3

Of Strategic Importance

I like to think that my projects are always on course, but in fact they never are. Sometimes a project is ahead of schedule, sometimes behind, but always close. The project zigzags along an imaginary line that plots the ideal course.

Even the best-run projects are never on course. But if you let a project coast, not knowing how far off course it is, you're going to wake up one morning to find that you've zigged so much that you can't zag enough to correct. In that respect, a project is like a rocket aimed at the moon—a tiny fraction of a degree off, and the rocket will miss the moon by thousands of miles. If your project is off track, even slightly, it will steadily get further off track unless you make regular, tiny adjustments to its course.

Effective leaders understand this principle. They take consistent, daily steps to nudge their projects back onto those imaginary trajectories. In this chapter, we'll look at simple, effective strategies you can use to keep your projects on track.

Freeway Overpasses

I'm convinced that the main reason projects go astray is that the people running the projects don't spend enough time thinking about how to keep them running smoothly. They don't anticipate problems and instead wait until problems show up. By then, it's too late. What could have taken 30 seconds of extra thought to prevent a month ago is now going to take hours or days to correct. This is known as "working in reaction," and many leads seem to do it.

The alternative to simply reacting is to actively look for potential problems and take little steps to avoid them. Suppose one of those house movers I talked about in Chapter 1 had hopped into his flatbed truck and started slowly on his way along the route to the house's new location, only to turn a corner and be blocked by a low freeway overpass. Oops—gotta retrace at least part of the route and have the same overhead power and phone lines taken down again. Or what if the planned route looked flat on the map but had hills too steep to pull a house up? Or what if the route were usually passable but road crews were out that week resurfacing a stretch of the road? Each of these obstacles could have been foreseen by the "house lead" if only he had taken the time to drive the route the day before and then again an hour before starting the house rolling. Can you conceive of a house lead's not taking that step? Why then do so many software leads fail to drive ahead and look for obstacles that could easily be avoided, allowing their projects to be stalled by those obstacles?

Leads don't always look ahead because that's harder to do than not looking ahead. How many times have you heard a lead faced with an unexpected obstacle say, "I could have prevented this if I'd spent time thinking about it earlier"? In my experience, few leads make such an admission. Rather, leads tend to be not at all surprised that something unexpected has come up. After all, they think, it happens to everybody, all the time. It's normal.

To get out of this mind set, you need to work proactively instead of reactively. There are many techniques you can use to train yourself to work proactively, but they can all be boiled down to a fairly simple practice:

> *Regularly stop what you're doing and look ahead, making little adjustments* now *to avoid big obstacles later on.*

Leads don't have trouble spotting the already big obstacles coming up—say, having to support Windows NT once the regular Windows version is done, or having to find time to create a demonstration-quality product in time for COMDEX. It's the little obstacles that blindside people, the ones that blossom into huge obstacles if they aren't foreseen and handled early, while they are still manageable. That kind of foresight is like stopping for gas before heading to the ski slopes—taking that simple step could prevent you from having to make a long, snowy trek because you ran out of gas halfway up the mountain.

The habit I've developed and used successfully for more than a decade is to spend the first 10 or 15 minutes of each day making a list of answers to this question:

> *What can I do* today *that would help keep the project on track for the next few months?*

It's a simple question, but if you ask it regularly, it will give you all the information you'll need to protect your projects from being clobbered by problems you could have foreseen. Note that the tasks you'll list probably won't be complex. In fact, most such tasks are simple and can be completed in a few minutes. My own list of tasks is usually like this one:

- Order the MIPS and Alpha processor manuals so that they'll be here well before Hubie needs them.
- Send e-mail to the Word team reminding them that they must make additional feature requests by Monday if they'll need those features in our next library release.
- Send e-mail to the Graphics lead to verify that the Graphics library we're depending on is still on track for delivery three weeks from now.

None of these tasks would take me much time to do, but they could save me an enormous amount of time later on. Ordering that MIPS processor manual may not seem like a big deal, but if the manual takes three weeks to arrive, that could cause a three-week slip in the MIPS work. How long does it take to order a manual? About 10 minutes? You could spend 10 minutes *now* and have the manual in time, or spend 10 minutes *and* three weeks later on. . .

Often, by means of such little tasks, you can discover that the Graphics lead thinks he might slip two weeks, or that the Word group does have another request but didn't think there was any need to hurry up to tell you. Without checking (looking ahead), you could get caught off guard by a slip of the Graphics library schedule, or you could have a last minute fire when the Word team realizes that the feature they need hasn't made it into the next release of the library.

In an ideal world the Graphics lead would tell you well in advance that his project was going to slip, but how many times does that really happen? In my experience, almost never, because leads don't want to alarm anybody until it's clear that they are definitely going to slip—three days before the scheduled drop.

Each day, ask, "What can I do today to help keep the project on track for the next few months?"

BAD INTELLIGENCE

During the development of Word for Windows, I was asked to take a look at an internal code library written by non-Word programmers. The library was a dialog manager whose purpose was to isolate the operating system from Microsoft's applications, allowing programmers to create dialogs without worrying about whether the applications were going to run on Windows, the Macintosh, or some other system.

I was called in to find out why the library was so slow—the Word for Windows project lead and program manager were irritated by the delay between the time a dialog was invoked and the time it was fully

displayed and ready for user interaction. The programmers working on the dialog manager had profiled their code and had made numerous optimizations, but the Word group was still dissatisfied and was making a ruckus, slowly ruining the library's reputation within the company. Other teams who were planning to use the library had begun to back off.

When I talked with Word's program manager to better understand the speed problem and find out what performance would be acceptable to the Word group, he handed me a list of acceptable display times—their quality bar. Each dialog had to be fully displayed in the time indicated next to its name. The program manager then demonstrated by bringing Word up and invoking a dialog with one hand while starting a stopwatch with the other. "See," he said, showing me the stopwatch. "This dialog takes too much time." Visually the dialog itself didn't seem to be that slow to me, so I reached over and invoked the dialog a second time to get another look. The dialog appeared almost instantly. I pointed out that the second invocation was well under the acceptable time limit.

"It's always fine the second time," he said. "We're only concerned about the first time, when the dialog is invoked after a period of inactivity. That's when the dialogs are too slow."

I understood the problem and went back to my office to look at the code. What I found was startling. It turned out that, at the time, Word itself contained an optimization that overrode the normal Windows code-swapping algorithm. The optimization was kicking out all "unnecessary" code segments after a certain period of inactivity, and that optimization was kicking out every byte of code related to the dialog manager. A little measuring showed that even if the dialog code were executed instantaneously, no dialog would pass its speed test; Word simply took too long to reload those "unnecessary" segments.

So the speed problem that the Word people had been complaining about wasn't a speed problem at all, but was instead a code-swapping problem. The Word team thought the dialog manager was much too slow, yet the dialog team couldn't see where the slowdown was—the code seemed fast enough in the library's test application, and there was no reason the dialog manager should have behaved differently when linked into Word. Of course, the test application didn't override the Windows swapping algorithm.

The Word team had been complaining about the speed problem for months, and the dialog team had been working long, hard hours to optimize code and algorithms in the library, hoping that each latest round of improvements would finally be enough to satisfy Word's requirements. Had anybody stopped to profile Word's handling of the dialog manager

The Debugging Game

Many programmers don't do research during debugging sessions. Some programmers try to fix a bug by jumping into the code, making a change, and then rerunning the program to see if the bug went away. When they see that the bug still exists, they make another change and do another run. Nope, that didn't work, better try something else. . .

I know that some programmers play the "maybe this is the problem" game because whenever they get a difficult bug for which none of their guesses seems to work, they ask me, their lead, what they should try next. The "next?" question is a dead-giveaway that they're playing the guessing game instead of actually looking for the cause of the bug.

In my experience, the most efficient way to track down a bug is to set a breakpoint in the debugger, determine which piece of data is bad, and then backtrack to the origin of that bad data, even if it means mucking around in data structures, following pointers, and other such tedious stuff. There's no question that it's sometimes easier to guess where a bug is and then fix it with a lucky hit, but it's consistently more efficient to look at the actual data and backtrack to its corruption.

I'm also skeptical of programmers who find bugs by "looking at the code." Andrew Koenig's *C Traps and Pitfalls* is an entire book of C examples that look perfectly correct but in fact contain subtle bugs. And Gimpel Software's marketing campaign for their PC-Lint product features magazine ads each month that point out obviously correct, yet buggy, code.

Looking for bugs by looking at the source code is lazy and inefficient; it shouldn't take a programmer any more time to view the code in a debugger, watching the *data* as he or she progressively steps through each line of source.

code, he or she would have seen that no amount of code optimization could have solved the problem the Word team was complaining about.

Granted, it's not always reasonable for a library team to regularly build and test all of the dozens (or sometimes thousands) of applications they support. It does make sense for a library team to use an aggressive—and I stress *aggressive* here—test application specifically designed to exercise every aspect of the library. But in this case, the Word team had been complaining loudly for quite some time, and the library team had found no obvious problems in their test application's use of the library. Somebody well before me should have built and profiled Word to see why it was behaving so differently from the test application. A little bit of research early on could have saved months of misguided optimization work, and the library probably wouldn't have developed an undeserved reputation.

As a lead, you should keep a wary eye open for any problem that persists and make sure that you, or someone, stops to do some focused research to figure out what's going wrong. The research may be tedious and time-consuming, but that's better than spending weeks or months trying to fix the wrong problem.

Don't waste time working on the wrong problem. Always determine what the real problem is before you try to make a fix.

OUTRAGEOUS MENUS

One time, the technical lead for the Windows-like user interface library I talked about in Chapter 1 came to me in a panic. He had just received a request from an application group for a feature that would take weeks to implement, yet our delivery schedules were pretty much carved in stone—we were not in a position to slip, at least not without severe repercussions. I asked him what the application group's request was.

"They want a modified form of our drop-down list boxes. They want to be able to use the list boxes outside dialogs; they want to be able to display the list boxes without their scroll bars and to be able to dim

some of the list box items. They also want to be able to click on a list box item and have it automatically pop up another list box, but if you move the mouse back into the original list box, the new list box automatically disappears."

Whew!

I had to agree: implementing those requests would kill our delivery schedules. After hearing a full description of the request, though, I wasn't concerned—anything that unusual didn't belong in a shared library. My initial thought was to give the application group the code for the standard list boxes so that they could implement those quirky list boxes themselves. Still, I was puzzled by their request. What were they going to use those list boxes for? I assumed it must be for some newfangled user interface I'd yet to see. So before saying we wouldn't do it, I asked the technical lead to find out what the application group was going to do with those bizarre list boxes. He returned a while later, a wide grin on his face.

"They want to use the list boxes to simulate hierarchical menus, like the menus in Windows and on the Macintosh."

Now I knew why the technical lead was grinning: we already had an add-on library that fully supported hierarchical menus; the other group was simply unaware of the fact.

I bring this story up because it's common for groups to ask for something without explaining the reason behind their request. I see this all the time, even outside work. At a diner I sometimes go to for an early lunch, people occasionally come in and, seeing that everybody is still eating breakfast, ask the waitress, "How late are you going to be serving breakfast?" I've seen dozens of hungry people turn on their heels, mumbling, "I really want lunch," and walk out the door before the waitress can tell them they can order lunch. The lunch menu is available around the clock.

Why did those people ask about breakfast when what they really wanted to know was "Can I get lunch?" Their thinking went off on a tangent that seemed to be related to what they wanted, and they asked the wrong question. It happens all the time. I'm sensitive to the problem of asking the wrong question, but I still find myself asking my wife, Beth, when she'll be home from her evening soccer game—when what I really want to know is what time she'd like to have dinner.

Asking the wrong question or raising the wrong issue seems to be a common problem, and if you're aware of this tendency in people, you can save everybody time and effort by making it a habit to determine what the other people are actually trying to accomplish. If what they're trying to achieve isn't clear from their request, be sure to ask them what they're trying to do before you spend much time working on the request—or you refuse it.

People often ask for something other than what they really need. Always determine what they are trying to accomplish before dealing with any request.

First Define the Context

A good way to avoid miscommunication in your own requests is to first define the context of what you're trying to accomplish and then make your specific request. Suppose the programmer from the application group who made the list box request had started his e-mail this way:

```
We need hierarchical menus for the next release of our
product. Since drop-down list boxes are similar to menus,
we think we can simulate hierarchical menus if you can
provide us with a modified form of drop-down list boxes
that allows us to. . .
```

If we had received that e-mail message, our technical lead wouldn't have panicked, he wouldn't have had to meet with me to figure out how to handle the request, and he could have immediately told the other group about the add-on library's support for hierarchical menus. Even more important, I wouldn't have almost rejected their request—in which case they could have spent weeks reimplementing a library we already had.

By first telling people what you're trying to accomplish, you get them focused on helping with your ultimate need, not on one possible solution to that need.

Just Say No

Suppose we hadn't bothered to find out why that group needed those weird list boxes and had simply turned down their request. Do you think they would have said, "OK, we understand. Thanks anyway"? Maybe. But plenty of groups would have argued that as custodians of the user interface library we had a responsibility to maintain the code and provide new features when they were asked for—that giving them some source code to adapt just wouldn't do.

Of course, the easiest way to resolve such disagreements is to knuckle under and agree to do the work, and that's exactly what I've seen many leads in troubled groups do. These leads would rather defuse a tense situation than fight for what's best for the product or their team.

Sometimes a group will make a perfectly reasonable request that, because your schedule is full, you can't meet, and you're put in the position of saying No to that group. I know from experience that there are plenty of leads who, to avoid the confrontation, will agree to fulfill the request anyway, without having any idea how they'll get the work done on time. Somehow, they think, they'll pull it off. And, of course, they rarely do.

What these leads don't realize is that by agreeing to work they shouldn't do or can't do they are dodging a bit of short-term pain in exchange for a lot of long-term pain—and not just for themselves, but for every single member of their teams. Which do you suppose is more painful all the way around: showing the lead of a dependent group why you can't possibly fulfill a request given your current schedule, or promising to finish the work on a specific date and then missing that date by six weeks?

Consider the difference. When the lead of the dependent group makes a request, the date on which that request needs to be fulfilled is often in the distant future; if you can't fulfill the request, there's plenty of time for you and the lead of the dependent group to consider alternatives. The only way you can be considered the villain is to reject the request without even trying to help the other lead work something out. Compare that approach to caving in and agreeing to deliver some new functionality, thinking you will somehow get the work done—and

missing the deadline you agreed to. Not only did your group miss its deadline, but you've possibly caused all the groups depending on you to miss their deadlines as well.

Think of it this way: if you were buying a house and needed a loan, which bank would upset you more, the one whose loan officer turned you down immediately, or the one whose loan officer agreed to give you the loan but changed his mind two months later as you were signing the closing papers?

I'm not saying that you should turn down requests just so that you'll have a cushy schedule. I'm saying that you should never commit to a date you know you can't meet. It might be tempting to think that you'll somehow make the date, but that's usually just wishful thinking. There are enough slips in dates leads "know" they can make, let alone in the dates they're unsure of.

It's not easy to fight these little battles up front, but it beats having the company CEO sitting on your desk several weeks or months later demanding to know why you waited until Marketing's ads had hit the magazine stands before you confessed that you couldn't possibly make the dates you promised.

Don't Halt the Machinery

Fighting your battles up front puts a critical process in motion—the search for a true solution. If you were truthful and realistic about what your team could actually accomplish and said No when you knew you couldn't meet a date, *the search for a workable solution would continue.* Maybe the other group would do the work themselves, or maybe they'd split the work with you, or maybe they'd ask other groups in your organization if they had a similar piece of code already written, perhaps buried in the guts of some application. Who knows?

Saying No may be unpleasant, but it keeps the problem-solving machinery chugging away until somebody, somehow, can say Yes and believe in what he or she is saying.

Never commit to dates you know you can't meet. You'll hurt everybody involved.

I Failed to Say No

One time, the Word for MS-DOS team asked our user interface library team to implement a costly add-on feature in time for Word's upcoming beta release. We were booked solid with work, and I couldn't see any way to meet their date without slipping our own date and affecting the more than 20 other groups using the library. I explained to the Word group that we could—and would—do the work, just not in time for their deadline. I proposed that, if they definitely needed the feature that quickly, they implement the add-on themselves, turning it over to us when it was completed. We would document the feature for other teams, enhance our test application to cover the feature, and support and continue to enhance the feature in the future. The Word team was upset. They felt we should do the work since it was a feature that every other group would eventually want to use. They were right on that point, but that didn't change the fact that we couldn't implement the code in time for their release. We battled over this feature for nearly two months. I finally got so frustrated with the arguing that I broke down and agreed to do the feature, figuring that I'd temporarily pull a programmer off one of the other projects I was leading.

Well, I couldn't find that spare programmer, and the result was disastrous. We missed Word's deadline by weeks, and they screamed bloody murder. We missed all of our other commitments too—which we had been on track for—affecting those 20-odd other teams. More screaming. What a mess. If I had stuck to my guns and said No as I knew I should have, everybody would have been a whole lot better off, including the Word group.

THE NEED TO PLEASE

As a lead, you're going to be faced with all sorts of demands. To be effective, you must learn to say No when it's appropriate. Others may not like it, and they may think you're wrong, but you have to realize that you can't always please everybody—there are often just too many conflicting requests.

If you're in charge of a shared library, one team may ask you to add a feature that benefits only their project. If you say No, they'll probably get upset. If you say Yes to their unique request, another team may complain about the increase in the size of the library. These no-win situations come up all the time, particularly when you're responsible for code shared by multiple projects.

Which course of action should you take when you're faced with conflicting demands? That's where your detailed project goals come in handy. If one of your goals is to provide functionality that will be useful to all of the groups using your library, you know to reject a request that doesn't match that criterion. Sure, you'll get complaints, but it doesn't take much time to explain your reasons and to point out that if you implement one unusual request you'll have to implement the special requests made by every other project you're supporting, which will pull you off features *all* groups want and bloat the library with features that most groups don't need.

There seems to be a human need to please everybody, and that need can get leads into trouble because, in their desire to please everybody, they can do things that don't make sense for the project.

In my experience, people don't like having their requests rejected, but if you have solid reasons, they do understand and often appreciate your not giving them false promises.

Don't let wanting to please everybody jeopardize your project. Use your goals to guide your decisions.

Not a Librarian?

I've been assuming for the sake of argument that you're leading a library project, and I know that that's probably not the case. The points I'm making apply to most projects, though. Instead of having other leads making demands on your group, you might have a marketing team making the demands, or the folks who'll use the finished product. Every project will have some outside demands made upon it—even top secret projects always seem to have people outside the development team poking their noses in and making suggestions.

SUPERIOR SUGGESTIONS

You should be especially conscious of not trying to please everybody when it's your boss who makes suggestions. I'm not talking about resisting authority. I just want to point out that superiors can make bad suggestions just as everybody else can, particularly if they aren't aware of your goals, your priorities, and the technical challenges you face. If you want to be an effective lead, you must weigh all suggestions (or demands), no matter where they originate, against the needs of your project.

If your boss asks you to do something you think is a bad idea, explain your concerns before you undertake the work. Sometimes your boss will agree with your concerns and drop the suggestion; other times, your boss will acknowledge your concerns and go on to ask you to honor his or her suggestion anyway—in the best case scenario, providing solid justification. Regardless of the outcome, one or both of you will probably learn something.

I once reviewed a large piece of code written by an experienced programmer. I was surprised to find several critical design flaws in the code, flaws I wouldn't have expected to appear in code from this particular programmer. I asked the programmer why he had chosen such a design.

"I just did the implementation. Kirby did the design." Kirby was his lead at the time.

"How do you feel about this design?" I was curious.

"It's not the way I would have done it."

"Did you feel that way at the time you did the implementation?"

"Yeah," he shrugged. "But I had just started at Microsoft, Kirby was the lead, and I figured he was more experienced than I was. I thought he saw something in the design that I didn't. I didn't want to rock the boat."

In fact, Kirby was less experienced than the programmer who did the implementation. Kirby had simply been fortunate in getting a more experienced programmer on his team.

In another case in point, I was leading the teams responsible for Microsoft's 680x0 cross development system. Periodically Mort, a manager who had the power to change my development plans, would drop by my office to chat about the progress of the 680x0 C/C++ compiler. During every visit, Mort would get around to asking what grew to be the inevitable question, "How's the FORTRAN work going?"

Now, Mort knew darn well we weren't trying to produce a FORTRAN compiler, but he had a fondness for FORTRAN and felt there was a market for a good Macintosh FORTRAN compiler. Besides, creating a FORTRAN compiler out of the C/C++ work we were doing wasn't a bad idea—especially if you knew, as Mort did, that Microsoft's compilers use the common three-stage process described in most compiler texts:

Front end:	Parse the specific language (C/C++, FORTRAN, Pascal, and so on) into a common intermediate language.
Optimizer:	Perform all compiler optimizations (code motion, common subexpression elimination, strength reduction, and so on) on the intermediate language.
Back end:	Generate optimized object code from the now-optimized intermediate language.

It's a bit more complicated than that, but you can see from this staging that to get a Macintosh compiler we needed only to write a new back end, one that generated Motorola 680x0 code instead of Intel 80x86 code.

In theory, then, once we had finished the 680x0 back end, we should have had our C/C++ compiler, plus FORTRAN and Pascal compilers—we just needed to link in the proper front ends. That's in theory. And that's why Mort was so interested in the possibility of a FORTRAN product. In reality, though, to build the FORTRAN compiler, we would have needed to fully implement the back end, and we were implementing only the 95 percent or so required by the C/C++ compiler.

Whenever Mort asked about the FORTRAN compiler, my answer was always the same: "We haven't done anything with that compiler." I would always follow with "But we're not doing anything in the back end that would prevent us from doing the FORTRAN work *at a later date.*"

Mort may have been right that there was a market for a good FORTRAN compiler on the Macintosh, but he was ignoring my team's project priorities. Just because there was a market and it was possible to create the product was no reason to temporarily halt work on the C/C++ compiler, which even he agreed had a significantly larger market potential. We wouldn't have had this discussion more than once if Mort hadn't been personally interested in the FORTRAN compiler. His personal interest was getting in the way of his business sense.

You must protect your project from outside manipulation, especially if the request comes from somebody who has clout. Somebody like Mort might not be right, but you might feel obliged to comply. In my early years as a lead, I probably would have bowed to Mort's pressure—I certainly caved in on similar requests. I eventually learned, though, that no matter where a request originates, you must question it. Does it improve the product? Is it strategically necessary according to your goals? Does it draw focus away from more important work? Will it be unnecessarily expensive or risky to implement? You must feel good about the answers to these questions, or you shouldn't do the work.

You are responsible for your project.
Don't let ill-considered suggestions from
superiors disrupt your progress.

THE TRUE COST

Why did Mort think that a Macintosh FORTRAN compiler was worth considering as a goal for the cross compiler project? Was it because people wouldn't stop calling Microsoft to ask why we didn't have such a compiler in our product line? Was it because coding in FORTRAN just made sense for the Macintosh environment? Of course not. The only reason the FORTRAN compiler was ever an issue was because one person who was fond of FORTRAN saw the possibility of getting a free FORTRAN compiler out of the C/C++ compiler work we were already doing.

I get excited about free products and features as much as the next person. There's that warm feeling you get when you realize that because you were such a brilliant designer, some unexpected functionality pops out. But free products are almost never strategic for your company, and free features are almost never strategic for your product. After all, if they were strategic, they would have been planned for, not serendipitously discovered.

It's interesting to note that we could also have gotten a Pascal compiler out of the C/C++ work by updating the older Pascal front end, but that idea never came up, even though the Macintosh was for many years a Pascal-only system—all the manuals and code examples from Apple Computer were in Pascal, and there were no serious development systems to compete with Apple's Pascal system. That's all changed now, of course; C/C++ has become the language of choice for the Macintosh. But if Microsoft were to ship a Macintosh compiler other than the C/C++ compiler, it would make far more sense, I think, to ship a Pascal, not a FORTRAN, compiler.

Mort was excited about the FORTRAN compiler because it was free, not because it was strategic. But how free would that FORTRAN compiler actually have been? To bring that free compiler to market, we would have had to

- Finish the remaining 5 percent or so of the back end to the compiler—a few programmer-months' worth of work.
- Find some way to enable FORTRAN programmers to interact in the Pascal-defined Macintosh operating system, which

makes heavy use of Pascal records—something FORTRAN doesn't directly support. We would also have had to find some way to allow everything from Macintosh "traps" to Pascal-style strings in FORTRAN.

- Write manuals and help files to accompany the product.
- Fully test the compiler, linker, debugger, and other tools that would go in the box.

I'm sure I could think of additional tasks that would be necessary (say, training a product support team), but these are the obvious chores that come to mind. How free does that compiler sound now? Granted, the technical writers could probably pull the manuals and help files together fairly quickly if they used the existing 80x86 FORTRAN documents as a starting point. But there's no shortcut to testing a compiler. The Macintosh FORTRAN compiler would have required the same full-blown testing effort that any release of the 80x86 compiler would undergo.

That FORTRAN compiler was anything but free. Yes, the compiler was cheap compared to what it would cost starting from scratch. But "cheap" can still be expensive—just ask anybody who's bought a used Boeing 747 lately.

Free products and features—like free puppies—simply do not exist. Anytime you hear, or even think, the word "free," your immediate reaction should be resistance, not acceptance. Think of free products and features as you would those cold-call offers in which you're told that you'll get a free dream vacation in Bermuda for simply dropping by a showroom to hear about some new downtown luxury condominiums. In rare instances, such opportunities may be gold bars to be picked up, but in most cases, they're merely lead weights. If you want to keep your projects focused and under control, stick to the strategic work and leave those lead weights alone.

There is no such thing as a free product or feature.

THE *LAYOFF* MACRO

Sometimes it's not a superior who makes questionable requests, but the marketing team. The scent of a big sale can cause the marketing team to consider features they'd never ask for in less heady situations. You need to protect your product from such requests.

When I was working on Microsoft Excel, the marketing team asked the development team to extend the product's macro language to include a new *LAYOFF* macro, which, as you can probably guess, was supposed to take a list of names and randomly pick people to lay off. A large corporate client had requested this *LAYOFF* macro so that they could lay people off without anybody being able to claim that the selections were biased. The company would be in a position to simply point to Excel to prove their innocence.

If you know Excel, you know that it doesn't contain such a *LAYOFF* macro. The task fell to me, and I refused to implement the request: I felt the macro would harm the product. My lead agreed, and for months we beat off the marketing team's persistent requests for the feature. Marketing felt they needed the macro to close the sale.

The feature became a big joke in the development group. "Let's do it, and we'll hardwire our names into the code so that we'll never be laid off! No, better than that, let's hardwire the marketers' names into the code so that they'll always be laid off!" Of course, none of that ever happened. In the end, Marketing wrote a simple user-defined macro to accomplish the same purpose. With that macro, the corporate client's request was met without Microsoft's having to build such an odious feature into the product.

In my experience, such ridiculous requests are rare. The marketing folks don't want to hurt the product. Just the opposite—they want the best product possible. But sometimes they're not too clear about what "best" means and ask for features you probably shouldn't implement. There are at least two types of such features: those that fill out feature sets and those that satisfy one of those product checklists you find in magazine reviews. Sometimes filling out feature sets or satisfying product checklists does improve the product, but just as often adding such features merely causes bloat and wastes development time.

The reason I say that—besides years of observation—is the motivation behind the requests. Think about it. Suppose the marketing team comes to you and says, "The Hewlett-Packard HP12c business calculator has these five functions that we don't yet support in our spreadsheet. We'd like you to add them to the standard set of functions." Would fulfilling such a request make for a strategic improvement to the product, or is it more likely that the request came about because a marketer realized, "Hey, we don't support the full set of HP12c features; we'd better add what we don't have"? Those additional features may actually be important, but if they are, why weren't they included in an earlier release? It's possible that those features simply weren't worth the time and effort. They still may not be.

Strategic Marketing

I don't want to leave you with the impression that you should adopt a cavalier attitude toward requests made by the marketing team. Every once in a while, they'll ask for something inappropriate, but usually they have sound reasons for their requests. At least that's been my experience.

Sometimes the marketing team will ask for features that aren't strategic for the product from a functional point of view but that are quite strategic for sales reasons. Does any application really need to read and write 23 different file formats, for instance? Of course not; users need only one file format to store their work in. Support for the other 22 formats is driven primarily by marketing needs. If your application isn't "file friendly," that can kill sales, if for no other reason than it discourages users from dumping competing products in favor of yours—they'd lose their preexisting work.

If you're faced with a feature you feel doesn't improve the product, consider whether the feature would measurably increase sales. That *LAYOFF* macro was inappropriate because it would have harmed the product, not because its only reason for being was to land that large corporate account.

If marketers are looking at magazine product-feature checklists, you'll run into the same problem—the requests will be for features that fill out the chart, not for features that are strategically necessary for the product. Sometimes the marketing team will see a questionably useful feature in a competing product and, in a knee-jerk conviction that your product has to do everything that the competitors' products do, ask for the feature. Watch out.

Implement features only if they are strategic to the product. Don't implement features merely to fill out feature sets or review checklists.

TOTALLY COOL, TECHNICALLY AWESOME

In Chapter 1, I mentioned that the user interface library lead and I reviewed the task list for the library. One of the items on that list was a six-week task to implement a feature that would allow third party vendors to hook little standalone applications into Microsoft's character-based applications. The idea was to make it possible to implement calculators, notepads, clock displays, and other types of desk accessories that Windows and Macintosh users take for granted. I thought the feature was interesting, but it didn't seem to me to be strategic for any of the 20 or so internal groups using the library.

When I asked the lead which group had asked for the feature, he told me that nobody had; it was on the task list because the *previous* lead had felt that it was important. I then asked if any of the groups had expressed interest in the feature when they had learned of it. Again, the lead said he didn't know, and he added that if I was considering cutting the feature, the previous lead would fight it if he found out.

I figured that if the previous lead felt that strongly about supporting desk accessories, there must be groups who really wanted the functionality and that the current lead must simply be unaware of them. So before cutting the feature, we asked the groups if they'd heard of the feature and whether they were interested in such support. The responses we got were all pretty much the same: "Yeah, we heard about that. So and So tried to convince us that it was important."

Most groups didn't want the functionality at all. A few were more interested than others, but only if we beefed up the feature so that there was strong communication between the accessory and the application. They didn't want calculators and clock displays; they wanted the ability to truly extend the application—for grammar checkers and other tools that could provide important functionality. Of course, providing a general purpose interface to allow such functionality was much more complicated than the original idea. We didn't have the time to implement the six-week feature, much less something more complex.

Our findings pretty much killed the feature, but before scratching it off the list, I talked with the previous lead to get his thoughts on the issue. He was disappointed by my decision to cut the feature, but nothing more. He couldn't provide any compelling reasons to implement the code except that it would be an interesting programming challenge and

What About Third Party Vendors?

It's possible that third party vendors would have loved to have seen support for those little pop-up applications. It's likely that some small company or enterprising individual would have seized upon that niche market and created numerous little add-ons for Microsoft's character-based applications. Nobody got the chance because I cut the feature. But I didn't cut the feature without first considering how beneficial such third party support might have been.

Had the add-on capability been much more powerful, as the applications groups wanted it to be, third party developers could have created some truly useful add-ons for other users, which in turn could have increased demand for the character-based products. But calculators? Notepads? Clocks? Nobody chooses a word processor, a debugger, or any other application simply because a third party vendor sells a nifty add-on scientific calculator.

Simply put, the users didn't need the functionality, which meant that the applications didn't need it. It would have been wasteful for us to spend six weeks working on pop-up code when we could work on code that users really did care about.

that it would have been cool if people could have used the little pop-up applications instead of TSRs, MS-DOS's problematic approach to achieving the same ends.

In effect, what we had was a six-week feature that was not at all strategic to the success of the user interface library, nor to the successes of the applications using the library. The task was on the schedule for only two reasons: it would have been fun to work on, and it would have been cool for the character-based applications to have desk accessories just as their Windows and Macintosh counterparts did.

Don't implement features simply because they are technically challenging or "cool" or fun or. . .

IS IT BETTER?

Sometimes tasks sneak onto the schedule because they seem truly important, but in fact they may not be if you consider whether they are strategic. For example, it has always irritated me that Excel uses a non-standard clipboard paradigm—the clipboard is not persistent. It's not that Excel's model is awkward or less useful; it just bugs me that Excel's clipboard doesn't behave the way clipboards found in every other Macintosh and Windows application behave. The saving grace is that Excel's clipboard implementation is close enough to the standard model that few people ever notice that it's different.

Now, I believe in following standards, particularly those that concern user interfaces. So you can imagine that if I were the Excel lead, I might think it important to bring Excel into line and would therefore put a "standardize the clipboard" task on the schedule. And, in fact, I do think that's important. However, I do not think that standardizing the clipboard is strategic in any way. Changing the clipboard's behavior could also break existing user-defined macros that rely on the current clipboard behavior.

If I were the Excel lead, I would *want* to standardize the clipboard, but I would strike that task from the schedule in an instant. I would feel

differently if users were confused or irritated by Excel's clipboard, but as I said, most people never notice that it's different.

Another type of important work that is rarely strategic is reformatting source files to adopt new coding styles or naming conventions. Suppose a project lead decides that all functions must have function headers that describe what the functions do and what the parameters mean. That seems perfectly reasonable. What I question is a lead's taking the next step—bringing development to a halt so that the entire team can spend days or weeks retroactively adding header comments to all the headerless functions written over the years. It's even worse when a lead halts development to institute a new naming convention. That can be incredibly costly if the team stops to rename every existing variable and function name. Such work may be important for maintainability, but it is rarely strategic. You can tell that the work is nonstrategic because it doesn't improve the product in any way.

True, you can view such file reformatting as an investment in maintainability that will ultimately improve the product, but stopping all development is a stiff price to pay. If you ask how you can get the benefits and eliminate the drawbacks, you can derive alternative approaches to adding those header comments all at the same time. An approach could be as simple as asking all programmers to spend half an hour a week writing headers and to add headers to any functions they touch during the day as they work on strategic tasks. Sure, it'll take longer before all the functions have header comments, but such an approach puts the initial investment more in line with the expected return.

Of course, if you're talking about stopping development to add debug code to the product, that might be another matter; adding debug code could definitely improve the quality of the product—and rapidly. The return on investment could be substantial, even in the short term.

Occasionally, I'll run across a Usenet news article in which a programmer says something like "We're in the process of rewriting all of our C code using objects in C++, and I can't figure out how C++ does. . ." When I read such notes, I shudder and hope that those programmers—actually their leads—aren't killing their products by taking the huge time hit that such a rewrite must entail.

You could argue that it would be beneficial to rewrite an assembly language program in a high-level language such as C—the resulting

productivity gains could outweigh the costs of doing the rewrite, and the resulting code might be more portable. But I've got to question rewriting a Pascal program in C, or rewriting a C program using object-oriented designs in C++. I suspect that many such rewrites are initiated by leads who get caught up in the hoopla of the latest industry trend. When C++ first started getting attention, there were programmers at Microsoft who wanted to recode anything and everything using object-oriented designs. It didn't matter that the original code worked fine. These programmers felt that it was absolutely necessary to rewrite the existing code. Fortunately, calmer minds prevailed, restricting object-oriented work to new code and to cases in which rewriting a product would provide strategic benefits.

Don't waste time on questionable improvement work. Weigh the potential value returned against the time you would have to invest.

The "Productivity" Cry

The reason I most often heard for rewriting existing C programs in C++ was that the development team would be so much more productive using object-oriented methodologies. That may be true, but the people making those claims were ignoring a significant detail: all the time lost doing the rewrite. Rewriting a C program to use object-oriented designs in C++ is not a line by line translation, as a Pascal to C translation can be; it's a total, ground-up rewrite.

If you're leading multiple groups and one of them comes to you wanting to move from C to C++, ask them whether they're talking about rewriting the application using object-oriented designs, or whether they're simply interested in using the more flexible C++ compiler to compile their existing C code. If they're talking about doing an object-oriented rewrite, be sure to determine whether the benefits would overcome the time lost doing the redesign and rewrite.

LET NOTHING INTERFERE

By now you should have a pretty strong awareness of the kind of work you should be focused on: the strategic work as defined by the project goals. But being focused on strategic work is not enough to prevent schedule slips. You can deflect "free" features, quash the impulse to go after "cool" features, and minimize effort on questionable improvement work. But if you don't learn to say No when you should or if you don't determine what others truly want, you can find yourself drowning in work that you shouldn't be doing.

The key to keeping your projects on track is knowing exactly what you should be doing and then letting nothing interfere with that effort. Of course, the trick is in knowing exactly what you should be doing. That's why it's vital that you create detailed project goals.

HIGHLIGHTS

- Don't let foreseeable problems surprise you. If you want your project to run smoothly, take time to look into the future. You can prevent many catastrophes by taking small actions today that either eliminate the problems in the future or steer you clear of them. If you regularly ask the question "What can I do *today* to help keep the project on track for the next few months?" you can determine the actions you need to take.
- Before you settle in to solve a problem, be sure you're attacking the right problem. Remember the misguided optimization work the dialog team was doing? The group complaining about the speed problem inadvertently misled the people on the library team. Get to the bottom of the problem before you try to treat it.
- Before spending any significant time on a task, do some research so that you know you'll be filling the real need. That request for those bizarre list boxes was misleading because the group really needed hierarchical menus. When you get requests, be sure to find out what it is the askers are trying to accomplish. It can save you lots of time.

- For a variety of reasons, some leads find it difficult to say No to demands made on their teams. In the most serious instances, a lead will commit to a ship date knowing the team can't make it. If you have trouble saying No, consider how you'd want groups you're depending on to respond to your own requests. Would you want to know up front that they couldn't make the date on which you need the feature, or would you rather they agreed and then missed that date? Be as responsible to other groups as you would want them to be to yours.
- Whenever you get a feature request, determine whether the feature is strategic to the product. If the feature isn't strategic, don't implement it. It doesn't matter that the feature appears to be "free" or that it's technically exciting or that a competitor has it. Especially watch for features that round out a set—such features can appear to be strategically necessary because it feels as though you must include them for completeness. If you're unsure whether a feature is strategic, consider the motivation behind the request for it.