (CMC) to indicate class maintenance-effort for our experiment (as suggested in [3]). A class is said to be costly to maintain if its maintenance-effort (i.e., number of lines of code added/deleted) is greater than the mean value of maintenance-effort of all classes under study. The CMC of such a class has been set to ''1''; otherwise, the CMC value has been set to ''0'' (Table 8). Classes with relatively high number of lines of code added/deleted value are costly as these classes supposedly require more maintenance [3].

Table 8. Logistic regression table.

SNO	LAMCC	CMC
C1	0.345	1
C2	0.62	1
C3	0.33	1
C4	0.725	0
C5	0.28	1
C6	0.95	0
C7	0.5	1
C8	0.392	1
C9	0.452	1
C10	0.19	1
C11	0.22	1
C12	0.356	0
C13	0.833	0
C14	0.93	0
C15	0.226	1
C16	0.77	0
C17	0.26	0
C18	0.73	1
C19	0.486	1
C20	0.666	0
C21	0.382	0
C22	0.775	1
C23	0.18	1
C24	0.52	1
C25	0.266	1
C26	0.82	0
C27	0.73	0
C28	0.41	1

The classification model built in univariate logistic regression uses the following equation

$$\pi(X)=\frac{1}{1+e^{-(C_0+C_1X)}} \quad (3)$$

Where π is the probability that a class is costly to maintain and X is the independent variable LAMCC. The following statistics are reported in order to evaluate the performance of the model:

1. B: it is the logistic regression coefficient for the independent variable in the equation. It tells the impact of the independent variable LAMCC on dependent variable CMC. The sign of the coefficient tells the direction of the impact.
2. Precision: It gives the ratio of the number of classes which are correctly classified as costly to maintain to the total number of classes which are classified costly.
3. Recall: It gives the ratio of the number of classes which are correctly classified as costly to the total number of actual classes that are costly.
4. Area Under Curve (AUC): It is the area under the curve in a plot called Receiver Operator Characteristic (ROC). In our case, ROC graph is a depiction of the proportion of classes which are correctly classified as costly to the classes which are incorrectly classified as costly (at different threshold levels) (Figure 1). Consequently, AUC indicates the effectiveness of the model to correctly classify the classes as costly. Therefore, higher values of AUC indicate a better model. We have used the criteria used in [3] to assess the classification performance of our model such that the values vary from AUC<=0.5 (not good) to 0.9<AUC<1.0 (outstanding)
5. R^2: We also report Nagelkerke R^2 to provide the goodness-of-fit. It denotes the proportion of the variance in the dependent variable that is explained by the variance in the independent variable.

Table 9 provides the contingency matrix of actual and predicted CMC variable, Table 10 gives the univariate logistic regression results.

Table 9. Contingency matrix.

		Predicted	
		Not Costly	Costly
Actual	Not Costly	9	3
	Costly	2	14

Table 10. Univariate logistic regression results.

	B	R^2	p-value	Precison	Recall	AUC
LAMCC	-6.156	.414	.007	.823	.875	81%

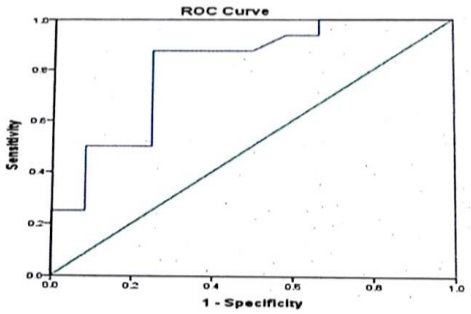

Figure 1. ROC curve.

The following points can be inferred from Table 9:

1. The regression coefficient (B) comes out to be negative, indicating an inverse relationship between LAMCC and maintenance effort. This is expected as classes with low cohesion values would require more effort to maintain [3].
2. One can infer the high values of precision (82.3%), recall (87.5%) and AUC (81%) from Table IX. This reinforces the fact that cohesiveness of a class can reasonably be used to predict the effort needed to maintain it.
3. Based on the above two observations, we reject the null hypothesis $H2_0$ and accept the alternate hypothesis $H2_1$.

6. Threats to Validity

Researchers have suggested that an experimental study may be subjected to certain threats to the validity related to the reported results [31, 32], limiting the generalization and interpretation of the reported results. We have therefore listed down these threats as follows:

6.1. Construct Validity

It deals with measurement of variables under study. If the measurement instrument/s accurately measure the variables (dependent as well as independent), we say that the variables are constructively valid. For our case, the theoretical validation (refer section 4) ensures the construct validity of the independent

variable LAMCC as a cohesion measure. The dependent variable is reported consistent with the approach used in other studies like [3, 28], so we consider the dependent variable (maint-effort) to be constructively valid also.

6.2. Internal Validity

It deals with the cause-effect relationship between the independent and the dependent variable. If a study is able to effectively establish this cause-effect relationship, we say that it is internally valid. In order to achieve this, one should try to control the effect of extraneous factors. Several of these factors which we dealt with are:

1. *Motivation of subjects*. All the subjects who were involved in this experiment participated voluntarily with great enthusiasm. Also, we motivated them from time to time that their effort will help them in growing as good software professionals. So, we believe that the subjects were reasonably motivated.
2. *Bias*. We tried to curb bias when experimental tasks were assigned to the subjects by using randomization which is an effective way of minimizing bias [31, 32].
3. *Persistent Effect*. The subjects were participating in an experimental study for the first time, therefore we reasonably believe the absence of persistent effect.
4. *Precision*. The students were responsible for recording the values for maintenance-effort. Although this approach has been reported in several studies [6], however, we are aware that this might have introduced imprecision.
5. *Plagiarism and influence among subjects*. This was not an issue as each subject received a different experimental task.

6.3. External Validity

It deals with whether the results of an experimental study can be generalized to other research settings. For our data set, all considered classes are implemented in Java. One should also consider the applicability of the proposed measure for other OO languages like C++ for generalization. The sample classes were chosen randomly yet keeping in mind that they represented different domains. Still, we are aware that the dataset might have influenced our conclusions. Therefore, a replicated study with different datasets should be performed. Lastly, the experiment used students as subjects. However, we feel that using students for the experimental study should not be an issue as researchers like [31] favour the use of students by arguing that "students are next generation of software professionals and therefore are close to the population of interest". Nonetheless, a study with professionals should be conducted.

7. Conclusions and Future Direction

We have proposed a new source code level class cohesion metric LAMMC based on instance variable usage by methods. Three types of cohesive relationships are defined i.e., received, manipulated and returned, which give rise to eight different categories of cohesive interactions. The proposed measure LAMCC is then computed by giving these cohesive interactions categories and weights depending upon the importance of the corresponding category in measuring class cohesion. The proposed measure is theoretically validated as well as has been empirically shown to be related with maintenance-effort of classes.

In future, we plan to perform a comparative analysis of the proposed measure LAMCC with other existing source code level cohesion measures for OO software. We also plan to conduct experiments to correlate the proposed cohesion measure with other external quality factors like reusability, fault-proneness etc.

References

[1] Aggarwal K., Singh Y., Kaur A., and Malhotra R., "Investigating Effect of Design Metrics on Fault Proneness in Object-Oriented Systems," *Journal of Object Technology*, vol. 6, no. 10, pp.127-141, 2007.

[2] Al-Dallal J. and Briand L., "A Precise Method-Method Interactionbased Cohesion Metric for Object-Oriented Classes," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, pp. 1-34, 2011.

[3] Al-Dallal J., "Object-Oriented Class Maintainability Prediction Using Internal Quality Attributes," *Information and Software Technology*, vol. 55, no. 11, pp. 2028-2048, 2013.

[4] Aman H., Yamasaki K., Yamada H., and Noda M., "A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts," *in Proceedings of 5th Joint Conference On Knowledge-Based Software Engineering*, Rozman, pp. 102-107, 2002.

[5] Badri L. and Badri M., "A Proposal of A New Class Cohesion Criterion: An Empirical Study," *Journal of Object Technology*, vol. 3, no. 4, pp. 145-159, 2004.

[6] Bandi R., Vaishnavi V., and Turk E., "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics," *IEEE Transactions on Software Engineering*, vol. 29, no. 1, pp. 77-87, 2003.

[7] Basili V. and Weiss D., "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728-738, 1984.

[8] Bieman J. and Kang B., "Cohesion and Reuse in an Object-Oriented System," *in Proceedings of the Symposium on Software Reusability*, Seattle, pp. 259-262, 1995.

[9] Bieman J. and Ott L., "Measuring Functional Cohesion," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 644-657, 1994.

[10] Bonja C. and Kidanmariam E., "Metrics for Class Cohesion and Similarity between Methods," *in Proceedings of the 44th Annual ACM Southeast Regional Conference*, Melbourne, pp. 91-95, 2006.

[11] Briand L., El Emam K., and Morasca S., "Theoretical and Empirical Validation of Software Product Measures," Technical Report ISERN-95-03, 1995.

[12] Briand L., Morasca S., and Basili V., "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-86, 1996.

[13] Briand L., Wüst J., Ikonomovski S., and Lounis H., "A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study," Technical Report, ISERN, 98-29, 1998.

[14] Briand L., Bunse C., and Daly J., "A Controlled Experiment for Evaluating Quality Guidelines on The Maintainability of Object-Oriented Designs," *IEEE Transactions on Software Engineering*, vol. 27, no.6, pp. 513-530, 2001.

[15] Chidamber S. and Kemerer C., "Towards a Metrics Suite for Object-Oriented Design," *in Proceedings of Object-Oriented Programming Systems, Languages and Applications*, Phoenix Arizona, pp. 197-211, 1991.

[16] Chidamber S. and Kemerer C., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[17] Deligiannis I., Shepperd M., Roumeliotis M., and Stamelos I., "An Empirical Investigation of An Object-Oriented Design Heuristic for Maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp.127-139, 2003.

[18] Fernandez L. and Pena R., "A Sensitive Metric of Class Cohesion," *International Journal of Information Theory and Applications*, vol. 13, no.1, pp. 82-91, 2006.

[19] Gosain A. and Sharma G., "Object Oriented Dynamic Complexity Measures for Software Understandability," *Innovations in Systems and Software Engineering*, vol. 13, no. 2-3, pp. 177-190, 2017.

[20] Gosain A. and Sharma G., "Predicting Software Maintainability Using Object Oriented Dynamic Complexity Measures," *in Proceedings of International Conference on Smart Trends for Information Technology and Computer Communications*, Jaipur, pp. 218-230, 2016.

[21] Gupta V. and Chhabra J., "Dynamic Cohesion Measures for Object-Oriented Software," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 452-462, 2011.

[22] Henderson-Sellers B., *Object Oriented Metrics: Measures of Complexity*, Prentice Hall Inc., 1996.

[23] Hitz M. and Montazeri B., "Measuring Coupling and Cohesion in Object Oriented Systems," *in Proceedings of International Symposium on Applied Corporate Computing*, Monterrey, pp. 25-27, 1995.

[24] IEEE Std. 610.12-1990. Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press, Los Alamitos, CA, 1993.

[25] Kabaili H., Keller R., and Lustman F., "Cohesion as Changeability Indicator In Object-Oriented Systems," *in Proceedings of 5th European Conference on Software Maintenance and Reengineering*, Lisbon, pp. 39-46 2001.

[26] Kaur K. and Singh H., "An Investigation of Design Level Class Cohesion Metrics," *The International Arab Journal of Information Technology*, vol. 9, no. 1, pp. 66-73, 2012.

[27] Kitchenham B., Pfleeger S., and Fenton N., "Towards A Framework for Software Measurement Validation," *IEEE Transaction on Software Engineering*, vol. 21, no. 12, pp. 929-944, 1995.

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

[29] Liang, Y., *Introduction to Java Programming*, Prentice Hall Inc, 2014.

[30] Naughton P., and Schildt H., *Java 2: The Complete Reference*, McGraw-Hill, 1999.

[31] Kitchenhem B., Pfleeger S., Pickard L., Jones P., Hoaglin D., El Emmam K., and Rosenberg J., "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721-734, 2002.

[32] Wohlin C., Runeson P., Höst M., Ohlsson M., Regnell B., and Wesslen A., *Experimentation in Software Engineering*, Kluwer Academic Publishers, 2000.

Anjana Gosain is currently working as Professor at University School of Information, Communication & Technology, Guru Gobind Singh Indraprastha University, Dwarka, New Delhi, India. She has worked in the areas of data warehousing, data mining, requirements engineering, conceptual modelling, software engineering and machine learning and has authored/co-authored over 90 research publications in peer-reviewed reputed international journals, book chapter sand conferenceproceedings.

Ganga Sharma is currently working as Assistant Professor at School of Engineering, G D Goenka University, Sohna, Gurgaon-122103, Haryana, India. Her research interests include software engineering, object oriented analysis and design, software metrics and aspect orientation.