“I can’t believe that, in this day and age, you suggest counting lines of code!” This was the essence of an email message I received following the publication of a column I wrote on scope-based project estimation. The message was clearly that the line of code (LOC) unit is old, it’s outdated, it’s passé, it’s so 1980s COBOL… There are many problems with counting and using LOC and, frankly, they have been there since the beginning, though their use was certainly more rational years ago. I started professionally in the business of software back in the early 1970s. If the assembly language systems we built ended up containing any usable knowledge, it was because we put it there. Since software development is a knowledge-acquisition activity, what we really need to count is the knowledge we have to get to make the system work. Back in the early days of computing, we had few reuse libraries, primitive operating systems, equally primitive programming languages, and little in the way of tools. Heck, we didn’t even have terminals. So there were very few sources of additional knowledge except the collective brains and efforts of the project team and the resulting lines of code were a very good measure of how much we had to learn. Fast-forward to the present era, and the situation has changed somewhat. We do a lot more integration of pre-built parts than writing code from scratch. Even when we do write code, we are ably supported by robust languages, powerful tools, and comprehensive libraries. These repositories contain tremendous amounts of knowledge that we have access to, we can create systems from, we can provide to the customer, but we don’t actually have to obtain. To some extent, we have decoupled the knowledge delivery from the knowledge discovery and this is a good thing.

As a simple example, to display a pane using the Visual Basic programming language we simply append the “.SHOW” method to an object that is displayable and voila! It appears! Imagine how much you would have to learn if you had to duplicate this capability in, say, IBM 360 Assembler. To build a functioning system, we do need to learn what data we must show on the pane. We must acquire the knowledge of when and to whom the window is visible, and we need to determine what we should do with any data that is input. But we don’t need to learn how to display a window. This means we have less to learn than we would do writing the same application in a lower-level language, so we can create the functioning system much quicker.

But it also means that the knowledge content of the final
system is not as closely related to the lines of code we produce as it used to be.

Even more effective can be the application of reuse libraries, commercial off-the-shelf software, and packaged software. Using these we can build a pretty big system while writing very little code. We do have to acquire the knowledge of what we want these reused functions or packages to do. We do have to learn how to apply them and integrate them effectively. And we do have to make sure they work properly by testing them. But this learning may not be very closely related to the set of command-line instructions we call code.

**Other Metrics**
The demise of the LOC (or “Line of Code Equivalent”—LOCE) has been somewhat exaggerated. Almost all estimation tools still use this unit or something similar as their basis for scope-based calibration. Even when other units are mooted as the basis for defining the “size,” knowledge content, or value of the system delivered to the customer, the other unit must be accompanied by a conversion factor that is basically the other-unit-to-LOC ratio. There are several reasons for this continued use of LOC in estimation tools. One is an accident of history—most of our calibrations were started back in the heyday of LOC and have been carried forward to today. Also, we have to use some common unit otherwise we could not consistently size any system. The other reason is simply that nobody has come up with anything much better than LOC. We cannot empirically count knowledge—there is no generally accepted unit for the quantity of knowledge or way of measuring such. And anyway, project effort is not determined by the knowledge we have, it is a function of the knowledge we don’t have, and how would we count something that is not there?

Proponents of other metrics systems may make claims for the supremacy of alternative counting mechanisms, but are they really better?

**Counting Requirements**
A few years ago, I took part in a study to see if alternative metrics to the perennially unpopular LOC could be used. The defense contractor with whom I worked did a very good job indeed of requirements management. Even fairly early in the life cycle, we could query the requirements management system and get an actual count of the requirements for a given project. This was exciting. Since we had actual counts, not predictions, not suppositions, but actual counts of an artifact that mapped onto the entire system, this held out great hope for an alternative to LOC.

The trouble is: what is a “requirement”? How big and complicated is a requirement? How much work would it take to transform a requirement into a functioning system component? How long will it take to integrate the component with all the other functioning components supporting all the other requirements? In essence: how much knowledge is in a requirement? We didn’t know.

**A Line of Code, A Can of Paint**
During this exercise, I came across two requirements quite close to each other. Both requirements were worded almost identically—to paraphrase: *This system...*
shall support MIL-STD-X. The first of these requirements referenced a set of communications protocols used in military vehicles. The second requirement meant, in effect, that the container should be painted a particular color. The knowledge content of the first requirement could result in a pretty big computer program or interface, the second might be satisfied by a can of paint. Clearly, these requirements were not equivalent.

If we look at the potential for knowledge content in the units we use to try to assess the size/complexity of a system, we see that different units can have different knowledge densities and different ranges of knowledge content.

For all its deficiencies, a LOC is a fairly precise unit of knowledge. One line of code does not vary much from another line of code. The amount of knowledge we must get to write one line is not different from another line by orders of magnitude (the knowledge we have to get to make all the lines of code to work together is another issue for another day). For the unit “requirement,” this is not the case. One requirement could be very big and complicated while the next requirement could be very small and simple. The knowledge range of “requirement” is much larger than that of LOC. It is intrinsically a less well-defined unit.

COUNT AND MEANING

There are two aspects to sizing a system: what is the count, and what is the meaning or definition of the unit being counted? In the early stages of a project, there is a lot we do not know about the project—the primary job of the project is to determine what we don’t know. This lack of knowledge we can call “uncertainty.” Depending on the project, we may have more or less uncertainty. Projects and systems that are similar to those we’ve worked on before have less uncertainty. Brand-new projects have more uncertainty. The COCOMO II estimation model acknowledges this with calibrations such as the Scale Factor “Precedentedness” [1].

When we attempt to size a system with some built-in degree of uncertainty, we have a choice of units we can use and the figure here shows the dilemma we face. If we measure the system using a “well-defined” unit that has a reasonably consistent definition and a narrow knowledge range, we find we have a poorly defined count of that unit. This is what happens when we try to measure early system size using a LOC unit. Project team members might justifiably say “The customer can’t tell us what the system should do—how on earth can we figure out how many lines of code we will have to write?” Certainly we can’t actually count the LOC in the early stages of a project because they don’t exist.

If we size the system using something we are able to count in early development, such as “requirements” we must find that the definition of the unit is uncertain. It seems we can have a good count or a good definition, but not both. But why do I assert that if the count is good, the definition must be poor, and if the definition is good, the count must be poor?

The answer is simple. In the system and environment at the point of early system sizing there is a certain amount of uncertainty. The product of the
We cannot reduce uncertainty in a system by counting differently any more than we will change the actual temperature of a body by switching the temperature scale from Fahrenheit to Celsius.

The Conservation of Uncertainty

We could call this principle the “Conservation of Uncertainty.” The combination of metric count uncertainty and metric meaning uncertainty will always be the same as the intrinsic uncertainty of the thing being measured. Uncertainty is like thermodynamic entropy—it cannot be reduced except through the application of energy. Ultimately, the only thing that will reduce the uncertainty present in a system development activity is to engage in the activity of system development. This means identifying the things that are not known about the system and the values of these things. These are the system variables and the values of these variables. Merely counting using a different metric will not do this by itself.

The are metric methods that have been used very effectively for many years, under certain circumstances. An example of these is “Function Points” (FP). The IFPUG Function Points standard (ISO/IEC 20926:2003), essentially counts the input, output, and storage aspects of a proposed system to derive a total system size in the unit of FP. If the conservation of uncertainty holds, why would this approach be any better than LOC or requirements or use cases? The answer lies in the work done to render the system specification into a form that is countable by FP. Anyone who uses FP metrics must parse the system specification or similar source document looking for evidence of discrete entities, inputs, outputs, inquiries, and systems interfaces. Then they must deduce the possible effects of a variety of “environmental factors.” In doing this work, which can be substantial, the counters inevitably find themselves having to resolve some of the uncertainty present in the specification. It is this effort that does the work, reduces the uncertainty, and results in a “better” size, not the counting and not the metric. As a unit, FP tend to fall somewhere between LOC and requirements in definition, so a FP count tends to be somewhat more ambiguous or variable than a requirements count, but much less than an attempt to estimate a LOC count early in a project.

The conservation of uncertainty means that this will be true of any metric we can devise to size a system. It seems that in system sizing, whatever we might gain on the unit-counting swings, we are doomed to lose on the unit-meaning merry-go-round.

Reference


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

© 2007 ACM 0001-0782/07/0900 $5.00