The Evolution of Software Size: A Search for Value©

Arlene F. Minkiewicz
PRICE Systems, LLC

Software size measurement continues to be a contentious issue in the software engineering community. This article reviews software sizing methodologies employed through the years, focusing on their uses and misuses. It covers the journey the software community has traversed in the quest for finding the right way to assign value to software solutions, highlighting the detours and missteps along the way. Readers will gain a fresh perspective on software size, what it really means, and what they can and cannot learn from history.

When I first started programming, it never occurred to me to think about the size of the software I was developing. This was true for several reasons. First of all, when I first learned to program, software had a tactile quality through the deck of punched cards required to run a program. If I wanted to size the software, there was something I could touch, feel, or eyeball to get a sense of how much there was. Secondly, I had no real reason to care how much code I was writing; I just kept writing until I got the desired results and then moved on to the next challenge. Finally, as an engineering student, I was expected to learn how to program but was never taught to appreciate the fact that developing software was an engineering discipline. The idea of size being a characteristic of software was foreign to me—what did it really mean and what was the context? And why would anyone care?

Now, 25 years later, if you Google the phrase *software size* you will get more than 100,000 hits. Clearly, there is a reason to care about software size and there are lots of people out there worrying about it. And still, I am left to wonder: What does it really mean and what is the context? And why does anyone care?

It turns out that there are several very good reasons for wanting to measure software size. Software size can be an important component of a productivity computation, a cost or effort estimate, or a quality analysis. More importantly, a good software size measure could conceivably lead to a better understanding of the value being delivered by a software application. The problem is that there is no agreement among professionals as to the right units for measuring software size or the right way to measure within selected units.

This article examines the various approaches used to measure software size as the discipline of software engineering evolved throughout the last 25 years. It focuses on reasons why these approaches were attempted, the technological or human factors that were in play, and the degree of success achieved in the use of each approach. Finally, it addresses some of the reasons why the software engineering community is still searching for the right way to measure software size.

Lines of Code

As software development moved out of the lab and into the real world, it quickly became obvious that the ability to measure productivity and quality would be useful and necessary. The lines of code (LOC) measure—including source LOC (SLOC), thousands of LOC, and thousands of SLOC—is a count of the number of machine instructions developed. It was the first measure applied to software, with its first documented use by R.W. Wolverton in his attempt to formally measure software development productivity [1].

> *"Now, 25 years later, if you Google the phrase software size you will get more than 100,000 hits. Clearly, there is a reason to care about software size and there are lots of people out there worrying about it."*

In the '70s, the LOC measure seemed like a pretty good device. Programming languages were simple and a fairly compelling argument could be made about the equivalence among LOC. Besides, it was the only measure in town.

In the late '70s, RCA introduced the first commercially available software cost estimation tool, which used SLOC converted to machine instructions as the size measure for software items being estimated. In the '80s, Barry Boehm's COCOMO was introduced, also using SLOC as the size measure of choice. As other cost models followed, they too used LOC measures to quantify the amount of functionality being delivered. It is important to note that while all of these models used SLOC as a primary cost driver, there are many other factors that influence the cost of software development as well. These software cost estimation models also need to gather information about factors such as the complexity of the software being estimated, the experience and capability of the software development team, expected reuse, the overall productivity of the development organization, constraints, and so forth. The quantification of these factors is applied to the software size to determine estimates of cost and effort.

I believe that SLOC will go down in the annals of engineering history as the most maligned measure of all time. There are many areas where criticism of SLOC as a software size measure is justified. SLOC counts are, by their nature, very dependent on programming language. You can get more functionality with a line of Visual C++ than you can with a line of FORTRAN, which is more than you get with a line of Assembly Language. This does make using SLOC as a basis for a productivity or quality comparison among different programming languages a bad idea.

Capers Jones has gone so far as to label such comparisons "professional malpractice" [2].

Concerns also surround the consistency of SLOC counts, even within the same programming language. There are several distinct methods for counting LOC. Counting physical LOC involves counting each line of code written while logical lines involve counting the lines that represent a single complete thought to the compiler. In many programming languages, spaces are inconsequential; because of this, the differences between physical and logical lines can be significant. Add to this the fact that even within each of these methods, there are questions as to how to deal with blanks, comments, and non-executable statements (such as data declarations). Programmer style also influences the number of LOC written as there are multiple ways a programmer may decide to solve a problem with the same language.

Additionally, if SLOC is the only characteristic of a software program that is measured, productivity and quality studies will overlook many important factors. Other important characteristics include the amount of reuse, the inherent difficulty of solving a particular problem, and environmental factors that model the approaches and practices of an organization. All of these things influence the productivity of a project.

In general, it is fair to say that SLOC measurement, considered in a vacuum, is a poor way to measure the value that is delivered to the end user of the software. It does continue to be a popular measure for software cost and effort estimation. Even as other metrics have emerged that are considered *better* by much of the software engineering community, many of the popular methodologies used for estimation rely on SLOC; many go so far as to convert the *better* measures into SLOC before actually performing estimates.

There are several likely factors as to why the SLOC method continues to be used despite its many limitations. Many of the organizations that care about software measurement have historical databases based on SLOC measures. So, although it is a valid argument that SLOC are impossible to estimate at the requirements phase of a project, it is not hard to understand why so many organizations find that they can do it successfully within their own product space. They have calibrated their processes and understanding around this and have met significant success using the SLOC method for estimation and measurement within the context of their projects and practices. Another important consideration is the fact that once an organization has agreed on measurement rules for SLOC, counting can be automated so that completed projects can be measured with minimal time and effort and without subjectivity.

Function Points

In 1979, Allan J. Albrecht introduced function points, which are used to quantify the amount of business functionality an information system delivers to its users [3]. Where SLOC represents something tangible that may or may not relate directly to value, function points attempt to measure the intangible of end user value. Function point counts look at the five basic things that are required for a user to get value out of software: Input, Outputs, Enquiries, Internal Data Stores, and External Data Stores. A function point count looks at the number and complexity of each of these components in order to determine the *amount* of end user functionality delivered. Function points create a context for software measurement based on the software's business value.

"Where SLOC represents something tangible that may or may not relate directly to value, function points attempt to measure the intangible of end user value."

Function points also offer a way to measure productivity that is independent of technology and environmental factors. It doesn't matter what programming language is being used or how mature the technology is, it doesn't matter how verbose or terse the programmers are, it doesn't matter what hardware platform is used—100 function points is 100 function points. This provides businesses a way of looking at various software development projects and assessing the productivity of their processes.

While I would be remiss not to acknowledge the great contribution that Albrecht made to the software engineering community with the introduction of function points, I would be equally remiss to stop the story here. Function points are not the answer to all software measurement woes, as they come with their own set of limitations.

Albrecht developed function points to address a specific problem within his organization, IBM. They, like many businesses that developed software, were concerned with the problem of runaway software projects and wanted to get a better handle on their software development processes. According to Tom DeMarco, "you can't manage what you can't measure" [4]. Function points related very closely to the types of business applications that IBM was developing at the time, proving to be a far superior measure of business value than SLOC; function points can be much better for an organization that develops these types of systems to use for productivity comparison studies.

It's fair to say that function points caught on like wildfire in the software engineering community. Many new and successful businesses grew around helping software development organizations use function points to improve their measurement and quality programs, especially for commercial IT software developments. Two problems grew out of the introduction of function points. The first was that the fervor to jettison the much-maligned SLOC measures caused many to embrace function points for all types of systems, many not well-suited to function points. The second was that many tried to use function points as a panacea for all measurement problems.

Function points work best for data-intensive systems where data flows, input screens, output reports, and database inquiries dominate. As the industry tried to use function points to measure the business value of real-time systems, command and control systems, or other systems with several internal logical functions, they consistently under-represented the value that these systems delivered. It turns out that information about inputs, outputs, and data stores is not adequate for determining the value of software that has a lot going on behind the scenes. In 1986, Software Productivity Research developed feature points to try to address this shortcoming with function points. The feature point

definition added algorithms to the entities that are counted and weighted. Mark II function points were introduced by Charles Symons and Boeing introduced three-dimensional function points. The Common Software Measurement International Consortium's (COSMIC) full function points were unveiled in the late '90s and became ISO-certified in 2003. COSMIC function points provide multiple measurement views, one from the perspective of the user and one from the perspective of the developer. All of these alternate methods were intended to address one or more of the weaknesses or limitations of Albrecht's function points—now commonly referred to as International Function Point Users Group (or IFPUG) function points. The industry loved the idea of having a point system to define value, but, as with SLOC, the industry could not agree on the best way to measure points.

Despite the limitations and obstacles, the industry finally had a better way to measure productivity for software development projects. And, if you can use it to measure productivity, it certainly can be used to estimate new projects as well. If your organization knows how many days it takes to build a function point, planning projects into the future should be a breeze. But a crazy thing happened when organizations started using function points to estimate projects: They discovered that things other than business value drove project costs. While function points were good for measuring organizational productivity, they weren't really fitting the bill for estimating cost and effort. The value adjustment factor (VAF) was added to the definition of a function point in a rather weak attempt to address this limitation. VAF takes into account general systems characteristics such as the amount of online processing, performance requirements, installation ease, and reusability. It then uses those characteristics to adjust a function point count based solely on functional user requirements. With the VAF, the function point community managed to stray from business value while adding very limited additional ability to accurately predict development costs. Estimating costs using value-adjusted function points became its own form of professional malpractice.

Function points, in their many variations, offer the software engineering community a better window into business value, although the existence of many definitions does not lead to the cross-cultural comparisons of the productivity desired. They still present a good tool for organizations that develop comparable software products to use for both benchmarking and determining best practices. There are, of course, additional limitations with function points. Although well-documented rules exist for counting function points, there is still subjectivity in the interpretation of these rules. Furthermore, the process of counting function points has yet to be effectively automated; the manual process is time-consuming and requires professional certification.

Other Size Measurements

Other sizing measures have been introduced over the years as well. In the '80s, as object-oriented (OO) design and development gained popularity, there was a flurry of activity to develop software measurements related specifically to artifacts that came from OO designs. These measures made it possible to perform productivity studies across similar projects. Little was done, however, to relate these artifacts to the value that the software delivers, making these studies less applicable outside of a specific application domain. Additionally, because a design was required in order to assess these artifacts, the measures were not particularly suited to estimation. OO metrics never really caught on in a widespread fashion, although there are pockets within the community that have found OO measures they are happy with and can use effectively.

"But a crazy thing happened when organizations started using function points to estimate projects: They discovered that things other than business value drove project costs."

There is a measure which grew out of object orientation that shows some promise in the representation of business value. Use case points were introduced in 1993 by Gustav Karner (see [5]), with use cases being introduced by Ivar Jacobson in the mid-80s [6]. Use cases provide a language for describing the requirements of a software system in a way that facilitates communication between developers and the eventual users of the system. Each use case describes a typical interaction that may occur between a user (human operator or other software system) and the software. The focus is on the functions that a user may want to perform or have performed, rather than on how the software will actually perform those functions. Use case points count and classify the actors in the use case and the transactions that are required to make the use case happen. Use case points describe the functionality being delivered rather than the way this functionality is implemented; in other words, they describe business value. As with function points, there are still technical and implementation details that must be addressed on top of business value when used for estimation. Unlike function points, the use case points can cover a wider spectrum of application types. The problem with utilizing use cases is their lack of standardization across the industry and even across organizations. An organization that has a well-defined process for defining use cases could successfully use them for productivity tracking and effort estimation.

Agile software development practices are adding additional options for measurement of the output and productivity of software projects. Agile development offers a relatively new paradigm for the successful production of software solutions. The tenets of Agile include very short, well-contained iterations of software development that can be carefully measured with respect to the output of business value. Measures of story points, acceptance tests passed, and unit tests created and/or passed replace more traditional measures with values that speak more directly to the business value added in each iteration. Story point measures focus on functionality that provides end-to-end business value. They are defined within the software development group and are used by the group to estimate effort and to measure the productivity of successive iterations. Over time, with discipline, these groups become proficient at assigning story points to the software features they are asked to develop. Test cases developed and/or passed measure the quality dimension of business value. Agile measures such as story points and

tests passed, while having little value outside of a specific software development group for benchmarking or comparison studies, offer a great deal of external value for communicating productivity and quality and provide an excellent tool for negotiating features with management.

Future of Software Sizing

The software industry has struggled during the last 25 years to find the right way to assess the productivity and quality of software development projects. The entire industry continues searching for solutions because high-quality assessment methods are necessary for proper project planning and execution. It is important to understand organizationally how productive our software development ventures are. Organizations hoping to improve software processes also measure in order to benchmark their organization against others considered *best in breed*. The formula for productivity is output divided by effort. Our struggle has centered on finding the right units to describe output.

Clearly, LOC are a very tangible output of the software development process. Just as clearly, they are unsuitable to measure productivity except in very tightly constrained environments because there is no clear relationship between a SLOC count and the amount and complexity of *features* delivered to the end user. Function points, feature points, and all the other derivations of this concept are not real and thus cannot be considered *output* of the software development process. They do, however, supply, in many cases, a quantification of features being delivered to the user. As such, they have promise, within a defined scope, as a measure for productivity across organizations. On their own, they are not sufficient to estimate future software development efforts because they don't measure non-functional requirements that sometimes have significant impacts on the amount of effort required in software development. Additional units of measure have been introduced and have gained some success within pockets of the community, but nothing has managed to achieve widespread popularity.

The software community continues to struggle with measurement issues because they continue to seek the *silver bullet* solution. Every measurement exercise needs to be conducted within a certain context and the temptation to apply one unit of measurement to answer all problems should be avoided. In a perfect world, it would be possible to establish a one-to-one correspondence between the effort associated with a software development project and the business value delivered by that project; in the real world, however, there are other factors that come into play. What seems clear is that with discipline, rigor, and well-defined practices, organizations can be successful using any unit for software size for internal project planning and productivity studies.

So far, we have failed to identify a universally applicable measure for size. The scope of a software project has multiple dimensions. The amount of user functionality is an important dimension but, if viewed alone, it has limited value outside of a very narrow context. External benchmarking and productivity studies need to be performed within stratified categorizations of feature complexity and non-functional requirements.

While there is still no answer to the question of what's the best way to measure the output of a software development project, technology appears to be leading us in a positive direction. You can't open your inbox without finding a few spam e-mails talking about service-oriented architecture (SOA), cloud computing, or some other configuration that separates the implementation of business rules from the implementation of the logistics necessary to deliver these business rules—or, in other words, configurations that separate IT-type functionality from business-type functionality. While still not a silver bullet, organizations that are truly able to achieve service orientation put themselves in a position to both measure the business value of software projects and predict the cost of delivery of future business value using the same units of measurement. This unit of measure, however, may still require definition; if a way can be found to quantify services, that may lead to a better solution to the software measurement quandary. As SOA and related technologies become more widespread (if they do actually become more widespread), this is definitely an area for further research.

The software engineering community should be commended for efforts in size measurements. There have been significant strides during the last quarter century in an effort to evolve measurement practices. There is a continued pursuit of a better measure to describe the output and productivity of software development projects. Simultaneously, the software engineering community is attempting to bridge the gap between IT and the business by working towards a business value-based language to describe our software.◆

References

1. Wolverton, R.W. "The Cost of Developing Large-Scale Software." IEEE Transactions on Computers Vol. C-23, No. 6: 615-636, June 1974.
2. Jones, Capers. "Measuring Defect Potentials and Defect Removal Efficiency." CROSSTALK June 2008.
3. Albrecht, Allan J. Measuring Application Development Productivity. Proc. of the Joint SHARE, GUIDE, and IBM Application Development Symposium. 14-17 Oct., Monterey, CA. IBM Corporation, 1979.
4. DeMarco, Tom. Controlling Software Projects: Management, Measurement and Estimates. Upper Saddle River, NJ: Prentice Hall PTR, 1986.
5. Banerjee, Guntam. "Use Case Points – An Estimation Approach." Aug. 2001 <www.comp.nus.edu.sg/~bimlesh/oometrics/15/1035194512861.pdf>.
6. Cockburn, Alistair. "Use Cases, Ten Years Later." Software Testing and Quality Engineering Magazine Mar./Apr. 2002.

About the Author

Arlene F. Minkiewicz is the chief scientist at PRICE Systems, LLC. In this role, she leads the cost research activity for the entire suite of cost estimating products that PRICE provides. Minkiewicz has more than 24 years of experience with PRICE building cost models. Her recent accomplishments include the development of new cost estimating models for IT projects. Minkiewicz has published many articles on software measurement and estimation and frequently presents her research at industry forums.

PRICE Systems, LLC
17000 Commerce PKWY
STE A
Mt. Laurel, NJ 08054
Phone: (856) 608-7222
E-mail: arlene.minkiewicz@pricesystems.com