8

THAT SINKING FEELING

When projects start slipping, the first two actions leads often take are the easy, obvious ones: hire more people, and force the team to work longer hours. These may seem like reasonable responses, but in fact they're probably the worst approaches leads can take to turning around a troubled project.

Imagine a sixteenth century merchant galleon crossing the Atlantic Ocean from the Old World to the New World. When the galleon is far out in the ocean, the first mate notices that the ship is taking on water and alerts the captain. The captain orders members of the crew to bail water, but despite their efforts, the water continues to rise. The captain orders more crew members to bail water—to no avail. Soon the captain has the entire crew bailing water in shifts, but the water continues to rise. . .

Realizing that he has no more sailors to call on, and with the ship continuing to take on water, the captain orders all crew members to bail ever longer hours, their days and nights becoming nothing but bailing water, collapsing from exhaustion, waking up, and going back to bailing. It works. The sailors are not only able to prevent the water from rising, but they're able to make headway, bailing water out faster than it's coming in. The captain is happy. Through his brilliant management of human resources, he has prevented the ship from sinking.

At least for the first week.

Soon the crew members get bone weary and bail less water than they did when they worked in shifts and were well rested. The ship again starts taking on more water than they can bail out. The first mate tries to convince the captain that he must allow the crew members to rest if he wants them to be effective. But because the ship is sinking the captain rejects all talk of giving the crew a break. "We're *sinking*. The crew *must* work long hours," the captain shouts. "*We—are—sinking!*"

The water continues to rise and the ship eventually sinks, taking everybody with her.

Could there have been a better approach to saving that ship than putting all the crew members on the bailing task and then forcing them to work long, hard hours? If you were on a ship that was taking on water, what would you do? I can tell you what I'd do: *I'd search for the leaks.* Wouldn't you?

This is such an obvious point, but why then do so many leads run their projects as if they were sinking ships? When a project starts to slip, many a lead will first throw more people onto the job. If the project continues to slip and the lead can't get more people, he or she will demand that the developers put in longer hours. Just as that ship captain did. The project can be waist-deep in water, but the lead won't stop to look for and fix the leaks. Fortunately for their companies, most project teams can bail water slightly faster than it comes in, and they end up shipping their products, but often not without an enormous amount of misplaced effort.

In Chapter 1, I described a user interface library team that had been working 80-hour weeks for more than a year, with no end in sight. Water was gushing in on that project, but nobody stopped to look for leaks.

The team was fully staffed, and they were working 12-hour days, seven days a week. What more could they do? But as I pointed out in Chapter 1, that team was spending most of its time on work they shouldn't have been doing. They were ignoring what should have been their primary goal: *to provide a library that contains only functionality that is useful to all groups who will use the library*. That was a leak.

In Chapter 3, I talked about a dialog manager team that was working hard to speed up their library for the Word for Windows team. Despite all their hard work, they kept falling short of the quality bar for speed that the Word team had set. Word's swapping hack that kicked out all "unnecessary" code segments was kicking out every byte of the library code, so that physically reloading the code, to say nothing of executing the code, took more time than Word's quality bar allowed. But nobody was looking at load issues. The dialog manager team members were focused on optimizing the code to make it run faster.

And in Chapter 5, I described the Excel team's working 80-hour weeks to meet an unrealistic and demoralizing schedule.

In all of those cases, the need to work long hours should have been a red flag, a clear indication that something, somewhere, was seriously wrong. Unfortunately, many leads take the two obvious steps when projects start to slip their schedules—hiring more people and demanding longer hours—instead of looking for the causes of the schedule slips.

HAVE A LIFE

As I've said, for several years at Microsoft, my job was to take floundering projects and make them functional again. In every case, the team members had been working long hours, seven days a week, in a desperate attempt to catch a ship date that was moving ever further away. Team morale was usually low, and often programmers had come to detest their jobs.

On my first day as the new lead, my initial actions were always to put a stop to the long hours and start looking for the causes of the slipping schedule. I would walk down the halls in the early evening and kick people out. "Get outta here. Go have a life."

Programmers would protest: "I can't leave—I'm behind on this feature."

"That's OK," I'd say. "The entire team has been working insane hours for nearly a year, and all that effort hasn't kept the project from regularly slipping. Working long hours won't bring this project under control. There's something fundamentally wrong here, something we need to find and fix, and continuing to work long hours is not going to help us find the problem. Go home. Get some rest. We'll look for the problem first thing tomorrow."

At first the team members would think I was joking. The message they had been getting—in some cases for more than a year—was work harder, longer hours, and I was telling them to go home while the sun was still out. They thought I was nuts. If the project was slipping so badly now, they thought, what would it look like if they stopped working those long hours?

But over the next few weeks, I'd hit the project with all the strategies I've described in the first seven chapters of this book. I'd put a stop to unnecessary reports and meetings and all other unnecessary interruptions. I'd toss out the existing task-list–driven schedule and replace it with a win-able schedule made up of subproject milestones of the type I've described in Chapter 5, cutting all nonstrategic features in the process. I'd promote the attitudes I've presented in Chapter 7, such as the attitude that it's crucial that the team fix bugs the moment they're found. I'd make sure that the project goals were clear and that the programmers understood that one of my goals as a lead was to help create large blocks of time during the day for them to work uninterrupted. I'd do all of the things I've encouraged you to do. A hard month or two later, the team would hit their first milestone, as planned, but they'd do it without working 80-hour weeks. They'd have their first win. In the following months, hitting those subproject milestones would get progressively easier as new work skills became habits.

If your project is slipping, something is wrong. Don't ignore the causes and demand long hours of the team members. Find and fix the problems.

THE COMMITMENT MYTH

Some teams work long hours, not to meet an ever slipping schedule, but because an upper-level manager demands that they work 80-hour weeks, believing that development teams must work long hours to get products out the door. When such a manager sees a team working 40-hour weeks, his or her immediate interpretation is that the team is not committed to the company. If you point out that the team hits all its drop dates, the upper-level manager will counter with the statement that the team must be padding its schedules with gobs of free time. That same manager will hold up a team whose members work 80-hour weeks as an example for other teams to follow. "This team shows commitment!" If the team isn't hitting its deadlines, well, that's just because the project's schedule is unattainable, just as a schedule should be if you want programmers to work as hard as possible.

Obviously, I disagree with that point of view. If I held that view, I would have to conclude that the user interface library project, the dialog manager project, and the Excel project were model projects to be emulated. And I'd have to conclude that any team who had concrete goals and objectives, who focused on strategic features, who constantly invested in training, and who as a consequence always hit their drop dates while working efficient 40-hour weeks was a team who were screwing up.

It sounds silly when I put it that way, but that's effectively what that manager is saying when he sees a team working only 40-hour weeks and demands that the lead force the team members to put in more hours: "This is *not* a company of clock-watchers. You tell your team they're expected to put in more hours. I want to see some commitment!"

What nonsense. Managers like that praise the teams who work inefficiently and think the worst of the teams who work well. Compare such a manager with a manager who looks at a 40-hour-per-week team and is grateful that at least one project is running smoothly. That manager asks the team what they're doing to achieve such success and works to get other teams to duplicate that success.

Why such opposite reactions to the same event? In a word, attitudes.

The two upper-level managers respond differently because their primary attitudes about projects that run smoothly are polar opposites: one manager assumes that teams who work only 40-hour weeks and

who consistently meet their schedules are doing something *wrong*; the other type of manager assumes those teams are doing something *right*. Either manager could be mistaken in the case of a particular project, but what good does it do to start out assuming the worst of a team?

Just as some leads ask first for long hours instead of looking for the real problem and then solving it, some upper-level managers have glommed onto that same uninventive approach, believing that long hours are good for the project and the corporate culture. Such managers forget that the business purpose of a development team is to contribute value to the company. A team can contribute value in numerous ways: reducing their cost-of-goods and thereby increasing the profit per box shipped, writing shareable code that saves development time, and so on. A manager who demands long hours focuses on one obvious way it might seem that programmers can add value to the company: giving the company all of their waking—and some of their sleeping—time.

It might seem logical that having the programmers work all of those hours would enable them to finish the product sooner. Unfortunately, it doesn't work that way, not in software development. If the company made widgets and managers demanded that workers run the widget-making machines for three extra hours every day, the company would get three hours' worth more of widgets—added value. There's a direct correlation between the number of hours worked and the amount of product produced, a correlation that in my experience doesn't exist in software development.

If upper management pressures programmers to put in 12-hour days, working, say, from 10 o'clock in the morning to 10 o'clock at night,

Don't Blame the Programmers

I've been picking on the user interface library and dialog manager projects, but the problems with those projects and with the Excel project were not the programmers. In all of these cases, the programmers were working hard, trying to do their best in a frustrating situation. It's easy to make the mistake of blaming the programmers when a project is slipping and not running smoothly, but if the entire team is in trouble, that indicates a management problem.

the programmers might leave the office three hours later than they would otherwise; but consider what actually goes on during those three extra hours.

Take those twelve hours, and subtract one hour for lunch and another hour for dinner since 10 o'clock is rather late to work without stopping to eat. Factor in the natural tendency of programmers who regularly work 12-hour days to fit other activities into their work schedules, such as taking an hour each day to jog in the park or work out at the health club. That leaves nine of the twelve hours for actual work. And since programmers who work 12-hour days don't feel they have time outside work, they wind up taking care of other personal business at the office. I've seen programmers working through their stacks of unpaid bills, writing checks and licking envelopes. I've seen programmers practicing their piano skills on keyboards they keep in their offices. I've seen programmers playing in the halls with other team members, everything from group juggling to "hall golf."

People who work 12-hour days rarely put in more than the standard eight work hours they'd put in if they worked a normal 9-hour day, such as the traditional 8 to 5 workday. A programmer who works 12-hour days might actually get some work done between 8 o'clock and 10 o'clock at night, making it appear to some managers that long hours do result in added productivity, but those two hours actually just make up for dinner and some of the other personal time the programmer spent earlier in the day.

Sometimes a programmer will actually get more than eight hours of work done when he or she stays late—mainly when driven, being kept awake by thoughts of an elusive bug or a feature that's almost finished. The desire to find a resolution keeps the programmer focused on the problem. But in such a case, the programmer will tend to stay late even without pressure from upper management.

As a lead, one of your jobs is to protect the team members from those upper-level managers who think that forcing team members to work long hours is going to be productive. It won't be easy, but you've got to stand firm and fight such demands, explaining to those upper-level managers why their demands will only hurt the project. When upper-level management demands long hours of teams, it's a lose-lose

situation for the lead: you have to either fight management or hurt the team. Personally, I'd rather fight upper-level management than force team members to do something I'm fundamentally opposed to, but thankfully, I haven't had to fight many of those battles. Most of the upper-level managers I've worked for at Microsoft and elsewhere have understood that demanding long hours of the team was a misguided and inefficient approach to increasing productivity.

Beware of the misguided belief that long hours result in greater productivity. If anything, long hours only hurt productivity.

But Successful People Work Their Guts Out

You've probably run across the argument that because extremely successful people, as a group, worked a punishing schedule every day before they "made it," it's clearly necessary to work long hours if you want to succeed.

If you dig deeper, you'll find that extremely successful people didn't become successful because they worked long hours. They became successful because they had an intense inner desire to accomplish something they had envisioned. They worked tenaciously toward their goals because of that inner drive, and it was their constant focus that made them successful. These successful people worked long hours because every fiber of their being drove them to work toward their goals; they didn't work all those hours because somebody else forced them to. There are countless examples of people who put enormous efforts into their businesses or other endeavors and who still did not succeed. Long hours is not the key ingredient. The key ingredients of success are a crystal-clear goal, a realistic attack plan to achieve that goal, and consistent, daily action to reach that goal.

WEEKEND WARRIORS

You can probably get those demanding managers to see that forcing the team to work long days won't increase productivity, that it's better to enable the development team to work more efficiently. But those upper-level managers may turn your argument against you: "You say your team can work efficiently without working long days. Fine. But I want them in here on the weekends. You can't tell me that having them work weekends won't increase productivity." In most cases, they would be right, at least for a while, particularly if the team already works efficient 40-hour weeks and has plenty of personal time in the evenings.

But those upper-level managers need to realize that if they demand that teams work weekends, they may create an adversarial relationship between the teams and management. The people on the development teams know that weekends properly belong to them, not to the company, and the more weekends they're forced to work, the more likely they're going to resent being taken advantage of. If programmers start leaving the team, or worse, the company, to work for less exploitative management, the company loses because those programmers will have to be replaced by new programmers who naturally will know less about the project and might be less experienced overall. The resulting loss of productivity might be great enough to cancel the gains made during all those weekends. And imagine the loss to a team—and this has been known to happen—when a fourth of its members leave the week after their product is released. Does that bother those short-sighted managers? No way: "Good. We've weeded out the wimps and the whiners."

One argument I've heard is that competition is so fierce in the software industry that if a company is to stay competitive, the development teams have to work long hours and weekends. *Have to* is another one of those expressions you should become sensitized to. Saying that developers *have to* work weekends to beat the competition is just another way of saying "We *can't* beat the competition unless programmers work weekends." Oh? The team isn't smart enough to find other ways to release a product earlier? I hope this book brings home the point that there are numerous ways to get the job done with much less effort than most teams are expending.

Weekends belong to the team members, not to the company. Teams don't need to work weekends in order to beat the competition.

The Initiation Process

Some people insist that teams must work long hours for an altogether different reason than getting more work done: the practice is vital to team-building, they say. They say that working long hours is an initiation, akin to boot camp, that wears programmers down and ultimately makes them feel that they've earned the right to be part of the team.

Let's assume that the point is true, that some sort of rigorous initiation is beneficial to team-building. Is working long hours really the best rigorous initiation?

In a field such as programming, where the ability to *think* is critical, why put a premium on working long hours? If there's to be an initiation, shouldn't it be one that forces programmers to exercise their brains, to *think* hard? When new programmers start out, they need to learn to think hard about their designs, to think hard about how to implement their designs cleanly, and to think hard about how to thoroughly and intelligently test their implementations. A new programmer needs to learn that when her code has a bug, she must never guess where it is and try to fix it with a lucky change—she must stop and *think* whether she has systematically tracked the bug all the way to its source. She must learn to think about the bugs she finds to determine whether there are related bugs that haven't shown up yet. She must learn to think about how a bug could have been more easily detected and how it could have been prevented in the first place. She needs to learn right at the outset that she is expected to *read* to keep abreast of the industry and to actively increase her skill levels.

These practices are tough to learn and follow through on. Really tough, because they can't be done mindlessly. Yet they must be mastered at some point. Make mastering these practices the initiation—not working long hours, which has nothing to do with programming well.

Stress the importance of thinking hard, not working hard.

I'll Lose My Bonus!

When I went down the halls kicking programmers out of their offices with "Go have a life," some programmers would protest: "But what about bonuses? If I don't work long hours, I won't get a big bonus at review time."

I would explain that I never base bonuses on how much overtime a programmer works, that in fact I view the need to work overtime as an indication of problems that need to be fixed, not as something to reward a programmer for.

"If you want large bonuses," I'd tell the programmer, "look for methods that will help bring our products to market more quickly and with higher quality. Point out areas in which we're duplicating effort, or where we could leverage code written by another team. If you've got an idea for a new type of testing tool that would automatically detect certain kinds of bugs that we have trouble spotting right now, bring it up. If you know of a commercial tool that will do the same thing, that's even better. If you think of a user interface feature that would be more intuitive to use, great—particularly if the idea would work across the product line."

"And if you want to get large *raises*," I'd continue, "increase your personal value to the company by actively learning new skills and developing good work habits—things that will make you work more effectively. If you want to really shine, develop the habit of *constantly* earning bonuses—look constantly for new ways to bring our products to market more quickly and with higher quality. That habit will earn you large bonuses and large raises."

I want programmers to work *better*, not longer.

Turning the Project Around

If your team is currently working long hours and you decide to put a halt to that backbreaking effort in order to focus on finding the causes of problems and fixing them, you'd better brace yourself. When you first start kicking people out, nobody will get any work done. That can be frightening, but it is an essential part of the turn-around process. Just as people don't naturally have study skills, they don't naturally have skills for working efficiently in a 40-hour week. Such skills must be developed, or relearned. Be prepared to do some immediate training.

When I find a programmer who is having trouble getting his work done in a 40-hour week—and I don't believe it's because the schedule is too ambitious—I ask him to make a list of how he spent his time that day, or the previous day, to get a snapshot of how he uses his time. The programmer would typically create a list similar to this one:

- Conducted an interview and wrote feedback for Human Resources
- Chatted with a programmer on the CodeView team for 30 minutes
- Read the daily drop of the *comp.lang.c* and *comp.lang.c++* news groups
- Read *PC Week*
- Took a two-hour lunch break to eat and run errands
- Reviewed a draft section of the user's manual
- Attended another team's status meeting to report on the progress of a feature they want
- Played air hockey in the game room for 30 minutes
- Read 27 e-mail messages and responded to 15 of them

That's how he would have spent his first seven or eight hours at the office, without having written any code. Am I joking? No. In my experience this is a typical list of activities for a programmer who is used to working 12-hour days.

Of course the programmer wasn't reading *PC Week* every day, but throughout the week he was reading *something* every day—the company

newsletter and his subscriptions to *InfoWorld, Microsoft Systems Journal, PC Magazine, Windows Sources,* and *Software Development.* E-mail would be a constant interruption. He would conduct one or two interviews a week, read those *comp.lang* news group drops daily, and regularly take two-hour lunches to run errands.

Flextime, or Do Time?

Microsoft, like many high-tech companies, has a "flextime" policy. You can work any hours you want as long as you get your job done. That's why I would find programmers who had no qualms about playing air hockey for 30 minutes or taking two-hour lunches. You can get fired at stricter companies for taking such liberties, but not at Microsoft—as long as you get your job done.

Flextime can be wonderful. If you have a dentist appointment, you just go. You don't need special clearance from your manager. If your daughter is in a school play, you go. If you happen to be a baseball fan, afternoon home games aren't a problem; you hop in your car and go. Flextime can dramatically improve the quality of life for employees because it allows them to design their work schedules around the needs of their personal lives.

But there is a dark side to flextime, one that the Human Resources folks don't tell you about as they itemize the reasons you should join the company. By definition, flextime means that there are no set working hours, so the primary way to gauge whether a programmer is working is to see whether he or she is knocking out features as scheduled. If you think this through a bit further, you can see that if a programmer starts slipping, the implication will be that he or she is not working enough. Nobody comes right out and says that, of course, but there's no question that you're expected to stay until you've finished. It doesn't matter that you've already put in a full day.

If you see that one of the programmers needs to work long days to do his or her job, that's an indication of a problem. Maybe the programmer chronically abuses flextime, using it to mask a pattern of procrastination throughout the day, or maybe the long hours indicate something more serious. Don't ignore the problem.

For a programmer working 12-hour days, such a schedule makes sense. When else is he going to run errands or read all those magazines? If not during "work hours," when? This is the point missed by those upper-level managers intent on having programmers work long hours. They badger the programmers into working long hours, and the programmers inevitably rearrange their lives to accommodate the longer work schedule.

Once I had the programmer's typical workday down in black and white, I would start asking questions.

"Now that you're leaving at a reasonable hour and not at 10 o'clock at night, do you still need to take two-hour lunches to run errands, or can you handle errands after work? Do you read e-mail in batches a few times a day, or do you let e-mail constantly interrupt you? If keeping regular hours meant you had to read your news groups and magazines at home, would you be willing to make that trade-off? Do these talks you're having with people on other teams concern project-related issues that I should be handling instead of you? . . ."

I'd work with the programmer to create a schedule that would allow him to get his work done during the day and leave at a reasonable time. It's not difficult to work with a programmer to create a win-able daily schedule. It just takes action on the lead's part.

Train the development team to work effectively during a normal workday. Don't allow them to work long hours, which serves only to mask time-wasting activity.

I Can't Work During the Day

Programmers themselves regularly complain that they can't get any work done during the day, and a look at that programmer's work list in the previous section supports that contention. Many of the tasks on that work list seem to be legitimate business items. Programmers have to conduct interviews, read and respond to e-mail, review draft sections of user manuals, and so on.

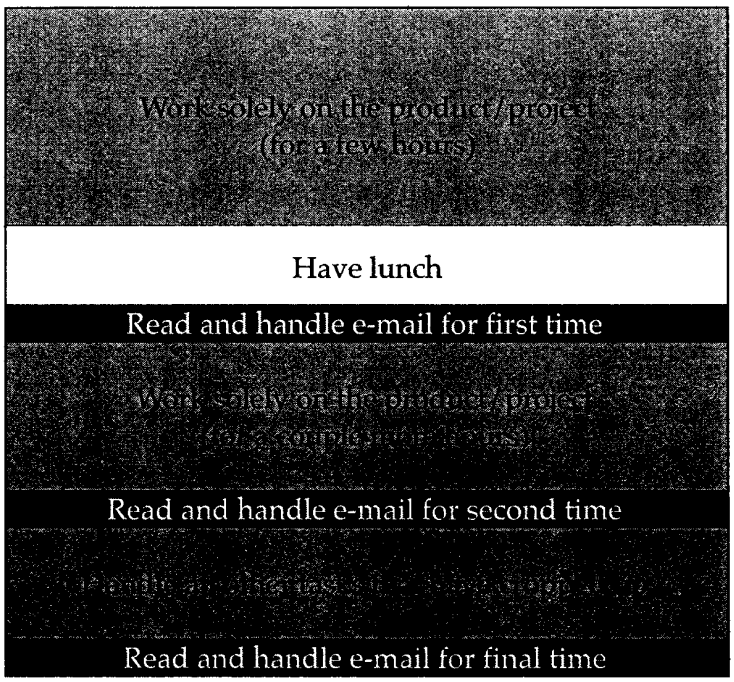
The problem with such necessary business tasks is that they constantly interrupt the primary job: improving the product. Just as reading each e-mail message the moment it arrives chops the workday into little, unproductive time chunks, so too does the regular stream of necessary business if team members don't have a plan for tackling such tasks efficiently. If they're handling each task the moment it lands on their desks, they'll have a difficult time getting work on the product done.

I've heard a lot of management advice recommending that you finish every task the moment it shows up. Either handle it immediately, or decide that you're *never* going to handle it and dismiss it forever. I agree with that advice because it prevents procrastination and helps people to stay on top of things, but I want to qualify the point. If programmers were to blindly follow that advice, interrupting their design and coding work to handle every distraction as it arrived, they wouldn't get much done on their product unless they worked late into the night, when there are usually far fewer interruptions.

The key idea in the advice is to "handle the task the moment it shows up." You might not think that programmers have any control over when tasks show up, but they do. Consider the e-mail example. If programmers respond to their e-mail at set times, only two or three times a day, they turn those random interruptions into predictable daily tasks. Then they can either respond to their messages (handling them immediately) or delete them (never to be considered again).

Programmers can apply the same principle to the other daily interruptions by turning them into predictable tasks that no longer disrupt their work. They just have to create a schedule describing how they'll work during the day—a plan that gives priority to improving the product, not handling interruptions. Take my daily schedule, for example, one which looks like the schedule shown on the next page.

I dedicate the time before lunch, when I'm freshest, to working solely on the product or the project, depending on whether I'm working primarily as a programmer or as a lead. I rarely answer my phone during those hours, and I certainly don't turn on my e-mail reader because reading and responding to e-mail is perhaps the most disruptive activity of the environments I work in. I try to get three or four solid hours of uninterrupted work completed before I do anything else. I don't read and respond to e-mail for the first time until I get back from lunch.



After I handle the post-lunch e-mail task, I have a second block of time devoted solely to working on the product or the project. If other tasks crop up during the day, I don't look at or think about them—they go right into my pile of tasks to tackle at the end of the day, where I have time scheduled to do them. When I finally get to those tasks, I handle them immediately or never. If for some reason I can't finish a task that day, I don't look at it again until the scheduled time the following day.

The point is that, with such a schedule, e-mail and other common interruptions don't distract me from my primary work. I take care of those tasks, but during the time I have *planned* for them, not when they happen to roll in. My schedule turns unpredictable interruptions into predictable tasks, and it puts those tasks lower in my list of priorities than working on the product—just where they should be.

Unfortunately, too many programmers unknowingly have their priorities reversed: they give e-mail and unforeseen tasks higher priority than improving the product, so at the end of the day, they haven't even begun to work on designs or write code. Instead, they have answered e-mail messages that didn't really need responses or tackled tasks that could have been spread over several days. What choice do they have, then, but to work long hours? If they didn't, they'd never get any product work done.

If you truly believe the project schedule is attainable and yet the programmers find they must work long hours to meet that schedule, you still have problems to find and solve. You should check these possible sources of the trouble:

- Programmers are allowing unpredictable interruptions to disrupt their work on the product instead of turning those unpredictable interruptions into predictable tasks.
- Programmers are giving interruptions higher priority than the primary task.

The schedule I've laid out works well for me, but I'm sure that for others it would be too restrictive or too *something* for their tastes. I'm sure that for some people the idea of not reading e-mail until after they get back from lunch seems impractical: "I can't do that." If reading and responding to e-mail is an integral part of their primary task, I'd agree with them. But if their primary task is working on the product, I'd urge them to try working for a few hours each day before first turning on their e-mail reader. At the very least, I'd urge them to consider reconfiguring their mailers to call their hosts less frequently and to turn off the notification beep that sounds when new mail arrives. In any case, the members of the development team should have daily schedules that help keep them focused on their primary work.

Work with programmers to create daily schedules that turn unpredictable interruptions into predictable tasks. The schedules should give their primary tasks priority over all other work.

"Working Solely on the Product" Defined

When I say "working solely on the product," I don't mean that programmers should lock themselves in their offices and barricade the doors, doing nothing but designing and writing code. Spontaneous discussions in the hall, brainstorming sessions, and code reviews are also part of working solely on the product.

CONSUMED BY EXCITEMENT

There are a few cases in which working long hours over the short term makes sense—working the weekend right before a drop to put all the finishing touches on the code, for example, or working hard the week before a COMDEX show to create a killer demonstration. But I stress *short term*. Long hours produce increased productivity for only the first week or two, when the sense of urgency is strongest. If you ask a team to work months of 80-hour weeks, they will work hard initially, but once the sense of urgency wears off, they'll fall into the pattern I described earlier—taking two-hour lunches to run errands, having long chats in the hall, and so on.

The exception to this tendency is when people are so excited about their project that you can't get them to leave. Such projects are truly wonderful because you eat, breathe, and sleep programming. I hope that everybody experiences such a project at least once, but I do have one reservation about such projects.

Early in my career, I spent nearly five years working on a handful of projects that were so exciting that I did little but write code, eat, and sleep. So did the other members of the development team. We didn't know what a social life was. We lived to code, often working until 2 or 3 o'clock in the morning, only to return six or seven hours later to start another day. And we loved it. We had that burning desire to see the product finished as we envisioned it.

After working on those projects, I worked on several more exhilarating projects, but I didn't program to the exclusion of all else. I worked a traditional 8-hour day, which gave me the opportunity to pursue an active social life after work—going to parties, taking 40-mile bike rides with friends, going to the theater, learning to ski, meeting new and interesting people. . .

What an eye-opener. If somebody had told me as I worked on those earlier projects to the exclusion of all else that I was missing out on an important part of life—a personal life—I would have laughed at them, just as people using 8-MHz IBM PC machines often laugh at people who suggest they should upgrade to the latest machines, which are 100 times faster. "I'm happy now. Why should I change?" But once the user's machine breaks and she buys a new one, her attitude undergoes a dramatic

transformation: "I can't believe I waited so long to upgrade. To think that I was actually *satisfied* with that old clunker!"

Like such computer users, I had no idea what I was missing out on, not having had an active social life for so long. Those projects were so exciting that I never felt the need for a social life; my life was complete as it was. But once I'd worked on exhilarating projects during which I also pursued an active social life, I learned how important it is to have a balanced life. And that has been the driving force behind my desire to do absolutely the best I can in a regular 8-hour day, so that I can balance that work with my personal life, getting the best of both worlds.

As exciting as it was when I was working on those all-consuming projects, I wish that somebody had pulled me aside back then to explain that there was more to life than work. I might not have listened, but I still wish that somebody had tried. So even though programmers on my teams are sometimes so thrilled with their work that they want to work long hours, I urge them, "Go home. Have a life."

HIGHLIGHTS

- The need to work long hours is a clear indication that something is wrong in the development process, whether it's because the team is doing nonstrategic work or because the team is being bullied by a misguided manager. No matter what the reason for the need to work long hours, leads must not ignore the problem and continue to let the team work late into the night over the long term. Leads must tackle that problem and make it possible for team members to work effectively in the scheduled 40-hour week.
- I often hear upper-level managers and project leads praise team members for working long hours. "Your commitment to the company is admirable. Excellent job!" That's exactly the wrong message that managers and leads should be sending. People should be praised for working well, not for the number of hours they're in the building. Managers and leads must never confuse "productivity" with "time at the office." One person might work far fewer hours and produce more than somebody who works twice as long.

- You can minimize meetings, reports, and other corporate processes, but unless you also focus on the wasted effort unique to each individual, you'll be missing a significant part of the problems you need to work on. Make it a priority to help each team member design large blocks of uninterrupted time into his or her daily work schedule.
- If you care about your team members, don't allow them to spend all their waking hours at work. Make sure they work a solid 8-hour day, and then kick them out. Taking that stand at your organization may seem sacrilegious, but if you believe, as I do, that people work better if they have an enjoyable personal life, take that stand.
- There's nothing sacred about the 40-hour work week. It's a U.S. tradition, so software projects tend to be scheduled on the assumption that each programmer will work a 40-hour week—five 8-hour workdays. If it takes a lot more than 40 hours per week per programmer to meet one of those schedules, something is wrong. The schedule might be unrealistic, or the programmers might need more training. Either way, there is a problem that needs to be fixed—not masked by having the programmers work long hours to compensate for the problem.