Source lines of code

From Wikipedia, the free encyclopedia

Source lines of code (**SLOC**), also known as **lines of code** (**LOC**), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.

Contents

Measurement methods

Many useful comparisons involve only the order of magnitude of lines of code in a project. Using lines of code to compare a 10,000 line project to a 100,000 line project is far more useful than when comparing a 20,000 line project with a 21,000 line project. While it is debatable exactly how to measure lines of code, discrepancies of an order of magnitude can be clear indicators of software complexity or man hours.

There are two major types of SLOC measures: physical SLOC (LOC) and logical SLOC (LLOC). Specific definitions of these two measures vary, but the most common definition of physical SLOC is a count of lines in the text of the program's source code excluding comment lines.[1]

Logical SLOC attempts to measure the number of executable "statements", but their specific definitions are tied to specific computer languages (one simple logical SLOC measure for C-like programming languages is the number of statement-terminating semicolons). It is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions. However, SLOC measures are often stated without giving their definition, and logical SLOC can often be significantly different from physical SLOC.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i = 0; i < 100; i++) printf("hello"); /* How many lines of code is this? */
```

In this example we have:

- 1 Physical Line of Code (LOC)
- 2 Logical Lines of Code (LLOC) (for statement and printf statement)
- 1 comment line

Depending on the programmer and coding standards, the above "line of code" could be written on many separate lines:

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
    printf("hello");
}
```

In this example we have:

- 4 Physical Lines of Code (LOC): is placing braces work to be estimated?
- 2 Logical Lines of Code (LLOC): what about all the work writing non-statement lines?

- 1 comment line: tools must account for all code and comments regardless of comment placement.

Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert E. Park (while at the Software Engineering Institute) and others developed a framework for defining SLOC values, to enable people to carefully explain and define the SLOC measure used in a project. For example, most software systems reuse code, and determining which (if any) reused code to include is important when reporting a measure.

Origins

At the time that people began using SLOC as a metric, the most commonly used languages, such as FORTRAN and assembly language, were line-oriented languages. These languages were developed at the time when punched cards were the main form of data entry for programming. One punched card usually represented one line of code. It was one discrete object that was easily counted. It was the visible output of the programmer so it made sense to managers to count lines of code as a measurement of a programmer's productivity, even referring to such as "card images". Today, the most commonly used computer languages allow a lot more leeway for formatting. Text lines are no longer limited to 80 or 96 columns, and one line of text no longer necessarily corresponds to one line of code.

Usage of SLOC measures

SLOC measures are somewhat controversial, particularly in the way that they are sometimes misused. Experiments have repeatedly confirmed that effort is highly correlated with SLOC, that is, programs with larger SLOC values take more time to develop. Thus, SLOC can be very effective in estimating effort. However, functionality is less well correlated with SLOC: skilled developers may be able to develop the same functionality with far less code, so one program with fewer SLOC may exhibit more functionality than another similar program. In particular, SLOC is a poor productivity measure of individuals, since a developer can develop only a few lines and yet be far more productive in terms of functionality than a developer who ends up creating more lines (and generally spending more effort). Good developers may merge multiple code modules into a single module, improving the system yet appearing to have negative productivity because they remove code. Also, especially skilled developers tend to be assigned the most difficult tasks, and thus may sometimes appear less "productive" than other developers on a task by

this measure. Furthermore, inexperienced developers often resort to code duplication, which is highly discouraged as it is more bug-prone and costly to maintain, but it results in higher SLOC.

SLOC is particularly ineffective at comparing programs written in different languages unless adjustment factors are applied to normalize languages. Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a "hello world" program written in C, and the same program written in COBOL - a language known for being particularly verbose.

C	COBOL

`# include <stdio.h>`
 	
`int main() {`
`  printf("\nHello world\n");`
`}`
	`identification division.`
	`program-id. hello .`
	`procedure division.`
	`display "hello world"`
	`goback .`
	`end program hello .`

Lines of code: 4 (excluding whitespace)	Lines of code: 6 (excluding whitespace)

Another increasingly common problem in comparing SLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, graphical user interface builders automatically generate all the source code for a graphical control elements simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance. By the same token, a hand-coded custom GUI class could easily be more demanding than a simple device driver; hence the shortcoming of this metric.

There are several cost, schedule, and effort estimation models which use SLOC as an input parameter, including the widely used Constructive Cost Model (COCOMO) series of models by Barry Boehm et al., PRICE Systems True S and Galorath's SEER-SEM. While these models have shown good predictive power, they are only as good as the estimates (particularly the SLOC estimates) fed to them. Many have

advocated the use of function points instead of SLOC as a measure of functionality, but since function points are highly correlated to SLOC (and cannot be automatically measured) this is not a universally held view.

Example

According to Vincent Maraia,[2] the SLOC values for various operating systems in Microsoft's Windows NT product line are as follows:

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4–5[2]
1994	Windows NT 3.5	7–8[2]
1996	Windows NT 4.0	11–12[2]
2000	Windows 2000	more than 29[2]
2001	Windows XP	45[3][4]
2003	Windows Server 2003	50[2]

David A. Wheeler studied the Red Hat distribution of the Linux operating system, and reported that Red Hat Linux version 7.1[5] (released April 2001) contained over 30 million physical SLOC. He also extrapolated that, had it been developed by conventional proprietary means, it would have required about 8,000 person-years of development effort and would have cost over $1 billion (in year 2000 U.S. dollars).

A similar study was later made of Debian GNU/Linux version 2.2 (also known as "Potato"); this operating system was originally released in August 2000. This study found that Debian GNU/Linux 2.2 included over 55 million SLOC, and if developed in a conventional proprietary way would have required 14,005 person-years and cost $1.9 billion USD to develop. Later runs of the tools used report that the following release of Debian had 104 million SLOC, and as of year 2005, the newest release is going to include over 213 million SLOC.

One can find figures of major operating systems (the various Windows versions have been presented in a table above).

Year	Operating System	SLOC (Million)
2000	Debian 2.2	55–59[6][7]
2002	Debian 3.0	104[7]
2005	Debian 3.1	215[7]
2007	Debian 4.0	283[7]
2009	Debian 5.0	324[7]
2012	Debian 7.0	419[8]
2009	OpenSolaris	9.7
	FreeBSD	8.8
2005	Mac OS X 10.4	86[9][n 1]
2001	Linux kernel 2.4.2	2.4[5]
2003	Linux kernel 2.6.0	5.2
2009	Linux kernel 2.6.29	11.0
2009	Linux kernel 2.6.32	12.6[10]
2010	Linux kernel 2.6.35	13.5[11]
2012	Linux kernel 3.6	15.9[12]
2015-06-30	Linux kernel pre-4.2	20.2[13]

Utility

Advantages

1. Scope for Automation of Counting: Since Line of Code is a physical entity; manual counting effort can be easily eliminated by automating the counting process. Small utilities may be developed for counting the LOC in a program. However, a logical code counting utility developed for a specific language cannot be used for other languages due to the syntactical and structural differences among languages. Physical LOC counters, however, have been produced which count dozens of languages.
2. An Intuitive Metric: Line of Code serves as an intuitive metric for measuring the size of software because it can be seen and the effect of it can be visualized. Function points are said to be more of an objective metric which cannot be

imagined as being a physical entity, it exists only in the logical space. This way, LOC comes in handy to express the size of software among programmers with low levels of experience.
3. Ubiquitous Measure: LOC measures have been around since the earliest days of software. As such, it is arguable that more LOC data is available than any other size measure.

Disadvantages

1. Lack of Accountability: Lines of code measure suffers from some fundamental problems. Some think it isn't useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only 30% to 35% of the overall effort.
2. Lack of Cohesion with Functionality: Though experiments have repeatedly confirmed that while effort is highly correlated with LOC, functionality is less well correlated with LOC. That is, skilled developers may be able to develop the same functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. In particular, LOC is a poor productivity measure of individuals, because a developer who develops only a few lines may still be more productive than a developer creating more lines of code - even more: some good refactoring like "extract method" to get rid of redundant code and keep it clean will mostly reduce the lines of code.
3. Adverse Impact on Estimation: Because of the fact presented under point #1, estimates based on lines of code can adversely go wrong, in all possibility.
4. Developer’s Experience: Implementation of a specific logic differs based on the level of experience of the developer. Hence, number of lines of code differs from person to person. An experienced developer may implement certain functionality in fewer lines of code than another developer of relatively less experience does, though they use the same language.
5. Difference in Languages: Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written in a language like COBOL. The number of function points would be exactly the same, but aspects of the application would be different. The lines of code needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different (hours per function point). Unlike Lines of Code, the number of Function Points will remain constant.
6. Advent of GUI Tools: With the advent of GUI-based programming languages and tools such as Visual Basic, programmers can write relatively little code and achieve high levels of functionality. For example, instead of writing a program

to create a window and draw a button, a user with a GUI tool can use drag-and-drop and other mouse operations to place components on a workspace. Code that is automatically generated by a GUI tool is not usually taken into consideration when using LOC methods of measurement. This results in variation between languages; the same task that can be done in a single line of code (or no code at all) in one language may require several lines of code in another.

7. Problems with Multiple Languages: In today's software scenario, software is often developed in more than one language. Very often, a number of languages are employed depending on the complexity and requirements. Tracking and reporting of productivity and defect rates poses a serious problem in this case since defects cannot be attributed to a particular language subsequent to integration of the system. Function Point stands out to be the best measure of size in this case.
8. Lack of Counting Standards: There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. Though organizations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice especially in the face of newer and newer languages being introduced every year.
9. Psychology: A programmer whose productivity is being measured in lines of code will have an incentive to write unnecessarily verbose code. The more management is focusing on lines of code, the more incentive the programmer has to expand his code with unneeded complexity. This is undesirable since increased complexity can lead to increased cost of maintenance and increased effort required for bug fixing.

In the PBS documentary *Triumph of the Nerds*, Microsoft executive Steve Ballmer criticized the use of counting lines of code:

> In IBM there's a religion in software that says you have to count K-LOCs, and a K-LOC is a thousand lines of code. How big a project is it? Oh, it's sort of a 10K-LOC project. This is a 20K-LOCer. And this is 50K-LOCs. And IBM wanted to sort of make it the religion about how we got paid. How much money we made off OS/2, how much they did. How many K-LOCs did you do? And we kept trying to convince them – hey, if we have – a developer's got a good idea and he can get something done in 4K-LOCs instead of 20K-LOCs, should we make less money? Because he's made

something smaller and faster, less K-LOC. K-LOCs, K-LOCs, that's the methodology. Ugh! Anyway, that always makes my back just crinkle up at the thought of the whole thing.

Related terms

- KLOC /ˈkeɪlɒk/ ***KAY**-lok*: 1,000 lines of code
  - KDLOC: 1,000 delivered lines of code
  - KSLOC: 1,000 source lines of code
- MLOC: 1,000,000 lines of code
- GLOC: 1,000,000,000 lines of code

See also

- Software development effort estimation
- Estimation (project management)
- Comparison of development estimation software

Notes

1. Possibly including the whole iLife suite, not just the operating system and usually bundled applications.

References

1. Vu Nguyen; Sophia Deeds-Rubin; Thomas Tan; Barry Boehm (2007), *A SLOC Counting Standard* (http://sunset.usc.edu/csse/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf) (PDF), Center for Systems and Software Engineering , University of Southern California
2. "How Many Lines of Code in Windows?" (http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/) (Scholar search (http://scholar.google.co.uk/scholar?hl=en&lr=&q=intitle%3AHow+Many+Lines+of+Code+in+Windows%3F&as_publication=&as_ylo=&as_yhi=&btnG=Search)). Knowing.NET. December 6, 2005. Retrieved 2010-08-30.
   This in turn cites Vincent Maraia's *The Build Master* as the source of the information.
3. "How Many Lines of Code in Windows XP?" (https://www.facebook.com/windows/posts/155741344475532). Microsoft. January 11, 2011.
4. "A history of Windows" (http://windows.microsoft.com/en-AU/windows/history#T1=era6). Microsoft.

5. David A. Wheeler (2001-06-30). "More Than a Gigabuck: Estimating GNU/Linux's Size" (http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html).
6. González-Barahona, Jesús M., Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. "Counting potatoes: the size of Debian 2.2" (https://web.archive.org/web/20080503001817/http://people.debian.org/~jgb/debian-counting/counting-potatoes/). *debian.org*. Archived from the original (http://people.debian.org/~jgb/debian-counting/counting-potatoes/) on 2008-05-03. Retrieved 2003-08-12.
7. Robles, Gregorio. "Debian Counting" (http://debian-counting.libresoft.es/). Retrieved 2007-02-16.
8. Debian 7.0 was released in May 2013. The number is an estimate published on 2012-02-13, using the code base which would become Debian 7.0, using the same software method as for the data published by David A. Wheeler. James Bromberger. "Debian Wheezy: US$19 Billion. Your price… FREE!" (https://web.archive.org/web/20140223013701/http://blog.james.rcpt.to/2012/02/13/debian-wheezy-us19-billion-your-price-free/). Archived from the original (http://blog.james.rcpt.to/2012/02/13/debian-wheezy-us19-billion-your-price-free/) on 2014-02-23. Retrieved 2014-02-07.
9. Jobs, Steve (August 2006). "Live from WWDC 2006: Steve Jobs Keynote" (http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/). Retrieved 2007-02-16. "86 million lines of source code that was ported to run on an entirely new architecture with zero hiccups."
10. "What's new in Linux 2.6.32" (https://web.archive.org/web/20131219054613/http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-32-872271.html?view=print). Archived from the original on 2013-12-19. Retrieved 2009-12-24.
11. Greg Kroah-Hartman; Jonathan Corbet; Amanda McPherson (April 2012). "Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It" (http://go.linuxfoundation.org/who-writes-linux-2012). The Linux Foundation. Retrieved 2012-04-10.
12. "Summary, Outlook, Statistics - The H Open: News and Features" (https://web.archive.org/web/20131219054847/http://www.h-online.com/open/features/What-s-new-in-Linux-3-6-1714690.html?page=3). Archived from the original on 2013-12-19. Retrieved 2012-10-08.. Retrieved on 2014-05-13.
13. http://heise.de/-2730780

Further reading

- Li, Luo; Herbsleb, Jim; Shaw, Mary (May 2005). *Forecasting Field Defect Rates Using a Combined Time-based and Metric–based Approach a Case Study of OpenBSD (CMU-ISRI-05-125)* (http://reports-archive.adm.cs.cmu.edu/anon/isri2005/CMU-ISRI-05-125.ps). Carnegie-Mellon University.
- McGraw, Gary (March–April 2003). "From the Ground Up: The DIMACS Software Security Workshop" (ftp://dimacs.rutgers.edu/pub/dimacs/Technical

Reports/TechReports/2003/2003-13.ps.gz). *IEEE Security & Privacy*. **1** (2): 59–66. doi:10.1109/MSECP.2003.1193213 (https://doi.org/10.1109%2FMSECP.2003.1193213).
- Park, Robert E.; et al. "Software Size Measurement: A Framework for Counting Source Statements" (http://www.sei.cmu.edu/library/abstracts/reports/92tr020.cfm). *Technical Report CMU/SEI-92-TR-20*.

External links

- Definitions of Practical Source Lines of Code (http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm) Resource Standard Metrics (RSM) defines "effective lines of code" as a realistics code metric independent of programming style.
- Effective Lines of Code eLOC Metrics for popular Open Source Software (http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm) Linux Kernel 2.6.17, Firefox, Apache HTTPD, MySQL, PHP using RSM.
- Wheeler, David A. "SLOCCount" (http://www.dwheeler.com/sloccount). Retrieved 2003-08-12.
- Wheeler, David A. (June 2001). "Counting Source Lines of Code (SLOC)" (http://www.dwheeler.com/sloc). Retrieved 2003-08-12.
- Tanenbaum, Andrew S. *Modern Operating Systems* (2nd ed.). Prentice Hall. ISBN 0-13-092641-8.
- Howard Dahdah (2007-01-24). "Tanenbaum outlines his vision for a grandma-proof OS" (http://www.computerworld.com.au/index.php/id;1942598204;pp;1). Retrieved 2007-01-29.
- C. M. Lott: Metrics collection tools for C and C++ Source Code (http://maultech.com/chrislott/resources/cmetrics/)
- Folklore.org: Macintosh Stories: -2000 Lines Of Code (http://folklore.org/StoryView.py?project=Macintosh&story=Negative_2000_Lines_Of_Code.txt&detail=medium/)

Categories: Software metrics
