KNOWLEDGEABLE RESEARCH (KR)
ISSN: 2583-6633 | CODEN: KRABAU
International Open-Access Peer-Reviewed Refereed Multidisciplinary Journal
https://knowledgeableresearch.com
Vol. 5, Issue 07,2026

Received: 08/06/2026 | Accepted: 24/07/2026 | Published: 30/07/2026

# STUDY OF EVALUATION OF COMPLEXITY METRICS

**Sundeep Kumar Awasthi**
*Department of Computer Science, Swami Shukdevanand College Shahjahanpur, Uttar Pradesh*

## Abstract

*Software complexity metrics are essential tools for assessing the quality, maintainability, reliability, and performance of software systems. They provide quantitative measures that help developers, testers, and project managers evaluate the structural and functional characteristics of software throughout its development lifecycle. This study examines the concept, significance, and evaluation of software complexity metrics, with particular emphasis on widely used measures such as Cyclomatic Complexity, Halstead Metrics, Lines of Code (LOC), and Object-Oriented metrics including the Chidamber and Kemerer (CK) metric suite. The paper analyzes the advantages, limitations, and practical applications of these metrics in identifying complex code, predicting software defects, estimating maintenance effort, and improving software quality. A comparative evaluation highlights the effectiveness of different metrics in various software development environments and demonstrates their role in supporting informed decision-making during software design, testing, and maintenance. The study concludes that no single complexity metric can comprehensively measure software complexity; instead, a combination of complementary metrics provides a more accurate and reliable assessment, leading to the development of robust, efficient, and maintainable software systems*

***Keywords:*** *Object-Oriented Programming (OOP), Software Complexity Metrics, Type, Complexity, Control Flow Complexity, Class-Level Metrics, Encapsulation*

***Corresponding Author:**
**Sundeep Kumar Awasthi**
Email: sndp67319@gmail.com

## Introduction

Object-oriented languages have achieved significant gains in popularity over the past few years. Capers Jones has predicted that "By 2000, it can be anticipated that object-oriented languages will dominate for standard systems software."[1] However, in order for object-oriented languages to sustain this growth in popularity it will become increasingly important for development managers, as well as for programmers, to

understand both the advantages and disadvantages of using an object-oriented language. In order to do so it will be necessary to better understand the effects of the various constructs of a language on the complexity of source code written in that language. This understanding can be achieved through the use of complexity metrics that are specifically intended to measure the complexity of object-oriented programs. Much as the claims that structured programming led to less complicated programs were supported by metrics such as those put forth by Halstead and McCabe, it will be useful to find metrics that support or refute the claims made about object-oriented languages. A result of the growth in acceptance of structured programming in the 1970s was the introduction of a number of software measurements, or metrics, intended to express the complexity of a given design or piece of source code. The quest was for a method by which potentially defect-prone software components could be identified a priori. If found this would give developers the opportunity to either redesign defect-prone modules or at least plan additional tests for them. Schneidewind and Hoffman, in a 1979 paper on metrics in a structured programming environment, describe the goal as follows:

The computer industry is searching for ways to produce reliable software and to reduce the cost of software development and maintenance. Structured programming has received wide acclaim as one answer to this search. If it can be shown that structural characteristics of computer programs are related to the difficulty of producing programs with few errors and to the difficulty of detecting errors during debugging and testing, then structural characteristics could be used as measures of program complexity for program design and testing purposes. Program designs with poor structural properties would be avoided because their use would likely result in many programming errors and great difficulty of error detection during debugging and testing. Secondly, after programs have been designed and coded, the complexity measures would be used to index the difficulty of debugging and testing the programs. Resources would be allocated to testing in accordance with the expected difficulty of error detection [2] What is noteworthy about this quote is that if the words “structured programming” were replaced with “object-oriented programming” it would serve as an apt description of the computer industry today. To rephrase Schneidewind and Hoffman, if it can be shown that object-oriented characteristics of computer programs are related to the difficulty of producing programs with few errors and to the difficulty of detecting errors during debugging and testing, then object-oriented characteristics could be

used as measures of program complexity for program design and testing purposes.

**Literature Review: -** The study of software complexity metrics has evolved significantly from structured programming to object-oriented programming paradigms. Early research by Capers Jones emphasized the growing dominance of object-oriented languages and the need for better evaluation techniques to understand their effectiveness.

Traditional complexity metrics such as those proposed by Maurice Halstead and Thomas McCabe focused primarily on control flow complexity. These metrics were effective for procedural programming but failed to capture the unique characteristics of object-oriented systems.

Research by N. F. Schneidewind and H. M. Hoffman highlighted that structural characteristics of programs are directly related to error detection and debugging difficulty. Their work laid the foundation for using metrics to predict software reliability and testing effort.

The concept of type complexity emerged as a key addition to traditional metrics. Unlike control flow complexity, type complexity measures how data structures and class relationships contribute to overall system complexity. This is particularly important because object-oriented languages increase expressiveness through inheritance, polymorphism, and abstraction.

The literature highlights that object-oriented systems often have smaller functions but more complex interrelationships between classes. Therefore, modern metrics must consider: Class structure, Encapsulation level, Reusability, Inter-class dependencies

**Objective**

The purpose of this research is to propose, and then attempt to show empirical evidence in support of, a set of object-oriented metrics that may be used to indicate the complexity of object-oriented programs. Existing, established software complexity metrics (generally referred to as traditional or procedural metrics for simplicity in the following), as used for non-object-oriented languages, are inadequate for expressing all of the complexities of object-oriented programs. Traditional metrics fail at capturing the complexity of object-oriented designs because traditional complexity metrics are heavily weighted toward measuring only control flow complexity. Walsh refers to this by saying that “code complexity is a valid error predictor, but in C++ programs it is overwhelmed by other factors more closely associated with errors.” [3] Typical object-oriented systems contain functions whose sizes are smaller than would be their functional counterparts [4] and measurements of complexity for systems of this type should take into account other considerations. Because object-oriented programming languages represent

evolutionary rather than revolutionary departures from the languages which preceded them, it is not appropriate to abandon all measures of control complexity; therefore, traditional complexity metrics may still be of some use. However, in addition to control complexity object-oriented languages possess what can be referred to as type complexity. Type complexity is that portion of complexity in a system that is attributable to the manner in which data types may be expressed in the system. Because object-oriented languages provide a more expressive set of language constructs for expressing type information than do traditional, procedural languages the type complexity in object-oriented systems will tend to exceed that found in procedural languages. This is, of course, the method through which object-oriented languages are able to reduce control flow complexity. There is a tradeoff between control flow complexity and type complexity. Since control flow complexity can already be measured with existing metrics, the goal of this research will be to identify a set of metrics intended for use with object-oriented systems that can be used to measure the type complexity of those systems. One of the features that makes an object-oriented language object-oriented is the ability of the language to combine the code and data used to model a real-world entity into a single structure. These structures are sometimes known as objects or classes. In this paper, class will refer to a type of structure and object will refer to a specific instance of a class. For example, if Automobile is a class then the 1967 Volkswagen in my garage is an object. Because much of the work in creating an object-oriented design is related to the correct partitioning of a system into the appropriates classes [ [5], [6], [7], [8] ] efforts at calculating type complexity are correctly targeted at the class level. The decision to measure complexity at the class level is based on three contributing factors:

• **Encapsulation:** One of the cited advantages of object-oriented design is the ability to encapsulate implementation details into a class.[7] [8] This occurs because the data and functions necessary to implement the data structure are both contained in the class definition. If everything necessary for implementing a data structure has been encapsulated into a class it seems logical to conclude that whatever complexity is inherent in that data structure has also been encapsulated into the class definition.

• **Code Reuse:** One of the argued advantages of using an object-oriented language is that it facilitates code reuse. Code reuse can occur more naturally and more simply than in non-object-oriented languages because all aspects of a class are tied together in the class concept. Classes cannot be partially reused, it is generally an all or nothing proposition. Because reuse occurs at the class level it will

be useful to have as a goal a metric that can be applied to individual classes.

• **Classes are Types:** Because the primary mechanism for defining a new data type is the definition of a new class, it makes sense that a metric targeted at measuring type complexity should occur at the class level.

3. Justification and Likely Benefits

This research will hopefully provide a foundation upon which the selection of high defect risk, complex classes may begin. Using the metrics identified in this thesis as possessing a statistically significant correlation when compared against discovered defect densities a software manager may be able to adjust code review or test schedules to emphasize those classes that are overly complex. The knowledge of which classes may be particularly prone to defects can then be used by project management to more accurately position the project between the potentially opposite goals of fewer defects and earlier delivery.

**References**

[1] C. Jones, Applied Software Measurement: Assuring Productivity and Quality,
McGraw-Hill, 1991.

[2] N.F. Schneidewind and H-M. Hoffman, "An Experiment in Software Error
Data Collection and Analysis,'' IEEE Transactions on Software Engineering,
Vol SE- 5(3), May 1979, pp. 276-286.

[3] J.F. Walsh, "Preliminary Defect Data from the Iterative Development of a Large
C++ Program,'' Object Oriented Programming Systems, Languages and Applications
(OOPSLA), Vol 11, 1992, pp 178-

[4] R. Kolewe, "Metrics in Object-Oriented Design and Programming,'' Software
Development, Vol. 1, No. 4, October 1993, pp. 53-62.

[5] G. Booch, "Object-Oriented Development,'' IEEE Transactions on Software
Engineering, Vol SE-12(2), February 1986, pp. 211-221.

[6] P. Coad and E. Yourdon, Object-Oriented Analysis, Yourdon Press, 1990

[7] G. Booch, Object-Oriented Design: With Applications, Benjamin/Cummings
Publishing Company, Inc., 1991.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object