CHAPTER 3

PREDICTING FAULT USING JAVA REFLECTION WHICH MEASURES THE COUPLING AND COHESION METRICS OF OOP LANGUAGES

3.1 INTRODUCTION

Development of good quality software is the more essential concern of the most of the industries. High concern needs to be taken on software development process to retrieve the good quality software which can leads to better outcome without any errors. Increasingly, object-oriented measurements are being used to evaluate and predict the quality of software (Harrison R et al. 1998). A growing body of empirical results supports the theoretical validity of these metrics (Glasberg D et al. 2000). The validation of these metrics requires convincingly demonstrating that the metric measures what it purports to measure (for example, a coupling metric really measures coupling) and the metric is associated with an important external metric, such as reliability, maintainability and fault-proneness (El Emam K 2000). Often these metrics have been used as an early indicator of these externally visible attributes, because the externally visible attributes could not be measures until too late in the software development process.
3.2 BACKGROUND STUDY

When code is analyzed for object-oriented metrics, often two suites of metrics are used, the Chidamber-Kemerer (CK) (1994) and MOOD (Abreu, F. B. e. & Melo, W 1996) suites. In this section, we enumerate and explain the specific measures that can be computed using this tool.

3.2.1 Coupling

In 1974, Stevens et al. first defined coupling in the context of structured development as "the measure of the strength of association established by a connection from one module to another (Stevens, W et al. (1974) )." Coupling is a measure of interdependence of two objects. For example, objects A and B are coupled if a method of object A calls a method or accesses a variable in object B. Classes are coupled when methods declared in one class use methods or attributes of the other classes.

The Coupling Factor (CF) is evaluated as a fraction. The numerator represents the number of non-inheritance couplings. The denominator is the maximum number of couplings in a system. The maximum number of couplings includes both inheritance and non-inheritance related coupling. Inheritance-based couplings arise as derived classes (subclasses) inherit methods and attributes form its base class (superclass). The CF metric is included in the MOOD metric suite.

Empirical evidence supports the benefits of low coupling between objects (Coad, P). The main arguments are that the stronger the coupling between software artifacts, (i) the more difficult it is to understand individual artifacts, and hence to correctly maintain or enhance them; (ii) the larger the sensitivity of (unexpected) change and defect propagation effects across
artifacts; and (iii) consequently, the more testing required to achieve satisfactory reliability levels. Additionally, excessive coupling between objects is detrimental to modular design and prevents reuse. To summarize, low coupling is desirable.

3.2.2 Cohesion

Cohesion refers to how closely the operations in a class are related to each other. Cohesion of a class is the degree to which the local methods are related to the local instance variables in the class. The CK metrics suite examines the Lack of Cohesion (LOCOM), which is the number of disjoint/non-intersection sets of local methods (Fenton, N. E. & Pfleeger, S. L, 1998).

There are at least two different ways of measuring cohesion:

- Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.

- Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates the design implementation as well as reusability.
3.2.3 Encapsulation

Information hiding is a way of designing routines such that only a subset of the module’s properties, its public interface, is known to users of the module. Information hiding gives rise to encapsulation in object-oriented languages. "Encapsulation means that all that is seen of an object is its interface, namely the operations we can perform on the object (Jacobson, I 1992)." Information hiding is a theoretical technique that indisputably proven its value in practice. "Large programs that use information hiding has been found to be easier to modify -- by a factor of 4 -- than programs that don't." The following two encapsulation measures are contained in the MOOD metrics suite.

3.2.4 Attribute Hiding Factor (AHF)

The Attribute Hiding Factor measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible. An attribute is called visible if it can be accessed by another class or object. Attributes should be "hidden" within a class. They can be kept from being accessed by other objects by being declared a private.

The Attribute Hiding Factor is a fraction. The numerator is the sum of the invisibilities of all attributes defined in all classes. The denominator is the total number of attributes defined in the project. It is desirable for the Attribute Hiding Factor to have a large value.
3.2.5 Method Hiding Factor (MHF)

The Method Hiding Factor measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.

The Method Hiding Factor is a fraction where the numerator is the sum of the invisibilities of all methods defined in all classes. The denominator is the total number of methods defined in the project.

Methods should be encapsulated (hidden) within a class and not available for use to other objects. Method hiding increases reusability in other applications and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden. Thus hiding methods also reduces modifications to the code. The Method Hiding Factor should have a large value.

3.2.6 Inheritance

Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

3.2.7 Depth of Inheritance Tree (DIT)

The depth of a class within the inheritance hierarchy is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases
involving multiple inheritances, the DIT is the maximum length from the node to the root of the tree.

Well-structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior and, therefore, more fault-prone. Deeper trees require greater design complexity, since more methods and classes are involved. Indeed, deep hierarchies are also a conceptual integrity concern because it becomes difficult to determine which class to specialize. Additionally, interface changes within the tree must be reflected throughout the entire class tree and object instances. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods.

Applications can be considered to be "top heavy" if there are too many classes near the root, and indication that designers may not be taking advantage of reuse of methods through inheritance. Alternatively, applications can be considered to be "bottom heavy" whereby too many classes are near the bottom of the hierarchy, resulting in concerns related to design complexity and conceptual integrity.

3.2.8 Number of Children (NOC)

This metric is the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be difficult to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and therefore must be more flexible.
3.2.9 Weighted Methods/Class (WMC)

WCM measures the complexity of an individual class. A class with more member functions than its peers is considered to be more complex and therefore more error prone. The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This reasoning indicates that a smaller number of methods is good for usability and reusability.

However, more recently, software development trends support having more, smaller methods over fewer, larger methods for reduced complexity, increased readability, and improved understanding. This reasoning contradicts the prior reasoning of the last paragraph. The most recent recommendation (of more, smaller methods) is most likely more prudent. However, if a method is in a large inheritance tree having a large number of methods may not be advisable. Often, the WMC calculation considers complexity and the count of the number of methods applies a weighted complexity factor.

3.3 ADDITIONAL MEASURES

3.3.1 Number of Classes

If a comparison is made between projects with identical functionality, those projects with more classes are better abstracted.
3.3.2 Lines of Code

If a comparison is made between projects with identical functionality, those projects with fewer lines of code has superior design and requires less maintenance.

Additionally, methods of large size will always pose a higher risk in attributes such as Understandability, Reusability, and Maintainability.

3.4 OPEN ISSUES IN SOFTWARE FAULT PREDICTION

This section introduces the problems faced in software defect prediction

- **Relationship between Attributes and Fault:** There is no methodology to identify a generalized subset of attributes which act as a substantial factor for a module to be incorrect or non faulty.

- **No Standard Measures for Performance Assessment:** There is no standard criterion has been set for analyzing and comparing the defect prediction models.

- **Issues with Cross-Project Defect Prediction:** The fault prediction model is developed on one project and examined on some other.

- **No General Framework Available:** There has been no standard framework or procedure to apply a software defect prediction.

- **Economics of Software Defect Prediction:** cost evaluation framework to actually evaluate whether the fault prediction is useful or not.
➢ **Class Imbalance Problem:** The number of instances belonging to one class is much more than the number of instances belonging to another class.

### 3.5 JAVA REFLECTION BASED METRICS MEASUREMENT IN JAVA METHOD

In the proposed research work, java reflection tool is used for the measuring the OO metric values of cohesion and coupling in the efficient manner. Coupling and cohesion plays a major role in the java development scenario in which multiple methods and functions would be interlinked with each other. In the following subsection general overview of java reflection, coupling and cohesion is given which is followed by the actual measurement values of coupling and cohesion obtained by the java reflection tool.

#### 3.5.1 OVERVIEW OF JAVA REFLECTION

Java Reflection is quite powerful and can be very useful. Reflection is commonly used by programs which require the ability to examine or modify the runtime behaviour of applications running in the Java virtual machine. This is an advanced feature and should be used only by developers who have a strong knowledge of the fundamentals of the language. Java Reflection makes it possible to investigate classes, interfaces, fields and methods at runtime without knowing class name or package name. At compile time it is also possible to create a new object, invoke methods and get/set field values using reflection. We can access Java classes, methods, attributes, annotations at runtime. Reflection API is a powerful technique to find out program environment and investigate the class itself. Reflection API was included in Java version 1.1. The class of Reflection API is the part in the package of java.lang.reflect and the methods of Reflection API is the part in
the package of java.lang.class. It allows the user to get the complete information about classes, constructors, fields and various methods being used, and it also investigates the name of subclass and super class. Its main functionalities are as follows:

- Get class
- Get all fields and methods in the class includes private ones
- Invoke methods

By using the Java reflection we can retrieve all declared fields, methods and constructors in a class. It is also possible to identify the parent class and grandparent class. Number of methods that access the field can be found by investigating a signature of a field inside the methods and constructors. This will help to measure the metrics.

3.6 COHESION

The cohesion describes the relationship within a class. The cohesion of a single module/component is the degree to which its responsibilities form a meaningful unit, higher cohesion is better for better quality and reliability.

Types of cohesion

1. Coincidental Cohesion: Module elements are not related to the other.
2. Logical Cohesion: Components perform similar activities as selected from outside module.
3. Temporal Cohesion: operations connected only by general time performed.
4. Procedural Cohesion: Components involved in different but consecutive activities, each with different data. Communicational Cohesion: isolated operations except need same data or input.

5. Sequential Cohesion: operations on the same data in significant other; output from one function is input to the next.

6. Informational Cohesion: a module executes a number of activities, each with its own entry point, with independent code for each activity, all executed on the same data structure.

7. Functional Cohesion: all components contribute to a single.

3.6.1 Reason for High Cohesion

1. Easy to understanding modules.

2. Easy to maintain a system, changes in one module do not require changes in related modules.

3. Easy to reusing a module because most applications will need the random set of operations provided by a module.

3.6.2 Cohesion Metrics

Cohesion is the measure to which the methods in a class are related to another and work together to provide a set of actions. LCOM measures the degree of similarity of methods by fields in the class.

The Lack of Cohesion in Methods Calculation

LCOM 1: The original object oriented cohesion metric as given by Chidamber and Kemmerer represents an inverse measure for cohesion (Briand, S. Morasca and V. R. Basili. 1996) . Take each pair of methods in the class and determine the set of fields they each access. If they do not share the
same field, the count $R$ increases by one. If they share at least one field access, $S$ increases by one. Each pair of methods taken into account:

$$\text{RESULT} = (R > S) \ ? (R - S) : 0$$

A low value indicates the class method having a good relationship between other.

LCOM 2: This is an improved version of LCOM1 as is given by Chidamber and Kemmerer. Definitions used mno of methods in class, ano of fields in class, mAno of methods that access a field $A_i$, Sum ($mA$): sum of $mA$ over attributes of a class

$$\text{LCOM2} = 1 - \frac{\text{sum} (mA)}{(m*a)}$$

If the number of methods or fields in a class is zero, LCOM2 is undefined.

LCOM 3: This is another improvement on LCOM1 and LCOM2 as given by Li & Henry, 1993. It is defined as follows:

$$\text{LCOM3} = \frac{(m - \text{sum} (mA)) / a}{(m-1)}$$

The LCOM3 value between 0 and 1. LCOM3>1 indicates low cohesion in the class.

LCOM 4: This is another improvement on LCOM, LCOM2 and LCOM3 as given by Hitz&Montazeri. It is defined as follows:

$$\text{LCOM4} = \frac{(m - (\text{sum} (mA) - m) / ac)}{(m-1)}$$

LCOM4=1 indicates a cohesive class (good).

LCOM4>=2 bad & split method.

LCOM4=0 there are no methods in a class (bad).
LCOM5: This is another improvement on LCOM4 as is given by Henderson-Sellers, LCOM2, LCOM3 and LCOM4. It is defined as follows:

$$LCOM5 = \frac{((1/a) \times (\sum mA)) - m}{(1 - m)}$$

If the result is 0, it is considered as a perfect cohesion.

3.7 COUPLING

The coupling describes the relationship between the modules or classes, it indicates the strength of program units. (Guigui, Paul D. Scott. 2009) Highly coupled program units are depending on each other, loosely coupled units are independent or almost independent. Lower coupling is better for high quality software.

Types of coupling

1. Content/Pathological Coupling: A module uses/changes data in another
2. Control Coupling: Two modules communicating with a control flag (first tells second what to do via flag)
3. Common/Global-data Coupling: Two modules are interacting via global data
4. Stamp/Data-structure Coupling: Interacting via a data structure passed as a parameter. The data structure holds more info than the receiver needs.
5. Data Coupling: Interacting via parameter passing. The passed parameters are requested by the receiver.
7. Subclass Coupling: The relationship between a child and its parent. The child is associated with its parent, but the parent isn't associated with the child.
8. Temporal coupling: When two actions are communicating with each other and act as a module.

3.7.1 Reasons for Low Coupling

1. A change in one module usually services a ripple effect of changes in other modules.
2. Assembly of the modules might need more effort and/or time due to the increased inter-module dependency.
3. A specific module might be harder to reuse and/or test because dependent modules must be included.

3.7.2 Coupling Metrics

Coupling is defined as Interactions between classes by invoking methods and accessing variables (Sukainah Husein and Alan Oxley 2009). This coupling metric takes account of the degree of coupling, functional complexity and transitive. In this proposed metric there are two kinds of measures.

- WICoup (Weighted Intransitive Coupling)
- WTCoup (Weighted Transitive Coupling)

3.7.3 Weighted Direct Coupling

This measure only considers the direct coupling between the classes. The system consists of classes C = {C1, C2, C3…….Cm}. The class consist of a set of fields and methods. Let Mj be the set of methods of class Cj, Vj be the set of fields of class Cj and MVj,i be the set of fields and methods of
Ci invoked by Cj. MVj is the set of all methods and fields in other classes that is invoked by class Cj can be defined:

\[ MV_j = \bigcup MV_{j,i} \]

The result of MVj is more, if the class invoke more methods and fields outside the class. CoupD(i,j) is the measure of direct coupling between class Ci, Cj. is defined as

\[ \text{CoupD} (i,j) = \frac{|MV_{i,j}|}{|MV_i| + |V_i| + |M_i|} \]

Coupling measure for an entire system consists of no of classes.

\[ \sum_{i,j=1}^{n} \frac{\text{CoupD} (i,j)}{n^2 - n} \]

### 3.7.4 Weighted Transitive Coupling

Suppose that coupD(i,j) and coupD(j,k) is finite values but that coupD(i,k) is zero. There is no relation between Ci and Ck. Ci coupled with Cj and Cj coupled with Ck. There is indirect coupling between Ci and Ck due to the path p.

![Diagram](image)

**Figure 3.1 Example of indirect coupling.**

For example in fig 3 there is an indirect coupling between class I and K through the path P.
Here e_{i,k} denotes the number of classes between i and k. The largest path is taken into account and hence define Coup (i,k), the strength of coupling between the two classes, C_i, C_k to be

\[ \text{Coup}(i,k) = \text{CoupT}(i,k,P_{\text{max}}(i,j)) \]

Where \( P_{\text{max}}(i,j) = \arg \max_p \text{CoupT}(i,k,p) \) and \( \pi \) is the set of all paths from C_i to C_k. The weighted transitive coupling is defined as

\[ \text{WTCoup} = \sum_{i,j=1}^{n} \frac{\text{Coup}(1,k)}{n^2 - n} \]

For n is the number of classes in the system.

### 3.8 CASE STUDY: INFORMATION RETRIEVAL AND METRIC MEASUREMENT

All the structural cohesion and coupling measures were computed using Java reflection. Java reflection is used as a reverse engineering tool that contains the components for analyzing Java code and extracting the information. The structural information needed to compute cohesion and coupling has been also extracted using Java reflection. It extracted all types of fields, methods and constructors from classes in source.
In order to validate and demonstrate the metric calculation, a case study has been completed. From Figure-4 consider an example a system consist of classes, including fields, methods and constructors. Number of field
accessing methods can be found by using the signature of the field inside the method. classA and classD having indirect coupling between them.

The system consist of classes (A, B, C), methods (m1, m2,...m11) and fields (v1, v2,..., v14). In this example class A inherited by class B & C. class B is inherited by class. Cohesion measurement only concentrates on a single class, it does not consider inherited classes.

3.9 MEASURING COHESION AND COUPLING USING JAVA REFLECTION

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection. Java Reflection is quite powerful and can be very useful. For instance, when mapping objects to tables in a database at runtime, like Butterfly Persistence does. Or, when mapping the statements in a script language to method calls on real objects at runtime, like Butterfly Container does when parsing its configuration scripts.

Java Reflection is a process of examining or modifying the runtime behavior of a class at run time. The java.lang.Class class provides many methods that can be used to get metadata examine and change the run time behavior of a class. The java.lang and java.lang.reflect packages provide classes for java reflection.

Reflection is considered to be an advanced technique, and can be quite powerful when used correctly.

Reflection in Java consists of 2 primary things that you should remember:
1. Metadata. Metadata literally means data about the data. In this case, metadata means extra data that has to do with your Java program – like data about your Java classes, constructors, methods, fields, etc.

2. Functionality that allows you to manipulate the metadata as well. So, functionality that would allow you to manipulate those fields, methods, constructors, etc. You can actually call methods and constructors using Java reflection – which is an important fact to remember.

In the following sub section, measurement of coupling and cohesion values using the java reflection concept is explained

3.10 COHESION MEASUREMENT

Java reflection retrieves all the fields, methods and constructors. Each field’s signature inspected in the method definition. If a field is used inside a method, the value of a variable Aj incremented by one. These steps repeated for all fields and methods. Fig: 5 cohesion measurement program flows gives the clear idea about it.
Figure 3.3  Cohesion measurement program flow.
In our example case study consider class A

No of methods = 5.

No of fields = 6.

No of Methods that access attribute = 16.

LCOM1 = (R > S) ? (R - S) : 0 R = 2, S = 8. = 0.

LCOM2 = 1 - sum (mA) / (m*a) = 0.466

LCOM3 = (m - sum (mA) / a) / (m-1) = 0.583

LCOM4 = (m - (sum (meA) - m) / ac) / (m-1) = 0.583

LCOM5 = {((1/a) (Σu (A_j) )) - m} / (1 - m) = 0.1716

Table 3.1 Result of various cohesion measurement.

<table>
<thead>
<tr>
<th>Class</th>
<th>LCOM 1</th>
<th>LCOM 2</th>
<th>LCOM 3</th>
<th>LCOM 4</th>
<th>LCOM 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>0</td>
<td>0.466</td>
<td>0.583</td>
<td>0.583</td>
<td>0.171</td>
</tr>
<tr>
<td>Class B</td>
<td>0</td>
<td>0.416</td>
<td>0.625</td>
<td>0.625</td>
<td>0.5</td>
</tr>
<tr>
<td>Class C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Class D</td>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

From the Table-3.1 contain the result of various cohesion measurement for an example case study.

3.11 COUPLING MEASUREMENT

In our example case study The system consist of classes (A, B, C), methods (m1, m2,..,m11) and fields (v1, v2,.., v14). In this example class A inherited by class B & C. class B is inherited by class. Coupling is a
relationship between the classes. So we have to find parent class and grandparent for a class. java reflection help to find a parent class for a class.

Before measuring coupling we have to find fields, methods and constructors for each class. If the class is inherited, find parent and grant parent classes. Find the signature of base class field in derived class. If a variable found inside the method increment the value of MVji. That is the set of all methods and fields in other classes that is invoked by class Cj

Consider the case study example, Class A inherited by Class B. Both classes having direct coupling. Method m6 access the field v1 & v2. Method m7 access the field v6 and method m6 access m4. Coupling between classes B & A is MVji =3. The value should be larger, if the class invokes more methods and fields

\[\text{CoupD}(i, j) = \frac{|\text{MV}_{ij}|}{|\text{MV}_{i}| + |V_{i}| + |M_{i}|}\]

\[\text{CoupD}(i, j) = \frac{|8|}{|11| + |4| + |8|} = 0.167\]

\[\text{WICoupD} = \sum_{i,j=1}^{n} \frac{\text{CoupD}(i, j)}{n^2 - n} = 0.014\]

There is an indirect coupling between class A & D.

\[\text{CoupT}(i, k) = 0.0714\]

\[\text{WTCoup} = 0.006\]

The results in this study using structural information to measure cohesion and coupling. They may help to find the quality of software and predict the fault.
3.12 SUMMARY

This chapter focuses on object oriented quality metric parameters. To extract this information for cohesion and coupling measurement, Java reflection can be used in a manner similar to measuring these metrics. Java reflection defines the novel structural information retrieval method for coupling and cohesion measurement. The low coupling and high cohesion improve the product quality and reliability. The outcome result helps to predict the fault of an object oriented system after the coding phase is completed.