CROSSTALK

April 2005 **The Journal of Defense Software Engineering** Vol. 18 No. 4

CO$T E$TIMATION

The Statistically Unreliable Nature of Lines of Code

Joe Schofield
Sandia National Laboratories

For the past three decades, the ill-defined line of code has been used to describe the size of a software project and often used as a basis for estimating schedule and resource needs. Concurrently, software projects are noted for cost and schedule overruns, and often, for poor quality. This article suggests that the venerable line of code measure is a major factor in poorly scoped and managed projects because it is itself a vague, ambiguous, and unsuitable parameter for sizing software projects. A series of Personal Software Process℠ courses is the source of the data in this article. Because the requirements, instructor, and the lines-of-code counting-specification for these programs were the same, the 60 sets of nine programs offers an extraordinary opportunity for comparing significant variation in software sizes for identical requirements. Given the variation, often greater than an order of magnitude for identical requirements, the use of lines of code as a reliable indicator of software size is challenged.

The Information Systems Development Center within Sandia National Laboratories began a journey with software process improvement using the Capability Maturity Model® for Software as its improvement yardstick in 1999. The Personal Software Process℠ (PSP℠) and Team Software Process℠ were adopted soon thereafter to improve the personal and team practices of the software engineers in the organization.

The rigorous and consistent collection of measurement data prescribed as part of the PSP (and in the examined classes) provides a fertile environment for understanding how software size is estimated versus its actual size upon completion. More interesting though is the study of the size of the software products developed by numerous classes and class attendees for class projects using homogenous and heterogeneous software languages. Both casual heuristic analysis and statistical analysis of these sets of data raise serious suspicions regarding the reliability of using lines of code (LOC) as a software sizing measure.

Software Size Has No Monopoly on Ambiguity

Parents deal with ambiguity when they ask their teenagers when they will be home, only to hear "pretty soon." Spouses experience ambiguity when asking, "How long until dinner?" only to hear, "In a minute." Most consumers at one time or another have purchased *jumbo* shrimp. Science describes distant galactic formations as *small* supernovas. Meteorologists contribute their share to ambiguity by using phrases like *partly cloudy*, *partly sunny*, and apparent synonyms like *mostly sunny* and *mostly cloudy*, respectively.

The least satisfying of these descriptions parallel software customers who are told that their proposed software will be 5 million LOC. Anyone who has ever suspected that the figure 5 million is neither reliable nor accurate will more fully understand some of that discomfort upon completing this article. Anyone who has provided similar numbers for project sizes in the past may be reluctant to ever do so again.

> *"This article suggests that using LOC as a measure for actual product delivery has such wide variation as to render the counts practically useless in the best case, harmful and misleading in the worst of cases."*

This article is not the first to raise questions surrounding the use of LOC. The Definition Checklist for Source Statements Counts identifies 66 variations in counting LOC to document, and as many as eight more that are language-specific [1]. Capers Jones offers this insight on LOC:

> This term is highly ambiguous and is used for many different counting conventions. The most common variance concerns whether physical lines of logical statements comprise the basic elements of the metrics. Note that for some modern programming languages that use button controls, neither physical lines nor logical statements are relevant. [2]

Why Substantial Data on LOC Studies Is Lacking

For data to be exchanged across organizations for benchmarking and eventual insights and learning, a standard definition of a line of code would need to be accepted and applied to participating groups. For an organization to apply data from its own projects for process insight and estimation, many factors need to be identified to minimize the sources of variation that could easily render any gleanings virtually useless. A preferred practice without a context is often a worst practice in another case. Some of the limitations of purported studies related to LOC suffer from one or more of the following challenges.

- **Too few controlled studies.** Many studies of LOC are merely reflections of the type of software, language, and environment in which it was developed. But requirements rigor, design constraints, and customer turnover often contribute as sources of undocumented variation in the development of software size and duration.
- **Too few controlled studies with multiple instantiations of the same set of specifications.** Few organizations can afford to sponsor the repeated development of software code by different software engineers for the

Course / Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
2 / 5	193	137	48	102	107	207	118	67	134
1 / 2	77	163	168	123	134	164	238	178	135
3 / 1	73	37	36	95	101	138	51	66	181
3 / 2	74	97	143	153	279	146	176	80	305
3 / 4	114	71	108	80	219	189	142	95	163
Min. Value	73	37	36	80	101	138	51	66	134
Max. Value	193	163	168	153	279	207	238	178	305
Percent Variation	264	441	467	191	276	150	467	270	228
Mean	106	101	101	111	168	169	145	97	184
Std. Dev.	51	50	58	28	78	29	69	47	71

Table 1: *Lines of Code Counts for PSP Classes by Programming Language No. 1*

purpose of measuring variations in the size of the software.

- **Too few controlled studies with multiple instantiations for different languages.** Few organizations can afford to sponsor the repeated development of software code using different languages for the purpose of measuring variations in the size of the software.
- **Inconsistent measurement approaches.** Few organizations can afford to sponsor the repeated development of software code and then analyze the source of variation attributed to how the software was measured.

Addressing the Preceding Challenges

A PSP course provides an environment that addresses the challenges related to collecting software size measures in the preceding section. Thus, the software measures in the following six tables are extracted from a series of PSP classes taught by the same Software Engineering Institute-certified PSP course instructor. Each class required the attendee to write nine software programs in a language of their choosing – typically the language with which the attendee was most proficient. Each program had associated requirements and acceptance criteria evaluated by the same instructor.

The data from the PSP course was collected in a controlled environment facilitating the close examination of 60 sets of nine software programs (60 students wrote nine programs each). The LOC for each program were counted using the same counting techniques, a point that is proven with the data from the courses (discussed on page 31 and Table 6 on page 32). One of the programs was itself a line-counting program, thus its specification and review reduces one significant source of variation in the counts – the counting method. Reduced variation in counting technique increases the reliability in the numbers used. In those PSP classes in which different languages were used, also present were different levels of education; all participants had at least a bachelor's degree, and about one-half of the attendees had an advanced degree.

Examining the Data

Tables 1-3 cluster the LOC counts for PSP classes by programming language.

Table 2: *Lines of Code Counts for PSP Classes by Programming Language No. 2*

Attendee (same course)	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	221	128	103	227	186	306	155	61	283
2	35	143	114	13	110	63	113	84	85
3	113	106	36	34	53	51	54	61	125
4	90	38	51	61	134	99	43	58	126
5	117	311	271	289	142	122	190	383	219
6	131	179	56	150	202	185	155	118	144
7	184	30	15	30	61	116	69	43	147
8	73	96	102	197	64	158	85	87	126
9	64	63	36	169	56	23	99	73	83
10	101	116	108	49	66	103	71	51	73
Min. Value	35	30	15	13	53	23	43	43	73
Max. Value	221	311	271	289	202	306	190	383	283
Percent Variation	631	1037	1807	2223	381	1330	442	891	388
Mean	113	121	89	122	107	123	103	102	141
Std. Dev.	56	81	73	97	56	81	49	101	65

Using the same format, each table includes columns for the course number and attendee identifier, and the number of LOC for each of the nine programs. The bottom rows include analytic data deriving the minimum and maximum line counts for that set of programs using the same language, the percent of variation between the minimum and maximum values, and the mean and standard deviation of the LOC counts.

The shaded *Percent Variation* (the shaded row in Table 1) for the first shaded cell should be read as a variance of 264 percent between the largest and the smallest programs in this data grouping. Recall that all of the values in each of the P1-P9 columns of this table are derived from software programs written from the same requirement set, validated by the same instructor, using the same language, and counted the same way. Note that a variance of 264 percent is probably not acceptable in purchasing a home (the same home, built to the same specification, inspected by the same inspector, and measured identically), a car, or most consumer or industrial products or services.

A second set of data in Table 2 demonstrates increasing concern. The data collected from this data set came from one class where all the attendees used the same language, but a different language than in Table 1. Note that the smallest percent variation with these programs is almost 400 percent and the largest is more than 2,200 percent. Imagine, for example, the variation on the amount of gasoline received at the local filling station varied between four and 22 times, or the accuracy on the fuel gauge in an aircraft varied this much, or the number of donuts in a dozen, or the amount of beef in your favorite hamburger.

A more troublesome question is, "Which value does the project leader use to make an estimate of the size and, eventually, the cost and resources associated with software?" Are the traditional reasons offered for runaway software projects likely to be as causal as the variations in the size of the code that is developed? Is requirements creep, requirements churn, or team turnover likely to cause a variation of 2,200 percent on a project? Is almost everything we believe about estimating and managing software projects incorrect? How might the true unpredictable size of software using LOC change what we believe about productivity, defects, or reuse?

Lastly, Table 3 contains the values of the third programming language used in the PSP courses. The range of variance is

between 252 percent and almost 1,800 percent. The comments that introduce Table 1 (under the subhead Examining the Data) and the questions that are triggered by analyzing Table 2 apply here as well.

Caution: Quick Fixes Create Other Unanticipated Effects

Attempts to *quick fix* (or pursue the *low hanging fruit)* of the measured variation by eliminating the weakest link on the project – the software engineer who writes the most unneeded code – is unlikely to produce the desired results. While such an approach may seem fruitful based on an initial review of the tables above, consider the following data in Table 4 taken from a class where all attendees used the same language.

In the following example, attendee No. 3 had four of the largest of nine possible programs. (These larger-sized programs are shown in italic, bold typeface.) But attendee No. 3 also had the shortest program, Program 7. (Shortest programs are shaded in cells that have attendee identifiers.) Four other attendees (Nos. 1, 2, 6, and 8) also had the largest program to their credit, while six others (Nos. 1, 2, 5, 6, 7, and 8) had the shortest program.

Please note that overall, attendee Nos. 1, 2, 3, 6, and 8 had both at least one largest and at least one smallest program. The weakest link depends on more than merely who writes the largest program. The weakest link also depends on the program that is selected.

Another erroneous argument could be made for the removal (removal may be a little harsh, maybe retrain, reassign, or *promote*) of attendee No. 3 based on the largest number of most lengthy programs. However, the total number of LOC written for the nine programs was higher for attendee Nos. 1, 2, and 4 than for attendee No. 3. The answer to the question of the weakest link becomes less obvious as different quantitative perspectives are considered.

Further examination of the programs from five classes all written with the same language reveals a significant overlap among software engineers that write both shorter and longer programs (see Table 5). The potential for different software engineers to write programs on both ends of the length spectrum suggests that sometimes the apparently more efficient programmer turns out to be the least efficient, and sometimes the apparently least efficient programmer turns out to be the most (judging efficiency by length since each program met the same stated requirements).

Course / Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1 / 1	89	34	67	40	102	235	23	38	168
1 / 3	82	23	33	48	61	34	33	27	52
1 / 4	177	119	67	85	136	276	165	112	233
1 / 5	76	48	305	244	61	121	66	77	127
1 / 7	46	33	17	37	60	95	129	46	186
3 / 5	22	40	100	58	68	131	58	58	102
3 / 6	46	20	30	42	73	82	51	72	82
2 / 7	95	155	147	94	54	191	174	102	218
Min. Value	22	20	17	37	54	34	23	27	52
Max. Value	177	155	305	244	136	276	174	112	233
Percent Variation	805	775	1794	659	252	812	757	415	448
Mean	79	59	96	81	77	146	87	67	146
Std. Dev.	47	50	95	69	28	82	60	30	66

Table 3: *Lines of Code Counts for PSP Classes by Programming Language No. 3*

Variation then should be attributed to context, which includes both the problem space and the engineer's ability to recognize and utilize strengths and features of the software environment to narrow the solution space.

Another source of variance usually attributed to the differences in size of LOC is the process for counting the LOC. In one study shared by Capers Jones, one-third of the participants counted comment lines as a LOC, one-third did not count comment lines, and one-third could not determine if comment lines were included or excluded. As mentioned previously, the attendees of these PSP classes wrote a program that counted LOC. To determine the effects of how each programmer counted their own source sizes, willing attendees shared their line-counting software and their programs so that they could be counted by each others' software.

Each of the line counts in Table 6 (see next page) was calculated from the LOC counting program written by four attendees. The values correspond as follows:

Table 4: *Example of Attendees With Largest and Smallest Programs*

Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	33	40	30	108	65	***176***	79	107	***284***
2	51	52	24	72	109	166	87	***145***	270
3	***76***	***56***	30	***115***	***175***	158	27	104	128
4	60	52	31	108	94	155	72	94	235
5	22	51	25	50	75	105	47	21	102
6	65	27	***80***	45	95	141	***91***	60	209
7	22	51	25	50	75	105	47	21	102
8	65	27	80	45	95	141	***91***	60	209
Min. Value	22	27	24	45	65	105	27	21	102
Max. Value	76	56	80	115	175	176	91	145	284
Percent Variation	345	207	333	256	269	168	337	690	278
Mean	49	45	41	74	98	143	68	77	192
Std. Dev.	21	12	24	31	34	26	24	44	73

Table 5: *Example of Attendees with Most Lengthy and Shortest Programs*

Class	Number of attendees	Number of attendees with largest program	Number of attendees with smallest program	Number of attendees with the smallest and largest program
1	10	3	6	0
2	8	5	4	2
3	13	7	6	2
4	8	5	6	5
5	10	5	4	1

Attendee No. 1 submitted the values for counting method No. 1, attendee No. 2 submitted the values for counting method No. 2, attendee No. 4 submitted the values for counting method No. 3, and attendee No. 5 submitted the values for counting method No. 4.

For the numbers used in Tables 1-5, please note that in every case, each attendee's submitted LOC values were consistent with the counts provided by others who counted their codes (the shaded rows). While attendee No. 2's software seems to overstate the value of attendee No. 5's sizes, these values were not submitted or included in the numbers used in Tables 1-5. Only the shaded rows below are used in Tables 1-5; that is, only counts submitted by their author are used in the first five tables.

The numbers in Table 6 demonstrate that *variation in counting approaches is not a source of the data variation in this study* because other attendees also counted the subject programs to be of very similar size. Nor did attendees inflate or deflate their own line-count totals, as evidenced by the counts.

Statistical Significance

The apparent differences in the data provoke questions around the statistical relevance of the data. A staff statistician was asked to independently review the data for statistical significance. After conducting a Box-Cox transformation on the data, and performing an analysis of variance, there was a 95 percent probability that the true number of line counts for an individual program from the given population was between 23 and 240 lines. And finally, as is often the case with count data and Poisson distributions, examined variability increased along with size of program.

What is the relevance of the statistical significance? Clearly a 95 percent probability of values that have a range of greater than 10 confirms earlier suspicions that estimating the number of LOC for a given problem is itself highly problematic. While the data in Tables 1-5 evidence this likelihood, the statistical analysis confirms it. A reasonable person, for example, would not procure a computer with such a potential order-of-magnitude variance in performance, cost, or delivery. But unpredictability and variation is the tolerated norm in constructing software.

> *"Because this analysis was conceived and conducted after the classes were conducted, the participants and instructor were unaware that analysis was forthcoming; they themselves were unable to introduce bias into the analysis."*

This norm is evidenced by project performance and by somewhat misdirected attempts at lessons learned and root-cause analyses to identify performance improvements for the future, all dealing with what is likely the wrong problem! The problem itself is often further masked in undocumented overtime and costs, scope containment or reduction, and attempted refinements in estimation variables.

Table 6: *Example From Attendees LOC Counting Program*

Counting Method	Attendee	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	1	91	123	45	121	101	403	553	211	516
1	2	74	97	218	194	279	406	311	181	368
1	4	108	95	205	162	300	484	499	143	706
1	5	193	137	182	229	127	353	353	112	510
2	1	93	133	51	123	107	441	580	213	580
2	2	74	97	218	194	279	406	310	181	368
2	4	110	98	218	317	219	513	523	148	706
2	5	256	172	229	310	170	675	445	122	649
3	1	91	123	45	119	108	380	516	202	479
3	2	74	96	217	194	279	406	310	181	368
3	4	114	78	187	149	303	440	462	130	619
3	5	193	137	181	219	127	517	353	112	510
4	1	91	124	45	120	108	399	548	210	511
4	2	75	98	221	197	282	408	312	182	375
4	4	109	92	202	160	295	476	492	141	672
4	5	193	137	182	209	127	517	353	112	510

Rebuttals Refuted

The data in this article was presented in similar form at conferences and professional meetings. Not too surprisingly, some attendees are quick to defend the widely used LOC for estimating and sizing. Some attendees have doubts that the data applies to their own organization. Despite the rebuttals, each opinion seems to be characterized by one common attribute: no supporting data. The following are some of the most frequently expressed thoughts.

The PSP class is not a good forum for conducting research.

Response: Rarely does an environment exist that controls the requirements and the validation of requirements through the same *control gate* (instructor). Rarely are organizations afforded the opportunity to write the same software 60 times. Rarely are the same programs written in the same language by different authors for comparison. Rarely are the same programs written in different languages for comparison. Rarely are software programs counted using the same counting requirements. And rarely are software programs counted (and cross-counted) by software. Because this analysis was conceived and conducted after the classes were conducted, the participants and instructor were unaware that analysis was forthcoming; they themselves were unable to introduce bias into the analysis. Finding a better environment for conducting LOC sizing is difficult to imagine.

Statistically, the differences between estimates and actual performance average out over time (aka bigger software programs will average out over time).

Response: Apply this principle in other life examples: The buyer of a car with 10 to 15 times the number of typical defects is hardly consoled by the fact that the next buyer may get a vehicle with 10 to 15 times fewer defects than normal. New homeowners will not be comforted that their 2,000-square-foot home was delivered at 100 square feet merely because the purchaser that preceded them received a 25,000-square-foot home; after all, it is merely the luck of the draw. Statistically

the buyers got what they ordered.

A related lesson taught in the PSP course is that granular estimates are more accurate than those developed at a higher level because the *error range* is significantly smaller. For instance, to estimate the time required to build an application applying the error range for the parts (modules, programs, etc.) will provide a more accurate estimate (under similar conditions of knowledge and practice) than an estimate of the application as a whole. This principle, for example, holds true for estimating the size or cost of the rooms of a house, which is a smaller error range than for estimating the house as a whole unit; or for reading the chapters of a book versus reading the book as a whole.

Further, variations in granular estimates tend to offset each other, resulting in an estimate that is closer to actual performance when summed than merely an overall estimate of the time needed to complete the effort. However, a difference exists between the smoothing of variation in *estimates* for a more accurate estimate and the belief that variations in performance (actual) will nullify each other over time. Please note that this lines-of-code analysis was based on actual size variations for the same product; comparisons to estimates were not the subject of this study.

What estimating problem? I'm fine.
Response: This reaction is classic denial when one or more of the following symptoms also exists: project teams that use heroics to complete and deliver a project on time, project teams that use unrecorded overtime to maintain schedule, project teams that use unrecorded resources to complete tasks, projects that are usually late, project teams (not customers) that attempt to renegotiate scope when other project management constraints remain constant, and project deliverables that have unpredictable defects rates compared to projects that predict and manage defects. Admittedly, poor estimating is not the sole source of project delays; team turnover, poor risk management, and true scope changes are additional sources.

The programs' sizes from the course are obviously too small to represent the real world.
Response: Before the introduction of modular programming decades ago, this argument might have had more validity. However, the trend toward modularization, objects, reuse, and architecture-based components challenges the notion that the programs from the PSP course are not in some way representative of much of the software developed today. Certainly the number of LOC that can be peer reviewed in a reasonable two-hour session exceed those represented by many of the programs in the numbers in this study (200 LOC per hour and assuming a two-hour peer review [3]).

Here is what I think ...
Response: The information in this analysis is often received with shock, sometimes relief, and sometimes anger. Many who will read this article are likely to say, "Well here's what I think," followed by a statement that reflects the world according to the lenses through which they choose to see reality. In this discussion, more than 60 sets of data were reviewed and more than 500 lines-of-code counts. An appropriate response to doubters is, "Show me your data." The availability of similar data (same requirements, same environment, similar knowledge-base of participants, no inflation/deflation bias introduced because attendees did not know the study would be conducted, same counting techniques, same instructor/exit criteria, and multiple instantiations of the same requirements set) is quite limited.

Do Not Miss the Point

The PSP course provides a rich observatory for gathering data about software productivity. The course itself teaches the student needed principles for estimating, reviewing, defect removal and analysis, scripting, and process improvement. While the PSP course is the source of the data used in this study, this data does not suggest that PSP is the source of the variation in that data; if anything, the practices from the PSP narrow the variations in lines-of-code counts.

This article suggests that using LOC as a measure for actual product delivery has such wide variation as to render the counts practically useless in the best case, harmful and misleading in the worst of cases.

To record lines-of-code data for estimation and calibration of productivity measures seems troubling based on the data.

Conclusion

The purpose of this article is clear: Statistically significant variation in LOC counts render those counts undesirable for estimating and planning, and deceptive as an accurate portrayer of product size. To those left pondering, "What is a better approach for measuring software size?" despite criticisms, function point analysis, endorsed by International Organization for Standardization/International Electrotechnical Commission 20926:2003, is used by thousands of companies worldwide to measure software size. However, function point analysis has its critics as well.

Further understanding of software size for repeatable and quantifiable sizing to improve estimation and project predictability is still needed. The improved collection and use of software size measures will enhance the credibility of software engineers who are plagued with variation in project cost and schedule.◆

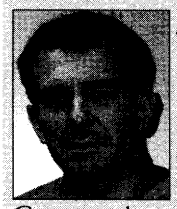
Acknowledgement

I gratefully acknowledge the statistical analysis conducted by Laura Halbleib, a technical staff statistician at Sandia National Laboratories, Albuquerque, N.M. Halbleib's contribution validated the intuitive inferences of the analysis by applying rigorous statistical methods. Her insights and knowledge increased the reliability and usefulness of this material.

References

1. Boehm, B. Software Cost Estimation with COCOMO II. Prentice Hall PTR, 2000: 77-81.
2. Jones, C. Software Quality. International Thomson Computer Press, 1997: 333.
3. Humphrey, W. Introduction to the Team Software Process. Dec. 1999.

About the Author



Joe Schofield is a technical staff member at Sandia National Laboratories. He chairs the organization's Software Engineering Process Group, is the Software Quality Assurance Group leader, and is accountable for introducing Personal Software Process℠ and Team Software Process℠ at Sandia. He has dozens of publications and conference presentations. Schofield is active in the local Software Process Improvement Network and has taught graduate-level software engineering classes since 1990.

Sandia National Laboratories
MS 0661
Albuquerque, NM 87185
Phone (505) 844-7977
Fax: (505) 844 2018
E-mail: jrschof@sandia.gov