5

SCHEDULING MADNESS

I explained in Chapter 2 why I believe it's critical that teams fix bugs as they're found. We didn't follow that practice back when I was working on the Microsoft Excel project. In fact, we were pressured to ignore bugs until all scheduled features had been completed. Why? Because if we had stopped to fix bugs we would have appeared to have slipped the schedule. It wouldn't have mattered that the ship date would actually have been pulled in; anything that appeared to cause intermediate slips was discouraged, and a growing bug-list didn't count as slipping—you'd slipped only if you hadn't "finished" a feature as scheduled. The schedule, not the project goals and priorities, not even common sense, was driving the development process.

At that time, Microsoft's Applications division used a type of schedule that seemed reasonable on paper but that in practice demoralized teams and created a situation in which the strongest motive was to hit deadlines at the expense of all else—including product quality. Of course, at the time nobody thought of it that way because the problems weren't apparent. It took Microsoft several years—a round of product cycles for its applications—to realize the problems inherent in the scheduling system it was using.

Once the problems with the scheduling system became apparent, the process was tossed out, and a more humane scheduling system was brought in. Still, that was a costly learning experience for Microsoft, and I'll describe that experience so that others don't follow the same mistaken path. I'll also describe the scheduling process that many groups at Microsoft have moved to and that I have found to work quite well.

On a Project Long, Long Ago. . .

My primary reason for joining Microsoft back in 1986 was to work on high-quality Macintosh applications. I was assigned to Microsoft Excel, then Microsoft's latest entry into the Macintosh market. By any measure, working on Excel should have been exciting for me. It met all of my criteria: it was a serious Macintosh application, it was a highly visible application, and users loved it. Even better, Microsoft wasn't about to go belly-up, so I knew that the product would have a long life. I could get the Macintosh experience I wanted and have an influence on one of the industry's most promising applications.

Working on Excel was exciting at first, but after several months the work had become dull and then finally just plain aggravating. The Excel project should have been a dream project. It didn't make sense that I should find it so aggravating, but other team members were aggravated too, and so were programmers I knew who were working on other Microsoft projects. The problem wasn't the people we worked with, nor was it the work setting—Microsoft's environment was the best I'd experienced, hands down, in 10 years of computer industry work, and I know the aggravated team members and programmers on other Microsoft projects felt that way too. No, the aggravation was a side

effect of the type of project schedules Microsoft had begun using right around the time I joined up.

In the projects I'd worked on before I joined Microsoft, the team members had been excited by the work, and the dominant feeling had been enthusiasm over how much better the product was getting with each passing day. The Excel project never felt that way. Although we regularly improved the product, we were bombarded perpetually with the message that we were slipping. I was slipping, he was slipping, *everybody* was slipping, *the project was slipping!* The focus wasn't on the quality or even the quantity of our work: it was on the schedule.

In Chapter 1, I mentioned weekly status reports that had the effect of regularly slapping the programmers in the face. Those reports were just one aspect of the demoralizing scheduling process Microsoft was using back then. Besides writing those weekly status reports, the team members had to meet each week with the testing and documentation teams for a general discussion of how we'd slipped that week. We'd learn that the writers were stopped cold because the programmers had slipped and that the testers were sitting on their hands because the programmers had slipped. All we talked about was slipping.

I think even worse than the status reports and those awful status meetings was the project task list. Each week the Excel lead would use the latest round of status reports to update the master task list. Then he'd distribute the updated master list to each team member. Nothing wrong with that. But the first item you'd notice on the cover page would be the chart showing exactly how much each team member had slipped that week and how much the project as a whole had slipped. The chart didn't explain that you'd slipped because you'd had to tackle several unlisted but nevertheless required tasks that hadn't been anticipated back when the schedule had been created. Upper management would get these reports, see that you'd slipped yet again, and demand to know what was going on. Slap! Slap! Slap! It was not pretty in those days.

After the sting had lessened a bit, you'd turn to subsequent pages of the master task list and see what seemed like thousands of unfinished items. Worse, the list would be almost identical to the one you'd seen the week before. Here we were, working our hardest, and almost nothing seemed to be getting done. It was like that joke, "How do you eat an

elephant? . . . A bite at a time." The task list was our elephant, and it seemed as if we'd never finish eating it.

The focus was so much on the schedule's deadline that no matter how solid our work was we couldn't feel any sense of accomplishment. Quite the contrary: we were overwhelmed by the feeling we were so far behind that even with our best efforts we couldn't make any headway. It wasn't the nature of the work that was the problem; it was the apparent hopelessness of the position we were in.

Until that Excel project, I'd never seen how destructive a schedule can be to morale. What should have been my dream job felt like a nightmare. We were constantly slipping our schedule, but we weren't goofing off. The reality was that the project's schedule was hopelessly unrealistic. The schedule made these assumptions, for instance:

- That all tasks—for a two-year project—were known and listed on the schedule
- That each week each programmer would complete 40 hours' worth of the tasks listed on the schedule
- That all task estimates were completely accurate
- That all features would be implemented perfectly, without bugs

The world's most accomplished programming team couldn't have met a schedule based on those assumptions—unless, that is, they had regularly worked 80-hour weeks from the outset to compensate for all the unforeseen tasks, inaccurate estimates, and bugs, to say nothing of the meetings, reports, interviews, and e-mail that steal hours each week. The schedule also failed to account for the 10 legal holidays each year and for each programmer's two-week vacation each year. For a two-year project, that was an *almost-two-team-months* scheduling error. The schedule was doomed to slip.

Never allow the schedule to drive the project or to demoralize the team.

Just Following Standard Procedure

I want to emphasize that the Excel lead didn't intend to create a demoralizing situation. He was following the accepted scheduling process, and he later even adjusted the 40-hours-per-week assumption to account for meetings and other regular but unscheduled tasks—something that some leads on other projects would never do. Nor do I believe the schedule was intentionally designed to extract 80-hour work weeks from the programmers, although that was the result and is perhaps the source of Microsoft's reputation for working people hard. I believe the schedule was a sincere attempt to accurately predict and track progress. After all, what makes more sense than using the sum of the estimates for all known tasks to derive a scheduled "done date"? Of course, nobody believed the task list was complete or that all the estimates were accurate, but that didn't stop people—particularly upper management—from treating the derived "done date" as though it were realistic. In time, most Microsoft groups scrapped these task-list–driven schedules for a type of schedule that was more successful and that I'll describe later in the chapter.

PRIMING THE PUMP

You've probably heard at least one lead say, "If you want the team to work hard, you have to give them an aggressive schedule." I think all leads believe that to some degree—I certainly do. The question is, how aggressive is "aggressive"? If *aggressive* means making the schedule challenging enough that it drives the project forward at a reasonable clip, that's fine; but if *aggressive* means *unattainable,* such a schedule can only demoralize the team as slip-hysteria sets in.

A schedule should be aggressive enough to instill a sense of urgency in the programmers, to help them stay more focused on the important work. Think about your own situation. If you were taking off for a three-week vacation tomorrow, would you work at the same pace today that you normally would? My guess is that you'd work much smarter today than you usually do. You'd probably focus squarely on

getting all high-priority items out of the way—no long chats in the halls, no time spent on unimportant e-mail or news, no unnecessary meetings. That's the sense of urgency in action—better focus.

At Microsoft, the same sense of urgency develops whenever a final ship date nears. The lead typically sends out an e-mail message similar to this one:

```
We're nearing our ship date, so we need to be particu-
larly careful about how we use our time. Everybody's time
is valuable now--we're all working toward this one final
goal. Think twice before calling a meeting. Think twice
before bothering somebody with a question you could
easily look up yourself. If you come across an unexpected
task, don't assume that somebody else is going to take
care of it; they're just as busy as you are. Don't keep
a private to-do list of tasks that you'll get around to
"eventually." There is no "eventually." Tell me about
every pending task so that we can decide whether the task
is critical for this release. If you find yourself with
nothing to do, don't kick back because you think you're
done. Unless the team is done, you're not done. I could
go on, but you're all smart. You've all got brains. You
know if you're wasting time.
```

Whenever I see a notice like this one (they get passed to other groups periodically), my question is, shouldn't the team be working that way *all* the time?

"Geez, Steve, that sounds pretty awful. I thought you said in the last chapter that you weren't a 'nose to the grindstone' kind of lead." I'm not. If you look at the essence of that message, you'll see that it says, "Don't do business-as-usual. Work smarter-than-usual. Question every task to prevent wasting time, be careful about wasting other people's time, and take an active role in moving the product forward." That's what I've been saying all along. The language in the e-mail is harsh because the lead wants to convey in one message what I've had the luxury of spending a few chapters on.

If you felt pressed for time, would you conclude a meeting with "George, find out about such and such, and we'll meet again to make a final decision"? I doubt it. When people are pressed for time, they don't put tasks off—they either kill those tasks or handle them immediately.

Do you think a team would crackle with energy if they didn't have a sense of urgency? Imagine a team with so much time to spare that they could arrive each morning, put their feet up, and mull over every aspect of their project. Such contemplation can be rewarding, and the findings can certainly be valuable, but would the team be filled with energy and enthusiasm? Would the project be exciting? Somehow I doubt it, just as I doubt that a slow exchange of ideas can be as exciting as a rapid-fire brainstorming session. I believe that for a team to get on a creative roll, you have to pump energy into the process. The sense of urgency—time pressure—is one source of that energy.

Make sure your schedules are attainable but aggressive enough to keep team members focused on steady progress.

HOW MUCH IS TOO MUCH?

Can you pump too much urgency into a situation? Sure. If the schedule starts to look unattainable, you risk having team members start to make stupid decisions. I've worked with programmers who felt so swamped they stopped testing their code. If the code compiled and didn't blow up the first time they ran it, they moved on. Those programmers knew they weren't doing quality work, but they felt they had no choice, given the pressure of the schedule. They crossed their fingers and prayed that the testing team would catch any bugs that slipped through.

As a lead, you must keep your eye on the decisions people make under schedule pressure and remind people, when you have to, that hitting the deadline is rarely so critical that they should jeopardize the product with ill-conceived designs, slapped-together implementations, or untested code. Missing a deadline will hurt the project once, but bad designs and implementations will haunt the product forever—unless someone further down the line decides to use valuable time to rewrite all the sloppy code.

Never allow team members to jeopardize the product in the attempt to hit what might be, after all, an arbitrary deadline.

THE WIN-ABLE SCHEDULE

A win-able schedule is one that benefits both the company and the developer. As I've pointed out, the schedule must be aggressive enough to get the product out the door but attainable enough to allow the developers to feel they have time to do what you and they believe is best for the product. Another essential aspect of a win-able schedule is that it emphasize the progress made by the team, creating situations in which the team can have "wins."

Do you remember that elephantine Excel task list I talked about earlier in this chapter, the one that stayed the same size from week to week? For almost two years I would routinely arrive at work each day and knock a few tasks off that huge list. How much urgency do you think I felt as I chipped away at features for a deadline two years away? I can tell you: not much. In fact, the project didn't begin to feel urgent until practically the last couple of months, when the deadline was in plain sight.

Maybe you've heard the saying that a goal without a deadline is just a wish. It's the deadline that pumps energy into the development effort and gets people to scrap the dreary procedures of business-as-usual in favor of more effective strategies. We had a deadline for the Excel project, but that deadline was so far out that it had no power to ignite the team. We might as well have said, "Someday we'll ship Excel."

Without exception, every exciting project I've worked on has had deadlines much closer than Excel's two-year release date. It's not that the projects weren't large and didn't undergo development over a long period of time—they were and they did—but they were broken up into smaller subprojects, each with its own deadline, and the deadlines were spaced roughly two months apart. The result was that each subproject had an attainable near-term deadline that promoted the sense of urgency and each contributed to our feeling of progress as we completed it. We didn't ship every two years. We "shipped" every two months.

Thankfully, most Microsoft teams have moved to some form of this milestone-scheduling since the days when I worked on that Excel project. But using milestone-scheduling isn't enough. If you were to take

Arbitrary Deadlines

In my experience, most deadlines are arbitrary, either derived from the list of known tasks or simply handed down from above: "Thou shalt ship on June 11, or else." If you agree to a deadline, you should try to hit it, but the fact that you or upper management has set a date doesn't mean that the date is a priority that overrides quality. The date is too arbitrary. Think about your own project. If you missed the ship date by a month, what would the long-term impact on the company be? Would anyone even care six months later? But suppose, instead, that you hit your deadline and shipped your code with bugs and ill-conceived features. Which would affect your product more, a slightly late release date or an onslaught of bad product reviews?

Unless your code has to be functional by a date that simply can't be changed—say, the arrival of Halley's Comet after a long 76 years—your release date is probably not so critical that you must hit it at all costs. If your not having a piece of equipment ready for a scheduled space shuttle launch would cost your company millions of dollars, it would probably be better to cut functionality and focus on getting all the bugs out of the remaining code than to send all the code aloft and have the equipment crash the first time the astronauts try to use it.

Of course, this discussion makes it sound as if it takes more time to do things right. In my experience, it takes *less* time to do something the right way. You do spend more time up front as you set goals and priorities, think through designs and implementations, create test suites, and set quality bars, but you save a lot of time later. Think about it. Which would be more valuable, writing test suites at the start of the project or at the very end? It's that simple. When other teams are working 80-hour weeks, scrambling to whittle down their huge bug-lists, your team can have almost no bugs and spend the last few weeks of the project cycle adding ever more thorough checks to the test suites and debug code just to find that one, last, unknown bug.

Excel's two-year task list and merely chop it into two-month chunks, you wouldn't change anything—you'd still have a two-year schedule, but now with artificial "ship" dates every two months. It's not the two-month period alone that creates the wins and fosters enthusiasm. It's the thrill of finishing an interesting subproject.

"Finishing all top-priority items" may be important, but the top-priority items don't make up a subproject. They're just a random list of things that happen to be important. There's no motivating theme behind such a list.

"Implementing the charting subsystem" is a subproject. All of the tasks that would be involved would relate to that common theme. You might use a task list to remind people of the known charting issues they'd have to handle, but ultimately the theme of the subproject would drive development. The goal wouldn't be for the team to finish 352 unrelated tasks. The goal would be to do everything necessary to fully complete—to "ship"—the charting subsystem, regardless of whether the tasks it would take were on a list somewhere. The subproject would be in "ship mode" from the outset.

Think of it this way: if you were throwing a dinner party and you went to the store for groceries, would you search only for the party items you'd thought to write down on your shopping list, or would you view that list as the "known items" you need for the party and walk the store aisles thinking "What else do I need? What have I forgotten? There must be something I haven't thought of. . ." Wouldn't you also have a sense of urgency? That's the difference between trying to fulfill a goal—"Get everything I need for my party"—and merely checking off items on a list of unrelated tasks.

Remember that typical e-mail message from the lead as a ship date nears? The emphasis is on wrapping everything up, especially all loose ends. When people focus on a task list, the question they ask themselves is "What's next on my list?" When they focus on a subproject, the question is usually quite different: "What else needs to get done?" The focus is on searching out and handling every task related to the subproject.

Any milestone without a theme ends up having to be driven by a task list because, without a theme, you need such a list in order to know what you're supposed to do.

Break long-term projects into shorter, well-defined subprojects.

The Wow! Factor

One way to view the difference between improving the product with unrelated high-priority tasks and completing specific subprojects is to look at your house as if you were going to remodel it. Which updates would have more impact: newly painted trim in one room, a new light fixture in another, a new end table in the living room, and so on, or the living room completely transformed—new paint, new carpet, new furniture, and new art on the walls? When you release a subproject, you get the "living room effect." Internal users, beta testers, upper management—in fact, everybody who fires up the code—thinks Wow! when they see what's been done. With the incremental task list approach, people notice a change here, a change there, but nothing major. That's not bad, but why settle for low impact when you could get more?

Of course, the only difficulty lies in choosing subprojects that present enough different aspects of the work that programmers won't be stumbling over each other, all needing to work on the same source file. I've never found this to be a difficult problem to solve, though.

Eliciting a Wow! can be a critical catalyst that gets a team going on a creative roll.

ENHANCING THE WOW! EFFECT

A milestone goal such as "Finish all top-priority items" is just a mishmash of probably unrelated items. If the ship date for such a subproject were threatened, the lead would be able to mask the problem by quietly reprioritizing the tasks. That might allow the lead to look better, but it would be a misleading and questionable way to go about things.

A more coherent milestone theme would be "Complete all features that affect the visual display so that we can finalize screen shots for the

user manual." This milestone is better because it has a theme that's easy to grasp and because it's easy to judge which tasks are appropriate. You can point to any known task and instantly determine whether it affects the visual display. Even better, if an unforeseen task crops up midway through the milestone period, anybody on the team, no matter how green, can easily determine whether it needs to be tackled or whether it can be postponed until work on an appropriate subproject begins.

Often you can attack a major project in any of several ways. Use that latitude to create subprojects that will result in exciting conclusions, to get that Wow! effect. When we were working on the Macintosh C/C++ cross development system, I broke the job up into these subprojects:

- Isolate all Intel 80x86–specific code in the compiler to enable support for other processor types.
- Implement a bare-bones MC680x0 code generator in the compiler.
- Implement MC680x0 assembly listing support in all tools.
- Implement MC680x0 object file support in all tools.
- Link a single-segment application, and run it.
- Link a multi-segment application, and run it.
- Add code optimizations to the code generator.
- . . .

I chose these specific milestones and others because each, according to my estimation, would take between one and two months to complete, each was easy to understand, and, with the exception of isolating the Intel 80x86 code, each had an exciting conclusion. Don't think we didn't hoot and holler the first time we had code generation working or when we first saw the generated code dumped on the screen in proper assembly language format. We cheered when we linked and ran our first test application and especially when, after adding some basic optimizations, we realized that the compiler was already generating code comparable to code from the two leading Macintosh compilers on the market.

It was exciting!

Don't Forget the Details

Of course, none of the milestone descriptions was as simple as the one-liners for the cross development system I listed on the opposite page. "Link a single-segment application and run it" doesn't provide enough detail. The actual milestone statement was more specific:

> We should be able to copy an arbitrary single-file Macintosh program into a build directory, rename the file *test.c*, and type *make*. The program should compile without problems, the object files should link without problems, and the executable should transfer automatically to the Macintosh over a cable that connects the Macintosh with the development machine. Then, on the Macintosh, we should be able to double-click on the new file and run the code without problems.

From this more detailed description, you can see that we had to handle all the loose ends in the compiler project, including support for the Macintosh-specific C-language extensions such as *\p* Pascal strings, Pascal-compatible calling conventions for use in call-back routines, ROM traps, 4-character *long*s for resource types, and so on. We had to modify the 80x86 linker to support MC680x0 code and to create Macintosh-formatted executables. We had to write the runtime startup code, some C library code, and the code to support the transfer mechanism between the PC development machine and the target Macintosh. There was a lot of stuff to do.

You won't always achieve such aggressive milestone goals. We didn't for this particular milestone, but we came close. Supporting some of the Macintosh C-language extensions required changes to the front end of the compiler, and a different team was in charge of that code. At the time, they were frantically putting the final touches on their own release and didn't have time enough to do that work, much less ours, nor did they want us mucking around in their code. We got those changes after their release. You take what you can get.

I could have organized the project so that all high-priority work was done first, followed by secondary work, and so on, but the subprojects would have been quite different, and they almost certainly wouldn't have been accompanied by the Wow! effect.

To keep the subprojects challenging (and realistic), we didn't use a simple "hello world" test application. That would hardly have exercised the compiler, the linker, and the other tools. We used a small but fully functional public domain Macintosh program. Because we used a real application, we not only saw that the compiler was truly viable but were also forced to handle numerous final-detail issues that a simpler program wouldn't have raised. In task list scheduling, handling such details would have been relegated to the end of the whole project, but the thought of seeing the real-life Macintosh application run spurred the team on, and what could have been boring final-detail work later turned into work we *wanted* to do—and quickly.

Granted, it can be quite disturbing to upper management if he or she doesn't understand that you're using this thematic method of scheduling. It will seem as though you're throwing darts to choose which tasks to do rather than picking the highest-priority tasks.

To foster creative rolls, make sure that each subproject results in an exciting conclusion.

THE BEST-CASE DATE

People often forget that the purpose of the schedule is to estimate a completion date given the tasks known at the time. Such a date is not a commitment in the sense that you must hit it at all costs; rather, the date is a good-faith estimate of when the known tasks could be done, with the understanding that there are usually plenty of unknown tasks. In short, the schedule predicts a *best-case* ship date, not *the* ship date. That may not be what upper management wants to hear, but it's reality. Using milestone scheduling instead of task-list–driven scheduling helps to bring the best-case date in line with a realistic ship date, but milestone scheduling isn't perfect either.

As a lead, you must protect your product by emphasizing to your team that product quality is more important than hitting an arbitrary deadline. Remember the lesson from this chapter:

> *The surest way to mismanage a project and jeopardize the product is to put so much emphasis on the schedule that it demoralizes the team and drives them to make stupid decisions despite their better judgments.*

I certainly believe that you should try to hit every deadline you commit to, but keep that "best-case date" idea in mind. That way, if you find yourself about to make a bad decision just to hit that best-case date, maybe you'll stop yourself before any serious damage can be done.

HIGHLIGHTS

- The schedule can have a devastating effect on a project if it creates slip-hysteria and causes team members to make bad trade-offs in order to hit arbitrary deadlines. If you create a schedule that has unattainable goals—in hopes of extracting as much overtime as you can get out of each developer—you're creating a situation that will demoralize the team. Once the team members feel they're in a hopeless position, you're going to get anything but optimum work from them, and once the project is finished—maybe sooner—they're going to look elsewhere for work.
- By using project milestones instead of task lists to schedule, you can shift the focus to completing subprojects, which creates "wins" for the team and emphasizes progress. If you space the milestones at roughly two-month intervals, you can create a sense of urgency that will help people stay focused, particularly if the milestones have strong, exciting themes. Try to create milestone subproject goals that result in the team's thinking "Wow! Look at what we've accomplished!" As they reach successive milestones, the team will have a growing sense that their work is important and that they're

doing something valuable for the product's users. That sense of contribution and the sense of value created can have a remarkable influence, making a team pull together to put out a great product—and have a blast doing it.