International Journal of Science and Engineering Applications
Volume 11-Issue 05, 66 - 71, 2022, ISSN:- 2319 - 7560
DOI: 10.7753/IJSEA1105.1003

A Literature Survey of Complexity Metrics for Object-Oriented Programs

Nevy Kimani Maina[1]
School of Computing and Information Technology
Murang' a University of Technology, Kenya

Geoffrey Muchiri Muketha[2]
School of Computing and Information Technology
Murang' a University of Technology, Kenya

Geoffrey Mariga Wambugu[3]
School of Computing and Information Technology
Murang' a University of Technology, Kenya

Abstract: Software complexity refers to the factors that determine the complexity level of a software project. High complexity is caused by the many attributes used in the system and the complex logic relationships among these attributes and features. The increased complexity of software is undesirable and affects maintenance. Over the years, Software Engineering scholars recommended several metrics like Halstead metric, cyclomatic complexity, and line of code metrics to deal with the complexity. With the complexity increasing as time goes by, there is a need for better metrics that can evaluate software more effectively. This research aims to develop a metrics model to determine the features that cause high complexity in software design architectures and to implement the multi-language complexity evaluation model for software architectures. Although this is the case, the literature on complexity metrics that implement diagram-centric complexity measures are inadequate. This study presents the outcomes obtained from our survey on metrics utilized in object-oriented environments. The survey comprises a small set of the most common and frequently implemented software metrics, which could be adopted to a group of object-oriented metrics and object-oriented programming. After reviewing the literature, Findings indicate that metrics that employ diagram-centric complexity measures are lacking.

Keywords: Software quality, Software metrics, complexity metrics, Object-oriented programs

1. INTRODUCTION

Attributes of a software are measured using a software metric to improve its quality. Many software metrics for software quality assurance have been proposed and continue to be presented. Software complexity metrics for procedural languages have been demonstrated to highlight program areas that are sophisticated to understand, test, or are prone to errors. Objected-oriented programs for software complexity metrics have been proposed by several researchers. Traditional procedural metrics (McCabe's Cyclomatic Complexity, and Halstead's Software Science) and modifications of them and class and inheritance measures are among the metrics presented so far. However, little research has been done to show that these measurements accurately reflect the complexity of object-oriented programs. Furthermore, it's unclear whether or not typical procedural sizes bear object-oriented complexity. Although most of these measures apply to all programming languages, some metrics are particular to a subset of the languages. Among metrics of this kind, are those that have been proposed for object–oriented programming languages.

Researchers agree that high complexity suggests poor design, which can be uncontrollable at times and impacts software quality. Measures of diagram design can be used to identify large diagrams that could be split or choose design reviews for select diagrams.

Thus, this paper is a literature survey analyzing the current software complexity metrics to determine whether there are gaps in the literature.

The study is partitioned in the following sections and format; section 2 is a brief overview of the basic ideas of object-oriented programs, and section 3&4 presents the existing complexity metrics for software. Future recommendations and the conclusions are presented in section 5.

2. BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMS

OOP (Object-Oriented Programming) has been advertised to lead to high-quality software and enhance efficiency of the programmer by reusing code.

The following are some of the most widely used terminologies in object-oriented metrics:

1. Object: An object is a type of entity that may save a state and perform various operations on that state.
2. Message: it can be defined as a request for an object to operate on another object.
3. Class: A collection of objects with a shared structure and behavior expressed by methods. It acts as a template from which an item can be created.
4. Method: A method on an object that is available to all class instances does not have to be unique.
5. Instantiation: Creating an object instance and binding or adding data to it.

International Journal of Science and Engineering Applications
Volume 11-Issue 05, 66 - 71, 2022, ISSN:- 2319 - 7560
DOI: 10.7753/IJSEA1105.1003

6. Inheritance: A class-to-class connection where an item in one class inherits features from more than one classes.
7. Cohesion: How closely the methods in a class are related to one another.
8. Coupling: Object A and Object B are connected if and only if A sends a message to B.

The main distinctions between object-oriented programming (OOP languages) with classic procedural programming (CPP) are message forwarding, encapsulation, and inheritance. OOP encapsulates data and behavior (methods) in classes and objects (instances of classes). The meaning of encapsulation is that a programmer only interacts with an object via its interface while the inner workings of an object are hidden. Encapsulation also prevents unintended consequences in other items. Rather than calling a procedure or function, objects in an OOP setup sends message to entities responsible for performing the activity. Inheritance enables programmers to create class hierarchies in which characteristics of more broad and straightforward parent classes are inherited by sub-classes. Sub-classes can also be specialized by overriding or including parts of the inherited code. One of the main benefits of OOP is that inheritance encourages and enables code reuse. Because OOP is so young, there are several ideas, recommendations, or methodologies accepted globally for writing programs that are of high-quality. Furthermore, little research has been conducted to analyze what makes an OOP application difficult and complex.

3. TRADITIONAL SOFTWARE COMPLEXITY METRICS

Software complexity measurements indicate how easy or difficult it is for a programmer to accomplish normal programming activities like understanding, testing, and maintaining a program. The degree to which the qualities assumed to lead to complexity within the code is measured by software complexity metrics rather than the complexity itself. The extent to which certain code qualities appear in the code influences how easy or difficult it is for a programmer to work with it. It might be difficult to test if a program has a convoluted control flow and multiple application routes. As a result, the number of conditional or looping statements might be used to measure complexity.

The metrics described here were chosen from among the most widely used traditional software metrics that have been proposed and could easily be applied to object-oriented programming.

3.1 Line of Code (LOC)

The LOC has been in existence for quite some time, it is more basic, and the most common metric for calculating the size of a program [1,2] Line of code LOC refers to a program's number of instructions in the SLOC (Source Line of Code), excluding comments and black lines. LOC has been criticized for lacking accountability, functionality, cohesiveness, lack of counting standards, and language and programmer dependency [2]. SLOC has other alternatives which include thousands or KLOC (Kilo Lines of Code), thousands of delivered source instructions (KDSI), bytes or number of characters, and non-commented lines of code (NCLOC) [2]. Both LOC and its derivatives, on the other hand, have restrictions.

3.2 McCabe Complexity Model

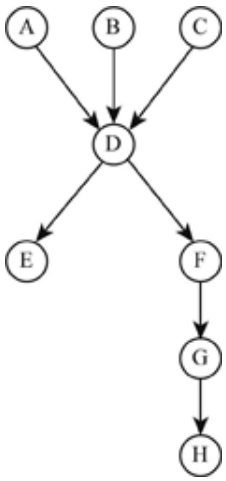
This model focuses on data flow in the architecture [3]. The program is represented by the metrics as a graph, and the definition of complexity, C is as follow;
C=E-N+2P

N represents the number of nodes, P represents the number of connected components, and E represents the number of edges.

One of the issues with McCabe's complexity is that it does not have different control flow statements (conditional statements) and nesting levels of varying control flow structures.

For instance, an edge can be a function/method call, a use relationship, or an inheritance link.

The restrictions must be remembered while using this metric, and the mapping between the graph and the model elements should be clearly defined.



3.3 Halstead Complexity

Software science by Halstead is based on the advancement of determining the size of the program through counting lines of code [4]. Halstead's metrics determine the number of operands and the number of operators and their respective occurrence in the code (program). The operands and operators are considered when measuring Program Vocabulary, Length, Estimated Program Length, Potential Volume, Effort, and Difficulty.

Critics have characterized Halstead as being complex to compute and depending on a complete code [5]. They are also criticized for being inadequate and confused. However, from a perspective of measurement theory, they are reasonable [6] and have solved line of code weaknesses where the computer algorithm is defined as a collection of tokens [7].

3.4 Henry and Kafura's Metrics

The complexity of a module based on the fan-out and fan-in of data flow is defined by the Henry-Kafura Information flow [8]. The module indicates that all sets of procedures refer to a certain global variable.

International Journal of Science and Engineering Applications
Volume 11-Issue 05, 66 - 71, 2022, ISSN:- 2319 - 7560
DOI: 10.7753/IJSEA1105.1003

A procedure's complexity considers the sophistication of its code in terms of the length of lines and how complex its connected to its surrounding in terms of whether it is fan-out or fan-in. **Fan-in** is the amount of local flows that end at the procedure and the number of worldwide variables from which the function obtains data. **Fan-out** is the amount of regional flows from the process and the number of variable updates.

$$\text{Complexity} = \text{length of the procedure} \times [\text{fan - in} \times \text{fan - out}]^2$$

The metrics used to determine the structural sophistication are fan-in and fan-out. They also help to define maintainability. These two measures can be defined for files and procedures. An example is shown in the figure below, which links them as edges and modules as nodes. For example, consider the following graph with the modules as nodes and the links as edges.

Fan-in of a particular module shows the total modules that depends on it. A particular module's fan-out indicates the number of modules that depend on this module. The figure above shows that Module-D has a fan-out of 2 links and a fan-in of 3 links.

If the fan-in of a module is higher, it represents a better design structure; that is, the module has been used several times. Thus, it can be utilized for re-usability and decreases cost redundancy.

Fan-out shows are coupled among various modules. If the fan-out is high, this is an indication that the module is highly coupled. The higher the fan-out, the more the maintainability.

Shepperd [9] recommended changing H&K's data flow metric. The measure of Information Flow in the Shepherd's refinement to the complexity of Henry and Kafura for module M is;

$$\text{Complexity} = (\text{fan-in} * \text{fan-out})^2$$

The refinement was proposed to measure while excluding the length factor. Improvements by Shepperd [9] recorded a particular perception of the structure of information flow; thus, they coincide with measurement theory. The empirical validation by Shepherd the relationship between the measure and a certain process measure referred to as development time. In Shepperd's data, the relationship between K&F measure and development time was insignificant. But, his pure-data flow structure was found to be significantly related. Thus, the level of data flow is closely related with development time [10].

The Shepherd's refinement to the H&K measure of IFC for a module was analyzed by Sofia Nystedt and Claes Sandros [11] and indicated that the two are not extremely helpful while predicting program's errors. However, various metrics packages calculate the information flow complexity with multiple formulas.

4. OOP SOFTWARE COMPLEXITY METRICS

Code reuse is the strongest argument that favors OOP. This is because it permits applications to be built faster and, at the same time, enhances software quality. Although this is the case, the benefits are only evident if the reused code is evident and is of high quality. In the recent past, various OOP software complexity metrics have been recommended as a measure of the quality of software. The majority of them are either quantitative measures of OOP or traditional software complexity metrics extension, measuring features perceived to lead to complexity.

The analysis of these metrics is presented in the subsequent sections.

4.1 Chdamber and Kemerer Metrics

Various metrics have been defined for the object-oriented domain. One of the most common metrics are Chidamber and Kemerer metrics. Chidamber & Kemerer (1994) are Weighted Methods per Class (MMC), and Depth of Inheritance Tree (DIP), which can be used to determine the maximum length from the root to the node of the tree, where greater design complexity is made by deeper trees. Number of Chidren (NOC) shows the number of immediate sub-classes that are subordinated to a class from the class hierarchy. Coupled between Object Classes), which is the count of number of classes which couples it. Response for a Class (RFC) refers to a set of methods can be adopted to respond to a message gotten by an object class and Lack of Cohesion in Methods (LCOM) which refers to the degree of similarity of methods. A class is more cohesive if the amount of similar methods is more significant. Various researchers have empirically approved the metrics [12,13,14,15]. Although this is the case, researchers have found them theoretically deficient [16,17].

4.2 MOOD Metrics Suite

The MOOD metrics object-oriented domain structural complexity measures. These metrics were proposed in 1994 [24]. Method Hiding Factor (MHF), Attribute Inheritance Factor (AIF), Attribute Hiding Factor (AHF), Coupling Factor (CF), Polymorphism Factor (PF) and Method Inheritance (MIF) were recommended in 1994 [18]. The MHF and AHF were proposed as measures of encapsulation. The MHF metric is the ratio of the invisibilities specified method in all classes to the sum of attributes defined.

In contrast, the Attribute Hiding Factor is the ratio of all attribute invisibilities declared in all classes to the sum of all attributes. Both AIF and MIF are based on inheritance. The Method Inheritance metric is the sum of all methods inherited in the entire classes divided by the sum of all available methods. The AIF statistics on the other hand, is the sum of all attributes inherited in all classes divided by the total number of attributes available in all classes. PF is the ratio of the real number polymorphic scenarios for a given class to the maximum number of various polymorphic scenarios for the same class. The coupling factor is the ratio of the greatest

possible number of non-inherited connections [25]. These measures have been chastised for failing to anticipate class errors [26].

4.3 Mishra Inheritance Metrics

Two inheritance metrics were proposed by Mishra (2012), which are program level ACI (Average Complexity Inheritance), and class level CCI (Class Complexity due to Inheritance). There is a light at the end of the tunnel since the metrics have been found to be mathematically correct through the use of Weyuker's properties. Although the metrics need to be verified empirically to determine whether they can be useful measures of software quality.

4.4 Li Metrics

Six metrics were proposed by Li (1998) to solve the limitations of C&K metrics [17]. The metrics include Number of Ancestor Classes (NAC), Number of Descendent Classes (NDC), Number of Local Methods (NLM), Couple Through Abstract Data Type (CTA), Class Method Complexity (CMC), and Coupled Through Message Passing (CTM). The NAC determines the total number of ancestor classes inherited by a class. The total number of local methods in a class is measured by the number of local methods that can be analyzed outside the class. The CMC metric totals the internal structure complexity of all local methods. An NDC metrics provides the sum of sub-classes of a class. The CTA measures the total number of classes that are utilized as abstract data types. In conclusion, the Coupling Through Message Passing metric returns the number of various messages sent from a class to different classes without considering the inheritance characteristic [23]. HoI metrics solved the gaps in C&K metrics since they required modifications to effectively approximate maintainability.

4.5 Abreu and Carapuca Metrics

Five metrics were defined by Abreu and Carapuca (1994) that are utilized to determine inheritance in OOP [18]. These include Total Progeny Count (TPC), Total Children Count (TCC), Total Parent Count (TPAC), Total Length of Inheritance Chain (TLI), Total Ascendancy Count (TAC). The TCC is the number of classes directly inherited. TPC is the number of classes that directly or indirectly inherits from a class. TPAC is the number of sub-classes from which a class is inherited directly. TAC was represented and defined as the number of super-classes from which a class inherits directly or indirectly. Lastly, the inheritance total length is the amount of edges in the inheritance hierarchy graph. The metrics focused only on the inheritance perception of the OOP and other structural perception of a program.

4.6 Lorenz and Kidd Metrics Suite

Three metrics were derived by Lorenz and Kidd (1994) which include NMI (Number of Methods), NNA (Number of New Methods), and NMO (Number of Methods Overridden) [19]. The number of methods measures the total number of methods which a subclass inherits from. In contrast, the number of methods overridden by a subclass and a class, and number of new methods measures the number of new strategies in a subclass [20]. The metrics have been criticized to measure class properties and to be simplistic. This is an indication that they cannot be depended on to analyze the quality of a software [21, 22].

4.7 Misra, Adewumi, Fernandez-Sanz and Damasevicius Metrics

Objected oriented complexity measures were proposed [27]. MC (Method Complexity), AC (Attribute Complexity), CWC (Coupling Weight for a Class), CLC (Class Complexity), and CC (Code Complexity) are some of the measures used. The MC metric is calculated by adding all of a class's allocated weights. The weights of calls and called methods are added to the CWC metric. The sum of features of a class is determined using the AC metric. By adding AC and MC, the CLC measure determines class complexity. Finally, the CC metric considers the interaction between classes, which increases the complexity of the classes. The weights of subclasses are multiplied, and all classes in the same level are allocated the same weight. These measures have been shown to be theoretically valid, but they must be tested in real-world applications to be useful.

5. CONCLUSIONS AND FUTURE WORK

This study's findings indicate that almost all software metrics calculate are model-centric measurements of software and not diagram-centric. Class metrics, for example, tally all of a class's attributes, affiliations, operations, and so on. It makes no difference whether these elements appear on any diagrams or the classes themselves. Diagram-centric metrics are also intriguing for practical reasons. We can utilize said diagram size and complexity metrics to find large diagrams that can be divided up or choose diagrams for design reviews and inspections.

In an attempt to solve the lack of diagram-centric complexity measures that implement diagram size and complexity measures, future studies should focus on defining complexity metrics for the measurement of complexity during the design modules. The metrics should be capable of use at the architectural and detailed design stages and assist in preventing module implementation and maintenance problems. Further, the results of the experimental evaluation of these metrics will assist in demonstrating the benefits of the design method in controlling complexity through the software life-cycle and hence in demonstrating the ability to assist in producing maintainable design products and software.

Nevy Kimani Maina is an ICT OFFICER at the Department of Information Communication Technology at Murang'a County Assembly, Kenya. He earned his Bachelor of Business and Information Technology (BBIT) from Murang' a University of Technology in 2016. He is currently pursuing his MSc. in Information Technology at Murang'a University of Technology, Kenya. His research interests include software metrics, software quality, and business intelligence.

Geoffrey Muchiri Muketha is the Director, Directorate of Postgraduate Studies Murang' a University of Technology, Kenya. He received his BSc. in Information Science from Moi University in 1995, his MSc. in Computer Science from Periyar University, India in 2004, and his Ph.D. in Software Engineering from Universiti Putra Malaysia in 2011. He has wide experience in teaching and supervision of postgraduate students. His research interests include software and business process metrics, software quality, verification and validation, empirical methods in software engineering, and component-based software engineering. He is a member of the International Association of Engineers (IAENG).

Dr. Geoffrey Mariga Wambugu Dean of the School of Computing and Information Technology, Murang' a University of Technology, Kenya. He obtained his BSc Degree in Mathematics and Computer Science from Jomo Kenyatta University of Agriculture and Technology in 2000, and his MSc Degree in Information Systems from the University of Nairobi in 2012. He holds a Doctor of Philosophy in Information Technology degree from JKUAT. His interests include Machine Learning and Text Analytics. Dr. Mariga has been involved in the design, development and implementation of IT/ICT and Computer Science Curricula in different Universities and Colleges in Kenya.

6. REFERENCES

[1] Debbarma, M. K., Debbarma, S., Debbarma, N., Chakma, K., & Jamatia, A. (2013). A review and analysis of software complexity metrics in structural testing. *International Journal of Computer and Communication Engineering*, *2*(2), 129-133.

[2] Albin, T, S. Art Of Software Architecture, vol. 1. John Wiley And Sons Ltd, New York, 2013.

[3] Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahao, S., & Ribeiro, A. (2015). Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, *101*, 245-259.

[4] Bagheri, H., Garcia, J., Sadeghi, A., Malek, S., & Medvidovic, N. (2016). Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software*, *119*, 31-44.

[5] Barillari, F., Gorga, I., & Piccinini, S. (2018). *U.S. Patent Application No. 10/025,586*.

[6] Fenton, N., & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC press.

[7] Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

[8] Bhatia, M. P. S., Kumar, A., & Beniwal, R. (2016). Ontologies for software engineering: Past, present and future. *Indian Journal of Science and Technology*, *9*(9).

[9] Bonet, R., & Salvador, F. (2017). When the boss is away: Manager–worker separation and worker performance in a multisite software maintenance organization. *Organization Science*, *28*(2), 244-261.

[10] Hourani, H., Wasmi, H., & Alrawashdeh, T. (2019, April). A code complexity model of object oriented programming (OOP). In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)* (pp. 560-564). IEEE.

[11] Sandros, C., & Nystedt, S. (1999). Software Complexity and Project Performance.

[12] Denaro, G., Lavazza, L., & Pezze, M. (2003, November). An empirical evaluation of object oriented metrics in industrial setting. In *The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal*.

[13] Basili, V.R. Rombach, H.D. 'The TAME Project: Towards Improvement- Oriented Software Environments'. IEEE Trans, on Softw. Eng. 14(6) pp758-773.1988.

[14] Muketha, G.M. (2011). Size and complexity metrics as indicators of maintainability of business process execution language process models (Doctoral dissertation, Universiti Putra Malaysia).

[15] Ndia, John & Muketha, Geoffrey & Omieno, Kelvin. (2019). A SURVEY OF CASCADING STYLE SHEETS COMPLEXITY METRICS. International Journal of Software Engineering & Applications. 10. 21-33. 10.5121/ijsea.2019.10303.

[16] McCall, J.A., Richards, P.K., and Walters, G.F., (1977) "Factors in Software Quality", RADC TR-77-369, Vols I, II, III, US Rome Air Development Centre Reports.

[17] Hansen, P., & Hacks, S. (2017). Continuous Delivery for Enterprise Architecture Maintenance. *Full-scale Software Engineering/The Art of Software Testing*, *56*.

[18] Lin, C. J., & Yeh, D. M. (2016, December). A Software Maintenance Project Size Estimation Tool Based On Cosmic Full Function Point. In *Computer Symposium (ICS), 2016 International* (pp. 555-560). IEEE.

[19] Linos, P., Lucas, W., Myers, S., & Maier, E. (2007, November). A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications* (pp. 324-329). ACTA Press.

[20] Baroni, A. L., & Abreu, F. B. (2003, July). An OCL-based formalization of the MOOSE metric suite. In *Proc. 7th ECOOP Workshop on Quantitative Approaches in Object-Orietend Software Engineering*.

[21] King'ori, Ann Wambui and Muketha, Geoffrey Muchiri and Micheni, Elyjoy Muthoni, A Literature Survey of Cognitive Complexity Metrics for Statechart Diagrams (July 31, 2019). International Journal of Software Engineering & Applications (IJSEA), Vol.10, No.4, July 2019,

[22] Harrison, R., Counsell, S., & Nithi, R. (1997, July). An overview of object-oriented design metrics. In *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering* (pp. 230-235). IEEE.

[23] Gupta, A., & Jha, R. K. (2015). A survey of 5G network: Architecture and emerging technologies. *IEEE access*, *3*, 1206-1232.

[24] e Abreu, F. B., & Carapuça, R. (1994). Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, *26*(1), 87-96.

[25] Neelamegam, C., & Punithavalli, M. (2009). A survey-object oriented quality metrics. *Global Journal of Computer Science and Technology*, *9*(4), 183-186.

[26] Shaik, A., Reddy, K., & Damodaram, A. (2012). Object oriented software metrics and quality assessment: Current state of the art. *International Journal of Computer Applications*, *37*(11), 6-15.

[27] Misra, J. R., & Irvine, K. D. (2018). The Hippo signaling network and its biological functions. *Annual review of genetics*, *52*, 65-87.