7

It's All About Attitude

In Chapter 6, I emphasized how important it is that you work with team members to improve their skills and knowledge. Exposing team members to new kinds of tasks promotes incremental learning, and getting the programmers to read books and develop new coding habits makes for even more impressive results. But the most profound improvements come about when a team adopts new attitudes about how to develop products.

Buggy Attitudes

As I said in Chapter 1, the job of the professional programmer is to write useful, bug-free code in a reasonable time frame. A key point in that idea is that the code be "bug-free." Unfortunately, writing bug-free code is hard. If it weren't, everybody would write bug-free code.

One pervasive attitude in programming shops is that bugs are inevitable and there's not much you can do about them except to fix them when they show up. While common, that attitude is completely wrongheaded. Programmers can make great strides toward writing bug-free code, but it requires extra effort, effort that programmers won't willingly make until they internalize the attitude that writing bug-free code is critical to product development.

One simple—and obvious—technique I use to catch an entire class of bugs is to turn on the compiler's optional warnings, the ones that display an error message for correct, yet probably buggy, code. For example, many C compilers have an optional warning to catch this common mistake:

```
if (ch = tab_char)        /* Note single = sign. */
    ...
```

The code above is perfectly correct C code, yet it contains a bug that the compiler can detect. The tab character is being assigned to *ch* when what the programmer intended was to *compare* the tab character to *ch*:

```
if (ch == tab_char)       /* Note double = sign. */
    ...
```

Enabling just one commonly supported compiler warning would allow the compiler to flag all such erroneous assignment bugs, yet I've worked with many programmers who absolutely refuse to use that option. The programmers feel that the warning interferes with writing code because the compiler gives them a warning even when they intentionally make an assignment in an *if* statement, forcing them to rewrite their code. Instead of writing

```
if (ch = readkeyboard())
    process character typed by the user
```

which would generate a warning, they would have to make a slight change, having to write either

```
ch = readkeyboard();
if (ch != nul_char)
    process character typed by the user
```

or the more terse

```
if ((ch = readkeyboard()) != nul_char)
    process character typed by the user
```

Neither of the two work-arounds would generate any additional object code because both simply make the test against the nul character explicit instead of implicit. And to most C programmers, either of the work-arounds is as clear as the original code—possibly more so if a programmer is reading the code quickly.

But some programmers are adamant. They refuse to use optional compiler warnings. "I should be able to write code any way I want," they say. "The compiler should never issue a warning for perfectly legal code." Given the intensity with which some programmers talk about this issue, you'd think I was suggesting that they give up their desktop PCs and go back to using punch cards.

This issue points up a difference in programmer attitudes toward bugs. Since I habitually use the compiler work-arounds, I never get warnings unless I've actually created a bug by mistake—and I want to know when I've made such a mistake. To me, being able to find bugs easily is far more important than what I view as an inconsequential style change. Programmers who refuse to enable any compiler warnings, it seems to me, are more concerned with personal expression than with detecting bugs. If those programmers aren't willing to make such minor changes, what are the odds of their making more critical changes? Would they adopt the team-wide or company-wide naming or coding style? Would they agree to give up favorite but error-prone coding tricks? Would they even entertain the idea of stepping through all their new code in a debugger to detect implementation bugs at the earliest possible moment?

Yes, writing bug-free code takes effort, effort that programmers won't make unless their attitude is that bugs are simply unacceptable.

On my own projects, I review every reported bug, keeping an eye out for bugs that should have been caught by someone's using the project's unit tests or stepping through the code with the debugger. Any programmer who allows such bugs to get into the master sources needs more training—he or she is failing to meet the quality bar.

Novice programmers tend to give up far too early because they have the basic attitude that their code probably *doesn't* contain bugs:

> *I'm done because the code compiles without error and appears to run correctly.*

Novice programmers have that attitude because they haven't yet been caught over and over again by overflow and underflow bugs, signed and unsigned data type bugs, general type conversion bugs, precedence bugs, subtle logic bugs, and all the other bugs that go unnoticed when novices read code in the editor and that show up only for special cases when they run their code—cases they haven't yet learned to test for.

Fix Bugs Early

The primary reason I push hard for programmers to step through their code the moment they write it and to run their unit tests is that it takes so much less time than letting even a single bug slip by and find its way into the product's master sources.

The moment a bug makes it into the master sources, it not only hurts the product but costs everyone huge amounts of time. The programmer on her end has to stop working on features and track down the bug, apply a fix, test the change (we hope), and report the bug as fixed. Back to Testing. Since a bug was found, the testers must retest the entire feature to ensure that the fix works and that the fix hasn't broken anything else. Then they must write a regression test for the bug. If the regression test can't be automated, a tester must manually verify that the bug has not returned in every future testing release.

Compare all that effort expended on a single bug to the effort it would take for the programmer to step through the code and run the unit test before ever merging the feature into the master sources. If the programmer finds the bug before sending the feature to Testing, none of that protracted effort I just outlined is necessary. That's why I say that it's so much cheaper for programmers to find their bugs before the testing team ever sees the code.

Experienced programmers who consistently have low bug rates have learned that they're more likely to find Bigfoot slurping ice cream at the local Baskin-Robbins than they are to write bug-free code. Unlike the novices, such experienced programmers assume that their code probably *does* contain bugs:

Until I find all the unknown bugs in this code, I'm not done.

It might seem that with such an attitude, programmers could go overboard in testing their code, but I've yet to see that happen. Anybody who is smart enough to write programs realizes when he or she is wasting time on redundant tests. Somebody smart enough to write programs doesn't always realize, though, when he or she isn't testing thoroughly enough. It's hard to know that you've forgotten to test a unique scenario or two.

Be sure programmers understand that writing bug-free code is so difficult that they can't afford not to use every means to detect and prevent bugs.

Resisting Effort

One question I regularly ask as I review both designs and implementations is "How error-prone is this design or implementation?" I look for weaknesses and try to judge how risky the code would be to modify. When I find a weakness, I take steps to overcome it, by either changing the design to get rid of the weakness or introducing debug code into the program to monitor the implementation for trouble.

I once reviewed a new feature that had been implemented using a large table of numbers. I like table-driven implementations, as a rule, because they're usually concise and less prone to errors, but they do have a weakness in that the data in the table could be wrong. I pointed this weakness out to the programmer who had implemented the code for the feature and asked him to add some debug code to validate the table during program initialization. Without thinking, the programmer blurted, "Writing that code will take too much time!"

Klaxons blared. Red lights flashed. Flares went skyward.

Those alarms went off because that programmer committed what I consider to be a fundamental error in intelligent decision making: he didn't ask himself whether my request made sense. Instead, he pounced on how much extra time he thought writing the debug code would take.

That programmer's first response should have been "Does the request make sense?" His second response should have been "Does it fulfill the project goals and coding priorities?" The question whether the task would take too much time or effort should have come third in the order of evaluation.

After the programmer had calmed down, I explained my objections to his decision-making strategy and asked him to start evaluating requests according to the order of questions I've described:

- Does adding the debug code make sense?
- If so, does adding the debug code fulfill the goals and coding priorities of the project?
- Finally, is adding the debug code important enough to justify the time that will have to be spent doing it?

After we stepped through this evaluation process, the programmer—still reluctant—agreed to implement the debug code.

Thirty minutes later he came into my office, having added the debug code to the program, and showed me three potential problems in the table that the debug code had flagged. Two of the problems were obvious bugs—once they had been pointed out. The third problem was confusing: neither he nor I could see the bug the debug code was reporting. We thought at first that the debug code itself might have a bug, causing an invalid report. But if the debug code was buggy, that bug wasn't obvious to us either. We pondered the suspected bug for nearly 10 minutes before we finally realized that the data in the table was indeed wrong. That bug was hard to spot even though the debug code pointed right at the erroneous table entry. Imagine how hard the bug would have been to spot without the debug code to lead us to it.

That programmer learned two valuable lessons that day. First, that it's still worthwhile to add debug support to code you already think is bug-free. And second, that the first reaction to any proposal should never be "That will take too much time" or its disguised sibling, "That's too hard (and would therefore require too much time)."

Watch out for and correct the "it's too much work" reaction. Train programmers to first consider whether the task makes sense and whether it matches up with the project goals and priorities.

Can'ttitude

I've worked with many programmers—and project leads—who hardly ever hit upon new ideas or employ new development strategies because they shut down their thought processes before they ever get started. Have you ever been at a meeting in which some poor soul proposed a new idea only to be bludgeoned by the others with all the reasons the idea couldn't possibly work, with how impossible it would be to get upper management to agree, or simply with the bald "You can't do that! It's never been done before!"

This "can't attitude"—can'ttitude—is so destructive to creativity and problem solving that I try to discourage it whenever I run across it. I have a rule—and in this case it *is* a rule—that nobody on my teams is allowed to say that something can't be done. They can say it would be "hard" or that it would "take tons of time," but they can't say "can't." My reason:

When somebody says that something can't be done, he or she is usually wrong.

I learned long ago to disregard most claims that you can't do such and such. More often than not, the person who says that hasn't given one iota of thought—at least not lately—to whether you really can't. Yes, of course, you can come up with hundreds of hypothetical, and absurd, situations in which something can't be done—getting all 2704 known bugs fixed by noon tomorrow, for instance. But usually when people make suggestions that get shot down with *can'ts*, the suggestions aren't absurd; if they were, the people wouldn't have proposed them.

Whenever you hear somebody say that something can't be done, ask yourself whether that person seems to have given any real thought to the question. If you know the person has, consider whether his or her evaluation is dated. Things change, especially in our industry. Maybe what couldn't have been accomplished last year can be accomplished fairly handily now—particularly if the proposal revolves around a size or speed trade-off. There was a time, after all, when people maintained, "You can't do a graphical user interface. It would take tons of memory and be unbearably slow." That was once true. Now it's not.

Sometimes it's a political or administrative matter that meets with the *can't* resistance. Microsoft leads will tell you that you can't give back-to-back promotions or a raise bigger than the biggest allowed, but I've done both of those things in exceptional circumstances. Was it easy? Definitely not. I had to go out of my way to prove that what I was asking for was in the best interest of the company. I was successful because what I asked for made sense, despite corporate policy. Those accomplishments weren't impossible to achieve, just hard.

Many times people latch onto the "can't be done" attitude simply because whatever you're talking about is outside their experience.

In 1988, when we were nearing completion of Microsoft Excel 1.5 for the Macintosh, upper management was already talking about the 2.0 release. The plan was that the Macintosh team would continue to port features from the Windows version of Excel, implementing look-alike features when the Windows Excel code couldn't merely be swiped and reworked to fit. Having spent two years doing just such work, I wasn't thrilled with the idea. I felt there were too many problems with that approach. Despite their external similarities, there were numerous differences between Excel for Windows and Excel for the Macintosh because they were, in fact, two different bodies of code. I also felt that Excel for the Macintosh would never be on a par with its Windows sibling. The Windows product was already considerably more powerful than the Macintosh product, and their team was larger than ours—a recipe for ever-widening feature disparity and incompatibility.

There was also a serious problem with the Macintosh implementation. Because of a design decision that had a pervasive influence on the code, the Macintosh application couldn't use more than 1 MB of RAM. Even worse, the code had to reside in the *first* 1 MB of RAM. Users were complaining loudly—why couldn't Excel use the other 7 MB of RAM in their systems? Outrageous!

Programmers at Apple Computer discovered Excel's predilection for low memory addresses as they were developing MultiFinder, their then-new multitasking operating system. The Apple programmers had designed MultiFinder to load applications from the top of memory down, but they discovered that Excel wouldn't work unless it was loaded at the very bottom of memory. Around their shop, Excel became known as "the application afraid of heights." To get Excel and MultiFinder to work together, Apple's programmers included special code in MultiFinder to look for and accommodate Excel, uniquely loading it into low memory. And they asked Microsoft to work on Excel's acrophobia, a phobia that had already been "cured" in the Windows version of the product. In fact, the Windows Excel team had done a line-by-line rewrite of the product and fixed numerous problems, with the result that their code far surpassed the Macintosh code in quality and maintainability.

When I looked at the 2.0 development plan to rip out Macintosh Excel's guts to fix the 1-MB problem and to port as many Windows Excel features as possible, I saw that the Macintosh team members would be spending all their time duplicating work that the Windows team had long ago completed. And we'd still end up with a somewhat incompatible and far less powerful product than theirs. That seemed like a big waste of time to me.

Why not instead, I thought, expend half as much energy to create a multi-platform version of Excel from the existing Windows sources? I'd spent years writing multi-platform code before joining Microsoft, so I knew what the challenges were in writing such code, and I couldn't see any reason why the Windows Excel code couldn't be modified to support the Macintosh. If we took that approach, I reasoned, the Macintosh product—being built from the same code—would be just as powerful as the Windows product and fully compatible. The 1-MB memory restriction would disappear, and instead of having to invest in the full development effort that would otherwise be required, Microsoft would be able to create future Macintosh releases at a fraction of the previous development cost.

When I talked to upper management about scrapping the 2.0 development plan in favor of creating a multi-platform version of Excel, they asked me to take a week to review the Excel for Windows sources and write an attack plan proposal for the work.

A week later, after I had released the attack plan to upper management and both Excel teams, I was taken aback by all the objections to what I proposed. Even though the attack plan was straightforward, people focused on all the problems they felt couldn't be overcome. I was surrounded by can'ttitude.

"Maguire is dreaming," said one programmer. "Windows and the Macintosh are just too different," said another. A third said, "Assuming we could create a multi-platform product, it would ruin Excel. The code would be too slow and too fat and wouldn't take advantage of the unique features of each platform." Still another said, "We don't have the time now. We should wait until after the next release"—as if there would be time *then*. One person even threatened to quit the company if we chose to take on the amount of work he thought it would take.

I had been expecting the plan to be wholeheartedly embraced. I got an education that day. I learned that fear of the unknown can affect even the best and most self-assured development teams.

A few days later, the Excel teams met with upper management, there was a vote—the only vote I ever saw at Microsoft—and the plan was shot down. There would be no multi-platform product, and work on Macintosh Excel 2.0 would go ahead as planned.

I was still reeling from the decision when we got word that Bill Gates, Microsoft's CEO, had read the proposed attack plan and thought that the multi-platform approach made sense. The work was on.

The team went on to do the multi-platform work in just eight months. And the application never fell prey to all those early concerns people had expressed. It's true that a few operations were a bit slower in the multi-platform version of Excel than in the original Macintosh version, but the slowdown was the result of lifting the 1-MB restriction, not of the multi-platform work. The product's speed would have been affected by the lifting of the restriction either way.

The Excel programmers were rightly proud of their accomplishment, and many went on to help other Microsoft project teams implement multi-platform code.

Don't let can'ttitude discourage innovation.

Don't Bring Me Problems! Bring Me Solutions!

The problem with can'ttitude—if there's enough of it—is that people stop speaking up when they see an opportunity for innovation, or worse, when they see a problem that needs to be fixed. Sadly, some project leads go out of their way to shut down people who would otherwise raise valid concerns. Have you ever been at a meeting in which somebody raised a problem and the lead barked back, "Don't bring up any problem you don't know how to solve—it wastes our time"?

Unfortunately, that approach leads team members to clam up until they can think of solutions for the problems they've noticed. A programmer could spot a serious problem affecting development but, not knowing how to solve the problem, might never bring it up for fear of getting a crushing and humiliating response.

Leads who insist that team members can't bring up any problems they don't know how to solve should instead realize that all problems need to be raised regardless of whether there is a known solution. Would you want a worker at a nuclear plant to clam up because she didn't know what to do about the green goo she found leaking from a critical part of the reactor? Of course not. She might not know how to handle the goo, but somebody else on the reactor team probably would know or would certainly be motivated to find a solution quickly.

Why should development teams be run any differently? Even if the person who brings up the problem doesn't have a solution, somebody else on the team might be able to come up with one. Problems that aren't brought up are problems that don't get solved.

IT'S GOOD ENOUGH FOR USERS

Occasionally I'll run into a programmer who thinks he or she is unique in requiring things from a product that mere users don't need.

One time I asked a programmer to demonstrate an important feature he had just completed. He launched the application and began showing me how the feature worked. The feature looked sharp, except that it seemed sluggish.

"Are you running the debug version of the code?" I asked, thinking that debug code must be responsible for the poky response.

"No, this is the ship version." He went on demonstrating.

"Have you thought about how to speed things up?"

"What do you mean?"

"I mean, don't you think the code is a bit slow?"

"Well, I wouldn't like it, but it'll be OK for the users."

I was shocked. "What makes you so different from the users? Especially in this case, when the users are other programmers *just like you?*"

I have never understood why some programmers think that users—whether they're other programmers or gourmet pasta shop owners—are any less concerned about speed and other aspects of quality than the programmer who wrote the code.

I'd argue that end users are *more* particular about speed and other aspects of quality since they actually use the features, whereas the programmers who write the code often don't. Do you think the programmers working on Microsoft's FORTRAN compiler use FORTRAN when they write code? Do the programmers who worked on Word's Mail Merge feature ever use that capability? What about Excel's macro language? Dozens of programmers have extended the macro language over the years, but how many have ever written their own user-defined macros? I'm not saying that all of these programmers are guilty of the gross disregard for the user expressed by that earlier programmer. That simply isn't so. My point is that programmers routinely implement code that they themselves never have occasion to use. Think about your own project. Do the programmers on your team actually use the code they write?

When programmers don't use the code they write, it's easy for them to distance themselves from the end user. This distancing may account for the occasional programmer who thinks that end users are bozos who aren't concerned about speed and other aspects of software quality—at least not to the same degree that the programmer himself would be.

To keep the end user in mind, programmers should measure their work against this reminder—you might want to put it on a large banner you hang over the entrance to your building:

> *The end user is at least as concerned about speed and other aspects of software quality as the programmer who implements the code.*

We all know that some users don't care much about the quality of the programs they use, as long as they aren't prevented from getting

Usability

When Microsoft first began conducting usability studies in the late 1980s to figure out how to make their products easier to use, their researchers found that 6 to 8 out of 10 users couldn't understand the user interface and get to most of the features. When they heard about those findings, the first question some programmers asked was "Where did we find eight dumb users?" They didn't consider the possibility that it might be the user interface that was dumb.

If the programmers on your team consciously or unconsciously believe that the users are unintelligent, you had better correct that attitude—and fast. Consider two teams, one on which the programmers believe that users are probably intelligent, discerning consumers and another on which the programmers assume that users are essentially dumb. Which team is more likely to take users' complaints seriously and act on them to improve the software? Which team is more likely to ask users for their opinions about new features that would improve the product? Which of the two teams is going to consistently put out a product that fits the users' needs? On the other hand, which team is more likely to ignore users' complaints and instead waste time on features that the users don't need or want? The basic attitude the team adopts toward the users can make a great difference in the quality of the product.

their jobs done. But if you want to ship great products, you can't target those unfussy people. You must target the users who do care whether a program is slow or quirky or contains bugs that can make it crash.

Don't let programmers believe that users don't care as much about software quality as programmers do.

Beware the Substandard Feature

I used to have the attitude that it was better to give the user a painfully slow feature, or an overly restrictive one, than to cut the feature and give the user nothing at all. "At least the user will have something between now and when we ship the more polished version in the next release," I'd reason. Eventually it dawned on me that users weren't aware of the choice I'd made—giving them something, even of substandard quality, over giving them nothing at all. Users, I realized, open the box, run the program, and see only that they've gotten another poorly implemented feature. "Why does it always take them two releases to get things right?" they wonder.

I've seen this reaction often enough now that rather than trying to give the user something, I cut any feature that doesn't meet the quality bar. Users rarely miss what they've never had, but if you give them a feature they feel is unpolished or frustrating to use, they're liable to think less of the whole program. If you give them several such features, they might start looking at your competitor's product.

It pains me to say this, but if a feature doesn't meet your quality bar, consider cutting it, even if it seems as if it could be a useful feature. Wait until the next release, and do it right. If the feature is so strategic that you feel you must ship it, it's also probably worth slipping your ship date to do it right.

Don't ship substandard features. Postpone them until you can implement them properly.

The Sensitive Programmer

In Chapter 1, I described a situation in which a lead for a Windows-like user interface library had never bothered to view the library as one of the library's "customers" would. The lead had never considered the possibility that a library that wasn't backwards compatible would be frustrating to its users. I've seen this lack of appreciation for the users' perspective so many times that it's worth talking about.

When the Windows Excel team was rewriting parts of the application so that it would work on the Macintosh, one programmer was implementing keyboard-driven menus, a capability many business users were asking for that the Macintosh operating system didn't offer. Macintosh users were required to use the mouse. Since there was no Macintosh standard for keyboard-driven menus to follow, the programmer implemented Windows-style keyboard-driven menus to minimize the user interface differences between the Windows and Macintosh versions of the product. When the programmer finished the feature, he called me into his office to demonstrate his new creation. The menus looked just as they did in Windows. I was impressed.

"Wow!" I said as I played with the menus. When the excitement wore off, I turned to the programmer: "How do I disable the Windows interface?"

"Why would you want to do that?" he said, puzzled. "The feature doesn't interfere with the Macintosh mouse-driven interface. There's no reason to disable the interface."

I was surprised by the programmer's response because, at the time, you couldn't pick up a Macintosh-oriented magazine that wasn't full of hatred for Windows. Macintosh users were upset that the industry was raving about Windows, which they considered a third-rate product, and that their beloved Macintosh was viewed as a whimsical toy. Windows was the archvillain to Macintosh users everywhere.

"If Excel ships with Windows-style menus as the default," I said, "it'll alienate Macintosh users. Excel will get killed in reviews if it has 'Windows' written all over it."

The programmer was reluctant to change his code—he'd been thinking he was done and was eager to move on to the next feature. We called over some other team members to talk about the interface. The consensus was unanimous: Excel for the Macintosh not only had to look like a Macintosh product right out of the box but had to bleed Apple's six colors as well. The programmer went back to work.

A while later the programmer emerged from his office, offering to demonstrate his new version of the feature. I was surprised to see that he hadn't merely added an on/off switch for Windows-style menus. He had implemented a smart feature in which the menus were drawn in standard Macintosh format by default but were redrawn as Windows-style menus the moment the user hit the lead-in key for keyboard-driven menus. The menus remained in Windows mode until they were dismissed; then reverted to Macintosh-style menus. Even better, the programmer responsible for implementing the new Macintosh dialogs carried the feature into that code as well. When you invoked a dialog using the mouse, you got a standard Macintosh dialog; when you invoked a dialog by means of a Windows-style menu, the dialog came up with the Windows-style interface. The best of both worlds.

Be sure that programmers always view the product as an end user would. Programmers must be sensitive to the end user's perceptions.

The Whole Product and Nothing But

For the longest time, Microsoft's Languages division—the division responsible for compilers, debuggers, linkers, and so on—viewed the tools as separate, autonomous products. That made sense from a development viewpoint, but it didn't make sense from an end user viewpoint. Programmers who bought a Microsoft development system didn't care whether the compiler and debugger development teams were different. From their viewpoint, Microsoft C/C++, the debugger, and the linker were parts of the same product. Pretty easy to understand.

Unfortunately, that wasn't the predominant attitude toward the tools in the Languages division. Programmers, both external and internal, were asking for improved debugging features, but the debugging team didn't have enough people to fill the requests. Meanwhile, the compiler team was merrily working on code optimizations that few people were asking for. The mindset was "We've got to keep improving the compiler." It should have been "We've got to improve the overall product."

For years, Microsoft's linker was clunky, slow, and tedious to use while competing products had fast linkers. Every programmer in the company knew that Microsoft's linker crawled, but very little was done to improve it. The one programmer assigned to the linker did his best to improve the tool, but he had other duties and didn't have time to make major speed improvements to the linker. Besides, the view in the Languages division seemed to be, it was the compiler that was important—the linker was just a support tool. Users didn't see it that way, though, because they didn't distinguish between the compiler and the linker. To users, they were part of the same product.

At least one Microsoft team dumped the company's own linker and used a competitor's linker. And in the Applications division, a programmer finally got so frustrated with the linker that he hacked together a quick and dirty incremental linker for the Applications teams to use. The Languages group eventually discovered the Applications incremental linker, cleaned it up a bit, and began shipping that linker with retail releases of the compiler.

Eventually, after a few rounds of management change, the Languages group caught on and began improving the *development system,* not just the compiler. The result was Visual C++, a product that reviewers hailed as a refreshing, long-needed change to Microsoft's development system.

The product is everything *that goes into the box.*

DOUBLE MEANS TROUBLE

As the Excel programmer was writing his keyboard-driven menu code, a Word programmer not more than ten doors away was implementing the same feature in Word for the Macintosh. Although I pointed out this duplicate effort to the Excel programmer and mentioned it to the manager in charge of both Excel and Word, nothing happened. The two programmers continued to implement the code in parallel. When the products eventually shipped, both sported keyboard-driven menus, but the user interfaces were totally different. I saw that as a lost opportunity to make the Excel and Word interfaces work identically, to save half the development effort, and to create a menu library that Microsoft's other Macintosh teams could have popped into their products. The attitude wasn't so much "not invented here" as it was indifference. Nobody seemed to be concerned that programmers were duplicating effort and creating unnecessary differences between products.

I take the other approach to development effort: if I can reuse code that has already been written and debugged, I'll grab it in an instant. Similarly, I always write code assuming that some other team is going to borrow it in the future. No, I don't write all my code so that it's portable, nor do I spend extra time just in case the code might be reused. But if I'm faced with the choice between two equally good designs, I always choose the design that can be more easily shared.

In Excel's initial release, one of the programmers implemented a feature never seen in a Macintosh application before: a "print preview" feature that enabled the user to view pages on the screen formatted as they'd actually be printed. The design for the print preview feature was straightforward. The "page viewer" would take a "picture" of a page and then display it. If the user wanted to preview a full document, another piece of code simply called the viewer to display pictures of successive pages.

The feature was such a hit with users that the Macintosh Word team added a print preview feature to their application, one with a much nicer and more useful page viewer. The Word implementation made Excel's look rough and unpolished. I was assigned the task of adding many of Word's bells and whistles to the Excel version.

My first thought was to scrap the Excel print preview code and transplant Word's implementation into Excel. Not only would transplanting take less time than implementing all the new code, I thought, but transplanting the code would make the two applications look and behave identically. When I explained to the Word programmer what I intended to do, he told me that his implementation of the print preview feature was inextricably tied to Word. He could have written the code to be more shareable, he said, but it had never occurred to him that we might want the code for the Excel project. After all, Excel already had a print preview feature. Sadly, I couldn't use his polished page viewer.

In the end, I enhanced Excel's existing print preview code, but the Word feature was still much nicer. Even more disappointing, because Excel's code was shareable, its version of print preview was the version that spread to Microsoft's other applications.

As I've said, one of the best ways to implement a solid new feature is to grab it from a team that has already done the work of writing and debugging the code. Most programmers appreciate this point. But most programmers, it seems, fail to recognize that they can't grab code unless they and other programmers write their own code so that it *can* be grabbed.

To increase the value of their code to the company, programmers should develop the attitude that all of their code is likely to be reused. With that objective in mind, they should reduce the code's dependence on the host application. It's a problem not unlike writing code to avoid explicit references to global data: sometimes it's necessary, but often by using a slightly different design you can eliminate the explicit dependence with little or no extra effort.

Programmers should ask,

Could this code be useful to other (even future) applications?

If the answer is Yes, the code is a candidate for reuse. Both the keyboard-driven menus and the print preview feature could have been coded in an application-independent way. Reusability just wasn't considered a priority. Too bad. It could have increased the quality of both Word and Excel, with *half* the effort.

Give some priority to writing easily shared code. Programmers can't share each other's code unless they're writing it so that it can *be shared.*

LEVERAGE YOUR LEVERAGEABILITY

If your team or company is to become successful, you have to ensure that people understand the power of leverage, how a little well-placed effort can yield a much greater return. Every team member should keep this fundamental principle in mind:

> *You can extract extra value from every task you do by either using existing leverage or creating new leverage.*

The one example of this principle that all programmers know about is reusing existing code or creating reusable code. But there are many ways to use or create leverage.

In Chapter 6, I described how you could make employees more valuable to the company by first teaching them skills they could use not just on your project, but on any project. That's creating leverage. As far as your project is concerned, the order in which you teach worthwhile skills doesn't matter. The order in which you teach skills is unimportant until a programmer moves to a new group. Then either the programmer must start at square one because the skills he or she has learned so far are worthless to the new group, or the programmer can leverage the skills learned on the previous project because those skills are more globally useful.

As I've said, you can create leverage out of almost any task—you just need to look for it and then exploit it. For example, during one of the feature reviews for the user interface library, the technical lead handed me his list of proposed library extensions. The functionality looked good—it reflected what the other teams were asking for.

"This looks good," I told him. "But some of these interfaces seem to differ from the way Windows does the same thing. Have you cross-checked the functionality with the Windows reference manuals?"

The lead blew up. "Steve, this library *isn't Windows*. Who cares how Windows does it as long as we provide the functionality in an intelligent way? It seems like a waste to keep pulling out the Windows manuals."

He had a good point. I realized then that I had never explained to him why I felt it was important to model Windows.

"Just so I'm sure I understand," I said, "you're saying that it doesn't matter what our interfaces look like as long as they do their job well. They could mirror Windows interfaces or be totally different. The choice is arbitrary."

"Yeah," he nodded.

"Let me ask you a question. Since Word for MS-DOS uses our library, could a Windows programmer mistake Word's source code for a Windows application if he or she didn't examine it closely?"

"Yeah, but it's *not* Windows code."

"Bear with me," I said. "More than 20 projects use our library. Do you think the programmers working on those projects will stay on those teams forever?"

"No. They'll probably switch to Windows projects."

"I think so too. So tell me, when those programmers switch to Windows projects, how easily will they pick up Windows programming?"

"Pretty easily since our library is like a subset of Windows." You could see the realization sweep across his face even as he said that.

"You mean, we're teaching them Windows programming?"

"And what does it cost the company?"

He thought a moment.

"Practically nothing, I guess—just my having to occasionally look up some functions in the Windows reference manuals."

"Right. And here's something else to think about: How will this Windows experience help *you* in the future? Will you be on this project forever, or will you also eventually move to a Windows project?"

It might seem that you couldn't get leverage out of something as simple as what you name your functions, but you can.

People don't often create new leverage because it calls for looking into the future and making the grand leap of faith, believing that if you create the leverage now, it will actually be used in the future. Will the leverage be used? Maybe not. But the business environment changes so quickly that, to be healthy, a company should create opportunities that can be exploited at a moment's notice. One truth I've seen proven over and over again is this:

If you create leverage and make others aware of it, they will someday exploit that leverage.

When I started the Macintosh cross development project, both the Applications division and the Languages division viewed the work as an in-house-only development system. My goal was to create a development system as an extension of the commercial 80x86 product so that the in-house Macintosh development system could continually inherit all improvements made to the commercial product. That's an obvious case of creating and using leverage, but I pushed for more. I believed that other, non-Microsoft, programmers who were writing applications for Windows would cross-compile those applications for the Macintosh if they had a good—and familiar—cross development system at their disposal. Most people thought I was crazy, but so what? I knew that if we assumed that the cross development system would never be a product, we'd make decisions inappropriate for a product. I also knew that if we wrote the code assuming that it would someday be a product, we'd make decisions that reflected that attitude.

In design meetings I would often point out that, yes, a particular design was workable for an in-house solution but that we'd have to rip it out and start over if Microsoft ever chose to ship the code as a product.

"But we're never going to ship this as a product," I'd hear.

"Well, not if we make that assumption," I'd say. "Let's just take a moment to see if there's an equally good design that would work for both the in-house and product solutions."

In most cases, not only did we come up with dual-purpose solutions, but often the designs were better and took less time to implement. The extra up-front thinking forced us to come up with more designs to consider. In a few cases, the only dual-purpose solution we could find looked as if it would take more time to implement than the in-house solution. In such a case, we chose the in-house design that would require the least additional rewriting if Microsoft ever chose to turn the cross development system into a product.

Whenever upper management asked about the state of the project, I would tell them what they wanted to know and always tell them again of our policy of not doing anything that would prevent the company from shipping the code. Upper management's only concern—one I shared—was that we not spend time doing product work that might never be used.

Nobody ever believed that the code would ship as a product, but one day Microsoft announced its "Windows Everywhere" campaign. All of a sudden it had become strategic for Microsoft to provide Windows solutions for non-80x86 platforms. The Macintosh cross development system was declared a product, given higher priority, and assigned more programmers.

Extract the most value possible from every task you do, by either exploiting existing leverage or creating new leverage.

LEVERAGING ATTITUDES

I've been talking about adopting the attitude that you'll exploit leverage whenever and wherever you see the possibility. That idea pervades this chapter even more than I've suggested. Instilling beneficial attitudes in your team is the ultimate use of leverage. With one small change in attitude you can get a tremendous return for the effort, more return than on any other training investment I'm aware of.

Constant, incremental improvement is great, and that alone is often enough to keep you ahead of your competitors, but if you want your teams to pull ahead, you must help them to develop beneficial attitudes that drive *them* to carry on, without supervision. That lead who was irritated because I asked him to refer to the Windows reference manuals never referred to the manuals himself until I explained the thinking behind my request. Once he understood my motivation—trying to create leverage—I never again had to pester him to check the Windows manuals. He became self-motivated.

HIGHLIGHTS

- Novice programmers must understand how difficult writing bug-free code is. If they have that understanding, they won't so readily assume that their code is bug-free. More experienced programmers must understand that even though writing bug-free code is difficult, it doesn't mean they should give up trying to write such code; it means that they must spend more time testing their code up front, before the code ever reaches the testing group. And because it's so difficult to write bug-free code, and so costly when bugs make it into the master sources, all programmers must use every tool at their disposal to detect and prevent bugs, even if that means adjusting their coding styles to weed out error-prone language idioms.

- Watch for the "it's too much work" and "it's too hard" reflex reactions. When you hear somebody object that a task will take too much time or that it will be too hard, ask yourself if the individual first considered whether the task was important and whether it matched the project goals and priorities. If it seems to you that he or she was merely responding reflexively, try to refocus the person on the merits of doing the task so he or she can evaluate the idea freshly and fairly.
- A common tendency is for people to think negatively when they're faced with something they haven't tried before. In one form or another, they latch onto the idea that the task is somehow impossible. Try to shake up this habitual response and instead help instill in team members the belief that most tasks *can* be done if only people would take some time to think about them. It's amazing how often you can respond to a "can't" judgment with the question "I realize it can't be done, but if it *could* be done, how would you do it?" and hear people rattle off exactly how they would do the thing they just said was impossible. The word "could" takes them out of reaction mode and puts them into thinking mode, right where they should be.
- The attitude that the user is neither demanding nor discerning is a detrimental one. Whenever you hear team members expressing such views, remind them that users—who by definition actually use the product—are at least as concerned about speed and the other aspects of software quality as the programmers who write the code.
- Teach programmers to view the product as an end user would. Programmers must recognize that end users view everything that goes into the box as a single product. Users don't care how

the individual pieces got into the box, they don't care if the product was built by 27 different teams, they don't care what language the code was written in—they don't care about any of that stuff. These points of information may be important to the company, and to the development teams, but users see only that the product is one item produced by one company. Programmers (and leads) may not work on every piece of the product, but they should be concerned when any piece doesn't meet the quality standards set for the product. When enough people express concern about a substandard piece, that piece will get fixed.

- Leverage is the most powerful tool at your disposal for adding value to your team, your project, your company, and even the industry. Take advantage of the principle of leverage by using it whenever you can. Strive to create new leverage in every task you undertake, whether it's writing code that could be shared, training team members in a way that makes them more valuable to the company as a whole and not just valuable for your own team, or taking a seemingly arbitrary decision like what you name a function and turning it into a way to prepare programmers for a future project. Think "leverage" in everything you do.