INTRODUCTION

Inspiring leaders look at the world in a funny way. The company building could be burning to the ground, and instead of panicking about the lost jobs, the inspiring leader takes one look at the flames and breaks out the hot dogs and marshmallows. When everybody around them is pessimistic, such leaders inspire confidence even though there may be every reason to be pessimistic. They're an optimistic bunch, tending to interpret events in a positive light. With that perspective, inspiring leaders tend to view failures not as failures but merely as learning experiences that will help them surmount the next obstacles that come along. And because inspiring leaders tend not to experience a sense of failure, they're willing to try the outlandish ideas that can lead to major breakthroughs. If an outlandish idea flops, the inspiring leader doesn't see the episode as a failure but merely as more information. Such leadership has little to do with experience. It's a combination of strong desire, an unusual way of looking at the world and its opportunities, and such a clear vision and the ability to communicate that vision that others are inspired to work with the leader to make that vision come true.

Despite the belief that such leaders are born and not made, it is possible to learn to be an inspiring leader. It isn't easy, though. Usually the person must change many of his or her fundamental beliefs and attitudes in order to view the world in that peculiar way. You might say that it calls for a personality makeover—an idea that most people would think impossible and that many would find repugnant. I think that's why it's rare for people to become inspiring leaders partway through their lives. People don't usually change their personalities to that extent.

The Rest of Us

Fortunately, most software project leads aren't starting new companies or venturing off into uncharted territory. The typical lead is usually embarking on the development of version 4.21 of an application or working on some other project that has a fairly straightforward future everybody is in basic agreement about. The typical software lead doesn't need to be a radically inspiring leader capable of getting team members to do outlandish things. The typical software lead simply needs to be *effective,* which is quite learnable and doesn't require anything like a personality transformation. It just requires learning the habits and strategies that have been found to work in bringing quality products to market on schedule—and without working 80-hour weeks.

All effective leads understand that for a project to be successful, every single member of the team must be in on the strategies that will be used to ship a quality product on schedule. You don't have to be the lead in order to make good use of the techniques and strategies I describe. This book is for every team member, not just the lead. Unless every team member knows what it takes to get a quality product out the door without working 80-hour weeks, it won't happen.

Writing Solid Code

A lot of steps are involved in the development team's effort to bring a software product to market—everything from designing the code to working with the marketing team. In every one of the steps in the development process, people make mistakes. There's nothing new in that observation. I've called this book *Debugging the Development Process* to get programmers to think of the development process as they would a coding algorithm: it's something that can contain bugs that will cause wasted and misguided effort, and it's something that can be optimized to function better.

In *Writing Solid Code,* the companion book to this one, I focused on what I believe is the most serious "bug" in the development process: that there are far too many programming bugs. *Writing Solid Code* described the techniques and strategies programmers can use to detect existing bugs at the earliest possible moment and how programmers can prevent those bugs in the first place.

In *Debugging the Development Process,* I focus on the techniques and strategies that programmers can use to get quality products out the door with a minimum of wasted effort. In the first three chapters, I talk about a number of basic concepts and strategies that a team should act on if they want to release products without working twelve hours a day, seven days a week. The final five chapters build on the earlier chapters, focusing singly on overblown corporate processes, the ins and outs of scheduling, programmer training, attitudes, and long hours.

Writing Solid Code and *Debugging the Development Process* are companion books. You'll find that the ideas in the two books interact with one another to a certain extent. When ideas in the two books overlap, you'll find that *Writing Solid Code* tends to be more focused on the code itself. In one instance I excerpt part of a section from *Writing Solid Code* in this book because I think that the point it makes is even more critical to the smooth running of a project than it is to writing bug-free code.

Development at Microsoft—A Snapshot

Most of the examples in this book are drawn from my experience at Microsoft. A brief description of how responsibilities are divided among leads and a sketch of how a typical project proceeds at Microsoft might put those examples in context for you.

A Microsoft project typically has at least three different types of leads working directly on the development of the product:

- Project Lead. The project lead is ultimately responsible for the code. He or she is also responsible for developing and monitoring the schedule, keeping the project on track, training the programmers, conducting program reviews for upper management, and so on. The project lead is usually one of the most experienced programmers on the team and will often write code, but only as a secondary activity.
- Technical Lead. The technical lead is the programmer on the team who knows the product's code better than anyone else. The technical lead is responsible for the internal integrity of the product, seeing that all new features are designed with the existing code in mind. He or she is also usually responsible for ensuring that all technical documents for the project are kept up-to-date: file format documents, internal design documents, and so on. Like the project lead, the technical lead is usually one of the most experienced programmers on the project.

- Program Manager. The program manager is responsible for coordinating product development with marketing, documentation, testing, and product support. In short, the program manager's job is to see that the product—everything that goes into the box—gets done, and that it gets done at the level of quality expected by the company. The program manager usually works with the product support team to coordinate external beta releases of the product and works with end users to see how the product might be improved. Program managers are often programmers themselves, but they limit their programming to using the product's macro language (if one exists) to write "wizards" and other useful end user macros. More than anyone else, the program manager is responsible for the "vision" of what the product should be.

  The name "program manager" can be misleading because it implies that the program manager is superior in rank to the project lead, the test lead, the documentation lead, and the marketing lead. In fact, the program manager is at the same level as the other leads. A more appropriate name for the program manager would be "product lead" since the program manager is responsible for ensuring that all the parts of the product—not just the code—get done on schedule and at an acceptable level of quality.

On a typical project, the program manager (or managers if the project is large enough) works up front with the marketing, development, and product support teams to come up with a list of improvements for the product. After the list of features has been created, the program manager writes the product specification, which describes in detail how each feature will appear to the user—providing, for instance, a drawing of a new dialog box with a description of how it will work, or the name of a new macro function with a description of its arguments. As soon as the product spec has been drafted, it is passed out to all of the teams involved with the product for a thorough review. Once the final spec has been nailed down, the teams go to work.

The program manager meanwhile uses mock-ups of features to conduct usability studies to be sure that all of the new features are as intuitively easy to use as everybody originally thought they would be. If a feature turns out to be awkward to use, the program manager proposes changes to the spec. The program manager also works on sample documents for the product disks and on those end user macros I mentioned earlier. As features are completed, he or she reviews each to ensure that it meets all the quality standards for shipping the product—in particular, that the feature is snappy enough on low-end machines.

Development continues and eventually reaches a point known as "visual freeze," meaning that all features that will affect the display have been completed. Once the code reaches the visual freeze point, the user manuals are finalized with screen shots of the program. Consequently, from that point on, developers have to be careful not to affect the display in any way so that the screen shots in the manuals won't differ from what the user sees in the program. The programmers, of course, would prefer that the screen shots be taken only after all the code is finished, but the manuals need a long lead time and have to be sent to the printer well before the code will be finalized. In some cases, in order to reach visual freeze on all the features in time for the manuals to be ready at the release date, the programmers will only partially implement the features—for instance, displaying a nonfunctional dialog good for screen shots but not much else. The programmers come back to the features and fully implement them later.

Once all of the features have been completed—the "code complete" stage—the programmers put their effort into fixing all outstanding bugs in the bug-list and making any necessary performance improvements. When the code is finally ready to be shipped, the project lead or the technical lead creates the "golden master disks." The program manager sends the golden masters off to manufacturing for duplication, and the software gets stuffed into the boxes with the manuals, the registration cards, and other goodies. A little bit of shrink-wrap, and the product is ready for an end user.

I've left out a lot of details, but this brief overview should be enough to enable you to put the occasional example in this book that might otherwise be too Microsoft-specific into context.

I should also mention that e-mail is the lifeblood of Microsoft. All internal business is conducted over e-mail, and, at least in development circles, you have to have a really good reason to interrupt someone with a telephone call. Most interaction among developers goes on over e-mail and in the numerous hall meetings that spring up spontaneously. This corporate sensitivity to interruptions accounts for Microsoft's policy of giving everyone a private office with a door. If you're working and you don't want to be interrupted, you simply close your door.

It's Harder than It Sounds

My final concern is that this book might make it sound as if applying all of its advice will, overnight, transform a less-than-model project. Certainly you can apply many of its techniques and strategies immediately, and you will get quick results; but others—some of the training techniques, for instance—take time to produce results. If your team is currently having trouble, you can't expect to read this book and a week later have your project turned around. In my experience, turning around a troubled project takes two to six months, with most of the improvement coming about in those first two months. From that point on, the improvements come more slowly and are less dramatic.