Journal of Theoretical and Applied Information Technology

www.jatit.org

COHERENCY OF CLASSES TO MEASURE THE QUALITY OF OBJECT ORIENTED DESIGN
AN EMPIRICAL ANALYSIS

M.V.VIJAYA SARADHI[1], B.R.SASTRY[2]

[1] Assoc.Prof &HOD, Dept. of CSE, ASTRA, Bandlaguda, Hyderabad, India
[2] Director, ASTRA, Bandlaguda, Hyderabad, India

ABSTRACT

Software engineering is an expensive venture to undertake mainly due to not knowing exactly where to focus the reengineering efforts. This is where coupling and cohesion play an important role. Both the metrics, especially cohesion metric, are a potential identification tools that can also measure progress. The current cohesion metrics for classes overlook the characteristics of indirect usage of the instance variables that are so often applicable in a class. Therefore, they are not an ideal reflection of the cohesiveness of the class. Nevertheless, when the variability factor is taking into consideration, the class cohesion factor has not been quantitatively studied. In this paper, we will propose the updated cohesion metrics based on the role of indirect usage of the instance variables. This updated version of cohesion and coupling relation can be used as an indicator to measure changeability. In this paper, we present an approach for remodeling the cohesion metrics by analyzing the characteristics of the indirect instance variable usage in an object-oriented program that can be used as a quality indicator in terms of changeability.

Keywords: *Cohesion metrics ; Instance Variable ; Changeability; LCOM; Co; Coh, TCC; LCC; CBMC*

1. INTRODUCTION

The application of Object-Oriented Technology for software development has become fairly popular. Research conducted so far has shown that Object Oriented practices guarantee a superior quality of
software by way of maintainability, reusability and flexibility. In order to evaluate the software industrial buyers acquire, they want to ensure the quality of product. This is where Object Oriented measures come into play. One of the important quality properties of Object Oriented designs is cohesion. A number of metrics have been suggested to quantify and measure this design property. In this paper, we attempt to evaluate cohesion as an indicator of changeability. In a few earlier works, coupling has been corroborated as a quality indicator. By establishing a link between cohesion and coupling, we will be able to affirm that cohesion is quality indicator, as well.

2. COHESION AND DESIGN QUALITY

One of the imperatives to building a quality Object Oriented system is a good design. For this, quantification of the design property is required. Several software metrics have been developed to assess and control the design phase and its products. One of the most vital criteria in Object Oriented design is cohesion. A module is said to have a strong cohesion if it closely characterized with one task of the problem domain, and all its components contribute to this single task. Module cohesion was introduced by Yourdon and Constantine as “how tightly bound or related the internal elements of a module are to one another”. According to them, cohesion is an attribute, not of any code, but of a design that can be utilized to forecast reusability, maintainability, and changeability. These assumptions, however, have never been backed up by any experimentation.

2.1 Cohesion and Cohesion Metrics

There is unanimity in the literature on the theory of class cohesion. A class is cohesive if it cannot be partitioned into two or more sets defined as follows Each set contains instance variables and methods. Methods of one set do not access variables of another set either directly or indirectly. By way of defining cohesion metrics,

many authors have effectually defined class cohesion. So far as the Object Oriented model is concerned, almost all of the cohesion metrics are influenced by the LCOM metric that is defined by Chidamber and Kemerer. According to them, "if an object class has different methods performing different operations on the same set of instance variables, the class is cohesive". The LCOM (Lack of Cohesion in Methods) defined by them is the result gained from deducting the number of pairs of methods in a class having no common attributes from the number of pairs of methods in a class sharing at least one attribute. If the value reached in this calculation is in the negative, the metric is set to zero. This is one metric for assessing cohesion.

Likewise, Li and Henry defined LCOM as the number of disjoint sets of methods accessing similar instance variables. Hitz and Montazeri reaffirm Li's definition of LCOM based on the graph theory which defines LCOM as the number of connected components of a graph. A graph consists of vertices and edges. Vertices represent methods. There is an edge between 2 vertices if the corresponding methods access the same instance variable. Hitz and Montazeri propose to divide a class into smaller, more cohesive classes, if LCOM > 1. However, Bieman and Kang proposed TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) as cohesion metrics, based on Chidamber and Kemerer's approach. Although they too consider pairs of methods using common instance variables, their way of defining attribute usage is different. Instance variable can be utilized directly or indirectly by methods. An instance variable is said to be directly used by a method M if it appears in the body of the method M. Likewise, an instance variable is said to be indirectly used, if it is directly used by another method M' which is called directly or indirectly by M. Two methods are said to be directly connected if they use a common attribute directly or indirectly.

TCC is defined as the percentage of pairs of methods that are directly connected. It measures the ratio between the actual numbers of visible directly connected methods in a class divided by the number of maximal possible number of connections between the visible methods of a class. LCC counts the pairs of methods that are directly or indirectly connected. Constructors and destructors are not considered for computing LCC and TCC. [0, 1] interval is the perpetual range of TCC and LCC. Bieman and Kang also propose three methods for calculating TCC and LCC:

1. Inclusion of inherited methods and inherited instance variables in the analysis

2. Exclusion of inherited methods and inherited instance variables from the analysis

3. Exclusion of inherited methods but inclusion of inherited instance variables from the analysis

They did not, however, give any special preference to any one of the three particular method of calculating their metrics. We preferred to opt for the first method i.e. considering inheritance as an intrinsic facet of Object Oriented systems, for evaluation. Because LCC is more comprehensive extension of TCC, we selected LCC in tandem with LCOM as the key cohesion metrics of our trialing procedure.

3. EMPIRICAL VALIDATION of COHESION and INSTANCE VARIABLE INDIRECT RELATIONSHIP

3.1 Objectives

A large number of software systems have longevity. With the passage of time, these software systems need improvisations vis-à-vis performance, adaptation to the dynamics of environment, and to address new requirements. We stressed on carrying out our experiment with regard to the changeability factor because our industrial collaborator has a deep-rooted awareness of software changeability. A good way of evaluating the changeability of a software system is to detect some design properties that can be utilized as changeability indicators. In the dominion of Object Oriented systems, experiments have been carried out that show coupling between classes as an indicator of changeability.

Chaumun and others defined a model of software changes and change impacts at the theoretical level. They detected a close link between changeability and some coupling metrics across diversified industrial systems and across diversified types of changes. Here again, no quantitative studies of class cohesion have been undertaken with respect to changeability. Weak class cohesion results in high coupling with the

rest of the system which in turn leads to high change impact. Poor changeability is thus, a result of weak class cohesion. One technique for investigating cohesion as a changeability indicator is to establish if low cohesion is in fact correlated to high coupling. Such a proof would substantiate our perception that there is indeed a correlation between cohesion and changeability. We are aware that this second hypothesis would entail a study in its own right, which is outside the purview of this paper.

4. COHESION METRICS AND INSTANCE VARIABLES INDIRECT USAGE

4.1 The Cohesion Metrics for Classes

Because of the increasing popularity of object-oriented methodology in industrialized and scholastic environs, much research has been conducted on Object-Oriented cohesion metrics. The most important existing cohesion metrics were assessed in [5] and abbreviated definitions of those metrics are presented in Table 1. Existing cohesion metrics are categorized into three groups according to the cohesion criterion espoused by its definition. The cohesion criterion is an indicator of what contributes to the cohesiveness of a class. Briand et al. [5] recapitulated that the cohesion criteria adopted by the existing cohesion metrics are both instance variable usage and sharing of instance variables. The criterion of instance variable usage indicates that cohesion is proportional to the number of instance variables in a class that are referenced by methods in the class. LCOM5 and Coh fall into this category. All other metrics excluding CBMC adopt the criterion of sharing of instance variables i.e. the bigger the number of method pairs of a class sharing instance variables is, the greater is the cohesiveness of the class.
Chae et al. [12] observe that these two criteria lead to a little discrepancy between the computed cohesion value and the intuitively expected value. To facilitate the resolution of these problems due to those criteria, Chae et al introduced member connectivity as a fresh criterion. According to this new criterion of member connectivity, a class with more robustly connected members is more cohesive. In this paper, discussion on the effects of dependent instance variables is concentrated on the metrics in Table 1 because they have been broadly acknowledged among the software engineering community; they have been used in many empirical studies for investigating the link between metrics and quality factors such as development/maintenance effort [1], [8], [15], [20], [21], fault-proneness [2], [6], [7], [17], [22], and testability [9]. Besides all these, their use is steadily increasing in industrial settings.

4.2 Indirect Usage Instance Variables:

We capture the state information common to the objects instantiated from the class from the instance variables in a class. This paper classifies instance variables into base instance variables and dependent instance variables. The values of dependent instance variables are calculated from other instance variables. In contrast, base instance variables values cannot be calculated from other instance variables; their values can be allotted only by the execution of methods in the class. As a result, we can obtain the values of the dependent instance variables from base instance variables and other dependent instance variables.

In this respect, dependent instance variables are redundant because they do not add any semantic information. But they are still made use of as they step up the understandability and performance of an object-oriented program. Additionally, UML presents data for describing attributes that can be derived from others. Examine the class Emp in Figure. 1. You can observe that class Emp has seven instance variables, out of which three are dependent instance variables. The value of gross instance variable can be computed from instance variables dow and ros. Likewise, the values of instance variables tax and netsal can be computes from instance variables gross and rot, and gross and tax, correspondingly. The dependency relation between the instance variables in class Emp is depicted in Figure. 2. Consider the fact that a method has using some instance variables indirectly along with base instance variables that can be clearly accessed. To put in simply, by accessing a dependent variable, a method interacts implicitly with its base instance variables in addition to the dependent variable. For instance, by referencing the dependent instance variable gross, method getGrossSalary() also interacts implicitly with its base instance variables dow and ros as the value of gross depends on those of dow and ros.

Also, when method getTaxAmount() interacts with instance variable tax, getTaxAmount interacts not only with instance variable tax but also with the tax's base instance variables gross

and rot. Figure. 3 depicts explicit and implicit interactions amongst the members of class Emp. A rectangle symbolizes a method and an ellipse stands for an instance variable. A solid edge flanked by a method and an instance variable shows that an explicit interaction exists between them because the method actually references the instance variable. A dotted edge illustrates an implicit interaction by means of dependency relation. As shown in Figure. 3, class Emp has 11 implicit interactions and 20 explicit interactions. According to us, in addition to explicit interactions that can be directly identified in a source code, implicit interactions also contribute to the cohesiveness of a class and, therefore, should be reflected in the definition of cohesion metrics. For example, if method getGrossSalary() references the instance variable gross, method.

```
class Emp{
private String name;
private int dow;//nuber of working days
private float ros; //rate of salary
private float rot; //rate of tax
private float netsal;//net salary
private float grsal; //gross salary
private float tax; //total tax to be paid

public Emp(String name,int dow,float ros,rot){
    this.name=name;
    this.dow=dow;
    this.ros=ros;
    this.rot=rot;
    netsal=0.0f;
    grsal=0.0f;
    tax=0.0f;
}
public void print(){
    System.out.println(name+"\t"+grsal+"\t"+netsal+"\t"+tax);
}
public void salary(){
    grsal=dow*ros;
    tax=grsal*rot;
    net=grsal-tax;
}
public float getGrossSalary(){
return grsal;
}
public float getNetSalary(){
    return netsal;
}
public float getTaxAmount(){
    return tax;
}

}
```

Figure 1: Class Emp

getGrossSalary() has a hidden relation with dow and ros because gross is computes from dow and ros. Thus, implicit interactions via dependencies between instance variables should be taking into consideration in order to more accurately appraise the cohesiveness of a class. Conversely, existing cohesion metrics have no notion of dependent instance variables and do not take into consideration those implicit interactions via dependent instance variables either. Because of this, they fail to accurately evaluate the cohesiveness of a class producing a cohesion value lower than expected. More specifically, metrics associated with the criterion of instance variable usage such as Coh and LCOM5 count explicit interactions but not implicit interactions. Observe that method getTaxAmount() has implicit interactions with dow, ros, and rot. However, their cohesion values are usually lower than anticipated because Coh and LCOM5 do not count those implicit interactions. Similarly, the metrics linked with the criterion of sharing of instance variables such as LCOM1, LCOM2, LCOM3, LCOM4, Co, LCC, and TCC do not consider the method pairs with indirectly shared instance variables because of implicit interactions.

For example, methods getGrossSalary() and getNetSalary() seem to have no common instance variables with respect to explicit interactions. However, those methods indirectly share instance variables dow, ros, and gross. Existing cohesion metrics such as LCOM1, LCOM2, Co, LCC, and TCC do not take account of indirect sharing of instance variables, thus, leading to a lower cohesion value. In the case of LCOM3 and LCOM4, although they are not inclusive of the effects of dependent instance variables, implicit interactions among instance variables are inadvertently reflected by their definitions. The connectivity factor of CBMC is defined as the ratio of the number of glue methods to the number of normal methods. Implicit interactions are ignored in computing glue methods i.e. a reference graph that is connected by implicit interactions is considered disjoint resulting in a smaller cohesion value.

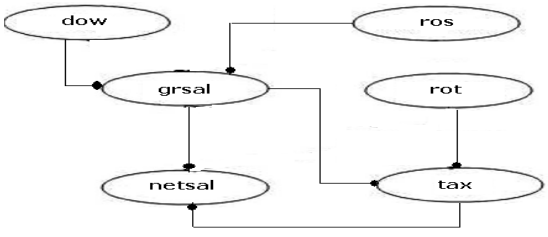

Figure 2: Indirect Usage Relation between Instance Variables of Class Emp

4.3 Consideration of Indirect Instance Variables Usage

In order to accurately evaluate the class cohesiveness, dependent instance variables are to be identified from the class, and implicit interactions via dependent instance variables are to be considered in calculating cohesion metrics. To evaluate the cohesiveness of a class considering dependent instance variables, our first step is to identify a dependency relation between instance variables. If the value of one variable is at least partly defined by that of the other variable, there is said to exist a dependency relation between those two variables. We can identify a dependency relation through investigating the implementation of each method in the class.

4.3.1 Indirect Usage of Instance Variable: Within the Method

Definition1: The dependency relation between variables in statement s of a method, denoted by $VIU_S(s)$, is a set of pairs of variables and is defined as follows: $VIU_S(s) = < vi; vj >| vi$, is a variable defined in s and vj is a variable used to define the variable vi in s. For example, consider class A shown in Figure. 4. Method Af1() has four assignment statements and a dependency relation occurs on each of them. That is,

$$VIU_S(s1) = \{< v,av1 >\}, VIU_S(s2) = \{< v,av2 >\}$$
$$VIU_S(s3) = \{< v,av3 >< x,av3 >\},$$
$$VIU_S(s4) = \{< av4, av2 >\}.$$

In the Object-Oriented literature, after a variable is declared, the variable can be explicitly initialized by means of an assignment statement. In this paper, a dependency relation is presumed to include initializations over and above the assignments.

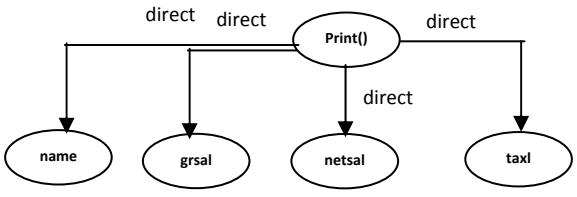

Figure 3a: Interactions among Instance variables of method Print ()

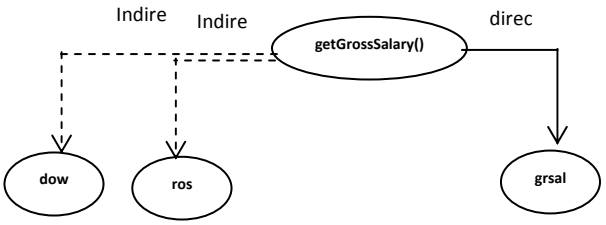

Figure 3b: Interactions among Instance variables of method getGrassSalary()

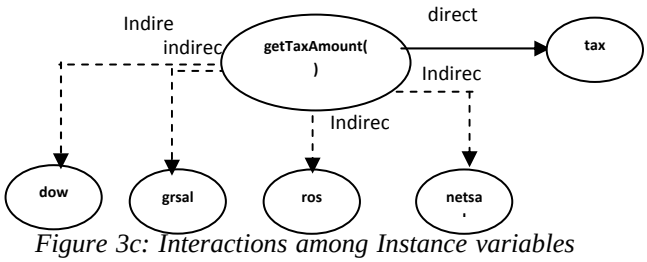

Figure 3c: Interactions among Instance variables of method getTaxAmount()

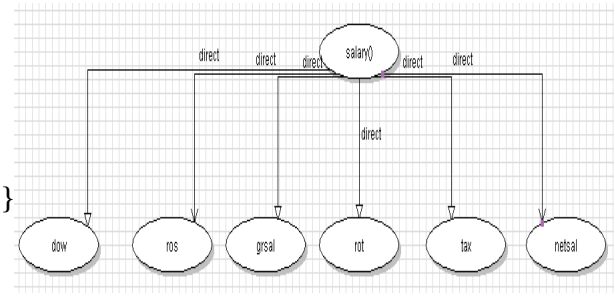

Figure 3d: Interactions among Instance variables of method Salary ()

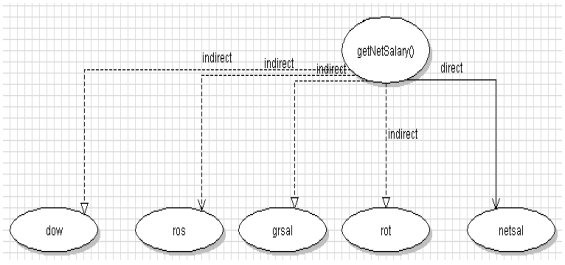

Figure 3e: Interactions among Instance variables of method getNetSalary()

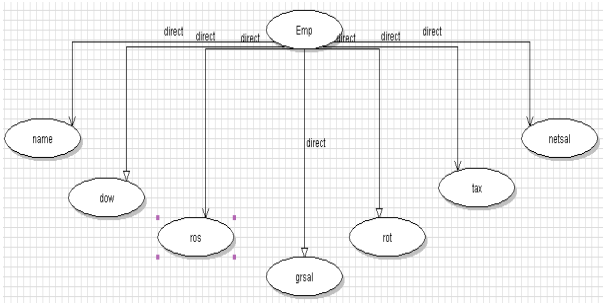

Figure 3f: Interactions among Members of Class Emp

```
class A{
    private int aa1;//instance variable 1 of class A
    private int aa2;//instance variable 2 of class A
    private int aa3;//instance variable 3 of class A

    public void am1(int x){//method one of class A
        int v=aa1;
        if(aa1>0) aa2=v;
        else
            aa3=v+x;
        aa4=aa2;
    }

    public void am2(){//method 2 of class A
        aa1=aa2;
    }
}
class B{
    private int ba;//instance variable 1 of class B

    public void bm1(int x){//method 1 of class B
        ba=x;
    }
    public void bm2(Integer x){//method 2 of class B
        x=new Integer(ba);
    }
    public int bm3(int x){//method three of class B
        return x+ba;
    }

}

class C{
    private int ca0,ca1,ca2//instance variables of class C
    private B bob;
    public void cm(){//method  of class C
        bob.bm1(ca1);
        bob.bm2(new Integer(ca2));
        ca1=bob.bm3(ca0);
    }

}
```

Figure 4: Classes to analyze indirect usage of instance variables

Definition 2: The relation based on instant variable indirect usage in method m, denoted by $VIU_M(m)$, is defined to consist of all the dependency relations in method m.

$VIU_M(m)$ can be computed by transitively collecting $VIU_S(s)$ for all the statements in the method m. For example, dependency relations in method1 of class A are

VIU_M(am1())={< v,aa1 >,< aa2, v >,< aa2, aa1 >,< aa4, aa2 >,< aa4, v >,< aa4, aa1 >,< aa3, v >,< aa3, x >,< aa3, aa1 >}:

4.3.2 Indirect Usage of Instance Variable Analysis: All Methods within a Class

Invocations of other methods contribute to dependencies among variables in a class. For instance, consider statement l1 of method cm() in class C shown in Figure. 4. Instance variable ca1 in class C is passed as an argument to method bm1 () of instance bob. A dependency relation arises between instance variable ba of instance bob and ca1 because of the assignment statement in method bm1(). As a consequence of this, instance bob depends on ca1. A method invocation can be bound at runtime to a number of methods in an object-oriented program due to polymorphism and dynamic binding. In this discussion, we have not concentrated on polymorphism because we cannot verify the method statically to which it will be bound. However, the method described herein can be extended by analyzing all the methods that can be invoked in a polymorphic way.

Let statement s be r = t.cm(a1; a2; . . . ; a_n). Assume variable t is an object of class T and cm(f1; f2; . . . ; fn) is a prototype of the called method cm. Let R(cm) be the set of variables that appear on a return statement of method cm and R`(m) the set of variables that can affect the return value of method cm. That is, $R`(m) = R(m) \bigcup \{v \mid \exists w \in R(cm). < v, w > \in VIU_M(cm)\}$.

We study three kinds of indirect instance variable usage approaches with respect to method invocations:

- A target object indirectly using actual arguments or vice versa

A target object indirectly using actual arguments if a formal argument fa corresponding to actual argument aa is used to change any instance variable of target object t, then the target object t depends on the actual argument aa.

$$\exists v \in V(T). < v, aa > \in VIU_M(cm) \rightarrow VIU_S(s) = VIU_S(s) \bigcup \{< t, aa >\}$$

For instance, for statement l1 of method cm(), the value of ba of target object bob is changed by actual argument ca1. Thus, VIUS (l1) = {<bob; ca1 >}.

- An actual argument indirectly using a target object.
  If fa as formal parameter corresponding to actual parameter aa is modified by any instance variable of target object tob, then the actual parameter aa depends on the target object tob.

$$\exists v \in V(T). < aa, v > \in VIU_M(cm) \rightarrow VIU_S(s) = VIU_S(s) \bigcup \{< aa, tob >\}$$

For instance, for statement l2 of function cm(), actual parameter ca2 is changed by instance variable ba of target object bob. Therefore, ca2 indirectly using bob. Thus, VUIS (s2) = {<ca2; bob >}. The return value of a called method is assigned. If any instance variable of target object tob affects the return value of method cm and the return value is assigned to r, then r depends on t.

$$\exists v \in V(T), w \in R`(cm). < v, w > \in VIU_M(cm) \rightarrow VIU_S(s) = VIU_S(s) \bigcup \{< r, tob >\}$$

For instance, for statement l3 of function cm(), instance variable ba of instance bob affects the return value of function bm3() and the return value of bm3() is assigned to ca1. Thus, ca1 indirectly using bob. VIU_S(l3) = {<ca1; bob >}. The return value affected by actual parameter is assigned. If a formal parameter fa corresponding to actual parameter aa affects the return value of method cm and the return value is assigned to r, then r depends on aa

$$\exists w \in R`(cm). < ai, w > \in VIU_M(cm) \rightarrow VIU_S(s) = VIU_S(s) \bigcup \{< r, aa >\}.$$

For instance, for statement l3 in function cm(), actual parameter ca0 affects the return value of function bm3() and the return value is assigned to ca1. Thus, ca1 indirectly using ca0. VIUS(l3) = {<ca1; bob >;< ca1; ca0 >}.

4.3.3 Instance Variables and their relations for Indirect Usage

Combining the indirect usage levels between instance variables in each method of the class will result in computing the indirect usage relation between instance variables in a class. The indirect usage relation also displays the following properties:

- The indirect usage relation is not true for reflexive: No instance variable indirectly using itself.
- The indirect usage relation is lopsided. If v using w indirectly, then w should not use v indirectly.
- The indirect usage relation is transitive. If v indirectly using w and w indirectly using x, then v by default using x indirectly.

Definition 3: Let iv (cl) and f(cl) be the sets of instance variables and methods in class cl, respectively, then an indirect usage relation between instance variables in class cl, denoted by IVIU(cl), is defined as follows:

$$IVIU_{(cl)} = \{< ai, aj > | < ai, aj > \in VIU_C(cl) \wedge i \neq j \wedge < aj, ai > \notin VIU_C(cl) \wedge aj \in V(cl)\}$$

$$where VIU_C(cl) = \bigcup_{m \in M(cl)} VIU_M(m)$$

That is, the indirect usage relation is determined by removing reflexive and non lopsided tuples.
For instance, the indirect usage relation of class C is
IVIU(c) = {<ca1; ca0 >;< ca2; bob >;< ca2; ca1 >;< ca2; ca0 >}

4.4 Empirical Model: A Case Study

Together with a Java class library, we carried out an exhaustive case study to analyze the importance of instance variables that are involving indirectly on cohesion measurement of object-oriented programs and show how we can deal with indirectly using instance variables in measuring cohesion values. We have also devised a cohesion measurement tool to computerize the method of measuring both the original cohesion metrics and the improved cohesion metrics that integrated the characteristics of dependent instance variables. The tool that we developed and used to measure the cohesion accepts a java

source program as input and extracts the class level information such as instance variables, methods, and interactions among them. The tool measures range of cohesion values by even considering the indirect usage instance variables. To perform this process initially it builds a data flow graph. An empirical study had been conducted on popular open source application freeCS. It was found out that most of classes have a rather small number of instance variables. These data mean that dependent instance variables are repeatedly used in a real system.

Figure. 7a shows the number of classes that are affected when we take into account dependent instance variables for each measure that we have studied. With the exception of LCOM3 and LCOM4, there exist one or more classes whose cohesion values are altered due to the action of dependent instance variables. Consideration of implicit interactions via dependent instance variables has no effect on LCOM3 and LCOM4. As earlier pointed out, although they did not mean to consider the characteristics of dependent instance variables, implicit interactions via dependent instance variables are coincidentally reflected by their definitions.

5. RESULTS ANALYSIS

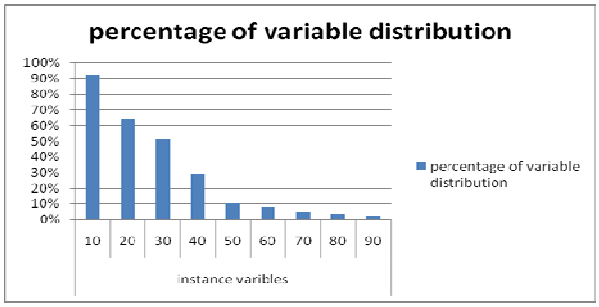

Figure 5a: Percentage of Variable Distribution (Bar Chart)

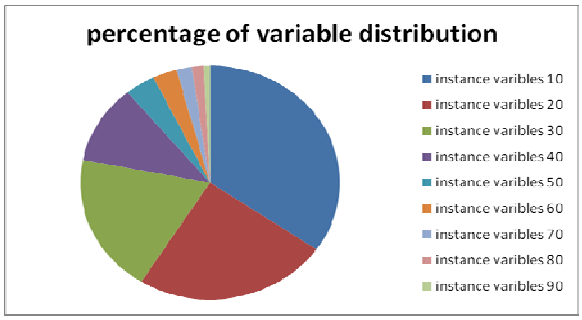

Figure 5b : Percentage of variable Disribution (Pie Chart)

5.1: Case 1: Inter class level indirect usage

Metric	Value
Lcom1	0.514523
iviulcom1	0.526151
Lcom2	0.52034
iviulcom2	0.522629
Lcom3	0.571424
iviulcom3	0.571424
Lcom4	0.562356
iviulcom4	0.562356
Co	0.587945
iviuCo	0.598763
Lcom5	0.6934
iviulcom5	0.720512
Metric	**Value**
Coh	0.59462
Iviucoh	0.609307
L cc	7.04
Iviulcc	7.121664
Tcc	5.31
Iviutcc	5.468238
Cbmc	8.01
Iviucbmc	9.752976

Table 1: The Increases of the Cohesion Measurement: interclass level

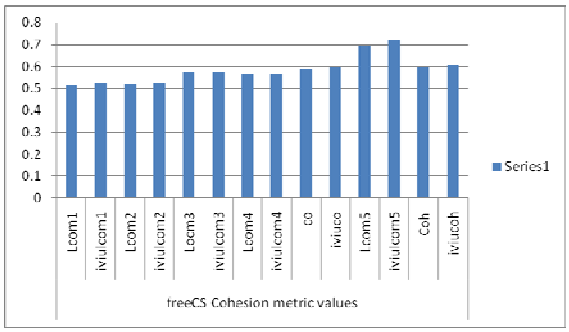

Figure 6a: shows the cohesion values for the cohesion measures for all classes in inter class state.

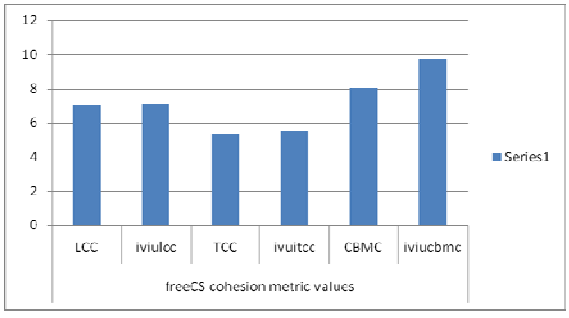

Figure 6b: shows the cohesion values for the cohesion measures for all classes in inter class state.

Table 1, Figure 6a and Figure 6b shows the cohesion values for the cohesion measures for all classes in inter class state. The more the relative increase, greater is the effect of the dependent instance variables to cohesion and describes the relative change in cohesion values for all classes in the Interviews. Barring CBMC, all the metrics have a small change because the percentage of classes whose cohesion values are affected by dependent instance variables is rather minute.

5.2: Case 2: Cohesion values of classes with multiple instance variables indirect usage:

Metric	Value
Lcom1	0.514523
iviulcom1	0.536082
Lcom2	0.52034
iviulcom2	0.52534323
Lcom3	0.571424
iviulcom3	0.571424
Lcom4	0.562356
Iviulcom4	0.562356
Co	0.587945
Iviuco	0.6316293135
Lcom5	0.6934
iviulcom5	0.7953731968
Coh	0.59462
Iviucoh	0.669720506
Lcc	7.04
Iviulcc	1.009536
Tcc	5.31
Iviutcc	5.944545
Cbmc	8.01
iviucbmc	8.744517

Table 2: The Increases of the Cohesion Measurement: multiple IVIU

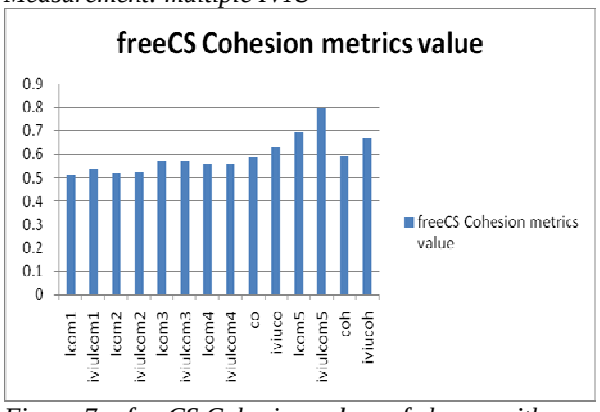

Figure 7a: freeCS Cohesion values of classes with multiple instance Variables indirect usage

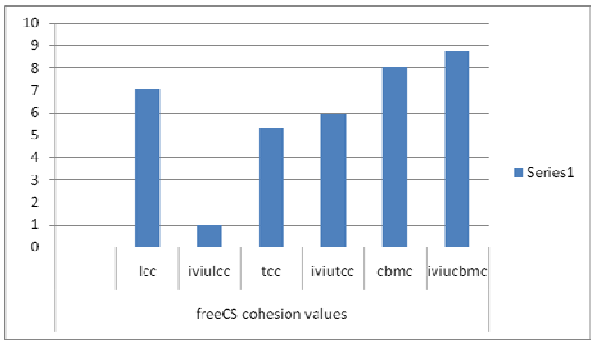

Figure 7b: freeCS Cohesion values of classes with multiple instance Variables indirect usage

This case specifies the relative change in cohesion values of classes which have multiple dependent instance variables. Except for CBMC, the changes in this case are larger than that in inter class model. This shows how many effects the dependent instance variables have on cohesion measurement since classes with no dependent instance variables are excluded.

5.3: Case 3: Cohesion values: Affected by indirect usage of instance variables

Metric	Value
Lcom1	0.514523
iviulcom1	0.6767006496
Lcom2	0.52034
iviulcom2	1.04068
Lcom3	0.571424
iviulcom3	0.571424
Lcom4	0.562356
iviulcom4	0.562356
Co	0.587945
Iviuco	0.683662446
Lcom5	0.6934
iviulcom5	0.953425
Coh	0.59462
Iviucoh	0.774433088
Lcc	7.04
Metric	**Value**
Iviulcc	42.24
Tcc	5.31
Iviutcc	5.944545
Cbmc	8.01
iviucbmc	10.791072

Table 3: Cohesion values: Affected by indirect usage of instance variables

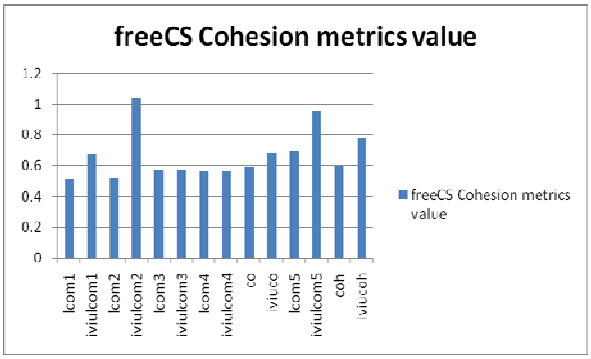

Figure 8a: freeCS Cohesion values: Affected by indirect usage of instance variables

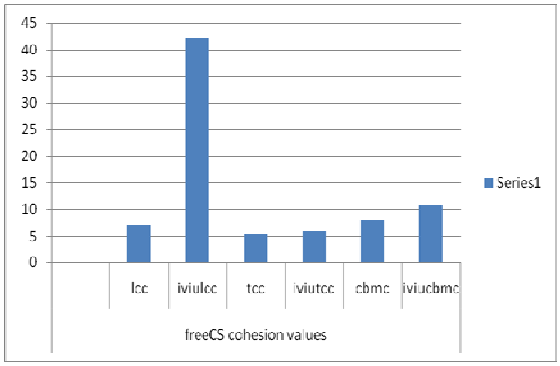

Figure 8b: freeCS Cohesion values: Affected by indirect usage of instance variables

This case concludes the relative change for classes whose cohesion values are affected by considering dependent instance variables. As seen in the Table 3, there is a sizable increase in cohesion values. The values for LCOM2 and LCC do not seem to be meaningful because, in the case of LCOM2 and LCC, a very small number of classes are influenced by dependent instance variables. Nonetheless, along with Case 2, Case 3 illustrates that dependent instance variables can have major outcomes on cohesion measurement. The case study was conducted by us with only one class library. Via this case study, however, we determined that dependent instance variables are usually encountered and, accordingly, the characteristics of dependent instance variables can to a great extent influence the cohesion measurement.

6. CONCLUSION & FUTUREWORK

In this paper, we scrutinized the effects of indirect instance variable usage on cohesion metrics for Object-Oriented programs and suggested a way to identify the dependency relations among instance variables. By way of performing a case study where many dependent instance variables were found and the cohesion values of many classes were clearly affected by considering dependent instance variables, we were able to show the importance of our approach. As part of the future endeavor, it is essential to probe empirically if advancement to cohesion metrics by considering dependent instance variables would yield improvements to the quality of classes. We will also need to explore the effects of dependent instance variables on coupling metrics.

REFERENCES

[1]. M. Alshayeb and W. Li, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes," IEEE Trans. Software Eng., vol. 29, no. 11, pp. 1043-1049, Nov. 2003.

[2]. V.R. Basili, L.C. Briand, and W. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 1996.

[3]. J.M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," Proc. Symp. Software Reusability, pp. 259-262, 1995.

[4]. G. Booch et al., The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[5]. L.C. Briand, J.W. Daly, and J.K. Wu¨ st, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," Empirical Software Eng., vol. 1, no. 1, pp. 65-117, 1998.

[6] .L.C. Briand, J.K. Wu¨ st, S. Ikonomovski, and H. Lounis, "Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study," Proc. Int'l Conf. Software Eng., pp. 345-354, 1999.

[7]. L.C. Briand, J.K. Wu¨ st, J.W. Daly, and D.V. Porter, "Exploring the Relationship between Design Measures and Software Quality in Object- Oriented Systems," J. Systems and Software, vol. 51, no. 3, pp. 245-273, 2000.

[8]. L.C. Briand and J.K. Wu¨ st, "Modeling Development Effort in Object-Oriented Systems Using Design Properties," IEEE Trans. Software Eng., vol. 27, no. 11, pp. 963-986, Nov. 2001.

[9]. M. Bruntink and A. Deursen, "Predicting Class Testability Using Object-Oriented Metrics," Proc. Fourth Int'l Workshop Source Code Analysis and Manipulation, 2004.

[10].N.N. Card, G.T. Page, and F.E. McGarry, "Criteria for Software Modularization," Proc. Eighth Int't Conf. Software Eng., pp. 372-377, 1985.

[11]. N.N. Card, V.E. Chruch, and W.W. Agresti, "An Empirical Study of Software Design Practices," IEEE Trans. Software Eng., vol. 12, no. 2, pp. 264- 271, Feb. 1986.

[12]. H.-S. Chae, Y.-R. Kwon, and D.-H. Bae, "A Cohesion Measure for Object- Oriented Classes," Software Practice and Experience, vol. 30, no. 12, pp. 1405-1431, 2000.

[13].S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object- Oriented Design," Proc. Sixth ACM Conf. Object-Oriented Systems, Languages and Applications, pp. 197-211, 1991.

[14]. S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June 1994.

[15]. S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object Oriented Software: An Explorary Analysis," IEEE Trans. Software Eng., vol. 24, no. 8, pp. 629- 39, Aug. 1998.

[16]. P. Devanbu, "GENOA a Customizable, Language and Front-End Independent Code Analyzer," Proc. Conf. Software Eng., pp. 307-317, 1992.

[17]. K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," IEEE Trans. Software Eng., vol. 27, no. 7, pp. 630-650, July 2001.

[18]. B. Henderson-Sellers, Software Metrics. Prentice Hall, 1996.

[19]. M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object- Oriented Systems," Proc. Symp. Applied Corporate Computing, 1995.

[20]. H. Kabaili, R.K. Keller, F. Lustman, and G. Saint-Denis, "Class Cohesion Revisited: An Empirical Study on Industrial Systems," Proc. Fourth Int'l ECOOP Workshop Quantitative Approaches in Object-Oriented Software Eng., pp. 29-38, 2000.

[21].H. Kabaili, R.K. Keller, and F. Lustman, "Cohesion as Changeability Indicator in Object-Oriented Systems," Proc. Fifth European Conf. Software Maintenance and Reeng., 2001.

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

[23].M. Linton, P.R. Calder, and J.M. Vlissides, "InterViews: A C++ Graphical Interface Toolkit," Technical Report CSL-TR-88-358, Stanford Univ., 1988, ftp://interviews.stanford.edu/pub.

[24]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design. Prentice Hall, 1991.

[25].W. Stevens, G. Myers, and L. Constantine, "Structured Design," IBM Systems J., vol. 12, no. 2, 1974.

[26]. R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implifications for Software Defects," IEEE Trans. Software Eng., vol. 29, no. 4, pp. 297-310, Apr. 2003.

AUTHOR PROFILES

M.V.Vijaya Saradhi is Currently Associated Professor in the Department of Computer Science and Engineering (CSE) at Aurora's Scientific, Technological and Research Academy, (ASTRA), Bandlaguda, Hyderabad, India, where he teaches Several Courses in the area of Computer Science. He is Currently Pursuing the PhD degree in Computer Science at Osmania University, Faculty of Engineering, Hyderabad, India.. He is a life member of various professional bodies like MIETE, MCSI, MIE, MISTE.

Dr. B. R. Sastry is currently working as Director, Astra, Hyderabad, India. He earlier worked for 12 years in Industry that developed indigenous computer systems in India. His areas of research includes Computer Architecture, Network Security, Software Engineering, Data Mining and Natural Language Processing, He is currently concentrating on improving academic standards and imparting quality engineering