By far the most common software sizing metric is source Lines of Code (LOC). When we count LOC we are trying to “size” a system by counting how many pre-compiled source lines we have created, or expect to create, in the final system. Since one of the primary purposes of counting LOC is to determine how big a system might be before we go and build it—say as an input to a project estimation process—we are presented with a dilemma. Before we build a system, we don’t have any lines of code yet, so how can we count them? Once we have written the lines of code and we can count them, we don’t need to estimate since we have the actual values. LOC counts for estimation, it seems, are only accurate when we don’t need them and are not available when we do.

So how about using another sizing method, say Function Points (FPts) [1]? The standard International Function Points User Group (IFPUG) FPts approach involves counting and weighting input, output, and data storage elements with an adjustment thrown in for some aspects of the environment in which the system will operate. The hope is that such counts are available early on in a project, around the time when an estimate is needed. But there are a number of observations we can make. The first is that a common step of the FPts sizing procedure involves translating between FPts and LOC through a process called “backfiring” [2]. Given that the original purpose of FPts was to get away from LOC altogether, this seems a little ironic. Another issue is that FPts as a software product measure suffers from a few drawbacks. Depending on which FPts convention we use, they may not count transform behavior (what many people actually mean when they say “function”), state behavior, platform interaction, design dependence, time-related activity, requirements volatility, and a number of other attributes that legitimately affect the “size” of a system. Some estimation procedures advertise the ability to size a system by module, component, object, business rule, or any number of other aggregations of requirements, design and executable software elements. However, in most cases, these counts must be accompanied by a factor that determines how many LOC there are per whatever is being counted. The procedure simply multiplies the input count by this factor and we are back at LOC again.

Clearly, in order to measure the putative size of a system, we must count something, but it seems every metric leads back to LOC. Some of the estimation...
tools available are very good at converting a size into a projected estimate including schedule, headcount, effort, even predicting defect discovery and risk—provided, of course, we can give them an “accurate” size in LOC. Since many methods employ exponential equations based on size input, any variance on the input predicted size tends to be compounded on the projected output.

**Why We Can’t Count**

There are two reasons why counting LOC (or their equivalent) is difficult and may be ineffectual:

- We actually want to count how much knowledge there is (will be) in the system and there is no way to empirically measure knowledge.
- Even worse, what we really want to count is not the knowledge we have, but the knowledge we don’t have, since it is this (lack of) knowledge that really takes the time in building systems.

Interestingly, if we look at any of our software size measures, we see we are not counting knowledge at all—we are really sizing the substrate on which the knowledge is placed. We are actually measuring how much room the knowledge would take up if we put it somewhere. A LOC count assesses the amount of paper the system would consume if we printed it out. In fact, given an average figure for paper density, we could conceivably produce a weight-based metric for system size(!). Other metrics such as requirements counts, memory size, object size, even FPts, really count how much space the knowledge would take up if we deposited it in different locations.

Since the days of Plato, we have pondered the subject of knowledge and wondered how to measure it. Plato believed that all knowledge was simply “remembered” from a previous life, as described in *The Republic*. In that regard, he predated most estimation methods, which look to historical calibration as the source of “accurate sizing.” But there is still no such metric as a “knowledge unit.” We can determine the number of pages or lines in a book, or even weigh it, but there is no way to empirically measure its knowledge content. The same is true, unfortunately, for computer systems.

Compounding this is the fact that we really don’t want to measure knowledge anyway. What we really want to measure is our lack of knowledge which, of course, we don’t know.¹

**Measurement and Indication**

The art of sizing systems is not really measurement as we usually know it. The metrics we collect, compounded on the projected output.

The business of software is not really measurement as we usually know it. The metrics we collect, such as LOC and FPts, are really indicators of the likely knowledge content and so, we expect and hope, of the time and effort necessary to build the system. The medical profession understands the difference between measures and indicators quite well. If I walk into a doctor’s office with a fever of 101°F, what is wrong with me? Well, we don’t know. This single metric does not diagnose my condition. What most doctors will do is collect further metrics: blood pressure, chemistry, and various other symptoms. When “sufficient” metrics have been collected, members of the medical profession use a specific phrase to describe their

¹We really want to count our Second Order Ignorance (what we don’t know we don’t know), which is the major component of effort. Armour, P.G. *The Laws of Software Process*, Auerbach, 2003, p. 8
analysis: “...these metrics indicate a particular condition.” Sometimes, metrics will contra-indicate the condition and must be explained. So it is with LOC or any other system size measure. If one system is expected to have twice as many LOC as another, it indicates it will take a lot longer to create (generally proportional to the exponent of the size). But it may not.

**LOC is Not Line of Code**

If we look more closely at what LOC represents we see there is a fundamental misassumption. For sizing and estimation purposes it does not actually mean “Line of Code.” Let me explain. Some estimation methods allow you to calculate a “productivity rate.” This is the average rate at which the sizing metric is created, which is derived by dividing the total system size by the total effort. Using LOC, this unit would be LOC per staff month. If we then determine which phase of the life cycle is most productive by dividing the total size by the effort expended in that phase, it sometimes turns out that the most productive phase is project management and the least productive is programming due to the high effort in that phase. Clearly this is bogus, since the only phase that actually produces LOC is the programming phase. LOC is an indicator of system size, but it is not the system size. In fact, if we consider the activity as knowledge acquisition rather than transcribing LOC, it is entirely possible that programming would be less efficient than planning at uncovering systems knowledge.

Another indicator of the true nature of LOC is the way we count them. A company I work with created a voluminous book just on how to count LOC: ignore comments, one source statement per line (what about Lisp?), don’t count test code (say for stubs), only count the invocation of reused code, and so forth. The question is why? Why don’t we count comments or test code? If we need comments or need test code, why should they be left out? The usual response to this is “they are not included in the final product delivered to the customer.” But so what? Comments require work, in a manner similar to actual LOC. If we need to write test code, we need to write it just as we need to write the final code. Anyway, what if we write executable code that doesn’t work, is redundant, or is written to support a future release that this customer will not use. Why do we count those?

**Fully Burdened LOC**

The “…not included in the product” criterion is not relevant. The real reason we don’t count comments is scalability. We make the assumption that the comments and test code (and redundant and future code) are scalable with respect to the executable lines of code. Therefore, we don’t need to count them separately if we assume that the executable LOC contains them. The same is true for requirements, designs, and plans, none of which we deliver to the customer either, but all of which we must create if we want to build executable LOC that will actually execute. We assume that if the LOC include all of these, then they are the LOC we are looking for. The concept is identical to the idea of “fully burdened labor rate.” The cost of an
employee to a company is not just that employee’s salary, it is the salary plus the apportioned overhead of all the expenses necessary to allow that employee to function in the workplace. This includes costs such as lighting, heating, and rent, none of which accrue to the employee, but are necessary. The employee does not work alone in a field, but in a building with other employees and an infrastructure that allows each employee to be effective within a system.

The same is true for LOC. A line of code doesn’t do anything, unless it is included with all the overhead necessary to make it work with all the other LOC in a real system. For estimation purposes LOC does not mean “line of code,” it means “line of commented, test-code-written, requirements gathered, planned, designed…code.” This is not the same as LOC.

**The Count is Not the Code**

We can cheat and produce more LOC by not commenting, not planning, and other tactics. But then we are not talking about the same LOC. In reality, an executable LOC count is simply a count of something we assume will scale proportionally to the knowledge content of the system. But it is not the knowledge and it is not the system. Function Points are not particularly special; they simply count other countable things. The estimation processes we have created are all tunable to some degree to get over the issues that limit the LOC -> knowledge content relationship. If we have experienced developers, we have less effort. The LOC count is not actually smaller, but because they are experienced, they have less to learn. If we reuse code, we can build a bigger system (more LOC) at the same level of effort, since we can borrow the knowledge without having to get it for ourselves. Real-time systems have a higher knowledge density than IT systems, so the knowledge-per-LOC is higher and the number of LOC is usually lower.

**Doctor the Numbers**

The trick is to take a page from the medical profession’s metrics book. Simply collecting one metric rarely gives us the answer—on the contrary, it usually gives us the question. What we should do is collect a bunch of sizing metrics and track them to see what the set of them indicate. A company I worked with did this. In collecting around 20 metrics to see how useful they might be in predicting performance, they found only 12 that seemed to have any correlation at all. Only eight had a strong correlation. The winner? The strongest independent correlating indicator was the number of customer-initiated change requests received in the first month of the project. Clearly, this number has very little to do with the actual final size of the system, but proved to be a powerful guide to the overall effort that would be expended and the time it would take, for reasons that are quite intuitive.

We have to count something and executable LOC are countable, albeit too late to really be useful in early estimating. But we must have some common metric that helps us size systems, and if a couple of millennia of epistemology haven’t come up with a unit of knowledge or a way of counting it, I guess we’ll have to make up our own.

**A LOC by Any Other Name**

I have thought for a long time that the worst thing about LOC is the name. If we had just thought to call it “System Sizing Metric” or “Knowledge Unit” which (coincidentally) has a nearly 1:1 ratio with a count of the executable LOC (allowing of course, for levels of commenting, type of system, amount of test code, and so forth) we might be closer to counting what we really need to count.

**References**


**Phillip G. Armour** (armour@corvusintl.com) is a vice president and senior consultant at Corvus International Inc., Deer Park, IL.

© 2004 ACM 0002-0782/04/0300 $5.00