Trends in Engineering and Technology (NCTET-2K17)
International Journal of Advanced Engineering Research and Science (IJAERS) Special Issue-3
https://dx.doi.org/10.22161/ijaers/nctet.2017.54 ISSN: 2349-6495(P) | 2456-1908(O)

Empirical Study of coupling and cohesion in an Object-Oriented system Metrics

P. Neelima
Assistant Professor,
Dept of CSE,
SPMVV College of Engineering
Tirupati
neelima.pannem@gmail.com

B.Sangamithra
Assistant Professor
Dept of CSE,
Sree Vidyanikethan Engineering College
Tirupati
Mithra197@gmail.com

Dr. M.Sunil Kumar
Professor & HOD
Dept of CSE,
Sree Vidyanikethan Engineering College
Tirupati
Sunilmalchi1@gmail.com

***Abstract*— In developing a good software product, including documentation, design, program, test, and maintenance can be measured statistically. Therefore the quality of software can be monitored efficiently. Software metrics is very important in research of software engineering and it has improved gradually. In this paper, software metrics definition were given and the history of and the types of software metrics were overviewed. Software complexity measuring is the important constituent of software metrics and it is concerning the cost of software development and maintenance. In order to improve the software quality and the project controllability, it is necessary to control the software complexity by measuring the related aspects. This paper respectively expounds MOOD metrics and C&K metric method for examples of complexity metrics.**

***Index terms*- metrics suite, complexity, Object Oriented ,quality measurement.**

1. INTRODUCTION

The term ***software engineering*** is combination of two words, software and engineering. **Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product. Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define ***software engineering*** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Define Software engineering:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

NEED OF SOFTWARE ENGINEERING:

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

Large software **-** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

Scalability**-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

Dynamic Nature**-** The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

Quality Management**-** Better process of software development provides better and quality software product.

REQUIREMENTS ANALYSIS AND SPECIFICATION

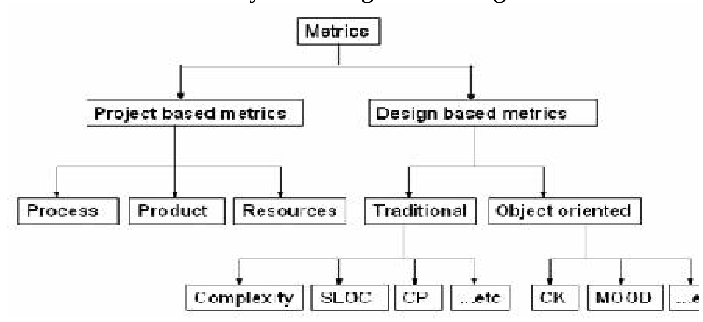
Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as ***system analysts***. The analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system, then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?

- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

Metrics

We categorized metrics into two groups: project based metrics and design based metrics. Project based metrics contain process, product and resources. Design based metrics contain traditional metrics and object oriented metrics. In traditional metrics, will discuss complexity metrics, SLOC (Source lines of code), and CP (Comment percentage). The following figure shows metrics hierarchy according to our categorization



Software metrics:

The first definition of *software metrics* is proposed by Norman Fenton , software metrics is a collective term used to describe the very wide range of activities concerned with measurement in software engineering. These activities range from producing numbers that characterize properties of software code through to models that help predict software resource requirement and software quality.

Metrics are a means for attaining more accurate estimations of project milestones, and developing a software system that contains minimal faults . Project based metrics keep track of project maintenance, budgeting etc. Design based metrics describe the complexity, size and robustness of object oriented and keep track of design performance

OBJECT ORIENTED MEASUREMENT:

On building a program code using object oriented paradigm can be measure by using metrics. The attributes (objects and classes) Which have been mention can play a major role in calculating metrics

1.Number of scenario scripts (NSS) : It's the number of scenario scripts counted in the use cases. This measure is correlated with application size and the number of tests. NSS mainly allow to predict development and testing efforts.
2.Number of key classes : This measure evaluate the high-design effort.
3.Number of support classes : This measures evaluates the low-level design. Average number of support classes per key.
4.Number of subsystems : This one provide more information on the system's structure.
5.Number of operations overridden by a class : Allow to evaluate inheritance effects.
6.Number of operations added by a subclass : Measures also the inheritance effects.

Efficiency - Are the constructs efficiently designed?
The amount of computing resource and code required by a program to perform its function.

Complexity - Could the constructs be used more effectively to decrease the architectural complexity?

Understandability- Does the design increase the psychological complexity?

Reusability - Does the design quality support possible reuse? Extent to which a program or part of a program can be reused in other application, related to the packaging and scope of the functions that the program performs.

Testability/Maintainability - Does the structure support ease of testing and changes?

Effort required locating and fixing an error in a program, as well as effort required to test a program to ensure that it performs its intended function.

Encapsulation is the packaging (or binding together) of a collection of items:

- Low-level examples of encapsulation include records and arrays.
- Subprograms (e.g., procedures, functions, subroutines, and paragraphs) are mid-level mechanisms for encapsulation.
- In object-oriented (and object-based) programming languages, there are still larger encapsulating mechanisms, e.g., C++'s classes, Ada's packages, and Modula 3's modules.

Objects encapsulate:

knowledge of state, whether statically maintained, calculated upon demand, or otherwise, advertised capabilities (sometimes called operations, method interfaces, method selectors, or method interfaces), and the corresponding algorithms used to accomplish these capabilities (often referred to simply as methods), In many object-oriented programming languages, encapsulation of objects (e.g., classes and their instances) is syntactically and semantically supported by the language. In others, the concept of encapsulation is supported conceptually, but not physically.

Information hiding is the suppression (or hiding) of details.

- The general idea is that we show only that information which is necessary to accomplish our immediate goals.
- There are degrees of information hiding, ranging from partially restricted visibility to total invisibility.
- Encapsulation and information hiding are not the same thing, e.g., an item can be encapsulated but may still be totally visible.

Information hiding plays a direct role in such metrics as object coupling and the degree of information hiding

Inheritance is a mechanism whereby one object acquires characteristics from one, or more, other objects.

- Some object oriented languages support only single inheritance, i.e., an object may acquire characteristics directly from only one other object.
- Some object-oriented languages support multiple inheritance, i.e. an object may acquire characteristics directly from two, or more, different objects.
- The types of characteristics which may be inherited, and the specific semantics of inheritance vary from language to language.

Many object-oriented software engineering metrics are based on inheritance, e.g.:

- number of children (number of immediate specializations),
- number of parents (number of immediate generalizations), and
- class hierarchy nesting level (depth of a class in an inheritance hierarchy).

Abstraction is a mechanism for focusing on the important (or essential) details of a concept or item, while ignoring the inessential details.

- Abstraction is a relative concept. As we move to higher levels of abstraction we ignore more and more details, i.e., we provide a more general view of a concept or item. As we move to lower levels of abstraction, we introduce more details, i.e., we provide a more specific view of a concept or item.
- There are different types of abstraction, e.g., functional, data, process, and object abstraction.
- In object abstraction, we treat objects as high-level entities (i.e., as black boxes).

There are three commonly used (and different) views on the definition for "class,":

- A class is a pattern, template, or a blueprint for a category of structurally identical items. The items created using the class are called instances. This is often referred to as the "class as a `cookie cutter'" view.
- A class is a thing that consists of both a pattern and a mechanism for creating items based on that pattern. This is the "class as an `instance factory'" view. Instances are the individual items that are "manufactured" (created) by using the class's creation mechanism.
- A class is the set of all items created using a specific pattern, i.e., the class is the set of all instances of that pattern.

A **metaclass** is a class whose instances are themselves classes. Some object-oriented programming languages directly support user-defined metaclasses. In effect, metaclasses may be viewed as classes for classes, i.e., to create an instance, we supply some specific parameters to the metaclass, and these are used to create a class. A metaclass is an abstraction of its instances.

A **parameterized class** is a class some or all of whose elements may be parameterized. New (directly usable) classes may be generated by instantiating a parameterized class with its required parameters. Templates in C++ and generic classes in Eiffel are examples of parameterized classes. Some people differentiate metaclasses and parameterized classes by noting that metaclasses (usually) have run-time behavior, whereas parameterized classes (usually) do not have run-time behavior.

Several object-oriented software engineering metrics are related to the class-instance relationship, e.g.:

- number of instances per class per application,
- number or parameterized classes per application, and
- ratio of parameterized classes to non-parameterized classes.

C & K Metric suite:

Shyam Chidamer and Chris Kemerer have developed a small metrics suite for object-oriented designs. The six metrics they have identified are:

- weighted methods per class: This focuses on the complexity and number of methods within a class.
- depth of inheritance tree: This is a measure of how many layers of inheritance make up a given class hierarchy.
- number of children: This is the number of immediate specializations for a given class.
- coupling between object classes: This is a count of the number of other classes to which a given class is coupled.

- response for a class: This is the size of the set of methods that can potentially be executed in response to a message received by an object.
- lack of cohesion in methods: This is a measure of the number of different methods within a class that reference a given instance variable.

Chidamber and Kemerer Metrics evaluation:

Identification of classes(and objects): In this step key abstractions in the problem space are identified and labeled as potential classes and objects.

2) Identify the semantics of classes(and objects):In this step, the meaning of the classes and objects identified in the previous step is established, this includes definition of the life cycles of each object from creation to destruction.

3) Identify relationships between classes(and objects):In this step, classes and objects interactions, such as patterns of inheritance among classes and patterns of visibility among objects and classes are identified.

4) Implementation of classes(and objects):In this step, detailed internal views are constructed, including definitions of methods and their various behaviors.

According to ontology, objects are defined independent of implementation considerations and encompass the notions of encapsulation, independence and inheritance. On the basis of this study the world is viewed as composed of things refer to as substantial individuals, and concepts.

$X=\{x,p(x)\}$where x is the substantial individual and

p(x) is the finite collection of its properties

In object oriented terminology, the instance variables together with its methods are the properties of the object. Intuitively, coupling refers to the degree of interdependence between parts of a design, while cohesion refers to the internal consistency within parts of the design.

Coupling: Two objects are coupled if and only if atleast one of them acts upon the other, X is said to act upon Y if the history of Y is affected by X where its stated thing traverses in time

Lets $X=\{x,p(x)\}$ and $Y=\{y,p(y)\}$ be two objects

$p(x)=\{M_X\}U\{I_X\}$

$p(y)=\{M_Y\}U\{I_Y\}$

where $\{M_i\}$ is the set of methods and$\{I_i\}$ is the set of instance variables of object i.The objects both X and Y are interdependent to each other .

Cohesion: Cohesion is defined as similarity σ( ) of two things to be the intersection of the sets of properties of the two things

$\sigma(X,Y)=p(x)\cap p(y)$

$\sigma(M_1,M_2)=\{I_i\}\cap\{I_2\}$

where $\sigma(M_1,M_2)$=degree of similarity of methods M_1 and M_2 and$\{I_i\}$=set of instance variables used by method M_i

Complexity of an object: It is defined as cardinality of set of properties.

Complexity of $\{x, p(x)\}=|p(x)|$,where $|p(x)|$ is the cardinality of p(x).

Two design decisions which relate to the inheritance hierarchy can be defined as depth of inheritance of a class of objects and the number of children of the class

Depth of inheritance= depth of the class in the inheritance tree

The depth of the node of a tree refers to the length of the maximal path from the node to the root of tree.

Number of children= Number of immediate descendants of the classes

The Properties on Metric suite

Property 1)Noncoarseness: Let the classes P and Q have a metric μ such that $\mu(P) \neq \mu(Q)$,this means that both have different metric value.

Property 2) Nonuniqueness (notion of equivalence): Let two distinct classes P and Q such that $\mu(P) = \mu(Q)$,this means both have same metric value with equal complexity.

Property 3)Design Details are Important: The two class designs, P and Q, which provide the same functionality, does not imply that $\mu(P) = \mu(Q)$, its specifies the class must influence the metric value.

Property 4) Monotonicity: For all classes P and Q, the following must hold: $\mu(P)\leq \mu(P+Q)$ and $\mu(Q)\leq \mu(P+Q)$ where P+Q implies combination of P and Q.

Property 5) Nonequivalence of Interaction: $\exists P$, $\exists Q$, $\exists R$ such that $\mu(P) = \mu(Q)$ does not imply that $\mu(P+R) = \mu(Q+R)$.This suggests that interaction between P and R can be different than interaction between Q and R resulting in different complexity values for P+R and Q+R.

Property 6) Interaction Increases Complexity: $\exists P$, $\exists Q$ such that $\mu(P) +\mu(Q) < \mu(P+Q)$

The principle behind this property is that when two classes are combined, the interaction between classes can increase the complexity metric value.

The above mentioned properties are mean for to calculate the metrics for Object Oriented concepts.

CK metrics have generated a significant amount of interest and are currently the most well
known suite of measurements for OO software [14]. Chidamber and Kemerer proposed
six metrics; the following discussion shows their metrics.

Weighted Method per Class (WMC)

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities of its methods. Class with low WMC is better when compare to high WMC since high WMC will have more complexity. As WMC is complexity measurement metric, we can get an idea of required effort to maintain a particular class.

Depth of Inheritance Tree (DIT)

DIT metric is the length of the maximum path from the node to the root of the tree. So this metric calculates how far down a class is declared in the inheritance hierarchy. This metric also measures how many ancestor classes can potentially affect this class. DIT represents the complexity of the behaviour of a class, the complexity of design of a class and potential reuse.
If DIT increases, it means that more methods are to be expected to be inherited, which makes it more difficult to calculate a class's behavior. Thus it can be hard to understand a system with many inheritance layers. On the other hand, a large DIT value indicates that many methods might be reused.

Number of children (NOC)

This metric measures how many sub-classes are going to inherit the methods of the parent class.
The size of NOC approximately indicates the level of reuse in an application. If NOC grows it means reuse increases. On the other hand, as NOC increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

Coupling between objects (CBO)

The idea of this metrics is that an object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible. CBO metric measure the required effort to test the class .

Response for a Class (RFC)

RFC is the number of methods that can be invoked in response to a message in a class.
If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism.

Lack of Cohesion in Methods (LCOM)

This metric uses the notion of degree of similarity of methods. LCOM measures the amount of cohesiveness present in a class . LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero.

MOOD METRICS SUITE:

MOOD Metrics are defined as set of metrics for Object Oriented Design concepts. MOOD metrics follows the theoretical validation of encapsulation, inheritance, coupling and polymorphism.

Abreu et al. defined MOOD (Metrics for Object Oriented Design) metrics. MOOD refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). In MOOD metrics model, two main features are used in every metrics; they are methods and attributes. Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects.
Attributes are used to represent the status of each object in the system.

Encapsulation:

The Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) were proposed together as measure of encapsulation. MHF and AHF represent the average amount of hiding between all classes in the system.
Method Hiding Factor (MHF)
The MHF metric states the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. Abreu et al. States, the MHF denominator is the total number of methods defined in the system under consideration. The MHF metric is defined as follows

Method hiding factor(MHF):

$$\sum_{i=1}^{TC}\sum_{m=1}^{Md(Ci)}(1-V(Mmi))/\sum_{i=1}^{TC}Md(Cj)$$

For visible methods

$$V(Mmi)=\sum_{j=1}^{tc} is\_visible\ \ (Mmi,Cj)/(TC-1)$$

is_visible (M_{mi} ,C_j)={1 iff $j \neq i \wedge C_j$ may call M_{mi} , 0 otherwise

The AHF metric shows the sum of the invisibilities of all attributes in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is hidden. MHF and AHF represent the average amount of hiding among all classes in the system.

Inheritance

Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) are proposed to measure inheritance.

Method Inheritance Factor (MIF)

The MIF metric states the sum of inherited methods in all classes of the system under consideration. The degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes. MIF is defined as the ratio of the sum of the inherited methods in all classes of the system as follow.

$$\mathrm{MIF} = \sum \mathrm{Mi(Ci)} / \sum \mathrm{Ma(Ci)}$$

where the summation occurs over i=1 to TC

$Ma(C_i)=Md(C_i)+M_i(C_i)$

Attribute Inheritance Factor (AIF):

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the total numberof available attributes for all classes. It is defined in an analogous manner and provides an indication of the impact of inheritance in the object oriented software.

Polymorphism:

Polymorphism is an important characteristic in object oriented paradigm. Polymorphism measure the degree of overriding in the class inheritance tree.

Polymorphism Factor (POF):

The POF represents the actual number of possible different polymorphic situation. It also represents the maximum number of possible distinct polymorphic situation for class. The POF is defined as follows.

$$PF=\sum_i M_o(C_i) / \sum_i [M_n(C_i)\times DC(C_i)]$$

Where the summation occur over I=1 to TC.

Coupling:

Coupling shows the relationship between module. A class is coupled to another class if it calls methods of another class.

Coupling Factor (COF):

The COF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of coupling is not imputable to inheritance. The COF is defined as follows.

$$\sum_{i=1}^{TC}[\sum_{j=1}^{TC} is\_client(Ci,Cj)]/(TC\hat{}2-TC)$$

Where is_client (Cc,Cs)={1 iff Cc =>Cs^Cc ≠ Cs, 0 otherwise

MOOD Metrics Notations

1. MIF (Method Inheritance Factor)

M_i	The number of methods
C_i	Number of classes
$M_a(C_i)$	The number of methods that can be invoked in association with C_i
$M_d(C_i)$	The number of methods declared in a class
T_c	Total number of classes
$M_i(C_i)$	The number of methods inherited in C_i

2. CF(coupling factor)

Is_client(C_c ,C_s)	The relationship between a client class C_c and supplier class C_s

3.PF(Polymorphism factor)

$M_n(C_i)$	Number of new methods
$M_o(C_i)$	Number of overriding methods
$DC(C_i)$	Number of classes descendants from C_i where the summation occur over I=1 to TC

4. EF(encapsulation factor)

$V_{(Mmi)}$	Visible methods of class
$M_d(C_i)$	The number of methods declared in a class

3.CONCLUSION

Research on software metrics devised to measure the quality of object-oriented software has come a long way. Right from the days of traditional metrics to the modern Object Oriented metrics, numerous metrics have been evolving. Out of all such metrics, most researchers have emphasized over the need and applicability of object-oriented metrics that are obtained from the static analysis of Object Oriented software. These metrics were called static as they were evaluated from the source code or design analysis of a software. Most of the dynamic metrics devised till date measure dynamic coupling. A few metrics measure dynamic cohesion and dynamic complexity. There are no metrics proposed till date for measuring the dynamic inheritance or their impact on dynamic coupling. This research work is primarily directed to propose software metric suite in order to evaluate the complexity of Object Oriented paradigm.

REFERENCES

[1] Chidamber and Kemerer , " Metrics suite for Object Oriented ",IEEE Transactions, Vol 60,1993.

[2] Aman kumar Sharma and Arvind kalia, " Metrics Identification for Measuring Object Oriented Software Quality", International journal of Soft Computing and Engineering (IJSCE) ISSN:2231-2307,Volume -2,Issue-5,

November 2012.

[3] Dr.R.Selvarani and P.Mangayarkarasi "A Dynamic Optimization Technique for Redesigning OO Software for Reusability" ACM SIGSOFT Software Engineering,March 2015 Volume 40, Number 2.

[4] Amandeep Kaur, Satwinder Singh, Dr. K. S. Kahlon and Dr. Parvinder S. Sandhu, "Empirical Analysis of CK & MOOD Metric Suit", International Journal of Innovation, Management and Technology, Vol. 1, No. 5, December 2010 ISSN: 2010-0248