A SLOC Counting Standard

Vu Nguyen **Sophia Deeds-Rubin*** **Thomas Tan** **Barry Boehm**

Center for Systems and Software Engineering

University of Southern California

{nguyenvu, deedsrub, thomast, boehm}@usc.edu

* also with Aerospace Corporation

ABSTRACT

Source Lines of Code (SLOC or LOC) is one of the most widely used sizing metric in industry and literature. It is the key input for most of major cost estimation models such as COCOMO, SLIM, and SEER-SEM. Although the SEI and the IEEE have established SLOC definitions and guidelines to standardize counting practice, inconsistency in SLOC measurements still exists in industry and research. This problem causes the incomparability of SLOC metric among organizations and the inaccuracy of cost estimation. This report presents a set of counting standards that defines what and how to count SLOC. Our experience with the development and use of the USC CodeCount™ toolset, a popular utility that automates the SLOC counting process, suggests that this problem can be alleviated by the use of a reasonable and unambiguous counting standard guide and with the support of a configurable counting tool.

1. INTRODUCTION

Size is one of the most important attributes of a software product. It is not only the key indicator of software cost and time but also a base unit to derive other metrics for project status and software quality measurement. According to Boehm et al's survey on cost estimation approaches, size metric is used as an essential input for most of cost estimation models [1]. For example, COCOMO, SLIM, SEER-SEM, and PriceS all use SLOC; Checkpoint and other functionality based models use function points or other functional sizing units as size input.

SLOC is the traditional and the most popular sizing metric. Its long-standing tradition is due to the fact that SLOC is the direct result of programming work. In the early age of software development, most of software cost was spent on programming, and SLOC emerged as the most perceivable indicator of software cost. Unfortunately, SLOC has a number of shortcomings [3]. One significant deficiency is the lack of precise and methodical guideline for determining what SLOC means. To mitigate this problem, researchers and practitioners have attempted to establish counting rules and framework. In his COCOMO book in 1981, Dr. Boehm gave definitions of delivered source instructions (DSI) or delivered source lines of code (DSLOC) to be used as primary size parameter for COCOMO model. IEEE Standard for Software Productivity Metrics also provided definitions and attributes of SLOC size metrics [4]. Park with the Size Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team of the Software Engineering Institute (SEI) at Carnegie Mellon University significantly extended SLOC metrics into a counting framework (hereafter SEI framework) which contains a set of counting definitions and checklists to use as a guideline [5]. The framework's main objective is to provide counting methods that could be used to define a consistent and repeatable SLOC measurement. However, because the framework focuses on what to count rather than on how many to count, it opens the door to a variety of interpretations and ambiguity in the development of counting tools.

In this report, we present a set of counting rules that is adapted in the CodeCount™ toolset developed by USC's Center for Systems and Software Engineering [6]. Our primary goal is to improve the applicability of SLOC metrics by clarifying and standardizing the process of counting SLOC. The source of inconsistency in source code size is primarily attributed to ambiguity in counting logical rather than physical SLOC, hence the standard places emphasis on both what and how many to count for logical SLOC. The SLOC counting standard is an extension of the SEI framework definition checklist set, whereby definitions of terms and classifications are analogous to those defined in the framework, IEEE standard, and COCOMO Source Statement Definition.

Section 2 of this paper discusses the reasons and needs of a detailed code counting standard, Section 3 describes our proposed set of counting rules, and Section 4 offers some discussions and conclusions on the subject.

2. Needs of a Detailed Counting Standard

Determining what and how many to count is a problem that is not unique to software engineering. Albert Einstein once stated "Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted". This statement may resonate with software metric developers who are faced with this dilemma when they try to establish rules for counting SLOC. In almost 40 years of software development we went from a complete non-existence of means to estimate the size, performance, or complexity of software [3] to a multitude of software measures and tools [4]. Capers

Jones, in the October 2004 CrossTalk article, stated that at least 75 commercial software cost estimating tools exist on the market, including COCOMO II, SEER-SEM, SLIM, True-S and CostExpert [7]. There are at least 20 SLOC counting applications, each producing different logical and physical SLOC count results. This peculiarity demonstrates the deficiencies in current methods and techniques, and suggests that better tools are required to satisfy the needs of the software industry. It is particularly true for projects of large magnitude, where cost estimation depends on automatic procedures to generate reasonably accurate predictions.

Conceptually, software is intangible. How can we put a number on an abstract thought or a piece of logic that go into development of a product? One solution to this conundrum is to simply count physical lines of code. They are visible; they have clear beginning and ending points; they are not language-dependent, and therefore could be counted rather easily. On the other hand, this approach could be problematic as not all lines represent logical statements, the focal point of the counting software. Some are used for formatting purposes while others guide the eye of a reader through the code or serve as separators between sections. An alternative solution is to count logical SLOC. This method is less sensitive to format and programming styles, but its imprecise definition has been the source of contention among tool developers.

Product (partial source code)	Physical	**CodeCount™**		**RSM**		**LocMetrics**	
		Logical	Ratio	Logical	Ratio	Logical	Ratio
OpenWbem	14,000	7,100	**1.97**	4,700	**2.98**	6,600	**2.12**
FlightGear	14,000	10,800	**1.30**	7,600	**1.84**	9,900	**1.41**
wxWidgets	50,300	30,700	**1.64**	21,300	**2.36**	27,300	**1.84**

Table 1 SLOC values of large files from three open source products

Despite the enormous contributions by the IEEE, the SEI, and many industry specialists in formalizing and standardizing the counting process, tool developers have not reached consensus on a universal SLOC counting standard.

Table 1 confirms this statement by showing SLOC values of a set of source files from three large open-source products generated by three popular counting applications: CodeCount™[10], RSM[11], and LocMetrics[12]. (The reason we chose those tools is because they support logical SLOC count and are readily available; RSM is a commercial application). Logical SLOC counts produced by each tool for the same product differ significantly: CodeCount™ results are approximately 150% higher than the RSM counts, and are 110% higher than the LocMetrics numbers.

Assuming that all drivers are nominal, COCOMO II effort calculation of logical SLOC values for wxWidgets product is 127 person-months for CodeCount™, 85 for RSM, and 112 for LocMetrics. In other words, the effort estimate of logical SLOC counted by RSM and LocMetrics is respectively 67% and 88% of the CodeCount™ estimate. This variation is significant considering that it is caused by the use of the counting tool alone. The main reason for this divergence is due to the use of different definitions for logical SLOC. For example, CodeCount™ counts each compiler directive as a logical SLOC while LocMetrics does not. Although each tool could have a set of defects that would account for some discrepancies, this prospect alone is not sufficient to produce these variances.

Another observation that is worth noting is that the physical and logical ratios vary considerably from one product to another, attesting to differences in formatting and programming conventions among these projects. The observations made with the products that we selected suggests that converting from physical to logical count using a fixed ratio is not a reasonable approach.

Even though a vast difference in results is likely to be observed when one method is chosen over the other, it is a common practice for researchers and practitioners to simply use generic terms like SLOC or LOC, and to use physical and logical SLOC interchangeably. Some papers cite the SEI framework as the baseline for their SLOC measure, without clarifying the fact that the framework includes both logical and physical SLOC and allows for variations in those counts.

The SEI framework intends to establish precise definitions for the SLOC metrics, but leaves the decision up to the user on how to treat many special and language-specific cases. For example, how many logical SLOC should be counted in the following cases?

```
if (x > 0) {
      printf(“x is a positive number”);
}
```

and

```
if (x > 0) printf(“x is a positive number”);
```

Both blocks perform the same actions, checking the value of variable *x* and printing the statement to the standard output. Intuitively, they should produce the same number of logical SLOC. Depending on the interpretation of the SEI framework guidelines, one may end up with a different count for each case. It is foreseeable that two organizations follow the same SEI framework guidelines for counting the same product but may come up with a different SLOC measurement.

Lack of specific details can result in ambiguity even when the same checklist or reference is being applied in the estimation effort. In his classic *PSP: a Self-Improvement Process for Software Engineers* book, Watts Humphrey provides an example of C++ SLOC counting standard [9]. The standard requires one count for every occurrence of *DO, WHILE*, *{}* or *};*. Strict adherence to the rule will produce three logical SLOC for the following statement.

```
do {
} while(i!=0);
```

Intuitively, one would likely count this block as a single logical SLOC.

The results clearly demonstrate the effects of existing ambiguities in code counting methodology. Our intention is to fill this gap by providing an approach to counting SLOC. Considering the fact that one of the top causes of software project failures is inaccurate estimates of required resources [8], which can result in a waste of billions of dollars, the development of better cost estimation models and techniques is in high demand.

3. A Detailed Code Counting Standard

3.1 What to Count

Although SLOC count is not the sole contributor to cost estimation techniques and only serves as one of many inputs to common productivity assessment tools, it is a good measuring stick for several reasons. SLOC serves as a foundation for a number of metrics that are derived throughout the software development cycle. Given a well defined set of rules, the actual process of SLOC counting can be automated, reducing the time and effort required to produce an estimate. Computed counts can be stored and re-used for future estimates and provide a good foundation for creating a baseline for new projects.

Although there are many source-code-based metrics that are being proposed and used, such as McCabe's Cyclometic Complexity or Halstead's operand and operator, the examination of these metrics is beyond the scope of this paper. Instead, we will concentrate our discussion on what to include and exclude in SLOC counting measures.

The two most popular and accepted SLOC measures are the number of physical and logical lines of code. The common definition of physical SLOC describes them as the lines that do not contain blanks or comments. This count can be viewed as language-independent as it does not take in account syntactic and other variations between multitudes of programming languages. The common definition of logical SLOC specifies the intention to measure statements, which would normally include lines that terminate by a semicolon. Counting logical SLOC is performed independently of the physical format of the statements that are being counted. It means that multiple logical statements could reside on one line, or that one logical statement could span multiple lines. Because of its advantages over physical SLOC, logical SLOC is recommended as a standard size input for COCOMO II cost estimation model [2].

Table 2 below contains a modified portion of the *Definition Checklist for Source Statements Count* from the SEI framework, which lists a set of program elements to be included as contributors to SLOC measures.

Measurement Unit
Source statement type
Executable
Nonexecutable
Declarations
Compiler directives
Comments
Blank lines
How produced
Programmed
Converted with automated translators
Copied or reused without change
Modified
Origin
New work
Previous work: taken or adapted from
A previous version, build, or release
A reuse library (software designed for reuse)

Other software component or library
Usage
In or as part of the primary product
Delivery and Development status
Delivered as source
System tests completed
Functionality
Operative (accessible, in-use code)
Functional (intentional dead code, reactivated for special purposes)
Replications
Master source statements (originals)
Physical replicates of master statements, stored in the master code
Language
Separate totals for each language

Table 2 Program elements included in physical and logical SLOC measures

3.2 How Many To Count

In USC CodeCount™, logical SLOC measures the total number of source statements in the source program. A source statement is considered as a block of code that performs some action at runtime or directs compilers at compile time [2][5]. Statements are classified into three types: executable, declaration, and compiler directive. Executable statements are eventually translated into machine code to cause runtime actions while declaration and compiler directive statements affect compiler's actions.

USC CodeCount™ treats the source statement as an "atomic" and relatively independent unit, at least at source code level. In other words, the statement is considered as the smallest increment of work that a programmer performs at a given unit of time. In that sense, when the programmer constructs a statement, he has to write the statement and its sub-statements completely in order to be compliable; and when he deletes the statement, he has to delete its sub-statements. Thus, simple and compound or structured statements end up with the same number of logical SLOC. For example, the *for* statement, which consists of initialization, condition, and increment statements, is counted as one logical SLOC rather than three (one for each enclosed statement).

The challenge of determining the beginning and the end of each statement complicates the method of counting logical SLOC. This difficulty stems from the syntactic and style differences in programming languages. At first glance it would appear that counting semicolons would solve the problem. That is not so. There is an abundance of popular languages that do not use semicolons or where semicolons are optional (SQL, JavaScript, Unix scripting languages, etc.) In addition, semicolons do not always play a role of a statement terminator. George E. Kalb, the original developer of USC CodeCount™, provided a list of definitions for the "almighty" semicolon; it included the following types: non-literal, terminal, limited terminal, essential, package body, all, and mixtures of the above [6]. Which ones should be counted and which ones should not be?

Using the SEI framework and COCOMO II SLOC counting standards as our baseline, we took a step further and defined "how many of what" for a set of commonly used languages. The effort included identification and grouping of different statement types within the category of executable instructions, defining the delimiters for each statement type, considering other source statements like data declarations and compiler directives. We attempted to provide as much commonality between the languages as possible to ensure consistency. Tables 3 and 4 illustrate the summary of SLOC counting rules for logical lines of code defined for C/C++, Java, and C# programming languages. Information for other common languages is provided in the Appendix. (Each statement is classified into only one type. The order of precedence specifies the priority a line is classified if it contains more than one type).

Measurement Unit	Order of Precedence	Physical SLOC	Logical SLOC
Executable lines			
Statements	1	One per line	Language-specific (see Table 4)
Non-executable lines			
Declaration (Data) lines	2	One per line	One per declaration
Compiler directives	3	One per line	One per directive

Table 3 Physical and Logical SLOC Counting Rules

Structure	Order of Precedence	Logical SLOC Rules
SELECTION STATEMENTS: *if, else if*, else, "?" operator, *try* , *catch, switch*	1	Count once per each occurrence. Nested statements are counted in the similar fashion.

ITERATION STATEMENTS:
 	
For, while, do..while
	2	Count once per each occurrence.
		
		Initialization, condition and increment within the "*for*" construct are not counted. i.e.
		
		`for ( i= 0; i < 5; i++)…`
		
		In addition, any optional expressions within the "*for*" construct are not counted either, e.g.
		
		`for (i = 0, j = 5; i < 5, j > 0; i++, j--)…`
		
		Braces {…} enclosed in iteration statements and semicolon that follows "*while*" in "*do..while*" structure are not counted.

JUMP STATEMENTS: *Return, break, goto, exit, continue, throw*	3	Count once per each occurrence. Labels used with "*goto*" statements are not counted.
EXPRESSION STATEMENTS: Function call, assignment, empty statement	4	Count once per each occurrence. Empty statements do not affect the logic of the program, and usually serve as placeholders or to consume CPU for timing purposes.
STATEMENTS IN GENERAL: Statements ending by a semicolon	5	Count once per each occurrence. Semicolons within "*for*" statement or as stated in the comment section for "*do..while*" statement are not counted.
BLOCK DELIMITERS, BRACES	6	Count once per pair of braces *{..}*, except where a closing brace is followed by a semicolon, i.e. *};*.

Structure	Order of Precedence	Logical SLOC Rules
		Braces used with selection and iteration statements are not counted. Function definition is counted once since it is followed by a set of braces.
COMPILER DIRECTIVE	7	Count once per each occurrence.
DATA DECLARATION	8	Count once per each occurrence. Includes function prototypes, variable declarations, "*typedef*" statements. Keywords like "*struct*", "*class*" do not count.

Table 4 Logical SLOC Counting Rules for C/C++, Java, and C#

4. DISCUSSION AND CONCLUSIONS

Although SLOC is the oldest and widely accepted sizing metric, there is no universal counting standard that enforces a consistency of what and how to count SLOC. There is no consensus on which elements of code to include and on how many they should be counted. One of the reasons of this problem is the difficulty in defining consistent standard across different programming languages. Another reason is the lack of details in SLOC counting definitions.

The lack of uniform counting standard results in the incomparability in the SLOC metrics in industry and in research. Products that are sized in SLOC are not comparable; and therefore the SLOC productivity is meaningful only within a project or an organization that uses the same counting standard. Software estimation models generate misleading results if they are built on a SLOC counting standard different from the one that was used for measuring SLOC size of the project being estimated.

With the popularity of COCOMO and other estimation models using SLOC metrics, SLOC counting tools have become an important part of the estimation process. SLOC tools help to reduce time and efforts in gathering software sizing data. The use of tools improves accuracy and consistency in the data collection task. The USC CodeCount™ is one of the most popular tools of its type, and it is one of the few utilities that support the logical SLOC count. The CodeCount™ consists of a set of tools to count physical and logical SLOC, along with other metrics for the source program. Based on our statistics, approximately 2,500 copies of the CodeCount™ toolset are being downloaded each year. (This number does not include the downloads from the USC Affiliate members who can access the early releases of the toolset through a password-protected website).

We described some problems associated with the current SLOC sizing practice by reviewing the SEI counting framework and conducting experimental analysis on three popular SLOC counting automation tools. Just as the Function Points Analysis requires counting practices manual, SLOC counting method is in great need of the universal SLOC counting guide. To further this goal, we presented a SLOC counting standard that lies at the foundation of the USC CodeCount™ toolset, and which we are eager to share with the user community at large. The wide acceptance of USC CodeCount™ motivates us to suggest its counting definition as a standard for industry and research.

REFERENCES

[1] B. Boehm, C. Abts, S. Chulani, "Software development cost estimation approaches: A survey", *Annals of Software Engineering, 2000*.

[2] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts, "Software Cost Estimation with COCOMO II". Prentice Hall, 2000.

[3] T.C. Jones, "Measuring programming quality and productivity", Tutorial: Programming Productivity: Issues of the Eighties

[4] IEEE, "Standard for Software Productivity Metrics (IEEE Std 1045 1992)". The Institute of Electrical and Electronics Engineers, Inc., 1993.

[5] R.E. Park, "Software Size Measurement: A Framework for Counting Source Statements", Technical Report CMU/SEI-92-TR-20 ESC-TR-92-020, 1992

[6] G. E. Kalb, "Counting Lines of Code, Confusions, Conclusions, and Recommendations", Briefing to the 3rd Annual REVIC User's Group Conference, 1990.

[7] C. Jones. Software Project Management Practices: Failure Versus Success©, CrossTalk, The Journal of Defense Software Engineering, October, 2004.

[8] R. N. Charette. Why Software Fails. IEEE Spectrum On Line, September 2005 issue.

[9] Watts S. Humphrey. PSP: A Self-Improvement Process for Software Engineers (SEI Series in Software Engineering). Addison-Wesley, March 2005.

[10] CodeCount™, USC's Center for Systems and Software Engineering. http://csse.usc.edu

[11] RSM, M Squared Technologies™, http://msquaredtechnologies.com/index.htm

[12] LocMetrics, http://www.locmetrics.com

APPENDIX

This section provides counting rules for three common languages: PERL, JavaScript, and ANSI SQL-92.

Structure	Order of Precedence	Logical SLOC Rules
SELECTION STATEMENTS: *If, elsif*, else, unless, "*?*" operator	1	Count once per each occurrence. Nested statements are counted in the similar fashion.

ITERATION STATEMENTS:
 	
For, foreach, while, do..while, do..until
	2	Count once per each occurrence.
		
		Initialization, condition and increment within the "*for*" construct are not counted. i.e.
		
		`for ( i= 0; i < 5; i++)…`
		
		In addition, any optional expressions within the "*for*" construct are not counted either, e.g.
		
		`for (i = 0, j = 5; i < 5, j > 0; i++, j--)…`
		
		Braces {…} enclosed in iteration statements and semicolon that follows "*while*" in "*do..while*" or "*until*" in "*do..until*" structures are not counted.

JUMP STATEMENTS: *Return, last, next, goto, die, exit, continue*	3	Count once per each occurrence. Labels used with "*goto*" statements are not counted.
EXPRESSION STATEMENTS: Function call, assignment, empty statement	4	Count once per each occurrence. Empty statements do not affect the logic of the program, and usually serve as placeholders or to consume CPU for timing purposes.
STATEMENTS IN GENERAL: Statements ending by a semicolon	5	Count once per each occurrence. Semicolons within "*for*" statement or as stated in the comment section for "*do..while*" statement are

Structure	Order of Precedence	Logical SLOC Rules
		not counted.
BLOCK DELIMITERS, BRACES	6	Count once per pair of braces *{..}*, except where a closing brace is followed by a semicolon, i.e. *};*. Braces used with selection and iteration statements are not counted. Function definition is counted once since it is followed by a set of braces.
COMPILER DIRECTIVE	7	Count once per each occurrence.
DATA DECLARATION	8	Count once per each occurrence.

Table 5 Logical SLOC Counting Rules for PERL

Structure	Order of Precedence	Logical SLOC Rules
SELECTION STATEMENTS: *If, else if*, else, *try , catch, with, switch*	1	Count once per each occurrence. Nested statements are counted in the similar fashion.
ITERATION STATEMENTS: *For, for..in, while, do..while*	2	Count once per each occurrence. Initialization, condition and increment within the "*for*" construct are not counted. i.e. `for ( i= 0; i < 5; i++)…` In addition, any optional expressions within the "*for*" construct are not counted either, e.g. `for (i = 0, j = 5; i < 5, j > 0; i++, j--)…` Braces {…} enclosed in iteration statements and semicolon that follows "*while*" in "*do..while*" structure are not counted.
JUMP STATEMENTS:	3	

Structure	Order of Precedence	Logical SLOC Rules
Break, continue		Count once per each occurrence.
EXPRESSION STATEMENTS: Function call, assignment, empty statement	4	Count once per each occurrence. Empty statements do not affect the logic of the program, and usually serve as placeholders or to consume CPU for timing purposes.
STATEMENTS IN GENERAL: Statements ending by a semicolon	5	Count once per each occurrence. Semicolons within "*for*" statement or as stated in the comment section for "*do..while*" statement are not counted.
BLOCK DELIMITERS, BRACES	6	Count once per pair of braces *{..}*, except where a closing brace is followed by a semicolon, i.e. *};*. Braces used with selection and iteration statements are not counted. Function definition is counted once since it is followed by a set of braces.
DATA DECLARATION	7	Count once per each occurrence. Includes function prototypes, variable declarations.

Table 6 Logical SLOC Counting Rules for JavaScript

Structure	Order of Precedence	Logical SLOC Rules
DATA STATEMENTS: SELECT UPDATE INSERT DELETE ALTER TABLE ALTER USER DECLARE FETCH CLOSE	1	Count once per each occurrence. Nested statements are counted in the similar fashion. Note that a SELECT may contain sub-queries, which each is counted as one logical SLOC.
SCHEMA STATEMENTS: CREATE CREATE TRIGGER CREATE SEQUENCE CREATE INDEX CREATE SYNONYM REPLACE COMMENT TRUNCATE RENAME DROP GRANT REVOKE	2	Count once per each occurrence.
TRANSACTIONAL STATEMENTS: COMMIT ROLLBACK	3	Count once per each occurrence.
CONDITIONAL STATEMENTS: WHERE	4	Count once per each occurrence.

Structure	Order of Precedence	Logical SLOC Rules
GROUP BY ORDER BY HAVING LIMIT JOIN UNION		
DATA DECLARATION	5	Count once per each occurrence. Includes function prototypes, variable declarations, "*typedef*" statements. Keywords like "*struct*", "*class*" do not count.

Table 7 Logical SLOC Counting Rules for ANSI SQL-92