Best Practices

***IEEE Software*, Vol. 15, No. 2, March/April 1998**

Dealing With Problem Programmers

"For me, programming is a head rush. I'm always working on the edge. When I put software out there, I don't know whether it's going to work. That's part of the excitement. If it's not exciting, I don't want to do it."

"What do I do with a problem programmer?" This is the question I hear most often in my consulting work. Stories of problem programmers abound. One developer initially refused to follow the project's design standards in creating a certain module. After finally being given an ultimatum, he coded the module—but he wrote the variable names, function names, and comments in German. Another developer insisted that he was making steady progress and that his Microsoft Windows code was finished; meanwhile he spent two months exploring the new Apple Newton. When the time came to integrate his code with the rest of the team's work, none of it was done. He had created some rough prototypes, but nothing was tested, debugged, reviewed, or even remotely ready to be released. Another belligerent developer had apparently never heard of egoless programming. Whenever she found a defect in another programmer's work, she would say, "OK Mr. Smarty Pants Programmer. If you're so great, how come I just found a bug in your code? I guess maybe you're not so smart after all."

How Bad is Bad?

In addition to attitudinal differences, significant productivity differences among programmers have been well documented. In the first study on the subject, Sackman, Erikson, and Grant found differences of more than 20 to 1 in the time required by different developers to debug the same problem ("Exploratory Experimental Studies Comparing Online and Offline Programming Performance." *Communications of the ACM,* January 1968). This was among a group of programmers who each had at least 7 years of professional experience.

This basic result—demonstrating at least 10 to 1 differences in productivity—has been reproduced numerous times, but I think it understates the real productivity differences among practicing programmers. Tom DeMarco and Timothy Lister conducted a coding war game in which 166 programmers were tasked to complete the same assignment ("Programmer Performance and the Effects of the Workplace," in *Proceedings of the 8th International Conference on Software Engineering,* August 1985). They found that the different programmers exhibited differences in productivity of about 5 to 1 on the same small project. From a problem employee point of view, the most interesting result of the study is that 13 of the 166 programmers

didn't finish the project *at all*—that's almost 10 percent of the programmers in the sample.

In a study with similar findings, Bill Curtis presented a group of 60 professional programmers with what he characterized as a "simple" debugging task ("Substantiating Programmer Variability," *Proceedings of the IEEE*, vol. 69, no. 7, 1981). In spite of its simplicity, 6 of the professional programmers weren't able to complete the task, and data on their performance was excluded from the results of the study. Curtis observed order of magnitude differences among the programmers who were able to complete the task.

What are the real-world implications of working with programmers who can't complete their work? On a real project, "not finishing at all" usually isn't an option, and so those programmers who didn't finish during the Coding War games or during Curtis's debugging test would require either huge amounts of time to complete their work or someone else would have to complete their work for them. On real projects, these 10 to 1 differences in productivity might well translate into *negative 10 to 1 differences in productivity*, because eventually someone else will have to redo the work of programmers who can't finish their assignments.

Low productivity by itself usually isn't the only problem. Strained to the limits of their abilities by the coding activity itself, low productivity programmers are either not able or not willing to follow project coding conventions or design standards. They don't remove most or all of the defects from their code before they integrate it with other people's work, or before other people are affected by it. They can't estimate their work reliably because they don't know for sure whether they will even finish. Considering the absence of direct contributions to the project and the extra work created for the rest of the team, it's no exaggeration to classify these programmers as "negative productivity programmers." The study data suggests that about 10 percent of professional programmers might fall into this category. A team of seven randomly selected programmers therefore has about a 50/50 chance of including at least one negative productivity programmer.

Whose Problem Is It?

As often as manager's ask what to do with problem programmers, individual team members probably ask that question more. In a review of 32 management teams, Larson and LaFasto found that the most consistent and intense complaint from team members was that their team leaders were unwilling to confront and resolve problems associated with poor performance by individual team members (*Teamwork: What Must Go Right; What Can Go Wrong*. Newbury Park, CA: Sage, 1989). They reported that, "More than any other single aspect of team leadership, members are disturbed by leaders who are unwilling to deal directly and effectively with self-serving or noncontributing team members." They go on to say that this is a significant management blind spot because managers nearly always think their teams are running more smoothly than their team members do.

On one of my projects, one programmer usually arrived at work about 10:30. He went to lunch between 12:00 and 1:30, he left the office to work out at a health club between 3:00 and 4:30, and he left work by 6:00. The project team members were well aware of the problem, and complained to the project manager. The project manager imposed "core hours" that required all team members to be at the office between 9:30 and 3:30, at which point the problem programmer threw a shouting tantrum and complained about the "abusive Draconian measures" that were being used to impose unfair restrictions on his personal liberties.

Warning Signs

Problem programmers are easy to identify if you know what to look for:

- They cover up their ignorance rather than trying to learn from their teammates. They actively resist having teammates review their designs or code.
- They are territorial. "No one else can fix the bugs in my code. I'm too busy to fix them now, but I'll get to them next week." They'll keep files checked out of source code control exclusively for weeks at a time even when that prevents their teammates from doing their work.
- They grumble about team decisions and continue to revisit old discussions long after the team has moved on. "I still think we ought to go back and change the design we were talking about two months ago. The one we picked isn't going to work."

Cutting Your Losses

If your organization permits it, here are three solid reasons to simply remove the negative productivity programmer from the team:

- It's rare to see a major problem caused by lack of skill. It's nearly always attitude, and attitudes are hard to change. If the problem is caused by lack of ability, that is even harder to change.
- The longer you keep a disruptive person around, the more legitimacy that person will gain in the eyes of other groups and managers, the more other people's work will be affected, the more code that person will be responsible for—overall, the harder it will be to remove him from the team.
- Some managers say that they have never regretted firing anyone. They've only regretted not doing it sooner.

You might worry about losing ground if you replace a team member, but on almost any size project you'll more than make up for the lost ground by eliminating a person who's working against the rest of the team.

Prevention

One of the best means of detecting problem programmers is by holding early design and code reviews. You can identify team members who don't want to share their work, who won't accept teammates' suggestions, who won't take the time to review other team members' work—in short, team members who are generally uncooperative.

If this early detection of problem employees fails, reviews provide a secondary benefit of lessening the dependence on any single developer. One problem typically associated with problem programmers is that no one else understands their designs or code. Through design and code reviews, you'll have at least two people on the team who are familiar with every part of the program. If you find a developer on your team who won't participate in reviews, treat that as an unacceptable risk to the project. Insist that the developer participate in reviews, or let him go.

For the Good of the Team

Tolerating even one problem programmer hurts the morale and productivity of the good developers. Problem programmers are often viewed as having "low productivity," but both software research and software experience suggest that such an assessment is too optimistic. Next time you need to improve productivity, instead of looking for what you can add, look for who you can take away.

Editor: Steve McConnell, Construx Software Builders,
P.O. Box 6922, Bellevue, WA 98008.
E-mail: stevemcc@construx.com *- WWW:* http://www.construx.com/stevemcc/