International Journal of Engineering and Science Invention (IJESI)
ISSN (Online): 2319-6734, ISSN (Print): 2319-6726
www.ijesi.org ||Volume 14 Issue 4 April 2025 || PP 113-116

Extending the Lines of Code Metric for Evaluating Software Quality

Paata Jokhadze[1], Levani Mtivlishvili[1]
[1](Faculty of Informatics and Control Systems, Georgian Technical University, Tbilisi, Georgia)

ABSTRACT: *The Lines of Code (LOC) metric is a traditional and widely used measure of software size, commonly employed to estimate project status and quality. However, the lack of a standardized method for calculating LOC has led to the development of more refined sub-metrics, including Logical Lines of Code (LLOC), Executable Lines of Code (ELOC), Comment Lines of Code (CLOC), Blank Lines of Code (BLOC), and Documented Lines of Code (DLOC). This paper explores each of these metrics in detail. LLOC captures only lines relevant to system functionality, while ELOC focuses solely on executable statements. CLOC and BLOC contribute to code readability and structural clarity. DLOC facilitates structured documentation to enhance code comprehensibility. Examples and counting methodologies for each metric are presented, along with recommended proportional distributions, emphasizing the importance of metric balance in high-quality software development.*
KEYWORDS: *LOC (Lines of Code) LLOC (Logical Lines of Code) ELOC (Executable Lines of Code) CLOC (Commented Lines of Code) BLOC (Blank Lines of Code) DLOC (Documented Lines of Code)*

Date of Submission: 08-04-2025 Date of Acceptance: 19-04-2025

I. Introduction

In software engineering, measuring the size, complexity, and quality of source code is fundamental to project planning, resource estimation, and long-term maintainability. One of the earliest and most widely adopted software metrics is Lines of Code (LOC), which quantifies the number of physical lines present in a software system. For decades, LOC has served as a proxy for estimating developer effort, project cost, and codebase volume. Despite its widespread use, the LOC metric has several limitations. It treats all lines of code equally, regardless of whether they contribute to program logic, represent declarations, or serve as comments and formatting. Moreover, LOC lacks a unified methodology for counting, which often results in inconsistencies across tools and practices. To address these shortcomings, researchers have introduced refined metrics that differentiate between various types of lines within the codebase. Among these, Logical Lines of Code (LLOC) identifies lines that serve a functional purpose, while Executable Lines of Code (ELOC) measures the subset of lines that actively perform operations. Additional metrics such as Comment Lines of Code (CLOC), Blank Lines of Code (BLOC), and Documented Lines of Code (DLOC) focus on code readability, structure, and formal documentation, respectively. These distinctions enable more accurate and meaningful evaluations of software quality, maintainability, and developer productivity. This paper explores each of these sub-metrics in depth, illustrating their counting rules, practical applications, and recommended distributions within a well-structured codebase.

II. Lines of code metrics

LOC (Lines of Code) is a measure of software size and is one of the most important characteristics used as a basic unit for evaluating the status and quality of a software project. LOC is the traditional and most widely used metric for measuring the size of software code. Its long-standing use stems from the fact that LOC directly reflects the volume of work performed. In the early days of software development, the majority of costs were associated with coding, which made LOC one of the most useful indicators of software expenses [1, 9].

However, LOC has had many drawbacks. One major issue was the absence of a clear and methodological guideline that precisely defined what should be considered a line of code. To address this problem, researchers and practitioners attempted to develop counting rules and frameworks. In 2007, researchers from the University of Southern California published a study describing the types of LOC and the rules for counting them [2].

Several types of commands may be used in the process of software development, but among them, only the following are essential for the actual functionality of the software [3]:

Directive - A directive is an instruction in a programming language intended specifically for the compiler or preprocessor. It does not constitute executable code within the program.

Declarator - A declarator is a part of the code that declares a program unit such as a variable, function, class, structure, or other element, by defining its name and type, without specifying its content or logic directly.

Statement - A statement is a part of the code that performs a specific task. In programming languages, a statement represents a command - "do this." A statement can involve assigning a value to a variable, calling a function, starting a loop, checking a condition, or any other action that directly affects the execution flow of the program.
The LOC metric accounts for the total number of physical lines present in the code [4]. It counts both logical and non-logical lines (such as comments and blank lines) [5, 6]. LOC does not indicate which parts of the code are actually used by the system for its functionality. To address this limitation, the Logical Lines of Code (LLOC) metric was introduced. LLOC counts only those lines of code that are intended for the functionality of the software (directives, declarators, and statements). See Table 1.

Table 1. Example of LLOC (Logical Lines of Code) counting

`int max(int a, int b) {`	*1. Function definition*
	Blank line
`    int c;`	*2. Variable declaration*
	Blank line
`    // Determine the maximum number`	*Comment*
`    if (a > b) {`	*3. Conditional statement*
`        c = a;`	*4. Assignment operation*
`    }`	*End of conditional statement*
`    else {`	*5. Conditional statement*
`        c = b;`	*6. Assignment operation*
`    }`	*End of conditional statement*
	Blank line
`    return c;`	*7. return result*
`}`	*End of function*

LLOC directly indicates the size of the software. In the software development process, directives and declarators are necessary however, they do not perform specific tasks (operations). The metric that reflects the number of executable tasks is Executable Lines of Code (ELOC), which counts the number of lines containing executable code (statements) [7, 8, 5].

Table 2. Example of ELOC (Executable Lines of Code) counting

`int max(int a, int b) {`	*Function definition*
	Blank line
`    int c;`	*Variable declaration*
	Blank line
`    // Determine the maximum number`	*Comment*
`    if (a > b) {`	*1. Conditional statement*
`        c = a;`	*2. Assignment operation*
`    }`	*End of conditional statement*
`    else {`	*3. Conditional statement*
`        c = b;`	*4. Assignment operation*
`    }`	*End of conditional statement*
	Blank line
`    return c;`	*5. return result*
`}`	*End of function*

The process of software development is a highly complex and intricate task. To improve the separation and readability of code blocks, it is a common practice to use blank lines within the code. To account for them, the metric Blank Lines of Code (BLOC) has been introduced. Depending on the task, certain code fragments may require the implementation of complex logic, which increases the overall complexity of the code. In such cases, it is considered good practice to place a comment above the code block that describes the execution logic of the code. This helps make the code easier to understand. To count comments, the Comment Lines of Code (CLOC) metric has been introduced. One of the key requirements of high-quality software is that it must be readable and easy to understand. This ensures that during the development process, there will be no need to isolate code fragments or re-establish their logic from scratch. It should be understandable both to the original author and to a new engineer involved in the process, who may need to modify the code in the future. For achieving such clarity, the use of comments and blank lines is a good solution, but not a sufficient one. Comments placed above a function or class declaration do not provide complete (or, if they do, unstructured) information about what operation the function performs, what types and kinds of variables are passed during its invocation, and what type and kind of value it returns as a result. In many programming languages, a documentation methodology has been introduced to address such issues. Documentation makes it possible to describe a function, structure, class, or other elements (depending on the programming language) in a structured format. We will introduce a documentation metric called Documented Lines of Code (DLOC)**.** The methodology for counting its lines is presented in Table 3, while an example of the counting process is shown in Table 5**.**

Table 3. DLOC (Documented Lines of Code) counting rules

Element	Description	Counting Rule
Summary	Summary	Count once per each occurrence.
Param	Input parameter	Count once per each occurrence.
Return	Return value	Count once per each occurrence.
Remark	Additional remarks	Count once per each occurrence.
Example	Code Example	Count once per each occurrence.
Exception	Exception	Count once per each occurrence.
Typeparam	Generic type parameter	Count once per each occurrence.
See	Link to another member	Count once per each occurrence.
Seealso	Additional references	Count once per each occurrence.

Table 4. Comparison of Documentation Tags Across Languages

Element	C# (XML Doc)	C / C ++ (Doxygen)	TS / JS (TSDoc / JSDoc)
Summary	<summary>	@brief	@summary
Param	<param name="x">	@param x	@param x
Return	<returns>	@return / @returns	@returns
Remark	<remarks>	@note	@remark
Example	<example>	@example, @code	@example
Exception	<exception cref="…">	@throws	@throws
Typeparam	<typeparam name="T"	@tparam T	@template T
See	<see cref="…">	@see	@see
Seealso	<seealso>	@see also	@seealso

Table 5. Example of DLOC (Documented Lines of Code) counting

```
/// <summary>
/// Calculates the maximum between two variables              1. Summary
/// </summary>
/// <param name="a">First number</param>                      2. Parameter to be passed
/// <param name="b">Second number</param>                     3. Parameter to be passed
/// <returns>The maximum number</returns>                     4. return value
int max(int a, int b) {
   ...
}
```

Research has shown that in high-quality software, the proportion of documented code should be between 10–20%. Table 6 presents the recommended proportion of each metric within the software code.

Table 6. Recommended proportional share of metrics in software code

Metric	Recommended Share	Comment
BLOC	5-15%	For visual clarity. Depends on style. Necessary, but in moderation.
CLOC	10-15%	General explanations, section descriptions. Too many comments often indicate unclear code.
DLOC	10-20%	Structured documentation for functions, parameters, and return values. Must be present in high-quality code.
ELOC	20-35%	Low-complexity code usually includes more declarations and less logic per file. A high share often points to overly complex logic.
LLOC	30-50%	Defines the functional volume. May include declarators, directives, and ELOC.
LOC	100%	Total number of lines. Composed of all the components listed above.

III. Conclusion

This study presents an expanded framework for software code analysis by refining the traditional Lines of Code (LOC) metric into more meaningful and purpose-driven sub-metrics, including LLOC, ELOC, CLOC, BLOC, and the newly introduced DLOC. The proposed approach enables more precise and multidimensional assessment of code quality, maintainability, and structural clarity. A key advantage of this framework lies in its ability to distinguish between functional code (LLOC, ELOC), visual structure (BLOC), explanatory commentary (CLOC), and structured documentation (DLOC), thus providing a more holistic view of software systems. In particular, the DLOC metric addresses the common gap in assessing documentation quality, offering a standardized way to quantify documentation coverage and ensure long-term comprehensibility of codebases. Despite its contributions, this framework is not without limitations. Differences in documentation standards across programming languages and tools can pose challenges for universal DLOC adoption. Additionally, the manual or tool-assisted counting of metrics may vary in precision depending on the implementation. Nevertheless, the extended LOC model opens avenues for practical applications in software quality auditing, maintainability forecasting, educational settings (e.g., teaching documentation practices), and tool development for static analysis. Future work may explore automation methods for DLOC extraction, integration with IDEs, and empirical validation across large-scale industry projects. By recognizing the limitations of LOC and embracing a richer metric system, software engineers and researchers can foster more maintainable, readable, and robust systems.

References

[1] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, *A SLOC counting standard*, University of Southern California, Center for Systems and Software Engineering, 2007.

[2] L.M. Laird and M.C. Brennan, *Software measurement and estimation* (Hoboken, NJ: John Wiley & Sons, Inc., 2006).

[3] R.E. Park, *Software size measurement: a framework for counting source statements*, Technical Report CMU/SEI-92-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1992.

[4] Á. Beszédes and I. Siket, *Differences in the definition and calculation of the LOC metric in free tools*, University of Szeged, Department of Software Engineering, Szeged, Hungary, 2014.

[5] J. Rosenberg, Some misconceptions about lines of code, *Proc. 4th Int. Software Metrics Symposium*, Washington, DC, IEEE Computer Society, November 1997, 137–142.

[6] L.G. Wallace and S.D. Sheetz, The adoption of software measures: A technology acceptance model (TAM) perspective, *Information & Management*, *51*(2), 2014, 249–259.

[7] T. Gyimóthy, R. Ferenc, and I. Siket, Empirical validation of object-oriented metrics on open-source software for fault prediction, *IEEE Trans. Softw. Eng.*, *31*, October 2005, 897–910.

[8] Aivosto, *Lines of code metrics (LOC)*, Project Metrics Help, n.d. [Online]. Available: https://www.aivosto.com/project/help/pm-loc.html

[9] B. Boehm, C. Abts, and S. Chulani, Software development cost estimation approaches: A survey, *Annals of Software Engineering*, *10*, 2000, 177–205.