Robert L. Glass

How Not To Prepare for a Consulting Assignment, and Other Ugly Consultancy Truths

"Tell me it's unrealistic before you commit. Then do it anyway."—Mark Servello of ChangeBridge and Ravi Apte of Citicorp, discussing upper management's view of schedule negotiation in their keynote address to the Pacific Northwest Software Quality Conference, 1997

I recently did some consulting for a computing organization in industry. For some personal reasons, I hadn't consulted for a while.

I worried about the risk that in my several years of uninvolvement I would be out of date on how an organization could and should improve itself. So I decided to do a little more preparation than usual for this assignment, and I prepared some checklists of problems to look for.

Given the popularity of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM), I decided to start there. I would not be conducting an official CMM evaluation, of course, but I figured the CMM was a good place to look for a set of activities that good software organizations should be engaged in. I picked out the key process areas—at all five levels, since I wasn't sure what the status of my client's organization might be—and made them into a checklist. Preparatory task number one was complete.

STEVE ADLER

But I wanted to go well beyond the CMM. In my view, the CMM is about management and about process, and there seemed to be good reason to believe that my clients could have problems outside those boundaries. Because I strongly believe people are the key to software success—they are more important, in my view, than process—I armed myself with another SEI product. After Watts Humphrey entered semiretirement several years ago, Bill Curtis joined the SEI to head up the CMM effort, bringing with him a strong belief in the importance of people on a software project (essentially the same view I have). While he was continuing to move the CMM forward, he also initiated work toward a People Capability Maturity Model (PCMM). That work has now gone sufficiently far so that documents describing it exist. PCMM is being used enthusiastically in some parts of industry, and it forms a second leg of the software maturity table the SEI is building. I abstracted the key process areas from the PCMM and made a checklist. Preparatory task number two was complete.

However, I still felt underprepared. Certainly people and process are important to the success of a software organization, but there is at least one other area that could make a difference in project success—technology. The SEI has, in fact, talked about an eventual

goal of building a three-legged software maturity table, including technology as that third leg. But, partly because of the success of their process and people approaches, they have simply never gotten around to it. (The SEI has also discussed a fourth leg, acquisition management, but again they have done little toward that goal, and it did not seem relevant to the organization for which I was to consult.) How would I come up with a checklist of key technology practices?

I turned to the book, *ISO Approach to Building Quality Software* (Prentice-Hall, 1996) I co-authored with Osten Oskarsson of Sweden for some technology key process areas. (It is important to point out that, unlike most of my software colleagues, I view quality as a primarily technical, not a management, topic. The reason for my believe is my view that the quality attributes, such as reliability and efficiency and maintainability and portability and more, can only be put into the software product—and monitored in the software product—by technologists. These attributes are considerably more technically complex than the average manager can—or should—be able to monitor.)

I believe the technical approaches to software projects are, of necessity, project-dependent. That is, one approaches software projects very differently according to their:

- *Size*. Large projects require more formality (in various forms, including communication/documentation) than small ones.
- *Application domain*. Information systems projects are quite different, for example, from real-time or scientific ones.
- *Criticality*. Reliability, for example, is an overwhelmingly important goal on life- and dollar-critical projects.
- *Levels of innovation*. If you've never solved a problem like the current one before, you are likely to use new and untried approaches to complement the more predictable ones.

To bring this increasingly long story about preparation to a close, I armed myself with some technical key process areas, carefully noting which were important for the kinds of projects I knew my client was engaged in. Preparatory task three, the last of those tasks, was now complete.

I was now prepared, I felt, for whatever category of problems my prospective client was facing. I could attack process problems with wisdom from the CMM. I could solve people problems using the PCMM. I could fix technical problems using my book. Boy, was I ready! Bring on the consulting trip.

But from my first day on site, I realized all my preparation was for naught. None of these three elaborate preparations even came close to addressing the problems my client faced. There was something else deeply wrong with the organization I worked with, and it was only marginally about process; it was only marginally about people; and it was only marginally about technology.

The problem was *schedule*. A choking, confining, impossible schedule. The organization with which I was meeting—the people with whom I was talking—knew about all those key process area points, all the right ways to build software. The problem: the schedule was so tight that they believed they couldn't do them.

Because of these schedule problems:

1. The system was underdesigned. Management, concerned about the amount of time design had taken, told the software engineers to "start coding." (I have always thought the story about managers telling programmers to start cutting code prematurely was a tall tale. Was I wrong!)
2. The code was terribly fragile. (What code wouldn't be, if the design was never completed!)
3. Enhancements took far too long to achieve, and regression errors were endemic. (Of course, given that the code was fragile and the design was inadequate …)
4. The code was hard to read, and lacked commentary. (Another frill abandoned in the interests of scheduling.)
5. Developer testing was inadequate, and the system test organization found most of the errors (same thought).

6 No one had ownership of the code. (Because of the schedule, consultant and student help were brought in, resulting in (at best) fragmented coding practices, and little sense of responsibility.)

The managers and technologists of my client knew all about making good design, writing good code, testing appropriately (including regression testing), and how to encourage people to take responsibility for the work they did. They knew all that stuff, in spades. The problem was, they

weren't being allowed to do it.

What was a consultant to do? I could go through my carefully prepared checklists, and give them all the wisdom of the SEI and myself. But it wouldn't have helped a whit. There is a reason you bring in consultants when you are suffering, of course. If all goes well on the consulting gig, your consultant finds out what you already knew about, and puts those findings into a presentation for your upper management. And upper management, always prepared to believe a consultant more than their own in-house talent, will finally realize that what they have been hearing all along from their own people is true.

consulting client. They took the pitch I made to them at the technical/line manager level, and turned around and presented it to upper management. And upper management said all the right things about understanding what was happening and why. Superficially, you would expect this story to have a happy ending. But I doubt it. There is something now happening in the software world: Competition is forcing enterprises into impossible schedules.

In retrospect, I should have known in advance that my client's problem was excessive schedule pressure. I think that's probably the most common characteristic of software projects today, across

might think of; that the problem is culturally ingrained, a product of our times; that the obvious solutions to the problem simply don't work; and that, without a solution to the problem, the software field is going to go right on doing the wrong thing in the wrong way at the wrong time—because there is no mechanism to fix it.

I have this terrible aversion to ending a column on a down note: and since this one has hit rock bottom, let me add one more bottom line: Schedule pressure is the most serious problem facing software projects today. There is no silver-bullet answer to the problem (if there were, I'd bottle it and make a fortune!) The only approach to

THINGS HAD LITERALLY CRAWLED TO A HALT IN SYSTEM TESTING,

with each new bug fix generating enough new bugs that

almost no forward progress was achieved.

And so it was on this consulting job I recommended all that stuff my clients already knew about, and made that the one focus of my consulting recommendations. And then I made a heavy, pointed pitch for schedule relief. I said things like, "The approach you are taking to accelerate schedule is actually costing you long term, in that enhancements and system testing are taking such unpredictably long periods of time that it is not possible to achieve any anticipated schedule." (Things had literally crawled to a halt in system testing, with each new bug fix generating enough new bugs that almost no forward progress was achieved.)

It is too early to know whether my advice has really helped my

industry lines and domain lines.

I want to say something about "guts management." It is time for first-line managers to stand up for what they believe in and muster the guts to just say "no" when management demands the impossible. But I now know this type of approach doesn't work in today's culture. For every manager who says "no" to achieving the impossible, there are 10 standing by who are willing to say "yes." And one of those will be brought on to replace that nay-saying manager.

So what's the bottom line? That the schedule problem is the most overwhelming problem of today's software organization, overriding almost any other problem you

working within the problem is plain, old-fashioned, communication. Tell upper management the truth, not what they want to hear. Work with them when they can't accept the truth. Define fallback positions for when the inevitable failure to achieve schedule occurs. Let upper management see those fallback positions in advance. Use a "we're in this together" approach, not a "look what they're doing to us" approach. And pray! C

Robert Glass (rglass@indiana.edu) is the publisher of *Software Practitioner* newsletter and editor of Elsevier's *Journal of Systems and Software*.