Familiar Metric Management - "Let Me Count the Ways"

By Lawrence H. Putnam and Ware Myers
Quantitative Software Management, Inc.

"*How do I love thee? Let me count the ways."*
Elizabeth Barrett Browning—Sonnet XLIII

Let me count the ways . . . . to alleviate the current software staff shortages.

There is a lot of talk about *somebody else* doing something about the staff shortages.

- The colleges are to turn out more computer science graduates.
- The government is to allow the immigration of more programmers.
- Human resources is to recruit more applicants.
- The outsourcing companies are to take up the load.
- Microsoft—or IBM, Oracle, SAP, Baan, Computer Associates, or somebody is to provide off-the-shelf software.

We've had the staff shortage for several years now, and all these *somebodies* haven't solved it yet. Hold on, we are going to let the cat out of the bag. They are not going to! In fact, the substantial drop in software development productivity—probably about one-third—(our column, ITMS, May 98) implies that we are reaping the effects of this shortage.

Once upon a time, second assistant commissars waited for first assistant commissars to tell them what to do. First assistant commissars waited for commissars to tell them what to do. And so on, up to Hitler or Stalin or Mao or somebody. History now tells us these know-it-alls did not solve the problems—they just concealed them under a weight of bodies.

That puts the McGuffin[1] right back in your hands. To put it in a nutshell: manage! (The Grand High Commissar is not going to find the solution for you.)

Manage, indeed. Easy to say, but managing software development is probably the most complex single activity the human race has ever faced. It is difficult because it programs every other activity. Software operates more and more of the world's activities. Why, we suspect that software played a role even in the development of Viagra.

So, the management of software development is not easy, but it is possible. The fact that a few organizations do it much better than most proves the possibility. *Let us count the ways.*

Establish feasibility. It seems to be common sense to have a small team spend a few months finding out if you can build that software system. It is a subject to which we devoted our December 1997 column. The Department of Defense has long had a "feasibility study" in its regulations. Civil engineers conduct foundation studies.

[1] Movie slang for the jade ring, the Holy Grail, the secret document, or whatever that the hero pursues through thick and thin.

Yet Capers Jones estimates that 15 percent of the US software effort is dissipated on projects that are eventually cancelled—but not before about $14 billion dollars per year goes down the tubes. [1] He estimates the loss of staff time at 285,000 person-years per year. That figure is in the same ballpark as the Information Technology Association of America's widely quoted 346,000 computer-related job vacancies this year. Ergo, do only feasible projects and the staff shortage goes away!

The urge to embark upon infeasible projects exists more in clients than in working software people. Between the clients and the software people stand the upper management of the software organization and they are often ambivalent. They suspect a marginal project may be infeasible, but they look longingly at the funds it promises to bring in. "Looking longingly" is an emotion. The software people needs facts to counter the emotion. Facts come from metrics.

Get a functional design first. You wouldn't build anything much larger than a backyard tool shed without an architectural design—unless you call it software. Then a lot of managements let you start writing code as the first order of business. In fact, we know people who proudly proclaim, "I am a programmer. I write code. I don't do all that up-front stuff."

That attitude is all right on small programs that you can get your arms—and your head—around It is not all right on big programs. Here the ages-long experience of the older engineering disciplines tells us there is a process to go through:

- Figure out what the system is to do;
- Analyze what that means in the problem domain;
- Lay out a flight plan—the functional design or the overall architecture;
- Convert this broad outline into detailed design;
- Along the way, review or inspect each step;
- From an early point, begin to develop the test plan;
- And only now, finally, write some serious code.

That is, you ought to write some illustrative code along the way, to establish that a new algorithm actually works, for instance, but you don't sit down to write the final code until you know what you are supposed to do. Programming, in the sense of writing code, is only 10 or 15 percent of software development. The rest is much more interesting and could be made attractive to young people who dread the idea of becoming nerds.

Abate risk. Developing big software systems is risky, as the high cancellation rate attests. Jones reports further ". . . of large software systems . . . that are completed, about two thirds experience schedule delays and cost overruns that may approach 100 percent." [2] The answer to the "risks" problem is to find them and abate them early. Don't wait for them to inundate you in system test. That is what the early stages of the engineering process are for. (We touched on risk a bit in July 1997.)

Plan realistically. Ok, so now we have established that we can do the project and that we have the risks under control. We know enough about what is to be done to make a realistic plan. Furthermore, if we have had the forethought to keep a few metrics (note the sarcasm), we can estimate schedule, effort, cost, and reliability. Good estimates have a very great effect on the staff shortage, for instance:

- If your estimates allow an adequate schedule, people have time to do the work and still have a *life*. They are less likely to leave to find a life elsewhere—staff turnover is currently very high at most companies. Mid-life parents are telling their college-age children, "Whatever you do, don't go into programming." They are themselves seeking other lines of work. To put it bluntly, poor estimates are creating the staff shortage, not solving it.
- Beyond adequate schedules lie still better schedules. Use a smaller team, allow a little more time and reap reduced effort and better reliability—the effort-time tradeoff law that we spent time on in October 1996 and March 1997. In August 1998 we reported our latest findings on the right team size—namely, small. These "better" schedules save a substantial amount of person-hours, as much as 60 percent, as shown on our diagram in October 1996. This tactic alone could end the staff shortage!

Reuse components. It has been evident for several decades that reinventing the wheel on every project took a lot of staff. Unfortunately, it also became evident that "inventing" reuse was tricky. Nevertheless, a few organizations have had considerable success in this area, so we know it is possible. See our columns in April 1997 and March 1998.

Improve your process. The effort-time tradeoff has a limit—the number of people on a project has to be positive, but we know of no limit to improving the software process. The first step is to have a *process.* About two thirds of the organizations assessed on the Software Engineering Institute's Capability Maturity Model fall in Level 1, meaning that they lack a repeatable process. On our own process-productivity index, projects range from 1 to 30, ignoring a few outliers. The average for business systems is around 17. There's still plenty of room for improvement!

The people reporting data to us averaged a gain of one index point every two and a half years. One index point is roughly equivalent to a 10 percent reduction in schedule, 25 percent reduction in effort, and 25 percent improvement in reliability. The exact amounts vary, depending on how individual projects make their plans. For our current purpose of considering the staff-shortage problem, however, if every software organization could improve its process by one process-productivity index point in the next 2.5 years, it would free up about 25 percent of its current staff. [***Strictly speaking this would apply only to projects that were about to begin, or were near the beginning. There is a time period or lag to set this in motion. The ones underway and toward the end usually don't benefit instantly; usually it is on the next project after the process improvement actions are in place and assimilated. So the concept is okay, but 25% is probably unreasonably high when applied across the portfolio of applications in development. A better number might be 1/3 to 1/2 of the portfolio or 0.25*0.3333 or 0.5 ~ 8 to 12%.]*** If the US presently has about 2,000,000 software people, that would free up about 500,000 people. That would fill the 346,000 estimated shortage and provide 154,000 people for whatever is pressing in 2001.

Well, we have "counted the ways." The means to resolve the staff shortage in-house are there. In fact, they are there several times over. Many of them have been there for a decade or more. The fact that the staff shortage is still there means there is more to alleviating it than just "counting the ways." Nevertheless, the McGuffin is out there. If you are a "hero," you will keep on looking for it.

1. Capers Jones, *Assessment and Control of Software Risks*, Prentice Hall Inc., Upper Saddle Rive, NJ,1993.

2. Capers Jones, “Patterns of large software systems: Failure and success,” *Computer,* March 1995, pp. 86-87.