An Evolution of Software Metrics: A Review

Rajender Singh Chhillar
Deptt. of Computer Science
Maharshi Dayanand University, Rohtak (India)
chhillar02@gmail.com

Sonal Gahlot
Deptt. of Computer Science
Maharshi Dayanand University, Rohtak (India)
sonalght@gmail.com

ABSTRACT
The main goal of software engineering is to produce good quality reusable software within the given time-period, with minimum cost and that satisfies the user's needs. To enhance the quality of the software, number of techniques should be adopted. In the present scenario lots of quantifiable methods/tools are used to measure the various aspects of the software but with the rapid pace of the technology/paradigms. It requires much more effort and research work with the new paradigms. Software metrics measure different attributes of software like size, complexity, reliability, thereby providing useful information about the external quality aspects of software like reusability, maintainability and testability. This paper discusses the evolution of software metrics from traditional function-oriented to object-oriented to component-based to aspect-oriented paradigm along with advantages and limitations of software metrics. As traditional function-oriented metrics lack in quality parameters like reusability, maintainability and so, object-oriented metrics have become one of the popular concepts in today's software development environment. Object-oriented metrics have been widely accepted because of many attributes like reusability, better abstraction, polymorphism, simplicity, time-saving, cost-effectiveness and easy maintenance in the software code. Similarly, component-based and aspect-oriented metrics take this journey forward to achieve these quality aspects in the software more effectively. This paper also describes how these new paradigms help to improve the software quality, system performance and productivity in software development.

CCS Concepts
• Software and its engineering~Software development process management

Keywords
Software Metrics; reusability; polymorphism; abstraction; object-oriented metrics; component-based metrics; aspect-oriented metrics; reliability; function-oriented metrics.

1. INTRODUCTION
Software metrics are used to measure various attributes of software like size, complexity, reliability, quality and so on. They play an important role in analyzing and improving software quality. Software metrics also provide useful information on external quality aspects of software such as its maintainability, reusability and reliability [1-7, 16, 17]. They are also useful in estimating the efforts needed for testing. Broadly speaking, software metrics are categorized into product and process metrics [16]. Process metrics are used to measure the properties of the process which is used to obtain the software. Process metrics include the cost metrics, efforts metrics, advancement metrics and reuse metrics. They help in predicting the size of final system and determining whether a project is running as per the schedule.

Product metrics are used to measure the properties of the software product. They involve reliability metrics, functionality metrics, performance metrics, usability metrics, cost metrics, size metrics, complexity metrics and style metrics. Products metrics help in improving the quality of different system components and comparisons between existing systems [16,17,18]. In this way, software metrics are very useful and having a number of advantages though with few limitations also.
The rest of this paper is organized as follows : Section 2 describes traditional Function-Oriented Metrics and McCabe's Cyclomatic metric. Section 3 explains Object-Oriented metrics. Section 4 describes Component-Oriented metrics. Aspect-Based Metrics have been explained in section 5.s way

2. TRADITIONAL FUNCTION-ORIENTED METRICS [16, 17]

2.1 Size Metrics deter
The two important size metrics are: Line of Code and Token Count

2.1.1 Line Of Code (LOC)
It is one of the earliest and simplest metrics for calculating the size of computer program. We have to just count the number of lines/statements in a program/software according to counting rules depending upon the programming language used to write the program. It is generally used in calculating and comparing the productivity of programmers. Productivity is measured as LOC/man-month.

Any line of program text excluding comment or blank line, regardless of the number of statements or parts of statements on the line, is considered a LOC. But, the major weakness of LOC is that it gives the same weight to each line i.e statement irrespective of its complexity. But, some statement/line is small in size, less complex and thus easy to understand while other are more in size and complexity and so difficult to understand.

2.1.2 Token Count
In this metric, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these

ICAIP 2017, August 25–27, 2017, Bangkok, Thailand

ACM ISBN 978-1-4503-5295-6/17/08…$15.00
DOI: https://doi.org/10.1145/3133264.3133297

basic symbols. These symbols are called as tokens. The basic measures are

n1 = count of unique operators
n2 = count of unique operands
N1 = count of total occurrences of operators
N2 = count of total occurrence of operands

In terms of the total tokens used , the size of the program can be expressed as N = N1 + N2

2.2 Software Science Metrics [16]

Halstead's model also known as theory of software science, is based on the hypothesis that program construction involves a process of mental manipulation of the unique operators (n1) and unique operands (n2). It means that a program of N1 operators and N2 operands is constructed by selecting from n1 unique operators and n2 unique operands. By using this Model, Halstead derived a number of equations related to programming such as program level, the implementation effort, language level and so on. An important and interesting characteristic of this model is that a program can be analyzed for various feature like size, efforts etc.

Program vocabulary is defined as

$$n = n1 + n2$$

And program actual length as

$$N = N1 + N2$$

One of the hypothesis of this theory is that the length of a well-structured program is a function of n1 and n2 only.

This relationship is known as length prediction equation and is defined as

$$Nh = n1 \log2\ n1 + n2 \log2\ n2$$

Program Volume (V)

The programming vocabulary n = n1 + n2 leads to another size measures which may be defined as :

$$V = N \log 2\ n$$

Besides these, Halstead has reported a number of other useful metrics as reported in [16].

2.3 McCabe's Cyclomatic Metric [17]

McCabe interprets a computer program as a set of strongly connected directed graph. Nodes represent parts of the source code having no branches and arcs represent possible control flow transfers during program execution. The notion of program graph has been used for this measure and it is used to measure and control the number of paths through a program. The complexity of a computer program can be correlated with the topological complexity of a graph. McCabe proposed the cyclomatic number, V(G) of graph theory as an indicator of software complexity. The cyclomatic number is equal to the number of linearly independent paths through a program in its graphs representation. McCabe gave a thumb rule that if any program having V(G) (complexity measure) more than or equal to 10, then that program is more unreliable and more error prone and so should be decomposed to reduce its complexity below 10.

For a program control graph G, cyclomatic number, V(G), is given as:

$$V(G) = E - N + P$$

E = The number of edges in graphs G

N = The number of nodes in graphs G

P = The number of connected components in graph G

Alternately, V(G) of a program P may be calculated as :

Number of Control Structures in program P + 1

3. OBJECT - ORIENTED METRICS

In object-oriented paradigm, emphasis is on data rather than the procedure. Programs are divided into entities known as objects. Data structures are designed to characterize the objects. Functions operating on the data of an object are tied together in the data structures. Data used is generally hidden and cannot be accessed by external functions. Functions help objects to communicate with each other. New functions and data can easily be added as per need. A bottom up approach is followed during program design. Object-oriented paradigm has many good features which are not available in traditional function-oriented paradigm. These features include object, class, inheritance, polymorphism, abstraction, coupling, cohesion, and information hiding. Due to these features object-oriented software development have an edge over traditional function-oriented development in terms of reusability, better design, less complexity and maintenance, more abstraction and so on. [1, 2, 3, 4, 6, 7, 15, 18]

On the basis of various features of OOPS, many researchers have reported various object-oriented metrics [1, 2, 3, 4, 15, 18]. Out of these, two prominent object-oriented metrics suites are explained below:

3.1 Chidamber and Kemerer's Metrics Suite [1]

3. 1. 1 Weighted Methods per Class (WMC)

It is defined as the sum of the complexities of all methods of a class.

The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods. But the second measurement is more difficult to implement because not all methods are accessible within the class hierarchy because of inheritance.

The larger the number of methods in a class means greater the impact on children, since children inherit all of the methods defined in a class.

3. 1. 2 Response for a Class RFC)

It is defined as number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class.

The RFC is the total number of all methods within a set that can be invoked in response to message sent to an object. This includes all methods accessible within the class hierarchy.

This metrics is used to check the class complexity. If the number of method is larger that can be invoked from class through message than the complexity of the class is increased.

3. 1. 3 Lack of Cohesion of Methods (LCOM)

It is defined as the number of different methods within a class that reference a given instance variable.

Cohesion is the degree to which methods within a class are related to one another and work together to provide well bounded behavior.

LCOM uses variable or attributes to measure the degree of similarity between methods.

We can measure the cohesion for each data field in a class; calculate the percentage of methods that use the data field. Average the percentage, then subtract from 100 percent. Lower percentage indicates greater data and method cohesion within the class. High cohesion indicates good class subdivision while a lack of cohesion increases the complexity.

3. 1. 4 Coupling between Object Classes (CBO)

Two classes are coupled when methods declared in one class use methods or instance variables of the other class.

Coupling is a measure of strength of association established by a connection from one entity to another. Classes are coupled in three ways. One is, when a message is passed between objects, the object are said to be coupled. Second way is, the classes are coupled when methods declared in one class use methods or attributes of the other classes. Thirdy, inheritance introduces significant tight coupling between super class and subclass.

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 hierarchy on which a class depends. Excessive coupling is detrimental to modular design and prevent reusability.

3. 1. 5 Depth of Inheritance Tree (DIT)

It is defined as the maximum length from the leaf node to the root of the tree.

Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects, including variables, methods and operators. Inheritance decreases the complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design more difficult.

Depth of class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods and is likely to inherit, making it more complex to predict its behavior. A support metric for DIT is the number of methods inherited.

3. 1. 6 Number of Children (NOC)

It is defined as the number of immediate subclasses.

The number of children is the number of immediate subclasses subordinates to class in the hierarchy. The greater the number of children, the greater the parent abstraction. The greater the number of children, greater the reusability, since the inheritance is a form of reuse. If the number of children in a class is larger than it require more testing time for testing the methods of that class.

3.2 MOOD'S Metrics for Object Oriented Design [19]

3. 2. 1 Method Hiding Factor (MHF)

MHF is defined as ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.

3. 2. 2 Attribute Hiding Factor (AHF)

AHF is defined as ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

3. 2. 3 Method Inheritance Factor (MIF)

MIF is defined as ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods for all classes.

3. 2. 4 Attribute Inheritance Factor (AIF)

AIF is defined as ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes.

3. 2. 5 Polymorphism Factor (PF)

PF is defined as the ratio of the actual number of possible different polymorphic situation for class C_i to the maximum number of possible distinct polymorphic situations for class C_i.

3. 2. 6 Coupling Factor (CF)

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

4. COMPONENT-ORIENTED METRICS [14]

Recently, component-based software development is getting accepted in industry as a new effective software development paradigm. The traditional software product and process metrics are neither suitable nor sufficient in measuring the complexity of software components, which ultimately is necessary for quality and productivity improvement within organisations adopting component based software development.

The term software component was first used by Mcllroy in 1969 with the idea of creating software component in a similar manner to the hardware components, according to some specifications; then constructing a software product by assembling those components together.

Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Components are divided, planned and then connected together in most effective way possible to provide a solution to a well-defined problem.

Components are connected by assembling, adapting and wiring into a complete application. Traditional metrics has to be redefined or enhanced to comply with component-based software development. Many researches are trying to build a suite of software metrics, with particular emphasis on software component assembly.

Narsimhan proposed several metrics, related with component interaction and integration. These metrics are based on the number of interactions (incoming, outgoing and total) among the components and the system. Based on these measures, he proposes Average Interaction Density (AID) and suggests that low value of AID indicates a simple system, which also means lower efforts to do the software risk analysis. It also performs experimentation on a small case study and calculates Incoming Interaction Density(IID), Outgoing Interaction Density (OID) and finally Average Interaction Density(AID) for the system [14].

5. ASPECT-BASED METRICS

A software product/system may be expressed as a set of structured modules. Each of these modules representing a concern such as a functionality or a requirement. Abstractions like objects, classes, methods, attributes offered by object-oriented and component-based paradigms are insufficient to express fully all the concerns of a software system. For example, code handling concerns like code tracing, logging tends to be scattered and weaved all across the objects of the whole program/software. Such concerns are known as crosscutting concerns because they cut functionality of the whole program/software. As a result of this, software modules implemented through object-oriented and component-based paradigms tend to have reduced quality attributes like reusability, comprehensibility and maintainability.

Aspect-oriented paradigm (AOP) is a solution to resolve the problem of crosscutting concerns [10]. This new emerging AOP provides a solution to separate concerns and to ensure a good modularization. [8, 9, 10, 11, 12]

In an aspect-oriented system, the basic program unit is an aspect rather than a procedure or a class. An aspect with its encapsulation of state with associated operations is a significantly different abstraction than the procedure units within procedural programs or class units within object-oriented programs.

The Aspect-Oriented Software Engineering Group aims to develop systematic means for the identification, modularisation, representation and composition of crosscutting concerns (the aspects) throughout the software life cycle.

Aspect-oriented software development (AOSD) is a new paradigm to support separation of concerns in software development. The technique of AOSD makes it possible to modularize crosscutting aspects of a software system. Aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, coding, and implementation. Some examples of crosscutting aspects are exception handling, synchronization, performance optimization, and resource sharing. The major Aspect-Oriented Programming language is AspectJ [20] which is an extension of the Java language. Since the Aspect-oriented paradigm aims at overcoming the limitations of object-oriented and component-based paradigms, it should also be implemented for object-oriented programming languages like C++ and Java.

Aspect-oriented software is supposed to have many advantages like easy to maintain, reuse, and evolve. A few quantitative studies have been conducted till today, and metrics to quantify the amount of maintenance, reuse, and evolution in aspect-oriented software systems are lacking.

Software metrics for aspect-oriented systems are required to verify claims concerning the maintainability, reusability, and reliability of software systems developed by using this emerging paradigm. Though Aspect-Oriented paradigm is still young, yet many researchers have reported a few metrics for this paradigm [11, 13]. Some of which are described below:

5. 1 Number of Aspects

It is equal to the count of the total number of aspects implemented by the developer of the software.

5. 2 Number of Pointcuts per Aspect (NPA)

It equals the number of pointcuts implemented within an aspect.

5. 3 Number of Advices per Aspect (NAA)

It equals the count of the number of advices implemented within an aspect.

5. 4 Degree of Crosscutting per Pointcut (DCP)

Counting the number of classes, which are being crosscut by a pointcut within an aspect.

5. 5 Response for Advice (RAD)

Counting the number of methods as assigned to a specific advice.

6. CONCLUSIONS

In software development, the main focus of software developers and managers is process improvement. This demand has led to the provision of a number of new and improved approaches/paradigms to software development. The most recent and prominent paradigms being object-orientation (OO) and Aspect-orientation (AO). For process improvement, there is a requirement of software measures or metrics to manage the process. This paper introduces the basic metric suite for traditional function-oriented to object-oriented to component-based to aspect-oriented paradigms and compares the same for quality attributes like reusability, reliability, testability and maintainability. It is almost impossible to devise universally valid quality measures and models suitable for all languages in all development environments for different paradigms. Therefore software metrics should be designed and validated for all paradigms including the most prominent object-oriented and aspect-oriented paradigms. At the same time, it is important to keep in mind that software metrics are only guidelines not rules and they provide an indication of the progress that a program/software has made and the quality of design.

7. REFERENCES

[1] Chidamber, Shyam and Kemerer, Chris, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, June, 1994, pp. 476-492.

[2] Lorenz, Mark and Kidd Jeff, "Object Oriented Software metrics", Prentice Hall Publishing,1994.

[3] Victor R. Basili, Lionel Briand and Walcelio L. Melo "A validation of object-oriented design metrics as quality indicators" Technical report, Uni. of Maryland, Deptt. of computer science, MD, USA.April 1995.

[4] "The Role of Object Oriented metrics" from archive. eiffel.com/ doc/ manuals/ technology.

[5] Rajender Singh, and Grover P. S., "A New Program Weighted Complexity Metrics" *Proc. International conference on Software Engg.* (CONSEG'97), January Chennai (Madras) India, pp 33-39.

[6] I. Brooks "Object Oriented Metrics Collection and Evaluation with Software Process" presented at

OOPSLA'93 Workshop on Processes and Metrics for Object Oriented software development, Washington, DC.

[7] "Software quality metrics for Object Oriented System Environments" by Software Assurance Technology Center, National Aeronautics and Space Administration.

[8] K Sirbi and P J Kulkarni, "Metrics for Aspect Oriented Programming- An Empirical Study" *International Journal of Computer applications* (0975-8887), August 2010,Vol 5 No 12.

[9] K Sirbi and P J Kulkarni, "Impact of Aspect Oriented Programming on Software Development Design Quality Metrics- A Comparative Study" Journal of Computing, July 2010, Vol. 2, No. 7.

[10] J. Y. Guyomareh and Y. G. Gueheneue, "On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics", University of Montreal, Canada.

[11] Zhao J., "Measuring coupling in Aspect Oriented Systems" Technical report, Information Processing Society of Japan (IPSJ), 2004.

[12] Zhao J., "Towards a Metrics suite for Aspect Oriented Software" Technical report, Information Processing Society of Japan (IPSJ), 2002.

[13] Kapetanios, E. and Black, S., "On the Notion of Semantic Metric Spaces for Object and Aspect Oriented Software Design", University of Westminister, London, UK.

[14] Gary Haines, David Carney and John Foreman, "Component-Based Software Development/COTS Integration,SEI : copyright 2007, Carnegie Mellon University, URL: http://www.sei.cmu.edu/str/descriptions/cbsd_body.html.

[15] M O Elish and David Rine, "Indicators of Structural Stability of Object-Oriented Designs: A Case Study", Proc. 29th Annual IEEE/NASA Software Engineering Workshop (SEW'05), 2005.

[16] M. H. Halstead, "Elements of Software Science", New York: Elsevier North Holland, 1977.

[17] T. J. McCabe, "A Complexity Measure", IEEE Trans. On Software Engg., vol. SE-2, No. 4, Dec. 1976, pp.308-320.

[18] K. K. Aggarwal et al,"Empirical Study of Object-Oriented Metrics", Journal of Object Technology, vol. 5, 2006, pp. 149-173.

[19] Rachel Harrison, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", IEEE Trans. On Software Engg., vol.24, 1998.

[20] G. Kiczales et.al "An Overview of AspectJ" Lecture notes in computer science 2072, 2001, pp 327-355.