Familiar Metric Management: Year 2000 Turns the Mythical Man-Month on its Head

In his long-famed book, *The Mythical Man-Month,* Frederick P. Brooks declared that,

> "The number of months of a project depends upon its sequential constraints," that is, one thing has to be done after another necessarily preceding thing.
>
> From that it followed that, "The maximum number of men depends upon the number of independent subtasks."
>
> And finally, "From these two quantities one can derive schedules using fewer men and more months."[fn]Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, Anniversary edition 1995 (First edition 1975) 322 pp.[/fn]

Larry reduced these concepts to the software equation in the 1970s and explained them in articles and books since then, including ITMS columns in October 1996 and March 1997. Briefly, the effort-time tradeoff shows that if you extend schedule by up to 30 percent beyond the minimum development time, effort declines dramatically. For example, if effort is taken as 100 at the minimum development time, it drops to about 40 at 130 percent of that time. That is an economy worth seizing—if you can afford the extra time.

Back in May 1997, when we first dealt with Y2K in ITMS, there was still time to take advantage of such savings. We hope many of you did. In fact, Howard Rubin reported that 86 percent of the companies he surveyed (a sample of Fortune 500 companies) had begun "a full-fledged Y2000 strategy." (ITMS Aug. 1998) That sounded encouraging until we read a little further. Their spending projections are too low, say 87 percent of them. They are slipping milestones, say 84 percent. In other words, those famous *bete noires* of software planning, effort (cost) and schedule, are now besetting the march to January 1, 2000.

With the Year 2000 problem, we have something new under the software sun, namely a fixed deadline. There won't be any extra time. We can't quietly run past it, hoping no one will notice or, at least, not complain very loudly, as often happens with less entrenched deadlines. Upper management can't extend it if we run into trouble. The ball game becomes,

- "How much work can we cram into a fixed amount of time, essentially at this writing, the year 1999?"
- "How much work can our (more or less fixed) staff accomplish in this year?'

Cut the amount of work

If you haven't done very much by now, or if you are lagging whatever schedule you did set up, we can start with the assumption that you are not going to get everything done in the coming year. You've got to cut "everything" down. The first step is to inventory your programs and take the following off your Y2K work schedule:

1. Programs that you are no longer using. With more or less infinite storage capacity nowadays, it has been cheaper to let programs sit there than it was to retire them. Take them off the disks and kill them dead, if you dare, but at least, don't remediate them.

2. If you don’t dare kill them (maybe the Antitrust Division will sue you some day and want to look at those old programs), label them not to use with current data. Mark them for use only with 1980s data, or whatever.
3. The program is buggy, slow, treacherous, and you’ve never liked it anyway. Junk it and buy a shrink-wrap that is already Y2K compliant.
4. Some of your recent programs may be compliant. Happy day!

How much can you do?

As a result of the inventory, we have an idea of how much we would like to do. The next step is to figure out how much we can do. We are all accustomed to estimating future projects. The basic process is something like this. We estimate the size of the project in source lines of code, function points, or some other size metric. We have our productivity capability in mind. That leads, via an estimating formula, to an estimate of effort (convertible to cost by multiplying by a labor ratio) and schedule time. Some estimating formulas employ two equations, separating the effort estimate and the time estimate. We employ one equation, as we have discussed in past columns, on the grounds that effort and time are themselves closely related, as Fred Brooks patiently explained to us long ago.

What we mean is that *developing software takes time.* You cannot develop a big system in one month just by putting a thousand men and women on it. Everyone knows that as a matter of practical knowhow. What takes some equations and some statistical work is just how much system some number of people can develop in some amount of time. In the Y2K situation, the time is now essentially fixed—one year. How much remediation, that is, how much system size, can some number of people do in one fixed year? Experience with remediation indicates that the actual number of lines of code that have to be worked on ranges from about three to ten percent of the total SLOC in a system. Just what that percentage is for your systems you have to find out the hard way—by fixing some of them and counting the lines.

We have discussed the software equation and its use in estimating in past columns, so we won’t repeat all that here. We will merely say that given the one-year time schedule, given your estimate of the number of people you can put on the work, the equation can tell you how many source lines of code you can fix in the year. Unfortunately, it can’t tell you *exactly.* That is because none of the numbers used in software estimating are precise.

They all vary over a range, usually following a more or less normal curve around a midpoint.

The software equation has four variables—size, time, effort, and process productivity. In the Y2K case, time—one year—is precise, but the others vary. Running the equation a thousand times (in a computer, of course) provides the statistical means of accounting for these variations. When the inputs are statistically uncertain, the outputs are similarly uncertain.

Table 1 shows the range of outputs for a sample calculation, assuming a given number of staff are available at a given process productivity. In this instance, the mean of the values obtained by the 1000-trial simulation is about 50 percent greater than the one based on a single computation. The values themselves distribute about this mean in the shape of a normal curve. That makes it possible to indicate the level of assurance we can have that the number of lines of code shown in the columns of the table can be remediated. For example, if we want to be fairly certain (84 percent) of getting the work done, we plan on the basis of the numbers in that column. (The table lists only three representative percentages, but the program that does this work can calculate SLOC at any percentage.) Note that you can do an amazing amount of work at a low assurance level! In other words, the odds are you are not going to finish that much.

Table 1. The body of the table contains the number of lines of source code that can be remediated in the 12 months of 1999. The actual SLOC in the programs could be 10 to 20 times that number, depending on the percentage to be remediated.

Estimated SLOC Remediated in One Year

Application Type	84% Assurance	50% Assurance	16% Assurance
Business	12,000	80,000	500,000
Engineering	1800	16,000	130,000
Real Time	3500	11,000	35,000

To apply these concepts to your own situation, you have to gather certain information:

- The approximate percentage of lines in your programs that your staff will work on, say 5 percent. You get this percentage by counting the lines in the first few jobs you do.
- The number of SLOC in each of the programs to be remediated. These programs exist, so you can count the number of lines in them. This number times the foregoing percentage gives the number of lines to be remediated.
- The number of people you have available for this work.
- The process productivity level at which they will probably work. In this connection, we have found that remediation goes more easily than the regular run of development work. That is, staff perform it at a higher process-productivity level (1.5 to 2.0 productivity index points) than they do regular development. That gain is in the range of 40 to 60 percent more lines of code remediated per personmonth than staff would produce on new work. Countering this thought, unfortunately, is Rubin's survey finding that "staff working on Y2000 projects are increasingly rating the work as extremely 'boring.'"

If your programs were all small and if you had a corresponding number of small teams, you could remediate

them all in the one year, 1999. If your large programs were divisible into quite independent pieces, then they, too could be assigned to the small teams that could each do a small piece in one year. The catch is this. Some of your large programs are not divisible and, therefore, cannot be done in one year. The second catch is: you probably don't have enough small teams.

If that turns out to be the case, you have to reduce the number of systems you remediate to the number you can realistically accomplish. Work first on the ones that involve big issues, like life and death. Add to that category those strategically important to your company's continued well being. Set aside for work in the new millenium those where Y2K boo boo's will be merely irritating.

Cheer up

As a software professional, you are aware of the regrettable fact that programs are *always* delivered with resident defects. QSM's defect model, for example, assumes that the average program is delivered at the 95-percent point. Five percent of the originally committed errors still remain to be found and fixed in operation. Even with great effort, real-time programs seldom get much beyond 99 percent.

Our point is the world has lived—quite happily, though with occasional expressions of outrage—with a lot of defects in its software. There may be more such defects in 2000. It may even be quite irritating. Just get the life-and-death stuff fixed!