CHAPTER 1

DOOMED

[JULY 2003]

Michael Toy places his palms on his cheeks, digs his chin into his wrists, squints into his PowerBook, and begins the litany.

"John is doomed. He has five hundred hours of work scheduled between now and the next release. . . . Katie's doomed. She has way more hours than there are in the universe. Brian is majorly doomed. Plus he's only half time. Andy—Andy is the only one who doesn't look doomed. There are no hundreds on his list."

They don't *look* doomed, these programmers sitting around a nondescript conference room table in Belmont, California, on a summer day. They listen quietly to their manager. Toy is a tall man with an impressive gut and a ponytail, but he seems to shrink into a space of dejection as he details how far behind schedule the programmers have fallen. It's July 17, 2003, and he's beginning to feel doomed himself about getting everything done in the less than two months before they are supposed to finish another working version of their project.

"Everybody who has a list with more time than there is in the universe needs to sit down with me and go over it."

These lists are the bug lists—rosters of unsolved or "open" problems or flaws. Together they provide a full accounting of everything these software developers know must be fixed in their product. The bug lists live inside a program called Bugzilla. Toy's programmers are also using Bugzilla to track all the programming tasks that must be finished in order to complete a release of the project; each one is responsible for entering his or her list into Bugzilla along with an estimate of how long each task will take to complete.

"Now let's talk about why we're behind. Does anyone have a story to tell?"

There's silence for a minute. John Anderson, a lanky programming veteran whose title is systems architect and who is, in a de facto sort of way, the project's lead coder, finally speaks up, in a soft voice. "There's a bunch of reasons. In order to build something, you have to have a blueprint. And we don't always have one. Then you hit unexpected problems. It's hard to know how long something's going to take until you know for sure you can build it."

"But you can't just throw up your hands and say, I quit." Toy usually prefers to check things off his agenda fast, running his developers' meetings with a brisk attitude of "let's get out of here as fast as we can" that's popular among programmers. But today he's persistent. He won't let the scheduling problems drop. "We need to make guesses and then figure out what went wrong with our guesses."

Jed Burgess, one of the project's younger programmers, speaks up. "There's a compounding of uncertainty: Your estimates are based on someone else's estimates."

Toy begins reviewing Anderson's bugs. "The famous flicker-free window resizing problem. What's up with that?"

Officially, this was bug number 44 in Bugzilla, originally entered on January 19, 2003, and labeled "Flicker Free window display when resizing windows." I had first heard of the flicker-free window resizing problem at a meeting in February 2003 when the Open Source Applications Foundation

(OSAF), whose programmers Toy was managing, had completed the very earliest version of its project, Chandler—an internal release not for public unveiling that came even before the 0.1 edition. Ultimately, Chandler was supposed to grow up into a powerful "personal information manager" (PIM) for organizing and sharing calendars, email, to-do lists, and all the other stray information in our lives. Right now, the program remained barely embryonic.

At that February meeting, Anderson had briefly mentioned the flicker bug—when you changed the size of a window on the Chandler screen, everything flashed for a second—as a minor matter, something he wanted to investigate and resolve because, though it did not stop the program from working, it offended him aesthetically. Now, nearly six months later, he still hasn't fixed it.

Today Anderson explains that the problem is thornier than he had realized. It isn't simply a matter of fixing code that he or his colleagues have written; its roots lie in a body of software called wxWidgets that the Chandler team has adopted as one of the building blocks of their project. Anderson must either wait for the programmers who run wxWidgets to fix their own code or find a way to work around their flaw.

"So you originally estimated that this would take four hours of work," Toy says. "That seems to have been off by an order of magnitude."

"It's like a treasure hunt," Anderson, unflappable, responds. "You have to find the first thing. You have to get the first clue before you're on your way, and you don't know how long it will take."

"So you originally estimated four hours on this bug. You now have eight hours."

"Sometimes," Anderson offers philosophically, "you just wake up in the morning, an idea pops into your head, and it's done—like that."

Mitchell Kapor has been sitting quietly during the exchange. Kapor is the founder and funder of the Open Source Applications Foundation, and Chandler is his baby. Now he looks up from his black Thinkpad. "Would it

be useful to identify issues that have this treasure-hunt aspect? Is there a certain class of task that has this uncertainty?"

"Within the first hour of working on the bug," Burgess volunteers, "you know which it's going to be."

So it is agreed: Bugs that have a black hole–like quality—bugs that you couldn't even begin to say for sure how long they would take to fix—would be tagged in Bugzilla with a special warning label.

Shortly after the meeting, Toy sits down at his desk, calls up the Bugzilla screen, and enters a new keyword for bug number 44, "Flicker Free window display when resizing windows": *scary*.

Toy's fatalistic language wasn't just a quirk of personality: Gallows humor is a part of programming culture, and he picked up his particular vocabulary during his time at Netscape. Though today Netscape is remembered as the Web browser company whose software and stock touched off the Internet boom, its developers had always viewed themselves as a legion of the doomed, cursed with impossible deadlines and destined to fail.

There was, in truth, nothing especially doomed about OSAF's programmers: Several of them had just returned from a conference where they presented their work to an enthusiastic crowd of their peers—who told them that their vision could be "crisper" but who mostly looked at the blueprint for Chandler and said, "I want it now!" Though the software industry had been slumping for three straight years, they were working for a nonprofit organization funded by $5 million from Kapor. Their project was ambitious, but their ranks included veteran programmers with estimable achievements under their belts. Andy Hertzfeld had written central chunks of the original Macintosh operating system. John Anderson had written one of the first

word processors for the Macintosh and later managed the software team at Steve Jobs's Next. Lou Montulli, another Chandler programmer who was not at the meeting, had written key parts of the Netscape browser. They'd all looked doom in the eye before.

Similarly, there was nothing especially scary about bug number 44. It was a routine sort of problem that programmers had accepted responsibility for ever since computer software had migrated from a text-only, one-line-at-a-time universe to today's graphic windows–and-mouse landscape. What scared Toy was not so much the nature of Bug 44 but the impossibility of knowing how long it would take to fix. Take one such unknown, place it next to all the other similar unknowns in Chandler, multiply them by one another, and you have the development manager's nightmare: a "black hole" in the schedule, a time chasm of indeterminate and perhaps unknowable dimensions.

Two months before, the entire Chandler team of a dozen programmers had met for a week of back-to-back meetings to try to solve a set of problems that they had dubbed "snakes"—another word Toy had salvaged from Netscape's ruins. A snake wasn't simply a difficult problem; it was an "important problem that we don't have consensus on how to attack." *Snake* superseded a looser usage at OSAF of the word *dragon* to describe the same phenomenon.

Black holes, snakes, dragons—the metaphors all daubed a layer of mythopoetic heroism over the most mundane of issues: how to schedule multiple programmers so that work actually got done. You could plug numbers into Bugzilla all day long; you could hold one meeting after another about improving the process. Software time remained a snake, and it seemed invincible.

This was hardly news to anyone in the room at OSAF. The peculiar resistance of software projects to routine scheduling is both notorious and widely accepted. In the software development world, lateness was so common that a new euphemism had to be invented for it: *slippage*.

Certainly, every field has its sagas of delay; the snail's pace of lawsuits is legendary, and any building contractor who actually finishes a job on time is met with stares of disbelief. But there's something stranger and more baffling about the way software time bends and twists back on itself like a Möbius strip. Progress seems to move in great spasms and then halt for no reason. You think you're almost done, and then you turn around and six months have passed with no measurable progress.

This is what it feels like: A wire is loose somewhere deep inside the workings. When it's connected, work moves quickly. When it's not, work halts. Everyone on the inside tries painstakingly to figure out which wire is loose, where the outage is, while observers on the outside try to offer helpful suggestions and then, losing their patience, give the whole thing a sharp kick.

Every software project in history has had its loose wires. Every effort to improve the making of software is an effort to keep them tight.

The earliest and best diagnosis of the problem of software time can be found in a 1975 book by Frederick Brooks, *The Mythical Man-Month*. Brooks was a veteran IBM programming manager who had seen firsthand the follies of the largest software project of its day, the creation of the operating system for the IBM System/360. That huge, expensive mainframe computer would become the mainstay of big business for the next two decades, but its gestation and birth were plagued with delays and cost overruns. It had become, as Brooks put it, a "tar pit," a sticky trap for corporate beasts, even ones as "great and powerful" as IBM. Brooks's group at IBM, confounded by how far behind they'd fallen, flung waves of new programmers at the job, like General Lee's ordering one brigade after another to

charge up Cemetery Ridge—only to find that the reinforcements, far from getting the project over its hump, actually made things worse.

And so he formulated Brooks's Law, which is both a principle and a paradox: "Adding manpower to a late software project makes it later."

The soundness of Brooks's Law has been borne out over and over in the three decades since its formulation. But it is so deeply counterintuitive that it still regularly flummoxes programmers and managers, who often would rather pretend that it does not apply to them than deal with its disturbing implications.

Brooks noted that software developers are typically optimists who assume that each bug can be fixed quickly and that the number of new bugs will diminish until the last one has been licked. This optimism, along with the programmers' desire to please the impatient patrons who have commissioned them to create something new, skews the schedule from the start. In practice, Brooks found, nearly all software projects require only one-sixth of their time for the writing of code and fully half their schedule for testing and fixing bugs. But it was a rare project manager who actually planned to allocate developers' time according to such a breakdown.

Next, Brooks argued, the "very unit of effort used in estimating and scheduling" was "a dangerous and deceptive myth." The "man-month" was a concept of scientific management that assumed productivity could be broken down into discrete, identical, fungible units. So if one hundred men (in those days the gender assumption simply came with the territory) could produce fifty widgets in one month, then a single widget required two man-months—and you ought to be able to produce the same number of widgets sooner by throwing more workers at the project. Want your fifty widgets in three days instead of thirty? Just throw one thousand workers at the job instead of one hundred.

Brooks observed that "men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them*." With software, where every project is different

and the tools are in constant flux, each time you add a new member to a team, the veterans must drop what they are doing to bring the latecomer up to speed, and everyone needs to pause to reapportion their tasks to give the newcomer something to do. Before you know it, you're even further behind schedule. In the worst cases, Brooks saw, this set up a disastrous loop of delay, a "regenerative scheduling disaster" in which each resetting of the schedule triggers the hiring of more bodies, forcing yet another new schedule into place. Brooks quailed at that prospect: "Therein lies madness."

In any case, Brooks found that much of the work in creating software also suffers from "sequential constraints" that limit how far you can go in splitting up tasks: One task must be completed before the next can be tackled, regardless of how many hands work on it. "The bearing of a child takes nine months," he wrote, "no matter how many women are assigned."

Finally, the extreme variations in productivity from programmer to programmer—a great programmer was typically able to produce ten times as much work in a given amount of time as a mediocre one, and the work was typically five times as good (measured in terms of speed and efficiency)—sealed Brooks's case against the man-month. If one man's work was so drastically different from another's, that yardstick was useless.

Brooks's Law implies that the ideal size for a programming team is one—a single developer who never has to stop to communicate with a colleague. This approach streamlines everything, and it also provides insurance that the project will retain what Brooks calls "conceptual integrity": the alignment of all its parts toward the same purpose and according to a harmonious plan. Indeed, the history of software is full of breakthroughs made by lone wolves. But too much software has simply grown too huge for one

person to produce—from the elaborateness of the graphic interface that makes programs easier to use to the complexity of the data structures required to reliably and usefully store the huge volumes of information we trust our computers with. Beginning in Brooks's day with operating systems—the basic code that manages a computer system's resources and makes its circuitry responsive to users and their programs—but more recently spreading to every arena of software, programming has become a group effort, a team sport.

The dozen or so programmers at OSAF represented a relatively small team by industry standards, but in their nine months of work since Chandler's official announcement in October 2002, they had already encountered most of the problems that Brooks had outlined decades before. Despite all the computer-based tools at their disposal, from mailing lists and blogs to bug-trackers and source-code version-control systems, staying in sync with one another was fiendishly difficult. Making a schedule was nearly as hard as meeting one. And each new delay introduced the temptation to hire more bodies, yet the new hires never seemed to speed the schedule along.

In one vital respect, however, the work at OSAF represented an entirely new wrinkle on Brooks. The open source software development methodology from which Kapor's foundation took its name simply did not exist in the days of *The Mythical Man-Month*. And more than any other development since then, open source has tantalized the programming world with the prospect of repealing Brooks's Law.