6

Constant, Unceasing Improvement

During this year's Winter Olympic Games, I was struck by one aspect of the figure skating events. The television footage of earlier gold medal performances seemed to suggest that 25 years ago you could win a gold medal with a few layback and sit spins, a couple of double toe loops, and a clean, graceful program. Today such a simple performance, however pleasing to watch, wouldn't win a hometown skating championship. Nowadays you must do at least three triple jumps, several combination jumps, a host of spins, and lots of fancy footwork. On top of that, your program must have *style,* or the scores for artistic impression will look more like grade point averages than the 5.8s and 5.9s you need to win the gold.

At one point in the TV coverage, the commentator mentioned that Katarina Witt planned to skate the same program with which she had won the gold medal six years earlier at the Calgary Olympics. He added that it was unlikely Ms. Witt would place near the top even if she gave a clean performance—the very best programs only six years ago simply weren't demanding enough for competition today.

Think about that. Are skaters today actually better than the skaters of a quarter century ago? Of course, but not because *Homo sapiens* has evolved to a higher state of athletic capability. Some of the improvements in today's performances, I'm sure, are a result of better skates and ice arenas. But the dominant reason for the improvement is that each year skaters raise their standards as they try to dethrone the latest national or world champion. Skaters 25 years ago could have included all those triple and combination jumps in their routines, but they didn't need to, so they didn't stretch themselves to master those feats.

In the book *Peopleware*, Tom DeMarco and Timothy Lister describe a similar difference in standards of performance among programmers who work for different companies. DeMarco and Lister conducted "coding wars" in which they gave a programming task to two programmers from every company participating in one of the contests. They found that the results differed remarkably from one programmer to the next, at times by as much as 11 to 1 in performance. That disparity is probably not too surprising. The surprising news is that programmers from the same company tended to produce similar results. If one did poorly, so would the other. Likewise, if one did well, both did well, even though the two programmers were working independently. DeMarco and Lister point out that the work environments at the companies could account for some of the difference in performance among the companies, but I believe the major reason for the 11 to 1 variance is that the acceptable skill level for the "average programmer" varies from one company to the next.

When was the last time you heard a lead say to a programmer, "I'm disappointed in you. You're doing just what you're expected to do"? Whether a company is aware of the phenomenon or not, its programmers have an average skill level, and once a programmer reaches that average level, the pressure to continue learning eases up even though the programmer might still be capable of dramatic improvement. The programmers are like those ice skaters 25 years ago—good enough. And leads tend not to spend time training people who are already doing their job at an acceptable level. They work with people who haven't yet reached that level.

Having a team of programmers who do what is expected is not good enough. An effective lead perpetually raises the standards, as coaches for Olympic-class skaters must do. As you raise the programming standards of your team, you'll ultimately raise the standards—the average—of your whole company.

FIVE-YEAR TENDERFEET

Occasionally I'll run across a programmer who after five or more years still works on the same project he or she was first assigned to. No problem with that, but in many cases I find that the programmer is not only on the same project but also doing the same job. If the programmer was assigned to the Microsoft Excel project to work on Macintosh-specific features, for instance, that's what he'll still be doing—as the specialist in that area. If the programmer was assigned to the compiler's code optimizer project, years later she'll still be working on that isolated chunk of code—again, as the specialist.

From a project standpoint, creating long-term specialists for specific parts of your product is a good idea, but creating specialists can backfire if you don't educate them wisely. You'll cripple those programmers and ultimately hurt your project and your company if you don't see to it that your specialists continue to learn new skills.

Suppose that Wilbur, a newly hired programmer, spends his first year becoming your file converter specialist and then spends the next four years writing filters to read and write the file formats of competing products. There's no question that such work is important, but Wilbur will have gained a year's worth of real experience and then tapered off, learning little else for the next four years. Wilbur would claim that he has five years of programming experience, but that would be misleading—he would in fact have one year's experience five times over.

If Wilbur had spent the last four of those five years working on other areas of the application, he'd have a much wider range of skills. If he had been moved around to work on different aspects of a mainstream

Windows or Macintosh application, for instance, he would have had an opportunity to develop all of this additional know-how:

- How to create and manipulate the user interface libraries—the menu manager, the dialog manager, the window manager—and all of the user interface gadgets you'd create with those libraries.
- How to hook into the help library to provide context-sensitive help for any new dialogs or other user interface extensions he incorporates into the application.
- How to use the graphics library to draw shapes, plot bit maps, do off-screen drawing, handle color palettes, and so on, for display devices with various characteristics.
- How to send output to printers, achieving the highest quality for each device, and how to make use of special features unique to each device, such as the ability of PostScript printers to print watermarks and hairlines.
- How to handle international issues such as double-byte characters, country-specific time and date formats, text orientation, and so on.
- How to handle the issues related to running an application in a networked environment.
- How to share data with other applications, whether the task is as simple as putting the data on the system clipboard or as complex as using the Windows Dynamic Data Exchange library or the Object Linking and Embedding library.
- How to write code that will work on all the popular microcomputer operating systems—MS-DOS, Windows, Windows NT, OS/2, and the Macintosh.
- . . .

You get the idea. These skills are easily within the grasp of any programmer who works on a Windows or Macintosh application for five years—provided that every new task contains an as-yet-unlearned element that forces a programmer to learn and grow.

Compare the two skill sets. If you were to start a new team, which Wilbur would you want more, the five-year file converter specialist or the Wilbur with one year's experience in writing file converters plus four more years' experience with the varied skills in the list? Remember, both Wilburs have worked for five years. . .

A lead's natural tendency when assigning tasks would be to give all the file converter work to Wilbur because he's the specialist in that area. It's not until the Wilburs of the world threaten to leave their projects for more interesting work that leads switch mental gears and start throwing new and different tasks their way.

But "if the specialists aren't doing the tasks they're expert in, wouldn't they be working more slowly on tasks they know less about?" Or to put it another way, "Don't you lose time by not putting the most experienced programmer on each task?"

If you view the project in terms of its specific tasks, the answer must be Yes, each task is being done more slowly than it could be done by a specialist. However, that little setback is more than compensated for when you look at the project as a whole. If you're constantly training team members so that they're proficient in all areas of your project, you build a much stronger team, one in which most team members can handle any unexpected problem. If a killer bug shows up, you don't need to rely on your specialist to fix it—anybody can fix it. If you need to implement a new feature in an existing body of code, any of many team members can efficiently do the work, not just one. Team members also know more about common subsystems, so you reduce duplicate code and improve product-wide design. The entire team has versatile skill sets.

Your team may be losing little bits of time during development as they learn new skills and gain experience, but for each minute they lose learning a new skill, they save multiple minutes in the future as they use that skill again and again. Constant training is an investment, one that offers tremendous leverage and tremendous rewards.

Don't allow programmers to stagnate.
Constantly expose each team member
to new areas of the project.

Reusable Skills

At Microsoft, when a novice programmer moves onto a project, he or she is typically given introductory work such as tracking down bugs and incorporating small changes here and there. Then gradually, as the programmer learns more about the program, the tasks become increasingly more difficult, until the programmer is implementing full-blown megafeatures. This gradualist approach makes sense because you can't very well have novices making major changes to code they know nothing about. My only disagreement with this approach is that the tasks are assigned according to their difficulty rather than according to the breadth of skills they could teach the programmer. As you assign tasks to programmers, keep the skills-teaching idea in mind. Don't assign successive tasks solely on the basis of difficulty; make sure that each task will teach a new skill as well, even if that means moving a novice programmer more quickly to difficult features. Even better, assign tasks at first that teach skills of benefit not only to your project but to the whole company.

In a spreadsheet program, for instance, tasks might range from implementing a new dialog of some sort to working on the recalculation engine. The skills a programmer would learn from these two tasks fall at two extremes: one skill has nothing to do with spreadsheets specifically, and the other historically has little use outside spreadsheet programming. Putting a programmer on the recalculation engine would be educational and would provide a valuable service to the project, but the skill wouldn't be as transferable as knowing how to implement a dialog would be. Learning how to create and manipulate dialogs could be useful in every project the company might undertake.

Creating a better "average programmer" means raising the standard throughout the company, not just on your project. You could assign programmers a random variety of tasks and ensure that team members would constantly learn, but you can do better than that. Analyze each task from the standpoint of the skills it calls upon, and assign it to the programmer who most needs to learn those skills. An experienced programmer should already know how to create dialogs, manipulate windows, change font sizes, and so on. She is ready to develop less globally useful—more specialized—skills such as the ability to add new macro functions to the spreadsheet's macro language. At some point, she'll know the program so well that in order to continue learning she'll have to move to extremely project-specific work such as implementing an even smarter recalculation engine.

A novice team member should be assigned a few tasks in which he must learn to create dialogs, followed by a few tasks that force him to manipulate windows, and so on. Deliberately assign tasks that cumulatively require all the general skills. That way, if the division should be reorganized and the programmer should find himself on another project, the skills he's learned will still be useful.

This is another example of a small system that produces greater results. Which specific work you assign to a novice programmer may not make much difference in the progress of your own project, but by first exposing a new programmer to a wide range of general skills that he or she can bring to any project, you make the programmer more valuable to the company.

When training programmers, focus first on skills that are useful to the entire company and second on skills specific to your project.

GIVE EXPERTS THE BOOT

If you constantly expose a team member to new tasks that call for new skills, he or she will eventually reach a point at which your project no longer offers room to grow. You could let the programmer's growth stall while your project benefited from his or her expertise, but for the benefit of the company, you should kick such an expert off your team. If you allow programmers to stagnate, you hurt the overall skill level of the company. You have a duty to the programmers and to the company to find the programmers positions in which they can grow.

Am I joking? No.

The tendency is to jealously hold onto the team's best programmers even if they aren't learning anything new. Why would you want to kick your best programmer off the team? That would be insane. . .

In Chapter 3, I talked about a dialog manager library that the Word for Windows group had been complaining about. Although I wasn't the lead of the dialog manager team then, I did eventually wind up in that position. And there came a point at which the main programmer on the team had reached a plateau: he wasn't learning anything new within the constraints of that project. Besides, he was tired of working on the same old code. He needed to stretch his skills.

When I asked whether he knew of any interesting openings on other projects, he described a position in Microsoft's new user interface lab in which he would be able to design and implement experimental user interface ideas. In many ways, it seemed like a dream job for the programmer, so I talked to the lab's director to verify that the job was a good opportunity for this programmer to learn new skills. The position looked great. In less than a week, the dialog team's best programmer was gone, leaving a gaping hole.

In these situations, you can either panic or get excited. I get excited because I believe that gaping holes attract team members who are ready to grow and fill them. Somebody always rises to the occasion, experiencing tremendous growth as he or she fills the gap. The dialog team's gap proved to be no different. Another member jumped headlong into the opening.

Occasionally I'd bump into the lab director and ask how the project was going. "Beyond my wildest dreams," he'd say. "We're accomplishing more than I ever imagined or hoped for." He had been expecting to get an entry-level programmer, but he'd gotten a far more experienced programmer, and his group was barreling along.

The dialog manager group with its new lead programmer was barreling along too. The new lead had just needed the room to grow, room that had been taken up by the expert programmer.

You might think that kicking your best programmer off the team would do irreparable harm to your project. It rarely works out that way. In this case, the dialog team experienced a short-term loss, but the company saw a huge long-term gain. Instead of a slow-moving user interface project and two programmers who had stopped growing, the company got a fast-moving user interface project and two programmers who were undergoing rapid growth. That outcome shouldn't be too surprising. As long as its people are growing, so is the company.

Don't jealously hold onto your best programmers if they've stopped growing. For the good of the programmers, their replacements, and the company, transfer stalled programmers to new projects where growth can continue.

The Cross-Pollination Theory—Dismissed

Occasionally I'll run across the idea that companies should periodically shuffle programmers around so that they can transfer ideas from one project to another. It's the cross-pollination theory.

The cross-pollination theory appeals to me because its purpose is to improve development processes within the company, but in my experience the cross-pollination practice falls short of its goal, and for a simple reason: it ignores human nature. Advocates of the theory assume that programmers who move to brand-new groups will teach the new groups the special knowledge they bring with them. But how many people feel comfortable doing that in a new environment? And even if a programmer would feel comfortable as an evangelist, how many groups would appreciate some newcomer's telling them what they should do? A new lead might feel fine propounding fresh ideas hours or days into the project, but nonleads? It might be years, if ever, before a programmer would feel comfortable enough to push his or her ideas beyond a narrow work focus.

Advocates of the cross-pollination theory assume that new people bring new knowledge into the group. In fact, that's backwards from what actually happens: new people don't bring their knowledge into the new group as much as they get knowledge from the new group. New people find themselves immersed in different, and possibly better, ways of doing things. And they learn. The primary benefit is to the person doing the moving. If that person can continue to grow on his or her current project, why cause disruption? Let the people who are stagnating move to other teams and learn more. Don't shuffle people around to other teams expecting them to spread the word. They usually won't.

THE NEW YEAR'S SYNDROME

Not all skills can be attained in the course of doing well-chosen project tasks. A skill such as learning to lead projects must be deliberately pursued as a goal in itself. The person must decide to be a good lead and then take steps to make it happen. It's proactive learning, as opposed to learning as a side effect of working on a task.

If you want your team members to make great leaps as well as take incremental daily steps to improvement, you must see that they actively pursue the greater goals.

The traditional approach to establishing such goals is to list them as personal skill objectives on the annual performance review. We all know what happens to those goals: except for a few self-motivated and driven individuals, people forget them before the week is over. Then along comes the next review, and their leads are dismayed to see that none of the personal growth goals have been fulfilled. I think we've all seen this happen—it's the New Year's Resolution Syndrome, only the date is different.

Such goals fall by the wayside because there are no attack plans for achieving them or because, if there are such plans, the plans have no teeth—just as those postmortem plans I spoke of in Chapter 4 had no teeth. Listing a goal on a review form with no provision for how it will be achieved is like saying "I'm going to be rich" but never deciding exactly how you're going to make that happen. To achieve the goal, you need a concrete plan, a realistic deadline, and a constant focus on the goal.

One way to ensure that each team member makes a handful of growth leaps each year is to align the personal growth goals with the two-month project milestones. One goal per milestone. That practice enables team members to make six leaps a year—more if there are multiple goals per milestone.

Improvement goals don't need to be all-encompassing. They can be as simple as reading one good technical or business book each milestone or developing a good habit such as stepping through all new code in the debugger to proactively look for bugs. Sometimes the growth goal can be to correct a bad work habit such as writing code on the fly at the keyboard—the design-as-you-go approach to programming.

Read Any Good Books Lately?

I read constantly to gain new knowledge and insights. Why spend years of learning by trial and error when I can pick up a good book and in a few days achieve insights that took someone else decades to formulate? What a deal. If team members read just six insightful books a year, imagine how that could influence their work. I particularly like books that transform insights into strategies you can immediately carry out. That's why I wrote both *Writing Solid Code* and this book as strategy books. But mine are hardly the first. *The Elements of Programming Style,* by Brian Kernighan and P. J. Plauger, was first published in 1974 and is still valuable today. *Writing Efficient Programs,* by Jon Bentley, is another excellent strategy book, as is Andrew Koenig's *C Traps & Pitfalls* for C and C++ programmers.

In addition to these strategy books, there are dozens of other excellent—and practical—books on software development, from Gerald Weinberg's classic *The Psychology of Computer Programming* to the much more recent *Code Complete,* by Steve McConnell, which includes a full chapter on "Where to Go for More Information," with brief descriptions of dozens of the industry's best books, articles, and organizations.

But don't limit yourself to books and articles that talk strictly about software development. Mark McCormack's *What They Don't Teach You at Harvard Business School,* for instance, may focus on project management at IMG, his sports marketing firm, and Michael Gerber's *The E-Myth* may focus on how to build franchise operations, but books like these provide a wealth of information you can apply immediately to software development. And don't make the mistake of thinking that such books are suitable only for project leads. The greenest member of the team can benefit from such books.

To ensure their personal interest in achieving such goals, I encourage team members to choose the skills they want to pursue, and I merely verify that each goal is worth going after:

- The skill or knowledge would benefit the programmer, the project, and the company. Learning LISP could be useful to an individual, but for a company such as Microsoft, it would be as useful as scuba gear to a swordfish.

- The goal is achievable within a reasonable time frame such as the two-month milestone interval. Anybody can read a good technical book in two months. It's much harder to become a C++ expert in that short a time.
- The goal has measurable results. A goal such as "becoming a better programmer" is hard to measure, whereas a goal such as "developing the habit of stepping through all new code in the debugger to catch bugs" is easy to measure: the programmer either has or hasn't developed the habit.
- Ideally, the skill or knowledge will have immediate usefulness to the project. A programmer might acquire a worthwhile skill, but if he has no immediate use for the new skill, he's likely to lose or forget what he's learned.

Such a list keeps the focus on skills that are useful to the individual, to his or her project, and to the company—in sum, it focuses on the kinds of skills a programmer needs in order to be considered for promotion. If the programmer can't think of a skill to focus on, choose one yourself: "What additional skills would this programmer need for me to feel comfortable about promoting him or her?"

Make sure each team member learns one new significant skill at least every two months.

Train Your Replacement

Programmers don't usually choose to pursue management skills unless they have reason to believe they're going to need those skills. Find the people who have an interest in becoming team leads, and help them acquire the skills they'll need to lead teams in the future. And remember, unless you plan to lead your current team forever, you need to train somebody to replace you. If you don't, you might find yourself in a tough spot, wanting to lead an exciting new project and unable to make the move because nobody is capable of taking over your current job.

IN THE MOMENT

A particularly good approach to identifying skills for your team members to develop is to set a growth goal the moment you see a problem or an opportunity. When I spot programmers debugging ineffectively, I show them a better way and get them to commit to mastering the new practice over the next few weeks. When a programmer remarks that she wants to learn techniques for writing fast code, I hand her a copy of Jon Bentley's *Writing Efficient Programs* and secure her commitment to reading it—and later discussing it. If I turn up an error-prone coding practice as I review some new code, I stop and describe my concern to the programmer and get him to commit to weeding the practice out of his programming style.

I'm big on setting improvement goals in the moment. Such goals have impact because they contain a strong emotional element. Which do you think would influence a programmer more: showing him code he wrote a year ago and asking him to weed out a risky coding practice or showing him a piece of code he wrote yesterday and asking him to weed out the practice?

I once trained a lead who would search me out every time he had a problem. He'd say, "The WordSmasher group doesn't have time to implement their Anagram feature, and they want to know if we can help out. What should we do?" The lead came to me so often that I eventually concluded he wasn't doing his own thinking. When I explained my feelings to him, he replied that he always thought through the possible solutions but didn't want to make a mistake. That was why he was asking me what to do. I pointed out that his approach made him seem too dependent and that we needed to work on the problem.

I understood the lead's need for confirmation, so I told him to feel free to talk to me about problems as they arose, on one condition: instead of dumping the problem in my lap, he was to

- Explain the problem to me.
- Describe any solutions he could come up with, including the pros and cons of each one.
- Suggest a course of action and tell me why he chose that course.

Once the lead began following this practice, my perception of him changed immediately and radically. On 9 out of 10 occasions, all I had to do was say, "Yes! Do it." to a fully considered plan of action. The few times I thought a different course of action made sense, I explained my rationale to him, we talked it over, and he got new insights. Sometimes I got the new insights. We'd go with his original suggestion if my solution was merely different and not demonstrably better.

This improvement took almost no new effort on either his part or mine, but the shift in his effectiveness was dramatic. We went from a relationship in which I felt as if I were making all his decisions to one in which I was acknowledging his own good decisions. My attitude changed from "this guy is too dependent and doesn't think things through" to "this guy is thoughtful and makes good decisions." His attitude changed too, from being afraid to make decisions to knowing that most of his decisions were solid. It didn't take too many weeks for our "What should I do?" meetings to all but disappear. He consulted me only for truly puzzling problems for which he couldn't come up with any good solution.

What caused this dramatic change? Was it a major revamping of this person's skills? No, it was a simple change in communication style provoked by my realization that he had become too dependent. A minor change, a major improvement.

Take immediate corrective action the moment you realize that an area needs improvement.

After-the-Fact Management

Note that I gave that lead on-the-spot feedback and a goal he could act on immediately. I didn't wait for the annual review. I don't believe the annual review is a good tool for planning personal improvement or achievement goals. In my experience such a delayed response to problems isn't effective—at least not unless the annual review also contains detailed attack plans for the goals. Another problem with using the annual review for improvement goals is that few leads are able to effectively evaluate anyone's growth over such a long period of time.

We've all heard stories about the review in which the manager brings up a problem with the programmer's performance that has never been mentioned before to justify giving the programmer a review rating lower than the programmer expected. In shock, the programmer stammers, "Can you give me an example of what you're talking about?" The manager stumbles a bit and comes up with something, that, yes, the programmer did do, or failed to do, but which sounds absurdly out of proportion in the context of the programmer's performance for the whole review period. "You've given me a low rating because of that?" Of course, it sounds ridiculous to the manager too, so she scrambles to come up with another example of the problem but usually can't because so much time has passed.

Then, of course, once the programmer leaves the meeting and has time to think about the review a bit, his or her reaction is anger. "Why didn't she *tell me* something was wrong, rather than waiting a year? How could I have fixed something I didn't even know was wrong?"

I've lost track of the number of times I've heard people say that about their managers.

What if professional football teams worked that way? What if coaches waited until the end of the season to tell players what they're doing wrong?

"Mad Dog, I'm putting you on the bench next season."

"Huh? What? I thought I played great," says Mad Dog, confused.

"You played well, but at each snap of the ball, you hesitated before running into position."

"I did?"

"Yes, you did, and that prevented you from catching as many passes as you could have. I'm putting you on the bench until something changes. Of course, this means that your yearly salary will drop from $5.2 million to $18,274. But don't worry, you'll still have your benefits—free soft drinks and hot dogs at the concession stand, and discounted souvenirs."

Mad Dog, particularly mad now: "If you spotted this, *why didn't you tell me earlier?* I could have *done* something about it."

"Hey, I'm telling you now, at our end-of-the-season contract review."

Sounds pretty silly, doesn't it? But how does it differ from the way many leads make use of the annual review?

Remember the lead I felt was too dependent and was not thinking things through? The common approach at most companies would be to wait until the end of the review period and note the problem on the review document:

```
Relies too much on other people to make his decisions;
doesn't take the time to think problems through.
```

Then, of course, after the confused exchange at the review meeting, the attack plan to fix the problem would be something like this:

```
I won't rely on other people to make my decisions for me;
I'll think my problems through.
```

That attack plan won't be effective because it is too vague. The plan doesn't describe what the person is to do, how he is to do it, or how to measure the results—the plan has no teeth. In all likelihood, the problem will still exist a year later, at the next review.

Personnel reviews, as I've seen them done, are almost totally worthless as a tool to promote employee growth. Don't bother with the new goals part of the review. Actively promote improvement by seizing the moment and aligning growth goals with your project milestones. Use the formal review to *document* employee growth during the review period—that's what upper management really needs to see anyway. Listing areas in which people could improve doesn't really tell upper management much. Documenting the important skills that people have actually mastered and how they applied those skills demonstrates constant growth and gives upper management something tangible with which to justify raises, bonuses, and promotions.

Don't use the annual personnel review to set achievement goals. Use the review to document the personal growth goals achieved during the review period.

THOROUGHLY KNOWLEDGEABLE

Most of the interviews I conducted at Microsoft were with college students about to graduate, but occasionally I interviewed a working programmer who wanted to join Microsoft. At first I was surprised to find that the experienced programmers who came from small, upstart companies seemed, in general, more skilled than the experienced programmers from the big-name software houses, even though the programmers had been working for comparable numbers of years. I believe that what I've been talking about in this chapter accounts for the difference. The programmers working for the upstart companies had to be knowledgeable in dozens of areas, not expert in one. Their companies didn't have the luxury of staffing 30-person teams in which each individual could focus on one primary area. Out of necessity, those programmers were forced to learn more skills.

As a lead—even in a big outfit that can afford specialists—you must create the pressure to learn new skills. It doesn't matter whether you teach team members personally or whether they get their training through books and technical seminars. As long as your teams continue to experience constant, unceasing improvement, the "average programmer" in your company will continue to get better—like those Olympic-class skaters—and that can only be good for your project, for your company, and ultimately for your customers.

HIGHLIGHTS

- Never allow a team member to stagnate by limiting him or her to work on one specific part of your project. Once programmers have mastered an area, move them to a new area where they can continue to improve their skills.
- Skills vary in usefulness from those that can be applied to any project to those that can be applied to only one specific type of project. When you train your team members, maximize their value to the company by first training them in the most widely useful skills and save the project-specific skills for last.

- It's tempting to hold onto your top programmers, but if they aren't learning anything new on your project, you're stalling their growth and holding the company's average skill level down. When a top programmer leaves the team for a new position, not only does he or she start growing again, but so does his or her replacement. A double benefit.
- To ensure that the skills of the team members are expanding on a regular basis, see that every team member is always working on at least one major improvement goal. The easiest approach is to align growth goals with the two-month milestones, enabling at least six skill leaps a year—which is six more per year than many programmers currently experience. If Wilbur, the file converter specialist, had read just 6 high-quality technical books a year, after his first five years of programming he'd have read 30 such books. How do you suppose that would have influenced his work? Or what if Wilbur had mixed the reading of 15 good books with the mastery of 15 valuable skills over that first five years?
- The best growth goals emerge from a strong, immediate need. If you find a team member working inefficiently or repeating the same type of mistake, seize the opportunity to create a specific improvement goal that the team member can act on immediately. Because such on-the-spot goals lend themselves to immediate action for a definite purpose, the programmer is likely to give them more attention than he would give to abstract goals devised for an annual review.