2

THE SYSTEMATIC APPROACH

I've been programming computers for almost two decades, so you might be surprised to learn that I don't use a word processor when I sit down to write technical documents or books such as this one. I write everything by hand on a pad of legal paper, and later I transcribe what I've written into a word processor for editing.

I'm obviously not computer-phobic, and writing the old-fashioned way with pen and paper certainly isn't easier than using a word processor. Nevertheless, that's what I do.

I discovered long ago that whenever I sat down to write using a word processor, I would get so caught up in editing every sentence the moment I wrote it that after a day's worth of effort I'd have written almost nothing. Editing was too easy, much easier than writing the next paragraph, and I naturally fell into the habit of doing the easy work. I had to do it sometime anyway, right? In reality, I was editing in order to procrastinate, and it worked all too well.

Once I realized I had been sabotaging my writing effort, I looked for a process I could use to get the results I needed: being able to write technical papers much more speedily. I tried to force myself not to edit as I wrote with the word processor, but I wasn't very successful. I needed a system in which writing would be easier than editing. That's when I stopped using a word processor to write my first drafts and went back to traditional longhand. I now use the word processor only for what it's especially suited for—editing what I've already written.

My new "writing system" solved my procrastination problem by getting me to focus on the writing part of writing.

The important point here is that adopting a trivial process or system can produce dramatic results. I now write five pages in the time it used to take me to write five paragraphs. Was this improvement the result of my becoming a more experienced writer? No. Was it because I worked harder and longer? Again, no. I became a more productive writer because I noticed that the tool I was using had drawbacks and I developed a new system for writing.

As you'll see throughout this chapter, the use of little systems can achieve amazing results. Once you grasp this concept and learn to apply it to your software projects, you can truly claim that you're working smart, not hard, and you can come that much closer to hitting your deadlines without the long hours and daily stress that seem to afflict so many software projects today.

Bad Coffee

A common problem for servers in coffee shops is remembering who's drinking regular coffee and who's drinking the decaffeinated stuff. A coffee shop manager with unlimited time and resources might send all the servers to Kevin Trudeau's Mega Memory seminar, where they'd learn to vividly imagine a calf with a hide that matches, say, the customer's paisley tie, so that seeing the tie at refill time would trigger the thought of the paisley calf—and de*caf* coffee. Most coffee shop managers take a much simpler approach to the problem, though: they just tell the servers to give you a different kind of cup if you order decaf. The server need only look at your cup to know what type of coffee you're drinking.

A trivial system for solving a common problem.

Now imagine a coffee shop that has a whole collection of such trivial "systems" that produce better results with little or no extra effort. Let's look at another example.

There are two coffee shops near my house. They have identical coffeemakers, they use the same supplier for their beans, and the servers in both places are college students. But one shop consistently brews great coffee, whereas coffee at the other shop is sometimes good, sometimes watery, sometimes too thick, and sometimes burned beyond drinking—you never know what you're going to get when you order coffee there.

Circumstances at the two shops are identical except for one seemingly insignificant detail: the shop that consistently serves great coffee has a short horizontal line embossed on the side of each of its coffee pots. That short line is part of a simple "quality system" that consistently produces good coffee. When a new employee first comes on duty at this shop, the manager pulls him aside and gives him a short lecture:

"Whenever you pour a cup of coffee and the level of coffee drops below this line," he says, pointing to the mark on the pot, "immediately start a new pot. Don't go on to do anything else before you start that new pot."

"What if it's really busy?"

"I don't care if the place is filled with Seattle Seahawks an hour after they've blown a Super Bowl game. Start that new pot before you give Mad Dog Mitchell the cup you've just poured."

The manager goes on to explain that by taking 15 seconds to start a new pot before the old one is empty, the server might make the current customer wait an extra 15 seconds but that the practice prevents the next customer from having to wait a full 7 minutes for a new pot to brew because the current pot ran out.

If you order coffee at the other coffee shop, it's not unusual for the server to reach for the pot only to find it empty, and you have to begin that 7-minute wait. Of course, sometimes you don't have to wait the full 7 minutes. To shorten your wait, some servers will watch until just enough coffee for one cup has brewed and pour you that cup. But for good coffee, you must let the entire pot of water drip through so that the initial sludge can mix with a full pot of hot water. If you pour a cup too early in the process, that cup will be so strong it will be undrinkable, and any other cups you pour from that pot will taste like hot water. That's one reason the quality of the second shop's coffee fluctuates. Depending on when your coffee is poured, you'll get sludge, coffee-colored hot water, or sometimes even normal coffee. And of course occasionally you'll get burned coffee—when the pot holds just enough coffee for one cup and there's not enough liquid to prevent the coffee from burning on the warmer as the water boils out.

The only difference between the two shops is that in one they make coffee when their pots get low and in the other they make coffee when their pots get empty. Their systems are so similar, yet they produce drastically different results, and *the results have nothing to do with the skill of the people involved.*

I wouldn't be talking about these coffee shop systems unless they made a point that related to software development. They do.

If I were to ask you if it mattered when in the software development process your team fixed bugs, provided the bugs were fixed before you shipped the product, what would your answer be? Would you argue that the team shouldn't focus on bugs until all the features have been implemented? Would you argue that bugs should be fixed as soon as they're found? Or would you argue that it doesn't matter, that it takes the same amount of time to fix a bug no matter when you get around to doing it?

If you thought that it doesn't matter when you fix bugs, you would be wrong, just as a coffee shop manager would be wrong if he thought it didn't matter exactly when his servers made new coffee. Possibly the worst position a project lead can find herself in is to be so overwhelmed by bugs that the bugs—not the goals—drive the project. If you want to stay in control of your project, one of your concrete goals must be to never have any outstanding bugs. To ignore this goal is to set a destructive process in motion, one I described in *Writing Solid Code.* There I noted that when I first joined the Microsoft Excel group, it was customary to postpone bug-fixing until the end of the project. I pointed out the many problems that approach created—the worst being the impossibility of predicting when the product would be ready. It was just too hard to estimate the time it would take to fix the bugs that remained at the end of the project, to say nothing of the new bugs programmers would introduce as they fixed old ones. And of course fixing one bug inevitably exposed latent bugs the testing group had been unable to find because the first bug had obscured them.

Concentrating on features and ignoring bugs enabled the team to make the product look much more complete than it actually was. But high-level managers would use the product and wonder why "feature complete" software had to spend six more months in development. The developers frantically debugging the code knew why. Bugs. Everywhere.

When a series of bug-ridden products ended with the cancellation of a buggy unannounced application, Microsoft was finally prompted to do some soul-searching. Here's how I summarized the results of that self-examination in *Writing Solid Code*:

- You don't save time by fixing bugs late in the product cycle. In fact, you lose time because it's often harder to fix bugs in code you wrote a year ago than in code you wrote days ago.
- Fixing bugs "as you go" provides damage control because the earlier you learn of your mistakes, the less likely you are to repeat those mistakes.
- Bugs are a form of negative feedback that keep fast but sloppy programmers in check. If you don't allow programmers to work on new features until they have fixed all their bugs, you prevent sloppy programmers from spreading half-implemented features throughout the product—they're too busy fixing bugs. If you allow programmers to ignore their bugs, you lose that regulation.
- By keeping the bug count near zero, you have a much easier time predicting when you'll finish the product. Instead of trying to guess how long it will take to finish 32 features and 1742 bug-fixes, you just have to guess how long it will take to finish the 32 features. Even better, you're often in a position to drop the unfinished features and ship what you have.

As I said in *Writing Solid Code,* I believe these observations apply to any software development project, and I'll repeat the advice I ended with there:

> If you are not already fixing bugs as you find them, let Microsoft's negative experience be a lesson to you. You can learn through your own hard experience, or you can learn from the costly mistakes of others.

Don't fix bugs later; fix them now.

When programmers fix their bugs matters a great deal, just as when servers make new coffee matters a great deal. Requiring programmers to fix their bugs the moment they're found introduces a small system into

"Unacceptably Slow"

Some groups at Microsoft have broadened the traditional concept of what constitutes a bug to include any flaw that has to be addressed before the product is shipped. In these groups, a feature could be considered buggy simply because it was unacceptably slow. The feature might function without error, but the fact that it would still require work before it was ready to ship would be considered a bug.

If they have a policy of fixing bugs as they're found, groups that define bugs so broadly are forced early on to define what is and is not "unacceptably slow." In fact, they're forced to define all their quality bars early on. The result: programmers don't waste time rewriting unshippable code, at least not more than once or twice, before they learn what quality levels they're aiming for.

The drawback to this approach is that some programmers might waste time writing complex, efficient code, say, when straightforward code would do just fine. But such a tendency could be easily detected (and corrected) in regular code reviews.

the development process that protects the product in many ways. In addition to the benefits I described in *Writing Solid Code,* the system produces these good side effects:

- The constant message to programmers is that bugs are serious and must not be ignored. This point is emphasized right from the start of the project and receives perpetual reinforcement.
- Programmers become solely responsible for fixing their own bugs. No longer do the careful programmers have to help fix the bugs of the sloppy programmers. Instead, the careful programmers get to implement the features the sloppy programmers can't get to because they're stuck fixing bugs in their earlier features. The effect is that programmers are rewarded for being careful. Justice!
- If programmers are fixing bugs as they're found, the project can't possibly have a runaway bug-list. In fact, the bug-list can never sneak up and threaten your project's timely delivery. How could it? You're always fighting the monster while it's little.
- Finally, and perhaps most important, requiring programmers to fix their bugs as they find them makes it quite apparent if a particular programmer needs more training—his or her schedule starts slipping, alerting you to a problem that might otherwise go unnoticed.

Whether you realize it or not, your development process is filled with little systems that affect the quality of the product and the course of the project. That coffee shop manager with the mark on his pot understood the power of developing a system and used that power to his advantage. You can do the same with your projects, coming up with little systems that naturally give you the results you want.

Actively use systems that improve the development process.

The E-Mail Trap

Electronic mail is a wonderful tool. I can't imagine working efficiently without it. Having said that, I have to add that when it isn't handled wisely, e-mail can destroy your productivity.

I've found that newly hired programmers allow e-mail to constantly interrupt their work. I don't mean that they're sending too much e-mail; I mean that they're stopping to read every new message as it arrives. New employees don't get much mail that they have to respond to; most e-mail they receive consists of passive information that's just making the rounds. You know, things like the closing price of Microsoft stock, what Spencer Katt had to say about this or that company that week, the business news wire releases for the day, and so on. This stuff trickles in throughout the day.

New employees tend to leave their e-mail readers running and to stop every 5 minutes to check out the latest "blip." They never get any work done because their entire day is broken into 5-minute time slices.

To combat this tendency, I routinely tell new hires to respond to their e-mail in batches: "Read it when you arrive in the morning, when you return from lunch, and just before you leave for the day." That tiny system for e-mail reading—governing only *when* they read their mail—allows developers to get their work done because the work is no longer subject to constant interruption.

The developers are reading the same number of messages; that hasn't changed. They're just reading those messages more efficiently and doing their other work more efficiently as a consequence.

LEANING ON CRUTCHES?

I've described using such trivial systems to programmers and leads on many occasions, and every once in a while I'll run into somebody who thinks systems are a bad idea. Such a person usually maintains that systems are a crutch: "You're cheating those people out of a learning experience. The next job they go to, they'll not have learned anything."

As much as I believe in using systems, I do take seriously the concern these people express. As you'll see throughout the book, I believe you must continually work to improve the skills of each member of your team. I just don't believe that the project has to be a casualty of that learning experience.

The beauty of setting up a system is that team members don't have to immediately grasp the rationales behind the system in order for it to work. But don't keep the rationales behind your system a secret. I'd urge you to do just the opposite: fully describe the thinking behind the system you set up and what you expect the system to accomplish. In time, the team members will begin to appreciate the thinking behind the system and probably start to add improvements that will make it even more effective. Encourage your team to understand and improve the systems you put in place.

Don't use systems in lieu of training.
Use systems and explain why you
expect them to work.

PLEASE PASS THE POPCORN

Well-designed systems for working are valuable because they can nudge people into doing what's best for the product. A strategy is valuable because it condenses a body of experience into a simple attack plan that anybody can immediately understand and act on. A collection of such strategies can catapult an individual (or a team) to a higher level of productivity, quality, or whatever it is that the strategies focus on.

As a lead, you should encourage your team to share the strategies they've found to be effective in achieving project goals and priorities. My highest priority for software products is that they always be bug-free, for instance, but as we all know, achieving that state is much easier to talk about than to accomplish. Even so, I can look at different programmers and see that some have much lower bug rates than others. Why? The programmers with lower bug rates have a better understanding of how to prevent bugs and of how to effectively find any bugs that do creep into their code. They have better strategies for writing bug-free programs.

To encourage developers to come up with strategies that result in bug-free code, I have them ask themselves two questions each time they track down the cause of a bug:

How could I have prevented this bug?

and

How could I have easily (and automatically) detected this bug?

As you can probably imagine, any programmer who habitually asks these questions begins to spot error-prone coding habits and starts to weed them out of his or her coding practice. Such a programmer also begins to discover better strategies for finding bugs. Of course, most programmers would, in time, develop such strategies anyway, but by constantly asking those two questions, they more rapidly—and consciously—learn how to prevent and detect bugs. As with anything else, if you systematically focus on an area, you get better results than you would if you haphazardly wandered over to it every now and then. There's no magic here.

As a lead, you can ask yourself similar questions for each problem you encounter:

How can I avoid this problem in the future?

and

What can I learn from this mistake/experience?

These are critical questions that successful leads habitually ask themselves as they actively improve their skills. Some leads forever repeat the same mistakes because they fail to ask these questions and act on their findings.

Of course, the quality of the questions you ask will determine the quality of the strategies you derive from them. Consider these two questions:

Why do our schedules always slip?

vs.

How can we prevent schedule slipping in the future?

Although the questions are quite similar, would you give the same answers to both? I doubt it. I doubt it because the first question gets you to focus on all the reasons your schedules slip: you have too many dependencies on other teams, your tools are lousy, your boss is a bozo and always gets in your way, and so on. The second question gets you to focus on what you can do to prevent slipping in the future: reducing your dependency on other groups, buying better tools, establishing a new work arrangement with your boss. The questions focus on different aspects of the problem—one on causes, the other on prevention—so the quality of the answers for the two is different. The first question elicits complaints; the second question elicits an *attack plan.*

Even if the questions you ask yourself have the right focus, they may not be precise enough to elicit effective strategies. Just as goals gain power as you increase their detail, questions become more powerful as you increase their precision. Let's take a look at another question:

How can we consistently hit our ship dates?

Some leads who asked that question might decide to pressure their teams to work overtime by threatening them. Others might decide to bribe their teams to work overtime with bonuses or free dinners or by projecting blockbuster movies at midnight and passing out buckets of popcorn. (Don't laugh. It has happened.)

But suppose those leads had asked a more precise, and in my opinion more beneficial, question:

How can we consistently hit our ship dates, without having developers work overtime?

The leads would obviously get a different kind of answer because threats or midnight movies wouldn't answer the requirements posed by this more precise question. The leads would have to toss out any "solution" that called for getting their teams to work overtime. They'd be forced to search for other possibilities. They might decide that to hit their ship dates without demanding overtime work they'd have to hire more developers. That's a possibility, but not one that companies usually like to consider, at least not until all other approaches have been exhausted. To eliminate that unacceptable solution from consideration, I'll increase the precision of the question even further:

How can we consistently hit our ship dates, without having developers work overtime, and without hiring additional people?

The question now eliminates two undesirable solutions, forcing leads to think more creatively and, not incidentally, to focus more on the work itself. Maybe a lead would decide that it wasn't so critical, after all, that his team write all the code in the product: he could hire a short-term consultant, or the team could use a code library another team might have offered them just the month before, or they could even buy a fully documented commercial library, which could cut their development time dramatically. Maybe they'd decide to cut features that, upon reflection, they'd see wouldn't really add much value to the product.

The Ideal Question

As we'll see throughout this book, there are numerous ways to increase productivity without resorting to 80-hour weeks. When you ask questions to elicit solutions, keep in mind that question from Chapter 1: What am I *ultimately* trying to accomplish? No lead is ultimately trying to get people to work overtime; most are in fact ultimately trying to get more work done in a shorter period of time.

The simplest technique for zeroing in on the best question to ask is to envision how you would ideally like your project to run and to tailor your question so that it reflects that ideal. Wouldn't your ideal project be one in which you made perfect estimates, you hit every feature milestone, nobody worked overtime, and all concerned thoroughly enjoyed their work? That's a lot to ask, but if you tailor your questions to reflect that ideal, you'll come up with the solutions that will bring you closer to those goals.

The point is that by asking a more precise question, one that takes into account the results they'd ideally like to see, leads force themselves to weed out all the less than ideal solutions—the ones they might have glommed onto simply because they were the first solutions that presented themselves. Asking increasingly detailed questions stimulates the thinking process that leads to inventive solutions.

Ask detailed questions that yield strategies and systems that help to achieve your ideal project goals.

GOTOS HAVE THEIR PLACE

As you go about creating and promoting strategies, regularly remind the development team that the strategies are not rules that are meant to be followed 100 percent of the time. You want to be sure that people are thinking about what they're doing, not blindly following a set of rules even when those rules don't make sense.

One coding strategy that many programmers treat as an ironclad rule is "Don't use goto statements." But experienced programmers generally agree that there are a few special scenarios—mostly dealing with complex error-handling—in which using goto statements actually improves the clarity of code. When I see that a programmer has implemented that kind of error-handling code, scrupulously avoiding gotos, I usually raise the issue with the programmer.

"Did you consider using gotos to improve this code?" I ask.

"What? Of course not! Gotos are evil and create totally unreadable spaghetti code. Only incompetent programmers use gotos."

"Well, there are a few cases in which using gotos can make sense," I tell the programmer. "This is one of those cases. Let's compare your code to an implementation that uses a goto statement." I hand the programmer the goto version. "Which implementation is easier to read and understand?"

"The goto version," the programmer will usually reluctantly admit.

"So which implementation will you use in the future?"

"Mine, because it doesn't use any gotos."

"Wait, I thought you just agreed that the goto version was easier to read and understand."

"It is easier to read and understand, but using gotos can cause the compiler to generate less than optimal code."

"Let's assume that you're right, that the compiler generates some less than optimal code in this function. How often would this coding scenario show up?"

"Not very often, I guess."

"And which is a higher coding priority for the project, code clarity or a questionable efficiency gain?"

"Code clarity."

"So which version is easier to read and understand *and* follows our project priorities?"

At this point there is usually a long pause.

"But gotos are *bad*," the programmer blurts out in a last, pitiful protest.

I'll be the first to admit that there aren't many places in which using gotos actually clarifies the code; you can be sure that whenever I'm reviewing code and I see a goto, alarms start going off. I am not pro-goto—the presence of a goto usually does indicate a quick and dirty

Show Me Code!

Perhaps the most thorough discussion ever published about the pros and cons of using gotos can be found in Chapter 16 of Steve McConnell's *Code Complete*. In addition to showing those instances in which the judicious use of a goto can actually improve code, McConnell fully delineates the arguments against and for gotos and goes on to show how often the goto debate is phony. He finishes up with a list of articles that have exhaustively covered the use of gotos, including Edsger Dijkstra's original letter to the editor on the subject and Donald Knuth's example-rich "Structured Programming with go to Statements." As McConnell points out, "[the goto debate] erupts from time to time in most workplaces, textbooks, and magazines, but you won't hear anything that wasn't fully explored 20 years ago."

design hacked together while the programmer sat in front of the keyboard with a sugar buzz. But while I'm generally against using gotos, I'm even more against blindly following rules when they don't make sense and actually work to the detriment of the product.

That's the major drawback to strategies. If you push them as inviolable rules, you risk having team members do stupid things.

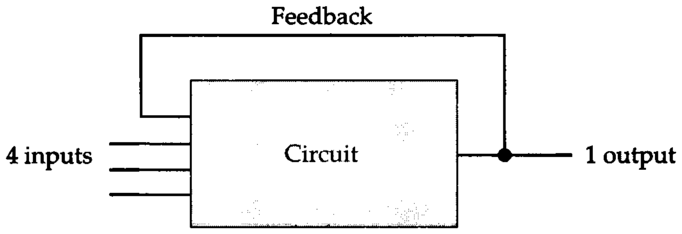
I know instructors mean well when they advise programmers not to use gotos, but I wish they would explain that gotos should be used rarely, instead of never. Even better, I wish that they'd demonstrate those few cases in which using gotos actually makes sense—it's not as if there are dozens of scenarios they'd have to cover. The problem, I think, is that many instructors were taught that gotos should never be used and they pass this advice on with ever-growing fervor. The mere presence of a goto is enough for some instructors (and programmers) to declare the code terrible, just as any form of nudity is enough for some people to proclaim a film immoral.

There are very few programming strategies that should be enforced as rules, and you need to make that clear. Otherwise, you may end up with developers blindly following a rule in situations in which it doesn't make sense. This disclaimer certainly applies to all the strategies in this book.

Don't present strategies as ironclad rules; present strategies as guidelines to be followed most of the time.

Feedback Loops

Electrical engineers use the concept of positive and negative feedback loops to describe the characteristics of a particular type of circuit, one in which the output of the circuit is fed back as an input to that same circuit. Here's a picture.



With the output contributing to its own result, such circuits behave in one of two ways: the output amplifies itself, so that the stronger it is, the stronger it gets; or just the opposite occurs, so that the stronger the output is, the weaker it gets. Feedback loops in which the output amplifies itself are known as positive feedback loops, and those in which the output weakens itself are known as negative feedback loops. From this admittedly simplified description of the two types of loops, it might seem that positive feedback loops are great because they leverage their own power whereas negative feedback loops are worthless because every time the output gets stronger, the effect is counteracted. In fact, negative feedback loops are far more useful than positive loops.

If you've ever been in an auditorium and heard a speaker and a microphone together cause an ear-shattering screech that could wake Elvis, you've been the victim of a positive feedback loop. The microphone has picked up and reamplified its own output, driving the amplifier into overload. That's the common problem with positive feedback loops: they typically overload themselves.

A negative feedback loop would take a high output and use it to reduce the loop's future output. Imagine welding the brake pedal of your car to the accelerator: step on the gas a bit, and the brakes go on a bit to counteract the acceleration; floor the gas, and you floor the brakes too. The stronger the output, the harder the circuit counteracts it. Such behavior may sound as useless as going into overload all the time, but negative feedback loops don't need to completely dominate the output; they just need to exert enough force to regulate and stabilize the circuit.

I've been talking about electrical circuits, but you'll find feedback loops in all sorts of systems, whether systems for personal relationships or for software development. Some of the feedback loops develop without conscious intention, and others are designed, but whatever their origins, you can achieve greater control over your project by becoming aware of feedback loops and making deliberate use of them.

Bugs, for example, are a common "output" of writing code. Wouldn't it be wonderful if you could design a negative feedback loop into your development process so that whenever the bug count grew, something would counteract that growth with equal force? We've already talked about exactly such a feedback loop:

Require that programmers fix their bugs the moment they're found.

If a programmer's code never has bugs, the requirement that bugs be fixed the moment they're found will never affect her and she can happily implement new features. But if a programmer writes code that's riddled with bugs, the requirement will kick in in full force, pulling that programmer off the implementation of new features and back to work on bugs, preventing her from spreading sloppy work throughout the program. The more bugs the programmer has, the harder the brakes are applied. The requirement that bugs be fixed immediately implements a negative feedback loop designed to keep the product bug-free at all times. And, of course, the practice gives you all those other benefits I mentioned earlier in the chapter—the relative ease with which recent bugs can be fixed, the speed with which programmers learn from fresh mistakes, easier prediction of project completion dates, and so on.

Negative feedback loops can hurt as well as help, though. Do you remember that lead I talked about in Chapter 1, the one who required his team members to submit status reports, attend status meetings, and then write follow-up reports on any insights they had come up with during the meetings? That lead was trying to get as much good information from the team as he could. Unfortunately, he'd set up a negative feedback loop that thwarted a desirable output. He wanted to hear any ideas his team members might come up with to solve a problem, but by asking them to write up those thoughts in reports, he discouraged them from saying anything. His system made people clam up—the more you spoke, the longer the report you had to write. Nobody liked writing those reports, so they learned to keep quiet. Just the opposite of what the lead was hoping for. Backfire.

You must also be careful not to unwittingly set up destructive *positive* feedback loops. If you base raises and bonuses on the number of new lines of code programmers write—and rewriting bad code doesn't count—don't be surprised if the programmers, over time, develop the

Negative Feedback Is Not Negative Reinforcement

Don't confuse negative feedback with negative reinforcement. I think of negative reinforcement as scolding, berating, or threatening an employee—like whipping a horse to get it to do what you want. Or, if an employee steps out of line, WHACK, giving him or her a solid dose of negative reinforcement to discourage stepping out of line in the future.

That kind of management style is reprehensible and certainly *not what I'm advocating*. Think about the negative feedback requirement that programmers fix their bugs as they're found. A programmer shouldn't be anxious about having to fix his bugs as they're reported. The requirement might have put him in a position he doesn't like—being stuck on the same feature for days on end—but that's very different from filling him with a sense of dread. The goal is to have the right things happen easily and naturally, without personal distress—not to assert who is boss or to put the employee in his or her place.

Many years ago at Microsoft, there were a couple of leads who, when a project was not running smoothly, would round up the development team and proceed to tell them that they were the worst programmers at Microsoft, that they weren't worthy of calling themselves Microsoft programmers, and other such nonsense. I'm not sure what those leads were trying to accomplish, but if their goal was to get the teams to rally and try harder, they picked a pretty strange way of doing it. As I'm sure you can imagine, those leads only succeeded in angering and depressing their development teams. Furthermore, in every case of which I was aware, the problems with the project were management related—the projects had no clear focus or were simply too ambitious. The programmers on those projects weren't any better or worse than other programmers in the company, and berating them didn't change anything for the better—only for the worse.

habit of sticking with their clunky first-draft code and patching flawed designs with new code instead of doing badly needed rewrites. You might intend the bonus to be an incentive for programmers to be more productive, but the long-term result would probably be a company full of programmers who are satisfied with slapped-together implementations.

I hope you'll take two points away from this discussion. First, whenever you design a new system, try to include beneficial negative feedback loops that help to keep the project on track. Second, consider the long-term effects of any feedback loops you decide to employ; make sure there are no feedback loops that can ultimately cripple the effort.

Deliberately use negative feedback loops in your systems to achieve desirable side effects.

Beware of feedback loops that create undesirable side effects.

THE SIMPLER, THE BETTER

Finally, make sure that the systems and strategies you come up with are easy to understand and follow.

Consider some of the systems I've covered: writing in longhand, using two kinds of coffee cups, watching a line on a coffee pot, reading e-mail in batches, fixing bugs the moment you find them. These systems are trivial, not hulking processes that will bog down the whole operation.

One tendency at workplaces is for simple processes to blossom into time-consuming busy-work because people get caught up in creating processes instead of working on the product. Having programmers ask, "How could I have prevented this bug?" is simple. Taking that system a step further and asking every programmer to write a "prevention report" for every bug he or she encounters is altogether different. All of a sudden the systematic asking of a simple question has turned into a cumbersome process. Such process growth is as natural as the growth of brambles, and you must actively keep that growth cut back.

Remember, the overall goal is to stay focused on improving the product, not on fulfilling process requirements. You want to gain the benefits that systems can provide and jettison the drawbacks. Well-designed systems and appropriately applied strategies accomplish both of these goals.

HIGHLIGHTS

- Simple work systems can produce dramatic results. Take a good look at the processes your team members are already following. Are there problems with those processes? Are they too time-consuming? Too error-prone? Are they frustrating and counterproductive in some way? If they are, look for simple changes you can make to improve those processes.
- As you put systems in place, explain the purposes behind them so that the development team can understand what aspect of the product the systems are meant to improve. This openness will educate team members over time and also enable them to intelligently improve the systems and create new, better ones.
- Refine the questions you ask as you look for solutions to problems. Develop the ability to ask precise questions to increase the quality of your answers. Unfortunately, it's not enough to be precise. A precise but wrong question will get you a bad answer. Be sure the question you ask focuses on what you're ultimately trying to achieve, on your ideal solution. Don't ask, "How can we get programmers to work longer hours?" Ask, "How can programmers get more done in less time?"
- The more appealing or effective a strategy is, the more people on your team will want to treat it as an ironclad rule. Remind your team that even the best strategies don't apply to every situation. "Avoid using gotos" is a strategy that can lead programmers to write more readable code. But you should encourage programmers to see that they should set aside even this strategy when avoiding gotos would make the code less readable.

- Whenever you create a feedback loop, be sure to consider the side effects and the long-term effects. The best feedback loops enhance the desirable aspects over time while simultaneously reducing the negative effects.