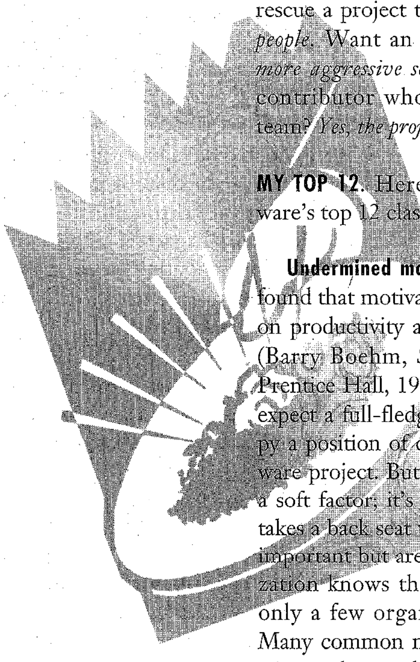
Avoiding Classic Mistakes

Prospecting for programmer's gold.

Editor:
Steve McConnell
Construx Software Builders
PO Box 6922
Bellevue, WA 98008
stevemcc@construx.com



SOME OF THE WORST PRACTICES IN THE software industry have been used so often, by so many people, to produce such predictably bad results, that they should be labeled "classic mistakes."

Most classic mistakes have seductive appeal, which is part of the reason they've been made often enough to be considered classics. Need to rescue a project that's behind schedule? *Add more people.* Want an earlier delivery date? *Just set a more aggressive schedule.* Should you keep a key contributor who's aggravating the rest of the team? *Yes, the project is too important to let him go.*

MY TOP 12. Here are my nominations for software's top 12 classic mistakes.

Undermined motivation. Study after study has found that motivation probably has a larger impact on productivity and quality than any other factor (Barry Boehm, *Software Engineering Economics*, Prentice Hall, 1981). Considering that, you would expect a full-fledged motivation program to occupy a position of central importance on every software project. But that's not the case. Motivation is a soft factor; it's difficult to quantify, and it often takes a back seat to other factors that might be less important but are easier to measure. Every organization knows that motivation is important, but only a few organizations do anything about it. Many common management practices are penny-wise and pound-foolish, trading huge losses in motivation and morale for minor methodology improvements or dubious budget savings.

Uncontrolled problem employees. Failure to deal with rogue programmers has been a well-understood mistake at least since Gerald Weinberg published *Psychology of Computer Programming* in 1971 (Van Nostrand Reinhold). But a study by Carl E. Larson and Frank M.J. LaFasto found that failure to deal with a problem employee is still the most common complaint that team members have about their leaders (*Teamwork: What Must Go Right; What Can Go Wrong*, Sage, 1989). This study was not specifically about software, but I think software teams are just as susceptible to this problem. At best, failure to deal with problem employees undermines the morale and motivation of the rest of the team. At worst, it increases turnover among the good developers and damages product quality and productivity.

Noisy, crowded offices. Need to save money? A common economy is to cram developers into low-budget office space. Most developers rate their working conditions as unsatisfactory and report that they are neither sufficiently quiet nor sufficiently private. Workers who occupy noisy, crowded work bays or cubicles tend to perform significantly worse than workers who occupy quiet, private offices (Tom DeMarco and Timothy Lister, *Peopleware*, Dorset House, 1987).

Every organization knows that motivation is important, but only a few do anything about it.

Abandoning planning under pressure. Project teams make plans and then routinely abandon them (without replanning) when they run into schedule trouble (Watts Humphrey, *Managing the Software Process*, Addison-Wesley, 1989). Without a coherent plan, projects tend to fall into a chaotic code-and-fix mode, which is probably the least effective development approach for all but the smallest projects.

Shortchanging upstream activities. Project teams that are in a hurry try to cut nonessential activities, and because requirements analysis, architecture, and design don't directly produce code, they are easy targets for the schedule ax. On one disaster project that I took over, I asked to see the design. The team leader told me, "We didn't have time to do a

Continued on p. 111

Continued from p. 112

design." Also known as "jumping into coding," the results of this classic mistake are all too predictable. Time is wasted implementing hacks, which are later thrown out and redeveloped with more care. Project teams that skimp on upstream activities typically must do the same work downstream at anywhere from 10 to 100 times the cost of doing it earlier (Barry W. Boehm and Philip N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, Oct. 1988). "If you can't find time to do the job right in the first place," the old chestnut goes, "how will you find time to do it again later?"

Shortchanging quality assurance to improve development speed. On a rush project, team members often cut corners by eliminating reviews, test planning, and all but the most perfunctory testing. This is a particularly unfortunate decision. Short-cutting a day of QA activity early in the project is likely to add 3 to 10 days of unnecessary activity downstream (Capers Jones, *Assessment and Control of Software Risks*, Yourdon Press, 1994).

Lack of feature-creep control. The average project experiences about a 25 percent change in requirements from the "requirements complete" stage to first release (Jones 1994). This produces at least a 25 percent addition to the software schedule—and probably much more, because of the multiplicatively higher costs associated with doing work downstream. Many projects lack formal change-control processes that could help limit changes to those that are absolutely necessary.

Silver-bullet syndrome. The silver-bullet syndrome occurs whenever managers or developers expect any single new tool or methodology to solve all their productivity problems. Silver-bullet tools and methodologies damage projects in two ways. First, the new tools or methodologies virtually never deliver improvements as dramatic as promised. Project-wide productivity improvements of more than 25 percent from first use of a new tool or methodology are virtually unheard of. Second, belief in silver bullets leads to serialization of improvements that could be made in parallel. Because managers or developers put all their faith into a single silver bullet, they try promising new tools and methods one at a time rather than two or more at a time, which slows the adoption of potentially beneficial new tools and methods other than the silver bullet. The bottom line is that organizations that succumb to silver-bullet syndrome tend not to improve their productivity at all; indeed, they often go backward (Jones 1994).

Wasting time in the "fuzzy front end." This is the time before the project starts, the time normally spent in the approval and budgeting process. It's easier, cheaper, and less risky to shave a few weeks or months off the fuzzy front end than it is to compress a development schedule by the same amount. But it's not uncommon for a project to spend months or years on these preliminaries and then to burst out of the starting gates with an aggressive, often unattainable schedule.

Insufficient user input. A 1994 Standish Group survey, "Charting the Seas of Information Technology," found that the primary reason IT projects succeed is because of end-user involvement. Projects without early end-user involvement increase the risk of misunderstood project requirements and are especially vulnerable to time-consuming requirements creep.

Overly aggressive schedules. The same survey found that the average IT project took about 220 percent of its planned schedule. Scheduling errors of this magnitude set up a project for failure. Plans based on estimates that are wrong by more than 50 percent cannot be effective. The most serious consequence is probably that, if upstream activities are abbreviated proportionately to the condensed schedule (more than 50 percent), the average project might be doing as much as half of its upstream work downstream, at 10 to 100 times its nominal cost. Overly aggressive schedules also put excessive pressure on developers, which ultimately hurts both morale and productivity.

Adding developers to a late project. Perhaps the most classic of the classic mistakes is adding developers to a project that's behind schedule. There are exceptions to the rule, but generally when a project is behind schedule, new people subtract more productivity from existing staff than they add through their own work. Fred Brooks likened adding people to a late project to pouring gasoline on a fire (*The Mythical Man-Month*, Addison Wesley, 1975).

Do postmortems; share war stories with colleagues in other organizations; post checklists; appoint a "classic mistakes" watchdog.

CALL TO ACTION. This list of mistakes is hardly exhaustive. I have simply identified the ones I have seen most often. Your list might be different.

Regardless of the exact contents, keep some list of classic mistakes in mind. Conduct project postmortems to identify the classic mistakes particular to your organization. Exchange war stories with colleagues in other organizations to learn about the mistakes they've made. Create checklists of mistakes for use in your project planning. Post lists of classic mistakes on your group's bulletin board for use in project monitoring. Appoint a "classic mistakes watchdog" to sound an alarm if your project begins to succumb to a classic mistake.

The classic mistakes' seductive allure brings them into play again and again, but we as an industry have gained enough experience to recognize them for what they are. Now that we recognize them, we just need to be hard-headed enough to resist their appeal. ◆