INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

MEASURING CLASS COHESION IN OBJECT-ORIENTED SYSTEMS

BY

MUHAMMAD WASIQ

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

MAY 2001

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by Muhammad Wasiq

Under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE.

Thesis Committee

Dr. Jarallah S. AlGhamdi (Chairman)

Dr. Muhammad Sarfraz (Member)

Dr. Moataz A. Ahmad (Member)

Department Chairman

Dean of Graduate Studies

7/7/2001

Date

To my parents

ACKNOWLEDGEMENTS

First of all, all the praise goes to Allah (SWT) who is the most merciful and beneficent. Peace and blessing of Allah be upon the last Prophet Muhammad (PBUH). I am grateful to Allah (SWT) for providing me the courage, intelligence, perseverance and support of well-wishers to accomplish all the achievements that I have made so far in life.

The part of my parents has been very crucial in my life. They provided me with all the needs and comforts of life besides their love and care, which is irreplaceable. I thank them for their prayers and support.

I would like to extend my appreciation to my thesis committee chairman, Dr. Jarallah Al-Ghamdi, for his continuous advice, guidance and cooperation. I also feel grateful to my thesis committee members, Dr. Muhammad Sarfraz and Dr. Moataz Ahmed, for their useful suggestions and cooperation.

I also extend thanks to all the teachers who taught me at KFUPM. All of them have been very helpful and I gathered priceless knowledge from them. In fact, I would like to thank all

of my teachers since my childhood for parting the knowledge and wisdom I have received from them.

Thanks go to all of my colleagues and friends at KFUPM as I had a pleasant, enjoyable and fruitful company with them. Specially, I would like to thank Faisal Alvi for his suggestions and moral support.

Last but not the least, is the part of King Fahd University of Petroleum and Minerals in pursuing this work. I would like to acknowledge King Fahd University of Petroleum and Minerals in general and Department of Information and Computer Science in particular for providing the facilities and support to carry out this research. I feel proud to be a part of KFUPM and I have learned a lot here. I hope that this prestigious institution will maintain its excellence in research and academics in years to come.

CONTENTS

2 EXISTING COHESION METRICS

3 COHESION AS A QUALITY FACTOR

4 AUTOMATED METRICS CALCULATION

5 ANALYZING LARGE OBJECT-ORIENTED SYSTEMS

LIST OF TABLES

LIST OF FIGURES

Thesis Abstract

Name: Muhammad Wasiq

Thesis Title: Measuring Class Cohesion in Object-Oriented Systems

Major Field: Computer Science

Date of Degree: May 2001

Cohesion is an important quality factor of the object-oriented as well as imperative design. A class in object-oriented software can have two types of cohesion: Class Cohesion and Inheritance Cohesion. In object-orientation it is a basic design requirement that a class should represent a single real world entity. To measure the extent to which a class meets this requirement, class cohesion is used as a tool. In this work, for the first time, we have proposed ways of using inheritance and class cohesion for measuring the quality of the inheritance hierarchy. We propose five graphical/visual cohesion-related metrics that provide designers of the object-oriented systems with the guidelines to enhance the quality of inheritance hierarchy to improve its maintainability, understandability and reusability. We also propose three new metrics to measure the cohesion of a class, i.e., CCM, CCCM and MCCM. We believe that CCM captures the connections among the methods of a class quite well and gives a good measure of class' cohesion. In the form of MCCM, we have proposed a metric that also takes into account the effect of overridden methods on inheritance cohesion of a class.

We have augmented our theoretical work by implementing the automated tool using which a designer can readily analyze his software against our proposed metrics. Finally as a test case study, we have analyzed the Java 1.3 API. Our analysis has revealed some very interesting quality features of the Java API.

King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

May 2001

خلاصة الرسالة

الاســـــــــم : محمد واثق
عنوان الرسالة : قياس ضم الأصناف والنظم الموجهة شيئياً
الدرجة العلمية : ماجستير
التخصـــــــص : علوم الحاسب الآلي
التاريـــــــخ : مايو ٢٠٠١م

الضم عامل جودة مهم بالنسبة لكل من التصميم الموجه شيئياً والتصميم المهم. في البرامج الموجهة شيئياً يمكن أن يكون للصنف نوعان من الضم : ضم الصنف وضم الوراثة يتطلب التصميم الموجه شيئياً أن يمثل الصنف موضوعاً حقيقياً واحداً بالضرورة. يستخدم ضم الصنف كأداة لقياس مدى ملاقاة الصنف لهذا المتطلب. هذا البحث يقترح طريقاً محدثة لقياس هيكل الوراثة باستخدام ضم الوراثة وضم الصنف. تم اقتراح خمس منظومات رسومية/بصرية متعلقة بالضم لتزويد مصمم الأنظمة شيئياً بمعالم لتحسين جودة الوراثة وتحسين قابليتها لكل من الصيانة والفهم واعادة الاستخدام كما تم اقتراح ثلاث منظومات جديدة لقياس ضم الصنف هي MCC ، CCCM و MCCM. تعتقد أن CCM تستوعب الروابط بين طرق الصنف بصورة مرضية وتقدم مقياساً جيداً لضم الصنف. على شاكلة MCCM اقترضنا منظومة تأخذ في الاعتبار تأثير الطرق المسودة على ضم الصنف.

تم توسيع العمل النظري بتطبيق الاستخدام التلقائي للأداء والذي يمكن المصمم من تحليل برامجه مقابل مصفوفاتنا المقترحة. أخيراً تم تحليل تطبيق (Java 1.3 AP). أظهرت الدراسة عدة جوانب جودة جاذبة بالنسبة لـــ Java AP1 .

جامعة الملك فهد للبترول والمعادن
الظهران
مايو ٢٠٠١م

Chapter 1

INTRODUCTION

Object-Orientation aims to model the real world as closely to a user's perspective as possible. Classes play an essential role in the object-oriented development. Entities in an application domain are captured as classes, and applications are built with the objects that are instantiated from them. Classes serve as a unit of encapsulation; that is, instances of a class can be manipulated only through the interface defined in the class. Therefore, internal representation of the classes can be changed without affecting any client as long as the new representation conforms to the interface. In other words, compatible changes can be made safely on classes, which facilitate program evolution and maintenance.

In order to take the full advantage of the desirable features provided by the classes, such as data abstraction and encapsulation, classes should be designed to have a good quality.

Otherwise, classes of bad quality can be a serious obstacle to the development of systems because object-oriented systems are often developed by reusing existing classes.

Cohesion that originated from structured design refers to the degree of relatedness among the elements of a class [Eder94], i.e., to what extent the elements of a class are concentrated to a single task. Before diving into the details of cohesion, we would like to review some of the basic object-oriented systems concepts that will be used through out this report.

1.1 Basic Concepts of Object-Oriented Systems

To be able to talk about cohesion of object-oriented systems we have to identify the basic building blocks of such systems and their possible relationship in advance. An object-oriented language should support the basic concepts of an object, a class, and inheritance [Wegner87].

The first appearance of object-oriented programming was in late 1960s in SIMULA 67 [Sebe93]. The popularity of object-oriented paradigm has increased in recent years due to its promises of portability, reusability, maintainability, etc. [Bucc98]. In the following subsections, the basic concepts of this paradigm will be described.

1.1.1 Object and Object Class

An *object* consists of a set of instance variables representing the internal state of the object and a set of methods representing the external behavior of the object. To put it in other words, an object is an encapsulation of state and behavior.

An *object class* can be seen as a kind of template specifying state and behavior of a set of similar objects, which are created as instances of the object class by some create method during run-time of a program. Object classes are the basic means of object-oriented design and development. The definition of the object class is based on the concept of *encapsulation* following the abstract data type approach and on the principle of *information hiding* distinguishing between visible and hidden parts of an object class' definition. The visible part is called the *specification* or an interface of an object class and in general consists of a set of method specifications. The hidden part is called the *implementation* of an object class and consists of the implementation of methods and the definition of the instance variables.

1.1.2 Inheritance and Inheritance Hierarchy

An object class may be related to other classes by an *inheritance relationship*, in which case it inherits instance variables and methods from them. An object class C related to object class C' by an inheritance relationship is called *direct subclass* of C'. An object class

C is an *indirect subclass* of C' if there exists some class $\hat{C}$ such that C is a direct subclass of $\hat{C}$ and $\hat{C}$ is a direct or indirect subclass of C'. An object class C is called *subclass* of object class C', if C is a direct or indirect subclass of C'. Conversely, an object class C' is called *superclass* of object class C, if C is a subclass of C'. The inheritance relationship is transitive, reflexive, and anti-symmetric [Nier89]. The directed acyclic graph built up by the inheritance relationship is called the *inheritance hierarchy*.

1.1.3 Message Passing

An object O communicates with an object O' by sending a message to O'. Adhering to the principle of information hiding *message passing* is the only means to access and alter an object's state [Nier89]. Message passing implies a second kind of relationship, the interaction relationship. The interaction relationship is defined for methods in the first place, and deduced for object classes for which the methods are specified in turn.

1.1.4 Overriding, Polymorphism and Dynamic Binding

Overriding refers to the redefinition of inherited instance variables and methods in subclasses. *Polymorphism* means that the same method may be invoked on objects of different classes. *Dynamic Binding* means that the binding between method invocation and code to be executed takes place during run-time and depends on the actual class of the object on which the method is invoked. The *static class* of a variable is the domain of this

variable defined at the compile time. In contrast, the *dynamic class* of a variable is the actual class of the object referenced by the variable during run-time. Due to polymorphism a variable with the static class being object class C may reference members of C during runtime.

We say that a method m is *implemented* at object class C, if it is initially defined at C, or its signature and/or implementation have been overridden at C. We further say that a method m is *declared* at object class C, if it is inherited by C but not overridden.

1.2 Cohesion in Object-Oriented Systems

Cohesion has been defined in the realm of procedure-oriented systems as "the degree of connectivity among the elements of a single module" [Stevens74]. Cohesion has been recognized as one of the most important software quality criteria. Modules with strong cohesion are easier to maintain, and furthermore, they greatly improve the possibility for reuse [Embley88]. A module has strong cohesion if it represents exactly one task of the problem domain, and all its elements contribute to this single task. Elements of a module are statements, sub-functions and possibly other modules. We recall that object-oriented counterparts of a module are methods and classes. The elements of a method are statements, local variables and also instance variables since they are accessed either directly or via access functions in the methods. Next to methods also object classes have to be analyzed. The elements of an object class are methods and instance variables. Thus

we have to distinguish the cohesion of a method from the cohesion of an object class. Thus the following kinds of cohesion may be defined for object-oriented systems [Eder94]:

- Method cohesion
- Class cohesion
- Inheritance cohesion

In the following sections, we will discuss the degrees of cohesion for all the above-mentioned cohesion classes as described in [Eder94].

1.2.1 Method Cohesion

Eder et al [Eder94] applied Myers' classical definition of module cohesion [Myers78] to methods. They defined seven degrees of method cohesion, based on the definition provided by Myers [Myers78] for modules. From weakest to strongest, the degrees of method cohesion are:

***Coincidental*:** The elements of method have nothing in common besides being within the same method.

***Logical*:** Elements with similar functionality such as input/output handling are collected in one method.

***Temporal*:** The elements of a method have logical cohesion and are performed at the same time.

***Procedural*:** The elements of a method are connected by some control flow.

***Communicational*:** The elements of a method are connected by some control flow and operate on the same set of data.

***Sequential*:** The elements of a method have communicational cohesion and are connected by a sequential control flow.

***Functional*:** The elements of a method have sequential cohesion, and all elements contribute to a single task in the problem domain. Functional cohesion fully supports the principle of locality and thus minimizes the maintenance efforts.

1.2.2 Class Cohesion

Class cohesion addresses the relationship between the elements of a class. The elements of a class are its non-inherited methods and non-inherited attributes. Eder et al [Eder94] define five degrees of class cohesion. From weakest to strongest these are:

***Separable*:** The objects of a class represent multiple unrelated data abstractions. For instance, the cohesion of a class is separable, if the methods and attributes can be grouped into two sets such that any method of one set invokes no methods and references no attributes of the other set, and vice versa.

***Multifaceted*:** The objects of a class represent multiple related data abstractions. The relation is caused by at least one method of the class that uses all these data abstractions. The cohesion of a class is rated multifaceted if the set of instance variables of the class interpreted as relation schema is not in second normal form. For example, consider the following class

```
Class Reorder{
      Item reorderedItem;
      Company reorderedFrom;
      int discount;
      int quantity;
      .........
      .........
      public bool expectedRevenue(){
            .........
     }
      .........
      .........
}
```

The method *expectedRevenue* computes the revenue expected by determining the difference between the price of an item and the discount given by the company and by multiplying this difference with the quantity of the reordered item.

If we interpret the set of instance variables as attributes of a relational schema, attributes *reorderedItem* and *reorderedFrom* form the key of this relation schema. Assume, the discount given depends only on the company, i.e., it is the same for all items. Thus, the second normal form is violated as the attribute discount is only partially related to the key,

i.e., it is only related to the *reorderedFrom* attribute of the key. Therefore, the class *Reorder* has multifaceted cohesion.

Non-Delegated: There exist attributes that do not describe the whole data abstraction represented by a class, but only a component of it. Hence, we may again use data normalization theory to detect non-delegated cohesion like we did for the analysis of multifaceted cohesion. We can say that the cohesion of a class is rated non-delegated if the set of instance variables interpreted as relation schema is not in a third normal form. For example, consider the following class

```
Class Employee {
      String name;
      Date birthDate;
      Project involvedInProject;
      Employee managerOfProject;
      .........
      public float computeSalary() {
            .........
            .........
      }
      .........
}
```

If we interpret the set of instance variables as attributes of a relation schema, the attribute *name* is the key of this relation schema. The attributes *birthDate* and *involvedInProject* depend directly on attribute *name*. However, the attribute *managerOfProject* depends directly on the project referenced by *involvedInProject* and transitively on *name*, thus the third normal form is violated. This definition of class *Employee* has non-delegated cohesion.

Concealed: There exists some useful data abstraction concealed in the data abstraction represented by the class. Consequently, the class includes some attributes and methods that might make another class. For instance, consider a class *Employee* having, amongst others, attributes *DayOfBirth*, *MonthOfBirth*, *YearOfBirth*, *DayOfHire*, *MonthOfHire*, and *YearOfHire*. These attributes describe a concealed data abstraction "date". In this case, we can define a new class *Date* with attributes *Day*, *Month* and *Year*, and replace the date attributes in class *Employee* by two attributes *BirthDate* and *HireDate* of type *Date*.

1.2.3 Inheritance Cohesion

Like class cohesion inheritance cohesion addresses the relationships between the elements of a class. However, inheritance cohesion takes all the methods and attributes of a class into account, i.e., inherited and non-inherited. Inheritance cohesion is strong if inheritance has been used for the purpose of defining specialized children classes. Inheritance cohesion is weak, if it has been used merely for the purpose of sharing code among otherwise unrelated classes. The degrees of inheritance cohesion are the same as those for class cohesion.

Chapter 2

EXISTING COHESION METRICS

In this chapter we present some of the important measures that have been proposed by researchers for measuring cohesion in object-oriented paradigm. During the last decade, quite a few number of class cohesion metrics have been proposed. Chidamber and Kemerer [Chid91, Chid94], Hitz and Montazeri [Hitz96], Bieman and Kang [Biem95], Henderson-Sellers [Hend96] and Briand et al. [Briand99] each have proposed different approaches to measure cohesion in object-oriented systems and defined various cohesion measures accordingly. We will discuss these measures in the following sections.

2.1 Approach by Chidamber and Kemerer [Chid91, Chid94]

Chidamber and Kemerer have proposed a cohesion measure *LCOM* (Low Cohesion Measure) defined as follows in [Chid91]:

Consider a class C with methods M_1, M_2,....., M_n. Let $\{I_i\}$ = set of attributes used by method M_i. There are n such sets, i.e., $\{I_1\}$, $\{I_2\}$,....., $\{I_n\}$.

$LCOM(C)$ = The number of disjoint sets formed by the intersection of n sets.

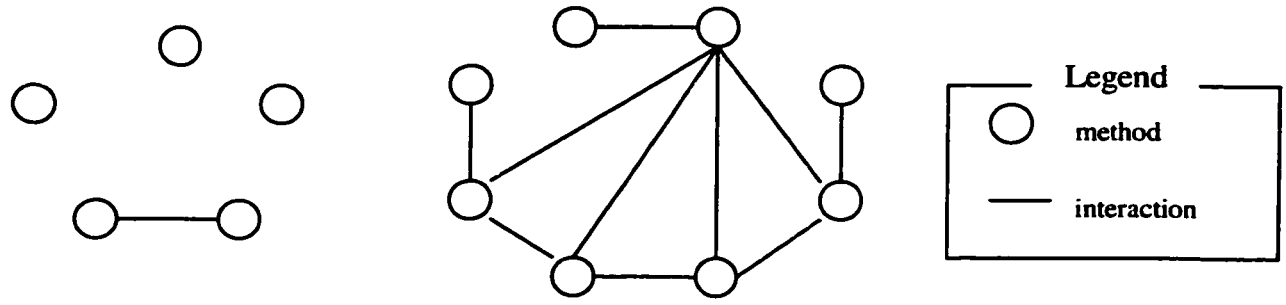
LCOM is an inverse cohesion measure. A high value of *LCOM* indicates low cohesion and vice versa.

In [Chid94], Chidamber and Kemerer have given the following new definition of *LCOM*.

Consider a class C with methods M_1, M_2,....., M_n. Let $\{I_i\}$ = set of attributes used by method M_i. There are n such sets, i.e., $\{I_1\}$, $\{I_2\}$,....., $\{I_n\}$. Let $P = \{ (I_i, I_j) \mid I_i \cap I_j = \varnothing \}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$. If all n sets $\{I_1\}$, $\{I_2\}$,....., $\{I_n\}$ are $\varnothing$ then let $P = \varnothing$.

$$LCOM(C) = \begin{cases} |P| - |Q|, \text{ if } |P| > |Q| \\ 0, \text{ otherwise} \end{cases}$$

By above definition, *LCOM* is the number of pairs of methods in a class having no common attribute references, $|P|$, minus the number of pairs of methods having common attribute references, $|Q|$. However, if $|P| < |Q|$, *LCOM* is set to zero.



(a) Class C (b) Class D

Figure 2.1: Two different classes with equal values for LCOM

One of the drawbacks of *LCOM* is that it is not normalized. There is no upper limit of values that this measure can take. Normalization is intended to allow for comparison of the cohesion of classes and systems of different size. Without normalization, this is not possible. Furthermore, *LCOM* is known to have little discriminating power. This is partly due to the fact that *LCOM* is set to zero whenever there are more pairs of methods that use an attribute in common than pairs of methods that do not. As a result, *LCOM* is zero for a large number of classes [Basi96]. But also for classes where *LCOM* is greater than zero, the measure is not discriminating. Consider the example classes C and D in figure 2.1, proposed by Henderson-Sellers [Hen96]. For class C, we have $|P| = 9$, $|Q| = 1$, and thus $LCOM(C) = 8$. For class D, $|P| = 18$, $|Q| = 10$, and so $LCOM(D) = 8$. Both classes have the same *LCOM* value, but we would intuitively say that class D is more cohesive than class C.

2.2 Approach by Hitz and Montazeri [Hitz95_2, Hitz96]

Hitz and Montazeri based their approach to measure cohesion on the work of Chidamber and Kemerer. They interpreted the definition of *LCOM* as follows [Hitz96]:

> Let X denote a class, I_x the set of its attributes, and M_x the set of its methods. Consider a simple undirected graph $G_x(V, E)$ with $V = M_x$ and $E = \{(m, n) \in V \times V \mid \exists I \in I_x: (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$.
>
> $LCOM(C)$ = Number of connected components of G_x.

Hitz and Montazeri identified a problem with the *access methods* for *LCOM*. An access method provides read or write access to an attribute of the class. Access methods typically reference only one attribute, namely the one they provide access to. If other methods of the class use the access methods, they may no longer need to directly reference any attributes at all. These methods are then isolated vertices in graph G_x. Thus, the presence of access methods artificially decreases the class cohesion as measured by *LCOM*. To remedy this problem, Hitz and Montazeri propose a second version of their *LCOM* measure. In this version, the definition of G_x is changed as follows: there is also an edge between vertices representing methods m_1 and m_2, if m_1 invokes m_2 or vice versa. The new definition of *LCOM* can be given as follows:

Let X denote a class, I_x the set of its attributes, and M_x the set of its methods. Consider a simple undirected graph $G_x(V, E)$ with $V = M_x$ and $E = \{(m, n) \in V \times V \mid (\exists I \in I_x: (m \text{ accesses } i) \wedge (n \text{ accesses } i)) \vee (m \text{ invokes } n) \vee (n \text{ invokes } m)\}$.

$LCOM(C)$ = Number of connected components of G_x.

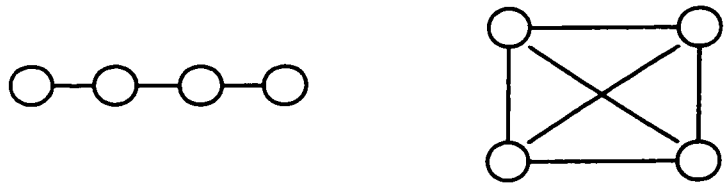
In the case where G_x consists of only one connected component, i.e., $LCOM = 1$, the number of edges $|E|$ ranges between $|V| - 1$ (minimum cohesion) and $|V|.(|V| - 1)/2$ (maximum cohesion). Hitz and Montazeri defined a measure C ("connectivity") [Hitz96] which further discriminates classes having $LCOM = 1$ by taking into account the number of edges of the connected component.

Hitz and Montazeri defined C as follows:

$$C(c) = 2 . \frac{|E_c| - (|V_c| - 1)}{(|V_c| - 1). (|V_c| - 2)}$$

Where E_c and V_c are the edges and vertices of the connection graph of the class c.

We always have $C(c) = [0,1]$. Values 0 and 1 are obtained for $|E_c| = |V_c| - 1$ and $|E_c| = |V_c|.(|V_c| - 1)/2$, respectively.



(a) Class A (b) Class B

Figure 2.2: Two different classes with equal values for the *LCOM* version of Hitz and Montazeri [Hitz96].

As pointed out by Hitz and Montazeri [Hitz96] themselves, their version of *LCOM* also exhibits a little discriminating power. For example, figure 2.2 shows two classes, *A* and *B*, both of which have value of *LCOM* equal to *1*. Although intuitively class *B* should be more cohesive than class *A* because class *B* has tighter interactions than class *A*. To overcome this problem Hitz and Montazeri [Hitz96] proposed *C* (connectivity) metric, but it can only be applied to classes whose connection graphs have only one connected component. But if the connection graph has more than one connected components, *C* metric can give negative values [Briand98]. Therefore, *C* metric can only be used in conjunction with *LCOM*, which is a great limitation for its use.

2.3 Approach by Bieman and Kang [Biem95]

The approach by Bieman and Kang to measure cohesion too is based on that of Chidamber and Kemerer's. They also consider pairs of methods that use common attributes. They have defined two different cohesion measures based on the direct and indirect connectivity between pairs of methods. Two methods that use one or more common attributes are said to be *directly connected.* Whereas, two methods that are connected through other directly connected methods are called *indirectly connected.* The indirect connection relation is the transitive closure of the direct connection relation. Thus, a method M_1 is indirectly connected with a method M_n if there is a sequence of methods M_2, M_3, ... M_{n-1} such that

$$M_1 \, \delta M_2, \ldots\ldots M_{n-1} \, \delta M_n$$

Where $M_i \, \delta M_j$ represents a direct connection.

Let $NDC(C)$ be the number of pairs of directly connected methods of a class C, $NIC(C)$ be the number of pairs of indirectly connected methods of C and $NP(C)$ be the maximum possible number of connections in C. It is clear that for a class with N methods, $NP(C) = N \,.(N - 1) \,/\, 2$.

Tight Class Cohesion (*TCC*) is defined to be a ratio of the number of pairs of directly connected methods in a class, $NDC(C)$, to the maximum possible number of connections in a class, $NP(C)$.

$$TCC(C) = \frac{NDC(C)}{NP(C)}$$

Loose Class Cohesion (*LCC*) is defined to be a ratio of the sum of the number of pairs of directly connected methods, $NDC(C)$, and number of pairs of indirectly connected methods, $NIC(C)$, in a class C to the maximum possible number of connections in C, $NP(C)$.

$$LCC(C) = \frac{NDC(C) + NIC(C)}{NP(C)}$$

With respect to inheritance, Bieman and Kang have stated three options for the analysis of cohesion of a class:

(a) Exclude inherited methods and inherited attributes from the analysis, or

(b) Include inherited methods and inherited attributes in the analysis, or

(c) Exclude inherited methods but include inherited attributes.

Bieman and Kang identified a problem with constructor methods for *TCC* and *LCC*. A class constructor is an initialization function. It generally accesses all attributes in the class,

and thus, shares attributes with virtually all other methods. Constructors create connections between methods even if the methods do not have any other relationships. Therefore, the presence of a constructor method artificially increases cohesion as measured by *TCC* and *LCC*. Bieman and Kang have therefore recommended to exclude constructors (and also destructors) from the analysis of cohesion [Biem95].

(a) Class A (b) Class B

Figure 2.3: Two different classes with equal values for TCC

Although *TCC* has better discriminatory power as compared to *LCOM* measures, but still it doesn't recognize the interaction patterns completely. For example, Consider two classes, *A* and *B*, as shown in figure 2.3 (a) and (b), respectively. The *TCC* values for *A* and *B* can be calculated as follows:

$$TCC(A) = 2.\frac{4}{(5).(4)} = 0.4$$

$$TCC(B) = 2.\frac{4}{(5).(4)} = 0.4$$

Although the values of $TCC(A)$ and $TCC(B)$ are equal, intuitively class B is more cohesive as its connection graph represents a single connected component, whereas, the connection graph of A is divided into two connected components.

2.4 Approach by Henderson-Sellers [Hend96]

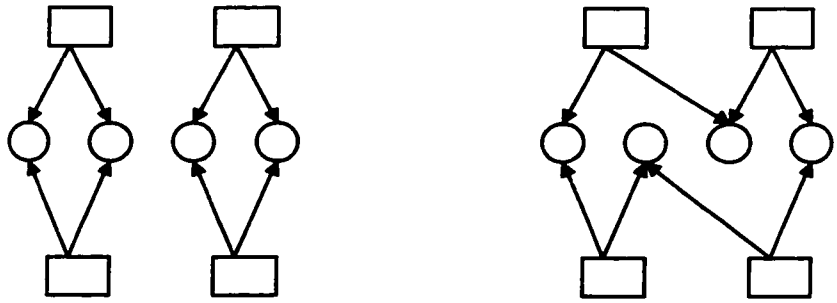
Henderson-Sellers set out to define a cohesion measure having the following properties:

- The measure yields 0, if each method of the class references every attribute of the class (this situation is called "perfect cohesion" by Henderson-Sellers").
- The measure yields 1, if each method of the class references only a single attribute.
- Values between 0 and 1 are to be interpreted as percentages of the perfect value.

Henderson-Sellers propose the following measure satisfying the above properties:

Consider a set of methods $\{M_i\}$ $(i = 1,\ldots..., m)$ of a class C accessing a set of attributes $\{A_j\}$ $(j = 1,\ldots..., m)$. Let the number of methods which access an attribute A_j be $\mu(A_j)$ and total number of attributes in $\{A_j\}$ is a. $LCOM$ is defined as follows:

$$LCOM^*(C) = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{1 - m}$$



(a) Class A (b) Class B

Figure 2.4: Two different classes with equal values for LCOM verson of Henderso-Sellers [Hend96].

Like other *LCOM* measures (by [chid94] and [Hitz96]), *LCOM* version of Henderson-Sellers [Hend96] is also weak in distinguishing between different interaction patterns. For example, consider classes *A* and *B* as shown in figure 2.4 (a) and (b), respectively. Although both *A* and *B* have similar value of *LCOM* (i.e., 2/3), intuitively class *B* is more cohesive than class *A* as it is connected, whereas, *B* is disjoint.

2.5 Approach by Briand et al. [Briand99]

Briand et al. proposed a cohesion measure in [Briand99] that is based on the visualization of a class as a collection of *data declarations* and *methods*. Data declarations are (i) local type declarations, (ii) the class itself (as an implicit public type), and (iii) public/private attributes (including constants). Briand et al. defined two types of interactions, *DD-interactions* (declaration-declaration interactions) and *DM-Interactions* (declaration-method interactions).

DD-interaction: A data declaration *a* DD-interacts with another data declaration *b*, if a change in *a*'s declaration or use may cause the need for a change in *b*'s declaration or use. We say that there is a *DD-interaction* between *a* and *b*. Following are the examples of *DD-interactions*:

- If the definition of a type *t* uses another public type *t'*, there is a DD-interaction between *t'* and *t*.
- If the definition of a public attribute *a* uses a public type *t*, there is a DD-interaction between *t* and *a*.
- If a public attribute *a* is an array and its definition uses public constant *a'*, there is a DD-interaction between *a'* and *a*.

DD-interactions need not be confined to one class. There can be DD-interactions between attributes and types of different classes. The DD-interaction relationship is transitive. If *a* DD-interacts with *b* and *b* DD-interacts with *c*, then *a* DD-interacts with *c*.

DM-interaction: Data declarations can also interact with methods. There is a *DM-interaction* between a data declaration *a* and method *m* either

- if *a* DD-interacts with at least one data declaration of *m* (Data declarations of methods include their parameters, return type and local variables), or
- if *a* is an attribute and *m* uses/accesses it.

Following are the examples of *DM-interactions*:

- If a method *m* of class *C* takes a parameter of type class *C*, there is a DM-interaction between *m* and the implicit type declaration of class *C*.
- If a method *m* uses an attribute *a*, there is DM-interaction between *a* and *m*.

Briand et al. defined *CI*(*C*) (*CI* for cohesive interactions) to be the set of all DD- and DM-interactions present in the class *C* and *Max*(*C*) to be the set of all possible DD- and DM-interactions that can be established in class *C*.

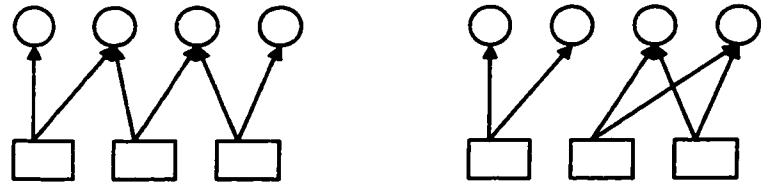
RCI can be defined as follows:

$$RCI(c) = \frac{|CI(c)|}{|Max(c)|}$$

RCI ranges between 0 and 1, where values 0 and 1 indicate minimum and maximum cohesion, respectively.

RCI metric is not good in distinguishing among different patterns of interactions [Chae98]. For example figure 2.5 (a) and (b) show the interactions among the members of classes *A* and *B*, respectively. Both of them have the same number of possible interactions (i.e. 12) and actual interactions (i.e., 6). According to the definition of *RCI*, they have the same cohesion value (i.e., 6/12). However, these classes show the distinct patterns of

interactions; the interaction graph of class *A* is connected, but that of class *B* is disjoint. From the definition of cohesion, relatedness among the members of a class, class *A* should be considered more cohesive than class *B*. This discrepancy originates from the fact that *RCI* depends only on the number of interactions and does not consider their pattern.



(a) Class A (b) Class B

Figure 2.5: Different classes with same value of *RCI*.

2.6 Conclusions Drawn from the Review

Although quite a few cohesion metrics have been proposed for measuring class cohesion in object-oriented systems, we feel that these metrics are not enough to measure the effects of cohesion on the quality of object-oriented software design due to the following reasons:

- Firstly, none of these measures addresses the effects of inheritance on class cohesion. Although Bieman and Kang [Biem96] have presented three options regarding inheritance (discussed in section 2.3), they haven't discussed about the implications of inheritance on class cohesion as measured by their metrics (*LCC* and *TCC*) and they haven't also provided any guidelines for using those options based on the requirements of the software being measured.

- Secondly, the authors of these metrics do not discuss that what quality attribute of object-oriented software is targeted by their measures, i.e., they haven't mentioned whether the values given by their metrics show the reusability potential of the class or its ease of maintainability, etc.
- Thirdly, these metrics do not have enough discriminatory power to distinguish between different patterns of interactions. They only count the number of interactions among the modules of a class.

Chapter 3

COHESION AS A QUALITY FACTOR

A good inheritance hierarchy is one in which a super-class is used as a generalization of its subclasses and subclasses are used as the specialized cases of the super-class [Eder94]. On the other hand, one can misuse the inheritance hierarchy just for sharing code among unrelated classes, in such a way that the super-class is not at all conceptually related to its subclasses and also, there exits no semantic association among the subclasses [Hitz95_1]. This type of misuse leads to a poor overall understandability of the system, which in turn makes the system difficult to maintain and reuse. Therefore, it is imperative to have metrics that can measure the quality of inheritance hierarchy. In this chapter, we discuss that how we can use inheritance and class cohesion to measure or assess the quality of inheritance hierarchy. We propose five cohesion related graphical metrics for this purpose. Since we think that a metric should have some goal that determines its purpose, we will start our discussion by emphasizing on the importance of goal-driven software metrics. Then we will proceed on to discuss the implications of inheritance and class cohesion on software quality. Finally, we will propose a metric to measure the inheritance/class cohesion of a class.

3.1 Goal-Driven Software Metrics

In order to control the software quality related attributes, such as, reliability, maintainability, reusability and so forth, it is necessary to measure to what extent and degree these attributes are achieved by a certain product [Hitz95_1]. For this purpose, many software metrics have been established in the past, mainly in the area of traditional structured software design. Moreover, in the recent years several metrics for object-oriented software have also been proposed. However, it is imperative to mention here that the exercise of collecting a metric is useless without specifying the motivation to do so [Basi88].

A metric should be backed up by a good reasoning, that is, why it is important for the quality of the software and what is its interpretation. However, many metrics have been proposed that do not have any direct implication on the quality of software, such as, *DIT* (Depth of Inheritance Tree) and *NOM* (Number of Methods). Talking about *DIT*, Li and Henry [Li93] say: "*It seems logical that the lower a class in inheritance tree, the more super-class properties this class may access due to inheritance. If the subclass accesses the inherited properties from the super-class without using the methods defined in the super-class, the encapsulation of the super-class is violated. One may intuit that the larger the DIT metric, the harder it is to maintain the class*". As mentioned in [Hitz95_1], the analysis of the above statement shows that it is not a high *DIT* that makes the class hard to maintain or hard to understand, but it is rather the coupling induced by accessing super-class attributes (thus breaking the encapsulation principle) that deserves the blame.

Consequently, we should not measure *DIT* but rather "*Access to Super-Class Attributes*". In other words, *DIT* alone does not give a good measure of the software's quality.

Moreover, the value of *DIT* doesn't depict whether the inheritance has been used properly or not. Misuse of inheritance (also called Inheritance Mutation by Hitz and Montazeri [Hitz95_1]) means that the inheritance hierarchy is merely used for code sharing among otherwise unrelated classes [Eder94]. One might say that *DIT* can be used as an estimate for the "*misuse of Inheritance Hierarchy*", but it doesn't mean that a high value of *DIT* always depicts the abuse of inheritance.

Similarly, it might be suggested to use *NOM* for estimating the cohesiveness of a class. One might argue that the greater the number of methods in a class, the higher is the chance that the class is less cohesive. However, there might be a class that represents a large entity and thus must have a large number of methods. Therefore, we can say that the value of *NOM* is not directly related to class cohesion and the attribute of interest is *cohesion* rather than *NOM*.

The above discussion proposes the following pre-requisites for a metric:

- A metric should be directly related to the quality of software, i.e., a bad value of metric should only depict the bad quality of the quality aspect that this metric targets.

- It should be easy to interpret the value of metric in terms of the software quality.

Using the above guidelines, we have divided our task into two steps: *defining the goal* and *proposing the metrics*. Before proceeding to propose the metrics, we have set our goal, that is, what quality-attribute we want to target. On the basis of our goal, we have proposed a set of graphical cohesion-related metrics. In the next section, we discuss about our goal and the metrics proposed to meet that goal.

3.2 Cohesion and the Quality of Inheritance Hierarchy

While reviewing the existing cohesion metrics in chapter 2, we found out that none of these measures considers the effects of inheritance on class cohesion. However, since inheritance is the backbone of object-oriented design, one cannot neglect it while assessing the quality of object-oriented software. In literature we do not find any work on using cohesion to assess the quality of the inheritance hierarchy. Realizing the importance of inheritance and inheritance hierarchy in object-oriented systems, we have directed our goal towards using inheritance and class cohesion for assessing the design quality of the inheritance hierarchy.

As defined by Eder et al. [Eder94], *Class Cohesion* only inspects the binding of the newly defined elements (i.e., only implemented methods and attributes) within a class. Whereas, *Inheritance Cohesion* also takes the inheritance hierarchy into account. It describes the binding of the newly defined elements together with the inherited elements. Since the latter are transitively inherited from direct and indirect super-classes inheritance cohesion

evaluates not only the cohesion of an immediate class-superclass relationship but inspects the whole inheritance hierarchy. Inheritance cohesion is strong if this hierarchy is a generalization hierarchy in the sense of conceptual modeling, and it is weak if inheritance hierarchy is merely used for code sharing among otherwise unrelated classes.

Talking in terms of measurement, by inheritance cohesion we mean: "*Consider inherited attributes and methods as well as implemented attributes and methods while measuring cohesion of a class*". Let M_{INH} be the set of public/protected inherited methods and M_{IMP} be the set of implemented methods of a class C. Similarly, let A_{INH} be the set of public/protected inherited attributes and A_{IMP} be the set of implemented attributes of C. We consider $M(C) = M_{INH} \cup M_{IMP}$ and $A(C) = A_{INH} \cup A_{IMP}$ as the complete sets of methods and attributes, respectively, of class C while measuring its inheritance cohesion. Whereas, while measuring class cohesion only implemented methods and attributes are considered.

In the following subsections, we propose the following three cohesion-related graphical metrics that use inheritance cohesion to assess the design quality of the inheritance hierarchy:

- Inheritance Cohesion Versus Inheritance Level.
- Average Inheritance Cohesion of Children Versus Number of Children of Parent Classes.
- Inheritance Cohesion Versus Number of Overridden Methods.

In addition to the above three graphical metrics, we also propose two cohesion-related graphical metric that use class cohesion:

- Class Cohesion of Parent Classes Versus Number of Children.
- Class Cohesion Versus Number of Implemented Methods.

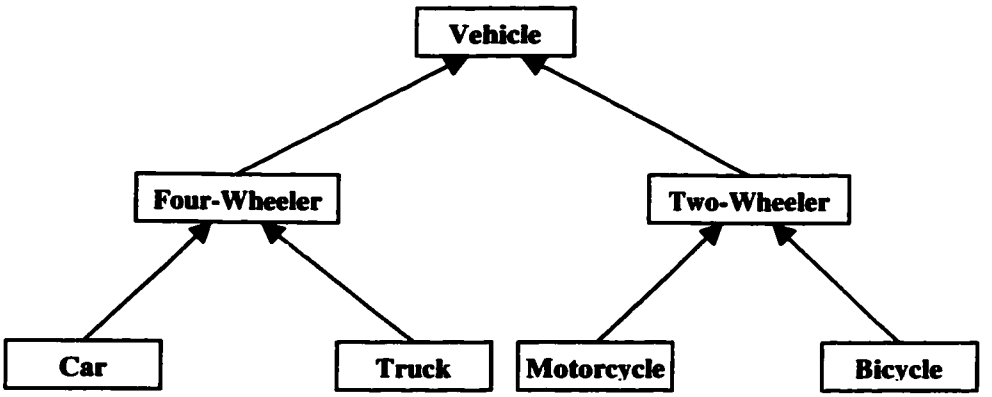


Figure 3.1: A good conceptual inheritance hierarchy that satisfies *is a* relationship

3.2.1 Inheritance Cohesion v/s Inheritance Level

If inheritance is used only to share code among unrelated classes, the cohesion will tend to decrease as we go down the inheritance hierarchy, since subclasses will inherit more and more unrelated methods and attributes. The graph of *Inheritance Cohesion* versus *Inheritance Level* visually shows whether inheritance has been properly used or not. Ideally this plot should be horizontal, i.e., cohesion doesn't change as we move down the inheritance hierarchy. This ideal situation implies that the inheritance hierarchy has not been used just for code sharing, but as a generalization hierarchy that satisfies the "*is a*"

relationship. Consider the inheritance hierarchy of figure 3.1. It is a good conceptual hierarchy, where each subclass satisfies the "*is a*" relationship. For example, a *car* is a *four-wheeler* and is a *vehicle*. Similarly, a *bicycle* is a *two-wheeler* and is a *vehicle*.

On the other hand, if *Inheritance Cohesion* versus *Inheritance Level* graph drops down as inheritance level increases, we can infer that inheritance may have been misused just for code sharing.

In some cases, this plot might also rise up with inheritance level, i.e., the children classes are more cohesive than their parent. The two possible reasons for this kind of behavior can be as follows:

1. The parent class has mostly empty/abstract methods, but its children override these empty/abstract methods to accomplish their tasks in such a way that these methods become related to the rest of their methods. For example, in the inheritance hierarchy of Java API, the root class is the class *Object* and most of its methods are empty. The cohesion of the class *Object* can be considered to be zero, since there are no connections among its methods. Therefore, the *Average Inheritance Cohesion Versus Inheritance Level* plot of java API starts from zero at level 1 (root) and then rises up to some value at level 2.

2. The parent class is used as a service class, i.e., it only contains unrelated methods that are used by its children classes merely as service methods. In such a case, the

parent might have a very low cohesion value, but its children might be more cohesive.

There can be the following two types of *Inheritance Cohesion* versus *Inheritance Level* graphs:

1. *Cohesion of Classes Sorted in Increasing Order of Inheritance Level*: In this type of plot, on horizontal axis, we have all the classes of the system sorted with respect to their inheritance level in ascending order and vertical axis shows the inheritance cohesion of these classes. An example of such a graph for a system of four classes is shown in figure 3.2.

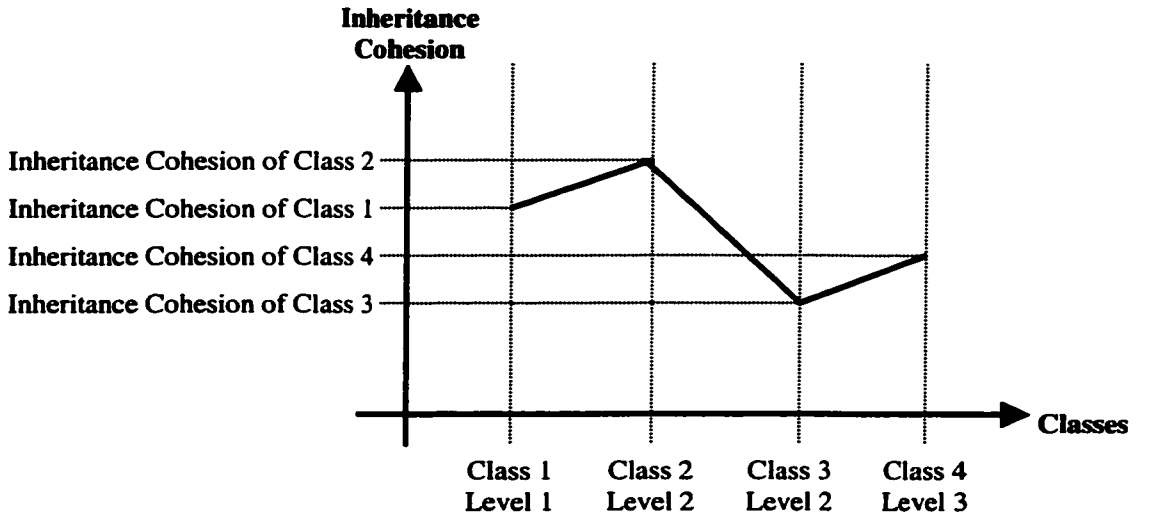


Figure 3.2: Plot of *Cohesion of classes sorted in increasing order of inheritance level* for a system of four classes.

2. *Average Inheritance Cohesion Versus Inheritance Level*: In this case, the horizontal axis represents the levels of inheritance tree from *1* to *n*, where *1* is the level of root and *n* is the level of leaves. Whereas, the vertical axis shows the average inheritance cohesion of each inheritance level. Average inheritance cohesion at a certain level means the average of the inheritance cohesion values of all the classes at that level. An example of such a plot is shown in figure 3.3.

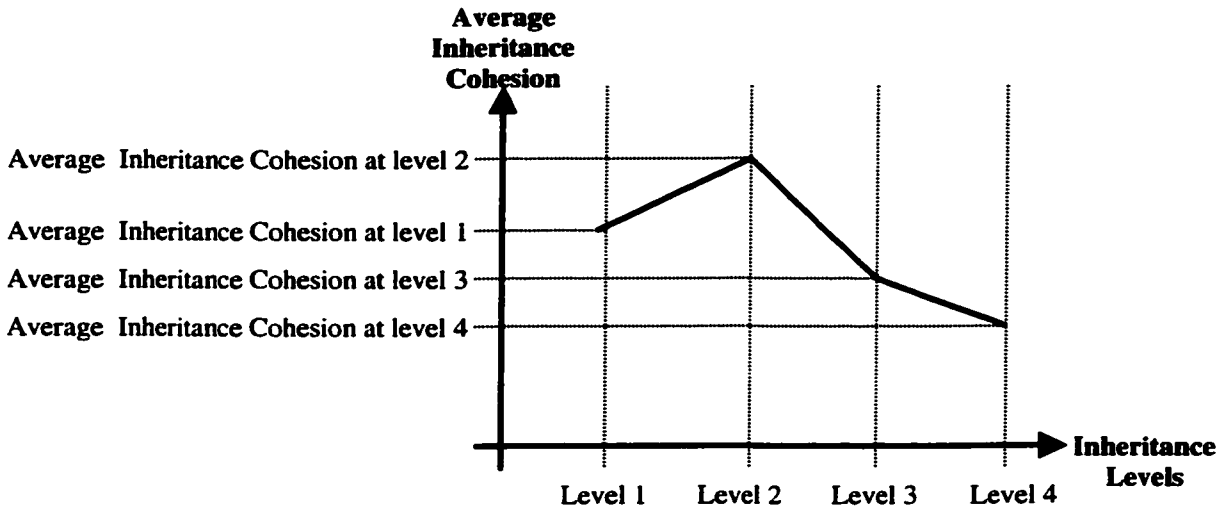


Figure 3.3: Plot of *Average Inheritance Cohesion Versus Inheritance level* for a system with four inheritance levels.

3.2.2 Average Inheritance Cohesion of Children Versus Number of Children of Parent Classes

This graph is intended to find out the effect of the number of direct children of a parent class on the inheritance cohesion of its children classes. In object-orientation, a designer might have a tendency to lump up the common methods of a group of unrelated classes into a single super-class. In this way, the super-class is used only as a service class or rather as a "*Container of Service Methods*" that are used by its subclasses. Furthermore, the designer might include more and more classes as the direct children of this "*Super-Cum-Service*" class to save the amount of coding. This will lead to a high number of direct children of this "*Super-Cum-Service*" class and adversely affect the cohesion of its children, since they inherit all the methods of the parent class but only few of those are related to their task. The plot of *Average Inheritance Cohesion of Children* versus *Number of Children of Parent Classes* gives a visual tool to detect this kind of discrepancy in an inheritance hierarchy. If this graph shows low values for the parent classes with high number of children, then it can be inferred that these parent classes may have been used merely as service classes and too many classes have been lumped up as their immediate children to save the amount of coding. Figure 3.4 shows an example of this type of plot for a fictitious hierarchy with four non-leaf classes, where, the horizontal axis represents all the parent classes of the hierarchy (i.e., the non-leaf classes of the inheritance tree) sorted in ascending order with respect to their number of (direct) children and vertical axis represents the average inheritance cohesion of the children of each parent class.

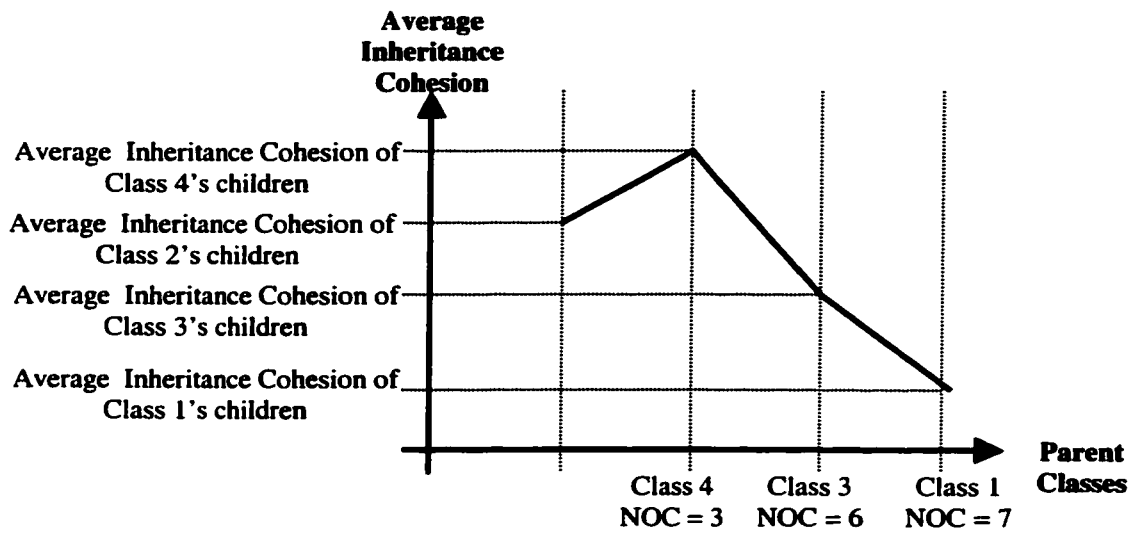


Figure 3.4: Plot of *Average Inheritance Cohesion of Children* versus *Number of Children* for a system with four non-leaf classes.

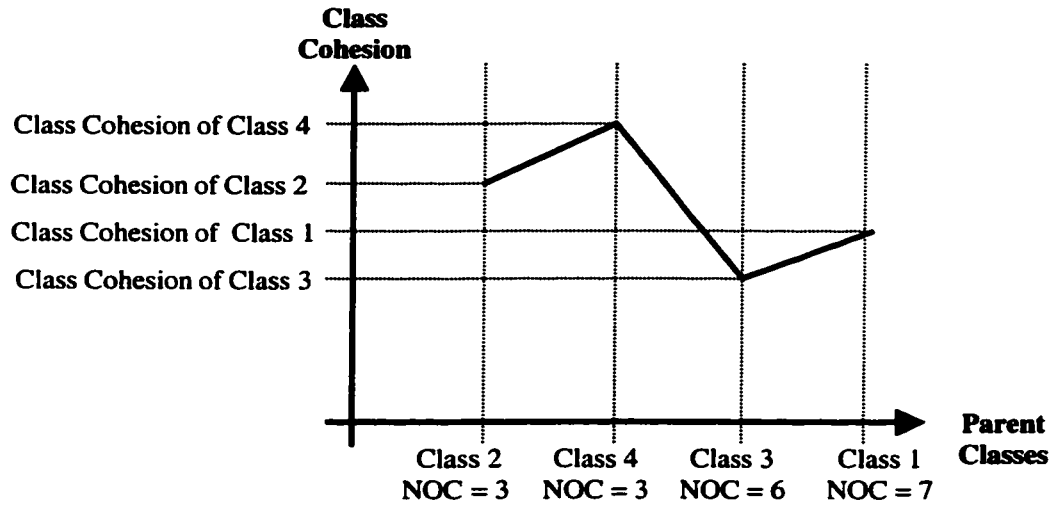


Figure 3.5: Plot of *Cohesion of Parent Classes* versus *Number of Children* for a system with four non-leaf classes.

3.2.3 Class Cohesion of Parent Classes Versus Number of Children

As discussed in section 3.2.2, sometimes the super-class is used only as a service class or rather as a *"Container of Service Methods"* that are used by its subclasses. In such a case, the methods of the super-class are usually unrelated to each other, since they have been implemented to fulfill the requirements of the subclasses rather than to accomplish the task of the super-class itself. Due to this lack of coherence among the methods of the super-class, its class cohesion drops down. The classes with a large number of children are more likely to have this kind of characteristic. The graph of *Class Cohesion of Parents Versus Number of Children* gives a tool to detect the existence of this type of parent classes in the inheritance hierarchy. If the classes with high number of (direct) children exhibit low class cohesion values, then it can be inferred that these parent classes might have been used merely as the containers of service methods that are used by their children. Figure 3.5 illustrates an example of this type of graph, where, the horizontal axis represents all the parent classes of the system (i.e., the non-leaf classes of the inheritance tree) sorted in ascending order with respect to their number of (direct) children and vertical axis represents the class cohesion of each parent class.

3.2.4 Class Cohesion Versus Number of Implemented Methods

From the perspective of cohesion, a badly designed class is one that abstracts more than one entity and can be subdivided. Based on this, one can say that a class with a large number of methods is more likely to be less cohesive, as a high value of *NOM* might be due to the reason that the class abstracts more than one entity. However, a high *NOM* value

doesn't always imply that the class is less cohesive. What is more logical to do is to examine the behavior of cohesion with respect to the number of methods. If cohesion deteriorates with increasing *NOM*, then it can be inferred that the bigger classes of the system may not have been properly designed and may need to be reworked. Figure 3.6 shows a graph of *Class Cohesion* Versus *Number of Implemented Methods* for a system of four classes. The horizontal axis shows the classes sorted with respect to their number of methods in ascending order and vertical axis gives the values of inheritance cohesion of these classes.

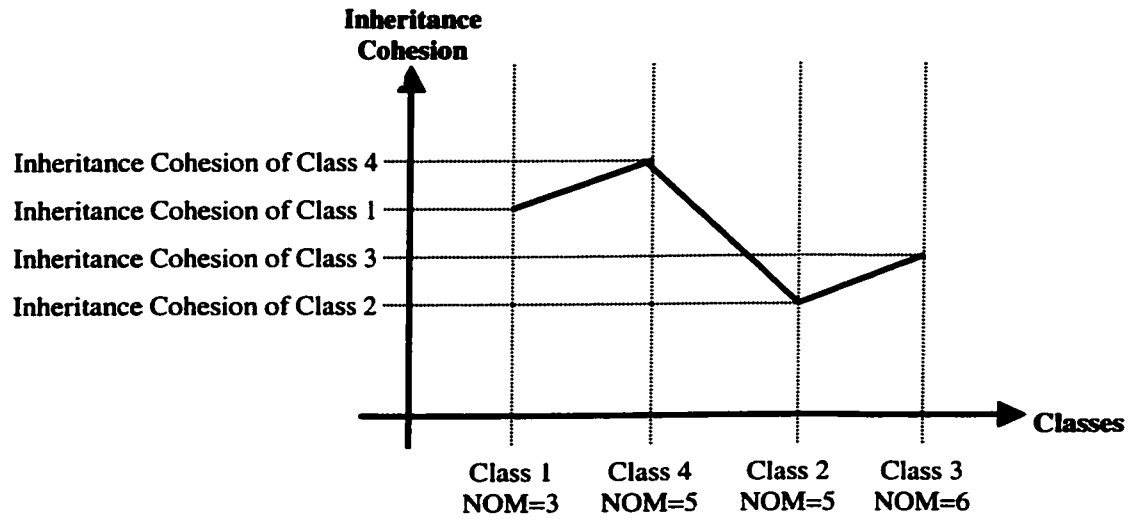


Figure 3.6: Plot of *Class Cohesion* versus *Number of Implemented Methods* for a system of four classes.

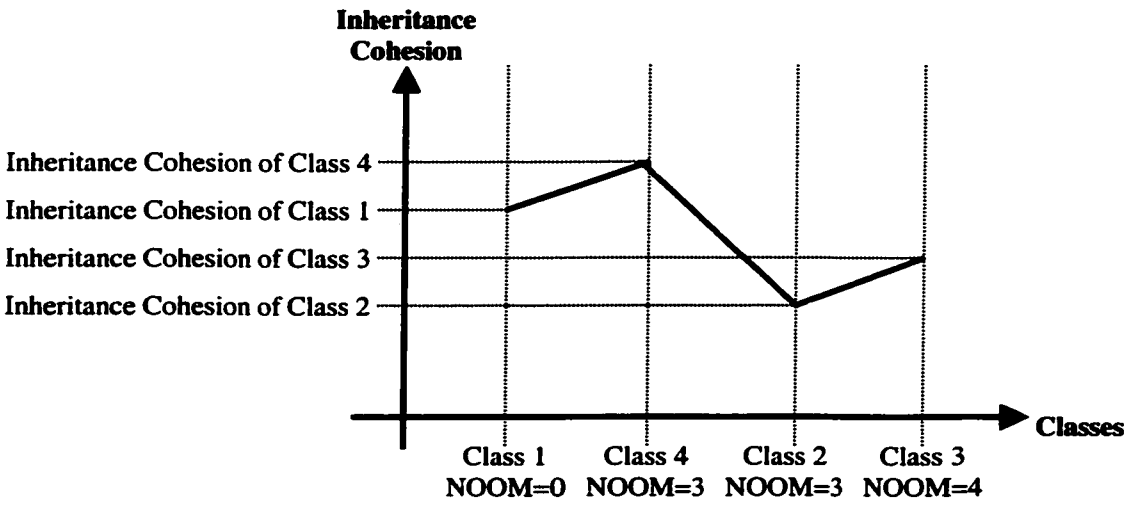


Figure 3.7: Plot of *Inheritance Cohesion* versus *Number of Overridden Methods* for a system of four classes.

3.2.5 Inheritance Cohesion Versus Number of Overridden Methods

Let a class *C'* inherits from a class *C* and overrides some of the methods of *C* in such a way that their implementation is completely changed in *C'*. Doing this, the cohesion of *C'* can be adversely affected as the part that is implemented/overridden in *C'* and the part that is inherited from *C* may get completely unrelated to each other. Using *Inheritance Cohesion* versus *Number of Overridden Methods* graph, one can visually detect the presence of such a class in the system. If a class with a good number of overridden methods exhibits a low cohesion value, it may imply that the methods of this class have not been properly overridden or it may be misfit at its present position in the inheritance hierarchy. Figure 3.7 shows a graph of *Inheritance Cohesion* Versus *Number of Overridden Methods* for a system of four classes. The horizontal axis shows the classes sorted with respect to their

number of overridden methods in ascending order and vertical axis gives the values of inheritance cohesion of these classes.

3.3 Capturing Cohesion

For plotting the graphs that were presented in the previous section, we first need to calculate Inheritance and Class Cohesion of all the classes in a system. For this purpose, we can use any of the cohesion metrics presented in chapter 2. Since we are interested in capturing the connections among the methods of a class as accurately as possible we have preferred to present our own method for measuring cohesion that is similar to some of the existing measures (like *TCC* and *LCC*), but we have tailored it to our requirements. We propose following three different cohesion measures:

- *CCM* (Class Connection Metric).
- *CCCM* (Connected Class Connection Metric).
- *MCCM* (Modified Class Connection Metric).

3.3.1 *CCM* and *CCCM*

These metrics are based on the connection graph G_C of the class C. The connection graph G_C has one vertex for each method of the class and there is an edge between two vertices if and only if the corresponding methods are connected according to the connection criterion defined by the metric. Let us first define our connection criterion. Two methods A and B

are connected in the connection graph G_C if they satisfy any or both of the following conditions:

- Methods *A* and *B* access one or more attributes in common.
- Methods *A* and *B* invoke one or more methods in common.

A good estimate for cohesion could be the ratio of the number of *Actual Connections* (i.e., number of edges in the connection graph G_C) to the number of *Maximum Possible Connections* (i.e., the maximum possible number of edges in the connection graph G_C) among the methods of a class *C*. However, another important thing to be considered is the *Number of Connected Components*, *NCC*(*C*), of the connection graph G_C. Let *NMP*(*C*) be the number of maximum possible connections among the methods of the class *C*. If the class *C* has *N* methods, $NMP(C) = N.(N - 1)/2$. Let *NC*(*C*) be the number of actual connections among the methods of the class *C*.

On the basis of the *Number of Connected Components* of G_C , i.e. *NCC*(*C*), we define *CCM* and *CCCM* as follows:

$$CCM(C) = \frac{NC(C)}{NMP(C) \,.\, NCC(C)} \qquad (3.1)$$

$$CCCM(C) = \begin{cases} \dfrac{NC(C)}{NMP(C)} & \text{if } NCC(C) = 1 \\ 0 & \text{if } NCC(C) > 1 \end{cases} \quad (3.2)$$

It can be noticed from equation 3.2 that *CCCM* gives nonzero values only for classes with connected connection graphs (i.e., $NCC(C) = 1$). Otherwise, it gives zero cohesion value.

From equation 3.1, it can be observed that *CCM* has some similarities with *TCC*. Both *CCM* and *TCC* capture the density of connections among the methods of a class, or in other words, they capture the connectivity of the class' connection graph. However, following are the two major differences between *CCM* and *TCC*:

1. *Connection Criterion of CCM is different from that of TCC*: *TCC* considers two methods *A* and *B* connected if they either access one or more attributes in common or any of *A* or *B* invokes the other. Whereas, *CCM* considers two methods *A* and *B* connected if they either access one or more attributes in common or invoke one or more methods in common.
2. *CCM also takes into account the Number of Connected Components of the class' Connection Graph*: Unlike *TCC*, *CCM* also penalizes a class for having more than one connected components in its connection graph. This has been done by including

the number of connecting components of the connection graph (i.e., *NCC*(*C*)) in the denominator of equation 3.1. Due to this, we can say that *CCM* is more sensitive than *TCC*.

As far as the *LCOM* measures are considered, *CCM* is quite different from these measures. First of all and most importantly, unlike the *LCOM* measures, *CCM* is normalized. Secondly, it is much more sensitive than the *LCOM* measures. For example, if we consider the *LCOM* version Hitz and Montazeri [Hitz96], we observe that it only counts the number of connected components of the class' connection graph and doesn't consider the connectivity of these connected components. On the other hand, *CCM* does not only take into account the number of connected components, but also considers the density of connections in the connected components of the class' connection graph.

CCM is quite different from *RCI* in nature. Unlike *CCM*, *RCI* doesn't measure the density of inter-method connections among the methods of a class, or in other words, it is not based on the connection graph of a class. Instead, *RCI* is based on counting the Declaration-Method interactions (*DM-interactions*) and Declaration-Declaration interactions (*DD-interactions*) as discussed in section 2.5.

3.3.2 Inheritance Cohesion and Method Overriding

There can be following two types of method overriding, based on the extent to which the implementation of the inherited method is changed through overriding:

1. *Overriding for Extension*: In this case, the subclass only extends the implementation of the inherited method, i.e., it doesn't rewrite it completely. For example consider a class called *Employee*, which is a generic class for all types of employees (such as Manager, Executive, Secretary, etc.). It has a method called *printEmployeeData* that prints the data of an employee. Since all types of employees have certain attributes in common (such as ID, Name, Birth Date, etc), the *Employee* class prints these common attributes in its own *printEmployeeData* method as shown below:

```
Public class Employee {
   private String name;
   private int ID;
   private Date bDate;
   -------
   -------
   public void printEmployeeData (){

        System.out.println("Name:" + name);
        System.out.println("ID:" + ID);
        System.out.println("Birth Date:" + bDate);
   }
   -------
   -------
}
```

Now consider the subclass *Manager* of the class *Employee*. Since every manager has a secretary, the manager class overrides the *printEmployeeData* method to include a statement for printing the secretary's name as shown below:

```
Public class Manager extends Employee{
    Private Secretary secretary;
    -------
    -------
    public void printEmployeeData (){

          super.printEmployeeData();

          System.out.println("SecretaryName:" +
                                        secretary.name);
    }
    -------
    -------
}
```

The *printEmployeeData* method of the class *Manager* is an example of *Overriding for Extension. Overriding for Extension* is not against the good design principles, as in this case the overridden method also uses the code of the original inherited method and doesn't discard it.

2. *Overriding for Re-implementation:* in this case the overridden method completely re-implements the inherited method. In other words, it completely discards the code of the original inherited method. When a subclass re-implements a method, then it means that the subclass doesn't need the original implementation of the inherited method, so it provides its own implementation of that method. In a way, it violates the "*is a*" relationship.

Since we think that a good inheritance hierarchy should satisfy the "*is a*" relationship, it is justified to penalize a subclass for having the overridden methods that exhibit the *Overriding for Re-implementation* characteristic. For this purpose, we introduce a *Penalty Factor*. Before defining the *Penalty Factor*, we want to describe the formulation needed to define it as follows:

- *Number of Overridden Methods*, *NOOM*(*C*): It is the number of methods overridden by the class *C*.
- *Number of Extended Methods*, *NOEM*(*C*): It is the number of those overridden methods of the class *C* that only extend the implementation of the original inherited method.
- *Number of Re-Implemented Methods*, *NORM*(*C*): It is the number of those overridden methods of the class *C* that re-implement the original inherited method.

- *Number of Unchanged Inherited Methods*, *NOUM*(*C*): It is the number of those inherited methods of the class *C* that are not overridden.
- *Number of Inherited Methods*, *NOIM*(*C*): It is the total number of methods inherited by the class *C*, i.e., $NOIM(C) = NOUM(C) + NOOM(C)$.

We define the *Penalty Factor* as follows:

$$Penalty\ Factor(C) = \frac{NORM(C)}{NOIM(C)} \tag{3.3}$$

Since $NORM(C) = NOIM(C) - NOUM(C) - NOEM(C)$, we can redefine the *Penalty Factor* as follows:

$$Penalty\ Factor(C) = \frac{NOIM(C) - NOUM(C) - NOEM(C)}{NOIM(C)} \tag{3.4}$$

Equation 3.3 shows that the *Penalty Factor* is higher for the classes having higher number of re-implemented overridden methods.

We present a modified version of *CCM* that also penalizes the class for having the overridden methods that completely change the implementation of their corresponding original inherited methods. We name this new version of *CCM* as *MCCM* (Modified Class Connection Metric). *MCCM* is defined as follows:

$$MCCM\ (C) = \frac{NC(C)}{NMP(C)\ .\ NCC(C)} \times (1 - Penalty\ Factor(C)) \tag{3.5}$$

Or,

$$MCCM\ (C) = CCM(C) \times (1 - Penalty\ Factor(C)) \tag{3.6}$$

3.3.3 Cohesion and Degree of Inter-Method Connections

Degree of inter-method connections can be another factor to be considered while measuring the cohesion. The degree of connection between two methods *A* and *B* is the number of attributes/methods accessed by these methods in common. One can consider the connection between a pair of methods with higher degree of connection stronger than the connection between another pair with lower degree of connection. Based on this idea we introduce a factor called *Sharing Factor* that measures the average (normalized) degree of connections of a connection graph G_C. Let *Connection*(*C*) be the connection matrix for the class *C* as shown below:

$$Connection(C) = \begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ M_1 & c_{11} & c_{12} & c_{13} & c_{14} \\ M_2 & c_{21} & c_{22} & c_{23} & c_{24} \\ M_3 & c_{31} & c_{32} & c_{33} & c_{34} \\ M_4 & c_{41} & c_{42} & c_{43} & c_{44} \end{array} \qquad (3.7)$$

Where, each c_{ij} is the number of attributes and methods accessed by methods M_i and M_j in common. The connection matrix *Connection*(*C*) actually represents the weighted connection graph of the class *C*, where the weight of each edge e_{ij} between the vertices V_i and V_j, corresponding to the methods M_i and M_j, is equal to the number of attributes and methods accessed by M_i and M_j in common. We then calculate the sharing matrix, *Sharing*(*C*), of the class *C* from its connection matrix, *Connection*(*C*).

Let $A(C)$ be the set of attributes of the class C. We calculate each element s_{ij} of the sharing matrix, *Sharing*(C), as follows:

$$s_{ij} = \begin{cases} \dfrac{c_{ij}}{|A(C)|} & \text{if } c_{ij} < |A(C)| \\ 1 & \text{if } c_{ij} \geq |A(C)| \end{cases} \tag{3.8}$$

$$Sharing(C) = \begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ \hline M_1 & s_{11} & & & \\ M_2 & s_{21} & s_{22} & & \\ M_3 & s_{31} & s_{32} & s_{33} & \\ M_4 & s_{41} & s_{42} & s_{43} & s_{44} \end{array} \tag{3.9}$$

We define the *Sharing Factor* as the average of the upper half values of the sharing matrix (shown as the shaded area in the matrix of equation 3.9). Therefore, formally we can define

$$Sharing\ Factor = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} s_{ij}}{NMP(C)} \tag{3.10}$$

Where $NMP(C)$ is the number of maximum possible connections among the methods of class C as defined previously.

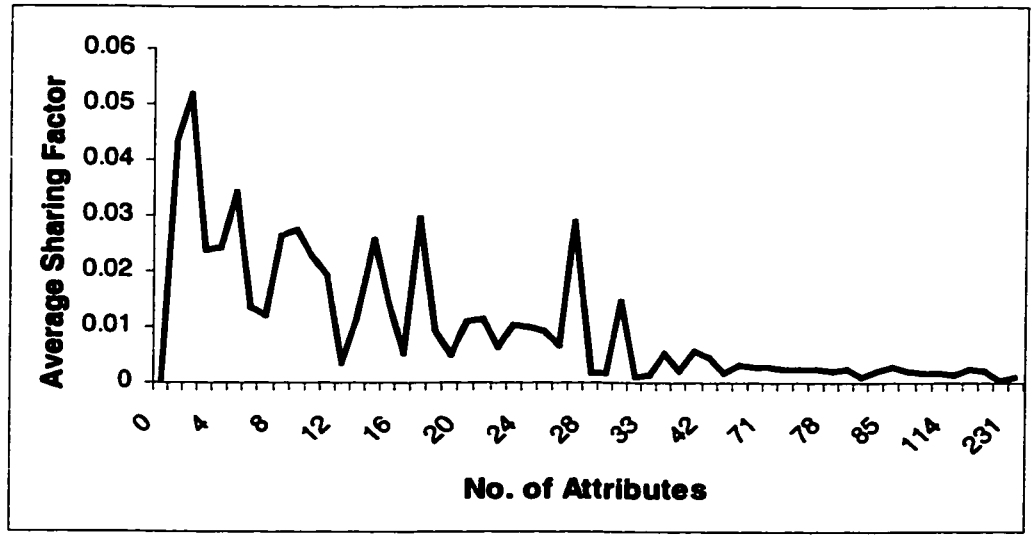


Figure 3.8: Average Sharing Factor against No. of Attributes graph for *java.awt* hierarchy.

To account for the degree of inter-method connections in cohesion value, we can simply multiply the *Sharing Factor* by the *CCM* value calculated by equation 3.1. However, there are the following problems associated with the *Sharing Factor*:

- The *Sharing Factor* unnecessarily penalizes the classes with higher number of attributes. Each element s_{ij} of the sharing matrix *Sharing*(C) is calculated using the formula c_{ij} / $|A(C)|$ in equation 3.7, where c_{ij} is the number of attributes/methods accessed by methods M_i and M_j in common and $|A(C)|$ is the total number of attributes of the class C. Due to the denominator term, $|A(C)|$, in equation 3.7, the s_{ij} values of the classes with a higher number of attributes (i.e., high $|A(C)|$) tend to

be lower. Figure 3.8 illustrates the graph of *Sharing Factor* against the *Number of Attributes* for *java.awt* hierarchy of Java 1.3 API, where, the horizontal axis represents the classes sorted in ascending order with respect to their number of attributes and vertical axis gives the values of *Sharing Factor* for these classes. It can be seen from the graph of figure 3.8, that the *Sharing Factor* of classes with higher number of attributes is too low.

- It is not necessarily a good design consideration that methods should share as much attributes as possible [Briand98]. As long as the methods are connected, it is alright, but their degree of connection may not be very important.
- It is almost impossible to get the maximum *Sharing Factor* (i.e., *Sharing Factor*(C) = 1), since most of the methods share a small percentage of the total number of attributes. In fact, the *Sharing Factor* even for the classes with quite good connection graphs can be quite low, since the methods usually share only a few attributes, but the dividing factor, $|A(C)|$, in equation 3.7 may be very high.

In order to rectify the above-mentioned flaws in the *Sharing Factor*, we can change the procedure for calculating it as follows: Instead of dividing c_{ij} by $|A(C)|$ in equation 3.8, we can divide it by the total number of attributes and methods accessed by both the methods i and j. Let,

NA_i = Number of attributes accessed by the method i.

NM_i = Number of methods accessed by the method i.

NA_j = Number of attributes accessed by the method j.

NM_j = Number of methods accessed by the method *j*.

We redefine the formula for calculating the elements of the *Sharing Matrix* as follows:

$$s_{ij} = \frac{c_{ij}}{NA_i + NM_i + NA_j + NM_j} \tag{3.11}$$

By using equation 3.11 for calculating the elements of the *Sharing Matrix*, we can rectify the major problem associated with the *Sharing Factor*, i.e., the *Sharing Factor* will not penalize the classes having higher number of attributes any more.

Chapter 4

AUTOMATED METRICS CALCULATION

Since the manual calculation of metrics is almost impossible, it is important to have an automated tool for calculating metrics directly from the code for the effective utilization of these metrics. We have developed a tool for the measurement and analysis of cohesion metrics for object-oriented systems. The user only has to supply the code of the system to be analyzed to the tool and the rest of the tasks are done automatically.

The tool works as follows: First, it parses the code of the object-oriented system to collect the cohesion metrics data. The collected data is then stored into the Central Metrics Repository. Finally, the cohesion metrics are calculated from the data stored in the Central Metrics Repository. Right now the tool only supports the Java software, but it is not limited to a particular language. In order to add support for a new language, the only thing to be done is to add the parser for that language.

In this chapter, we discuss the design and implementation details of the tool. Since the first task was to design a suitable mathematical modeling for the object-oriented systems, we start our discussion by presenting the way we have modeled the object-oriented systems for cohesion measurement. Then we discuss about the parser that we have developed for the Java language. Afterwards, we talk about the design of Central Metrics Repository. Finally, we give the description of different facilities provided by the tool.

4.1 Modeling the Object-Oriented Systems

For calculating metrics, it is necessary to model the software system in the form of some data-structures. We have used two-dimensional arrays for representing object-oriented systems. Following three different types of two-dimensional arrays have been used to model object-oriented systems for cohesion measurement:

- System Definition Matrix.
- Class Definition Matrix.
- Class Cohesion Matrix.

4.1.1 System Definition Matrix

System Definition Matrix gives the list of attributes and methods of all the classes of the system. Its rows represent the classes and columns represent all the methods and attributes of the system. Figure 4.1 illustrates the System Definition Matrix for a system with n classes, p methods and q attributes.

Class	Methods				Attributes			
	M_1	M_2	...	M_p	A_1	A_2	...	A_q
C_1	D_{11}	D_{12}	...	D_{1p}	$D_{1(p+1)}$	$D_{1(p+2)}$	...	D_{1q}
C_2	D_{21}	D_{22}	...	D_{2p}	$D_{2(p+1)}$	$D_{2(p+2)}$	...	D_{2q}
...			...				...	
C_i	D_{i1}	D_{i2}	...	D_{ip}	$D_{i(p+1)}$	$D_{i(p+2)}$	...	D_{iq}
...			...				...	
C_m	D_{n1}	D_{n2}	...	D_{np}	$D_{n(p+1)}$	$D_{n(p+2)}$	...	D_{nq}

Figure 4.1: An example of the System Definition Matrix.

In the matrix of figure 4.1, each entry D_{ij} shows whether the jth element (method/attribute) is present in class i or not. Therefore,

$$D_{ij} = \begin{cases} 1 \text{ if the } j\text{th element (method/attribute) is present in class } i. \\ 0 \text{ if the } j\text{th element (method/attribute) is not present in class } i. \end{cases} \quad (4.1)$$

As discussed in section 2.3, we can have the following three options with respect to inheritance:

1. Exclude inherited methods and inherited attributes from the analysis, or
2. Include inherited methods and inherited attributes in the analysis, or
3. Exclude inherited methods but include inherited attributes.

We make three versions of the System Definition Matrix, one for each of the above three inheritance options. For option 1, we have ones only in those columns of row i that

correspond to the implemented methods and attributes of class *i*, whereas for option 2, we also have ones in those columns of row *i* that correspond to the inherited methods and attributes of class *i*. Similarly, for option 3, we have ones in the columns of row *i* corresponding to the inherited attributes of class *i* as well as the columns corresponding to its implemented methods and attributes.

4.1.2 Class Definition Matrix

The Class Definition Matrix gives the description of the attributes and methods accessed/used by the methods of the class. Its rows represent the methods and columns represent the methods as well as the attributes of the class. Figure 4.2 shows the Class Definition Matrix of a class with *n* methods and *k* attributes.

Methods	**Methods**				**Attributes**			
	M_1	M_2	...	M_n	A_1	A_2	...	A_k
M_1	d_{11}	d_{12}	...	d_{1n}	$d_{1(n+1)}$	$d_{1(n+2)}$	...	d_{1k}
M_2	d_{21}	d_{22}	...	d_{2n}	$d_{2(n+1)}$	$d_{2(n+2)}$	...	d_{2k}
...			...				...	
M_i	d_{i1}	d_{i2}	...	d_{in}	$d_{i(n+1)}$	$d_{i(n+2)}$	...	d_{ik}
...			...				...	
M_n	d_{n1}	d_{n2}	...	d_{nn}	$d_{n(n+1)}$	$d_{n(n+2)}$	...	d_{nk}

Figure 4.2: An example of the Class Definition Matrix.

In the matrix of Figure 4.2, each entry d_{ij} shows whether the jth element (attribute/method) is used/accessed by method i or not. Therefore,

$$d_{ij} = \begin{cases} 1 & \text{if the } j\text{th element (method/attribute) is accessed/used by method } i. \\ 0 & \text{if the } j\text{th element (method/attribute) is not accessed/used by method } i. \end{cases} \quad (4.2)$$

The System Definition Matrix is used to calculate the number of rows and columns of the Class Definition Matrix of each class. For example, consider the System Definition Matrix of figure 4.1. To find the number of rows and columns of the Class Definition Matrix of class i, we use the following equation:

$$ROWS_i = \sum_{j=1}^{p} D_{ij} \quad (4.3)$$

$$COLS_i = \sum_{j=1}^{p+q} D_{ij} \quad (4.4)$$

Where, $ROWS_i$ and $COLS_i$ are the number of rows and columns, respectively, of class i's Class Definition Matrix, p and q are the number of methods and attributes, respectively, in

the System Definition Matrix of figure 4.1 and D_{ij} is the element of the System Definition Matrix corresponding to the ith row and jth column.

The elements of the Class Definition Matrix are filled with zeros and ones using the data stored in the Central Metrics Repository. For each method m of the class C, the list L of attributes and methods accessed/used by m is obtained from the Central Metrics Repository. The elements of the mth row (i.e., the row corresponding to the method m) of C's Class Definition Matrix that are present in the list L are filled with ones, whereas, the remaining elements are filled with zeros. Like the System Definition Matrix, we also keep three versions of the Class Definition Matrix – one for each inheritance option.

4.1.3 Class Connection Matrix

The Class Connection Matrix is used to store the information about the connections between the methods of the class C. The Class Connection Matrix for the class C is derived from its Class Definition Matrix. Its rows and columns represent the methods of the class C. For example, consider the Class Definition Matrix of figure 4.2. Its corresponding Class Connection Matrix is shown in figure 4.3, where each element C_{ij} gives the number of attributes and methods accessed/used by the method i and j in common. Using the Class Definition Matrix, the elements of the Class Connection of the class C Matrix can be obtained as follows:

$$C_{ij} = \sum_{j=1}^{n+k} (d_{ik} \wedge d_{jk}) \qquad (4.4)$$

Where d_{ik}, d_j are the elements of the Class Definition Matrix (Figure 4.2), n is the number of methods of the Class Definition Matrix (Figure 4.2), k is the number of attributes of the Class Definition Matrix (Figure 4.2) and $\wedge$ is the logical AND operator.

Methods	**Methods**					
	M_1	M_2	...	M_i		M_n
M_1	C_{11}	C_{12}	...	C_{1i}		C_{1n}
M_2	C_{21}	C_{22}	...	C_{2i}		C_{2n}
...			...			
M_i	C_{i1}	C_{i2}	...	C_{ii}		C_{in}
...			...			
M_n	C_{n1}	C_{n2}	...	C_{ni}		C_{nn}

Figure 4.3: An example of the Class Connection Matrix.

Like the System Definition and Class Definition Matrices, we also have three different versions of each Class Connection Matrix, i.e., one for each inheritance option. It can be noticed that the Class Connection Matrix actually represents the weighted connection graph of the class C, where the weight of each edge e_{ij} between the vertices V_i and V_j, corresponding to the methods M_i and M_j, is equal to the number of attributes/methods accessed by M_i and M_j in common. Using the Class Connection Matrix illustrated in figure 4.3, we can easily calculate the values of the *CCM*, *CCCM*, *MCCM* and *Sharing Factor* as described in section 3.3.

4.2 The Tool's Architecture

The tool is composed of three main components: Parsing Engines, Central Metrics Repository and Computing Module. Figure 4.4 illustrates the structure of the tool. The architecture that we have used is similar to the one described in [Elish99] for the Inheritance Coupling Measurement Tool.

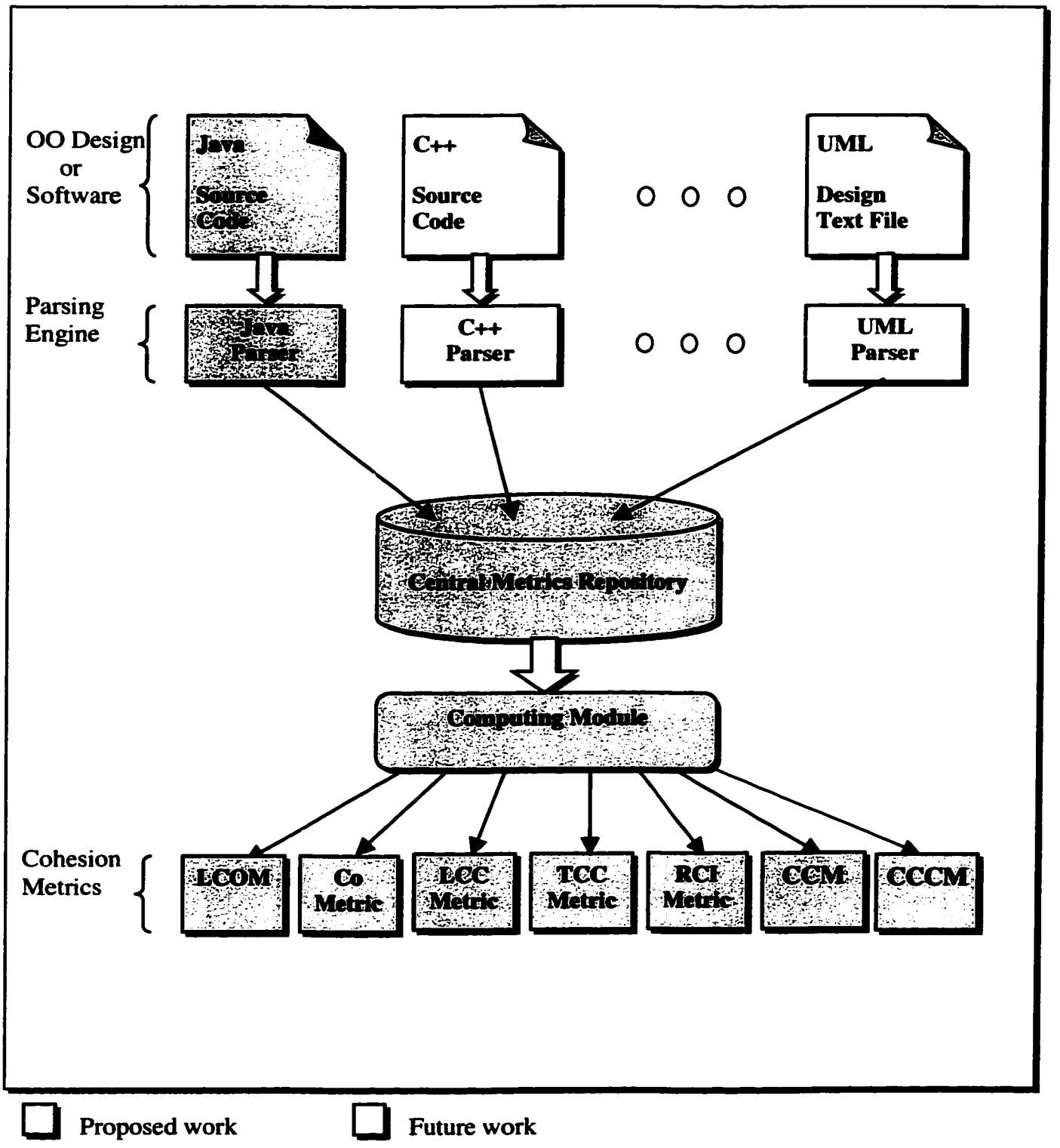


Figure 4.4: Architecture of the tool.

4.2.1 Parsing Engines

Data collection is the first step in calculating any software metric. Parsing Engines aim to extract the information needed to compute cohesion metrics from the code of the object-oriented system. The parsers are language dependent, i.e., a separate parser need to be implemented for each object-oriented language like Java, C++, Smalltalk, etc. The data collected by these parsers is abstracted into a language independent format according to the conceptual model shown in figure 4.5.

In this work we have developed a Java parser. It will be able to parse any Java source code to collect cohesion metrics data into the Central Metrics Repository. We have used SableCC[1] for implementing the Java 1.2/1.3 parser. SableCC is a compiler-compiler kit written in Java. To generate the parser for a language using SableCC, we need to write an LALR(1) grammar for that language in the syntax accepted by SableCC. We used official Java 1.2/1.3 specifications[2] for writing the grammar for Java 1.2/1.3.

Once the grammar is submitted to SableCC, it generates the Abstract Syntax Tree (AST) for the grammar in the form of a java class called *DepthFirstAdapter.java*. This class has one method for each non-leaf node of the AST. In order to define the functionality at any node of the AST, we can override the method corresponding to that node in *DepthFirstAdapter.java*. For example, the grammar for the package declaration in Java can be given as follows:

---

[1] Information about SableCC is available from the website: http//:www.sablecc.org/
[2] Official Java 1.2/1.3 specifications are available from the websites: http://java.sun.com/docs/books/vmspec/

```
Package_declaration =
                    Package name semicolon;
```

Where *package* is a Java reserved word, *name* is a variable to hold the package name and *semicolon* is a helper word for ";" symbol. The method for the package-declaration node of the AST in *DepthFirstAdapter.java* is as follows:

```
Public void
  caseAPackageDeclaration(APackageDeclaration node){
        inAPackageDeclaration(node);
        if(node.getPackage() != null){
            node.getPackage().apply(this);
        }
        if(node.getName() != null){
            node.getName().apply(this);
        }
        if(node.getSemicolon() != null{
            node.getSemicolon().apply(this);
        }
        outAPackageDeclaration(node);
}
```

If we want to print the name of the package to the standard output, we can override the above method as follows (the added line is written in the bold face letters):

```
Public void
  caseAPackageDeclaration(APackageDeclaration node){
        inAPackageDeclaration(node);
        if(node.getPackage() != null){
            node.getPackage().apply(this);
        }
        if(node.getName() != null){
             System.out.println(node.getName());
            node.getName().apply(this);
         }
        if(node.getSemicolon() != null{
```

```
            node.getSemicolon().apply(this);
        }
        outAPackageDeclaration(node);
}
```

Using the similar strategy, we have extended the class *DepthFirstAdapter.java* generated by SableCC for Java 1.2/1.3 grammar in the form of a class called *interpreter.java.* We have overridden the methods of *DepthFirstAdapter.java* in *interpreter.java* to provide the functionality for storing the class data into the Central Metrics Repository. The details of writing parsers and interpreters using SableCC can be found in [Gagnon98].

4.2.2 Central Metrics Repository

The data collected by the Parsing Engines is stored into the Central Metrics Repository. The repository has been designed based on the ER model presented in Figure 4.6. This model is language independent, i.e., it represents an abstract format of the object-oriented system. Although we have adopted this ER model for representing the object-oriented systems from [Elish99], but we have tailored it to the requirements of cohesion measurement.

4.2.3 Computing Module

The Computing Module processes the data stored in the Central Metrics Repository for calculating the cohesion metrics. A simplified process flow diagram for the calculations done by the Computing Module is illustrated in figure 4.6. Basically, the Computing Module has the following four types of functions:

1. *makeSystemDefinitionMatrix*: This function makes the System Definition Matrix using the metrics data stored in the Central Metrics Repository as described in section 4.1.1.
2. *makeClassDefinitionMatrix*: This function makes the Class Definition Matrix, for each class of the system, using the System Definition Matrix and data stored in the Central Metrics Repository as described in section 4.1.2.
3. *makeClassConnectionMatrix*: This function makes the Class Connection Matrix, for each class of the System, using the Class Definition Matrix of that class as described in section 4.1.3.
4. *calculateCohesion*: This function calculates the cohesion metrics for all the classes of the system using their Class Connection Matrices as described in section 4.1.4. In case of *MCCM*, the *calculateCohesion* function also uses the Central Metrics Repository to get the number of overridden methods (*NOOM*(*C*)), number of extended methods (*NOEM*(*C*)), number of re-implemented methods (*NORM*(*C*)) and number of unchanged inherited methods (*NOUM*(*C*)).

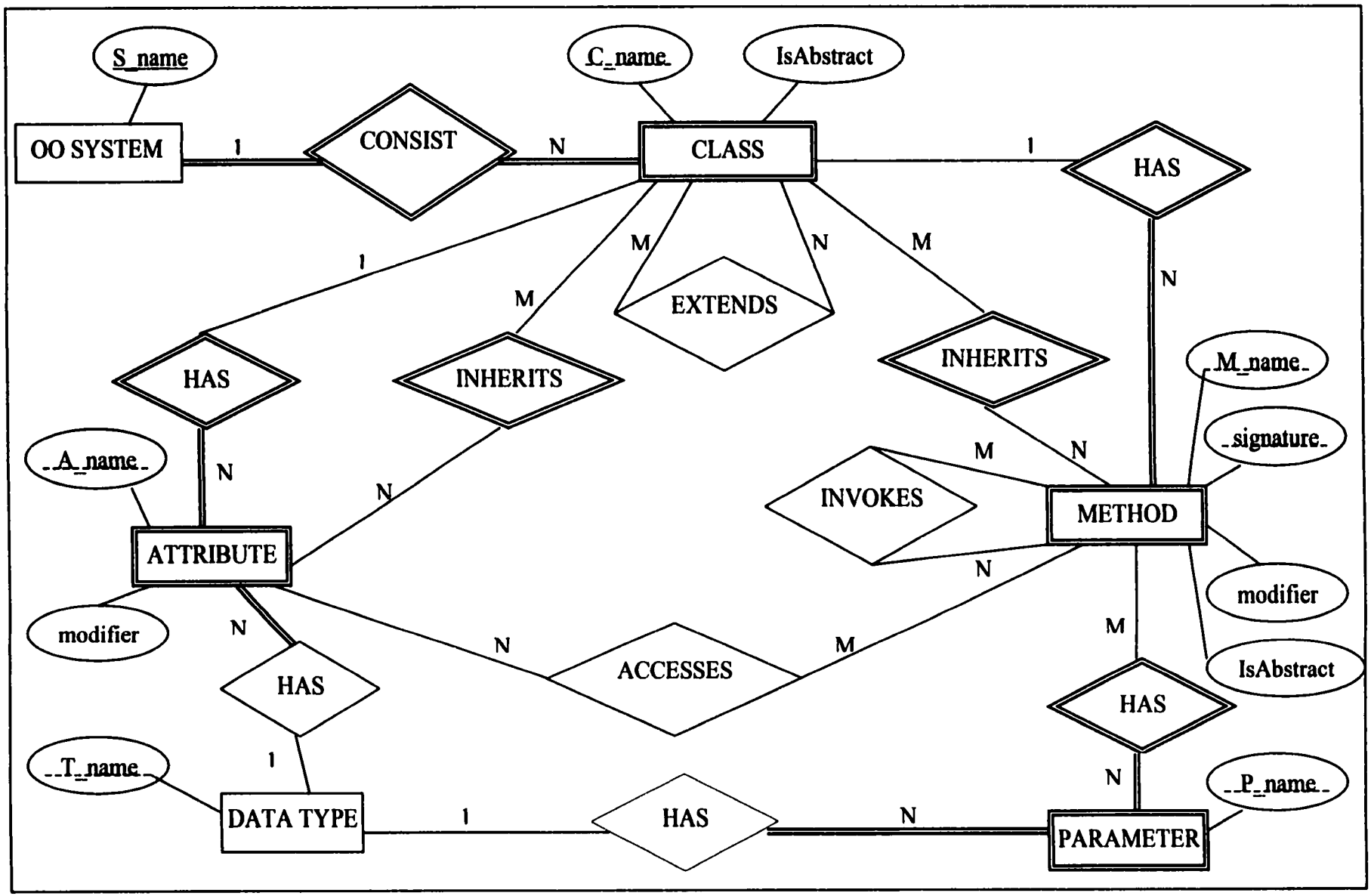


Figure 4.5: ER Diagram of the Central Metrics Repository.

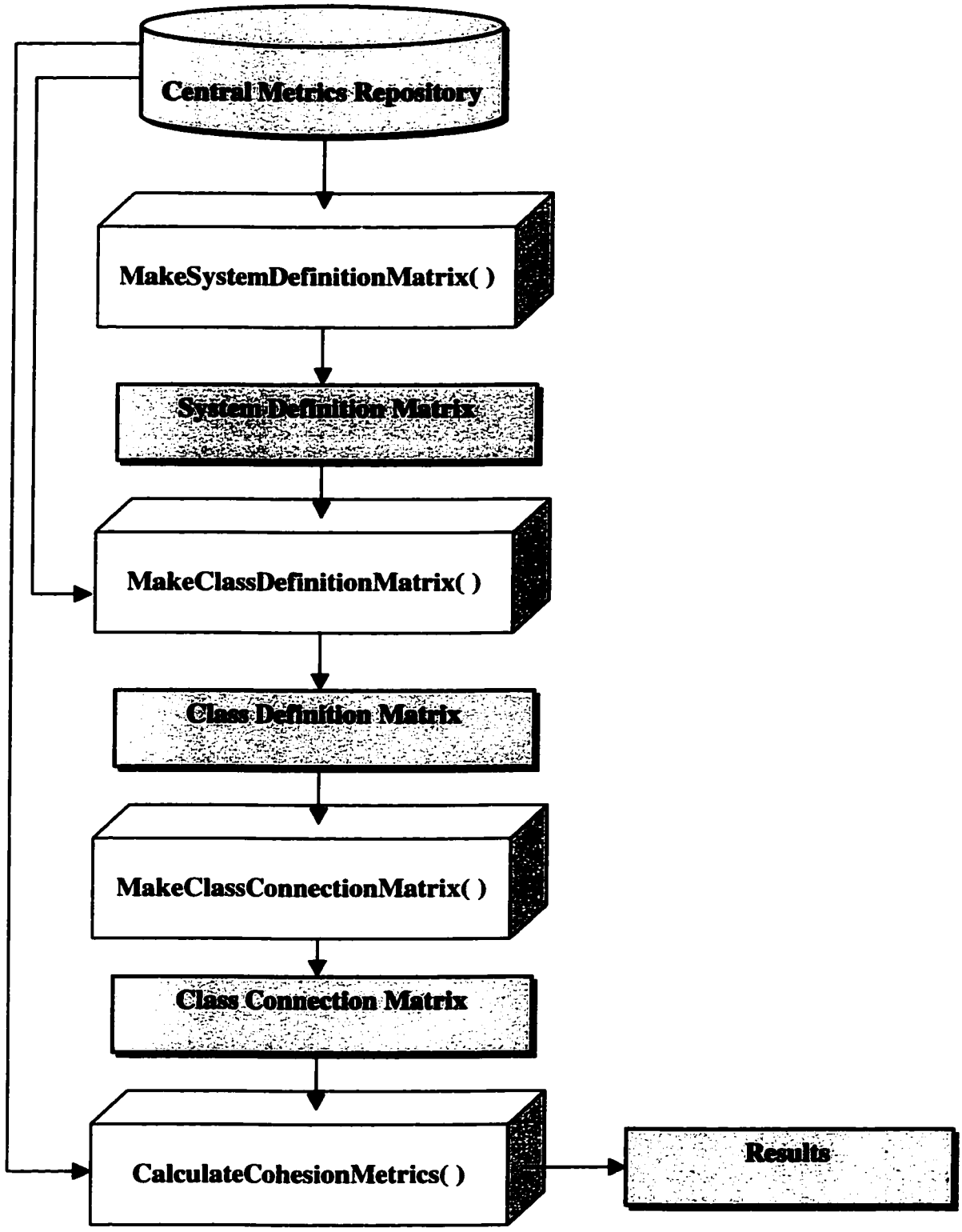


Figure 4.6: Process flow diagram of the Computing module.

4.3 The Graphical User Interface

The tool provides a comprehensive graphical user interface for calculating and analyzing the cohesion metrics. We have divided the GUI features into the following two different parts:

- Cohesion Statistics.
- Graphical Analysis.

4.3.1 Cohesion Statistics

This part deals with cohesion-related numerical statistics of the software system being analyzed. User submits the code of the system to be analyzed to the tool. The tool automatically parses the code, generates the System Definition Matrix, Class Definition Matrices and Class Cohesion Matrices and calculates various Cohesion Metrics. User can then view the results in the form of various tables and with different options. All the results can be viewed for the following three inheritance options (presented in section 2.3):

1. *Without Inheritance*: Selecting this option, the user can view results for inheritance option 1, i.e., Exclude inherited methods and inherited attributes from the analysis.
2. *Including Both Inherited Attributes and Methods*: Choosing this option, the user can view results for inheritance option 2, i.e., Include inherited methods and inherited attributes in the analysis.
3. *Including Only Inherited Attributes*: Using this option, the user can view results for inheritance option 3, i.e., Exclude inherited methods but include inherited attributes.

We have also provided support for the following two different options with respect to the constructor methods:

1. *Without Constructor*: With this option, the results are shown for the metrics calculations done without including the class constructors.
2. *With Constructor*: With this option, the results are shown for the metrics calculations in which the class constructors are also included with the other methods.

Some of the important results that can be viewed are as follows:

1. *System Definition Matrix*: The System Definition Matrix can be viewed for any of the above-mentioned three inheritance options as well as two constructor options.
2. *Class Definition and Class Connection Matrices*: The Class Definition Matrix as well as Class Connection Matrix can be viewed for any class of the system. User also has the flexibility of selecting any of the above-mentioned inheritance options and constructor options.
3. *Cohesion Metrics*: The tool displays the values of the following cohesion metrics:
    - Low Cohesion Metric (LCOM).
    - Loose Class Cohesion (LCC).
    - Tight Class Cohesion (TCC).
    - Connectivity Metric (C).
    - Ratio of Cohesive Interactions (RCI).
    - Class Connection Metric (CCM).

- Modified Class Connection Metric (MCCM).
- Connected Class Connection Metric (CCCM).

The user can select any of the above mentioned inheritance options and constructor options while viewing the results of these metrics.

4.3.2 Graphical Analysis

Graphical analysis is an effective tool for analyzing the results. It presents the overall picture of the results. The list of the different types of graphs supported by the tool is given below. All of these graphs can be viewed for any of the cohesion metrics listed in section 4.3.1.

1. *System Bar Charts*: This plot shows the cohesion values of all the classes of a system in the form of a bar chart. It can be viewed for any of the three inheritance options and two constructor options.
2. *Inheritance Effect*: This plot provides the comparison among the three inheritance options. It displays three curves – one for each inheritance option. User can select any of the two constructor options while viewing this plot.
3. *Constructor Effect*: This plot provides the comparison between the two constructor options. It displays two curves – one for each constructor option. User can select any of the three inheritance options while viewing this plot.
4. *Inheritance Cohesion Versus Inheritance Level*: This plot corresponds to the graphical metric proposed in section 3.2.1. As mentioned in section 3.2.1, there are two subtypes of this plot, i.e., *Cohesion of Classes Sorted in Increasing Order of*

Inheritance Level and *Average Inheritance Cohesion Versus Inheritance Level.* While viewing these graphs, the user can choose any of the two constructor options.

5. *Average Inheritance Cohesion of Number of Children of Parent Classes*: This plot corresponds to the graphical metric proposed in section 3.2.2. The user can select any of the two constructor options while viewing this plot.
6. *Class Cohesion of Parents Versus Number of Children*: This plot corresponds to the graphical metric proposed in section 3.2.3. The user can select any of the two constructor options while viewing this plot.
7. *Class Cohesion Versus Number of Methods*: This plot corresponds to the graphical metric proposed in section 3.2.4. The user can select any of the two constructor options while viewing this plot.
8. *Inheritance Cohesion Versus Number of Overridden Methods*: This plot corresponds to the graphical metric proposed in section 3.2.5. Any of the two constructor options can be chosen while viewing this plot.
9. *Comparison of Systems*: This plot displays the comparison between the results of two different software systems.

Chapter 5

ANALYZING LARGE OBJECT-ORIENTED SYSTEMS

Using the tool described in chapter 4, it is possible to test and analyze considerably large Java systems with an effort of just a few man-hours. We have analyzed the inheritance hierarchy of Java 1.3 API using our tool. In this chapter, we discuss the observations and results that we obtained from the analysis.

We have divided the results into two parts: the first part talks about the various observations that we obtained for the Java 1.3 API and second part discusses the differences that we noticed among different cohesion metrics presented in chapter 2 and 3.

5.1 Analysis of Java 1.3 API

The API of Java 1.3 is divided into the following seven parts:

1. *com*
2. *java*
3. *javax*

4. *launcher*
5. *org*
6. *sun*
7. *sunw*

The core API of Java 1.3 (or any other version of Java) is contained in the *java* part and it is the largest part of the API. We have analyzed the *java* part of the API. The *java* part has 861 classes in total and it is further sub-divided into the following twelve parts:

1. *java.applet*
2. *java.awt*
3. *java.beans*
4. *java.io*
5. *java.lang*
6. *java.math*
7. *java.net*
8. *java.rmi*
9. *java.security*
10. *java.sql*
11. *java.text*
12. *java.util*

All these parts can be considered as separate inheritance hierarchies with the class *Object* as their root class and rest of their classes as the direct or indirect descendents of the class *Object*. As an example, figure 5.1 illustrates the inheritance hierarchy of *java.math*. The number of classes and average *DIT* values of these twelve hierarchies are illustrated in table 5.1.

Hierarchy	No. Of Classes	Average DIT	Standard Deviation in DIT
Java.applet	16	2.9375	1.436140662
java.awt	308	2.87012987	1.09307145
Java.beans	44	2.636363636	0.809562302
Java.io	82	3.341463415	1.239469343
java.lang	102	3.558823529	1.411224659
java.math	7	2.285714286	0.755928946
java.net	53	3	1.270977819
java.rmi	62	3.919354839	1.711086196
Java.security	115	3.139130435	1.330393403
java.sql	20	3.05	1.316894273
java.text	62	2.451612903	0.739459691
java.util	132	3.045454545	1.097280722

Table 5.1: List of the Number of Classes and Average *DIT* values for the twelve sub-hierarchies of the *java* part of Java 1.3 API.

Although we will present the observations based on the aggregate analysis of the *java* part as a whole, we will also present the graphs for some of the above-mentioned twelve sub-hierarchies to keep things simple and brief to explain. The Following sub-sections discuss

various observations that we have obtained as a result of our analysis. All the observations mentioned in these subsections are based on the cohesion values calculated using *CCM* (Class Connection Metric) and *MCCM* (Modified Class Connection Metric). Wherever needed, we will explicitly identify the graphs of *CCM* and *MCCM* to clarify the differences in their results.

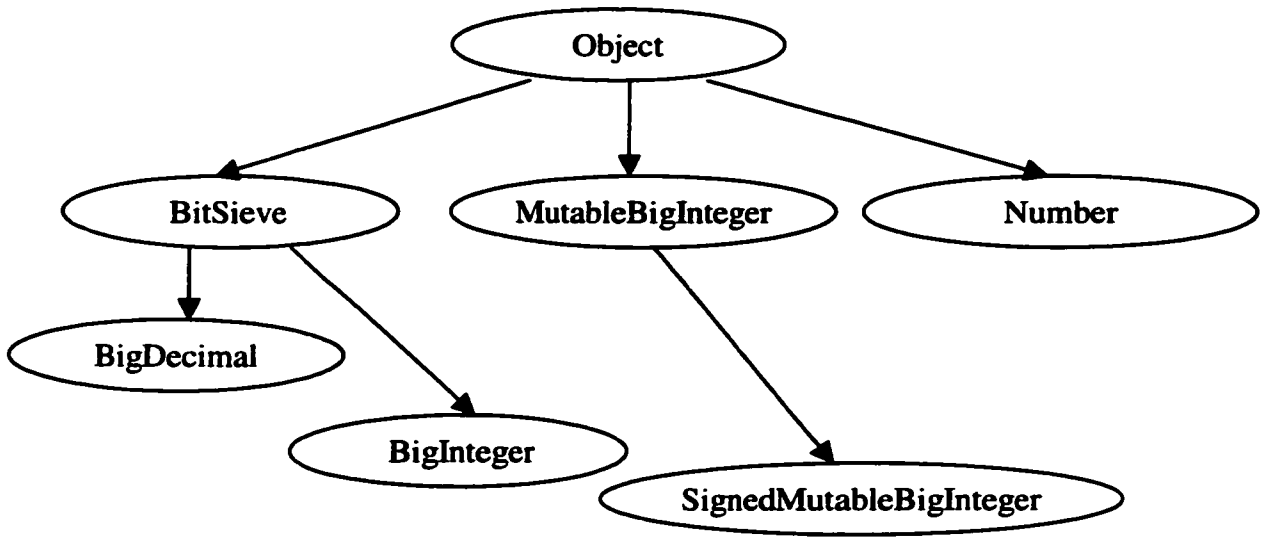


Figure 5.1: Inheritance hierarchy of *java.math.*

5.1.1 Observation No. 1

There is a considerable difference between the cohesion values obtained for the following two inheritance options:

1. *Without Inheritance*: Including only implemented methods and attributes in the analysis.
2. *With Inheritance*: Including also inherited attributes and methods in the analysis as well as the implemented attributes and methods.

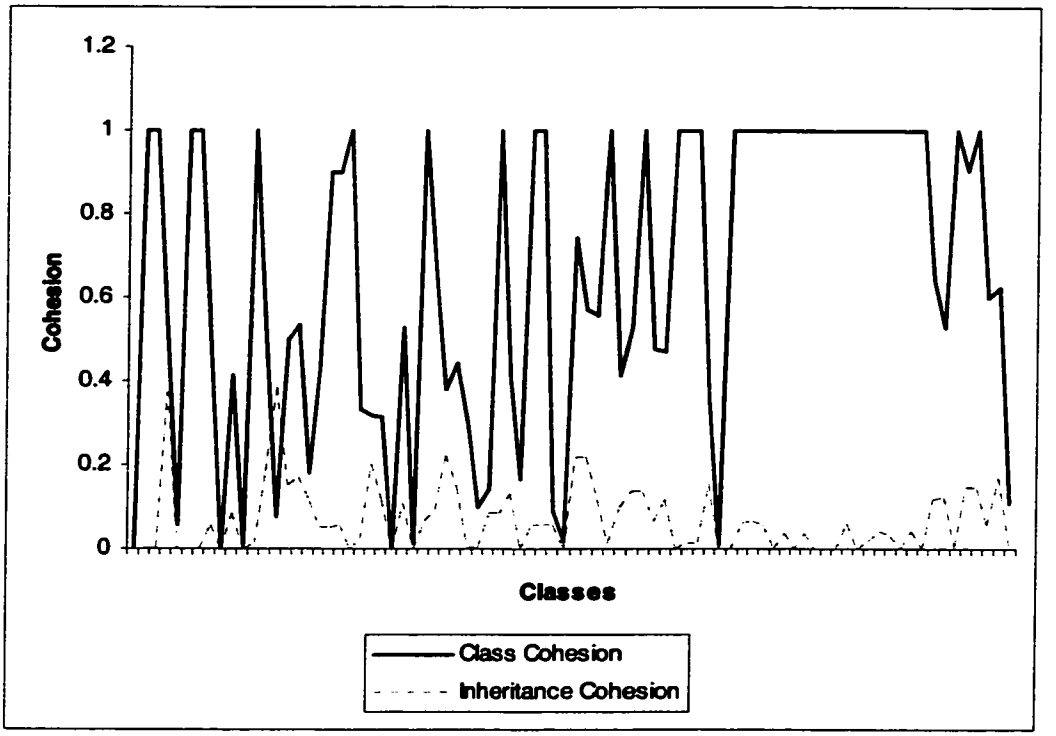


Figure 5.2: Graph showing the comparison between the Class and Inheritance Cohesion of the classes of *java.io* hierarchy.

Cohesion values for option 1 are quite higher than those for option 2. In other words, we can say that the Class Cohesion of the majority of the classes is higher than their Inheritance Cohesion. Figure 5.2 illustrates the graph for the *java.io* hierarchy that shows the comparison between the inheritance cohesion and class cohesion of its classes

calculated using *CCM*. This graph has two separate curves: one for the class cohesion and the other for the inheritance cohesion. It can be seen that the class cohesion is much higher than the inheritance cohesion for most of the classes.

This considerable difference between the class and inheritance cohesion is due to the reason that all the classes inherit from few top-level classes, which mostly have empty/abstract methods. For example, the root class is always the class *Object*. It has 10 methods, out of which 7 are empty. Since there are no connections among the methods of the class *Object*, its cohesion is zero. Every other class has to inherit from the class *Object*, but it may not override all of its methods. Therefore, its inheritance cohesion drops down drastically as the methods that it has inherited (but not overridden) from the class *Object* are unconnected to the rest of its methods and also to one another in its connection graph. For example, consider the class *ObjID*, which is a direct descendent of the class *Object*. Table 5.2 gives the complete list of the methods of the class *ObjID*. Its class cohesion is 0.933 and inheritance cohesion is 0.179. Therefore, its class cohesion is 5.6 times higher than its inheritance cohesion. Figure 5.3 (a) shows the connection graph of *ObjID* including only those methods that have been implemented/overridden in *ObjID*, which is quite strongly connected. Whereas, figure 5.3 (b) shows its connection graph including also its non-overridden inherited methods. It can be seen that the methods that it has inherited (but not overridden) from the class *Object* drastically reduce the overall connectivity of its connection graph. Due to this, its inheritance cohesion drops down quite below its class cohesion. Same is the case with nearly all the classes of the Java API.

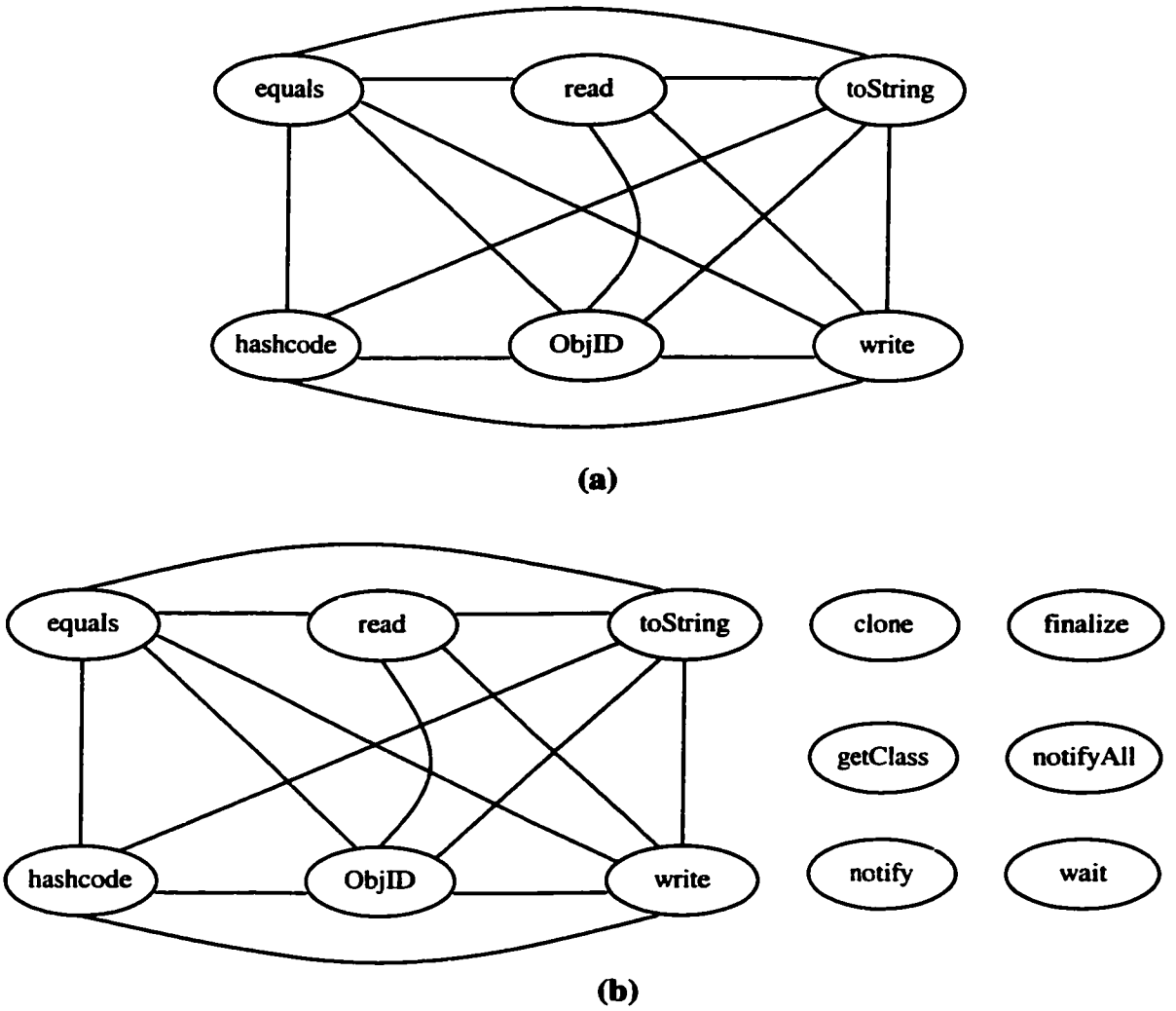


Figure 5.3: **(a)** Connection graph of the class *ObjID* without considering the inherited methods. **(b)** Connection graph of the class *ObjID* including the inherited methods also.

S. No.	Method Name	Original Class	Comments
1	Equals	Object	Inherited, overridden
2	hashcode	Object	Inherited, overridden
3	ObjID	ObjID	Constructor
4	Read	ObjID	Implemented
5	toString	Object	Inherited, overridden
6	Write	ObjID	Implemented
7	Clone	Object	Inherited, not overridden
8	Finalize	Object	Inherited, not overridden
9	getClass	Object	Inherited, not overridden
10	Notify	Object	Inherited, not overridden
11	notifyAll	Object	Inherited, not overridden
12	registerNatives	Object	Inherited, not overridden
13	Wait	Object	Inherited, not overridden

Table 5.2: List of methods of the class *ObjID*.

5.1.2 Observation No. 2

Slightly higher inheritance and class cohesion values are obtained when constructor methods are included in the analysis as compared to the case when constructor methods are excluded. Figure 5.4 illustrates a graph for *java.io* with two different curves – one for the inheritance cohesion values obtained from the calculations done by including the constructor methods and the other for the values obtained from the calculations done by excluding the constructor methods. It can be noticed that the curve for the option "With Constructor" is slightly higher than the curve for the option "without Constructor". This is due to the fact that the constructor method usually accesses most of the attributes of the

class and thus, shares attributes with most of the other methods of the class. Therefore, the inclusion of constructor method improves the over all connectivity of the class' connection graph and in turn improves its cohesion. For example, figure 5.3 (a) illustrates the connection graph of the class *ObjID* including its constructor method *ObjID*. Whereas, figure 5.5 shows the connection graph of *ObjID* excluding its constructor method. It is clear that the graph of figure 5.3 (a) is more tightly connected than the graph of figure 5.5. Due to this reason, the class cohesion of the class *ObjID* slightly reduces from 0.933 (including constructor) to 0.90 when the constructor method is excluded.

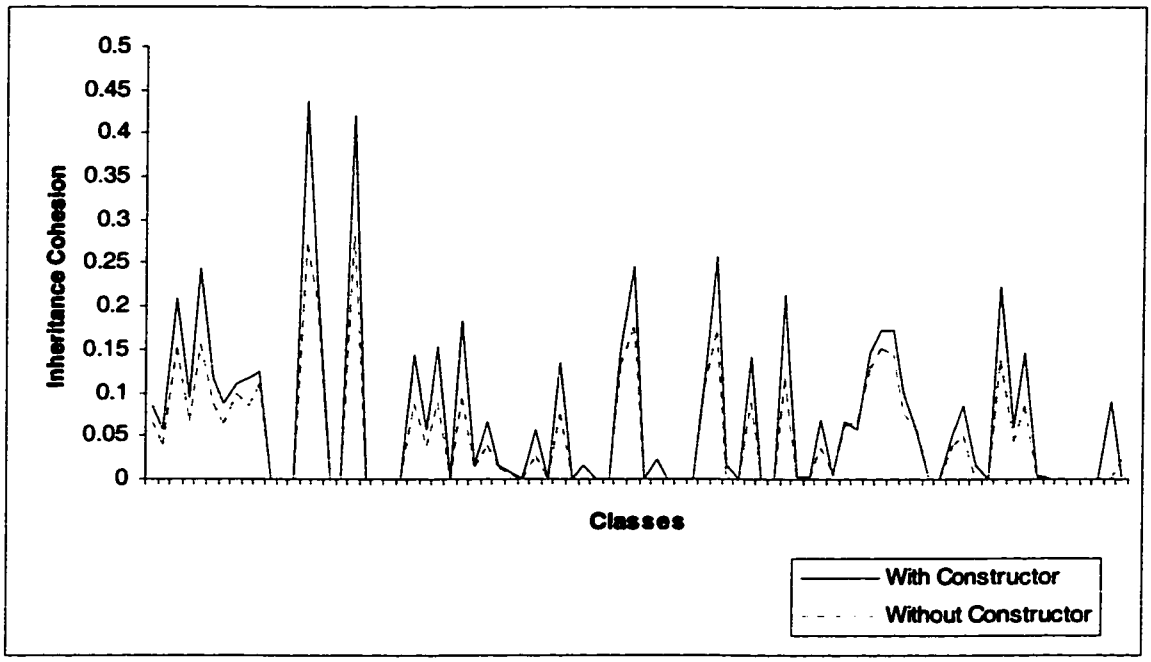


Figure 5.4: The graph showing the inheritance cohesion values of the classes of *java.io* hierarchy for the two cases: with constructor and without constructor.

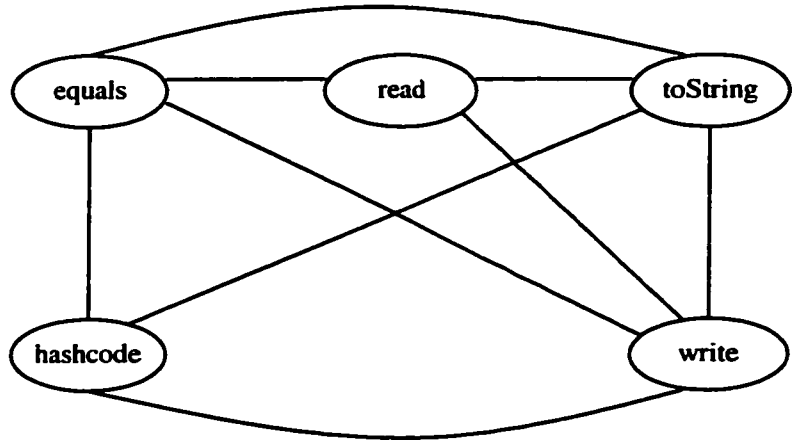


Figure 5.5: The connection graph of the class *ObjID* excluding the constructor method.

Inheritance Option	*Constructor Option*	
	Without Constructor	*With Constructor*
Including only Implemented Methods & Attributes	484	472
Including also Inherited Methods & Attributes	0	0
Including also Inherited Attributes but not Inherited Methods	511	492

Table 5.3: Number of classes with disconnected connection graphs for various inheritance and constructor options.

5.1.3 Observation No. 3

Table 5.3 shows the number of classes with disconnected connection graphs for various inheritance and constructor options. The analysis of the effects of inheritance on the connectivity of the connection graphs of the classes reveals the following:

- Connection graphs of Nearly fifty percent of the classes are connected when we do not include the inherited methods.
- Connection graphs of none of the classes are connected when we include inherited methods. This is due to the reason that every class directly or indirectly inherits from the class *Object* whose methods are completely disconnected to each other. Although, the inherited methods that are overridden by the subclass may get connected to the rest of its methods, the methods that are not overridden still remain disconnected to the other methods. For example, in case of the class *ObjID*, among the methods that it inherits from the class *Object*, only *hashcode*, *toString* and *equals* are overridden and thus, they are connected to the rest of its methods. Whereas, the remaining of the methods inherited from the class *Object* are completely disconnected as shown in figure 5.3 (b).
- The number of classes with disconnected connection graphs is lower for the case in which the constructor methods are included in the connection graphs as compared to the case in which the constructor methods are excluded. This is because the constructor method sometimes acts as a connecting method between the two or more connected components. In such a case, if the constructor method is excluded, the connection graph is divided into two or more connected components.

Level	Inheritance Cohesion	
	CCM	*MCCM*
1	0	0
2	0.073397	0.061813
3	0.085673	0.065858
4	0.079211	0.065459
5	0.025807	0.023349
6	0.020965	0.020598
7	0.033333	0.033333

Table 5.4: Inheritance Cohesion values for all the levels of the Java 1.3 API's hierarchy.

5.1.4 Observation No. 4

Inheritance cohesion is quite low at certain levels of the inheritance hierarchy – especially at the lower levels. Table 5.4 gives the average inheritance cohesion for each of the seven inheritance levels of the Java 1.3 API calculated using *CCM* and *MCCM*. It can be seen that if we exclude level 1, level 6 has the least and level 5 has the second least average inheritance cohesion among the rest of the levels. We have excluded level 1 from the discussion because it has only one class, i.e., *Object*, whose inheritance/class cohesion is zero. Figure 5.6 visually shows the average inheritance cohesion of all the levels of the hierarchy in the form of a graph for both *CCM* and *MCCM*. To find out the reason of this drop in the average inheritance cohesion at level 5 and 6, we examined some of the classes at these levels. Most of the classes at level 5 and 6 are either exception handling or error handling classes, all of which have a common ancestor class *Throwable*. The class

Throwable is a direct subclass of the class *Object* and has very poor class as well as inheritance cohesion. Its inheritance and class cohesion as calculated by *CCM* are 0.00794 and 0.00055, respectively. Since most of the classes at levels 5 and 6 indirectly inherit from the class *Throwable*, there inheritance cohesion is also low. For example, consider the class *ArrayIndexOutOfBoundException* (level 6). It is a leaf class in the inheritance tree and has only one method, i.e., the constructor method. The root to leaf path for *ArrayIndexOutOfBoundException* is shown in figure 5.7. It can be seen that it indirectly inherits from *Throwable* and there are three classes in between, i.e., *Exception*, *RuntimeException* and *IndexOutOfBoundException*. Due to the bad inheritance cohesion of the class *Throwable* and due to the fact that the classes from the class *Exception* to *ArrayIndexOutOfBoundException* in the traversal of figure 5.7, only have constructor methods (i.e., they don't override any of the methods that they inherit from *Throwable*), the inheritance cohesion of the class *ArrayIndexOutOfBoundException* is also quite low. We observed the similar pattern in almost all of the exception and error handling classes, which comprise the majority of the classes at level 5 and 6. Due to this reason, the average inheritance cohesion at level 5 and 6 is very low.

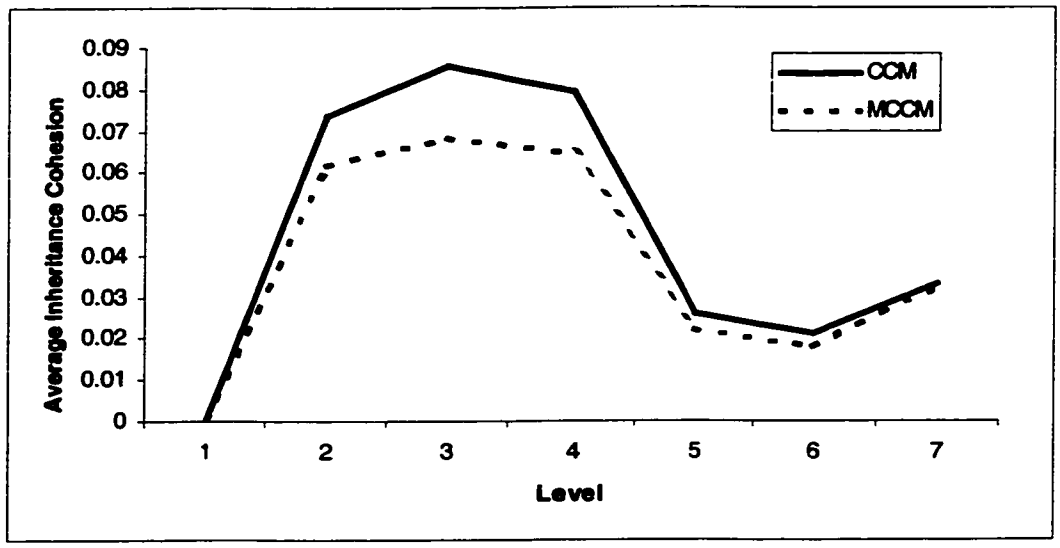


Figure 5.6: Average Inheritance Cohesion versus Inheritance Level graph of Java 1.3 API for both *CCM* and *MCCM*.

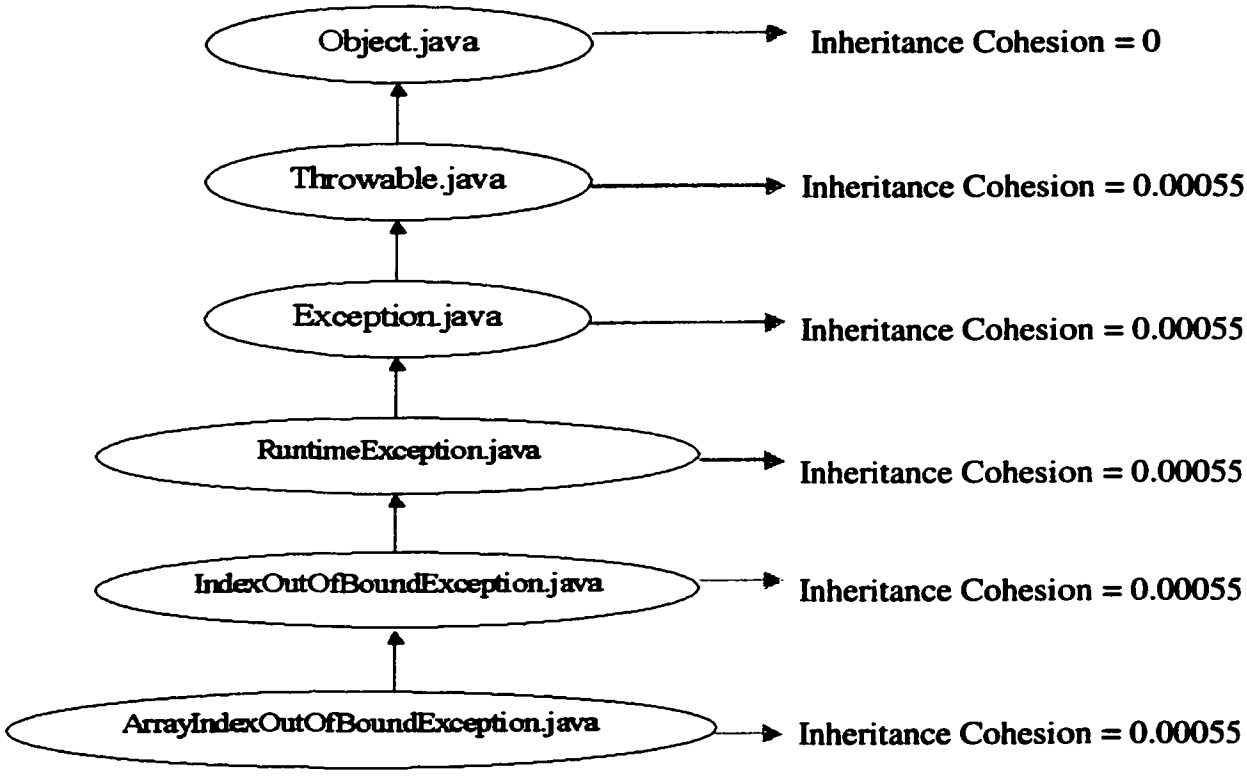


Figure 5.7: Root to leaf traversal for *ArrayIndexOutOfBoundException.java*

5.1.5 Observation No. 5

Although the overall average inheritance cohesion of the classes of Java 1.3 API is very low (0.0654), but there are still some classes with quite good inheritance cohesion. Table 5.5 lists the names and inheritance cohesion values of the ten top most classes. It can be seen that the inheritance cohesion of all these classes is well above the average.

S. No.	Class Name	Inheritance Cohesion (*CCM*)
1	SynchronizedSortedMap	0.5689655
2	SynchronizedSortedSet	0.53968257
3	MutableBigInteger	0.5090498
4	SynchronizedList	0.505291
5	SignedMutableBigInteger	0.4996506
6	RenderingHints	0.49275362
7	BitSet	0.45666668
8	SynchronizedSet	0.45454547
9	SynchronizedMap	0.45454547
10	String	0.43361345

Table 5.5: List of top ten classes with the highest inheritance cohesion calculated by *CCM*.

To find out the reason behind the good inheritance cohesion of these classes, we examined some of them. We found the following two important reasons for their good inheritance cohesion:

1. All of these classes have very good class cohesion values. In fact, most of them have class cohesion equal to 1 (i.e., maximum cohesion).
2. All of these classes override most of the empty methods that they inherit from the class *Object* and their other ancestors. Due to this, these inherited-overridden methods often get connected to the other methods and do not remain as disconnected vertices in the connection graph as they would have been if they haven't been overridden.

However, when we also penalize these classes for having re-implemented overridden methods (as described in section 3.3.2), the inheritance cohesion of many of these classes drops down by up to 30 % as shown in the table 5.6. Table 5.6 shows the *MCCM* values and *Penalty Factors* for the classes of table 5.5. The *Penalty Factor* is the penalty incurred on the class for having re-implemented overridden methods. It can be seen that some of the classes have *Penalty Factors* as high as 0.3. It means that these classes re-implement nearly 30 % of their inherited methods.

S. No.	Class Name	Penalty Factor	Inheritance Cohesion (*MCCM*)
1	SynchronizedSortedMap	0	0.5689655
2	SynchronizedSortedSet	0	0.53968257
3	MutableBigInteger	0.1	0.45814478
4	SynchronizedList	0.22727275	0.39045212
5	SignedMutableBigInteger	0.01960784	0.48985353
6	RenderingHints	0.3	0.34492752
7	BitSet	0.3	0.31966668
8	SynchronizedSet	0.09090906	0.41322318
9	SynchronizedMap	0.3	0.3181818
10	String	0.3	0.3035294

Table 5.6: *MCCM* values and *Penalty Factor* for the classes listed in table 5.5.

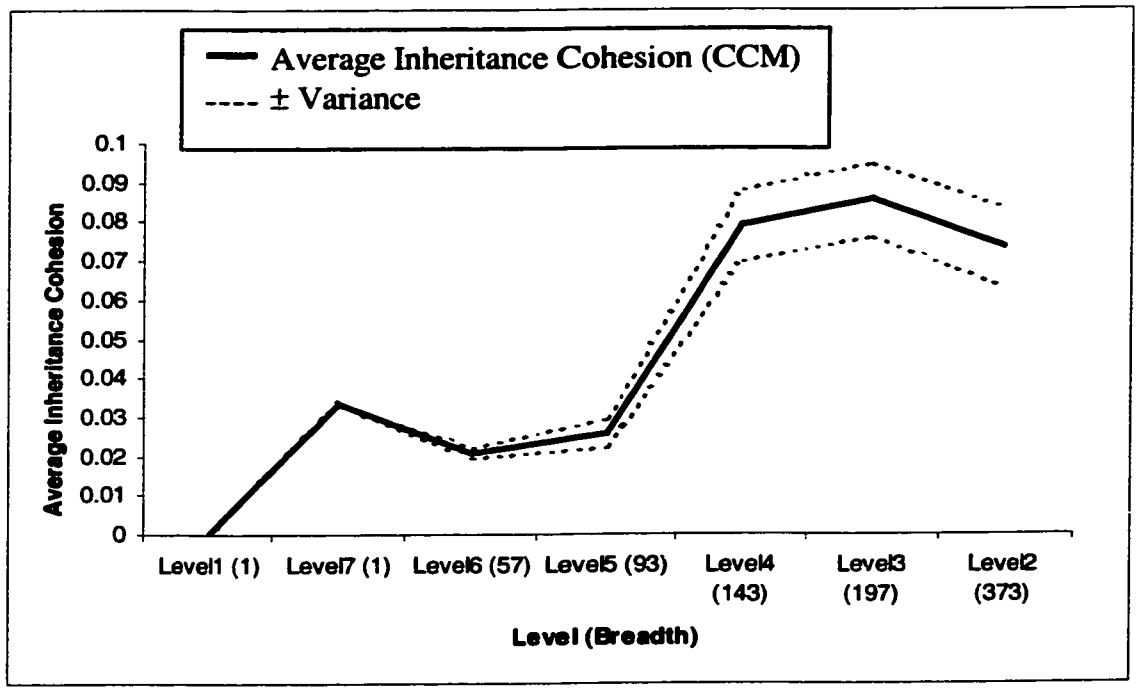


Figure 5.8: Average Inheritance Cohesion versus Breadth of Level graph for Java 1.3 API.

5.1.6 Observation No. 6

The levels of the inheritance hierarchy with greater breadths have higher average inheritance cohesion as shown by the graph of figure 5.8. By breadth of level, we mean the number of classes at that level of the inheritance tree. Analyzing the graph of figure 5.8, we notice the following observations:

1. Level 1 has zero average inheritance cohesion. This is because it has only one class, i.e., *Object* (root class), whose inheritance cohesion is zero.
2. Level 5 and 6 have very low average inheritance cohesion and their breadths are also less than the other levels. Low average inheritance cohesion values of level 5 and 6 are not related to their breadths, but actually these low values are due to the reason described in section 5.1.4.
3. Although the average inheritance cohesion at level 4, 3 and 2 is better than that of level 5 and 6, the difference between the class and inheritance cohesion at these levels is quite high. For example, the average class cohesion at level 2 is 0.44, whereas, the average inheritance cohesion is 0.073. Therefore, the average class cohesion at level 2 is 6.1 times higher than the average inheritance cohesion. In section 3.2.2, we discussed that sometimes the super-class is merely used as the container of service methods that are used by its subclasses and the designer includes more and more classes as the direct children of this "*Super-Cum-Service*" class to save the amount of coding. This leads to a high breadth of level just below this "*Super-Cum-Service*" class and adversely affects the inheritance cohesion of its subclasses, since they inherit all the methods of the super-class but use (or override)

only few of them. We notice the similar kind of problem at level 2 of the Java API. The super-class of the classes at this level is *Object*. Although the classes at level 2 inherit 9 methods from the class *Object*, they override only 2 out of these 9 methods on average. Due to this, their inheritance cohesion is drastically lower than their class cohesion.

4. Variance from the average in the values of inheritance cohesion is not very significant. The two lighter curves below and above the *Average Inheritance Cohesion Versus Breadth of Level* curve of figure 5.8 show the variance from the average values. It can be seen that the variance at levels 5 and 6 is very low. It means that most of the classes at these levels have inheritance cohesion close to the average inheritance cohesion. Whereas, the variance is higher at levels 2, 3, and 4, but it is still not very significant. For example, level 2 has the maximum variance, i.e., 0.01015, which is nearly 15 % of its average value.

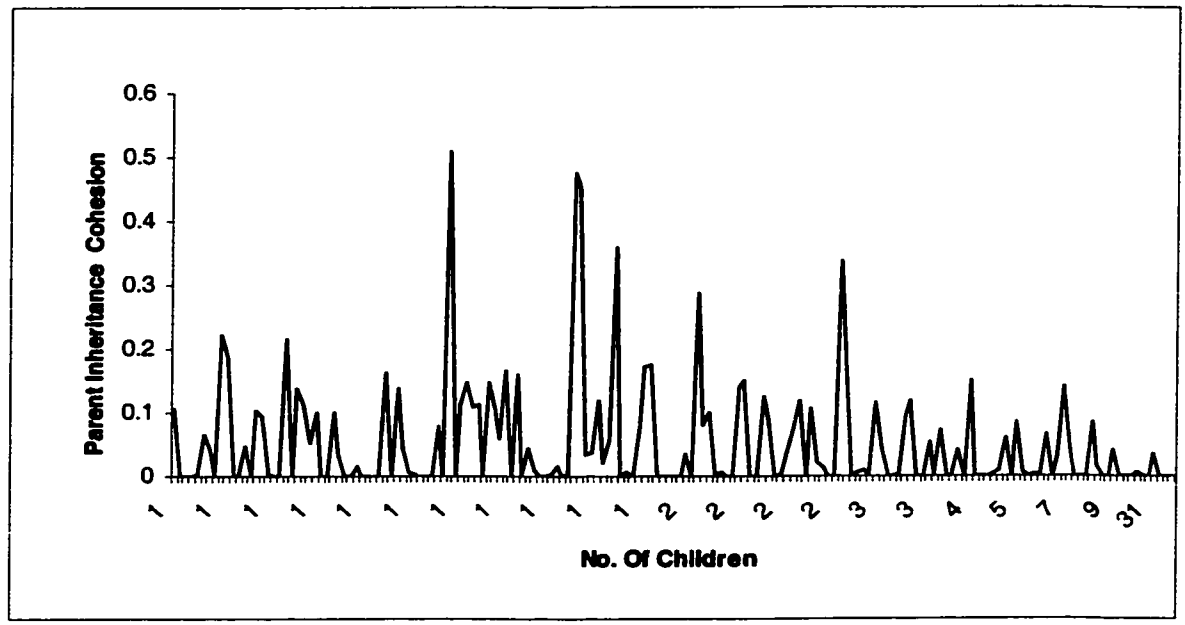


Figure 5.9: Parent Inheritance Cohesion Versus Number of Children graph for Java 1.3 API.

Class Name	No. of Children	Class Cohesion	Inheritance Cohesion
InputStream	9	0	0
Component	9	0.14497484	0.07124616
AWTEvent	9	0.003496504	0.001443001
AccessibleAWTComponent	10	0.007692308	0.011475409
AttributeValue	10	1	0.006060606
GeneralSecurityException	12	1	0.000555556
IOException	15	1	0.000555556
RemoteException	16	1	0.033333335
RuntimeException	22	1	0.000555556
Exception	31	1	0.000555556
Object	373	0	0

Table 5.7: *Inheritance* and *Class Cohesion* values of classes with 9 or more children.

5.1.7 Observation No. 7

As we discussed in section 3.2.3, when a super-class is used as a container of service methods that are used by its subclasses, its methods are usually unrelated to each other, since they have been implemented to fulfill the requirements of the subclasses rather than to accomplish the task of the super-class itself. Due to this lack of coherence among the methods of the super-class, its cohesion drops down. The classes with a large number of children are more likely to have this kind of characteristic.

To find out whether the cohesion of parent classes is related to the number of their children in Java 1.3 API, we plotted a graph of *Parent Inheritance Cohesion Versus the Number of Children* as shown in figure 5.9. It can be observed that the classes with a considerably higher number of children (i.e., more than 8 children) have poor inheritance cohesion. For example, consider the class *AWTEvent*. It is the super-class of all the event-handling classes of *java.awt* hierarchy. It has the following 9 direct descendents:

1. *ActionEvent*
2. *AdjustmentEvent*
3. *ComponentEvent*
4. *EmptyEvent*
5. *HierarchyEvent*
6. *InputMethodEvent*
7. *InvocationEvent*
8. *ItemEvent*

9. TextEvent

Since *AWTEvent* is the generic event class that covers all the above-mentioned specialized event classes, its methods are tailored according to the needs of its children. Due to this diversity in the methods of *AWTEvent*, its methods are not tightly related to one another. Therefore, its class as well as inheritance cohesion is quite low (class cohesion = 0.003496, inheritance cohesion = 0.001443).

Table 5.7 gives the list of classes with 9 or more children. Although all of these classes have quite low inheritance cohesion, but the class cohesion of some of these classes is as high as 1. This high class cohesion is due to the fact that these classes only have constructor methods (i.e., they have only one implemented method). It can be noticed that all but one of these classes are exception-handling classes. As we saw earlier, all the exception-handling classes are descendents of the class *Throwable*. Therefore, the bad inheritance cohesion of these classes actually reflects the bad class cohesion of the class *Throwable*. Since *Throwable* is the ancestor of a wide variety of error and exception-handling classes, its methods are very generic, as they have to cater for the requirements of all types of errors and exceptions. Due to this, its methods are not tightly connected to each other and thus it has low class cohesion.

5.1.8 Observation No. 8

To find out whether the inheritance cohesion of children is affected by their parent's cohesion, we plotted the graph of figure 5.10, which has two different curves. The darker curve shows the inheritance cohesion of the parent classes sorted in ascending order with respect to their number of children on the horizontal axis and lighter curve gives the average inheritance cohesion of the children of these parent classes. It is clear that both of these curves show similar pattern, i.e., the children of parents with good inheritance cohesion tend to have good inheritance cohesion and those of parents with poor inheritance cohesion tend to have poor inheritance cohesion. This is due to fact that besides inheriting the public/protected methods and attributes of the parent, the child also inherits the part of the parent's connection graph. Although the child's own connection graph (i.e., the connection graph made of the child's implemented methods only) may be strongly connected, but if the connection graph that it has inherited from its parent is poor, its overall connection graph (i.e., the connection graph made of the implemented plus inherited methods) will tend to be poor also. Due to this, the child will have poor inheritance cohesion. On the other hand, if the parent's connection graph is strongly connected, the child is also likely to have a strongly connected overall connection graph.

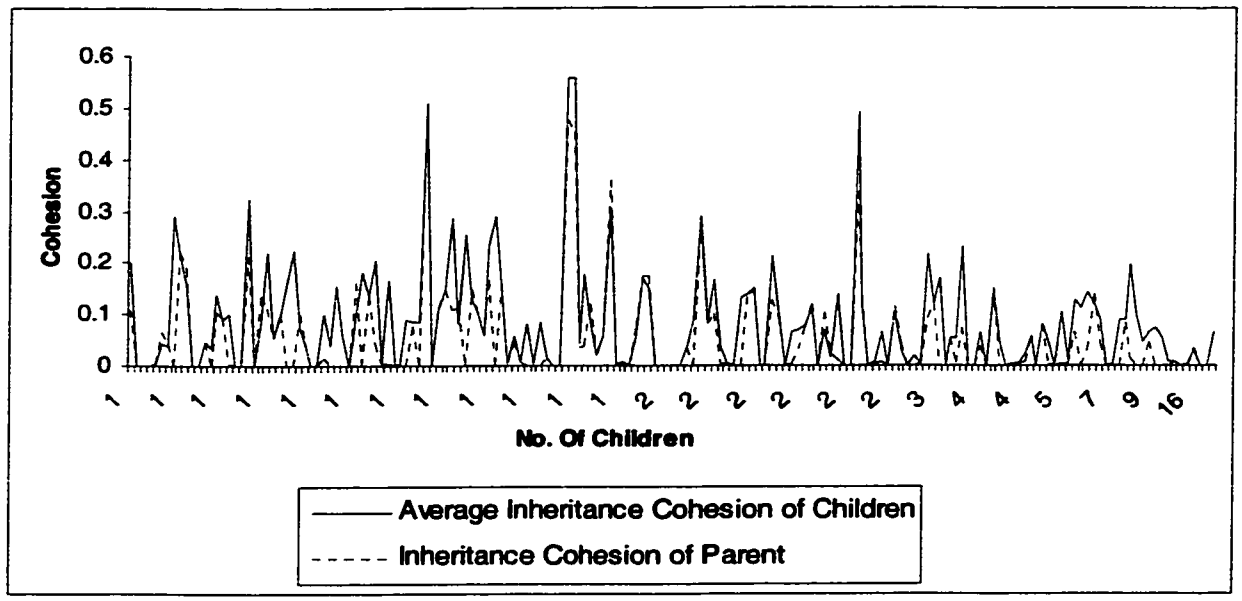


Figure 5.10: Comparison of the Inheritance Cohesion of Parents and the Average Inheritance Cohesion of their Children.

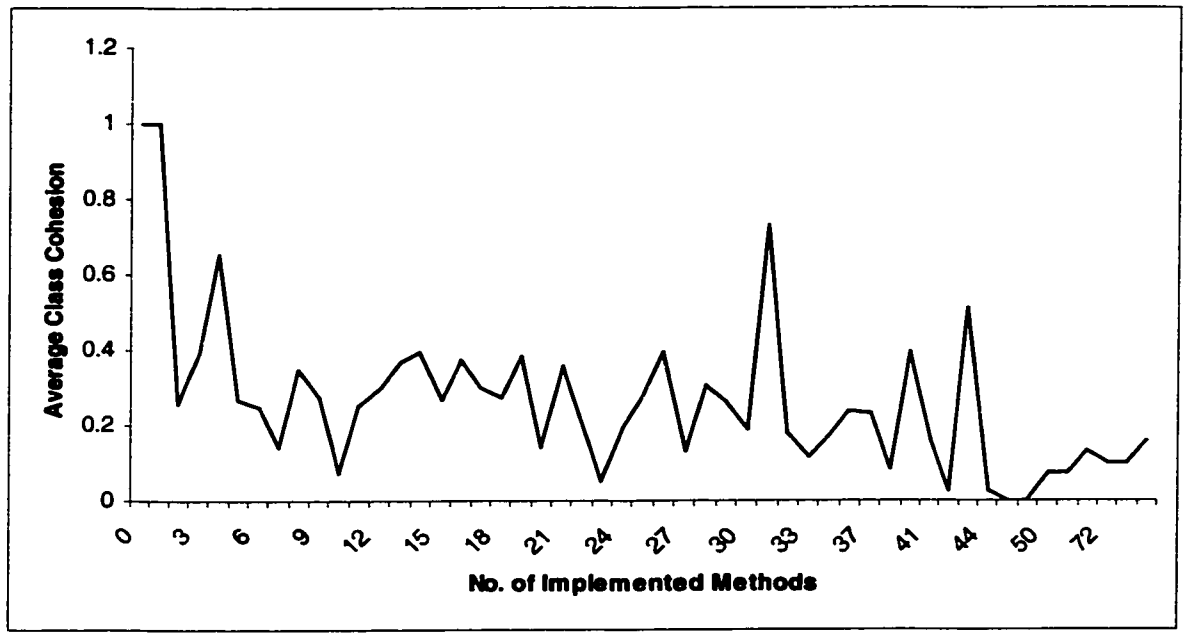


Figure 5.11: Graph of Average Class Cohesion Versus Number of Implemented Methods for Java 1.3 API.

Class Name	No. of Implemented Methods	Class Cohesion
TreeMap	51	0.16705883
Class	53	0.09361393
TextLayout	54	0.1955276
BeanContextSupport	57	0.18107769
ObjectInputStream	57	0.18421052
BigInteger	76	0.27684212
Container	76	0.11387388
Component	172	0.14497484

Table 5.8: List of classes with more than 50 implemented methods.

5.1.9 Observation No. 9

We plotted the graph of average class cohesion versus the number of implemented methods as illustrated in figure 5.11 to find out the effect of the number of implemented methods on class cohesion of the classes. The horizontal axis represents the number of implemented methods sorted in increasing order and the vertical axis gives the values of average class cohesion of the classes having these numbers of implemented methods. Although we do not notice any particular pattern in this graph, but it can be seen most of the classes with 50 or more implemented methods have quite low class cohesion. To illustrate this, table 5.8 presents a list of classes with more than 50 methods. It can be noticed that the class cohesion of these classes is quite below the average. The average class cohesion of the

classes having more than 50 implemented methods is 0.17, whereas, the overall average class cohesion is 0.5567.

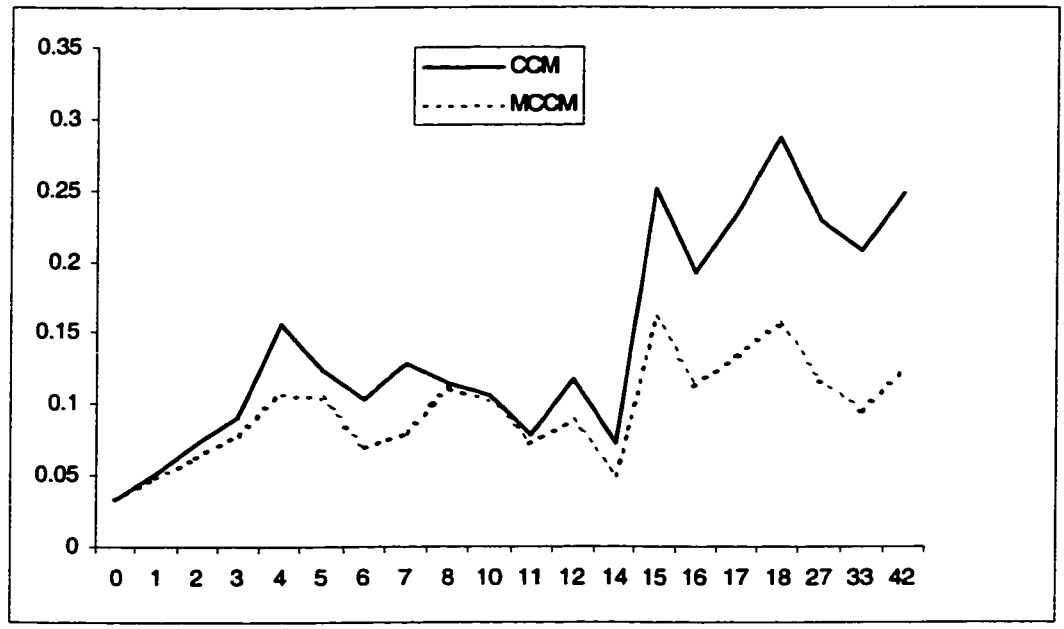


Figure 5.12: The graph of Average Inheritance Cohesion Versus the Number of Overridden Methods of *java.awt* for both *CCM* and *MCCM*.

5.1.10 Observation No. 10

Inheritance cohesion (as calculated by *CCM*) of classes having greater number of overridden methods is higher. Figure 5.12 shows the graph of average inheritance cohesion against the number of overridden methods of *java.awt* hierarchy for both *CCM* and *MCCM*. It can be seen that the classes with more than 14 overridden methods have better inheritance cohesion as measured by *CCM* (darker curve).

As we said earlier, all the classes of Java 1.3 API have to inherit from few top-level classes including the class *Object*. Since these top-level classes mostly have empty/abstract methods, their connection graphs are very poorly connected. For example, the connection graph of the class *Object* is completely disconnected. The subclasses inherit these connection graphs, if they do not override the empty methods inherited from their ancestors. On the other hand, if a subclass overrides the empty inherited methods, these methods often get connected to the other methods and do not remain as disconnected vertices in the connection graph as they would have been if they haven't been overridden. Therefore, this overriding of inherited methods often improves the inheritance cohesion of the subclasses. However, when we also penalize the classes for having re-implemented overridden methods, as done by *MCCM*, the inheritance cohesion doesn't show improvement as the number of overridden methods increases as shown by the lighter curve of figure 5.12.

5.1.11 Observation No. 11

To find out whether the size of the hierarchy affects its average inheritance cohesion or not, we plotted the following two types of graphs:

1. The graph of average inheritance cohesion of hierarchies against their number of classes (figure 5.13). In this graph the horizontal axis represents the hierarchies sorted in ascending order with respect to their number of classes and vertical axis gives the average inheritance cohesion of these hierarchies.

2. The graph of average inheritance cohesion of hierarchies against their number of methods (figure 5.14). In this graph the horizontal axis represents the hierarchies sorted in ascending order with respect to their number of methods and vertical axis shows the average inheritance cohesion of these hierarchies.

It can be seen that the average inheritance cohesion of hierarchies does not depend on their size either with respect to their number of classes or with respect to their number of methods.

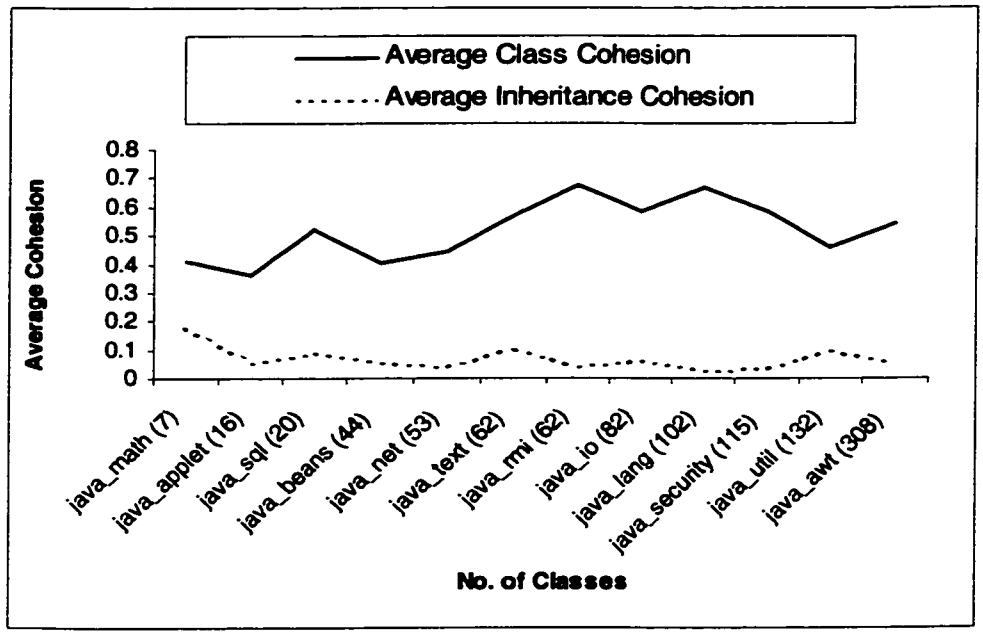


Figure 5.13: The graph of average inheritance cohesion of hierarchies against their number of classes.

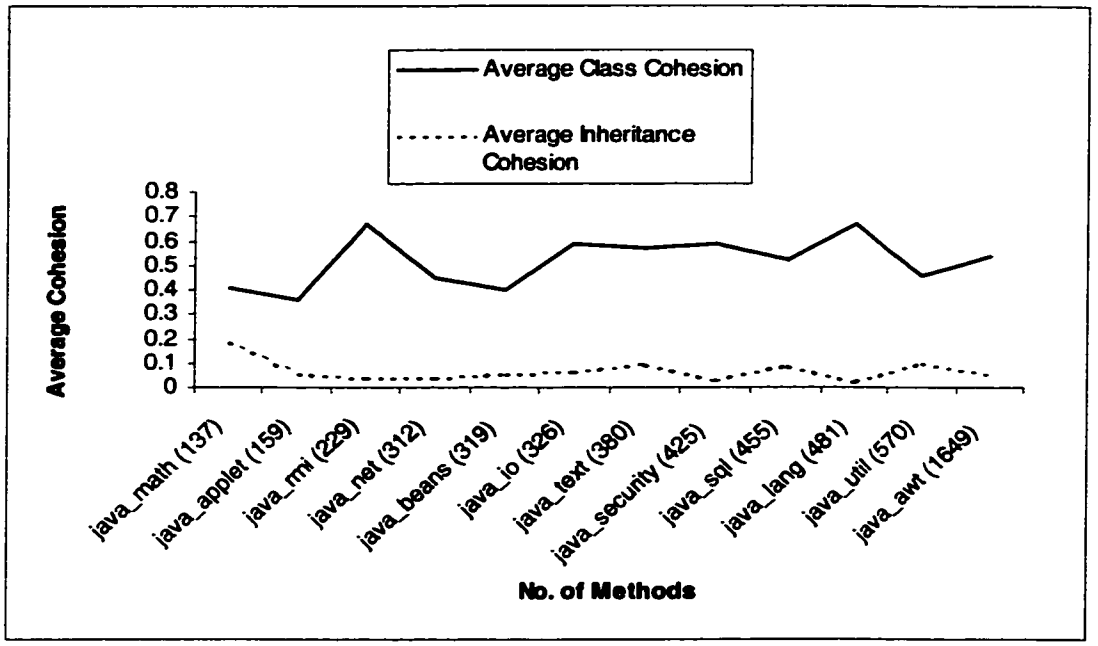
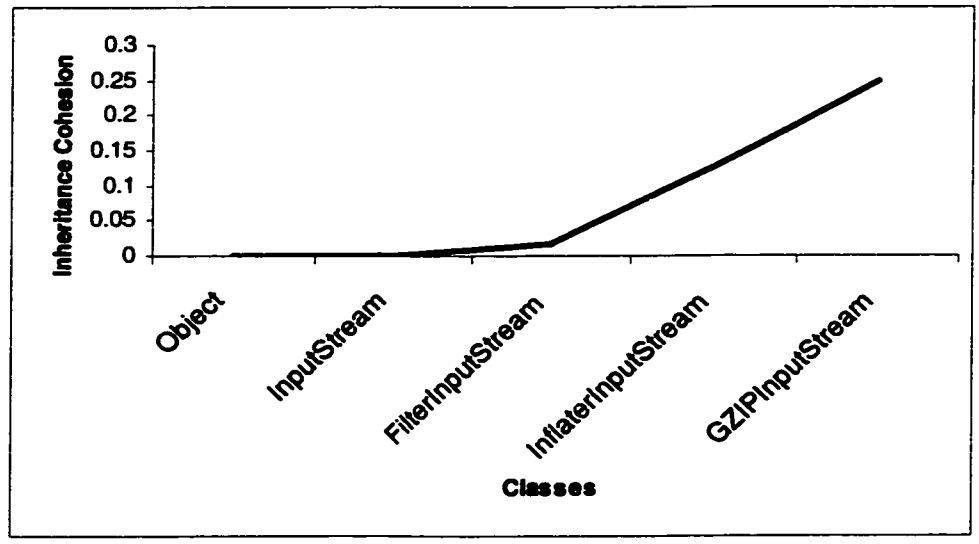


Figure 5.14: The graph of average inheritance cohesion of hierarchies against their number of methods.

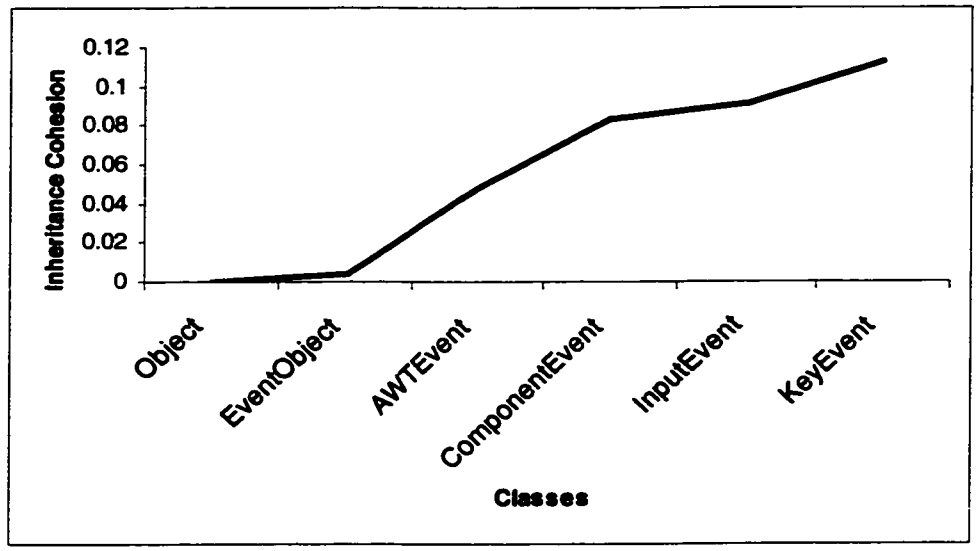
5.1.12 Observation No. 12

While tracing the path from the root class to different leaf classes with above average inheritance cohesion, it is observed that the inheritance cohesion (as measured by CCM) rises as we move down from the root to the leaf. For example, figure 5.15 (a) shows the root to leaf traversal graph for the class *GZIPInputStream*. The horizontal axis lists the classes along the root to leaf path, starting from the root class (*Object*) and ending at the leaf class (*GZIPInputStream*), whereas, the vertical axis gives the inheritance cohesion of the corresponding classes as measured by *CCM*. A similar graph for the class *KeyEvent* is shown in figure 5.15 (b). In both of these example graphs, it can be seen that the inheritance cohesion rises steadily as we move from the root class to the leaf class. This steady rise in the inheritance cohesion is due to the fact that the top-level classes usually

have empty/abstract methods and thus, their connection graphs are weak, but as we go down, the subclasses override more and more of these empty methods. Due to this overriding of inherited methods, the inheritance cohesion of the subclasses improves over their super-classes.



(a)



(b)

Figure 5.15: Root to leaf path traversal for **(a)** *GZIPInputStream* and **(b)** *KeyEvent*.

5.1.13 Observation No. 13

To find out the number of leaf classes (or terminal classes) at each level of the hierarchy, we plotted the graph of number of leaf and non-leaf classes against the levels of inheritance hierarchy as illustrated in figure 5.16. Examining the graph of figure 5.16, we can notice following observations:

- The number of leaf classes is much higher than the number of non-leaf classes.
- Level 2 has the highest number of leaf classes, i.e., 292. This means 292 terminal classes of the API are direct descendents of the class *Object*.
- Nearly 64 % of the leaf classes are at level 2 and 3.

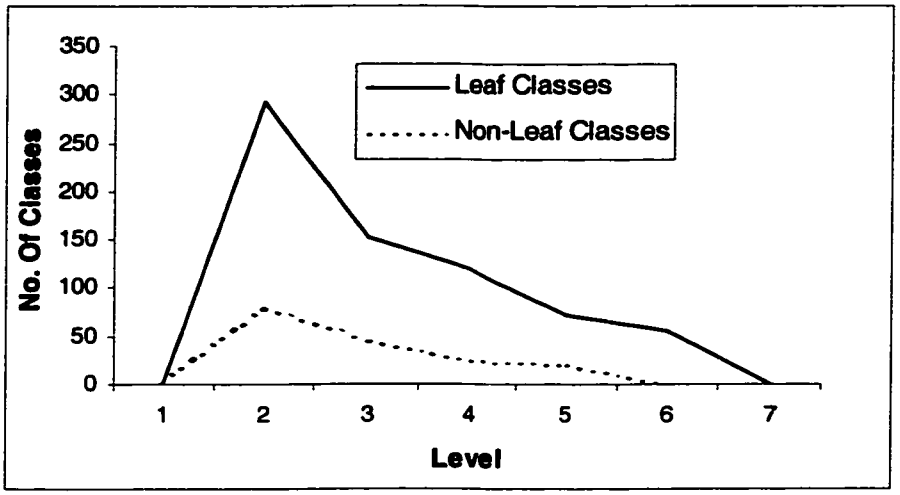


Figure 5.16: Graph of Number of Leaf and Non-Classes at each level of the inheritance hierarchy of Java1.3 API.

level	**Leaf Classes**			**Non-Leaf Classes**		
	No. of Classes	*Average Inheritance Cohesion*	*Average Class Cohesion*	*No. of Classes*	*Average Inheritance Cohesion*	*Average Class Cohesion*
1	0	0	0	1	0	0
2	292	0.07889	0.472944	81	0.04992	0.305755
3	152	0.08623	0.484619	45	0.07445	0.3889801
4	119	0.07783	0.579064	24	0.06689	0.4619222
5	73	0.02567	0.833981	20	0.02629	0.7376797
6	56	0.02030	0.871837	1	0.03333	1
7	1	0.03333	1	0	0	0

Table 5.9: Numbers and Average Cohesion values of Leaf and Non-Leaf Classes at all the levels of the inheritance hierarchy of Java1.3 API.

Table 5.9 shows the number of leaf classes, number of non-leaf classes and average inheritance and class cohesions of leaf and non-classes for each level of the hierarchy. It can be seen that the average inheritance cohesion of leaf classes is higher than that of non-leaf classes at all levels except level 5 and 6. However, the average inheritance cohesion is quite lower than the average class cohesion for both leaf and non-leaf classes at all levels.

5.2 Comparison of Cohesion Metrics

Since the cohesion metrics discussed in chapter 2 and 3 differ to some extent from each other with respect to their method of measuring cohesion, the types of interactions they consider and their way of visualizing cohesion, we can expect varying results from these metrics. To find out the differences among these cohesion metrics, we have compared the

results, which were obtained from these metrics from the analysis of Java 1.3 API. The observations that we noticed in this regard are presented in the following subsections.

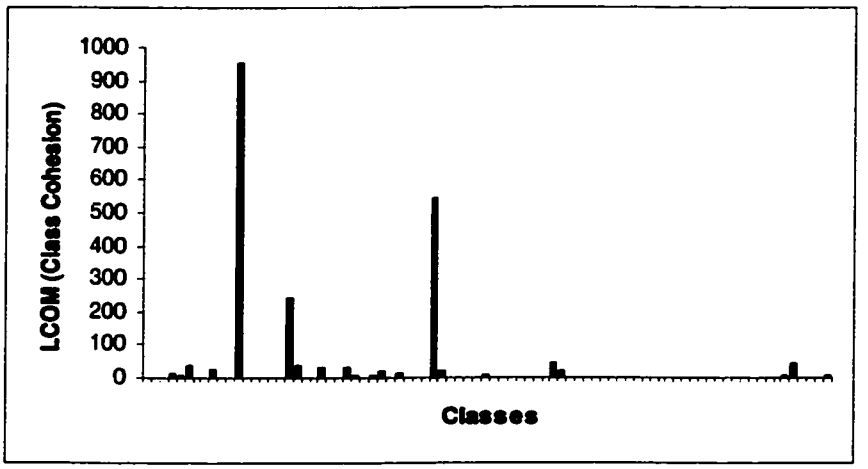


Figure 5.17: Bar chart showing the *LCOM* values for the classes of *java.io* hierarchy.

5.2.1 Observation No. 1

Class cohesion values obtained by *LCOM* for more than 50% classes are zero. Figure 5.17 shows the bar chart of the class cohesion values obtained by *LCOM* for *java.io* hierarchy. It can be seen that most of the classes have zero class cohesion. As we saw in section 2.1, if the number of pairs of methods that share one or more attributes is greater than the number of pairs of methods that do not share any attribute, *LCOM* is set to 0. Due to this, most of the classes get 0 *LCOM* value, which corresponds to the perfect *LCOM* cohesion. However, even if a class has *LCOM* value of 0 (i.e., perfect *LCOM* cohesion), it is not necessary that this class is perfectly cohesive. This is due to the reason that the *LCOM* measures are not

very sensitive. For example, if we consider the *LCOM* version of Hitz and Montazeri [Hitz96], we observe that it only counts the number of connected components of the class' connection graph, but doesn't consider the density of connections in these connected components.

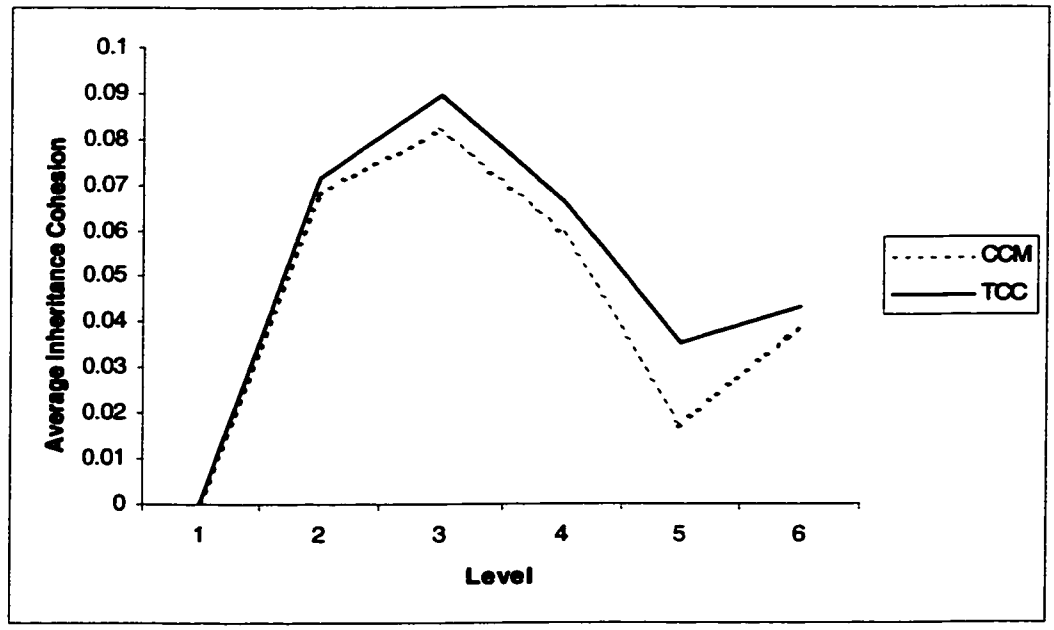
5.2.2 Observation No. 2

CCM and *TCC* give quite similar patterns on different result graphs. Figure 5.18 (a) and (b) show the *Average Inheritance Cohesion Versus Inheritance Level* and *Average Inheritance Cohesion Versus Number of Methods* graphs, respectively, of *java.awt* hierarchy for both *CCM* and *TCC*. It can be seen that the curves for *CCM* and *TCC* show quite similar patterns on the graphs of both figure 5.18 (a) and (b). This similarity is due to the fact that both *TCC* and *CCM* concentrate on the number or density of the number of connections among the methods in the class' connection graph. *TCC* is the ratio of actual inter-method connections to the maximum possible number of inter-method connections. Whereas, *CCM* also divides the ratio of actual inter-method connections to the maximum possible number of inter-method connections by the number of connected components of the class' connection graph. Still there is some difference between the results of *CCM* and *TCC* due to the following reasons:

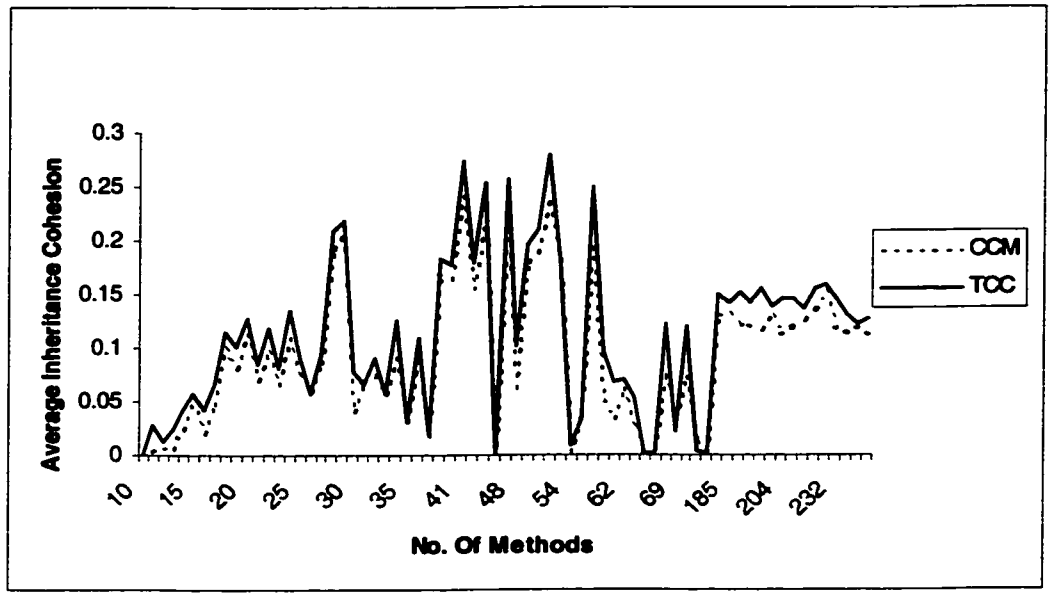
1. The methods of finding out whether the two methods *A* and *B* are connected to each other or not are different in case of *TCC* and *CCM*. *TCC* considers two methods *A* and *B* connected if they either access one or more attribute in common or any of *A* or *B* invokes the other. Whereas, *CCM* considers two methods *A* and *B* connected if

they either access one or more attribute in common or invoke one or more methods in common.

2. *CCM* also penalizes the class for the number of connected components of its connection graph, i.e., it also divides the ratio of actual inter-method connections to the maximum possible number of inter-method connections by the number of connected components of the class' connection graph. Due to this, it can be noticed that the curve for *CCM* is always lower than the curve for *TCC* in the graphs of both figure 5.18 (a) and (b).

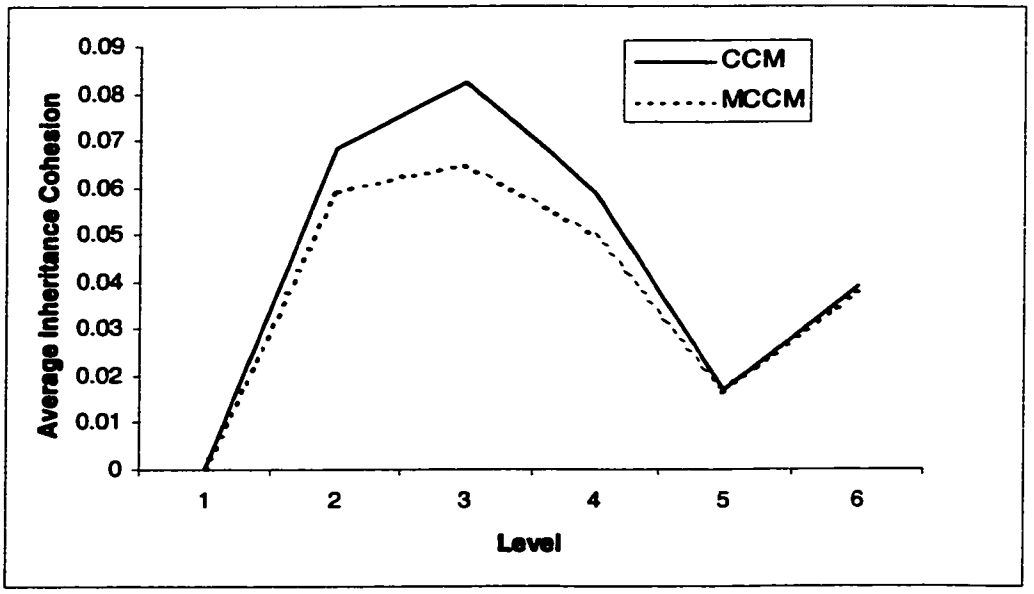


(a)

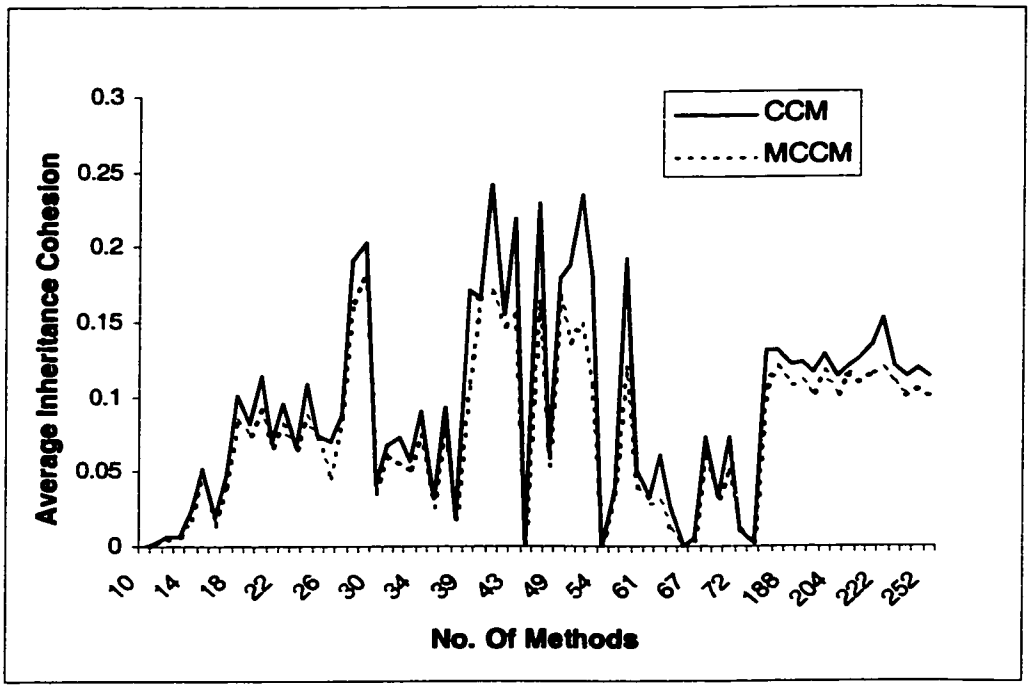


(b)

Figure 5.18: (a) Average Inheritance Cohesion Versus Inheritance Level and (b) Average Inheritance Cohesion Versus Number of Methods graph of *java.awt* for both *CCM* and *TCC*.

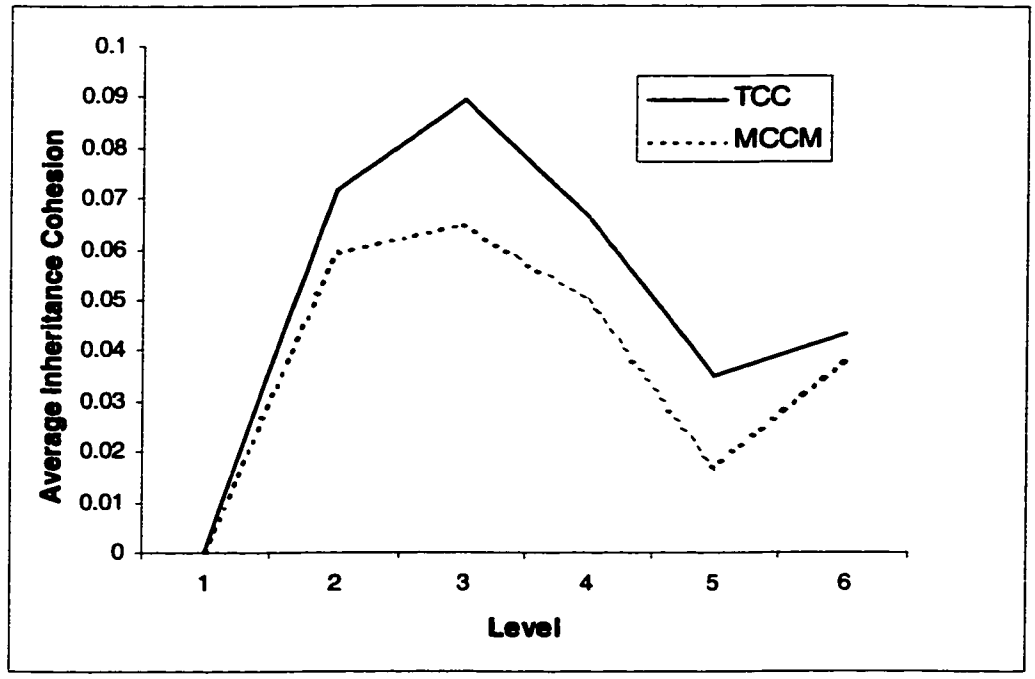


(a)

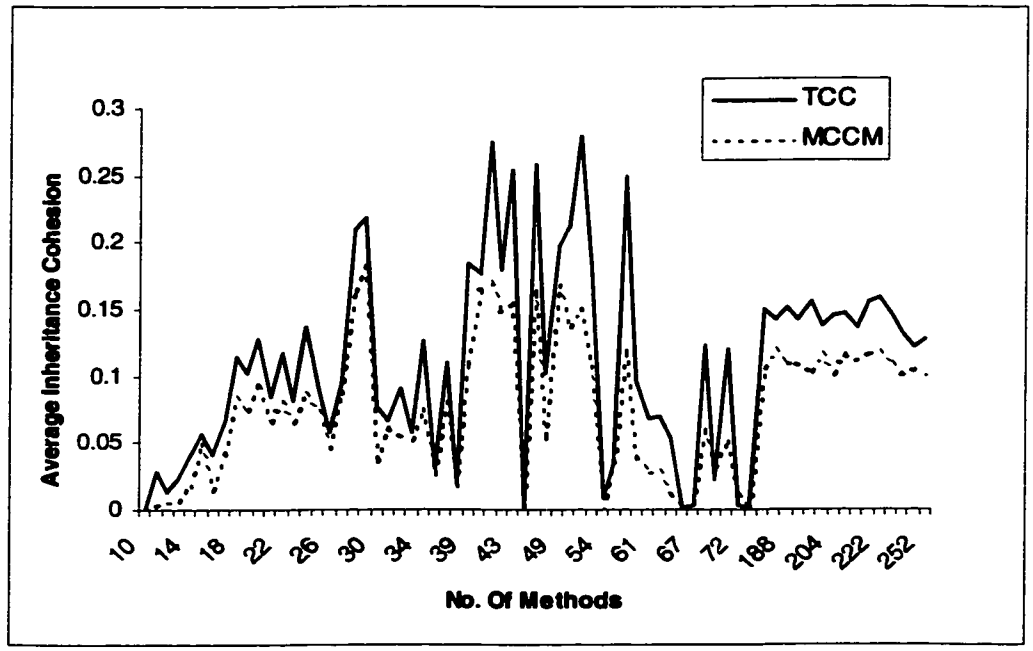


(b)

Figure 5.19: (a) Average Inheritance Cohesion Versus Inheritance Level and (b) Average Inheritance Cohesion Versus Number of Methods graph of *java.awt* for both *CCM* and *MCCM*.



(a)



(b)

Figure 5.20: (a) Average Inheritance Cohesion Versus Inheritance Level and (b) Average Inheritance Cohesion Versus Number of Methods graph of *java.awt* for both *TCC* and *MCCM*.

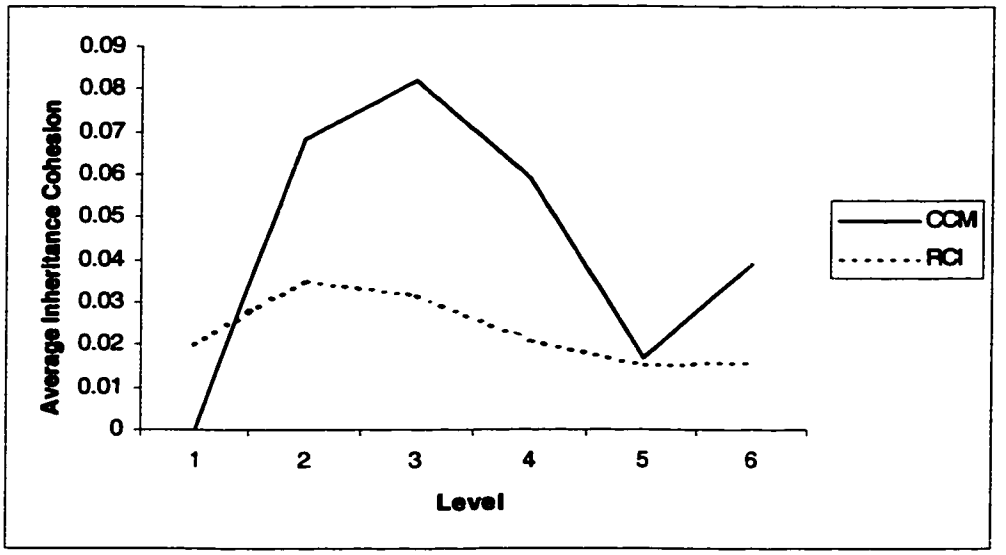
5.2.3 Observation No. 3

Figure 5.19 (a) and (b) show the *Average Inheritance Cohesion Versus Inheritance Level* and *Average Inheritance Cohesion Versus Number of Methods* graphs, respectively, of *java.awt* hierarchy for both *MCCM* and *CCM*. It can be observed that although the graphs of *MCCM* show similar patterns to those of *CCM*, there is a noticeable difference between the values obtained by *MCCM* and *CCM*, i.e., the cohesion values of *MCCM* are mostly lower than those of *CCM*. This is due to the reason that *MCCM* is obtained by multiplying *CCM* by the factor *1 – Penalty Factor*(*C*), which cannot be more than *1*. Therefore, *MCCM* value can never be more than that of *CCM*. In case of Java 1.3 API, *MCCM* values obtained for most of the classes are noticeably lower than the *CCM* values of these classes. This shows that there is a significant presence of re-implemented overridden methods in the classes of Java 1.3 API. However, in the graph of figure 5.19 (a), we notice that the average inheritance cohesion values obtained by *MCCM* and *CCM* are nearly same at level 5 and 6. This is due to the fact that nearly all the classes at level 5 and 6 do not have re-implemented overridden methods, since they only have constructor methods.

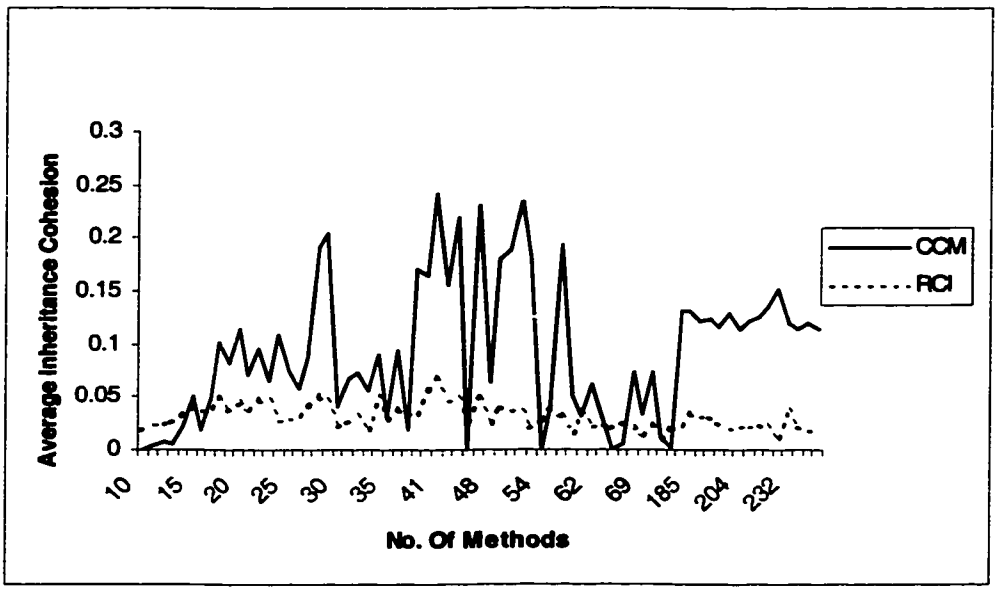
It can be noticed from the graphs of figure 5.20 (c) and (d) that the similarity between the patterns of the graphs of *MCCM* and *TCC* is not as much as the similarity between the patterns of the graphs of *CCM* and *TCC*. This shows that the inclusion of *Penalty Factor* in the definition of *MCCM* has made it quite different from *TCC*.

5.2.4 Observation No. 4

RCI gives different patterns on various result graphs as compared to *CCM* and *TCC*. Figure 5.21 (a) and (b) show the *Average Inheritance Cohesion Versus Inheritance Level* and *Average Inheritance Cohesion Versus Number of Methods* graphs, respectively, of *java.awt* hierarchy for both *RCI* and *CCM*. It can be noticed that the curves for *RCI* show quite different patterns than those for *CCM* on the graphs of both figure 5.21 (a) and (b). This difference is due to the reason that *RCI* concentrates on counting the number of *attribute-method* interactions (referred as *DM-interactions* in section 2.5) rather than counting the *method-method* connections as done by *CCM* and *TCC*.



(a)

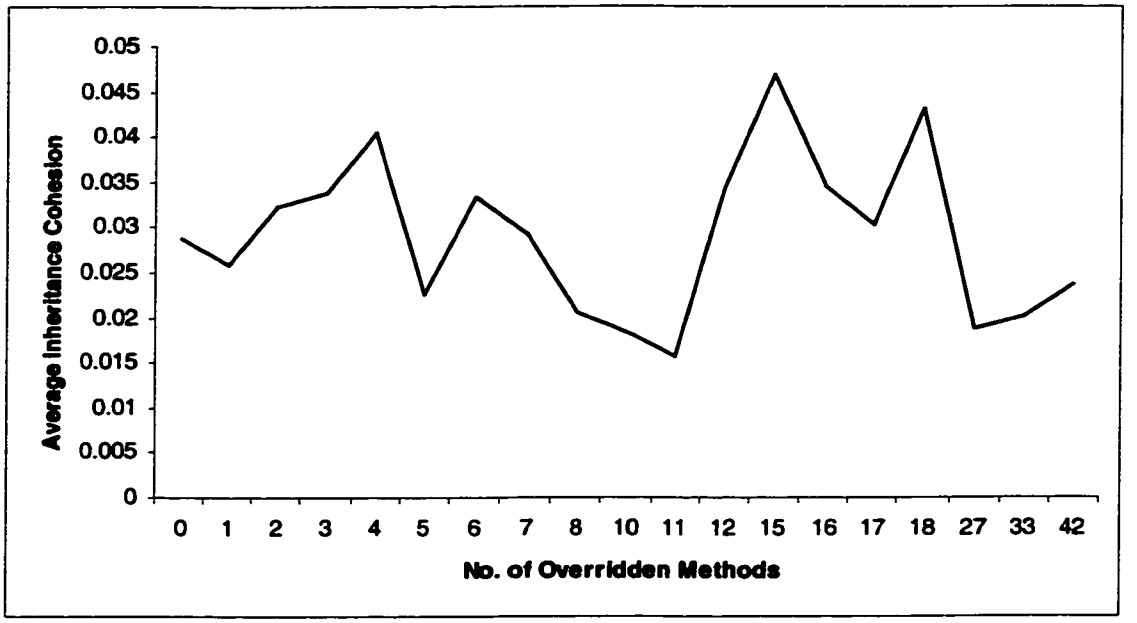


(b)

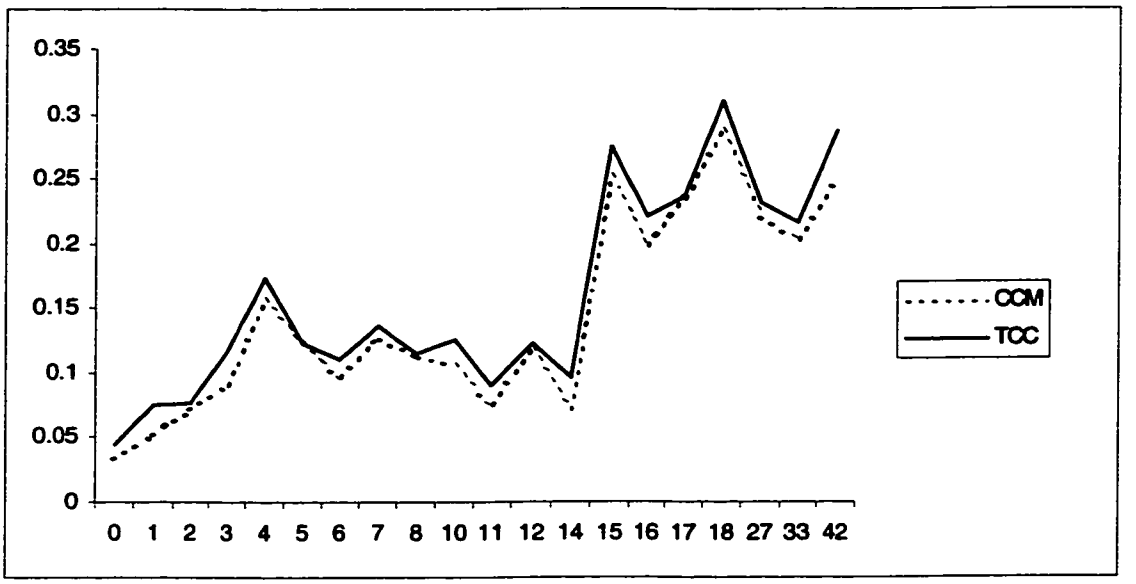
Figure 5.21: (a) Average Inheritance Cohesion Versus Inheritance Level and (b) Average Inheritance Cohesion Versus Number of Methods graph of *java.awt* for both *CCM* and *RCI*.

5.2.5 Observation No. 5

RCI shows no improvement as the number of overridden methods increases as illustrated in the *Average Inheritance Cohesion Versus Number of Overridden Methods* graph of figure 5.22 (a). Whereas, the inheritance cohesion values measured by *CCM* and *TCC* improve as the number of overridden methods rises as shown in the graph of figure 5.22 (b). We saw in section 5.1.10 that a class with a large number of overridden methods is likely to have a strongly connected connection graph, but since *RCI* doesn't depend on the connection graph (or on the inter-method connections) its value doesn't improve with the rise in the number of overridden methods.



(a)



(b)

Figure 5.22: Average Inheritance Cohesion Versus Number of Overridden Methods graph of *java.awt* hierarchy **(a)** for *RCI* and **(b)** for both *CCM* and *TCC*.

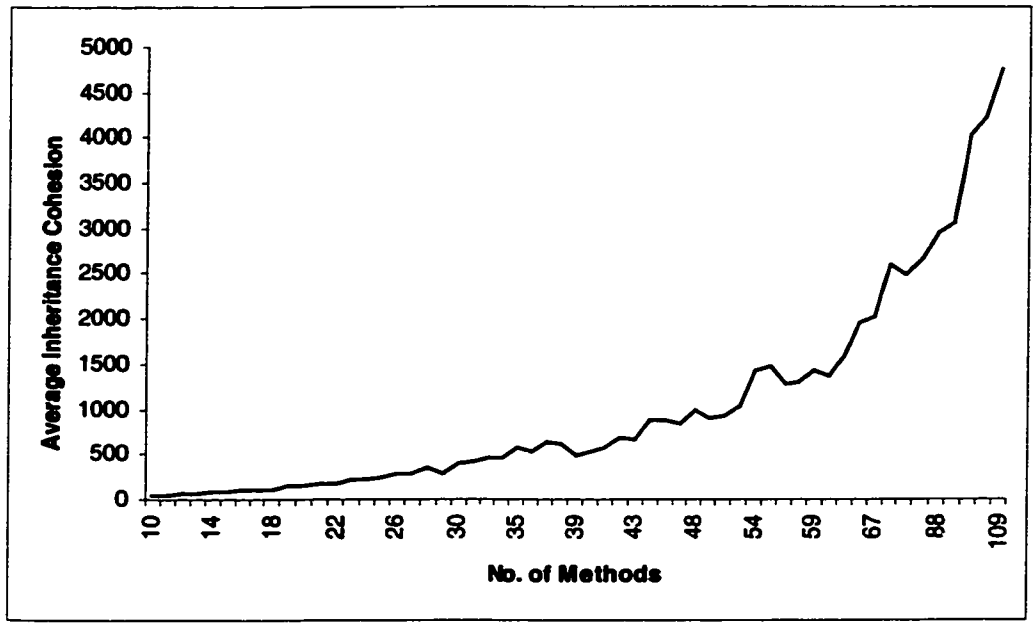


Figure 5.23: Average Inheritance Cohesion Versus Number of Methods graph of *java.awt* hierarchy for *LCOM*.

5.2.6 Observation No. 6

The inheritance cohesion as measured by *LCOM* rises steadily with the number of methods of a class (inherited plus implemented) as illustrated in the *Average Inheritance Cohesion Versus Number of Methods* graph of figure 5.23. The rise in the *LCOM* value as the number of methods increases means that the inheritance cohesion actually falls, since *LCOM* is an inverse cohesion measure. This behavior of *LCOM* is quite different from that of *CCM* and *TCC*, for which the inheritance cohesion values neither rise nor fall steadily with the number of methods, instead they show quite a random pattern as illustrated in the graph of figure 5.18 (b). As we saw in section 2.5, *LCOM* is equal to the difference $p - q$, if $p > q$,

where q is the number of pairs of methods accessing one or more attributes in common and p is the number of pairs of methods not accessing any attribute in common. Since the difference $p - q$ tends to rise as the number of methods increases (at least in case of Java 1.3 API), *LCOM* values also increase steadily with the number of methods.

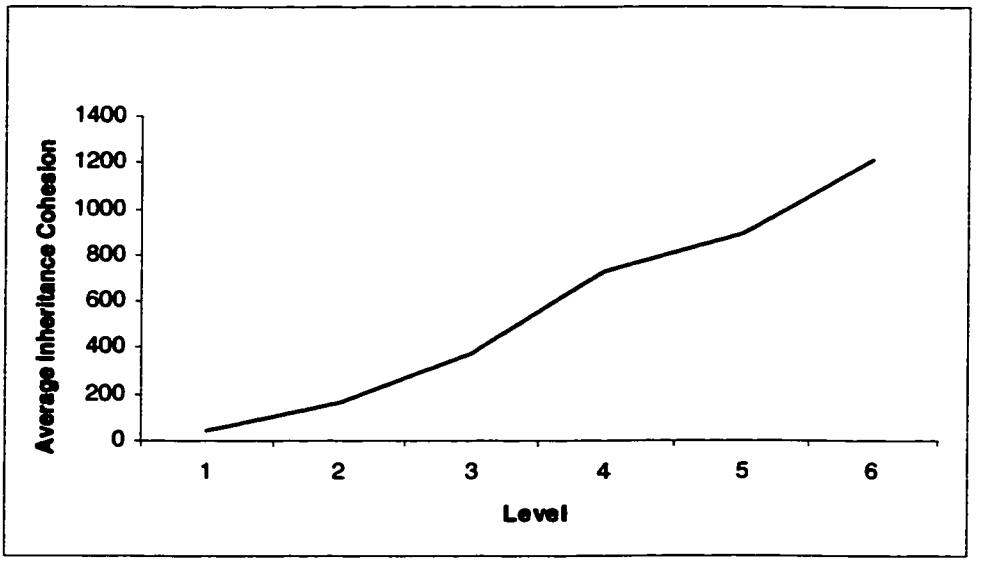


Figure 5.24: Average Inheritance Cohesion Versus Inheritance Level graph of *java.awt* hierarchy for *LCOM*.

5.2.7 Observation No. 7

The inheritance cohesion as measured by *LCOM* rises steadily as we move down the inheritance hierarchy as illustrated in the *Average Inheritance Cohesion Versus Inheritance Level* graph of figure 5.24. This steady rise in the *LCOM* value as the inheritance level increases is quite different from the behavior of *CCM* and *TCC* against the inheritance level, for which the *Average Inheritance Cohesion Versus Inheritance Level* graph shows a

random pattern as illustrated in the graph of figure 5.18 (a). Actually when we move down the inheritance tree the average number of methods of classes keeps increasing as the subclasses keep inheriting more and more methods from the classes of the higher levels. Since *LCOM* rises as the number of methods increases as we saw in section 5.2.5, *LCOM* also rises steadily as the inheritance level increases.

Metric	Class Cohesion	Inheritance Cohesion
CCM	98	124
RCI	209	0

Table 5.10: Comparison of the number of zero cohesion classes between *CCM* and *RCI*.

5.2.8 Observation No. 8

The number of classes with zero class cohesion is higher than the number of classes with zero inheritance cohesion for *RCI* (Table 5.10). In fact, *RCI* has no class with zero inheritance cohesion, whereas, for *CCM* the number of classes with zero class cohesion is lower than the number of those with zero inheritance cohesion (Table 5.10). Since *RCI* depends on the number of method-attribute interactions, it gives zero cohesion only if none of the methods interacts with any of the attributes. As we know that every class of Java has to inherit from the class *Object*, which already has some method-attribute interactions, the

inheritance cohesion (as measured by *RCI*) of these classes can never be zero as they all inherit the method-attribute interactions of the class *Object*.

On the other hand, in case of *CCM* the number of classes with zero inheritance cohesion is greater than the number of those with zero class cohesion due to the following reason: The classes that have only the constructor method have class cohesion (as measured by *CCM*) equal to one, since their connection graphs are completely connected. But when we also consider their inherited methods, the connection graphs of some of these classes get completely disconnected, since their constructor methods are not connected to any of their inherited methods, which too do not have any connections among themselves. Due to this reason, the inheritance cohesion of these classes becomes zero. Therefore, because of construction classes, the number of classes with zero inheritance cohesion is higher than those with zero class cohesion in case of *CCM*.

5.2.9 Observation No. 9

We found out the agreement among different cohesion metrics in finding the best and worst classes according to the inheritance cohesion. For this purpose, we found the best 5%, 10% and 15% of classes with respect to inheritance cohesion using different metrics and also the worst 5%, 10% and 15% classes according to inheritance cohesion using these metrics. We then found the percentage of agreement between these metrics on finding those classes. The results are as follows:

CCM and TCC

As we saw previously that the results of *CCM* and *TCC* are quite similar, we also found out that there is quite good agreement between these two metrics on finding the best and worst classes according to the inheritance cohesion. For example, 91% of the best 15% classes found by *CCM* and *TCC* are similar. Similarly, the percentage of agreement between these two metrics in finding the worst 10% classes is 95%. The details of comparison between *CCM* and *TCC* are given in table 5.11.

Percentage of Best/Worst Classes	Percentage of Agreement
Best 15% Classes	91.03%
Best 10% Classes	90%
Best 5% Classes	90%
Worst 15% Classes	85%
Worst 10% Classes	95%

Table 5.11: Percentage of agreement on finding the bets and worst classes between *CCM* and *TCC*.

CCM and RCI

Owing to the differences between the nature of *CCM* and *RCI*, as discussed previously, we don't find to much agreement between these two metrics in finding the best and worst classes according to the inheritance cohesion. For example, only 58% of the best 15% classes found by *CCM* and *RCI* are similar. Similarly, the percentage of agreement between

these two metrics in finding the worst 15% classes is 51%. The details of comparison between *CCM* and *RCI* are given in table 5.12.

Percentage of Best/Worst Classes	Percentage of Agreement
Best 15% Classes	58%
Best 10% Classes	50%
Best 5% Classes	30%
Worst 15% Classes	51%
Worst 10% Classes	40%

Table 5.12: Percentage of agreement on finding the bets and worst classes between *CCM* and *RCI*.

CCM and MCCM

Although *MCCM* is merely a modified version of *CCM*, but the inclusion of *Penalty Factor* in the definition of *MCCM* makes its results noticeably different from that of *CCM*. Due to this reason, we don't find extremely high percentage of agreement between these two metrics in finding the best and worst classes with respect to the inheritance cohesion. For example, 85% of the best 15% classes found by *CCM* and *MCCM* are similar and the percentage of agreement between these two metrics in finding the worst 15% classes is 80.7%. The details of comparison between *CCM* and *MCCM* are given in table 5.13. However, it can be noticed that the agreement between *CCM* and *MCCM* in finding the worst 10% classes is 99%, which is too high. This high percentage of agreement is due to the fact that nearly all of the worst 10% classes found by *CCM* and *MCCM* have 0

inheritance cohesion. If the *CCM* value for a class is 0, its *MCCM* value will also be 0, as *MCCM* is merely a product of *CCM* and the factor *1 – Penalty Factor.* Due to this, nearly all of the worst 10% classes found by *CCM* and *MCCM* are similar, as they all have 0 *CCM* value and thus, also have 0 *MCCM* value.

Percentage of Best/Worst Classes	Percentage of Agreement
Best 15% Classes	85%
Best 10% Classes	77%
Best 5% Classes	74%
Worst 15% Classes	80.7%
Worst 10% Classes	99%

Table 5.13: Percentage of agreement on finding the bets and worst classes between *CCM* and *MCCM*.

MCCM and TCC

Due to the differences between the definition of *MCCM* and *TCC*, especially the inclusion of *Penalty Factor* in the definition of *MCCM*, the percentage of agreement between these two metrics in finding the best and worst classes is not very high. For example, 80% of the best 15% classes found by *MCCM* and *TCC* are similar. Similarly, the percentage of agreement between these two metrics in finding the worst 15% classes is 71%. However, the agreement between *MCCM* and *TCC* in finding the worst 10% classes is 94%, which is quite high. The reason for this high percentage of agreement is similar to the reason of the

99% agreement between *CCM* and *MCCM* in finding the worst 10% classes, as explained above. The details of comparison between *MCCM* and *TCC* are given in table 5.14.

Percentage of Best/Worst Classes	Percentage of Agreement
Best 15% Classes	80%
Best 10% Classes	72%
Best 5% Classes	72%
Worst 15% Classes	71%
Worst 10% Classes	94%

Table 5.14: Percentage of agreement on finding the bets and worst classes between *MCCM* and *TCC*.

Percentage of Best/Worst Classes	Percentage of Agreement
Best 15% Classes	57.3%
Best 10% Classes	47%
Best 5% Classes	30%
Worst 15% Classes	58%
Worst 10% Classes	40%

Table 5.15: Percentage of agreement on finding the bets and worst classes between *TCC* and *RCI*.

TCC and RCI

Owing to the different nature of *TCC* and *RCI*, we don't find too much agreement between these two metrics in finding the best and worst classes according to the inheritance cohesion. For example, only 57.3% of the best 15% classes found by *TCC* and *RCI* are similar. Similarly, the percentage of agreement between these two metrics in finding the worst 15% classes is 58%. The details of comparison between *TCC* and *RCI* are given in table 5.15.

5.3 Conclusions Drawn from the Analysis of Java 1.3 API

We have drawn the following conclusions regarding the design quality of Java 1.3 API based on its analysis that we have done:

- As illustrated in table 5.1, the average *DIT* of all the 12 sub-hierarchies of the Java 1.3 API is around 3 with a small standard deviation. Therefore, it can be mentioned that as far as *DIT* is concerned, the quality of API is quite good. Nearly 67 % of all classes (i.e., 570 out of 861 classes) have *DIT* value of either 2 or 3. Small *DIT* value of most of the classes depicts good reusability of the API's classes, since classes with lower *DIT* are easier to reuse.
- Most of the leaf classes are concentrated at level 2 and 3 as illustrated in table 5.9. Nearly 64 % of all the leaf classes are at either level 2 or 3 of the hierarchy and thus have *DIT* value of 2 or 3. This small *DIT* is good from the point of view of reusability of these leaf classes. However, the inheritance cohesion of these classes is very low. This is due to the reason that many of the methods that they inherit

from the class *Object* or their direct parent (if they are at level 3) are unrelated to their core task. For example, consider the class *Arrays*, which is a direct descendent of the class *Object*. It inherits the method *toString* from *Object*, but the *toString* method doesn't make much sense for *Arrays*. Therefore, we can say that the objects of these leaf classes of the Java API are like inflated bodies with some methods useful to their task, but they also have some unrelated methods.

- We saw above that many of the leaf classes of the API have unrelated inherited methods. When user classes extend these API classes, the unrelated methods are also inherited into these user classes and most of the time the implementer of these classes is not aware of these unrelated inherited methods. The presence of unrelated inherited methods in user classes might make their maintenance and use difficult.
- There is too much abstraction and generalization in the non-leaf classes at the higher levels of the hierarchy. These classes usually have a large number of direct and indirect descendents. If a change is made in these classes, it propagates to all of their descendents. Therefore, to make any change in these classes, one has to keep in view all of their direct and indirect descendents. Due to this, their maintenance must be quite difficult. We found quite low inheritance cohesion values for these classes. These low inheritance cohesion values reflect the difficulty of maintenance of these classes.
- The leaf classes at level 4, 5 and 6 may also be difficult to maintain due to the following reasons:

1. There is quite big difference between their total number of methods (i.e., implemented plus inherited methods) and the number of methods implemented in them. Therefore, to get a complete picture of these classes, one has to consider all of their ancestor classes from the root class to their direct parent.
2. These classes inherit too many unnecessary and unrelated methods from their parents.

This conclusion is superimposed by the low inheritance cohesion values that we obtained for the classes at level 5 and 6.

- There is considerable proportion of abstract methods in the top-level classes of the hierarchy. Due to this considerable presence of abstract methods, these top-level classes have low cohesion. In addition to that, these abstract methods also make it difficult to reuse (through inheritance) these classes, since every abstract method has to be implemented in the subclass.
- As a bottom line, we would like to mention that although the Java API is good at providing a large variety of facilities for software development, but it may be difficult to maintain and modify.

Chapter 6

CONCLUSION

As mentioned in the literature, cohesion is an important quality factor of the object-oriented as well as imperative design [Biem95, Briand98, Chid94, Eder94]. In object-orientation it is a basic design requirement that a class should represent a single real world entity [Eder94]. Class cohesion is used as a tool to measure the extent to which a class meets this requirement. In this work, for the first time, we have proposed ways of using inheritance and class cohesion for measuring the quality of the inheritance hierarchy. The metrics that we have proposed provide designers of the object-oriented systems with the guidelines to enhance the quality of inheritance hierarchy to improve its maintainability, understandability and reusability. We have augmented our theoretical work by implementing the automated tool using which a designer can readily analyze his software against our proposed metrics.

6.1 Major Contributions

The major contributions made by this work can be summarized as follows:

- Cohesion related graphical metrics were proposed that assess the quality of the inheritance hierarchy of a system based on the class and inheritance cohesion of its classes. These metrics provide guidelines to the designer to make the design of inheritance hierarchy coherent to the object-oriented design principles. Using these metrics, the designer can assess to what extent the design of inheritance hierarchy satisfies the "*is a*" relationship.
- Three new cohesion metrics were proposed, i.e., *CCM*, *CCCM* and *MCCM*. We believe that *CCM* captures the connections among the methods of the class quite well and gives a good measure of class' cohesion. In the form of *MCCM*, we have proposed a metric that also takes into account the effect of overridden methods on inheritance cohesion of the class. Since overridden methods might result in the violation of the "*is a*" relationship as explained in section 3.3.2, we think that *MCCM* gives a finer measure of the class' inheritance cohesion. There isn't any other cohesion metric in literature that considers the effect of overridden methods.
- An automated tool was developed that automatically calculates our proposed metrics as well as the existing cohesion metrics discussed in chapter 2. The user only has to submit the root directory of the object-oriented system to be analyzed to the tool. The tool then parses the code of the system and generates the results for the supported metrics. The tool shows the results in the form of different tables.

Various graphs including the proposed graphical metrics can be generated. The tool has a simple GUI and is easy to use. Presently the tool only supports the Java language.

- Java 1.3 API was analyzed using the developed tool. Some very interesting quality features of the Java API were revealed as a result of the analysis.
- Results of different cohesion metrics presented in chapter 2 and 3 were compared and some major differences were identified among these metrics.

6.2 Future Directions

Following are the future directions that this work opens for the future research in this area:

- Empirical validation of the proposed graphical metrics and cohesion metrics can be done using the developed tool.
- Similar type of graphical metrics can be proposed to measure the quality of inheritance hierarchy based on inheritance coupling.
- Support for other object-oriented languages, such as, Small Talk and C++ can be added to the tool.
- Analysis of some of the good systems can be done based on the proposed metrics using the tool to identify the similar design patterns for providing the set of guidelines to design good inheritance hierarchies.

- A study of relationship between the inheritance coupling and inheritance cohesion can be done and some optimal range of inheritance coupling and cohesion values can be proposed.

BIBLIOGRAPHY

[Bala96] N. V. Balasubramanian. *Object-oriented Matrics*. Proceedings of International Conference on Software Quality, Maribor, Slovenia, 1995.

[Biem95] J. M. Bieman, B. K. Kang. *Cohesion and Reuse in an Object Oriented System.* Proceedings of ACM Symposium on Software Reusability (SSR'94), pp. 259-262, 1995.

[Basi96] V. R. Basili, L. C. Briand, W. L. Melo. *A Validation of Object-Oriented Design Metrics as Quality Indicators*. IEEE Transactions on Software Engineering, Vol. 22, No. 10, pp. 751-761, Oct. 1996.

[Basi88] V. R. Basili, H. D. Rombach. *The TAME Project: Towards Improvement-Oriented Software Environments*. IEEE Transactions on Software Engineering, Vol. 14, No. 6, pp. 758-773, June 1998.

[Briand96] L. C. Briand, S. Morasca, V. Basili. *Property-Based Software Engineering Measurement*. IEEE Transactions on Software Engineering, Vol. 22, No. 1, pp. 68-86, 1996.

[Briand99] L. C. Briand, S. Morasca, V. Basili. *Defining and Validating Measures for Object-Based High-Level Design*. IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 722-743, Oct. 1999.

[Briand98] L. C. Briand, J. Daly, J. Wuest. *A Unified Framework for Cohesion Measurement in Object-Oriented Systems*. Empirical Software Engineering; An International Journal, Vol. 3, No. 1, pp. 65-117, 1998.

[Bucc98] G. Bucci, F. Fioravanti, P. Nesi and S. Perlini, *"Metrics and Tool for System Assessment"*, Proceedings of 4th IEEE International Conference on Engineering of Complex Systems (ICECCS 98), pp. 36-46, 1998.

[Chae98] H. S. Chae, Y. R. Kwon. *A Cohesion Measure for Classes in Object-Oriented Systems*. IEEE, pp. 158-166, 1998.

[Chid91] Shyam Chidamber and Chris Kemerer. *Towards a Metrics Suite for Object Oriented Design*. A. Paepcke, ed., Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA 91, Oct. 1991.

[Chid94] Shyam Chidamber and Chris Kemerer, *"A Metrics Suite for Object Oriented Design"*, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, June 1994.

[Eder94] Johann Eder, Gerti Kappel and Michel Schrefl, *"Coupling and Cohesion in Object-Oriented Systems"*, Technical Report, Univ. of Klagenfurt, Austria, 1994.

[Elish99] M. Elish. *Measuring Inheritance Coupling in Object-Oriented Systems.* M.S. Thesis, Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, KSA, Dec. 1999.

[Embley88] D. W. Embley and S. N. Woodfield. *Cohesion and Coupling for Abstract Data Types.* International Conference on Software Engineering, pp. 144-153, IEEE Computer Society Press, 1988.

[Gagnon98] Etienne Gagnon. *Sablecc, an Object-Oriented Compiler Framework.* M.S. Thesis, School of Computer Science, McGill University, Montreal, March 1998.

[Hitz95_1] M. Hitz, B. Montazeri. *Measuring Product Attributes of Object-Oriented Systems.* Proceedings of 5th European Software Engineering Conference (ESEC 95), 1995.

[Hitz95_2] M. Hitz, B. Montazeri. *Measuring Coupling and Cohesion in Object-Oriented Systems*. Proceedings of International Symposium on Applied Corporate Computing, Montarrey, Mexico, October 1995.

[Hitz96] M. Hitz, B. Montazeri. *Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective*. IEEE Transactions on Software Engineering, Vol. 22, No. 4,pp. 276-270, 1996.

[Hend96] Brain Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall PTR, 1996.

[Kang96] B. K. Kang, J. Bieman. *Design-level Cohesion Measures: Derivation, Comparison, and Applications*. Computer Science Technical Report CS-96-104. Colorado State University, 1996.

[Li93] W. Li and S. Henry, *"Object-Oriented Metrics that Predict Maintainability"*, Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, 1993.

[Mose97] Simon Moser and Vojislav Misic. *Measuring Class Coupling and Cohesion: A Formal Metamodel Approach*. Proceedings of International Computer Science Conference (ICSC 97), pp. 31-40, 1997.

[Myers78] G. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, 1978.

[Nier89] O. M. Nierstrasz. *A Survey of Object-Oriented Concepts.* Object-Oriented Concepts, Databases and Applications, pp. 3-21, ACM Press and Addison-Wesley, 1989.

[Ott95] L. Ott, J. M. Bieman, B. K. Kang, B. Mehra. *Developing Measures of Class Cohesion for Object-Oriented Software.* In Proc. Annual Workshop on Software Metrics (AOWSM'95), June 1995.

[Sebe93] Robert Sebesta. *Concepts of Programming Languages.* Benjamin/Cummings, Second Edition, 1993.

[Stevens74] W. Stevens, G. Myers and L. Constantine. *Structured Design.* IBM Systems Journal, Vol. 13, pp. 115-139, 1974.

[Wegner87] P. Wegner. *Dimensions of Object-Based Language Design.* In Object-Oriented Programming Systems Languages an Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 22, pp. 168-182, Dec. 1987.