# 1

# LAYING THE GROUNDWORK

Have you ever stopped to consider what makes one project lead or programmer more effective than another? Is it one or two profound pieces of wisdom, or is it a grab bag full of little snippets of knowledge that when taken together produce this thing we call "mastery"?

I wish the answer were that mastery comes from just one or two profound insights—that would certainly simplify training. The reality is that mastery is a collection of numerous little bits of knowledge, beliefs, skills, and habits that beginners have yet to accumulate. Ironically, none, or at least very few, of these little bits of experience are particularly hard to come by. But there are a lot of them, and they are often learned inefficiently, through trial and error.

Trial and error is the time-honored approach to gaining mastery, but that can be a long, arduous undertaking, even if you dramatically

speed the process through active study. A much faster method of jump-starting your skills is to take on the beliefs and habits of people who already excel in your area of interest. They've already learned what you want to know, so why go through all the trouble yourself when you can look at their practices, mimic them, and get similar results?

In this first chapter, I will describe what I have found to be the most important practices that project leads and their team members should embrace if they want to stay focused and hit their ship dates without having to work 80-hour weeks. These points lay the groundwork for the following chapters.

## Focus on Improving the Product

Companies pay programmers to produce useful, high-quality products in a reasonable time frame. But programmers often get sidetracked into doing work that has nothing to do with creating a product. They, or their leads, fail to recognize a basic truth of product development:

> *Any work that does not result in an improved product is potentially wasted or misguided effort.*

If you don't immediately see why this point is so important, consider two extremes. Which programmer is more likely to produce a useful, high-quality product in a reasonable time frame: the programmer who regularly attends meetings, writes status reports, and is buried in e-mail or the programmer who uses all her time to research, design, implement, and test new features? Is there any question that the first programmer's schedule will slip whereas the second, much more focused, programmer not only is likely to finish on schedule but may even finish early?

I've found that groups regularly get into trouble because programmers are doing work they shouldn't be doing. They're spending too much time preparing for meetings, going to meetings, summarizing meetings, writing status reports, and answering e-mail. Some programmers initiate this kind of activity themselves. More often, such distractions are at the behest of a misguided lead.

One lead with whom I worked required every team member to send a weekly e-mail message reporting on the status of his or her work. The entire team would then meet for an hour or so to rehash what everybody had been doing and to discuss any external issues that had cropped up. After the meeting, anybody who had offered new information would have to write those thoughts down in another piece of e-mail and send it off to the lead.

Now, this lead was just trying to be thorough. What he didn't realize was that he was choking his team with a lot of pointless process work. Was it really necessary to have both status reports and status meetings? And what about those follow-up reports? Were they really necessary, or could they have been eliminated in 99 percent of the cases if the lead had simply taken better notes during meetings?

Obviously, your answers to such questions will depend on your particular corporate environment, but in the actual case I've just described, the only process work that ever turned out to have any value was the initial status report. I don't remember a single status meeting that was worth the time it took to attend, and every time the lead asked for follow-up reports I winced, thinking, "Why? They just told you what they thought."

I was only an occasional visitor to these regular status meetings, so I wasn't often affected by the status work. I always wondered, though, how much other unnecessary process work that group was routinely saddled with.

In his well-intentioned zeal to be thorough, that group's lead violated what I consider to be a fundamental guideline for project leads:

> *The project lead should ruthlessly eliminate any obstacles that keep the developers from the truly important work: improving the product.*

There's nothing earth-shattering about this observation, yet how many leads do you know who make it a priority to actively look for and eliminate unnecessary obstacles?

If the lead I've been talking about had been actively trying to eliminate unnecessary work, I'm sure he could have come up with a much

simpler and more effective method of determining the state of his project. Having status reports, status meetings, *and* follow-up reports was overkill.

*Don't waste the developers' time on work that does not improve the product.*

***Don't Take Me Too Literally. . .***

When I say that developers shouldn't do any work that doesn't improve the product, don't take that imperative so literally that you keep them from doing their design and testing work and from getting the training they need. None of these activities contributes directly to a single line of code, but they all influence the quality of the products you release. If a developer thinks through and tosses out three flawed designs, for instance, that's far better for the product than having her implement the first design she comes up with.

And team interaction might not have much to do with improving the product, but getting the team together under pleasant circumstances can do a lot to improve morale and ultimately the quality and efficiency of the team's work.

## Run Interference

In your own groups, if you want to consistently hit your deadlines, you must protect your development team from unnecessary work. In particular, any time you find yourself about to delegate work to the entire team, stop and ask whether you can protect the team by doing the work yourself. If you have to present a project review to the folks upstairs, for example, is it really necessary to bring development to a halt and require that every programmer write a report summarizing what he or she has

done? Not in my opinion. As the lead, you should be able to compile that information without help, and you can then present the information more effectively since it's coming from one source. Yes, it might cost you a couple of hours of your own time, but that's better than disrupting the entire team for a task that does nothing to improve the product.

I often go a step further. If I find that a programmer is getting bogged down in a task that is necessary but that does not improve the product, I will take that task from the programmer, if I can, so that he or she can stay focused. There's no reason—except perhaps for training purposes—for programmers to answer project e-mail questions if they're questions the lead can field. Nor should programmers be attending meetings or writing reports the lead can handle, or better, eliminate altogether.

I know this advice contradicts what many management courses and books have to say about delegating work. I'm not saying that those courses and books are wrong, but you must be smart, that is, selective, about the tasks you delegate. If you're delegating work just to lighten your own load, you're probably hindering the development team with work that does not improve the product. Just because the other team members *can* do the work doesn't mean they *should* do the work.

Have you ever seen a house being moved across town? I don't mean the contents. I mean the house itself—pulled off its foundation and shifted to a large flatbed truck trailer. I like to think of projects as those houses in transit, and of the leads as the people who drive ahead, arranging to have overhead power lines disconnected and removing other obstacles that would block progress. These "leads" make it possible for the house to roll steadily toward its destination, not having to stop along the way.

While the house is rolling, the leads don't expect the truck drivers to pull over at every intersection to help the public utility people disconnect and reconnect the hanging stoplights. Nor do they ask the drivers to stop at tollbooths along the way, or to stop for meetings with the public utility folks who are moving the power lines.

Those "house leads" understand something that many software leads don't: if you want your project to move forward unimpeded, you

must actively search out and eliminate all obstacles to progress. Sure, the driver could pull over and pay the toll-taker—he is, after all, the one driving the truck. But doesn't it make more sense for the lead to take care of that task so that progress can continue unabated? Unfortunately, too many software leads delegate when they shouldn't, making their developers do the equivalent of negotiating with the public utility folks and pulling over to deal with the toll-takers. Their projects get slowed—or stopped—by every obstacle that comes along.

*Shield the development team from any work that interferes with progress.*

### *But I Lead Other Leads*

I've been assuming that you lead programmers; but if you lead testers, documentation writers, or some other type of team, your job is only slightly different from the one I've been talking about. The general idea is that you should make it possible for the members of your team to stay focused on their jobs, whether they're programming, testing code, or writing the manuals.

Even if your team is composed entirely of other leads, you should determine what their jobs should be and protect them from unnecessary distractions. Holding status meetings for leads can be just as wasteful as holding status meetings for programmers, particularly if the leads work on independent projects and don't need to know the status of other groups' projects. You may not be pulling those leads from the important work of directly improving their products, but in such cases you are pulling them from the important work of clearing obstacles to the improvement of their projects.

## THERE'S ALWAYS A BETTER WAY

As a lead, I'm always asking myself one question, in all phases of the project:

> *What am I ultimately trying to accomplish?*

I constantly ask this question because it's so easy to get sidetracked on work that isn't important. If you've ever spent more time formatting a memo—playing with fonts and styles—than you did writing the memo in the first place, you know what I mean. In the moment, you get caught up because the work seems important, but if you step back and get some perspective, you see that it's the message that's important, not how artistic you can make it.

We've already seen an example of misdirected effort in the status meetings and status reports I've talked about. How would you answer this question:

> *What am I ultimately trying to accomplish by holding status meetings and requiring status reports?*

Isn't the primary goal of gathering project status information to detect, at the earliest possible moment, whether the project is going astray? Think about that. Suppose all projects were finished exactly as scheduled—no project end date ever slipped, and nobody ever worked overtime. Would anybody ever gather status information? Of course not. There'd be no reason to.

If the ultimate purpose of status meetings and status reports is to determine whether a project's schedule is in danger of slipping, is it really necessary to pull the development team away from their work to collect this information? I don't think so. I have never held status meetings, and they are the first bit of pointless process I eliminate whenever I become the new lead of a group. I simply don't believe it's necessary to hold status meetings to determine whether a schedule is going to slip—that is, not if you're also collecting status reports.

So what about those status reports? How important are they? I think status reports—of some sort—are a necessary evil. A lead does, after all, need to know when problems occur. But note that while status

reports are necessary, they—like status meetings—do not improve the product in any way. When you believe that a task is necessary but see that it doesn't improve the product, you should always ask a more specific form of this general question:

*How can I keep the benefits of this task yet remove the drawbacks?*

Status reports do serve a valuable purpose, but they take time to write and can create a negative mind set in the team—at least they can the way they have been done in many Microsoft groups.

If each week programmers must write a report accounting for the hours they've worked and explaining why any tasks took more time than originally estimated, the status report causes unnecessary stress and engenders in the developers and everybody else the feeling that the

### *"Status Meeting" Defined*

No doubt, what I've been calling a status meeting is going to differ from one company to the next. When I say "status meeting," I mean those dreary get-togethers in which each team member describes what he or she did and didn't do that week. You can spot these meetings easily because the major point is to talk about what did and didn't get done.

Another type of status meeting is one in which leads from different teams get together and describe what they did and didn't get done. Although similar to the project status meeting, these meetings are held to coordinate multi-team projects. The leads don't report every little thing that happened—they report only those items that affect the other groups in the project. Did they miss or make a drop date? Are they still on track for some future date? Is another group now making demands on their time? The purpose of these meetings is to resolve *dependency* issues. Any team that is dependent on another team is in a precarious position as far as its own schedule is concerned, and it's essential that members know, at the earliest possible moment, when a team they're relying on is going to slip a schedule, cut features, or otherwise threaten their own project.

But again, notice that it's the leads who are meeting—not the programmers, who should be off working on their respective projects.

project is always slipping. More often than not, a programmer sits down to write the status report and realizes that she can account for only 27 hours of scheduled work yet knows that she worked seven 12-hour days that week. And she knows that she wasn't goofing off all that time.

If you've never been in this position, imagine how frustrating it would be to realize you've slipped your schedule even after you've put in a seven-day week, not to mention that you have to somehow account for your time. And suppose that the same scenario repeats itself week after week. Are you going to jump out of bed each morning, enthusiastic and ready to start another productive day? Or—more likely—are you going to be exasperated, frustrated, and depressed? Each week you work harder, trying to get more work done, yet you continue to slip. . .

I hate such status reports because they force the development team to focus on all the work they *didn't* do instead of putting the emphasis on what they *did* do. Rather than feeling enthusiastic because they are steadily improving the product, the team members are forced to remind themselves that they're behind schedule, screwing up in some way they can't immediately see. They know they're working hard, but they can't seem to keep from slipping.

A team isn't that different from an individual. If a team sees itself as on a roll, it will tend to keep rolling, but if a team sees itself as constantly slipping, the laws of inertia and self-fulfilling prophecy will apply there too, and that is ultimately demoralizing.

Don't misunderstand me: something is definitely wrong if a programmer is working 84-hour weeks but can account for only 27 hours of scheduled work. Perhaps she's agreeing to interview too many job candidates, or attending too many unnecessary meetings, or possibly she's too concerned about how her e-mail reads, so that she edits and re-edits replies that aren't really worth spending that kind of time on. You and she need to address those problems. But even if the programmer is having trouble allocating her time, that's no reason to have the status report regularly slap her in the face. As we'll see later, there are better ways to handle such problems.

Let's return to the earlier question: how can you keep the benefits of having status reports yet remove the drawbacks? One answer is to create a new type of status report, one that takes little or no time to put together and that also makes doing such a report a positive experience

rather than a negative one. I'm sure there are many alternative ways of achieving these goals, but this is what I ask my teams to do:

- Each time a team member merges a new feature into the master sources, he or she is to send a short piece of e-mail announcing the new functionality to the rest of the team.
- Anytime there's a possibility that a feature will slip, the team member responsible for that feature is to drop by my office to discuss the cause and brainstorm a solution.

That's it. A typical status report might look like this:

```
I just checked in the new search and replace feature.
It stomps on the S&R feature in FaxMangler! Check it out.
        - Hubie
```

Imagine how the team members would feel if they were constantly sending and receiving such positive e-mail. Quite a bit different than the hated status reports I talked about earlier would make them feel. Programmers actually enjoy sending little notes like this one—and nobody thinks of such a note as a status report.

When a programmer thinks the feature he or she is responsible for might slip, we talk about the cause and how it can be prevented in the future. Did we forget to schedule time for an important side item? Was the schedule too ambitious? Is a bug somewhere else in the product making this feature difficult to implement or test? Whatever the problem, we try to find some way to prevent it from recurring in the future.

The point is that I can easily gauge the project status solely on the basis of these two kinds of feedback. And if I have to, I can easily pass project status news up the chain of command—the individual programmers don't need to participate in that chore at all.

Even better, both types of feedback have secondary benefits. The first kind reinforces the perception among the team members that the project is continually improving, and the second creates a learning experience for the programmer and the lead. We don't just shrug and say, "Oh well, schedules slip all the time. It's no big deal."

Going overboard in gathering status information is just one example of how process work tends to expand and get formalized into grandiose procedures if people forget what they are really trying to accomplish. They get caught up in the process instead of the product.

Only when you're clear about what you and your team should be doing can you fulfill the project's needs with the least amount of effort and frustration. Review any task assignments that either are unpleasant or pull programmers from working on the product. Can you eliminate the unpleasant tasks, or at least find more enjoyable approaches to accomplishing them? And what about those tasks that don't contribute to improving the product? Get rid of them too, if you can—at least as far as the programmers are concerned.

*Always determine what you're trying to accomplish, and then find the most efficient and pleasurable way to have your team do it.*

### *Bombarded by Success?*

You'd think that if you asked team members to send little "check it out" notes to each other, the entire team would be bombarded by e-mail messages announcing their successes. In practice, the number of messages per day is small. The reason: people don't send these messages to everybody on the whole project, just to the lead and the four or five other programmers who are working on their specific part of the project.

One of the larger Microsoft teams might have 50 programmers, but that large team is typically subdivided into much smaller teams, with no more than 5 or 6 programmers working on any specific piece of the project. Each of these "feature teams" has a well-defined area of responsibility, a lead, and its own schedule. Programmers on feature teams are part of the larger team, of course, but on a day-to-day basis, their true team is the 4 or 5 other programmers with whom they share a common project goal.

In practice, you could be on a 50-person project yet receive only a handful of "check it out" notes on any given day—a steady, but not overwhelming number. Just enough messages to give you a sense of constant progress.

## STATE YOUR OBJECTIVES

How many people do you know who woke up one day to find that, miraculously, they had taken just the right courses to obtain a computer science degree? How many people do you know who accidentally packed up their houses and moved to new cities? Pretty silly-sounding. Clearly, people don't get college degrees or move across the country by accident. They plan to do those things. At some point they think, "I'm going to become a computer programmer" or "I'd like to live next door to Disney World." Then they take action to make those things happen.

Unfortunately, the random approach to goal achievement works all too well in many other areas of life. You can find a great job by chance, make a killing in the stock market with a lucky pick, and even, sadly, ship a software product without a goal more concrete than "We have to get WordSmasher finished."

In each of these situations, you can achieve the goal, but the question is, How much time and energy will you waste getting there? Are you more likely to get a great job by bouncing from one company to the next, or would it be more effective to take a day to determine what a great job would have to be like and then interview only at companies with jobs that meet your criteria?

One common trait I found among the half-dozen floundering groups I've worked with was that they all had vague goals. In one case, a group was providing a user interface library to 20 or so other groups at Microsoft. Not only was the group swamped with work, but the groups using the library were complaining about the size and bugginess of the code.

After the lead and I reviewed the library's task list, I asked the lead what his goals for the project were.

"To provide a Windows-like user interface library for the MS-DOS character-based applications," he said.

I asked him what else.

"What do you mean?"

"'Providing a Windows-like user interface library' is a pretty vague goal," I said. "Do you have more concrete goals than that?"

"Well, the library should be bug-free."

I nodded. "Anything else?"

He paused a moment and then shrugged. "Not that I can think of."

I then pointed out that a primary goal for any library is to contain only code that will be useful to all the groups using that library. The lead thought that point was obvious, but I wasn't so sure as we began to review the list of features he had agreed to implement.

I pointed to an unusual item near the top of the list. "What's this for?"

"The Works group asked for that. It allows them to. . . ." he said.

"Is it useful to any other group?"

"No. Just the Works group."

I pointed to the next item. "What about this feature?"

"That's for the CodeView team."

"And this item here?"

"Word wants that."

. . .

As we went down the task list, it became clear that the lead had agreed to implement every request that came in. He may have known that a library should contain only code that will be useful to all groups, but he wasn't using that criterion in his decision-making process.

The lead's goal for the library was simply "to provide a Windows-like user interface library." What if his goal had been a bit more detailed?

> *Goal: To provide a Windows-like user interface library that contains only functionality that is useful to all the groups who will use the library.*

With this slightly more precise goal, the lead would have seen that many of the requests from individual groups were inappropriate for a shared library.

After we reviewed the task list, I moved to another problem.

"Many of the groups are complaining that they have trouble linking whenever you release an updated library. What's causing that problem?"

"Oh, that's easy. They're forgetting to change the names of the functions in their source code."

I was confused, so I asked him to show me an example. In one case, he (or another programmer on the team) had fixed a bug in a function, and while he was at it, had changed the function's name so that it was

more consistent with the names of other entry points. In another case, a programmer had implemented a new function similar to an existing one. The programmer had then renamed the original function to emphasize the difference between it and the new function.

The lead didn't understand why the other groups were fussing—changing a name is simple. He had never stopped to consider that every time his group changed a name in the library, the 20 or so other groups that used the library would have to search through all their files, changing the names at all the call sites. The lead also hadn't realized that link problems reflected poorly on the library. If the team couldn't do something as simple as release a library that consistently linked, what, the other groups and I had to wonder, must their code be like?

If that lead had spent a moment looking at the library from the other groups' points of view, he would have seen that backward compatibility was important. Groups want to be able to take a new library, copy it to their project, and link. They don't want unexpected errors.

Again, a more concrete set of project goals could have prevented this link problem:

> *Goals: To provide a Windows-like user interface library that contains only functionality that is useful to all the groups who will use the library and that is backwards compatible with previous releases. . .*

Once I understood the issues affecting the user interface library, the lead and I worked out a complete set of goals. What's important to note is that all of the details were apparent, once looked for, and could have been established in advance. If the lead had bothered to ask the question "What am I trying to accomplish with this user interface library?" he could have derived a list of project goals in a matter of minutes.

A more thorough lead would spend several hours or even several days creating detailed project goals. The goals wouldn't have to be profound; they would just need to be written down and put in plain sight so that they could be a constant guide.

By ensuring that all new code would be useful to all groups, the library lead could have kept the library much smaller, he could have finished important features more quickly, and his team probably wouldn't

have had to work 80-hour weeks in a desperate attempt to deliver all the features he had promised. Think about that: just one refinement of the goal, and the course of the project could have been dramatically different.

*Establish detailed project goals to prevent wasting time on inappropriate tasks.*

### *Dependent on Dependencies*

One of the easiest ways for your project to spin out of control is to have it be too dependent on groups you have no control over. Using shared libraries is strategically important for a number of well-known reasons. But as a lead, you must weigh the benefits of leveraging such code against the drawbacks of not having control over the development effort. To keep the dependencies issue in mind—and in sight—you should make this one of the refinements of your project goals:

*Minimize the project's dependencies on other groups.*

Considering the damage a late library can do to other groups' schedules, a library lead owes it to his or her "customers" to be up front about the library's schedule and warn dependent groups the moment a slip seems likely.

Similarly, a development team relying on shared libraries should listen to a library lead who says a given request can't be fulfilled by a given date. By badgering library teams into accepting requests they don't think they can fulfill on time, pushy leads create not only dependencies for their projects but risky dependencies at that.

These two points are obvious. But having spent years turning around struggling library groups, I've seen both mistakes far too many times.

## Make the Effort

Management books often make setting goals sound like some mystical ideology you must simply have faith in: "We don't know exactly why setting goals works, but our studies show conclusively that groups who have concrete, detailed goals consistently outproduce those who don't—by a wide margin."

I don't know why such management books make the effectiveness of goal setting sound so surprising—goals simply help you compose a more vivid picture of what it is you're trying to do. If your goal is merely to buy a house, you're going to look at a lot more houses before finding one you like than if your goal is to get a turn-of-the-century, tricolor Victorian with four bedrooms, two bathrooms, and a statue of St. Francis in the back yard. The more detailed goal makes you more efficient because it allows you to instantly reject anything that doesn't match the picture in your head. Specific project goals work because they help you sift through the daily garbage that gets thrown at a project. They help you stay focused on the strategic aspects of your project.

Unfortunately, there's nothing in the software development process that forces project leads to stop and come up with detailed goals. In fact, there's plenty of pressure to skip the whole goal-setting process. Who has time to set goals when a project is out of control from the outset and already slipping like crazy? And some leads skip goal-setting for an entirely different reason: nobody else sets goals—why should they? Leads who don't set goals for either reason cause themselves, and their team members, a lot of unnecessary frustration.

If you want your group to run smoothly, you must take the time to develop detailed goals. It's usually not fun, but taking a day or two to set goals is a small price to pay for having a focused project. No group should have to work long hours under constant pressure—that's a symptom of unfocused work.

*Don't skip the goal-setting process simply because you think it would take too much time or because nobody else sets goals. The extra effort you exert up front will more than repay you.*

## KNOW YOUR CODING PRIORITIES

If you were to ask three different friends to drop by the supermarket to pick up some asparagus, green beans, and corn, would it surprise you to find that one friend bought canned vegetables because they were the cheapest, another bought frozen vegetables because they were easiest to cook, and the third bought fresh vegetables because they were organically grown and tasted the best? Can you at least imagine such a thing happening?

The three friends buy different types of vegetables for the same reason one programmer will emphasize speed in his code, another will emphasize small size in hers, and a third will emphasize simplicity—their choices differ because their priorities are different.

Suppose your product has to be blindingly fast but the programmers on your team are writing code with simplicity in mind. It's unlikely that those programmers are going to use fancy cache-lookups or other faster yet more complicated algorithms. Suppose that your primary goal is to create a robust application in the shortest time possible but the programmers are following their standard policy of writing highly optimized—and risky—code. Again, their misplaced priorities are going to thwart your goal.

Project goals and coding priorities are not the same thing. Goals and priorities do tend to overlap, mainly because the project goals help define what the coding priorities should be. Here's a good generalization:

- Project goals drive the direction of the project.
- Coding priorities drive the implementation of the code.

Obviously, if your goal is to create the fastest Mandelbrot plotter on the planet, efficiency is going to be a top coding priority.

Despite the importance of coding priorities, in my experience leads rarely convey their coding priorities to the programmers. Should the programmers focus on speed? On size? On safety? Robustness? Portability? Maintainability? Every programmer has his or her personal views about the importance of one coding priority over another and left to his or her own devices will produce code that reflects those views. It's common for one programmer, left alone, to consistently write code that's clean and maintainable while another team member, left alone, focuses

on efficiency even if the result is unreadable spaghetti code filled with obscure micro-optimizations and tons of assembly language.

If you want your team to achieve the project goals as efficiently and precisely as possible, you must establish and promote coding priorities to guide the programmers. At the very least, you should establish a ranking order for these priorities:

- Size
- Speed
- Robustness
- Safety
- Testability
- Maintainability
- Simplicity
- Reusability
- Portability

The only item on this list of priorities that may need some explanation is "safety." If you chose safety as a higher priority than speed, you'd choose one design over another because you'd think you'd be more likely to implement the feature without any bugs. Table-driven code, for example, can be slower than logic-driven code, assuming you're scanning the table and not doing a simple lookup, but table solutions are often much safer to implement than logic-driven solutions. If you chose safety as a higher priority than speed in this hypothetical situation, you'd implement the table solution unless there were overriding concerns.

In addition to ranking coding priorities, you should also establish a quality bar for each priority. If robustness is a high priority for you, how robust should the code be? At the very least, the code should never fail for legal inputs, but what about when the code receives garbage as input? Should the code take extra pains to handle garbage intelligently (trading both size and speed for robustness), should the code use program assertions to check for garbage, or should you let Garbage In, Garbage Out rule? There is no right answer to this question; the answer depends on what you're doing.

An operating system should probably accept garbage without crashing; an application program in which an end user can make mistakes entering data most certainly shouldn't crash. But if you're talking about a function deep in the guts of your program, where the only conceivable way the function could get garbage inputs would be if there were a bug elsewhere in your code, an assertion failure would be more appropriate. In such a case, you might still choose to handle the garbage safely if it didn't cost much extra code.

The point is that you must decide, in advance, what the coding priorities and quality bars will be; otherwise, the team will have to waste time rewriting misconceived or substandard code.

*Establish coding priorities and quality bars to guide the development team.*

### *Safety vs. Portability*

In my own priority lists, I usually make safety a higher priority than portability—I'd rather have correct code than portable code. This has led to some confusion because portable code is often seen as the safest code of all. In fact, the two priorities aren't really linked; it just happens that portable code is usually quite safe given the constraints that govern the writing of such code.

When writing C code, programmers commonly write macros that look and behave as though they were functions. The problem is that these "macro functions" can cause subtle bugs if they're not written carefully, and even when they're written carefully, they can cause other bugs if they aren't "called" carefully. The problem is well known among experienced C programmers. Macro functions are beneficial but risky.

You can gain the benefits of macro functions without the risks if you're willing to use the nonstandard *inline* directive found in some C compilers. The only cost is that the *inline* directive is not universally portable. Safety over portability. . .

### *Snap Decisions*

You've probably heard that most extremely successful people have a tendency to make on-the-spot decisions. That may seem contrary to what you'd expect—you'd think that people who make snap decisions would fall flat on their faces most of the time. But the difference between these accomplished people and the average person is that they have concrete goals and clear priorities. If you hand such people a problem or a proposal, they instantly measure it against the goals and priorities etched in their brains, and you get an instant answer. The clarity of their goals and priorities also accounts for the other well-known trait of such people: they rarely change their minds once they've made a decision. Changing their minds would mean betraying what they believe in.

These successful people are not actually making snap decisions—that idea implies that no thought is involved. It's simply that these people know their goals and priorities so well that they don't have to wade through all the possibilities that don't match their criteria. The result: they spend their time acting on their decisions, not deliberating over them.

## Stick to the Basics

If you look back at the points raised in this chapter, you'll see that they boil down to a simple formula for software development: figure out what you're trying to do and how you should do it, and then make sure that every team member stays focused on the project goals, coding priorities, and quality bars you've come up with. Pretty basic stuff.

Now take a look at the teams in your company. How many have detailed goals for their projects? In how many do the programmers understand exactly how they should be writing their code and to what standards of quality? Then ask yourself, "Are the programming teams focused fully on improving their products?"

Now look at the project leads in your company. Do they habitually call meetings to discuss every little thing, or do they reserve meetings for truly important issues? Do they put obstacles in the programmers'

paths—asking them to write questionably useful reports, for instance—or do the leads strive to remove obstacles to development work?

The points in this chapter are basic, but in my experience few groups focus on these fundamental concepts. And that, I believe, is why you can't pick up *InfoWorld* or *MacWEEK* without reading about some project that has slipped another six months or on which the programmers are working so hard that they don't even bother to go home at night.

## HIGHLIGHTS

- Companies have hired their programmers to focus on creating high-quality products, but programmers can't do that if they're constantly pulled away to work on peripheral tasks. Make sure that every team member is focused on strategic work, not on housekeeping tasks; look for and ruthlessly eliminate any work that does not improve the product.
- Unfortunately, some housekeeping work is necessary, at least in larger companies. One way to keep such work to a minimum is to regularly ask the questions "What am I ultimately trying to accomplish?" and "How can I keep the benefits of the task yet eliminate the drawbacks?" Fulfill the *need,* not some overblown corporate process.
- The benefits of establishing specific goals might not be easy to see, but it's easy to see the chaos that ensues in groups that don't set such goals. Yes, creating detailed goals can be tedious; but that up-front work is much less painful than leading a project that slips two days every week. Keep that user interface library project in mind. One small refinement of the project goals could have prevented that project from turning into the pressure cooker it was. A second refinement could have made it fly.
- Every team member needs to know the coding priorities. Is maintainability important? What about portability? Size? Speed? If you want the code to reflect the goals for the product, you must tell programmers what trade-offs to make as

they implement features. You must also establish quality bars so that team members won't waste time writing code that will have to be rewritten before you ship. The earlier you define the quality bars, the earlier you'll minimize wasted effort.