Research Article

Don Bosco Institute of Technology Delhi Journal of Research
Year 2025, Volume-2, Issue-2 (Jul-Dec)

Assessing Interface Dependency Complexity in Components based Software Systems (CBSS)

Rachit Kadian

Research Scholar, Amity University, Gurugram

ARTICLE INFO

***Keywords*:** Software Complexity, Software metric, Information flow, Traditional metrics, CBSS

doi: 10.48165/dbitdjr.2025.2.02.03

ABSTRACT

Systems with high complexity typically exhibit a balanced combination of order and disorder, characterized by the coexistence of randomness and regularity. In component-based software engineering, each component contributes to overall system complexity through its inherent constraints, interaction mechanisms with other components, and the degree of customizability it supports. As software systems grow in scale and interconnectivity, understanding and managing such complexity becomes critical to ensuring system reliability, maintainability, and quality. Measuring system dependency and interaction complexity provides valuable insights into potential design weaknesses and helps assess software quality at both component and system levels. Complexity metrics play a crucial role in enabling objective comparison across different software systems by applying standardized measurement frameworks. Existing complexity measures, however, often focus primarily on internal code structure, overlooking the significance of component interfaces that govern communication, data exchange, and functional coordination. The author has pointed out a novel interface complexity metric for software components in the study. Based on the quantity, kind, and structural features of interface methods and attributes that a component exposes, the suggested metric measures complexity. Factors such as method signatures, parameter types, data dependencies, access constraints, and interaction frequency are considered to capture the true interaction burden imposed by component interfaces. By focusing on interface-level complexity, the metric provides a more accurate representation of component interdependencies and their impact on overall system behaviour. The proposed approach supports improved software design evaluation, facilitates early detection of architectural vulnerabilities, and aids developers in enhancing modularity, reusability, and long-term system quality.

Introduction

Software complexity is defined as follows: "A software complexity is a term that encompasses numerous properties of a piece of software, all of which affect internal interactions". One common term used to describe "the relationship between a program and a programmer working on certain programming tasks is software complexity." Each component of a CBS's framework plays a specific purpose in the services provided by the system. The system's overall functionality changes as soon as a new component is added. Shorter delivery deadlines and lower development costs are the advantages of component-based development.

• Lower maintenance expenses and increased dependability

*Corresponding author.
E-mail address: rachitkadian0707@gmail.com

• Allows developers to concentrate on their primary skills and business requirements instead of repeatedly resolving the same technical issues.
• Offers extensibility since parts can be put together in a variety of ways to create distinct system variations as needed. These days, this is particularly prevalent in sectors like consumer electronics, automotive systems, and cellular technologies.
• It is possible to mix and match components that employ various languages and technologies.
• Component-based development is the most effective method for handling system complexity as it grows in size and scope. Higher level models make complicated systems easier to comprehend. Current systems are constantly being updated and modified, which leads to dependency problems. Junk libraries and files may still be present in the system even after special uninstall processes have been carried out to get rid of the problematic application. The primary reason for these issues is the absence of mechanisms for managing dependencies between system and application components as well as a representation model for such dependencies. An essential component of software research for a component-based system's comprehensibility, testability, maintainability, and reusability is the analysis of CBSS dependencies. Dependency metrics may therefore have a significant effect on the system's quality that is provided to the user.

Literature Review

Comparing Component-Based System Metrics with Conventional Software Metrics

Tagen et al., 2025 stated that Component-based software systems (CBSS) have become increasingly popular among academics and professionals. CBSS can improve productivity, quality, and reusability while lowering time-to-market and maintenance costs by constructing software from reusable components. Developers may concentrate on particular features thanks to this modular approach, which results in codebases that are more effective and maintainable. However, because the elements affecting dependability are complicated and frequently only their impacts are visible, assessing a CBSS's reliability presents difficulties. This complexity is a result of the different operational profiles, interactions between the separate components, and their integration into the overall architecture. In order to conform with component-based software development, traditional metrics must be improved or reinterpreted. Since 1976, structured systems' software complexity has been measured using traditional measures. Some common traditional software metrics include the following:

Cyclomatic complexity – To evaluate the structural complexity of software modules, in 1976, McCabe introduced a metrics named "cyclomatic complexity". According to Watson et al., 1996 and Siahaan et al., 2025, a program's control flow graph can be used to calculate the number of linearly independent pathways, which provides a clear indicator of the complexity of decision-making inside a code module.
SLOC - Source lines of code assess a project based on code volume. We use physical SLOC to count code lines, excluding comments and blank lines (Li et al., 2025).
function point analysis (FPA) - FPs gauge a project's size from a functional standpoint. FPs examine how a software interacts with users and are independent of language. Function points include things like data structures, input, user's interaction, output, documents and the software's links to various systems (Wicaksono & Sandaa, 2024).
bugs per lines of code – The term "bugs per lines of code" (sometimes called "defect density") is used to calculate the number of proven problems in relation to the software's size. It is commonly stated as bugs per thousand lines of code (KLOC).
code coverage - A testing statistic called "code coverage" calculates the proportion of the codebase that is run during testing.
Traditional software metrics are usually applicable to small programs, while CBSS measures should be based on the resolution and interoperable characteristics of the components. While most standard size measurements, such as code coverage and source lines of code are dependent on code line. Additionally, component developers typically do not know the size of a component. This do not apply on the software of component-based system.
The conventional cyclomatic complexity metric suite is likewise not relevant in CBSS as the number of linearly independent paths cannot be ascertained and operator and operand counts are unknown. Although certain enhanced measures, such as modifying the counting method, have been used, FPA is dependent on the weights that were generated in a specific setting, raising questions regarding the validity of this method for widespread applicability. The usual software measurements are not relevant for CBSS due to the numerous intrinsic differences between CBSS and non-CBSS. Furthermore, standard software measures do not account for integration-level metrics and interface complexity, both of which are equally irrelevant to CBSS.

Software complexity's Nature

Most software metrics experts concur that the number of resources needed for a project is directly correlated with its complexity. The concept is that a more complex problem or solution calls for additional project resources, including man-hours, computer time, support software, etc. Since a large number of resources are used to find errors, debug, and retest software, the number of defects is a linked indication of complexity. Based on the literature available (Zuse, 1991; Fenton & Pfleeger, 1996; Ohlsson, 1996), we would like to suggest that software complexity consists of the following elements:

1. **Algorithmic Complexity** of the algorithm applied to solve the problem that is reflected in its complexity. This is the application of the concept of efficiency. By empirically testing different algorithms, we can ascertain which approach offers the most efficient solution to the problem and, thus, has the least amount of extra complexity. This kind of complexity can be measured as soon as a solution's algorithm is developed, typically during the design stage. However, algorithmic complexity has historically been measured on code since the mathematical structure is more visible in code.
2. The **problem complexity** (also known as **computational complexity**) quantifies the intricacy of the underlying problem. When the problem is defined at the requirements phase, this kind of complexity can be found. If the problem can be explained by using algorithms and functions, it is also feasible to compare many problems with one another and determine which is the most complex.
3. The algorithm's implementation software's structure is measured by **structural complexity.** Researchers and practitioners have long acknowledged that there might be a connection between a product's quality and its structure. To put it another way, the structure of specifications, the design, and source code may help us understand the difficulties we sometimes encounter when testing a product such as validating requirements or code testing, converting one product to another (e.g., implementing a design as a source code), or forecasting external software attributes from early product measures.
4. C**ognitive complexity** measures the amount of work required to understand the software. It has to do with how a system is seen psychologically or how tough it is to implement or finish. However, as psychological researchers and scientists are particularly interested in investigating this element of complexity, we are not including it in our paper. But it's important to understand that the way people think is a constraint on the creation of software and that it influences the program's attributes that we want to evaluate, including quality and productivity.

Structural Complexity

Algorithmic complexity

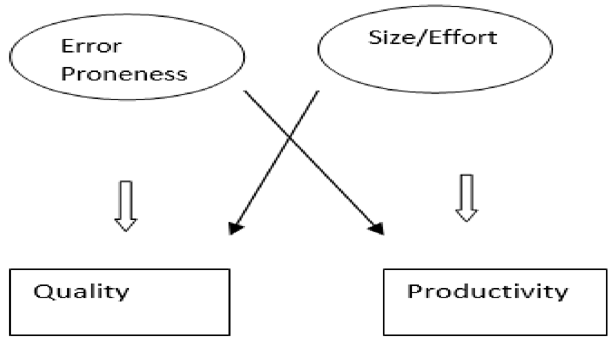

Fig.1 Simplified model of complexity
Source: Author's own

Component's Complexity

Many scholars have defined software complexity differently in the literature. According to IEEE, software complexity is "the degree to which a system or component has a design or implementation that is difficult to understand and verify." Complexity is quantified through the lens of the processing duration and storage required to finish a computation when a computer operates as an interacting system. Numerous researchers have occasionally put forth different measures for assessing, forecasting, and managing software complexity. Since 1976, the software complexity of structured systems has been measured using traditional software metrics.

Halstead's software science metrics, SLOC, McCabe's cyclomatic number, FPA, errors or faults per line of code, and Kafura & Henry's fan-in, fan-out, and code coverage are the most well-known early reported complexity metrics for the traditional function-oriented approach. Bill Curtis distinguishes between mathematical and psychological software complexity. Programmers' attempts to comprehend or modify a class or module are influenced by psychological complexity, whereas algorithmic or computational complexity characterizes an algorithm's run-time performance. Software complexity cannot be determined using just one parameter of a program, component, or piece of software since it is a complicated property of software. The primary factors that increase the complexity of a (CBSS) component-based software system are as follows:

Size of each component: The size of a program, class, or component also affects its complexity.

Since a class with more methods is harder to understand than one with fewer, it adds additional complexity. Large programs or components have issues simply because of the amount of information needed to comprehend them and the additional resources required for their upkeep. Therefore, one aspect that increases a component's complexity is its size.

Interfaces of each component: A component in CBSD has interfaces with other components because it is related to them. If there is a link between two or more components—that is, when one component sends a message and other aspects receive it—they are said to be interfaced. Which component depends on the other or seeks services is indicated by the link's direction. Incoming and outgoing interactions can be used as an interface between two components. A component-based software system is made more complex by these two kinds of interactions.

Interfaces are a component's access points that enable an aspect to reach out for a service that is specified in the service provider's interface. The mathematical definition of I (Component) is the entire complexity of the interface methods in the class. The complexity of interface approaches is based on their nature. The interface methods' nature is defined by their arguments and return types. Arguments and return types can be any of the three previously discussed data types: primitive, structured and user-defined—that were previously covered can be used in arguments and return types. By taking into account the overall number of methods in each category, these methods can be given weight values. The value assigned will likewise rise if there are more methods than the weight. It is now possible to mathematically express the Interface Factor, or I (Component), as follows:

I (Component$_x$) =

$$\sum_{i=1}^{m1} w_{simple_i} + \sum_{j=1}^{m1} w_{medium_j} + \sum_{k=1}^{m1} w_{complex_k}$$

The above algorithm represents that- m1, m2, and m3 stand for the total number of interaction techniques that are simple, medium, and complex, respectively.

An interface complexity metric for a component-based system that solely considers interface complexity is proposed as follows:

AIIC = $\dfrac{\sum_{i=1}^{m} II_i}{m}$

(AIIC - Average Incoming Interactions Complexity)

AOIC = $\dfrac{\sum_{i=1}^{m} OI_i}{m}$

(AOIC - Average Outgoing Interactions Complexity)

Average Interface Complexity of a Component Based System (CBSS)=(AIC(CBS)) = $\dfrac{\sum_{i=1}^{m} II_i}{m} + \dfrac{\sum_{i=1}^{m} OI_i}{m}$

Therefore, ‘M’ is the Component-based System’s (CBS) component count.

Incoming Interactions is represented by ‘II’.

Outing Interactions is what ‘OI’ stands for.

‘I’ is an acronym for index variable.

Conclusion

In order to describe and assess dependence connections between components in CBSS- Component-Based Software Systems, a comprehensive set of metrics was developed in this work. The proposed metrics enable designers to identify critical components that are more prone to errors and assess the impact of changes across the entire system. By quantifying component dependencies, the approach supports informed decision-making regarding corrective modifications and helps improve overall design quality, maintainability, and reliability. The use of a linked list–based approach, along with the concept of a component dependency life cycle, provides a structured and systematic way to represent, understand, and manage inter-component dependencies. This method offers a clear visualization of dependency propagation and facilitates better control over component interactions during system evolution. Given the inherent complexity of large-scale CBSS, manual calculation of dependencies across multiple levels is neither efficient nor practical. Therefore, the need for an automated tool becomes essential to accurately compute dependency metrics at all levels and support real-time design evaluation. Automation not only improves accuracy but also enhances scalability and usability for practitioners.

Implications of the Study

For Component-Based Software Systems (CBSS), this work has significant theoretical, practical, and methodological ramifications. By highlighting component dependency and interface-level interactions as crucial factors influencing system quality, it theoretically advances software complexity research by going beyond conventional code-centric complexity metrics. In order to improve maintainability and overall software quality, software designers and architects can use the suggested dependency metrics to identify crucial and error-prone components, evaluate the impact of changes, and prioritize components that need to be redesigned or refactored. Effective dependency control during system evolution is supported by the linked list-based dependency modelling approach, which offers a manageable and transparent depiction of dependent connections. In terms of methodology, the study provides a common framework for quantifying component dependencies, making it easier to compare CBSS designs objectively and promoting the creation of automated tools for scale dependency analysis.

Future Research Direction

Future research can focus on developing a fully automated dependency analysis tool integrated with modern development environments. Further extensions may include empirical validation of the proposed metrics on large industrial systems, incorporation of dynamic runtime dependencies, and the application of machine learning techniques to predict fault-prone components. Additionally, exploring the interaction between dependency metrics and software quality traits such as versatility, maintainability, and efficiency further strengthen the practical relevance of this work.

References

C. Szyperski, Component Software: Beyond Object Oriented Programming, Second Editioned, Addison Wesley, New York, 2002,

L. Narasimhan and B. Hendradjaya, “Some theoretical considerations for a suite of metrics for the integration of software components,” Information Sciences, vol.177, 2007, pp. 844-64

B. Li, “ Managing dependencies in component-based systems based onmatrix model,” Proc. Proceedings Of Net. Object. Days, Citeseer, 2003, pp.22-25

A. Sharma, P.S. Grover and R. Kumar, "Dependency analysis forcomponent-based software systems," SIGSOFT Softw. Eng. Notes, vol.34, 2009, pp. 1-6

Singh, R., Grover, P.S. (1997): A New Program Weighted Complexity Metric, Proc. International conference on Software Engg. (CONSEG'97), Chennai, India, pp. 33- 39.

Harrison, W. (1982). Magel, K, Kluezny, R., dekock, A.: Applying Software Complexity Metrics to Program Maintenance, IEEE Computer, 15, pp. 65-79.

A. De Lucia, A.R. Fasolino and M. Munro, "Understanding functionbehaviors through program slicing," wpc, 1996, pp. 9.

S. Bates and S. Horwitz, " Incremental program testing using programdependence graphs," Proc. Proceedings of the 20th ACM SIGPLANSIGACT symposium on Principles of programming languages, ACM,1993, pp.384-396

Halstead, M.H. (1977): Elements of Software Science, New York: Elsevier North Holland.

Chidamber, S. R., Kemerer, C.F. (1994): A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, pp. 476-492.

Kafura, D., Henry, S., 1981. Software Quality Metrics, Based on Interconnectivity, Journal of Systems and Software. Vol. 2, pp: 121-131.

Li, B., Zhong, D., Nadendla, D., Terceros, G., Bhandar, P., & Nicholas, C. (2025). MASCOT: Analyzing Malware Evolution Through a Well-Curated Source Code Dataset. *arXiv preprint arXiv:2512.00741*.

Watson, A. H., Wallace, D. R., & McCabe, T. J. (1996). *Structured testing: A testing methodology using the cyclomatic complexity metric* (Vol. 500, No. 235). US Department of Commerce, Technology Administration, National Institute of Standards and Technology.

Siahaan, V., Ginting, H., & Amri, M. (2025). Cyclomatic Complexity and Maintainability in Modern Software: A Systematic Review. *Journal of Data Science, Technology, and Computer Science*, *5*(1), 8-16.

N. E. Fenton and S. L. Pfleeger, "Software Metrics, A Rigorous and Practical Approach," 2nd Edition, International Thomson Computer Press, Boston, 1996.

Zuse, H. (1991). *Software Complexity: Measures and Methods*. Berlin, Boston: De Gruyter. https://doi.org/10.1515/9783110866087

Ohlsson, S. (1996). Learning from performance errors. *Psychological Review, 103*(2), 241–262. https://doi.org/10.1037/0033-295X.103.2.241

Wicaksono, S. R., & Sandaa, I. E. E. (2024). Function Point Analysis for Quality Evaluation of a Natural Resource Information System in Bulungan Regency. In *Proceedings of the National Conference on Electrical Engineering, Informatics, Industrial Technology, and Creative Media* (Vol. 4, No. 1, pp. 1163-1172).

Tagen, I. A., Wassef, K. T., Moawad, R., & Mohammad, S. S. (2025). Software Reliability Estimation of Component-Based Systems. *IEEE Access*.