A SHORT HISTORY OF THE LINES OF CODE (LOC) METRIC

Version 6.0 December 29, 2012

Abstract

The oldest metric for software projects is that of "lines of code" (LOC). This metric was first introduced circa 1960 and was used for economic, productivity, and quality studies. The economics of software applications were measured using "dollars per LOC." Productivity was measured in terms of "lines of code per time unit." Quality was measured in terms of "defects per KLOC" where "K" was the symbol for 1000 lines of code. The LOC metric was reasonably effective for all three purposes.

As additional higher-level programming languages were created the LOC metric began to encounter problems. LOC metrics were not able to measure non-coding activities such as requirements and design which were becoming increasingly expensive. LOC metrics were not able to measure defects in requirements and design.

These problems became so severe that a controlled study in 1994 that used both LOC metrics and function point metrics for 10 versions of the same application coded in 10 languages reached an alarming conclusion: LOC metrics violated the standard assumptions of economic productivity so severely that using LOC metrics for studies involving more than one programming language comprised professional malpractice.

Capers Jones, Vice President and CTO, Namcook Analytics LLC

Email: Capers.Jones3@Gmail.com

Web: www.Namcook.com

A SHORT HISTORY OF THE LINES OF CODE (LOC) METRIC

INTRODUCTION

It is interesting to consider the history of lines of code (LOC) metrics and some of the problems with LOC metrics that led IBM to develop function point metrics. Following is a brief history of LOC metrics from 1960 through today, with projections to 2020:

Lines of Code Metrics Circa 1960

The oldest metric for software projects is that of "lines of code" (LOC). This metric was first introduced circa 1960 and was used for economic, productivity, and quality studies. The economics of software applications were measured using "dollars per LOC." Productivity was measured in terms of "lines of code per time unit." Quality was measured in terms of "defects per KLOC" where "K" was the symbol for 1000 lines of code. The LOC metric was reasonably effective for all three purposes.

When the LOC metric was first introduced there was only one widely used programming language and that was basic assembly language. Programs were small and coding effort comprised about 90% of the total work. Physical lines and logical statements were the same thing for basic assembly language.

In this early environment, LOC metrics were useful for both economic, productivity, and quality analyses. Unfortunately as the software industry changed, the LOC metric did not change and so became less and less useful until by about 1980 it had become extremely harmful without very many people realizing it.

The advent of COBOL, FORTRAN, ALGOL, APL, and other languages would soon lower coding effort. Larger applications would expand requirements and design effort. LOC began to lose accuracy.

Lines of Code Metrics Circa 1970

By 1970 basic assembly had been supplanted by macro-assembly. The first generation of higher-level programming languages such as COBOL, FORTRAN, and PL/I were rapidly expanding in use. Basic assembly language was beginning to drop out of use as many better alternatives became available. Basic assembly was perhaps the first instance of a long series of programming languages that died out, leaving a train of aging legacy applications that would be difficult to maintain as programmers, assemblers, and compilers stopped being available.

The first known problem with LOC metrics was in 1970 when many IBM publication groups exceeded their budgets for that year. It was discovered (by the author) that technical publication group budgets had been based on 10% of the budget for programming. The publication projects based on assembly language did not overrun their budgets, but manuals for the projects coded in PL/S (a derivative of PL/I) had major overruns. This was because PL/S reduced coding effort by half, but the technical manuals were as big as ever.

The initial solution to this problem was to give a formal mathematical definition to language "levels." The "level" was defined as the number of statements in basic assembly language needed to equal the functionality of 1 statement in a higher-level language. Thus COBOL was a "level 3" language because it took 3 basic assembly statements to equal 1 COBOL statement. Using the same rule, SMALLTALK is a level 18 language.

For several years before function points were invented, IBM used "equivalent assembly statements" as the basis for estimating non-code work such as user manuals, requirements, and design. Thus instead of basing a publication budget on 10% of the effort for writing a program in PL/S, the budget would be based on 10% of the effort if the code were written in basic assembly language. This method was crude but reasonably effective. However neither customers nor IBM executives were comfortable with the need to convert the sizes of modern languages into the size of an antique language. Therefore a better form of metric was felt to be necessary.

The documentation problem plus dissatisfaction with the "equivalent assembler" method were two of the reasons IBM assigned Allan Albrecht and his colleagues to develop function point metrics. Additional programming languages such as APL were starting appear, and IBM wanted both a metric and estimating methods that could deal with non-coding work as well as coding in an accurate fashion.

The use of macro-assembly language had introduced reuse, and this caused measurement problems too. It raised the issue of how to count reused code in software applications or any other reused material. The solution here was to separate productivity into two topics: 1) development productivity; 2) delivery productivity.

The former or d*evelopment productivity* dealt with the code and materials that had to be constructed from scratch.

The latter or *delivery productivity* dealt with the final application as delivered, including reused material.

For example using macro-assembly language a productivity rate for *development productivity* might be 300 lines of code per month. But to due to reusing code in the form of macro expansions, *delivery productivity* might be as high as 750 lines of code per month for the same application.

This is an important business distinction that is not well understood even in 2013. The true goal of software engineering is to improve the rate of delivery productivity and not development productivity. To be successful reused code needs to approach or achieve zero-defect status. It does not matter what the development speed is, if once completed the code can then be used in hundreds of applications.

As "service-oriented architecture" (SOA), "software as a service" (SaaS), and the "mash up" method begin to be widely used, their goal is to make dramatic improvements in the ability to ***deliver*** software features. Development speed is comparatively unimportant so long as quality approaches zero-defect levels.

Another issue shared between macro-assembly language and other new languages was the difference between physical lines of code and logical statements. Some languages, such as Basic, allowed multiple statements to be placed on a physical line. Other languages, such as COBOL, divided some logical statements into multiple physical lines. The difference between a count of physical lines and a count of

logical statements could differ by as much as 500%. For some languages there would be more physical lines than logical statements, but for other languages the reverse was true. This problem was never fully resolved by LOC users and remains troublesome even in 2013.

Due to the increasing power and sophistication of high-level programming languages, the percentage of project effort devoted to coding was dropping from 90% down to about 50%. As coding effort declined, LOC metrics were no longer effective for economic, productivity, or quality studies.

After function point metrics were developed circa 1975 the definition of "language level" was expanded to include the number of logical code statements equivalent to 1 function point. COBOL, for example requires about 105 statements per function point in the procedure and data divisions. This expansion is the mathematical basis for "backfiring" or direct conversion from source code to function points. Of course individual programming styles make backfiring a method with poor accuracy even though it remains widely used for legacy applications where code exists but specifications may be missing.

There are tables available from several consulting companies such as David Consulting, Gartner Group, and Software Productivity Research (SPR) that provide values for source code statements per function point for hundreds of programming languages.

Lines of Code Metrics Circa 1980

By about 1980 the number of programming languages had topped 50 and object-oriented languages were rapidly evolving. As a result, software reusability was increasing rapidly.

Another issue that surfaced circa 1980 was the fact that many applications were starting to use more than one programming language, such as COBOL and SQL. The trend for using multiple languages in the same application has become the norm rather than the exception. However the difficulty of counting lines of code with accuracy was increased when multiple languages were used.

In the middle of this decade the first commercial software cost estimating tool based on function points had reached the market, SPQR/20 which was released in 1984. This tool supported estimates for 30 common programming languages and also could be used for combinations of more than one programming language. This tool also included sizing and estimating of paper documents such as requirements, design, and user manuals. It also estimated non-coding tasks including testing and project management.

Because LOC metrics were still used, the SPQR/20 tool expressed productivity and quality results using both function points and LOC metrics. Because it was easy to switch from one language to another, it was interesting to compare the results using both function point and LOC metrics when changing from Macro-assembly to Fortran or Ada or PL/I or Java.

As the level of a programming language goes up, productivity expressed in terms of function points per staff month also go up, which matches standard economics. But as language levels get higher, productivity expressed in terms of "lines of code per month" drop down. This reversal by LOC metrics violates all rules of standard economics, and is a key reason for asserting that LOC metrics constitute professional malpractice.

It is a well-known law of manufacturing economics that when a development cycle includes a high percentage of fixed costs, and there is a decline in the number of units manufactured, the cost per unit will go up. This has been understood by most manufacturing industries since before 1700.

If a "line of code" is considered to be a manufacturing unit and there is a switch from a low-level language to a high-level language, the number of "units" will decline. But the paper documents in the form of requirements, specifications, and user documents do not decline. Instead they stay almost constant and have the economic effect of fixed costs. This of course will raise the cost per unit. Because this situation is poorly understood an example will clarify the situation.

In Case A suppose we have an application that consists of 1,000 lines of code in basic assembly language. (We can also assume that the application is 5 function points in size.) Assume the development personnel are paid at a rate of $5,000 per staff month.

Suppose that coding took 1 staff month and production of paper documents in the form of requirements, specifications, and user manuals also took 1 staff month. The total project took 2 staff months and cost $10,000. Productivity expressed as LOC per staff month is 500. The cost per LOC is $10.00. Productivity expressed in terms of function points per staff month is 2.5. The cost per function point is $2,000.

In Case B assume that we are doing the same application using the Java programming language. Instead of 1,000 lines of code the Java version only requires 200 lines of code. The function point total stays the same at 5 function points. Development personnel are also paid at the same at a rate of $5000 per staff month.

In Case B suppose that coding took only 1 staff week, but the production of paper documents remained constant at 1 staff month. Now the entire project took only 1.25 staff months instead of 2 staff months. The cost was only $6,250 with Java instead of $10,000 with Assembly. Clearly economic productivity has improved since we did the same job as Case A with a savings of $3,750.

But when we measure productivity for the entire project using LOC metrics our rate has dropped down to only 160 LOC per month. Our cost per LOC has soared up to $31.25 per LOC. Using the LOC metric the assembly version looks better than the Java version, and this is a reversal of the real economic productivity of the two cases.

Obviously LOC metrics cannot measure true economic productivity. Also obviously LOC metrics penalize high-level languages. In fact other studies have proven that the penalty exacted by LOC metrics is directly proportional to the level of the programming language, with the highest-level languages looking the worst!

Since the function point total of both Case A and Case B versions is the same at 5 function points, Case B has a productivity rate of 4 function points per staff month. The cost per function point is only $1,250. These improvements when measured with function points match the rules of standard economics, because the faster and cheaper version has better results than the slower more expensive version.

What has happened of course is that the paperwork portion of the project did not decline even though the code portion declined substantially. This is why LOC metrics are professional malpractice if used to

compare projects that used different programming languages. They move in the opposite direction from standard economic productivity rates and penalize high-level languages. Table 1 summarizes both Case A and Case B:

Table 1: Comparing Low-Level and High-Level Languages

	Case A	**Case B**	**Difference**
Language	Assembly	Java	
Lines of Code (LOC)	1000	200	-800
Function Points	5.00	5.00	0
Monthly Compensation	$5,000.00	$5,000.00	$0.00
Paperwork Effort (months)	1.00	1.00	0
Coding Effort (months)	1.00	0.25	-0.75
Total Effort (months)	2.00	1.25	-0.75
Project Cost	$10,000.00	$6,250.00	-$3,750.00
LOC per month	500	160	-340
Cost per LOC	$10.00	$31.25	$21.25
Function points per month	2.50	4.00	1.5
Cost per Function Point	$2,000.00	$1,250.00	-$750.00

As can be seen by looking at Cases A and B, LOC metrics actually reverse the terms of the economic equation and make the large, slow, costly version look better than the small, quick, cheap version.

It might be said that the reversal of productivity with LOC metrics is because paperwork was aggregated with coding. But even when only coding by itself is measured, LOC metrics still violate standard economic assumptions.

The 1000 LOC of assembly code was done in 1 month at a rate of 1000 LOC per month. The pure coding cost was $5,000 or $5.00 per LOC.

The 200 LOC of Java code was done in 1 week, or 0.25 months. Converted into a monthly rate that is only 800 LOC per month. The coding cost for Java was $1,250 so the cost per LOC was $6.25.

Thus Java costs more per LOC than Assembly even though Java took only one fourth the time and one fourth the cost! When you try and measure the two different languages using LOC, assembly looks better

than Java which is definitely a bad conclusion. Table 2 shows the comparison between Assembly and Java for coding only.

Table 2: Comparing Coding Only for Low-Level and High-Level Languages

	Case A	Case B	Difference
Language	Assembly	Java	
Lines of Code (LOC)	1000	200	-800
Function Points	5.00	5.00	0
Monthly Compensation	$5,000.00	$5,000.00	$0.00
Coding Effort (months)	1.00	0.25	-0.75
Coding Cost	$5,000.00	$1,250.00	-$3,750.00
LOC per month	1000	800	-200
Cost per LOC	$5.00	$6.25	$1.25
Function points per month	5	20	15
Cost per Function Point	$1,000.00	$250.00	-$750.00

In real economic terms, the Java code only cost $1,250 while the assembly code cost $5,000. Obviously Java has better economics because the same job was done for a savings of $3,750.

But the Java LOC production rate is lower than assembly, and the cost per LOC has jumped from $5.00 to $6.25!

Unfortunately LOC metrics end up as professional malpractice no matter how you use them if you are trying to measure economic productivity. By contrast, the Java code's cost per function point was $250 while the assembly code's cost per function point was $1,000.

Function point production for Java was 20 function points per staff month versus only 5 function points per staff month for Assembly. Thus function points match the assumptions of standard economics while LOC metrics violate standard economics.

Returning to the main thread, within a few years all other commercial software estimating tools would also support function point metrics, so that CHECKPOINT, COCOMO, KNOWLEDGEPLAN, PRICE-S, SEER, SLIM, Software Risk Master (SRM), SPQR/20 and others could express estimates in terms of both function points and LOC metrics.

By the end of this decade coding effort was below 35% of total project effort, and LOC was no longer valid for either economic or quality studies. LOC metrics could not quantify requirements and design

defects, which now outnumbered coding defects. LOC metrics could not be used to measure any of the non-coding activities such as requirements, design, documentation, or project management.

The response of the LOC users to these problems was unfortunate: they merely stopped measuring anything but code production and coding defects. The bulk of all published reports based on LOC metrics cover less than 35% of development effort and less than 25% of defects, with almost no data being published on requirements and design defects, rates of requirements creep, design costs, and other modern problems.

Lines of Code Metrics Circa 1990

By about 1990 not only were there more than 500 programming languages in use, but some applications were written in 12 to 15 different languages. There were no international standards for counting code, and many variations were used sometimes without being defined.

In 1991 the first edition of the author's book Applied Software Measurement included a proposed draft standard for counting lines of code based on counting logical statements. One year later Bob Park from the Software Engineering Institute (SEI) also published a proposed draft standard, only based on counting physical lines.

It is interesting that the Park book did not cite many of the existing books on measurement, including Applied Software Measurement. A survey of software journals in 1993 found that about one third of published articles used physical lines, one third used logical statements, and the remaining third used LOC metrics without even bothering to say how they were counted. Since there is about a 500% variance between physical LOC and logical statements for many languages, this was not a good situation.

The technical journals that deal with medical practice and engineering often devote as much as 50% of the text to explaining and defining the measurement methods used to derive the results. The software engineering journals, on the other hand, often fail to define the measurement methods at all.

The software journals seldom devote more than a few lines of text to explaining the nature of the measurements used for the results. This is one of several reasons why the term "software engineering" is something of an oxymoron. In fact it is not even legal to use the term "software engineering" in some states such as Texas and in some countries because software development is not a recognized engineering discipline nor a licensed engineering discipline.

But there was a worse problem than ambiguity in counting lines of code. The arrival of Visual Basic introduced a class of programming languages where counting lines of code was not even possible!

This is because a lot of Visual Basic "programming" was not done with procedural code but rather with buttons and pull-down menus. Of the approximate 2,500 programming languages and dialects in existence in 2013, there are only published counting rules for about 50. About another 500 are similar to other languages and could share the same counting rules.

But for at least 50 languages that use graphics or visual means to augment procedural code, including the whole family of Microsoft "visual" languages there are no reliable code counting rules at all.

Unfortunately some of the languages without code counting rules tend to be most recent languages that are used for web site development.

In the middle of this decade a controlled study was done that used both LOC metrics and function points for 10 versions of the same application written in 10 different programming languages including four object-oriented languages. This study was published in American Programmer in 1994.

This study found that LOC metrics violated the basic concepts of economic productivity and penalized high-level and OO languages due to the fixed costs of requirements, design, and other non-coding activities. This was the first published study to state that LOC metrics constituted professional malpractice if used for economic studies where more than one programming language was involved.

By the 1990'a most consulting studies that collected benchmark and baseline data used function points. There are no large-scale benchmarks based on LOC metrics. The International Software Benchmark Standards Group (ISBSG) was formed in 1997 and only publishes data in function point form. Consulting companies such as SPR and the David Consulting Group also use function point metrics.

By the end of the decade, some projects were spending less than 20% of the total effort on coding, so LOC metrics could not be used for the 80% of effort outside the coding domain. The LOC users remained blindly indifferent to these problems, and continued to measure only coding, while ignoring the overall economics of complete development cycles that include requirements, analysis, design, user documentation, project management, and many other non-coding tasks. By the end of the decade non-coding defects in requirements and design outnumbered coding defects almost 2 to 1. But since non-code defects could not be measured with LOC metrics the LOC literature simply ignores them.

Lines of Code Metrics Circa 2000

By the end of the century the number of programming languages had topped 2,000 and continued to grow at more than 1 new programming language per month. Web applications are mushrooming, smart phone applications are mushrooming, cloud applications are and all of these are based on very high-level programming languages and substantial reuse.

The Agile methods are also mushrooming, and also tend to use high-level programming languages. Software reuse in some applications now tops 80%. LOC metrics cannot be used for most web applications, smart phone aps, cloud applications, and are certainly not useful for measuring Scrum sessions and other non-coding activities that are part of Agile projects.

Unfortunately the Agile world has developed a few special metrics of its own such as "story points" based on user stories. However there are no international standards for counting user stories and the magnitude of individual user stories varies by more than 4 to 1 from other stories for the same application.

Function point metrics have become the dominant metric for serious economic and quality studies. But two new problems have appeared that have kept function point metrics from actually becoming the industry standard for both economic and quality studies.

The first problem is the fact that some software applications are now so large (>300,000 function points) that normal function point analysis is too slow and too expensive to be used. There are gaps at both ends

of normal function point analysis. Above 15,000 function points the costs and schedule for counting function point metrics become so high that large projects are almost never counted. (Function point analysis operates between 400 and 600 function points per day per counter. The approximate cost is about $6.00 per function point counted.)

At the low end of the scale, the counting rules for function points do not operate below a size of about 15 function points. Thus small changes and bug repairs cannot be counted. Individually such changes may be as small as 1/50th of a function point and are rarely larger than 10 function points. But large companies can make 30,000 or more changes per year, with a total size that can top 100,000 function points.

The second problem is that the success of function points has triggered an explosion of function point "clones." As of 2013 there are at least 24 function point variations. This makes benchmark and baseline studies difficult, because there are very few conversion rules from one variation to another. In addition to standard IFPUG function points there are also Mark II function points, COSMIC function points, Finnish function points, Netherlands function points, story points, feature points, web-object points, and many others.

Earlier this month in October of 2011 the International Function Point Users Group (IFPUG) released new counting guidelines for non-functional requirements called SNAP. Because this metric is only a few months old as this is written, there are no solid rules for counting SNAP points. IFPUG has also stated that SNAP points will not be additive to normal function points.

The new SNAP points will require changes in all commercial software estimation tools and probable changes to historical benchmarks such as those published by the International Software Benchmark Standards Group (ISBSG) as well as all other benchmarks cited in the author's Catalog of Software Benchmark Sources.

Although LOC metrics continue to be used, they continue to have such major errors that they constitute professional malpractice for economic and quality studies where more than one language is involved, or where non-coding issues are significant.

There is also a psychological problem. LOC usage tends to fixate attention on coding and make the other kinds of software work invisible. For large software projects there may be many more non-code workers than programmers. There will be architects, designers, data base administrators, quality assurance, technical writers, project managers, and many other occupations. But since none of these can be measured using LOC metrics, the LOC literature ignores them.

Lines of Code Metrics Circa 2010

As of 2010 the software industry had more than 2,500 programming languages of which about 2.400 were already obsolete or becoming dead languages. The industry had more than 20 variations for counting lines of code, more than 25 variations for counting function points, and probably another 20 unreliable metrics such as "cost per defect" or percentages of unknown numbers. (The software industry loves to make claims such as "improve productivity by 10 to 1" without defining either the starting or the ending point.)

Future generations of sociologists will no doubt be interested in why the software industry spends so much energy on creating variations of things, and so little energy on fundamental issues.

For the foreseeable future no doubt large projects will still be cancelled, litigation for failures will still be common, software quality will still be bad, software productivity will remain low, security flaws will be alarming, and the software literature will continue to offer unsupported claims without actually presenting quantified data.

What the software industry needs is actually fairly straightforward: 1) measures of defect potentials from all sources expressed in terms of function points; 2) measures of defect removal efficiency levels for all forms of inspection and testing; 3) activity-based productivity benchmarks from requirements through delivery and then for maintenance and customer support from delivery to retirement using function points; 4) certified sources of reusable material near the zero-defect level; 5) much improved security methods to guard against viruses, spyware, and hacking; 6) licenses and board-certification for software engineering specialties. But until measurement becomes both accurate and cost-effective, none of these are likely to occur. An occupation that will not measure its own performance with accuracy is not a true profession.

Lines of Code Circa 2020

If we look forward to 2020, which is only a little more than seven years away, there are best-case and worst-case scenarios to consider.

The *best-case scenario* for lines of code metrics is that usage diminishes even faster than it has been and that economic productivity becomes the industry focus rather than lines of code. For this scenario to occur the speed of function point analysis needs to increase and the cost per function point counted needs to decrease from about $6.00 per function point counted to less than $0.10 per function point counted, which is technically possible.

(Note: in 2012 the author filed a U.S. utility patent application on a method of early sizing of software applications. An earlier provisional patent # 61434091 was converted to a utility patent application. This patent application is embedded in the author's Software Risk Master ™ tool (SRM). The SRM tool can produce size estimates using both function points and logical code statements in about five minutes or less per application. A working version is on the author's web site, www.namcook.com. It can be used by requesting a password from within the web site.)

If these changes occur, then function point usage will increase at least 10-fold and many interesting new kinds of economic studies can be carried out. Among these will be measurement of entire portfolios which might top 10,000,000 function points in size.

Corporate backlogs can be sized and prioritized. Risk/value analyses for major software applications could become both routine and professionally competent. It will also be possible to do economic analyses of interesting new technologies such as the Agile methods, service-oriented architecture (SOA), software as a service (SaaS), and of course total cost of ownership (TCO).

Under the best-case scenario, software engineering would evolve from a craft or art form into a true engineering discipline. Reliable measures of all activities and tasks will lead to greater success rates on

large software applications. The goal of software engineering should be to become a true engineering discipline with recognized specialties, board certification, and accurate information on productivity, quality, and costs.

But that cannot be accomplished when project failures outnumber successes for large applications. So long as quality and productivity are ambiguous and uncertain, it is difficult to carry out multiple regression studies and select really effective tools and methods. LOC metrics have been a major barrier to economic and quality studies for software.

Also under the best-case scenario empirical data will become available for SNAP non-functional costs.

Even more important, the software community will adopt generally accepted accounting principles (GAAP) and international financial reporting standards (IFRS) so that all software projects are measured and reported in a consistent fashion.

The ***worst-case scenario*** is that LOC metrics continue at about the same level as 2013. The software industry will continue to ignore economic productivity and remain fixated on the illusory "lines of code per month" metric.

Under the worst-case scenario, "software engineering" will remain an oxymoron. Trial and error methods will continue to dominate, in part because effective tools and methodologies cannot even be studied using LOC metrics. Under the worst-case scenario failures and project disasters will remain common for large software applications.

Under the worst-case scenario the software industry will not adopt generally accepted accounting practices (GAAP) or international financial reporting standards (IFRS). Productivity data will continue to be only about 37% complete and quality data only about 38% complete.

Function point analysis will continue to serve an important role for economic studies, benchmarks, and baselines but only for about 15% of software applications of small to medium size.

The cost per function point will remain so high that usage above 15,000 function points will continue to be very rare. There will probably be even more function point variations, and the chronic lack of conversion rules from one variation to another will make large-scale international economic studies almost impossible.

Function point metrics remain the metric of choice for economic studies, but the plethora of function point variations are degrading the value of function points.

It is astonishing that the function point community spends so much time on minor variations of counting rules but ignores other critical topics such as the need to measure data quality, the need to speed up counting, the need for micro function points for small changes, and a host of other related measurement topics that have essentially been ignored.

SUMMARY AND CONCLUSIONS

The history of lines of code metrics is a cautionary tale for all people who work in software. The LOC metric started out well and was fairly effective when there was only one programming language and coding was so difficult it constituted 90% of the total effort for putting software on a computer.

But the software industry began to develop hundreds and then thousands of programming languages. Applications started to use multiple programming languages and that remains the norm today.

Applications grew from less than 1,000 lines of code up to more than 10,000,000 lines of code. Coding is the major task for small applications, but for large systems the work shifts to defect removal and production of paper documents in the forms of requirements, specifications, user manuals, test plans, and many others.

The LOC metric was not able to keep pace with either change. It does not work well when there is ambiguity in counting code, which always occurs with high-level languages and multiple languages in the same application. It does not work well for large systems where coding is only a small fraction of the total effort.

As a result LOC metrics became less and less useful until sometime around 1985 they started to become actually harmful. Given the errors and misunderstandings LOC metrics bring to economic, productivity, and quality studies it is fair to say that in many situations usage of LOC metrics can be viewed as professional malpractice if more than one programming language is part of the study or the study seeks to measure real economic productivity.

The final point is that continued usage of LOC metrics is a significant barrier that is delaying the progress of "software engineering" from a craft to a true engineering discipline. An occupation that cannot even measure its own work with accuracy is hardly qualified to be called "engineering."

READINGS AND REFERENCES ON METRICS AND FUNCTION POINT ANALYSIS

1. Boehm, Barry Dr.; Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981; 900 pages.

2. Brooks, Fred: The Mythical Man-Month, Addison-Wesley, Reading, Mass., 1974, rev. 1995.

3. DeMarco, Tom; Why Does Software Cost So Much?; Dorset House, New York, NY; ISBN 0-9932633-34-X; 1995; 237 pages.

4. Fleming, Quentin W. & Koppelman, Joel M.; Earned Value Project Management; 2nd edition; Project Management Institute, NY; ISBN 10 1880410273; 2000; 212 pages.

5. Gack, Gary; Managing the Black Hole; The Software Executives Guide to Software Project Risk; Business Expert Publishing, 2010.

6. Galorath, Daniel D. & Evans, Michael W.; Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves; Auerbach, Philadelphia, AP; ISBN 10-0849335930; 2006; 576 pages.

7. Garmus, David & Herron, David; Function Point Analysis; Addison Wesley, Boston, MA; ISBN 0-201069944-3; 363 pages; 2001.

8. Garmus, David & Herron, David; Measuring the Software Process: A Practical Guide to Functional Measurement; Prentice Hall, Englewood Cliffs, NJ; 1995.

9. Harris, Michael D., Herron, David, and Iwanicki, Stasia; The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results; CRC Press, Boca Raton, FL; ISBN 978-14200-6474-2; 2008; 266 pages.

10. Jones, Capers: A Catalog of Software Benchmark Sources; version 11 with 15 benchmark groups and 68,100 benchmark projects; available from the author upon request.

11. Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, 2011.

12. Jones; Capers; Software Engineering Best Practices; McGraw Hill, 2010.

13. Jones, Capers; Program Quality and Programmer Productivity; IBM Technical Report TR 02.764, IBM San Jose, CA; January 1977.

14. Jones, Capers; Sizing Up Software; Scientific American Magazine; New York NY; Dec. 1998, Vol. 279 No. 6; December 1998; pp 104-109.

15. Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978-0-07-150244-3; 575 pages; 3rd edition (March 2008).

16. Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA, 2000; 659 pages.

17. Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Version 6; Software Productivity Research, Burlington, MA; June 2006; 54 pages.

18. Jones, Capers; Estimating Software Costs; McGraw Hill, New York; 2nd edition, 2007; 644 pages; ISBN13: 978- 0-07-148300-1.

19. Jones, Capers; "The Economics of Object-Oriented Software;" American Programmer Magazine, October 1994; pages 29-35.

20. Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

21. Kaplan, Robert S & Norton, David B.; The Balanced Scorecard; Harvard University Press, Boston, MA; ISBN 1591391342; 2004.

22. Love, Tom; Object Lessons – Lessons Learned in Object-Oriented Development Projects; SIG Books Inc., New York NY; ISBN 0-9627477-3-4; 1993; 266 pages.

23. McConnell, Steve; Software Estimation – Demystifying the Black Art; Microsoft Press, Redmond, Wa; ISBN 10: 0-7356-0535-1; 2006.

24. Park, Robert E.: SEI-92-TR-20: Software Size Measurement: A Framework for Counting Software Source Statements; Software Engineering Institute, Pittsburgh, PA; 1992; 220 pages.

25. Parthasarathy, M.A.; Practical Software Estimation – Function Point Methods for Insourced and Outsourced Projects; Addison Wesley, Boston, MA; ISBN 0-321-43910-4; 2007; 388 pages.

26. Putnam, Lawrence H.; Measures for Excellence – Reliable Software On-Time Within Budget; Yourdon Press, Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-567694-0; 1992; 336 pages.

27. Putnam, Lawrence & Myers, Ware; Industrial Strength Software – Effective Management Using Measurement; IEEE Press, Los Alamitos CA; ISBN 0-8186-7532-2; 1997; 320 pages.

28. Royce, Walker; Software Project Management – A Unified Framework; Addison Wesley, 1998.

29. Strassmann, Paul; The Squandered Computer; Information Economics Press, Stamford, CT; 1997.

30. Stutzke, Richard D.; Estimating Software-Intensive Systems – Projects, Products, and Processes; Addison Wesley, Boston, MA; ISBN 0-301-70312-2; 2005; 917 pages.

31. Yourdon, Ed; Outsource – Competing in the Global Productivity Race; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-147571-1; 2004; 251 pages.

32. Yourdon, Ed: Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects, Prentice Hall PTR, Upper Saddle River, N.J., ISBN 0-13-748310-4, 1997.