Class Break Point Determination Using CK Metrics Thresholds

GJCST Classification D.2.8

Dr. E. Chandra [1], P. Edith Linda [2]

***Abstract*-The design and development of any project has got a well-defined project development cycle. But once the project or the product has been developed, it is subject to change due a lot a policy changes on the part of the organization or the government. These changes are implemented on the code but most of the time these changes are not reflected on the design document. This leads to inconsistencies in terms of design and code thereby causing depreciation in terms of quality. In this work we propose to use the object oriented metrics which uses the parameters mentioned in the CK metrics suite to assess the quality of the software at the class level. The proposed tool namely "Class Break Point" which could be used to determine the class design validity. This tool can be used to check if the class is adhering to the OO design specifications. The tool is useful in predicting the decomposition point of the class.**

Keywords- OO metrics, Weighted Method Per Class, Depth of Inheritance Tree, LCOM, Response for a class, Number of Children, design refinement.

I. Introduction

The quality of software can be evaluated using metrics.Software quality has been a major challenge in various software projects. Quality has been a major issue in software development but it lacks in standards measuring quality. Metrics are the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information together with the use of techniques to improve the process and its products [2]. The use of Metrics can help us ***understand*** more about our software products, processes and services. Metrics can be used to ***evaluate*** our software products, processes and services against established standards and goals. Metrics can provide the information we need to ***control*** resources and processes used to produce our software. Metrics can be used to ***predict*** attributes of software entities in the future [4].A metrics program that is based on the goals of the organizations will help communicate those goals. People will work to accomplish what they believe to be important. Well-designed metrics with documented objectives can help our organization obtain the information it needs to continue to improve its software products, processes and services while maintaining a focus on what is important to that organization.

About[1]-Director, Department of MCA, D J. Academy of Managerial Excellence, Coimbatore -32, India. Telephone:+919894255832. Email: crcspeech@gmail.com

About[2]-Assistant Professor, School of IT and Science, Dr. G. R Damodaran College of Science, Coimbatore-14, India. Telephone:+919894876043. Email:p.lindavinod@gmail.com

II. Introduction to the tool

When Object Oriented design was introduced, it guaranteed robust, maintainable and reusable systems. But these claims given by the object oriented designs are not fulfilled in large. Simply by using object oriented language or design does not assure a robust and reusable system. It mainly depends on the pattern or interdependencies between the subsystems and the communication between them. So the design of the object oriented system has to be validated to develop high quality applications which could be easily maintained and can be reused. The Design quality metrics is used to check the conformance of the object oriented design in the particular application. These design quality metrics provide information to the designers regarding the ability to survive to the changes [7].

The Code metrics is a set of software measures that provide developers better insight into the code they are developing. By taking advantage of code metrics, developers can understand which types and/or methods should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development [6].

Here in this paper we propose a tool called Class Break Point" to evaluate the code quality of the software. Most of the works mentioned in the literature are covering the metrics that can be computed at the design level. But here in this paper we demonstrate the same set of metrics to evaluate the code quality. This tool could be used in the maintenance phase or even the developers could use this tool to find the adherence of code to the object oriented designing principles. Hence here the input to the tool would be the source code of existing software. Another type of input to the tool can be the code developed by the programmer to check the design quality of the code. Here in the second case the tool can be used as a self-evaluation tool to the programmer. In the first case the tool can be used to evaluate the breakthrough point of the class. In continuation with our previous work as mentioned in [1], this tool in the initial phase extracts only the parameters of the CK metrics tool.The main objective of the class break point tool is to check the quality of the source code and the adherence of the source code to the design documents. This tool finds its usage in the maintenance phase of the software. Here in this

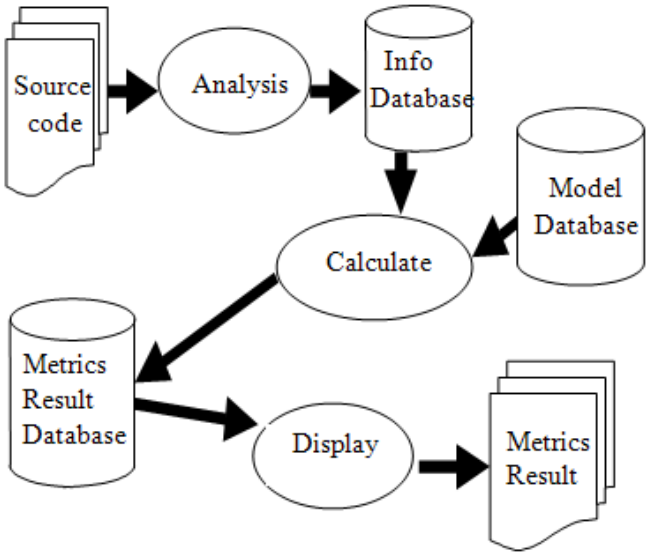

Fig1: Architecture of the Class Break Point Tool

phase after a period of year or two the software would have undergone lots of changes in code due to changing requirements. The changing requirements may be due to the change in the government policies or due the change in the company's policies. Hence the code is changed to meet the new set of requirements. When the code is changed, the developers usually change the code making the software ready to meet the new set of requirements. But most of the time this change is not reflected in the design document. Due to this factor it makes the next developer in jeopardy as the developer is not aware if the design document is up to date with the latest changes made.

1) *Explanation of the parameters to be extracted by the code analyzer tool:*

Parameters Identified to check the design validity of the class are same as the CK metrics suite[9, 10]**.**

S.No	PARAMETERS OF CK METRIC	THRESHOLD
1	WMC	**0-15**
2	DIT	**0-6**
3	NOC	**0-6**
4	CBO	**0-8**
5	RFC	**0-35**
6	**LCOM**	**0-1**

Table 1: The threshold values for the CK metric suite parameters.

Metric 1: Weighted Methods per class (WMC):

The Weighted Method Per Class (WMC) – It is a count of sum of complexities of all methods in a class. To calculate the complexity of a class, the specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on value 1.0. Consider a class *K1*, with methods *M1,…….. Mn* that are defined in the class.Let *C1,……..Cn* be the complexity of the methods[4].

$$WMC = \sum ci..cn$$

For the sake of simplicity we assume that the complexity of all the class is the same. Hence WMC is the sum of all the methods in the class. To compute WMC use the method getDeclaredMethods() to compute the number of methods in the class If the number of methods in the class is high then the class is considered to be very complex. If the number of methods in the class is low the complexity of the class is less. If the number of methods in a class is less than or equal to 15, then the class can be considered to have normal complexity. The threshold limit is set to 15 per class.

```
public class MetricsWMC {
public int getMethods(String className) {
    try {
          Class classObj = Class.forName(className);
          return classObj.getDeclaredMethods().length;
        }
catch (Exception e) {
                e.printStackTrace();
          }
    return 0;}}
```

Metric2:

Depth of Inheritance Tree (DIT)

Assess how deep, a class is in hierarchy structure. This metric assesses the potential reuse of a class and its probable ease of maintenance. A class with a small DIT has much potential for reuse it tends to be a general abstract class. On the other side, as a class gets deeper into a class hierarchy, it becomes more difficult to maintain due to the increased mental burden needed to capture it functionally. There are certain viewpoints regarding DIT The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex" hence higher the value of DIT it is bad. Another viewpoint is Its useful to have a measure of how deep a particular class is in the hierarchy so that the class can be designed with reuse of inherited methods" " hence higher the value of DIT it is good.

DIT = maximum inheritance path from the class to the root class.

In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree. The theoretical Basis of the DIT metric is a measure of how many ancestor classes can potentially affect this class. The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deeper the tree the greater is the design complexity. Inheritance has to manage the complexity and to increase the reusability of the class and not to create issues with the design. A high DIT has been found to increase faults. Since the fault-prone classes may be at the middle of the tree, it

may contribute to the fault to the rest of the inheriting classes. This work suggests that lower DIT has great potential of reuse; hence a threshold value of 6 levels is set for DIT. To compute the value of DIT getSuperclass() is used to find the super class of an existing class. getName() is used to return the name of the super class. These two methods can be used in combination to get the list level of the inheritance of the particular class.

```
public int getDITValue(String className)
 {
int ditValue = 1;
try {
Class classObj = Class.forName(className);
String superClass = classObj.getSuperclass().getName();
while( ! "java.lang.Object".equals(superClass))
    {
     classObj = Class.forName(superClass);
superClass = classObj.getSuperclass().getName();
     System.out.println(superClass);
    ditValue++;
    }
    } catch (Exception e)
            { e.printStackTrace(); }
   return ditValue;
 }
```

Metric 3: Number of Children (NOC):

It is a simple measure of the number of classes associated with a given class using an inheritance relationship. It could be used to assess the potential influence that a class has on the overall design. NOC measures how many classes inherit directly methods or fields from a super-class. The greater the number of children in the inheritance hierarchy the greater the reuse. Then again a large number of children of a class might indicate improper abstraction for a parent class. High DIT value and low NOC means better reusability but the issue of maintainability is at stake It also has a negative impact on understandability and is more difficult to modify. Since there are no empirical or theoretical boundary values, then we should find the proper threshold value for the system under development. Here in the system we have set the value of NOC to be 6 as same as DIT.

NOC = number of immediate sub-classes of a class

To determine the value of NOC the reflection classes, such as Method, are found in java.lang.reflect. There are three steps that must be followed to use these classes. The first step is to obtain a java.lang.Class object for the class that you want to manipulate. java.lang.Class is used to represent classes and interfaces in a running Java program. To obtain the class object use Class c =Class.forName("java.lang.String"): Then call a method such as getDeclaredMethods, to get a list of all the methods declared by the class. Using both the information on hand tackle the specific application using reflection

```
public int getDITValue(String className)
{
int ditValue = 0;
try{
String packageName[] = {"com.test.MWCTest2","com.test.MWCTest3","com.test.MWCTest4"};
Class classObj = Class.forName(className);
for (String string : packageName)
{
        Object obj = Class.forName(string).newInstance();
        if(obj instanceof MWCTest)
{
        ditValue++;
        }}
        } catch (Exception e)
        {
          e.printStackTrace();
          }
          return ditValue;
}
```

Metric 4: Lack of Cohesion in Methods (LCOM)

It is the difference between the number of methods whose similarity is zero and not zero. The similarity of two methods is the numbers of attributes used were common. LCOM can judge the cohesiveness among the class methods. Low LCOM indicates high cohesiveness and vice versa. High LCOM indicates that a class shall be considered for splitting into two or more classes. However, a LCOM measure of zero is not strong evidence that a class enjoys cohesiveness.The single responsibility principle states that a class should not have more than one reason to change. Such a class is said to be cohesive. A high LCOM value generally pinpoints a poorly cohesive class. There are several LCOM metrics. The LCOM takes its values in the range 0 to 1. The computation of LCOM is as follows:

$$LCOM = 1 - \sum MF / (M * F)$$

Where:

- M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods).
- F is the number of instance fields in the class.
- MF is the number of methods of the class accessing a particular instance field.
- Sum(MF) is the sum of MF over all instance fields of the class.

To find the value for LCOM find the number of methods in a class(M) using the method getDeclaredMethods(). Find the number of classes in the package level using the getclass(). Then find the number of instance fields in the class(F) using the InstanceFieldAccess class and its associated objects. The InstanceFieldAccess class defines an instance field s. The main method creates an object, sets the instance field, and then calls the native method InstanceFieldAccess.accessField. This native method prints out the existing value of the instance field and then sets the field to a new value. To know more about the class objects emphasis has to be put to the instanceof operator. Class.isInstance method can be used to simulate the

instanceof operator. field.get(objectInstance) where can be used to find the number of methods of the class accessing a particular instance field. After determining the value of the above mentioned parameter the formula mentioned above could be used for the computation of LCOM.At this juncture there are certain issues to be addressed with the LCOM metric. The reusability parameter is to be negatively influenced by LCOM. Higher the values for LCOM lower the scope for reusability. Also the maintainability for a class containing higher LCOM values is higher as it directly affects other classes also. Higher cohesion also decreases the changeability, stability and the portability of the classes as it triggers changes in the other classes with are closely coupled. Hence a good solution to this issue could be to keep the value of LCOM to a minimum value.

The Lack of Cohesion in Methods metric can be computed using the following three formats:

LCOM1: Take each pair of methods in the class and determine the set of fields they each access. If they have disjointed sets of field accesses, the count P increases by one. If they share at least one field access, Q increases by one. After considering each pair of methods:

RESULT = (P > Q) ? (P - Q) : 0

A low value indicates high coupling between methods. This also indicates potentially high reusability and good class design. Chidamber and Kemerer provided the definition of this metric in 1993.

LCOM2: This is an improved version of LCOM1. Say you define the following items in a class:

m: number of methods in a class

a: number of attributes in a class.

mA: number of methods that access the attribute a.

sum(mA): sum of all mA over all the attributes in the class.

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

If the number of methods or variables in a class is zero (0), LCOM2 is undefined as displayed as zero (0).

LCOM3: This is another improvement on LCOM1 and LCOM2 and is proposed by Henderson-Sellers. It is defined as follows:

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

where m, a, mA, sum(mA) are as defined in LCOM2.

The LCOM3 value varies between 0 and 2. LCOM3>1 indicates lack of cohesion and is considered a kind of alarm. If there is only one method in a class, LCOM 3 is undefined and also if there are no attributes in a class LCOM3 is also undefined and displayed as zero (0). Each of these different measures of LCOM has a unique way to calculate the value of LCOM. An extreme lack of cohesion such as LCOM3>1 indicates that the particular class should be split into two or more classes. If all the member attributes of a class are only accessed outside of the class and never accessed within the class, LCOM3 will show a high-value. A slightly high value of LCOM means that you can improve the design by either splitting the classes or re-arranging certain methods within a set of classes [3].

Metric 5: Coupling between objects (CBO).

When one object interacts with another object that is a coupling. Strong coupling means that one object is strongly coupled with the implementation details of another object. Strong coupling is discouraged because it results in less flexible, less scalable application. However, coupling can be used so that it enables objects to talk to each other while also preserving the scalability and flexibility. OO metrics can help you to measure the right level of coupling. CBO is defined as the number of non-inherited classes associated with the target class. It is counted as the number of types that are used in attributes, parameters, return types, throws clauses, etc. Primitive types and system types (e.g. java.lang.*) are not counted. Method Invocation Coupling (MIC)is defined as the relative number of classes that receive messages from a particular class.

MIC = nMIC / (N -1)

Where N = total number of classes defined within the project.

nMIC = total number of classes that receive a message from the target class

to find the value of N(total number of classes in the system) use the following the methods. The fully qualified class name (including package name) is obtained using the getName() method . The class name without the pacakge name can be obtained using the getSimpleName() method. You can access the modifiers of a class via the Class object. The class modifiers are the keywords "public", "private", "static" etc The modifiers are packed into an int where each modifier is a flag bit that is either set or cleared. You can check the modifiers using these methods in the class java.lang.reflect.Modifiers. The method used to get the modifier is getModifiers().The Method.invoke(Object target, Object ... parameters) method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the method you are invoking. Using the above methods to extract the values and apply in the formulae the value of CBO can be obtained.

Metric 6: Response for Class(RFC)

It is defined as a count of the set of methods that can be potentially executed in response to a message received by an instance of the class. Response set of an class ={ set of all methods that can be invoked in response to a message to the object }

RFC = |RS| where RS is the response set for the class

The RFC is defined as the total number of methods that can be executed in response to a message to a class. This count includes all the methods available in the whole class hierarchy. If a class is capable of producing a vast number of outcomes in response to a message, it makes testing more difficult for all the possible outcomes. Response For a Class (RFC) is the sum of the number of its methods and the total of all other methods that they directly invoke. If the number of methods invoked in response to a message received by an object is large, the maintenance and testing are more demanding. Large number of method invocation means more testing and debugging. Larger the method invocation greater is the complexity. The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set.

RFC = M + R (First-step measure)
RFC‘ = M + R‘ (Full measure) M = number of methods in the class
R = number of remote methods directly called by methods of the class
R‘ = number of remote methods called, recursively through the entire call tree
A given method is counted only once in R (and R‘) even if it is executed by several methods M. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated. It counts only the first level of calls outside of the class. RFC‘ measures the full response set, including methods called by the callers, recursively, until no new remote methods can be found. If the called method is polymorphic, all the possible remote methods executed are included in R and R‘. The use of RFC‘ should be preferred over RFC. RFC was originally defined as a first-level metric because it was not practical to consider the full call tree in manual calculation. With an automated code analysis tool, getting RFC‘ values is not longer problematic. As RFC‘ considers the entire call tree and not just one first level of it, it provides a more thorough measurement of the code executed.

```
private void incRFC(String className, String methodName,
Type[] arguments) {
String argumentList = Arrays.asList(arguments).toString();
String args = argumentList.substring(1,
argumentList.length() - 1);
String signature = className + "." + methodName + "(" +
args + ")";
responseSet.add(signature);
```

III. Results and Discussions

As per the description given in the previous sections the parameters identified are: WMC, DIT, NOC, LCOM, CBO, RFC. Each of these parameters are extracted from the source code. A threshold is set for each of the parameters and the conformance of the extracted values are expected with the set threshold values i.e the parameters values are expected to lie within the range of the thresholds. If the value lies within the range then the class satisfies the OO design paradigms. If the parameters lie outside the threshold then design refinement suggestion has to be provided to the developer or user. The results for the first three parameters in three projects are depicted below in the table:

Class Name	WMC	DIT	NOC
com.test.MWCTest2	6	1	2
com.test.MWCTest3	5	3	2
com.test.MWCTest	8	2	7
com.test.MWCTest4	5	4	2

Table2. The extracted parameter values for the first three parameters of the CK metrics suite.

Hence in the above classes the extracted parameter values are within the threshold so no design refinements regarding the reusability are suggested. But in classes where the extracted value does not fall into the threshold then decision has to be made to split the class to meet the OO design specifications.

IV. References

1) Dr. E. Chandra and P. Edith Linda, Assessment of of Software Quality through Object Oriented Metrics, CIIT International Journal of Software Engineering, Vol2, Iss:2 Feb 2010
2) Goodman, Paul. 1993. Practical Implementation of Software Metrics. London: McGraw Hill.
3) Heung Seok Chae,Yong Rae Kwon, andDoo Hwan Bae, Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 11, NOVEMBER 2004
4) Humphrey, Watts S. 1989. *Managing the Software Process.* Reading: Addison-Wesley.
5) http://javaboutique.internet.com/tutorials/coupcoh/index-2.html
6) http://msdn.microsoft.com/enus/library/bb385914.aspx
7) http://www.objectmentor.com/resources/articles/oodmetrc.pdf
8) Ramanath Subramanyam and M.S. Krishnan, Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Transactions on Software Engineering, Vol. 29, No. 4, April 2003
9) Shyam R. Chidamber and Chris F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering. Vol. 20, No. 6, June 1994
10) Yuming Zhou and Hareton Leung, Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults, IEEE Transactions On Software Engineering, Vol. 32, No. 10, October 2006