IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Software Metrics Evaluation Using Various Lines Of Code And Function Point Metrics

Avinash Gaur[*1], Anurag Punde[2]

[*1,2] Acropolis Institute of Technology & Research, Indore, India

Abstract

This paper describes the major characteristics of software engineering such as Maintainability, Reliability, Complexity, Understandability, Reusability and Testability. These characteristics measure by some software metrics. There are many software metrics but in this paper our emphasis on those software metrics, which effects the maintainability, Reliability, Complexity and Reusability of the software. Such characteristics can be measured with the help of Coupling, Cohesion, Cyclomatic Complexity, Inheritance, and Comment Percentage and size metrics. These characteristics are used to improve the quality, reliability and understandability of the software and reduced the complexity and cost of the software.

Keywords: Function Point Matrices, Testability, software Metrics.

Introduction

Object-oriented design and development are popular concepts in today's software development environment. There is a general shift in the industry from the structured (traditional) programming and development environment to an object-oriented paradigm. If organizations wish to make a successful change, they need the appropriate metrics for this new paradigm.

Software metrics are necessary for any organization serious about assessing and improving its development process as well as the quality of its products. Developers need to assess the "ileitis" of the system such as reliability, maintainability, reusability, etc. These attributes cannot be evaluated without first being measured. A key element of any engineering process is measurement. Measures are used to better understand the attributes of the model that we create. But, most important, we use measurements to assess the quality of the engineered product or the process used to build it.

Software developers need to explicitly state the relation between the different metrics measuring the same aspect of software. software, we need to identify the necessary metrics that provide useful information, otherwise the managers will be lost into so many numbers and the purpose of metrics would be lost. Since metrics are crucial source of information for decision making

A measure, in general, is the assignment of a number to an entity for the purpose of characterizing a specific attribute of the entity. In software, there are three categories of entities in which all measurable attributes fall: processes, products, and resources. The measures described in this paper fall into the category of product metrics, that is, metric, that measure an attribute of a specific software artifact like a design or code.

There are number of metrics proposed in literature such as coupling, cohesion, information hiding, inheritance, size metrics etc. Our aim is to find out all these metrics are independent or we can take a subset from them.

S. NO.	METRIC OBJECT-ORIENTED	ATTRIBUTE	SOURCES
1	Response for a class (RFC)	Class	[Chidamber94] [1][2]
2	Number of Attributes per Class (NOA)	Class	[Henderson96] [2]
3	Number of Methods per Class (NOM)	Class	[Henderson96][3]

4	Weighted Methods per Class (WMC)	Class	[Chidamber94] [1]
5	Coupling between Objects (CBO)	Coupling	[Chidamber94] [1]
6	Data Abstraction Coupling (DAC)	Coupling	[Henderson96] [3]
7	Message Passing Coupling (MPC)	Coupling	[Henderson96] [3]
8	Coupling Factor (CF)	Coupling	[Harrison98] [4]
9	Lack of Cohesion (LCOM)	Cohesion	[Chidamber94] [1]
10	Tight Class Cohesion (TCC)	Cohesion	[Braind99] [5][6] [7]
11	Loose Class Cohesion (LCC)	Cohesion	[Braind99] [5][6][7]
12	Information based Cohesion (ICH)	Cohesion	[Lee95] [8]
13	Method Hiding Factor (MHF)	Information Hiding	[Harrison98] [4]
14	Attribute Hiding Factor (AHF)	Information Hiding	[Harrison98] [4]
15	Number of Children (NOC)	Inheritance	[Chidamber94] [1]
16	Depth of Inheritance (DIT)	Inheritance	[Chidamber94] [1]
17	Method Inheritance Factor (MIF)	Inheritance	[Harrison98] [4]
18	Attribute Inheritance Factor (AIF)	Inheritance	[Harrison98] [4]
19	Number of Methods Overridden by a subclass (NMO)	Polymorphism	[Henderson96] [3]
20	Polymorphism Factor (PF)	Polymorphism	[Harrison98] [4]
21	Reuse ratio Reuse	Reuse	[Henderson96] [3]
22	Specialization ratio Reuse	Reuse	[Henderson96] [3]
23	Cyclomatic complexity(CC)	Traditional	[Mc Cabe] [9][10]
24	Comment percentage	Traditional	[Mc Cabe] [9][10]
25	Size	Traditional	[Fenton96][11]

Criteria for Metrics Evolution

While metrics for the traditional functional decomposition and data analysis design approach measure the design structure and/or data structure

independently, object-oriented metrics must be able to focus on the combination of function and data as an integrated object . The evaluation of the utility of a metric as a quantitative measure of software quality was based on the measurement of a software quality attribute. The metrics selected, however, are useful in a wide range of models. The object-oriented metric criteria, therefore, are to be used to evaluate the following attributes:

- **Efficiency** - Are the constructs efficiently designed?
- **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?
- **Testability/Maintainability** - Does the structure support ease of testing and changes?

Metrics Definition And Applications Methods

In an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. A method is a component of an object that operates on data in response to a message and is defined as part of the declaration of a class. Methods reflect how a problem is broken into segments and the capabilities other classes expect of a given class. Two traditional metrics are discussed here: cyclomatic complexity and size (line counts).

METRIC 1: Cyclomatic Complexity (CC)

Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. A method with a low cyclomatic complexity is generally better, although it may mean that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. In general, the cyclomatic complexity for a method should be below ten, indicating decisions are deferred through message passing. Although this metric is specifically applicable to the evaluation of quality attribute Complexity, it also is related to all of the other attributes.

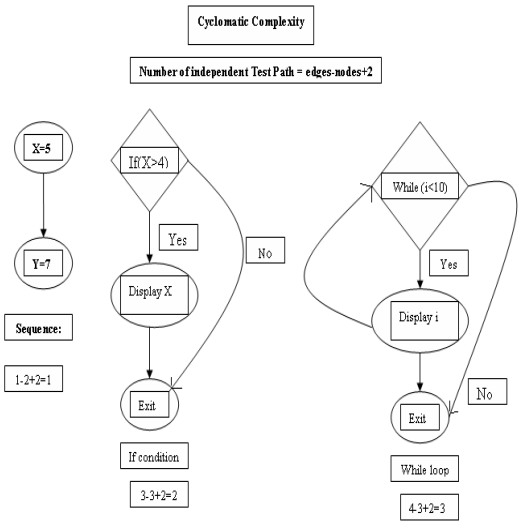

Figure-1 ***Title***

Figure 1 shows a method with a low cyclomatic complexity is generally better. This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of complexity, it also is related to all of the other attributes.

METRIC 2: Size

Size of a method is used to evaluate the ease of understandability of the code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, and the number of blank lines. Lines of Code (LOC) count all lines. Non-comment Non-blank (NCNB) is sometimes referred to as Source Lines of Code and counts all lines that are not comments and not blanks Executable Statements (EXEC) is a count of executable statements regardless of number of physical lines of code. For example, in FORTRAN and IF statement may be written:

IF X=3 THEN Y=0
LOC = 3
NCNB = 3
EXEC = 1

Executable statements is the measure least influenced by programmer or language style. Executable statements evaluate project size. Thresholds for evaluating the size measures vary depending on the coding language used and the complexity of the

method. However, since size affects ease of understanding, routines of large size will always pose a higher risk in the attributes of Understandability, Reusability, and Maintainability.

METRIC 3: Comment Percentage
The line counts done to compute the Size metric can be expanded to include a count of the number of comments, both on-line (with code) and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. It has found a comment percentage of about 30% is most effective. Since comments assist developers and maintainers, this metric is used to evaluate the attributes of Understandability, Reusability, and Maintainability.

Comment percentage=Total number of comments (Total lines of code-Number of blank spaces)

OBJECT-ORIENTED Specific Metrics
As discussed, many different metrics have been proposed for object-oriented systems. The object-oriented metrics that were chosen to measure principle structures that, if improperly designed, negatively affect the design and code quality attributes.
The selected object-oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. For some of the object-oriented metrics discussed here, multiple definitions are given, since researchers and practitioners have not reached a common definition or counting methodology. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics.

A Class
A class is a template from which objects can be created. This set of objects share a common structure and a common behavior manifested by the set of methods. Three class metrics described here measure the complexity of a class using the class's methods, messages and cohesion.

A.1 Method
A method is an operation upon an object and is defined in the class declaration.

METRIC 4: Weighted Methods per Class (WMC)
The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by Cyclomatic complexity). The second measurement is difficult to implement since not all methods are accessible within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children since children inherit all of 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 metric measures understandability, maintainability, and reusability.
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.

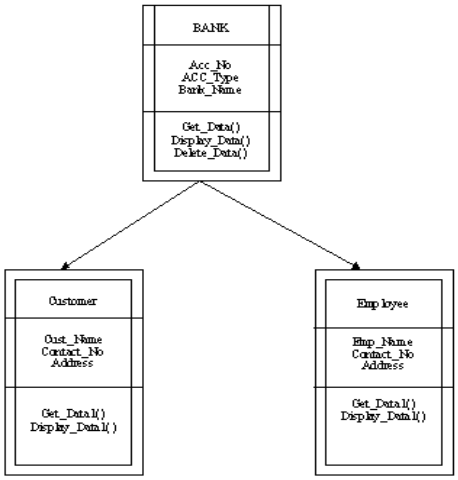

Figure-2: Class Diagram of BANK

Consider a class K1, with methods M1… Mn that are defined in the class. Let C1 ….Cn be the complexity of the methods [Chidamber94].

$$WMC = \sum_{i=1}^{n} Ci$$

If all method complexities are considered to be unity, then WMC = *n*, the number of methods in the class. **In Figure 2**, WMC for BANK is 3 (considering each method complexity to be unity).

A.2 Message
A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. The next metric looks at methods and messages within a class.

METRIC 5: Response for a Class (RFC) (size metrics)
The **RFC** is the carnality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.

This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. This metric evaluates Understandability, Maintainability, and Testability.

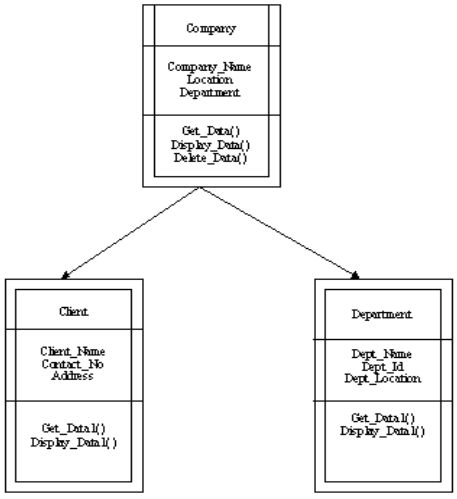

Figure-3: Class Diagram of Company

The response set of a class (RFC) is defined as set of methods that can be potentially executed in response to a message received by an object of that class. It is given by RFC=|RS|, where RS, the response set of the class, is given by

RS= Mi U all j{R ij}

Where Mi = set of all methods in a class (total n) and Ri = {Rij} = set of methods called by Mi.

In Figure 3, class Company has two functions Get_Data and Display_Data which call methods Client::Get_data1 (), Department::Get_data1(), Client :: Display_Data1(), Department :: Display_Data1().

RS = {Company:: Get_Data, Company :: Display_Data, Company :: Delete_Data} U {Client::Get_Data1, Client:: Display_Data1} U {Department :: Get_Data1, Department:: Display_Data1 }

RFC=7

A.3 Cohesion

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective object-oriented designs maximize cohesion since it promotes encapsulation. The third class metrics investigates cohesion.

METRIC 6: Lack of Cohesion of Methods (LCOM)

LCOM measures the degree of similarity of methods by data input variables or attributes (structural properties of classes. Any measure of separateness of methods helps identify flaws in the design of classes. There are at least two different ways of measuring cohesion:

1. 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.

2. 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 Efficiency and Reusability.

Consider a class C1 with n methods M1, M2…., Mn. Let (Ij) = set of all instance variables used by method Mi. There are n such sets {I1} … {In}.

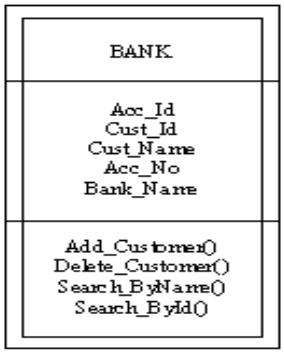

Figure-4: Class Diagram of BANK

Let,P {(I I) | I I 0} and Q{((I I) | I I 0} = , j i ∩ j = = , j i ∩ j ≠ i i . If all n sets{(I1} ... (In)} are 0 then P=0

LCOM= | P | - | Q |, if | P | >| Q |
=0 otherwise

In Figure 4, there are four methods M1, M2, M3 and M4 in class BANK.

I1= {Acc_Id, Cust_Id, Cust_Name, Acc_No, Bank_Name}

I2= {Cust_id}

I3= {Bank_Name}

I4= {Cust_Name}
I1 ∩I2, I1 ∩I3, I1 ∩I4 are non-null but I2 ∩I3,
I2 ∩I4, I3 ∩I4 are null sets.
LCOM is 0 if numbers of null intersections are not greater than number of non-null intersections. Hence LCOM in this case is 0 [|P|=|Q|=3]. Thus a positive high value of LCOM implies that classes are less cohesive. So a low value of LCOM is desirable.

A.4 Coupling

The degree to which components depend on one another. Classes (objects) are coupled three ways:
1. When a message is passed between objects, the objects are said to be coupled.
2. Classes are coupled when methods declared in one class use methods or attributes of the other classes.
3. Inheritance introduces significant tight coupling between super classes and their subclasses.
Since good object-oriented design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling. The next object-oriented metric measures coupling strength.

METRIC 7: Coupling Between Object Classes (CBO)

CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates Efficiency and Reusability.
In Figure 5, Company class contains declarations of instances of the classes Client and Department. The Company class delegates its Client and Department issues to instances of the Client and Department classes. The value of metric CBO for class Company is 2 and for class Client and Department is zero.

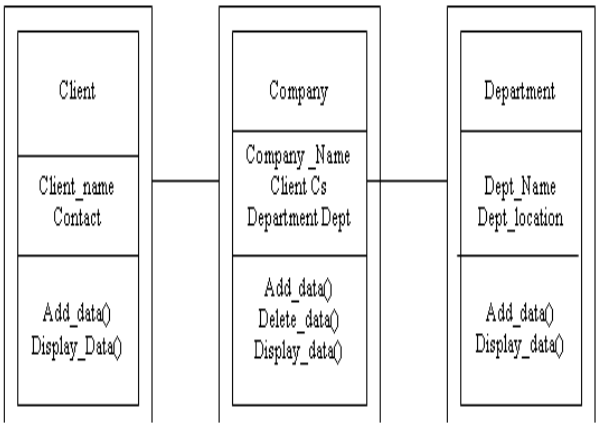

Figure-5: Class Diagram of Company

B Inheritance

Another design abstraction in object-oriented systems is the use of inheritance. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects including variables and operators. 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.

METRIC 8: Depth of Inheritance Tree (DIT)

The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited (NMI). This metric primarily evaluates Efficiency and Reuse but also relates to Understandability and Testability.
In Figure 6, DIT for TotalEmp class is 2 as it has 2 ancestor classes Domestic/International and Company.
DIT for Domestic and International class is 1 as it has one ancestor class Company.

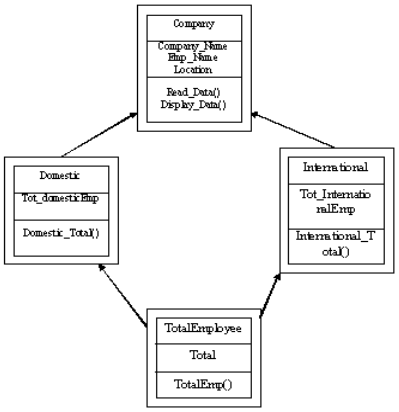

Figure-6: Class Diagram of Company

METRIC 9: Number of Children (NOC)

The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of sub classing. But the greater the number of children, the greater the reusability since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates Efficiency, Reusability, and Testability.

In figure-5, NOC for Class Company is 2.

Summary

In this paper we have shown the use of different matrices to properly understand and effectively increase the quality of the software. We have created own programs and try to find the complexity that arises in the quality measurement. The matrices are the essential part in the software industries that helps us to judge the quality of the product before implementing it. All the matrices that were proposed by different researchers can be used efficiently to assure the quality of the product. Not all the matrices are required. It depends on the type of software that we need to evaluate. The new technologies that are arising in the market are supposed to evaluate by the use of matrices. So, in the next paper we will try to implement software that can measure the quality of the product and tell use the appropriate steps if we fail to achieve the quality

References

[1] S.R.Chidamber and C.F.Kamerer, A metrics Suite for Object-Oriented Design. IEEE Trans. Software ngineering, vol. SE-20, no.6, 476-493, 1994.

[2] Shyam R. Chidamber, Chris F. Kemerer, A METRICS SUITE FOR OBJECT ORIENTED DESIGN, 1993

[3] B.Henderson-sellers, Object-Oriented Metrics, Measures of Complexity. Prentice Hall, 1996.

[4] R.Harrison, S.J.Counsell, and R.V.Nithi, An Evaluation of MOOD set of ObjectOriented Software Metrics. IEEE Trans. Software Engineering, vol. SE-24, no.6, pp. 491-496 June1998.

[5] L.Briand, W.Daly and J. Wust, Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering, 3 65-117, 1998

[6] L.Briand, W.Daly and J. Wust, A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on software Engineering, 25, 91-121,1999.

[7] L.Briand, W.Daly and J. Wust, Exploring the relationships between design measures and software quality. Journal of Systems and Software, 5 245-273, 2000.

[8] Y.Lee, B.Liang, S.Wu and F.Wang, Measuring the Coupling and Cohesion of an Object-Oriented program based on Information flow, 1995.

[9] McCabe & Associates, McCabe Object Oriented Tool User's Instructions, 1994.

[10] McCabe, T.J. "A Complexity Measure." IEEE Transactions on Software Engineering 2, 4 (April 1976): 308-320.

[11] [Fenton96] N.Fenton et al, Software Metrics: A Rigorous and practical approach. International Thomson Computer Press, 1996.

[12] [Venugopal97] K.R. Venugopal, Rajkumar, T.Ravishankar, Mastering C++, Tata McGraw Hill,1997.