Build It Faster!

Lawrence H. Putnam and Ware Myers

Quantitative Software Management, Inc.

"Surviving in the marketplace means 'first to market.'"
Stephen E. Cross, director, Software Engineering Institute, Carnegie Mellon University [i]

*"It's still hard to build software quickly, reliably and of high quality. . . Software is increasingly more critical, it needs to be delivered **faster**, and it needs to be of higher quality."*
Eric Schurr, in charge of Marketing, Rational Software Corporation, at Toronto Worldwide Symposium, 1999.

You have probably seen statements like these a few times. You may have even made them yourself when you were called upon to make a speech. "Faster and better"--that is the price we pay for living in a competitive society. But we also reap the rewards. If we are a user of software--and we all are nowadays--we get a wide range of desirable products--faster and better.

Enough of that! Back to the anvil--how can we develop software faster? In particular, how can metrics help us accomplish this goal? Let's take another look at the relationship that describes software development. First, let's make the general statement:

A *product* of *quality* is achieved with *effort* over *time*
at a *process productivity*

Second, let's substitute metrics--some terms that we can measure--for the italicized terms:

$$\text{Size (at Reliability)} = \text{Effort}^a \times \text{Time}^b \times \text{Process Productivity}$$

(We suspect that the software relationship is complicated, so we put a couple of exponents in it.)

Now we have a relationship that is subject to the rules of algebra. That permits us to rearrange the relationship to show what Time depends on:

$$\text{Time}^b = \text{Size (at Reliability)} \;/\; (\text{Process Productivity})(\text{Effort}^a)$$

After years of investigation of data from completed projects, we established the value of the exponents:

$$(\text{Time}^{4/3}) = \text{Size}/(\text{Process Productivity})(\text{Effort}^{1/3})$$

In this form, we see that schedule Time depends upon the other three metrics: Size, Process Productivity, and Effort. "Faster" means that we want to reduce Time. Algebraically, we can do that by reducing Size, increasing Process Productivity, or adding to Effort. On account of those fractional exponents, however, all three of these effects are nonlinear, especially the Effort one.

Let's look first at Effort

The first recourse of managers, when they suspect the schedule is tight, is often to pour on the horses. In terms of the foregoing equation, they beef up the Effort term. They fail to call to mind Fred Brooks' long-standing advice:

"When schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster." [ii]

Now, looking at the equation, we see the formal justification for his famous, but also widely ignored, law:

"Adding manpower to a late software project makes it later."

The reason is that the equation reduces the Effort term to its cube root. The cube root of 3 person-years, for instance, is only 1.44 person-years. Suppose we really go gung-ho on the Effort and increase it one third to 4 person-years. The cube root of 4 person-years is 1.59 person-years, That is only 10 percent greater than 1.44 person-years. The equation greatly reduces the influence of Effort on Time.

At the same time, the equation raises the Time term to a small power: four thirds. That adds to the effect. All together, there is a power of 4 ratio between Time and Effort--our Fourth Power law. To put the effect in numerical terms, tripling the Effort reduces the Time schedule by less than one third. Among those who have examined this relationship, there is disagreement as to just what this ratio is. Some think it is less than four, but it is certainly large.

Moreover, for a given Size project and a given Process Productivity, there is a minimum development Time. That means that no matter how much you increase Effort, you can't reduce Time below that minimum. What that minimum is, for any given project, depends upon the Size of the project and the Process Productivity at which the work is pursued.

Above the minimum development Time, the maximum practical schedule is also limited. To put it another way, you can reduce Effort to about one third of that required at the minimum development Time by extending your schedule to about 130 percent of minimum. We reach the conclusion that merely increasing Effort is not going to get software delivered much *faster*.

Next, let's look at Size

What else could we do?

According to the equation, as well as common sense, if we can reduce the Size of the project, we reduce the schedule Time.

Bells and whistles. Projects resort to the time-honored dropping of features when they run out of time. They let things that the customer doesn't need much anyway slide into the next release. With an estimating method that determined in advance that they could not build all these features in the time the customer desired, they could trim these bells and whistles earlier – before they started. Knowing that these additional features would probably come on future releases, the project could plan the architecture to accommodate them.

Series of releases. A long time ago the software industry used to set up projects on a five-year time scale. That usually turned out to be impractical. The underlying technology changed more rapidly than that:

- tubes to transistors to integrated circuits;
- mainframes to time sharing to minicomputers to personal computers to workstations to Internet-based devices connected to server farms;
- assembly language to third-generation languages to fourth generation languages to GUI environments and code generators;
- structured analysis and design to object-oriented analysis and design.

Besides the technology changes, the society in which the software was to operate changed. For example, business re-engineering came along. By the time the software was completed in five years, say, it was a poor match to its application.

The answer was to divide a big project into a number of releases. Then we try to hold each release to no more than, say, two years. In that period, while change does take place, it is usually within the scope for which we can plan. At the same time we do not bite off, especially in the first release, more than we can chew in the schedule time available to us.

Reuse. The third way to reduce the Size of the system, at least the *effective* size (in new and modified LOC) from the developers' standpoint, is to *not* design and implement every part of the system. In other words, to insert already designed and built components.

Of course, you are going to rebut that reuse is a figment of our fevered imagination. And we are going to freely admit that it is not easy. But we also know that more than a few companies are having various degrees of success with what we now call *component-based development.* You architect your system into subsystems, sub-subsystems, and ultimately components with clearly defined interfaces so that you can insert available components into your architecture.

It takes a while. You are not going to learn how to reduce Size overnight by going to the series of releases or to component-based development. Experience indicates that they take considerable thought over a period of years. If we pursue something of this sort, we might expect to reduce project size gradually. That reduces the time to market.

For one thing, dividing a long time-scale project into a number of releases implies that you cooked up a long-term architecture early in the first release, that is, an architecture on the basis of which you could grow successive releases. If your first architecture, on the contrary, turned out to be impractical, when incorporated as the first release, then you might have to start all over again with the second release. Instead of putting out a second generation in two years, it might then take four years. The point is: thinking through an enduring architecture, an architecture that can support the changes that several releases will bring, is not easy. When you can do it, however, you can reduce Size and get to market faster.

Finally, Process Productivity

If we can increase Process Productivity, we reduce development Time. That is what you suspected all along. That is what the Software Engineering Institute's Capability Maturity is trying to do. Experience indicates that increasing productivity takes time at best. At worst, it is fraught with difficulties. Not everyone has succeeded in doing it.

Still, the companies reporting to our database have been improving their Process Productivity during the 1980s and 1990s. In business systems applications, the average rate of improvement in Process Productivity has been 10 percent per year; in engineering systems, 8 percent; in real-time systems, 6 percent. These rates are well in excess of the average productivity gain in the United States economy during those years: around one percent. Of course, the companies reporting to our database are probably well above the average of all companies. Still, the figures show what good companies can accomplish.

From another point of view, however, progress has been *slow*. One company in the business-systems database sustained a 16 percent per year improvement rate for about 15 years. That is the best record we have. It shows what is possible. That is close to twice as good as the average.

Another way of looking at what is possible is to look at the process productivity gap between companies at the 84^{th} percentile (one standard deviation above the mean) and those at the 16^{th} percentile (one standard deviation below the mean). This middle range

excludes the exceptionally good software organizations and the very poor ones. It is the range where most of us fall.

In business systems that gap is 1060 percent; in engineering systems, it is 590 percent; in real-time systems, it is 550 percent. Faced with differences of those magnitudes, annual gains in the range of 5 to 10 percent look pretty small. Much more is certainly possible for individual companies.

So, you have Process Productivity, Size, and Effort to play with to *Build it Faster*. Just don't expect the thought alone to work a miracle. You have to reason it out and then work to make it happen.

---

[i] Stephen E. Cross, "A Message from the Director," *Bridge,* Issue Two, 1997, p. i.
[ii] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, Reading, Mass. (Anniversary Edition) 1995, 323 pp.