Microsoft

RAPID DEVELOPMENT

Taming Wild Software Schedules

Steve McConnell

Author of *Code Complete*

Best Practices

Critical acclaim for Steve McConnell's *Code Complete*

"Every half an age or so, you come across a book that short-circuits the school of experience and saves you years of purgatory.... I cannot adequately express how good this book really is. *Code Complete* is a pretty lame title for a work of brilliance."

PC Techniques

"Microsoft Press has published what I consider to be the definitive book on software construction: *Code Complete* by Steve McConnell. This is a book that belongs on every software developer's shelf."

Software Development

"Every programmer should read this outstanding book."

Computer

"If you are or aspire to be a professional programmer, this may be the wisest $35 investment you'll ever make. Don't stop to read the rest of this review: just run out and buy it. McConnell's stated purpose is to narrow the gap between the knowledge of industry gurus and common commercial practice.... The amazing thing is that he succeeds."

IEEE Micro

"*Code Complete* should be required reading for anyone...in software development."

C Users Journal

"I'm encouraged to stick my neck out a bit further than usual and recommend, without reservation, Steve McConnell's *Code Complete*.... My copy has replaced my API reference manuals as the book that's closest to my keyboard while I work."

Windows Tech Journal

"This well-written but massive tome is arguably the best single volume ever written on the practical aspects of software implementation."

Embedded Systems Programming

"This is the best book on software engineering that I have yet read."

.EXE Magazine

"This book deserves to become a classic, and should be compulsory reading for all developers, and those responsible for managing them."

Program Now

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 1996 by Steve McConnell

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
McConnell, Steve.
    Rapid Development : Taming Wild Software Schedules / Steve
  McConnell.
      p.   cm.
    Includes index.
    ISBN 978-1-556-15900-8
    1. Computer software--Development.   I. Title.
  QA76.76.D47M393 1996
  005.1'068--dc20   96-21517
                                                       CIP

Printed and bound in the United States of America.

14 15 16 17 18 19 MLML   06 05 04 03 02 01

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

AT&T is a registered trademark of American Telephone and Telegraph Company. Apple and Macintosh are registered trademarks of Apple Computer, Inc. Boeing is a registered trademark of The Boeing Company. Borland and Delphi are registered trademarks of Borland International, Inc. FileMaker is a registered trademark of Claris Corporation. Dupont is a registered trademark of E.I. Du Pont de Nemours and Company. Gupta is a registered trademark of Gupta Corporation (a California Corporation). Hewlett-Packard is a registered trademark of Hewlett-Packard Company. Intel is a registered trademark of Intel Corporation. IBM is a registered trademark of International Business Machines Corporation. ITT is a registered trademark of International Telephone and Telegraph Corporation. FoxPro, Microsoft, MS-DOS, PowerPoint, Visual Basic, Windows, and Windows NT are registered trademarks and Visual FoxPro is a trademark of Microsoft Corporation. Powersoft is a registered trademark and PowerBuilder is a trademark of PowerSoft Corporation. Raytheon is a registered trademark of Raytheon Company. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: David J. Clark
Project Editor: Jack Litewka

Contents

Case Studies

Reference Tables

Preface

Software developers are caught on the horns of a dilemma. One horn of the dilemma is that developers are working too hard to have time to learn about effective practices that can solve most development-time problems; the other horn is that they won't get the time until they do learn more about rapid development.

Other problems in our industry can wait. It's hard to justify taking time to learn more about quality when you're under intense schedule pressure to "just ship it." It's hard to learn more about usability when you've worked 20 days in a row and haven't had time to see a movie, go shopping, work out, read the paper, mow your lawn, or play with your kids. Until we as an industry learn to control our schedules and free up time for developers and managers to learn more about their professions, we will never have enough time to put the rest of our house in order.

The development-time problem is pervasive. Several surveys have found that about two-thirds of all projects substantially overrun their estimates (Lederer and Prasad 1992, Gibbs 1994, Standish Group 1994). The average large project misses its planned delivery date by 25 to 50 percent, and the size of the average schedule slip increases with the size of the project (Jones 1994). Year after year, development-speed issues have appeared at the tops of lists of the most critical issues facing the software-development community (Symons 1991).

Although the slow-development problem is pervasive, some organizations are developing rapidly. Researchers have found 10-to-1 differences in productivity between companies within the same industries, and some researchers have found even greater variations (Jones 1994).

The purpose of this book is to provide the groups that are currently on the "1" side of that 10-to-1 ratio with the information they need to move toward the "10" side of the ratio. This book will help you bring your projects under control. It will help you deliver more functionality to your users in less time. You don't have to read the whole book to learn something useful; no matter what state your project is in, you will find practices that will enable you to improve its condition.

Who Should Read This Book?

Slow development affects everyone involved with software development, including developers, managers, clients, and end-users—even their families and friends. Each of these groups has a stake in solving the slow-development problem, and there is something in this book for each of them.

This book is intended to help developers and managers know what's possible, to help managers and clients know what's realistic, and to serve as an avenue of communication between developers, managers, and clients so that they can tailor the best possible approach to meet their schedule, cost, quality, and other goals.

Technical Leads

This book is written primarily with technical leads or team leads in mind. If that's your role, you usually bear primary responsibility for increasing the speed of software development, and this book explains how to do that. It also describes the development-speed limits so that you'll have a firm foundation for distinguishing between realistic improvement programs and wishful-thinking fantasies.

Some of the practices this book describes are wholly technical. As a technical lead, you should have no problem implementing those. Other practices are more management oriented, and you might wonder why they are included here. In writing the book, I have made the simplifying assumption that you are Technical Super Lead—faster than a speeding hacker; more powerful than a loco-manager; able to leap both technical problems and management problems in a single bound. That is somewhat unrealistic, I know, but it saves both of us from the distraction of my constantly saying, "If you're a manager, do this, and if you're a developer, do that." Moreover, assuming that technical leads are responsible for both technical and management practices is not as far-fetched as it might sound. Technical leads are often called upon to make recommendations to upper management about technically oriented management issues, and this book will help prepare you to do that.

Individual Programmers

Many software projects are run by individual programmers or self-managed teams, and that puts individual technical participants into de facto technical-lead roles. If you're in that role, this book will help you improve your development speed for the same reasons that it will help bona fide technical leads.

Managers

Managers sometimes think that achieving rapid software development is primarily a technical job. If you're a manager, however, you can usually do as much to improve development speed as your developers can. This book describes many management-level rapid-development practices. Of course, you can also read the technically oriented practices to understand what your developers can do at their level.

Key Benefits of This Book

I conceived of this book as a *Common Sense* for software developers. Like Thomas Paine's original *Common Sense*, which laid out in pragmatic terms why America should secede from Mother England, this book lays out in pragmatic terms why many of our most common views about rapid development are fundamentally broken. These are the times that try developers' souls, and, for that reason, this book advocates its own small revolution in software-development practices.

My view of software development is that software projects can be optimized for any of several goals—lowest defect rate, fastest execution speed, greatest user acceptance, best maintainability, lowest cost, or shortest development schedule. Part of an engineering approach to software is to balance trade-offs: Can you optimize for development time by cutting quality? By cutting usability? By requiring developers to work overtime? When crunch time comes, how much schedule reduction can you ultimately achieve? This book helps answer such key trade-off questions as well as other questions.

Improved development speed. You can use the strategy and best practices described in this book to achieve the maximum possible development speed in your specific circumstances. Over time, most people can realize dramatic improvements in development speed by applying the strategies and practices described in this book. Some best practices won't work on some kinds of projects, but for virtually any kind of project, you'll find other best practices that will. Depending on your circumstances, "maximum development speed" might not be as fast as you'd like, but you'll never be completely out of luck just because you can't use a rapid-development language, are maintaining legacy code, or work in a noisy, unproductive environment.

Rapid-development slant on traditional topics. Some of the practices described in this book aren't typically thought of as rapid-development practices. Practices such as risk management, software-development fundamentals, and lifecycle planning are more commonly thought of as "good software-development practices" than as rapid-development methodologies.

These practices, however, have profound development-speed implications that in many cases dwarf those of the so-called rapid-development methods. This book puts the development-speed benefits of these practices into context with other practices.

Practical focus. To some people, "practical" means "code," and to those people I have to admit that this book might not seem very practical. I've avoided code-focused practices for two reasons. First, I've already written 800 pages about effective coding practices in *Code Complete* (Microsoft Press, 1993). I don't have much more to say about them. Second, it turns out that many of the critical insights about rapid development are not code-focused; they're strategic and philosophical. Sometimes, there is nothing more practical than a good theory.

Quick-reading organization. I've done all I can to present this book's rapid-development information in the most practical way possible. The first 400 pages of the book (Parts I and II) describe a strategy and philosophy of rapid development. About 50 pages of case studies are integrated into that discussion so that you can see how the strategy and philosophy play out in practice. If you don't like case studies, they've been formatted so that you can easily skip them. The rest of the book consists of a set of rapid-development *best practices*. The practices are described in quick-reference format so that you can skim to find the practices that will work best on your projects. The book describes how to use each practice, how much schedule reduction to expect, and what risks to watch out for.

The book also makes extensive use of marginal icons and text to help you quickly find additional information related to the topic you're reading about, avoid classic mistakes, zero in on best practices, and find quantitative support for many of the claims made in this book.

A new way to think about the topic of rapid development. In no other area of software development has there been as much disinformation as in the area of rapid development. Nearly useless development practices have been relentlessly hyped as "rapid-development practices," which has caused many developers to become cynical about claims made for any development practices whatsoever. Other practices are genuinely useful, but they have been hyped so far beyond their real capabilities that they too have contributed to developers' cynicism.

Each tool vendor and each methodology vendor want to convince you that their new silver bullet will be the answer to your development needs. In no other software area do you have to work as hard to separate the wheat from the chaff. This book provides guidelines for analyzing rapid-development information and finding the few grains of truth.

This book provides ready-made mental models that will allow you to assess what the silver-bullet vendors tell you and will also allow you to incorporate new ideas of your own. When someone comes into your office and says, "I just heard about a great new tool from the GigaCorp Silver Bullet Company that will cut our development time by 80 percent!" you will know how to react. It doesn't matter that I haven't said anything specifically about the GigaCorp Silver Bullet Company or their new tool. By the time you finish this book, you'll know what questions to ask, how seriously to take GigaCorp's claims, and how to incorporate their new tool into your development environment, if you decide to do that.

Unlike other books on rapid development, I'm not asking you to put all of your eggs into a single, one-size-fits-all basket. I recognize that different projects have different needs, and that one magic method is usually not enough to solve even one project's schedule problems. I have tried to be skeptical without being cynical—to be critical of practices' effectiveness but to stop short of *assuming* that they don't work. I revisit those old, overhyped practices and salvage some that are still genuinely useful—even if they aren't as useful as they were originally promised to be.

Why is this book about rapid development so big? Developers in the IS, shrink-wrap, military, and software-engineering fields have all discovered valuable rapid-development practices, but the people from these different fields rarely talk to one another. This book collects the most valuable practices from each field, bringing together rapid-development information from a wide variety of sources for the first time.

Does anyone who needs to know about rapid development really have time to read 650 pages about it? Possibly not, but a book half as long would have had to be oversimplified to the point of uselessness. To compensate, I've organized the book so that it can be read quickly and selectively—you can read short snippets while you're traveling or waiting. Chapters 1 and 2 contain the material that you *must* read to understand how to develop products more quickly. After you read those chapters, you can read whatever interests you most.

Why This Book Was Written

Clients' and managers' first response to the problem of slow development is usually to increase the amount of schedule pressure and overtime they heap on developers. Excessive schedule pressure occurs in about 75 percent of all large projects and in close to 100 percent of all very large projects (Jones 1994). Nearly 60 percent of developers report that the level of stress they feel is increasing (Glass 1994c). The average developer in the U.S. works from 48 to 50 hours per week (Krantz 1995). Many work considerably more.

In this environment, it isn't surprising that general job satisfaction of software developers has dropped significantly in the last 15 years (Zawacki 1993), and at a time when the industry desperately needs to be recruiting additional programmers to ease the schedule pressure, developers are spreading the word to their younger sisters, brothers, and children that our field is no fun anymore.

Clearly our field can be fun. Many of us got into it originally because we couldn't believe that people would actually pay us to write software. But something not-so-funny happened on the way to the forum, and that something is intimately bound up with the topic of rapid development.

It's time to start shoring up the dike that separates software developers from the sea of scheduling madness. This book is my attempt to stick a few fingers into that dike, holding the madness at bay long enough to get the job started.

Acknowledgments

Heartfelt thanks go first to Jack Litewka, the project editor, for making the creation of this book a thoroughly constructive and enjoyable experience. Thanks also go to Kim Eggleston for the book's design, to Michael Victor for the diagrams, and to Mark Monlux for the terrific illustrations. Sally Brunsman, David Clark, Susanne Freet, Dean Holmes, Wendy Maier, and Heidi Saastamoinen also helped this project go smoothly. Literally dozens of other people contributed to this book in one way or another; I didn't have personal contact with any of them, but I appreciate their contributions, too. (Chief among these, I am told, are layout artist Jeannie McGivern and production manager Jean Trenary of ArtSource and Microsoft Press's proof/copy-edit platoon supervised by Brenda Morris: Richard Carey, Roger LeBlanc, Patrick Forgette, Ron Drummond, Patricia Masserman, Paula Thurman, Jocelyn Elliott, Deborah Long, and Devon Musgrave.)

Microsoft Corporation's technical library provided invaluable aid in digging up the hundreds of books and articles that laid the foundation for this book. Keith Barland spearheaded that effort, making my research efforts much less arduous and time-consuming than they otherwise might have been. Other people at the library who helped included Janelle Jones, Christine Shannon, Linda Shaw, Amy Victor, Kyle Wagner, Amy Westfall, and Eilidh Zuvich.

I expound on the virtue of reviews in several places in this book, and this book has benefited greatly from extensive peer reviews. Al Corwin, Pat Forman, Tony Garland, Hank Meuret, and Matt Peloquin stuck with the project from beginning to end. Thanks to them for seeing that the book you

hold in your hands doesn't look very much like the book I originally set out to write! I also received valuable comments from Wayne Beardsley, Duane Bedard, Ray Bernard, Bob Glass, Sharon Graham, Greg Hitchcock, Dave Moore, Tony Pisculli, Steve Rinn, and Bob Stacy—constructive critics, all. David Sommer (age 11) came up with the idea for the last panel of Figure 14-3. Thanks, David. And, finally, I'd like to thank my wife, Tammy, for her moral support and good humor. I have to start working on my third book immediately so that she will stop elbowing me in the ribs and calling me a Two-Time Author!

Bellevue, Washington
June 1996

3

Classic Mistakes

Contents

Related Topics

Risk management: Chapter 5
Rapid-development strategy: Chapter 2

SOFTWARE DEVELOPMENT IS A COMPLICATED ACTIVITY. A typical software project can present more opportunities to learn from mistakes than some people get in a lifetime. This chapter examines some of the classic mistakes that people make when they try to develop software rapidly.

3.1 Case Study in Classic Mistakes

The following case study is a little bit like the children's picture puzzles in which you try to find all the objects whose names begin with the letter "M". How many classic mistakes can you find in the following case study?

Case Study 3-1. Classic Mistakes

Mike, a technical lead for Giga Safe, was eating lunch in his office and looking out his window on a bright April morning.

"Mike, you got the funding for the Giga-Quote program! Congratulations!" It was Bill, Mike's boss at Giga, a medical insurance company. "The executive committee loved the idea of automating our medical insurance quotes. It also

(continued)

Case Study 3-1. Classic Mistakes, *continued*

loved the idea of uploading the day's quotes to the head office every night so that we always have the latest sales leads online. I've got a meeting now, but we can discuss the details later. Good job on that proposal!"

Mike had written the proposal for the Giga-Quote program months earlier, but his proposal had been for a stand-alone PC program without any ability to communicate with the head office. Oh well. This would give him a chance to lead a client-server project in a modern GUI environment—something he had wanted to do. They had almost a year to do the project, and that should give them plenty of time to add a new feature. Mike picked up the phone and dialed his wife's number. "Honey, let's go out to dinner tonight to celebrate…"

The next morning, Mike met with Bill to discuss the project. "OK, Bill. What's up? This doesn't sound like quite the same proposal I worked on."

Bill felt uneasy. Mike hadn't participated in the revisions to the proposal, but there hadn't been time to involve him. Once the executive committee heard about the Giga-Quote program, they'd taken over. "The executive committee loves the idea of building software to automate medical insurance quotes. But they want to be able to transfer the field quotes into the mainframe computer automatically. And they want to have the system done before our new rates take effect January 1. They moved the software-complete date you proposed up from March 1 to November 1, which shrinks your schedule to 6 months."

Mike had estimated the job would take 12 months. He didn't think they had much chance of finishing in 6 months, and he told Bill so. "Let me get this straight," Mike said. "It sounds like you're saying that the committee added a big communications requirement and chopped the schedule from 12 months to 6?"

Bill shrugged. "I know it will be a challenge, but you're creative, and I think you can pull it off. They approved the budget you wanted, and adding the communications link can't be that hard. You asked for 36 staff-months, and you got it. You can recruit anyone you like to work on the project and increase the team size, too." Bill told him to go talk with some other developers and figure out a way to deliver the software on time.

Mike got together with Carl, another technical lead, and they looked for ways to shorten the schedule. "Why don't you use C++ and object-oriented design?" Carl asked. "You'll be more productive than with C, and that should shave a month or two off the schedule." Mike thought that sounded good. Carl also knew of a report-building tool that was supposed to cut development time in half. The project had a lot of reports, so those two changes would get them down to about 9 months. They were due for newer, faster hardware, too, and that could shave off a couple weeks. If he could recruit really top-notch developers, that might bring them down to about 7 months. That should be close enough. Mike took his findings back to Bill.

(continued)

"Look," Bill said. "Getting the schedule down to 7 months is good, but it isn't good enough. The committee was very clear about the 6-month deadline. They didn't give me a choice. I can get you the new hardware you want, but you and your team are going to have to find some way to get the schedule down to 6 months or work some overtime to make up the difference."

Mike considered the fact that his initial estimate had just been a ballpark guess and thought maybe he could pull it off in 6 months. "OK, Bill. I'll hire a couple of sharp contractors for the project. Maybe we can find some people with communications experience to help with uploading data from the PC to the mainframe."

By May 1, Mike had put a team together. Jill, Sue, and Tomas were solid, in-house developers, and they happened to be unassigned. He rounded out the team with Keiko and Chip, two contractors. Keiko had experience both on PCs and the kind of mainframe they would interface with. Jill and Tomas had interviewed Chip and recommended against hiring him, but Mike was impressed. He had communications experience and was available immediately, so Mike hired him anyway.

At the first team meeting, Bill told the team that the Giga-Quote program was strategically important to the Giga Safe Corporation. Some of the top people in the company would be watching them. If they succeeded, there would be rewards all around. He said he was sure that they could pull it off.

After Bill's pep talk, Mike sat down with the team and laid out the schedule. The executive committee had more or less handed them a specification, and they would spend the next 2 weeks filling in the gaps. Then they'd spend 6 weeks on design, which would leave them 4 months for construction and testing. His seat-of-the-pants estimate was that the final product would consist of about 30,000 lines of code in C++. Everyone around the table nodded agreement. It was ambitious, but they'd known that when they signed up for the project.

The next week, Mike met with Stacy, the testing lead. She explained that they should begin handing product builds over to testing no later than September 1, and they should aim to hand over a feature-complete build by October 1. Mike agreed.

The team finished the requirements specification quickly, and dove into design. They came up with a design that seemed to make good use of C++'s features.

They finished the design by June 15, ahead of schedule, and began coding like crazy to meet their goal of a first-release-to-testing by September 1. Work on the project wasn't entirely smooth. Neither Jill nor Tomas liked Chip, and Sue had also complained that he wouldn't let anyone near his code. Mike

(continued)

Case Study 3-1. Classic Mistakes, *continued*

attributed the personality clashes to the long hours everyone was working. Nevertheless, by early August, they reported that they were between 85-percent and 90-percent done.

In mid-August, the actuarial department released the rates for the next year, and the team discovered that they had to accommodate an entirely new rate structure. The new rating method required them to ask questions about exercise habits, drinking habits, smoking habits, recreational activities, and other factors that hadn't been included in the rating formulas before. C++, they thought, was supposed to shield them from the effects of such changes. They had been counting on just plugging some new numbers into a ratings table. But they had to change the input dialogs, database design, database access, and communications objects to accommodate the new structure. As the team scrambled to retrofit their design, Mike told Stacy that they might be a few days late releasing the first build to testing.

The team didn't have a build ready by September 1, and Mike assured Stacy that the build was only a day or two away.

Days turned into weeks. The October 1 deadline for handing over the feature-complete build to testing came and went. Development still hadn't handed over the first build to testing. Stacy called a meeting with Bill to discuss the schedule. "We haven't gotten a build from development yet," she said. "We were supposed to get our first build on September 1, and since we haven't gotten one yet, they've got to be at least a full month behind schedule. I think they're in trouble."

"They're in trouble, all right," Bill said. "Let me talk to the team. I've promised 600 agents that they would have this program by November 1. We have to get that program out in time for the rate change."

Bill called a team meeting. "This is a fantastic team, and you should be meeting your commitments," he told them. "I don't know what's gone wrong here, but I expect everyone to work hard and deliver this software on time. You can still earn your bonuses, but now you're going to have to work for them. As of now, I'm putting all of you on a 6-day-per-week, 10-hour-per-day schedule until this software is done." After the meeting, Jill and Tomas grumbled to Mike about not needing to be treated like children, but they agreed to work the hours Bill wanted.

The team slipped the schedule two weeks, promising a feature-complete build by November 15. That allowed for 6 weeks of testing before the new rates went into effect in January.

The team released its first build to testing 4 weeks later on November 1 and met to discuss a few remaining problem areas.

(continued)

Case Study 3-1. Classic Mistakes, *continued*

Tomas was working on report generation and had run into a roadblock. "The quote summary page includes a simple bar chart. I'm using a report generator that's supposed to generate bar charts, but the only way it will generate them is on pages by themselves. We have a requirement from the sales group to put the text and bar charts on the same page. I've figured out that I can hack up a report with a bar chart by passing in the report text as a legend to the bar-chart object. It's definitely a hack, but I can always go back and reimplement it more cleanly after the first release."

Mike responded, "I don't see where the issue is. We have to get the product out, and we don't have time to make the code perfect. Bill has made it crystal clear that there can't be any more slips. Do the hack."

Chip reported that his communications code was 95-percent done and that it worked, but he still had a few more tests to run. Mike caught Jill and Tomas rolling their eyes, but he decided to ignore it.

The team worked hard through November 15, including working almost all the way through the nights of the 14th and 15th, but they still didn't make their November 15 release date. The team was exhausted, but on the morning of the 16th, it was Bill who felt sick. Stacy had called to tell him that development hadn't released its feature-complete build the day before. Last week he had told the executive committee that the project was on track. Another project manager, Claire, had probed into the team's progress, saying that she had heard that they weren't making their scheduled releases to testing. Bill thought Claire was uptight, and he didn't like her. He had assured her that his team was definitely on track to make their scheduled releases.

Bill told Mike to get the team together, and when he did, they looked defeated. A month and a half of 60-hour weeks had taken their toll. Mike asked what time today they would have the build ready, but the only response he got was silence. "What are you telling me?" he said. "We are going to have the feature-complete build today, aren't we?"

"Look, Mike," Tomas said. "I can hand off my code today and call it 'feature complete', but I've probably got 3 weeks of cleanup work to do once I hand it off." Mike asked what Tomas meant by "cleanup." "I haven't gotten the company logo to show up on every page, and I haven't gotten the agent's name and phone number to print on the bottom of every page. It's little stuff like that. All of the important stuff works fine. I'm 99-percent done."

"I'm not exactly 100-percent done either," Jill admitted. "My old group has been calling me for technical support a lot, and I've been spending a couple hours a day working for them. Plus, I had forgotten until just now that we were supposed to give the agents the ability to put their names and phone numbers on the reports. I haven't implemented the dialogs to input that data yet, and I still have to do some of the other housekeeping dialogs, too. I didn't think we needed them to make our 'feature-complete' milestone."

(continued)

Case Study 3-1. Classic Mistakes, *continued*

Now Mike started to feel sick, too. "If I'm hearing what I think I'm hearing, you're telling me that we're 3 weeks away from having feature-complete software. Is that right?"

"Three weeks *at least*," Jill said. The rest of the developers agreed. Mike went around the table one by one and asked the developers if they could completely finish their assignments in 3 weeks. One by one, the developers said that if they worked hard, they thought they could make it.

Later that day, after a long, uncomfortable discussion, Mike and Bill agreed to slip the schedule 3 weeks to December 5, as long as the team promised to work 12-hour days instead of 10. Bill said he needed to show his boss that he was holding the development team's feet to the fire. The revised schedule meant that they would have to test the code and train the field agents concurrently, but that was the only way they could hope to release the software by January 1. Stacy complained that that wouldn't give QA enough time to test the software, but Bill overruled her.

On December 5, the Giga-Quote team handed off the feature-complete Giga-Quote program to testing before noon and left work early to take a long-awaited break. They had worked almost constantly since September 1.

Two days later, Stacy released the first bug list, and all hell broke loose. In two days, the testing group had identified more than 200 defects in the Giga-Quote program, including 23 that were classified as Severity 1—" Must Fix"—errors. "I don't see any way that the software will be ready to release to the field agents by January 1," she said. "It will probably take the test group that long just to write the regression test cases for the defects we've already discovered, and we're finding new defects every hour."

Mike called a staff meeting for 8 o'clock the next morning. The developers were touchy. They said that although there were a few serious problems, a lot of the reported bugs weren't really bugs at all but were misinterpretations of how the program was supposed to operate. Tomas pointed to bug #143 as an example. "The test report for bug #143 says that on the quote summary page, the bar chart is required to be on the right side of the page rather than the left. That's hardly a Sev-1 error. This is typical of the way that testing overreacts to problems."

Mike distributed copies of the bug reports. He tasked the developers to review the bugs that testing had assigned to them and to estimate how much time it would take to fix each one.

When the team met again that afternoon, the news wasn't good. "Realistically, I would estimate that I have 2 weeks' worth of work just to fix the bugs that have already been reported," Sue said. "Plus I still have to finish the referential integrity checks in the database. I've got 4 weeks of work right now, total."

(continued)

Tomas had assigned bug #143 back to testing, changing its priority from Sev-1 to Sev-3—"Cosmetic Change." Testing had responded that Giga-Quote's summary reports had to match similar reports generated by the mainframe policy-renewal program, which were also similar to preprinted marketing materials that the company had used for many years. The company's 600 agents were accustomed to giving their sales pitches with the bar chart on the right, and it had to stay on the right. The bug stayed at Sev-1, and that created a problem.

"Remember the hack I used to get the bar chart and the report to print on the same page in the first place?" Tomas asked. "To put the bar chart on the right, I will have to rewrite this particular report from scratch, which means that I will have to write my own low-level code to do the report formatting and graphics." Mike cringed, and asked for a ballpark estimate of how long all that would take. Tomas said it would take at least 10 days, but he would have to look into it more before he would know for sure.

Before he went home for the day, Mike told Stacy and Bill that the team would work through the holidays and have all the reported defects fixed by January 7. Bill said he had almost been expecting this one and approved a 4-week schedule slip before leaving for a monthlong Caribbean cruise he had been planning since the previous summer.

Mike spent the next month holding the troops together. For 4 months, they had been working as hard as it was possible to work, and he didn't think he could push them any harder. They were at the office 12 hours a day, but they were spending a lot of time reading magazines, paying bills, and talking on the phone. They seemed to make a point of getting irritable whenever he asked how long it would take to get the bug count down. For every bug they fixed, testing discovered two new ones. Bugs that should have taken minutes to fix had projectwide implications and took days instead. They soon realized there was no way they could fix all the defects by January 7.

On January 7, Bill returned from his vacation, and Mike told him that the development team would need another 4 weeks. "Get serious," Bill said. "I've got 600 field agents who are tired of getting jerked around by a bunch of computer guys. The executive committee is talking about canceling the project. You have to find a way to deliver the software within the next 2 weeks, no matter what."

Mike called a team meeting to discuss their options. He told them about Bill's ultimatum and asked for a ballpark estimate of when they could release the product, first just in weeks, then in months. The team was silent. No one would hazard a guess about when they might finally release the product. Mike didn't know what to tell Bill.

(continued)

Case Study 3-1. Classic Mistakes, *continued*

After the meeting, Chip told Mike that he had accepted a contract with a different company that started February 3. Mike began to feel that it would be a relief if the project were canceled.

Mike got Kip, the programmer who had been responsible for the mainframe side of the PC-to-mainframe communications, reassigned to help out on the project and assigned him to fix bugs in the PC communications code. After struggling with Chip's code for a week, Kip realized that it contained some deep conceptual flaws that meant it could never work correctly. Kip was forced to redesign and reimplement the PC side of the PC-to-mainframe communications link.

As Bill rambled on at an executive meeting in the middle of February, Claire finally decided that she had heard enough and called a "stop work" on the Giga-Quote program. She met with Mike on Friday. "This project is out of control," she said. "I haven't gotten a reliable schedule estimate from Bill for months. This was a 6-month project, and it's now more than 3 months late with no end in sight. I've looked over the bug statistics, and the team isn't closing the gap. You're all working such long hours that you're not even making progress anymore. I want you all to take the weekend off; then I want you to develop a detailed, step-by-step report that includes everything—and I do mean *everything*—that remains to be done on that project. I don't want you to force-fit the project into an artificial schedule. If it's going to take another 9 months, I want to know that. I want that report by end-of-work Wednesday. It doesn't have to be fancy, but it does have to be complete."

The development team was glad to have the weekend off, and during the next week they attacked the detailed report with renewed energy. It was on Claire's desk Wednesday. She had the report reviewed by Charles, a software engineering consultant who also reviewed the project's bug statistics. Charles recommended that the team focus its efforts on a handful of error-prone modules, that it immediately institute design and code reviews for all bug fixes, and that the team start working regular hours so that they could get an accurate measure of how much effort was being expended on the project and how much would be needed to finish.

Three weeks later, in the first week in March, the open-bug count had ticked down a notch for the first time. Team morale had ticked up a notch, and based on the steady progress being made, the consultant projected that the software could be delivered—fully tested and reliable—by May 15. Since Giga Safe's semi-annual rate increase would go into effect July 1, Claire set the official launch date for June 1.

Epilogue

The Giga-Quote program was released to the field agents according to plan on June 1. Giga Safe's field agents greeted it with a warm if somewhat skeptical reception.

(continued)

Case Study 3-1. Classic Mistakes, *continued*

The Giga Safe Corporation showed its appreciation for the development team's hard work by presenting each of the developers with a $250 bonus. A few weeks later, Tomas asked for an extended leave of absence, and Jill went to work for another company.

The final Giga-Quote product was delivered in 13 months rather than 6, a schedule overrun of more than 100 percent. The developer effort, including overtime, consisted of 98 staff-months, which was a 170-percent overrun of the planned 36 staff-months.

CROSS-REFERENCE
For a table of ballpark estimates such as these for projects of various sizes, see Section 8.6, "Ballpark Schedule Estimates."

The final product was determined to consist of about 40,000 nonblank, noncomment lines of code in C++, which was about 33 percent more than Mike's seat-of-the-pants guess. As a product that was distributed to 600 in-house sites, Giga-Quote was a hybrid between a business product and a shrink-wrap product. A product of its size and type should normally have been completed in 11.5 months with 71 staff-months of effort. The project had overshot both of those nominals.

3.2 Effect of Mistakes on a Development Schedule

Michael Jackson (the singer, not the computer scientist) sang that "One bad apple don't spoil the whole bunch, baby." That might be true for apples, but it isn't true for software. One bad apple *can* spoil your whole project.

A group of ITT researchers reviewed 44 projects in 9 countries to examine the impact of 13 productivity factors on productivity (Vosburgh et al. 1984). The factors included the use of modern programming practices, code difficulty, performance requirements, level of client participation in requirements specification, personnel experience, and several others. They divided each of the factors into categories that you would expect to be associated with low, medium, and high performance. For example, they divided the "modern programming practices" factor into low use, medium use, and high use. Figure 3-1 on the next page shows what the researchers found for the "use of modern programming practices" factor.

The longer you study Figure 3-1, the more interesting it becomes. The general pattern it shows is representative of the findings for each of the productivity factors studied. The ITT researchers found that projects in the categories that they expected to have poor productivity did in fact have poor productivity, such as the narrow range shown in the Low category in Figure 3-1. But productivity in the high-performance categories varied greatly, such as the wide range shown in the High category in Figure 3-1. Productivity of projects in the High category varied from poor to excellent.

CROSS-REFERENCE
For more discussion of this specific graph, see Section 4.2, "Technical Fundamentals."

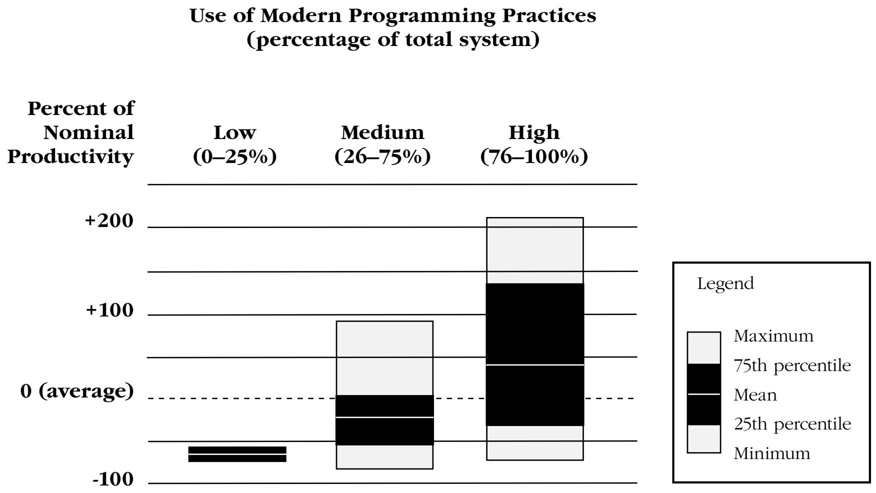

Figure 3-1. *Findings for "Use of Modern Programming Practices" factor (Vosburgh et al. 1984). Doing a few things right doesn't guarantee rapid development. You also have to avoid doing anything wrong.*

CROSS-REFERENCE
For more on the role that mistakes play in rapid development, see Section 2.1, "General Strategy for Rapid Development."

That projects that were expected to have poor productivity do in fact have poor productivity shouldn't surprise you. But the finding that many of the projects expected to have excellent productivity actually have poor productivity just might be a surprise. What this graph and other graphs like it throughout the book show is that the use of any specific best practice is necessary but not sufficient for achieving maximum development speed. Even if you do a few things right, such as making high use of modern programming practices, you might still make a mistake that nullifies your productivity gains.

When thinking about rapid development, it's tempting to think that all you have to do is identify the root causes of slow development and eliminate them—and then you'll have rapid development. The problem is that there aren't just a handful of root causes of slow development, and in the end trying to identify the root causes of slow development isn't very useful. It's like asking, 'What is the root cause of my not being able to run a 4-minute mile?' Well, I'm too old. I weigh too much. I'm too out of shape. I'm not willing to train that hard. I don't have a world-class coach or athletic facility. I wasn't all that fast even when I was younger. The list goes on and on.

When you talk about exceptional achievements, the reasons that people don't rise to the top are simply too numerous to list. The Giga-Quote team in Case Study 3-1 made many of the mistakes that have plagued software developers since the earliest days of computing. The software-development road is mined with potholes, and the potholes you fall into partially determine how quickly or slowly you develop software.

In software, one bad apple can spoil the whole bunch, baby. To slip into slow development, all you need to do is make one really big mistake; to achieve rapid development you need to avoid making *any* big mistakes. The next section lists the most common of those big mistakes.

3.3 Classic Mistakes Enumerated

Some ineffective development practices have been chosen so often, by so many people, with such predictable, bad results that they deserve to be called "classic mistakes." Most of the mistakes have a seductive appeal. Do you need to rescue a project that's behind schedule? Add more people! Do you want to reduce your schedule? Schedule more aggressively! Is one of your key contributors aggravating the rest of the team? Wait until the end of the project to fire him! Do you have a rush project to complete? Take whatever developers are available right now and get started as soon as possible!

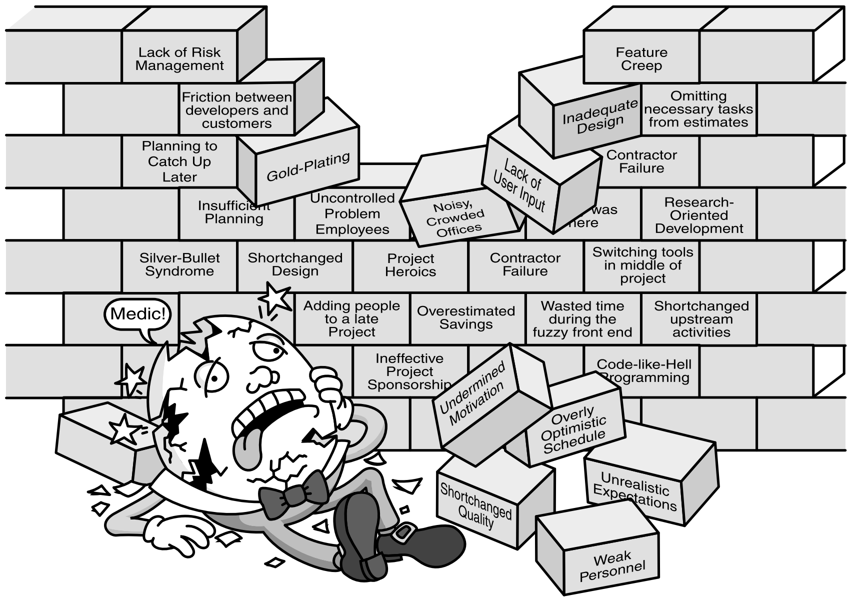
Lack of Risk Management
Friction between developers and customers
Planning to Catch Up Later
Gold-Plating
Insufficient Planning
Uncontrolled Problem Employees
Noisy, Crowded Offices
Lack of User Input
Feature Creep
Inadequate Design
Omitting necessary tasks from estimates
Contractor Failure
was here
Research-Oriented Development
Silver-Bullet Syndrome
Shortchanged Design
Project Heroics
Contractor Failure
Switching tools in middle of project
Medic!
Adding people to a late Project
Overestimated Savings
Wasted time during the fuzzy front end
Shortchanged upstream activities
Ineffective Project Sponsorship
Code-like-Hell Programming
Undermined Motivation
Overly Optimistic Schedule
Shortchanged Quality
Unrealistic Expectations
Weak Personnel

Figure 3-2. *The software project was riddled with mistakes, and all the king's managers and technical leads couldn't rescue the project for anyone's sake.*

Developers, managers, and customers usually have good reasons for making the decisions they do, and the seductive appeal of the classic mistakes is part of the reason these mistakes have been made so often. But because they have been made so many times, their consequences have become easy to predict. And classic mistakes rarely produce the results that people hope for.

This section enumerates about two dozen classic mistakes. I have personally seen each of these mistakes made at least once, and I've made more than a few of them myself. Many of them crop up in Case Study 3-1. The common denominator of these mistakes is that you won't necessarily get rapid development if you avoid these mistakes, but you will definitely get slow development if you don't avoid them.

If some of these mistakes sound familiar, take heart—many other people have made them too. Once you understand their effect on development speed you can use this list to help with your project planning and risk management.

Some of the more significant mistakes are discussed in their own sections in other parts of this book. Others are not discussed further. For ease of reference, the list has been divided along the development-speed dimensions of people, process, product, and technology.

People

Here are some of the people-related classic mistakes.

CROSS-REFERENCE
For more on the uses and misuses of motivation, see Chapter 11, "Motivation."

1: Undermined motivation. Study after study has shown that motivation probably has a larger effect on productivity and quality than any other factor (Boehm 1981). In Case Study 3-1, management took steps that undermined morale throughout the project—from giving a hokey pep talk at the beginning to requiring overtime in the middle, from going on a long vacation while the team worked through the holidays to providing end-of-project bonuses that worked out to less than a dollar per overtime hour at the end.

2: Weak personnel. After motivation, either the individual capabilities of the team members or their relationship as a team probably has the greatest influence on productivity (Boehm 1981, Lakhanpal 1993). Hiring from the bottom of the barrel will threaten a rapid-development effort. In the case study, personnel selections were made with an eye toward who could be hired fastest instead of who would get the most work done over the life of the project. That practice gets the project off to a quick start but doesn't set it up for rapid completion.

CROSS-REFERENCE
For more on creating effective teams, see Chapter 12, "Teamwork."

3: Uncontrolled problem employees. Failure to deal with problem personnel also threatens development speed. This is a common problem and has been well-understood at least since Gerald Weinberg published *Psychology* *of Computer Programming* in 1971. Failure to take action to deal with a problem employee is the most common complaint that team members have about their leaders (Larson and LaFasto 1989). In Case Study 3-1, the team knew that Chip was a bad apple, but the team lead didn't do anything about it. The result—redoing all of Chip's work—was predictable.

CROSS-REFERENCE
For more on heroics and commitment-based projects, see Section 2.5, "An Alternative Rapid-Development Strategy," "Commitment-Based Scheduling" in Section 8.5, and Chapter 34, "Signing Up."

4: Heroics. Some software developers place a high emphasis on project heroics (Bach 1995). But I think that they do more harm than good. In the case study, mid-level management placed a higher premium on can-do attitudes than on steady and consistent progress and meaningful progress reporting. The result was a pattern of scheduling brinkmanship in which impending schedule slips weren't detected, acknowledged, or reported up the management chain until the last minute. A small development team and its immediate management held an entire company hostage because they wouldn't admit that they were having trouble meeting their schedule. An emphasis on heroics encourages extreme risk taking and discourages cooperation among the many stakeholders in the software-development process.

Some managers encourage heroic behavior when they focus too strongly on can-do attitudes. By elevating can-do attitudes above accurate-and-sometimes-gloomy status reporting, such project managers undercut their ability to take corrective action. They don't even know they need to take corrective action until the damage has been done. As Tom DeMarco says, can-do attitudes escalate minor setbacks into true disasters (DeMarco 1995).

CROSS-REFERENCE
For alternative means of rescuing a late project, see Chapter 16, "Project Recovery."

5: Adding people to a late project. This is perhaps the most classic of the classic mistakes. When a project is behind, adding people can take more productivity away from existing team members than it adds through new ones. Fred Brooks likened adding people to a late project to pouring gasoline on a fire (Brooks 1975).

CROSS-REFERENCE
For more on the effects of the physical environment on productivity, see Chapter 30, "Productivity Environments."

6: Noisy, crowded offices. Most developers rate their working conditions as unsatisfactory. About 60 percent report that they are neither sufficiently quiet nor sufficiently private (DeMarco and Lister 1987). Workers who occupy quiet, private offices tend to perform significantly better than workers who occupy noisy, crowded work bays or cubicles. Noisy, crowded work environments lengthen development schedules.

CROSS-REFERENCE
For more on effective customer relations, see Chapter 10, "Customer-Oriented Development."

7: Friction between developers and customers. Friction between developers and customers can arise in several ways. Customers may feel that developers are not cooperative when they refuse to sign up for the development schedule that the customers want or when they fail to deliver on their promises. Developers may feel that customers are unreasonably insisting on unrealistic schedules or requirements changes after the requirements have been baselined. There might simply be personality conflicts between the two groups.

The primary effect of this friction is poor communication, and the secondary effects of poor communication include poorly understood requirements, poor user-interface design, and, in the worst case, customers' refusing to accept the completed product. On average, friction between customers and software developers becomes so severe that both parties consider canceling the project (Jones 1994). Such friction is time-consuming to overcome, and it distracts both customers and developers from the real work of the project.

CROSS-REFERENCE
For more on setting expectations, see Section 10.3, "Managing Customer Expectations."

8: Unrealistic expectations. One of the most common causes of friction between developers and their customers or managers is unrealistic expectations. In Case Study 3-1, Bill had no sound reason to think that the Giga-Quote program could be developed in 6 months, but that's when the company's executive committee wanted it done. Mike's inability to correct that unrealistic expectation was a major source of problems.

In other cases, project managers or developers ask for trouble by getting funding based on overly optimistic schedule estimates. Sometimes they promise a pie-in-the-sky feature set.

Although unrealistic expectations do not in themselves lengthen development schedules, they contribute to the perception that development schedules are too long, and that can be almost as bad. A Standish Group survey listed realistic expectations as one of the top five factors needed to ensure the success of an in-house business-software project (Standish Group 1994).

9: Lack of effective project sponsorship. High-level project sponsorship is necessary to support many aspects of rapid development, including realistic planning, change control, and the introduction of new development practices. Without an effective executive sponsor, other high-level personnel in your organization can force you to accept unrealistic deadlines or make changes that undermine your project. Australian consultant Rob Thomsett argues that lack of an effective executive sponsor virtually guarantees project failure (Thomsett 1995).

10: Lack of stakeholder buy-in. All the major players in a software-development effort must buy in to the project. That includes the executive sponsor, team leader, team members, marketing staff, end-users, customers, and anyone else who has a stake in it. The close cooperation that occurs only when you have complete buy-in from all stakeholders allows for precise coordination of a rapid-development effort that is impossible to attain without good buy-in.

11: Lack of user input. The Standish Group survey found that the number one reason that IS projects succeed is because of user involvement (Standish Group 1994). Projects without early end-user involvement risk misunderstanding the projects' requirements and are vulnerable to time-consuming feature creep later in the project.

CROSS-REFERENCE
For more on healthy politics, see Section 10.3, "Managing Customer Expectations."

12: Politics placed over substance. Larry Constantine reported on four teams that had four different kinds of political orientations (Constantine 1995a). "Politicians" specialized in "managing up," concentrating on relationships with their managers. "Researchers" concentrated on scouting out and gathering information. "Isolationists" kept to themselves, creating project boundaries that they kept closed to non-team members. "Generalists" did a little bit of everything: they tended their relationships with their managers, performed research and scouting activities, and coordinated with other teams through the course of their normal workflow. Constantine reported that initially the political and generalist teams were both well regarded by top management. But after a year and a half, the political team was ranked dead last. Putting politics over results is fatal to speed-oriented development.

13: Wishful thinking. I am amazed at how many problems in software development boil down to wishful thinking. How many times have you heard statements like these from different people:

> "None of the team members really believed that they could complete the project according to the schedule they were given, but they thought that maybe if everyone worked hard, and nothing went wrong, and they got a few lucky breaks, they just might be able to pull it off."

> "Our team hasn't done very much work to coordinate the interfaces among the different parts of the product, but we've all been in good communication about other things, and the interfaces are relatively simple, so it'll probably take only a day or two to shake out the bugs."

> "We know that we went with the low-ball contractor on the database subsystem, and it was hard to see how they were going to complete the work with the staffing levels they specified in their proposal. They didn't have as much experience as some of the other contractors, but maybe they can make up in energy what they lack in experience. They'll probably deliver on time."

> "We don't need to show the final round of changes to the prototype to the customer. I'm sure we know what they want by now."

> "The team is saying that it will take an extraordinary effort to meet the deadline, and they missed their first milestone by a few days, but I think they can bring this one in on time."

Wishful thinking isn't just optimism. It's closing your eyes and hoping something works when you have no reasonable basis for thinking it will. Wishful thinking at the beginning of a project leads to big blowups at the end of a project. It undermines meaningful planning and may be at the root of more software problems than all other causes combined.

Process

Process-related mistakes slow down projects because they squander people's talents and efforts. Here are some of the worst process-related mistakes.

CROSS-REFERENCE
For more on unrealistic schedules, see Section 9.1, "Overly Optimistic Scheduling."

14: Overly optimistic schedules. The challenges faced by someone building a 3-month application are quite different from the challenges faced by someone building a 1-year application. Setting an overly optimistic schedule sets a project up for failure by underscoping the project, undermining effective planning, and abbreviating critical upstream development activities such as requirements analysis and design. It also puts excessive pressure on developers, which hurts long-term developer morale and productivity. This was a major source of problems in Case Study 3-1.

CROSS-REFERENCE
For more on risk management, see Chapter 5, "Risk Management."

15: Insufficient risk management. Some mistakes are not common enough to be considered classic. Those are called "risks." As with the classic mistakes, if you don't actively manage risks, only one thing has to go wrong to change your project from a rapid-development project to a slow-development one. The failure to manage such unique risks is a classic mistake.

CROSS-REFERENCE
For more on contractors, see Chapter 28, "Outsourcing."

16: Contractor failure. Companies sometimes contract out pieces of a project when they are too rushed to do the work in-house. But contractors frequently deliver work that's late, that's of unacceptably low quality, or that fails to meet specifications (Boehm 1989). Risks such as unstable requirements or ill-defined interfaces can be magnified when you bring a contractor into the picture. If the contractor relationship isn't managed carefully, the use of contractors can slow a project down rather than speed it up.

CROSS-REFERENCE
For more on planning, see "Planning" in Section 4.1.

17: Insufficient planning. If you don't plan to achieve rapid development, you can't expect to achieve it.

CROSS-REFERENCE
For more on planning under pressure, see Section 9.2, "Beating Schedule Pressure," and Chapter 16, "Project Recovery."

18: Abandonment of planning under pressure. Project teams make plans and then routinely abandon them when they run into schedule trouble (Humphrey 1989). The problem isn't so much in abandoning the plan as in failing to create a substitute, and then falling into code-and-fix mode instead. In Case Study 3-1, the team abandoned its plan after it missed its first delivery, and that's typical. The work after that point was uncoordinated and awkward—to the point that Jill even started working on a project for her old group part of the time and no one even knew it.

19: Wasted time during the fuzzy front end. The "fuzzy front end" is the time before the project starts, the time normally spent in the approval and budgeting process. It's not uncommon for a project to spend months or years in the fuzzy front end and then to come out of the gates with an aggressive schedule. It's much easier and cheaper and less risky to save a few weeks or months in the fuzzy front end than it is to compress a development schedule by the same amount.

CROSS-REFERENCE
For more on shortchanging upstream activities, see "Effects of Overly Optimistic Schedules" in Section 9.1.

20: Shortchanged upstream activities. Projects that are in a hurry try to cut out nonessential activities, and since requirements analysis, architecture, and design don't directly produce code, they are easy targets. On one disastrous project that I took over, I asked to see the design. The team lead told me, "We didn't have time to do a design."

HARD DATA

The results of this mistake—also known as "jumping into coding"—are all too predictable. In the case study, a design hack in the bar-chart report was substituted for quality design work. Before the product could be released, the hack work had to be thrown out and the higher-quality work had to be done anyway. Projects that skimp on upstream activities typically have to do the same work downstream at anywhere from 10 to 100 times the cost of doing it properly in the first place (Fagan 1976; Boehm and Papaccio 1988). If you can't find the 5 hours to do the job right the first time, where are you going to find the 50 hours to do it right later?

21: Inadequate design. A special case of shortchanging upstream activities is inadequate design. Rush projects undermine design by not allocating enough time for it and by creating a pressure cooker environment that makes thoughtful consideration of design alternatives difficult. The design emphasis is on expediency rather than quality, so you tend to need several ultimately time-consuming design cycles before you can finally complete the system.

CROSS-REFERENCE
For more on quality assurance, see Section 4.3, "Quality-Assurance Fundamentals."

HARD DATA

22: Shortchanged quality assurance. Projects that are in a hurry often cut corners by eliminating design and code reviews, eliminating test planning, and performing only perfunctory testing. In the case study, design reviews and code reviews were given short shrift in order to achieve a perceived schedule advantage. As it turned out, when the project reached its feature-complete milestone it was still too buggy to release for 5 more months. This result is typical. Shortcutting 1 day of QA activity early in the project is likely to cost you from 3 to 10 days of activity downstream (Jones 1994). This shortcut undermines development speed.

CROSS-REFERENCE
For more on management controls, see "Tracking" in Section 4.1 and Chapter 27, "Miniature Milestones."

23: Insufficient management controls. In the case study, few management controls were in place to provide timely warnings of impending schedule slips, and the few controls that were in place at the beginning were abandoned once the project ran into trouble. Before you can keep a project on track, you have to be able to tell whether it's on track in the first place.

CROSS-REFERENCE
For more on premature convergence, see "Premature convergence" in Section 9.1.

24: Premature or overly frequent convergence. Shortly before a product is scheduled to be released, there is a push to prepare the product for release—improve the product's performance, print final documentation, incorporate final help-system hooks, polish the installation program, stub out functionality that's not going to be ready on time, and so on. On rush projects, there is a tendency to force convergence early. Since it's not possible to force the product to converge when desired, some rapid-development projects try to force convergence a half dozen times or more before they finally succeed. The extra convergence attempts don't benefit the product. They just waste time and prolong the schedule.

CROSS-REFERENCE
For a list of commonly omitted tasks, see "Don't omit common tasks" in Section 8.3.

25: Omitting necessary tasks from estimates. If people don't keep careful records of previous projects, they forget about the less visible tasks, but those tasks add up. Omitted effort often adds about 20 to 30 percent to a development schedule (van Genuchten 1991).

26: Planning to catch up later. One kind of reestimation is responding inappropriately to a schedule slip. If you're working on a 6-month project, and it takes you 3 months to meet your 2-month milestone, what do you do? Many projects simply plan to catch up later, but they never do. You learn more about the product as you build it, including more about what it will take to build it. That learning needs to be reflected in the reestimated schedule.

CROSS-REFERENCE
For more on reestimation, see "Recalibration" in Section 8.7.

Another kind of reestimation mistake arises from product changes. If the product you're building changes, the amount of time you need to build it changes too. In Case Study 3-1, major requirements changed between the original proposal and the project start without any corresponding reestimation of schedule or resources. Piling on new features without adjusting the schedule guarantees that you will miss your deadline.

CROSS-REFERENCE
For more on the Code-and-Fix approach, see Section 7.2, "Code-and-Fix."

27: Code-like-hell programming. Some organizations think that fast, loose, all-as-you-go coding is a route to rapid development. If the developers are sufficiently motivated, they reason, they can overcome any obstacles. For reasons that will become clear throughout this book, this is far from the truth. This approach is sometimes presented as an "entrepreneurial" approach to software development, but it is really just a cover for the old Code-and-Fix paradigm combined with an ambitious schedule, and that combination almost never works. It's an example of two wrongs not making a right.

Product

Here are classic mistakes related to the way the product is defined.

28: Requirements gold-plating. Some projects have more requirements than they need, right from the beginning. Performance is stated as a requirement more often than it needs to be, and that can unnecessarily lengthen a software schedule. Users tend to be less interested in complex features than marketing and development are, and complex features add disproportionately to a development schedule.

CROSS-REFERENCE
For more on feature creep, see Chapter 14, "Feature-Set Control."

29: Feature creep. Even if you're successful at avoiding requirements gold-plating, the average project experiences about a 25-percent change in requirements over its lifetime (Jones 1994). Such a change can produce at least a 25-percent addition to the software schedule, which can be fatal to a rapid-development project.

CROSS-REFERENCE
For an example of the way that developer gold-plating can occur even accidentally, see "Unclear or Impossible Goals" in Section 14.2.

30: Developer gold-plating. Developers are fascinated by new technology and are sometimes anxious to try out new features of their language or environment or to create their own implementation of a slick feature they saw in another product—whether or not it's required in their product. The effort required to design, implement, test, document, and support features that are not required lengthens the schedule.

31: Push-me, pull-me negotiation. One bizarre negotiating ploy occurs when a manager approves a schedule slip on a project that's progressing slower than expected and then adds completely new tasks after the schedule change. The underlying reason for this is hard to fathom, because the manager who approves the schedule slip is implicitly acknowledging that the schedule was in error. But once the schedule has been corrected, the same person takes explicit action to make it wrong again. This can't help but undermine the schedule.

32: Research-oriented development. Seymour Cray, the designer of the Cray supercomputers, says that he does not attempt to exceed engineering limits in more than two areas at a time because the risk of failure is too high (Gilb 1988). Many software projects could learn a lesson from Cray. If your project strains the limits of computer science by requiring the creation of new algorithms or new computing practices, you're not doing software development; you're doing software research. Software-development schedules are reasonably predictable; software research schedules are not even theoretically predictable.

If you have product goals that push the state of the art—algorithms, speed, memory usage, and so on—you should assume that your scheduling is highly speculative. If you're pushing the state of the art and you have any other weaknesses in your project—personnel shortages, personnel weaknesses, vague requirements, unstable interfaces with outside contractors—you can throw predictable scheduling out the window. If you want to advance the state of the art, by all means, do it. But don't expect to do it rapidly!

Technology

The remaining classic mistakes have to do with the use and misuse of modern technology.

CROSS-REFERENCE
For more on the silver-bullet syndrome, see Section 15.5, "Silver-Bullet Syndrome."

33: Silver-bullet syndrome. In the case study, there was too much reliance on the advertised benefits of previously unused technologies (report generator, object-oriented design, and C++) and too little information about how well they would do in this particular development environment. When project teams latch onto a single new practice, new technology, or rigid process and expect it to solve their schedule problems, they are inevitably disappointed (Jones 1994).

CROSS-REFERENCE
For more on estimating savings from productivity tools, see "How Much Schedule Reduction to Expect" in Section 15.4.

34: Overestimated savings from new tools or methods. Organizations seldom improve their productivity in giant leaps, no matter how many new tools or methods they adopt or how good they are. Benefits of new practices are partially offset by the learning curves associated with them, and learning to use new practices to their maximum advantage takes time. New practices also entail new risks, which you're likely to discover only by using them. You are more likely to experience slow, steady improvement on the order of a few percent per project than you are to experience dramatic gains. The team in Case Study 3-1 should have planned on, at most, a 10-percent gain in productivity from the use of the new technologies instead of assuming that they would nearly double their productivity.

CROSS-REFERENCE
For more on reuse, see Chapter 33, "Reuse."

A special case of overestimated savings arises when projects reuse code from previous projects. This kind of reuse can be a very effective approach, but the time savings is rarely as dramatic as expected.

35: Switching tools in the middle of a project. This is an old standby that hardly ever works. Sometimes it can make sense to upgrade incrementally within the same product line, from version 3 to version 3.1 or sometimes even to version 4. But the learning curve, rework, and inevitable mistakes made with a totally new tool usually cancel out any benefit when you're in the middle of a project.

CROSS-REFERENCE
For more on source-code control, see "Software Configuration Management" in Section 4.2.

36: Lack of automated source-code control. Failure to use automated source-code control exposes projects to needless risks. Without it, if two developers are working on the same part of the program, they have to coordinate their work manually. They might agree to put the latest versions of each file into a master directory and to check with each other before copying files into that directory. But someone invariably overwrites someone else's work. People develop new code to out-of-date interfaces and then have to redesign their code when they discover that they were using the wrong version of the interface. Users report defects that you can't reproduce because you have no way to re-create the build they were using. On average, source code changes at a rate of about 10 percent per month, and manual source-code control can't keep up (Jones 1994).

Table 3-1 contains a complete list of classic mistakes.

Table 3-1. Summary of Classic Mistakes

People-Related Mistakes	Process-Related Mistakes	Product-Related Mistakes	Technology-Related Mistakes
1. Undermined motivation	14. Overly optimistic schedules	28. Requirements gold-plating	33. Silver-bullet syndrome
2. Weak personnel	15. Insufficient risk management	29. Feature creep	34. Overestimated savings from new tools or methods
3. Uncontrolled problem employees	16. Contractor failure	30. Developer gold-plating	35. Switching tools in the middle of a project
4. Heroics	17. Insufficient planning	31. Push-me, pull-me negotiation	36. Lack of automated source-code control
5. Adding people to a late project	18. Abandonment of planning under pressure	32. Research-oriented development	
6. Noisy, crowded offices	19. Wasted time during the fuzzy front end		
7. Friction between developers and customers	20. Shortchanged upstream activities		
8. Unrealistic expectations	21. Inadequate design		
9. Lack of effective project sponsorship	22. Shortchanged quality assurance		
10. Lack of stakeholder buy-in	23. Insufficient management controls		
11. Lack of user input	24. Premature or overly frequent convergence		
12. Politics placed over substance	25. Omitting necessary tasks from estimates		
13. Wishful thinking	26. Planning to catch up later		
	27. Code-like-hell programming		

3.4 Escape from *Gilligan's Island*

A complete list of classic mistakes would go on for pages more, but those presented are the most common and the most serious. As Seattle University's David Umphress points out, watching most organizations attempt to avoid these classic mistakes seems like watching reruns of *Gilligan's Island*. At the beginning of each episode, Gilligan, the Skipper, or the Professor comes up with a cockamamie scheme to get off the island. The scheme seems as though it's going to work for a while, but as the episode unfolds, something goes wrong, and by the end of the episode the castaways find themselves right back where they started—stuck on the island.

Similarly, most companies at the end of each project find that they have made yet another classic mistake and that they have delivered yet another project behind schedule or over budget or both.

Your Own List of Worst Practices

Be aware of the classic mistakes. Create lists of "worst practices" to avoid on future projects. Start with the list in this chapter. Add to the list by conducting project postmortems to learn from your team's mistakes. Encourage other projects within your organization to conduct postmortems so that you can learn from their mistakes. Exchange war stories with your colleagues in other organizations, and learn from their experiences. Display your list of mistakes prominently so that people will see it and learn not to make the same mistakes yet another time.

Further Reading

Although a few books discuss coding mistakes, there are no books that I know of that describe classic mistakes related to development schedules. Further reading on related topics is provided throughout the rest of this book.

Index