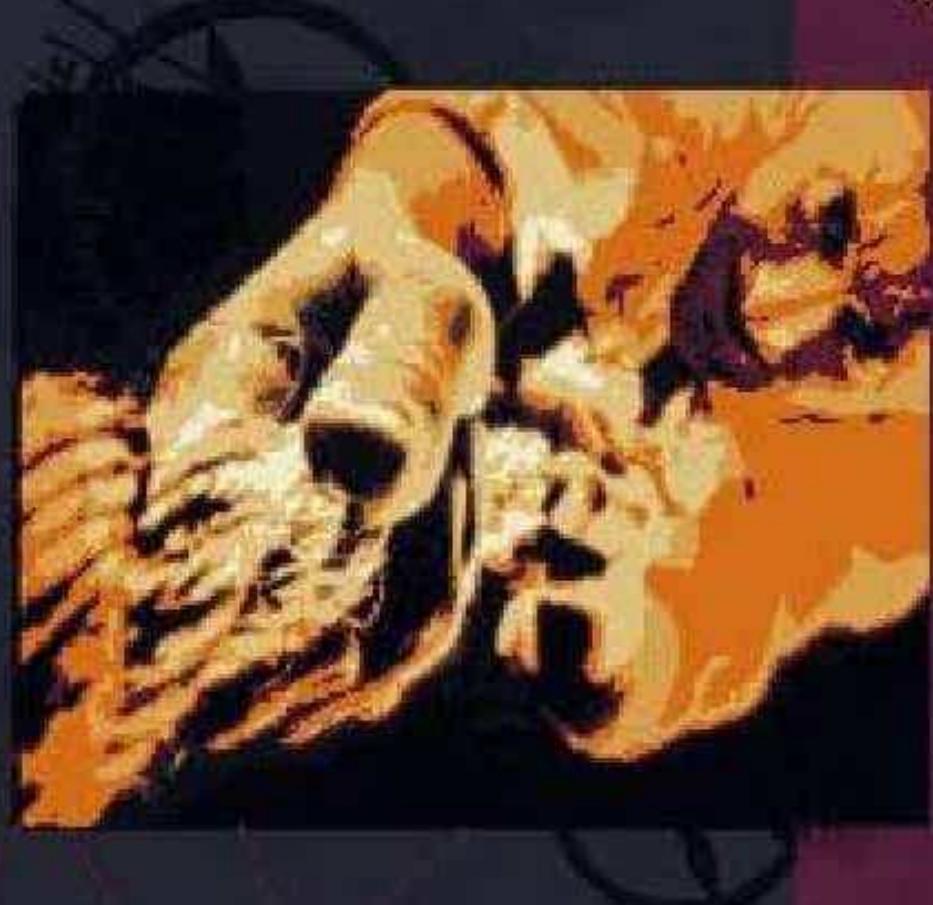


RAPID DEVELOPMENT

*Taming
Wild
Software
Schedules*



Steve McConnell
Author of Code Complete

Microsoft Press

Brought to you by Team FlyHeart



Early Reviews of Steve McConnell's *RAPID DEVELOPMENT*

"*Rapid Development* is instantly recognizable as another member of that rare breed of highly original and definitive books. It addresses a dire need in mainstream commercial or "shrinkwrap" software development that was previously unmet and only dimly perceived. It integrates a vast amount of practical information within a logical, easily grasped structure. It is soundly grounded in the author's mastery of his subject and common sense, and it is backed up by hundreds of references. And, last but hardly least, it is beautifully written in an economical, direct style that makes every page count.

"In *Rapid Development*, we are privileged to see a virtuoso author/programmer and a superb editing and publishing team working together at the top of their form. Very few books I have encountered in the last few years have given me as much pleasure to read as this one."

Ray Duncan, Electronic Review of Computer Books

"One of McConnell's key strengths is the sheer range and volume of his research. It seems every other page contains some empirical data on the best and worst ways to run a software project, and McConnell isn't afraid to let you know where his own opinions lie. You could enjoyably read this book from cover to cover thanks to McConnell's excellent prose, and that's certainly the way to get the most from it. But even if you just dip into it occasionally for ideas and guidance, or splash out on a copy for your team leader, this book is a bargain, and highly recommended."

Richard Stevens, Delphi Magazine

"Computer programs are perhaps the most complex of human creations, and their very utility and pervasiveness makes rapid development of software essential. Steve McConnell is a master at explaining the basic elements of advanced programming, and his new book provides an indispensable roadmap for those who wish to write as fast as humanly possible. Even those who simply admire the artistry and craft of code writing can profit immensely from McConnell's knowledge."

G. Pascal Zachary, author of SHOWSTOPPER!

"I can hear some of you exclaiming, 'how can you possibly recommend a book about software scheduling published by Microsoft Press and written by a consultant to Microsoft?!" Well, put aside any preconceived biases that you might have, as I did mine. This is in fact a tremendous book on effective scheduling of software development, and it drinks deeply from the wisdom of all the classics in the field ... The nine page section entitled 'Classic Mistakes Enumerated' is alone worth the price of admission, and should be required reading for all developers, leads, and managers."

Amazon.com

"This book is an essential guide to controlling development schedules and keeping projects moving on schedule."

Computer Literacy

If you read only one book in the coming year, *Rapid Development* should be that book.

Kevin Weeks, announcing Star Tech Award, Windows Tech Journal

RAPID DEVELOPMENT

*Taming
Wild
Software
Schedules*

Steve McConnell

Microsoft Press

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

Copyright © 1996

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 1-55615-900-5

1. Computer software—Development, I. Title.

QA76.76.D47M393 1996

O05.ro68~dc20

96-21517

CIP

Printed and bound in the United States of America.

11 1213 MLML04 03 02 01 00

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 (206) 936-7329.

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. All other trademarks and service marks are the property of their respective owners.

Acquisitions Editor: David J. Clark

Project Editor: Jack Litewka

Case Studies	ix
Reference Tables	x
Preface	xiii

PART I EFFICIENT DEVELOPMENT

1 Welcome to Rapid Development 1

What Is Rapid Development? • Attaining Rapid Development

2 Rapid-Development Strategy 5

General Strategy for Rapid Development • Four Dimensions of Development Speed • General Kinds of Fast Development • Which Dimension Matters the Most? • An Alternative Rapid-Development Strategy • Further Reading

3 Classic Mistakes 29

Case Study in Classic Mistakes • Effect of Mistakes on a Development Schedule • Classic Mistakes Enumerated • Escape from *Gilligan's Island* • Further Reading

4 Software-Development Fundamentals 51

Management Fundamentals • Technical Fundamentals • Quality-Assurance Fundamentals • Following the Instructions • Further General Reading

5 Risk Management 31

Elements of Risk Management • Risk Identification • Risk Analysis • Risk Prioritization • Risk Control • Risk, High Risk, and Gambling • Further Reading

PART II RAPID DEVELOPMENT

6 Core issues in Rapid Development 109

Does One Size Fit All? • What Kind of Rapid Development Do You Need? • Odds of Completing on Time • Perception and Reality • Where the Time Goes • Development-Speed Trade-Offs • Typical Schedule-Improvement Pattern • Onward to Rapid Development • Further Reading

- 7 Lifecycle Planning 133**
Pure Waterfall • Code-and-Fix • Spiral • Modified Waterfalls • Evolutionary Prototyping • Staged Delivery • Design-to-Schedule • Evolutionary Delivery • Design-to-Tools • Commercial Off-the-Shelf Software • Choosing the Most Rapid Lifecycle for Your Project • Further Reading
- 8 Estimation 163**
The Software-Estimation Story • Estimation-Process Overview • Size Estimation • Effort Estimation • Schedule Estimation • Ballpark Schedule Estimates • Estimate Refinement • Further Reading
- 9 Scheduling 205**
Overly Optimistic Scheduling • Beating Schedule Pressure • Further Reading
- 10 Customer-Oriented Development 233**
Customers' Importance to Rapid Development • Customer-Oriented Practices • Managing Customer Expectations • Further Reading
- 11 Motivation 249**
Typical Developer Motivations • Using the Top Five Motivation Factors • Using Other Motivation Factors • Morale Killers • Further Reading
- 12 Teamwork 273**
Software Uses of Teamwork • Teamwork's Importance to Rapid Development • Creating a High-Performance Team • Why Teams Fail • Long-Term Teambuilding • Summary of Teamwork Guidelines • Further Reading
- 13 Team Structure 297**
Team-Structure Considerations • Team Models • Managers and Technical Leads • Further Reading
- 14 Feature-Set Control 319**
Early Project: Feature-Set Reduction • Mid-Project: Feature-Creep Control • Late Project: Feature Cuts • Further Reading
- 15 Productivity Tools 345**
Role of Productivity Tools in Rapid Development • Productivity-Tool Strategy • Productivity-Tool Acquisition • Productivity-Tool Use • Silver-Bullet Syndrome • Further Reading
- 16 Project Recovery 371**
General Recovery Options • Recovery Plan • Further Reading

PART III BEST PRACTICES

Introduction to Best Practices 390

Organization of Best-Practice Chapters • Summary of Best-Practice Candidates • Summary of Best-Practice Evaluations

- 17 Change Board** 403
- 18 Daily Build and Smoke Test** 405
- 19 Designing for Change** 415
- 20 Evolutionary Delivery** 425
- 21 Evolutionary Prototyping** 433
- 22 Goal Setting** 445
- 23 Inspections** 447
- 24 Joint Application Development (JAD)** 449
- 25 Lifecycie Model Selection** 465
- 26 Measurement** 467
- 27 Miniature Milestones** 481
- 28 Outsourcing** 491
- 29 Principled Negotiation** 503
- 30 Productivity Environments** 505
- 31 Rapid-Development Languages. (RDLs)** 515
- 32 Requirements Scrubbing** 525
- 33 Reuse** 527
- 34 Signing Up** 539
- 35 Spiral Lifecycie Model** 547
- 36 Staged Delivery** 549
- 37 Theory-W Management** 559
- 38 Throwaway Prototyping** 569
- 39 Timebox Development** 575
- 40 Tools Group** 585
- 41 Top-10 Risks List** 587

42 User-Interface Prototyping 589

43 Voluntary Overtime 599

Bibliography 609

Index 625

Case Studies

- 2-1. Rapid Development Without a Clear Strategy 6
- 2-2. Rapid Development with a Clear Strategy 25
- 3-1. Classic Mistakes 29
- 4-1. Lack of Fundamentals 52
- 5-1. Lack of Contractor Risk Management 82
- 5-2. Systematic Risk Management 103
- 6-1. Wandering in the Fuzzy Front End 124
- 7-1. Ineffective Lifecycle Model Selection 134
- 7-2. Effective Lifecycle Model Selection 159
- 8-1. Seat-of-the-Pants Project Estimation 164
- 8-2. Careful Project Estimation 200
- 9-1. A Successful Schedule Negotiation 229
- 10-1. The Requirements Club 234
- 10-2. The Requirements Club Revisited 246
- 11-1. A Disheartening Lunch with the Boss 250
- 11-2. A Highly Motivational Environment 270
- 12-1. You Call This a Team? 274
- 12-2. A High-Performance Team 277
- 12-3. Typical Team-Member Selection 282
- 12-4. A Second High-Performance Team 294
- 13-1. Mismatch Between Project Objectives and Team Structure 297
- 13-2. Good Match Between Project Objectives and Team Structure 315
- 14-1. Managing Change Effectively 342
- 15-1. Ineffective Tool Use 346
- 15-2. Effective Tool Use 368
- 16-1. An Unsuccessful Project Recovery 372
- 16-2. A Successful Project Recovery 385

Reference Tables

- 2-1. Characteristics of Standard Approaches to Schedule-Oriented Development 18
- 2-2. Code-Like-Hell Approach Compared to This Book's Approach 24
- 3-1. Summary of Classic Mistakes 49
- 5-1. Levels of Risk Management 84
- 5-2. Most Common Schedule Risks 86
- 5-3. Potential Schedule Risks 87
- 5-4. Example of a Risk-Assessment Table 92
- 5-5. Example of a Prioritized Risk-Assessment Table 95
- 5-6. Means of Controlling the Most Common Schedule Risks 98
- 5-7. Example of a "Top-10 Risks List" 101
- 6-1. Approximate Activity Breakdown by Size of Project 122
- 7-1. Lifecycle Model Strengths and Weaknesses 156
- 8-1. Estimate Multipliers by Project Phase 169
- 8-2. Function-Point Multipliers 176
- 8-3. Example of Computing the Number of Function Points 177
- 8-4. Example of a Risk-Quantification Estimate 180
- 8-5. Example of a Case-Based Estimate 181
- 8-6. Example of a Confidence-Factor Estimate 182
- 8-7. Exponents for Computing Schedules from Function Points 185
- 8-8. Shortest Possible Schedules. 190
- 8-9. Efficient Schedules 194
- 8-10. Nominal Schedules 196
- 8-11. Example of a Single-Point-Estimation History 197
- 8-12. Example of a Range-Estimation History 198
- 9-1. Scheduling History of Word for Windows 1.0 208
- 11-1. Comparison of Motivators for Programmer Analysts vs. Managers and the General Population 252

11-2	Team Performance Ranked Against Objectives That Teams Were Told to Optimize	256
12-1.	Practical Guidelines for Team Members and Leaders	295
13-1.	Team Objectives and Team Structures	301
15-1.	Example of Savings Realized by Switching from a 3GL to a 4GL for 50 Percent of a 32,000 LOG Project	361
15-2	Example of Savings Realized by Switching from a 3GL to a 4GL for 100 Percent of a 32,000 LOG Project	362
III-1.	Summary of Best-Practice Candidates	396
III-2.	Summary of Best-Practice Evaluations	400
26-1.	Examples of Kinds of Measurement Data	470
26-2.	Example of Time-Accounting Activities	472
28-1.	Vendor-Evaluation Questionnaire	497
28-2.	Contract Considerations	498
30-1.	Differences in Office Environments Between Best and Worst Performers in a Programming Competition	512
31-1.	Approximate Function-Points to Lines-of-Code Conversions	517
31-2.	Approximate Language Levels	519
36-1.	Example of a Staged-Delivery Schedule for a Word Processor	552
37-1.	Project Stakeholders and Their Objectives	560
37-2.	Steps in Theory-W Project Management	562

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 *bestpractices*. 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.

Case Study 2-1. Rapid Development Without a Clear Strategy, *continued*

As the project began, the developers were happy about their private offices, new computers, and soda pop, so they got off to a strong start. It wasn't long before they were voluntarily working well into the evening.

Months went by, and they made steady progress. They produced an early prototype, and continued to produce a steady stream of code. Management kept the pressure on. John reminded Mickey several times of his commitment to a visual freeze at the 8-month mark, which Mickey found irritating, but everything seemed to be progressing nicely.

Bob returned from his bike trip during the project's fourth month, refreshed, and jumped into the project with some new thoughts he'd had while riding. Mickey worried about whether Bob could implement as much functionality as he wanted to in the time allowed, but Bob was committed to his ideas and guaranteed on-time delivery no matter how much work it took.

The team members worked independently on their parts, and as visual freeze approached, they began to integrate their code. They started at 2:00 in the afternoon the day before the visual freeze deadline and soon discovered that the program wouldn't compile, much less run. The combined code had several dozen syntax errors, and it seemed like each one they fixed generated 10 more. At midnight, they decided to call it a night.

The next morning, Kim met with the team. "Is the program ready to hand over to documentation and testing?"

"Not yet," Mickey said. "We're having some integration problems. We might be ready by this afternoon." The team worked that afternoon and evening, but couldn't fix all of the bugs they were discovering. At the end of the day they conceded that they had no idea how much longer integration would take.

It took two full weeks to fix all the syntax errors and get the system to run at all. When the team turned over the frozen build two weeks late, testing and documentation rejected it immediately. "This is too unstable to document," John said. "It crashes every few minutes, and there are lots of paths we can't even exercise."

Helen agreed. "There's no point in having testers write defect reports when the system is so unstable that it crashes practically every time you make a menu selection."

Mickey agreed with them and said he'd focus his team's efforts on bug fixes. Kim reminded them of the 10-month deadline and said that this product couldn't be late like the last one.

It took a month to make the system reliable enough to begin documenting and testing it. By then they were only two weeks from the 10-month mark, and they worked even harder.

(continued)

Case Study 2-1. Rapid Development Without a Clear Strategy, *continued*

But testing began finding defects faster than the developers could correct them. Fixes to one part of the system frequently caused problems in other parts. There was no chance of making the 10-month ship date. Kim called an emergency meeting. "I can see that you're all working hard," she said, "but that's not good enough. I need results. I've given you every kind of support I know how, and I don't have any software to show for it. If you don't finish this product soon, the company could go under."

As the pressure mounted, morale faded fast. More months went by, the product began to stabilize, and Kim kept the pressure on. Some of the interfaces turned out to be extremely inefficient, and that called for several more weeks of performance work.

Bob, despite working virtually around the clock, delivered his software later than the rest of the team. His code was virtually bug-free, but he had changed some of the user-interface components, and testing and user documentation threw fits.

Mickey met with John and Helen. "You won't like it, but our options are as follows: We can keep Bob's code the way it is and rev the test scripts and user documentation, or we can throw out Bob's code and write it all again. Bob won't rewrite his code, and no one else on the team will either. Looks like you'll have to change the user documentation and test scripts." After putting up token resistance, John and Helen begrudgingly agreed.

In the end, it took the developers 15 months to complete the software. Because of the visual changes, the user documentation missed its slot in the printer's schedule, so after the developers cut the master disks there was a two-week shipping delay while Square-Tech waited for documents to come back from the printer. After release, user response to Square-Calc version 3.0 was lukewarm, and within months it slipped from second place in market share to fourth. Mickey realized that he had delivered his second project 50 percent over schedule, just like the first.

2.1 General Strategy for Rapid Development

The pattern described in Case Study 2-1 is common. Avoidance of the pattern takes effort but is within reach of anyone who is willing to throw out their bad habits. You can achieve rapid development by following a four-part strategy:

1. Avoid classic mistakes.
2. Apply development fundamentals.

3. Manage risks to avoid catastrophic setbacks.
4. Apply schedule-oriented practices such as the three kinds of practices shown in Figure 1-2 in Chapter 1.

As Figure 2-1 suggests, these four practices provide support for the best possible schedule.



Figure 2-1. *The four pillars of rapid development. The best possible schedule depends on classic-mistake avoidance, development fundamentals, and risk management in addition to the use of schedule-oriented practices.*

Rapid product development is not a quick fix for getting one product—which is probably already late—to market faster. Instead, it is a strategic capability that must be built from the ground up.

Preston G. Smith and Donald G. Seinertsen,
Developing Products in Half the Time

Pictures with pillars have become kind of hokey, but the pillars in this picture illustrate several important points.

The optimum support for the best possible schedule is to have all four pillars in place and to make each of them as strong as possible. Without the support of the first three pillars, your ability to achieve the best possible schedule will be in jeopardy. You can use the strongest schedule-oriented practices, but if you make the classic mistake of shortchanging product quality early in the project, you'll waste time correcting defects when it's most expensive to do so. Your project will be late. If you skip the development fundamental of creating a good design before you begin coding, your program can fall apart when the product concept changes partway through development, and your project will be late.. And if you don't manage risks, you can find out just before your release date that a key subcontractor is three months behind schedule. You'll be late again.

Motivation. A person who lacks motivation is unlikely to work hard and is more likely to coast. No factor other than motivation will cause a person to forsake evenings and weekends without being asked to do so. Few other factors can be applied to so many people on so many teams in so many organizations. Motivation is potentially the strongest ally you have on a rapid-development project.

Variations in Productivity

I refer to several ratios related to variations in productivity in this book, and keeping them straight can get confusing. Here's a summary of the variations that researchers have found:

- Greater than 10-to-1 differences in productivity among individuals with different depths and breadths of experience.
- 10-to-1 differences in productivity among individuals with the same levels of experience.
- 5-to-1 differences in productivity among groups with different levels of experience.
- 2.5-to-1 differences in productivity among groups with similar levels of experience.

Process

Process, as it applies to software development, includes both management and technical methodologies. The effect that process has on a development schedule is easier to assess than the effect that people have, and a great deal of work is being done by the Software Engineering Institute and other organizations to document and publicize effective software processes.

Process represents an area of high leverage in improving your development speed—almost as much as people. Ten years ago it might have been reasonable to debate the value of a focus on process, but, as with peopleware, today the pile of evidence in favor of paying attention to process has become overwhelming. Organizations such as Hughes Aircraft, Lockheed, Motorola, NASA, Raytheon, and Xerox that have explicitly focused on improving their development processes have, over several years, cut their times-to-market by about one-half and have reduced cost and defects by factors of 3 to 10 (Pietrasanta 1991a, Myers 1992, Putnam and Myers 1992, Gibbs 1994, Putnam 1994, Basili et al. 1995, Raytheon 1995, Saiedian and Hamilton 1995).

Some people think that attention to process is stifling, and there's no doubt that some processes are overly rigid or overly bureaucratic. A few people have created process standards primarily to make themselves feel powerful. But that's an abuse of power—and the fact that a process focus can be abused should not be allowed to detract from the benefits a process focus

can offer. The most common form of process abuse is neglect, and the effect of that is that intelligent, conscientious developers find themselves working inefficiently and at cross-purposes when there's no need for them to work that way. A focus on process can help.

Rework avoidance. If requirements change in the late stages of project, you might have to redesign, recede, and retest. If you have design problems that you didn't find until system testing, you might have to throw away detailed design and code and then start over. One of the most straightforward ways to save time on a software project is to orient your process so that you avoid doing things twice.



CROSS-REFERENCE

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

CROSS-REFERENCE

For more on development fundamentals, see Section 4.2, "Technical Fundamentals."

CROSS-REFERENCE

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

Quality assurance. Quality assurance has two main purposes. The first purpose is to assure that the product you release has an acceptable level of quality. Although that is an important purpose, it is outside the scope of this book. The second function of quality assurance is to detect errors at the stage when they are least time-consuming (and least costly) to correct. This nearly always means catching errors as close as possible to the time that they are introduced. The longer an error remains in the product, the more tune-consuming (and more costly) it will be to remove. Quality assurance is thus an indispensable part of any serious rapid-development program.

Development fundamentals. Much of the work that has been done in the software-engineering field during the last 20 years has been related to developing software rapidly. A lot of that work has focused on "productivity" rather than on rapid development per se, and, as such, some of it has been oriented toward getting the same work done with fewer people rather than getting a project done faster. You can, however, interpret the underlying principles from a rapid-development viewpoint. The lessons learned from 20 years of hard knocks can help your project to proceed smoothly. Although standard software-engineering practices for analysis, design, construction, integration, and testing won't produce lightning-fast schedules by themselves, they can prevent projects from spinning out of control. Half of the challenge of rapid development is avoiding disaster, and that is an area in which standard software-engineering principles excel.

Risk management. One of the specific practices that's focused on avoiding disaster is risk management. Developing rapidly isn't good enough if you get your feet knocked out from under you two weeks before you're scheduled to ship. Managing schedule-related risks is a necessary component of a rapid-development program.

Resource targeting. Resources can be focused effectively and contribute to overall productivity, or they can be misdirected and used ineffectively. On a rapid-development project, it is even more important than usual that you get the maximum bang for your schedule buck. Best practices such as productivity offices, timebox development, accurate scheduling, and voluntary overtime help to make sure that you get as much work done each day as possible.

CROSS-REFERENCE

For more on lifecycle planning, see Chapter 7, "Lifecycle Planning."

Lifecycle planning. One of the keys to targeting resources effectively is to apply them within a lifecycle framework that makes sense for your specific project. Without an overall lifecycle model, you can make decisions that are individually on target but collectively misdirected. A lifecycle model is useful because it describes a basic management plan. For example, if you have a risky project, a risk-oriented lifecycle model will suit you; and if you have vague requirements, an incremental lifecycle model may work best. Lifecycle models make it easy to identify and organize the many activities required by a software project so that you can do them with the utmost efficiency.

CROSS-REFERENCE

For more on customer orientation, see Chapter 10, "Customer-Oriented Development."

Customer orientation. One of the gestalt shifts between traditional, main-frame software development and more modern development styles has been the switch to a strong focus on customers' needs and desires. Developers have learned that developing software to specification is only half the job. The other half is helping the customer figure out what the product should be, and most of the time that requires an approach other than a traditional paper-specification approach. Putting yourself on the same side as the customer is one of the best ways to avoid the massive rework caused by the customer deciding that the product you just spent 12 months on is not the right product after all. The best practices of staged releases, evolutionary delivery, evolutionary prototyping, throwaway prototyping, and principled negotiation can all give you leverage in this area.

Who Is "The Customer"?

In this book, when I refer to "customers," I'm referring to the people who pay to have the software developed and the people who are responsible for accepting or rejecting the product. On some projects, those will be the same person or group; on others, they'll be different. On some projects, the customer is a real flesh-and-blood client who pays your project's development costs directly. On other projects, it's another internal group within your organization. On still other projects, the customer is the person who plunks down \$200 for a shrink-wrap software package. In that case, the real customer is remote, and there is usually a manager or marketer who represents the customer to you.

Depending on your situation, you might understand the term "customer" to mean "client," "marketeer," "end-user," or "boss."

Efficient Development

CROSS-REFERENCE

For an example of the benefits of efficient development, see Section 4.2, "Technical Fundamentals" and Chapter 4, "Software Development Fundamentals," generally.

As you can see from Table 2-1, average practice is...average. The second approach listed in the table is what I call "efficient development," which is the combination of the first three pillars of maximum development speed as shown in Figure 2-4. That approach produces better than average results in each of the three categories. Many people achieve their schedule goals after they put the first three pillars into place. Some people discover that they didn't need rapid development after all; they just needed to get organized! For many projects, efficient development represents a sensible optimization of cost, schedule, and product characteristics.

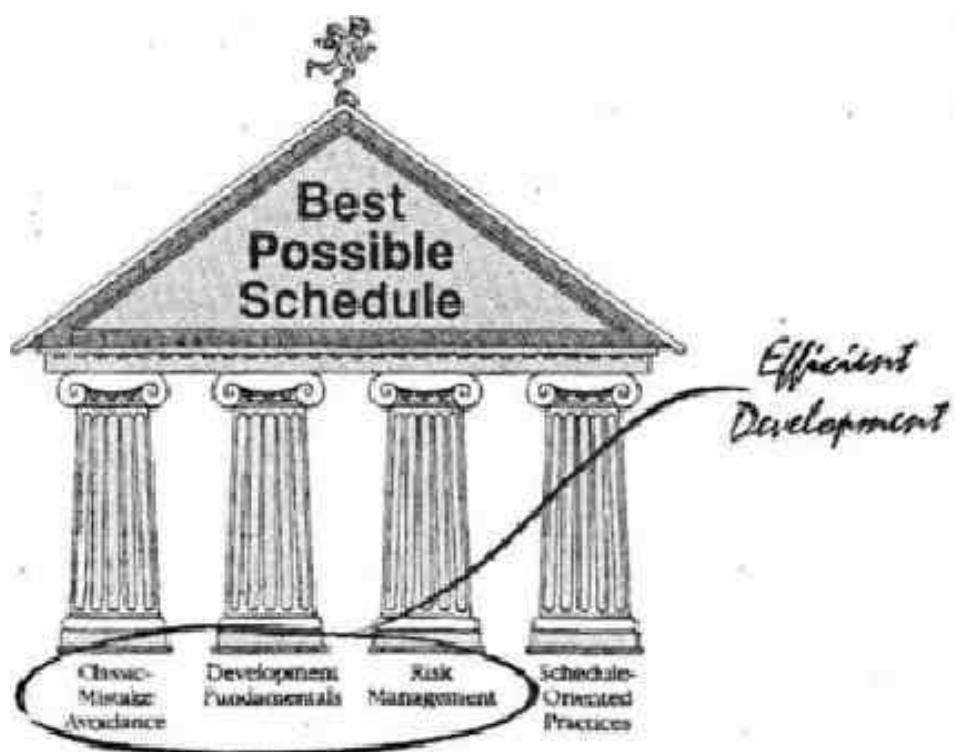


Figure 2-4. *Efficient development. The first three steps in achieving the best possible schedule make up "efficient development." Many project teams find that efficient development provides all the development speed they need.*

Can you achieve shorter schedules without first attaining efficient development? Maybe. You can choose effective, schedule-oriented practices and avoid slow or ineffective practices without focusing on efficient development per se. Until you attain efficient development, however, your chances of success in using schedule-oriented practices will be uncertain. If you choose specific schedule-oriented practices without a general strategy, you'll have a harder time improving your overall development capability. Of course, only you can know whether it's more important to improve your overall development capabilities or to try completing a specific project faster.

CROSS-REFERENCE
For more on the relationship between quality and development speed, see Section 4.3, "Quality-Assurance Fundamentals,"

Another reason to focus on efficient development is that for most organizations the paths to efficient development and shorter schedules are the same. For that matter, until you get to a certain point, the paths to shorter schedules, lower defects, and lower cost are all the same, too. As Figure 2-5 shows, once you get to efficient development the roads begin to diverge, but from where they are now, most development groups would benefit by setting a course for efficient development first.

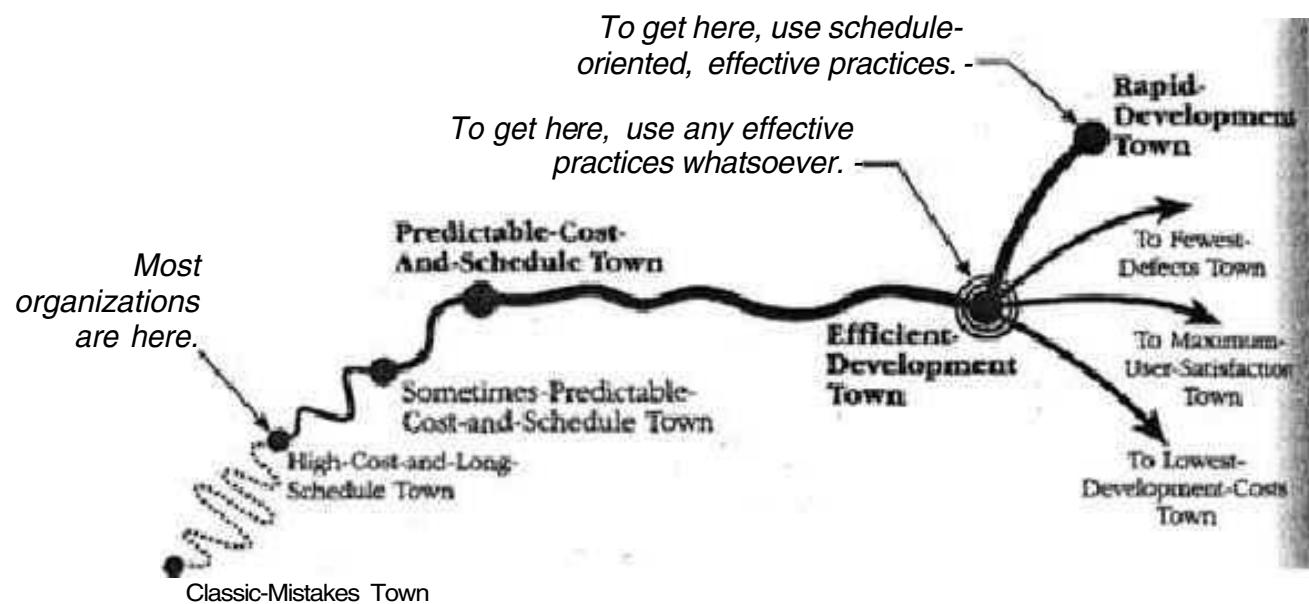


Figure 2-5. *The road to rapid development. From where most organizations are now, the route to rapid development follows the same road as the route to fewest defects, maximum user satisfaction, and lowest development costs. After you reach efficient development, the routes begin to diverge.*

Efficient Development Tilted Toward Best Schedule

CROSS-REFERENCE
For more on deciding between speed-oriented and schedule-risk-oriented practices, see Section 1.2, "Attaining Rapid Development merit," and Section 6.2, "What Kind of Rapid Development Do You Need?"

The third development approach listed in Table 2-1 is a variation of efficient development. If you are practicing efficient development and find that you still need better schedule performance, you can choose development practices that are tilted toward increasing development speed, reducing schedule risk, or improving progress visibility. You'll have to make small trade-offs in cost and product characteristics to gain that speed or predictability; if you start from a base of efficient development, however, you'll still be much better off than average.

All-Out Rapid Development

CROSS-REFERENCE
For more on nominal schedules, see "Nominal Schedules" in Section 8.6. For more on the costs of schedule compression, see "Two Facts of Life" in Section 8.6.

The final schedule-oriented development approach is what I call "all-out rapid development"—the combination of efficient and inefficient schedule-oriented practices. There comes a point when you're working as smart as you can and as hard as you can, and the only thing left to do at that point is to pay more, reduce the feature set, or reduce the product's polish.

Here's an example of what I mean by an "inefficient" practice: you can compress a project's nominal development schedule by 25 percent simply by adding more people to it. Because of increased communications and management overhead, however, you have to increase your team size by about 75 percent to achieve that 25-percent schedule reduction. The net effect of a shorter schedule and larger team size is a project that costs 33 percent more than the nominal project.

CROSS-REFERENCE

For more on whether you need all-out rapid development, see Section 6.2, "What Kind of Rapid Development Do You Need?"

The move to all-out rapid development is a big step and requires that you accept increased schedule risk or large trade-offs between cost and product characteristic—or both. Few projects welcome such trade-offs, and most projects are better off just choosing some form of efficient development.

2.4 Which Dimension Matters the Most?

CROSS-REFERENCE

For more on customizing software processes to the needs of specific projects, see Section 6.1, "Does One Size Fit All?"

Boeing, Microsoft, NASA, Raytheon, and other companies have all learned how to develop software in ways that meet their needs. At the strategy level, these different organizations have a lot in common. They have learned how to avoid classic mistakes. They apply development fundamentals. And they practice active risk management. At the tactical level, there is a world of difference in the ways that each of these successful organizations emphasize people, process, product, and technology.

Different projects have different needs, but the key in all cases is to accept the limitations on the dimensions you can't change and then to emphasize the other, dimensions to get the rest of the schedule benefit you need.

If you're developing a fuel-injection system for a car, you can't use 4GLs or a visual programming environment to develop the real-time, embedded software; you need greater performance and better low-level control than these tools can provide. You're prevented from exercising the technology dimension to the utmost. Instead, you have to emphasize technology as much as you can—and then get your real leverage from the people, process, and product dimensions.

If you're working on an in-house business program, perhaps you can use a 4GL, a visual programming environment, or a CASE tool. You're able to exercise technology to the utmost. But you might work for a stodgy corporation that prevents you from doing much in the people dimension. Emphasize people as much as the company allows, and then get the remaining leverage you need from the product and process dimensions.

If you're working in a feature-driven shrink-wrap market, you might not be able to shrink your feature set much to meet a tight schedule. Shrink it as much as you can, and then emphasize people, process, and technology to give you the rest of what you need to meet your schedule.

Further Reading

I know of no general books that discuss the topics of product or technology as they have been described in this chapter. This book discusses the topics further in Chapter 14, "Feature-Set Control," Chapter 15, "Productivity Tools," and Chapter 31, "Rapid-Development Languages."

The next three books provide general information on peopleware approaches to software development. The first is the classic.

DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1987.

Constantine, Larry L. *Constantine on Peopleware*. Englewood Cliffs, N.J.: Yourdon Press, 1995.

Plauger, P. J. *Programming on Purpose II: Essays on Software People*. Englewood Cliffs, N.J.: PTR Prentice Hall, 1993.

The following books provide information on software processes, the first at an organizational level, the second at a team level, and the third at an individual level.

Carnegie Mellon University/Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley, 1995. This book is a summary of the Software Engineering Institute's latest work on software process improvement. It fully describes the five-level process maturity model, benefits of attaining each level, and practices that characterize, each level. It also contains a detailed case study of an organization that has achieved the highest levels of maturity, quality, and productivity.

Maguire, Steve. *Debugging the Development Process*. Redmond, Wash.: Microsoft Press, 1994. Maguire's book presents a set of folksy maxims that project leads can use to keep their teams productive, and it provides interesting glimpses inside some of Microsoft's projects.

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, Mass.: Addison-Wesley, 1995. Humphrey lays out a personal software process that you can adopt at an individual level regardless of whether your organization supports process improvement.

3

ClassicMistakes

Contents

- 3.1 Case Study in Classic Mistakes
- 3.2 Effect of Mistakes on a Development Schedule
- 3.3 Classic Mistakes Enumerated
- 3.4 Escape from *Gilligan's Island*

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)

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, thinking that certain kinds of heroics can be beneficial (Bach 1995). But I think that emphasizing heroics in any form usually does 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."

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

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

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

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

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.

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.

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.

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

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.



HARD DATA

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."

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 (Pagan 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

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

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

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

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

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.

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.

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.

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.

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 negotiatingploy 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 die 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
<ol style="list-style-type: none"> 1. Undermined motivation 2. Weak personnel 3. Uncontrolled problem employees 4. Heroics 5. Adding people to a late project 6. Noisy, crowded offices 7. Friction between developers and customers 8. Unrealistic expectations 9. Lack of effective project sponsorship 10. Lack of stakeholder buy-in 11. Lack of user input 12. Politics placed over substance 13. Wishful thinking 	<ol style="list-style-type: none"> 14. Overly optimistic schedules 15. Insufficient risk management 16. Contractor failure 17. Insufficient planning 18. Abandonment of planning under pressure 19. Wasted time during the fuzzy front end 20. Shortchanged upstream activities 21. Inadequate design 22. Shortchanged quality assurance 23. Insufficient management controls 24. Premature or overly frequent convergence 25. Omitting necessary tasks from estimates 26. Planning to catch up later 27. Code-like-hell programming 	<ol style="list-style-type: none"> 28. Requirements gold-plating 29. Feature creep 30. Developer gold-plating 31. Push-me, pull-me negotiation 32. Research-oriented development 	<ol style="list-style-type: none"> 33. Silver-bullet syndrome 34. Overestimated savings from new tools or methods 35. Switching tools in the middle of a project 36. Lack of automated source-code control

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.

Software-Development Fundamentals

Contents

- 4.1 Management Fundamentals
- 4.2 Technical Fundamentals
- 4.3 Quality-Assurance Fundamentals
- 4.4 Following the Instructions

Related Topics

- Rapid-development strategy: Chapter 2
- Summary of inspections: Chapter 23

RED AUERBACH, THE LONG-TIME COACH of the Boston Celtics and until recently the winningest coach in the history of professional basketball, created a videotape called "Red on Roundball." Auerbach drives home the point that the key to success in professional basketball is fundamentals. He says at least 20 times that a pass is only a successful pass *if someone catches it.* The key to successful rebounding is *getting the ball*, Auerbach's roadmap to eight consecutive NBA championships relied on fundamentals.

In software, one path to success is paying attention to fundamentals. You might be the Bob Cousy, Kareem Abdul Jabbar, or Michael Jordan of your software organization. You might have a battery of schedule-oriented practices at your disposal. But if you don't put fundamental development practices at the heart of your development effort, you will seriously risk failing to meet your schedule goals.

People often tell you to use good software engineering practices because they're "right" or because they'll promote high quality. Their admonitions take on religious tones. But I don't think this is a religious issue. If the practices work—use them, if they don't—don't! My contention is that you should use the fundamental software-engineering practices described in this chapter not because they're, "right," but because they reduce cost and time to market.

Everybody wants to be on a championship team, but nobody wants to come to practice.

Bobby Knight

This position is less theoretical than you might think. In a review of 10 software projects that organizations had selected as their "best projects," Bill Hetzel concluded that "If there was one high-level finding that stood out, it is that best projects get to be best based on fundamentals. All of us know the fundamentals for good software—the difference is that most projects don't do them nearly so well and then get into trouble" (Hetzel 1993).

The best place to start looking for information on software-development fundamentals is a general software-engineering textbook. This book is not a software-engineering textbook, so this chapter confines itself to identifying the development fundamentals, explaining how they affect development schedules, quantifying how large their effect is (whenever possible), and providing pointers to more information.

The practices in this chapter are divided into management, technical, and quality-assurance practices. Some of the practices don't fit neatly into one category, so you may want to browse all the categories even if you're most interested in a particular one. But first, you might want to read Case Study 4-1 to put you in an appropriate frame of mind.

Case Study 4-1. Lack of Fundamentals

"We thought we had figured out what we were doing," Bill told Charles. "We did pretty well on version 3 of our Sales Bonus Program, SBP, which is the program we use to pay our field agents their commissions. But on version 4, everything fell apart." Bill had been the manager of SBP versions 1 through 3, and Charles was a consultant Giga-Safe had called in to help figure out why version 4 had been so problematic.

"What were the differences between versions 3 and 4?" Charles asked.

"We had problems with versions 1 and 2," Bill responded, "but by version 3 we felt that we had put our problems behind us. Development proceeded with hardly any problems at all. Our estimates were accurate, partly because we've learned to pad them with a 30-percent safety margin. The developers had almost no problems with forgotten tasks, tools, or design elements. Everything went great."

"So what happened on version 4?" Charles prompted.

"That was a different story. Version 3 was an evolutionary upgrade, but version 4 was a completely new product developed from scratch.

"The team members tried to apply the lessons they'd learned on SBP versions 1 through 3. But partway through the project, the schedule began to slip. Technical tasks turned out to be more complicated than anticipated. Tasks that the developers had estimated would take 2 days instead took 2 to 3 weeks. There were problems with some new development tools, and the team lost

(continued)

Case Study 4-1. Lack of Fundamentals, *continued*

ground fighting with them. The new team members didn't know all the team's rules, and they lost work and time because new team members kept overwriting each other's working files. In the end no one could predict when the product would be ready until the day it actually was ready. Version 4 was almost 100 percent late."

"That does sound pretty bad," Charles agreed. "You mentioned that you had had some problems with versions 1 and 2. Can you tell me about those projects?"

"Sure," Bill replied. "On version 1 of SBP, the project was complete chaos. Total-project estimates and task scheduling seemed almost random. Technical problems turned out to be harder than expected. Development tools that were supposed to save time actually added time to the schedule. The development team took one schedule slip after another, and no one knew when the product would be ready to release until a day or two before it actually was ready. In the end, the SBP team delivered the product about 100 percent over-schedule."

"That sounds a lot like what happened on version 4," Charles said.

"That's right," Bill shook his head. "I thought we had learned our lesson a long time ago."

"What about version 2?" Charles asked.

"On version 2, development proceeded more smoothly than on version 1. The project estimates and task schedules seemed more realistic, and the technical work seemed to be more under control. There were fewer problems with development tools, and the development team's work took about as long as they had estimated. They made up the estimation errors they did have through increased overtime.

"But toward the end of the project, the team discovered several tasks that they hadn't included in their original estimates. They also discovered fundamental design flaws, which meant they had to rework 10 to 15 percent of the system. They took one big schedule slip to include the forgotten tasks and the rework. They finished that work, found a few more problems, took another schedule slip, and finally delivered the product about 30 percent late. That's when we learned to add a 30-percent safety margin to our schedules."

"And then version 3 went smoothly?" Charles asked.

"Right," Bill agreed.

"I take it that versions 1 through 3 used the same code base?" Charles asked.

"Yes."

"Did versions 1 through 3 use the same team members?"

(continued)

Case Study 4-1 . Lack of Fundamentals, *continue*

"Yes, but several developers quit after version 3, so most of the version 4 team hadn't worked on the project before."

"Thanks," Charles said. "That's all helpful."

Charles spent the rest of the day talking with the development team and then met with Bill again that night. "What I've got to tell you might not be easy for you to hear," Charles said. "As a consultant, I see dozens of projects a year, and throughout my career I've seen hundreds of projects in more than a hundred organizations. The pattern you experienced with SBP versions 1 through 4 is actually fairly common.

"Earlier, you implied that the developers weren't using automated source-code control, and I confirmed that this afternoon in my talks with your developers. I also confirmed that the development team doesn't use design or code reviews. The organization relies on seat-of-the-pants estimates even though more effective estimation methods are available."

"OK," Bill said. "Those things are all true. But what do we need to do so that we never experience another project like version 4 again?"

"That's the part that's going to be hard for you to hear," Charles said. "There isn't any one thing you need to do. You need to improve on the software development fundamentals or you'll see this same pattern again and again. You need to strengthen your foundation. On the management side, you need more effective scheduling, planning, tracking, and measurement. On the technical side, you need more effective requirements management, design, construction, and configuration management. And you need much stronger quality assurance."

"But we did fine on version 3," Bill objected.

"That's right," Charles agreed. "You will do fine once in awhile—when you're working on a familiar product with team members who have worked on the same product before. Most of the version 3 team had also worked on versions 1 and 2. One of the reasons that organizations think they don't need to master software-development fundamentals is that they do have a few successes. They can get pretty good at estimating and planning for a specific product. They think they're doing well, and they don't think that anyone else is doing any better."

"But their development capability is built on a fragile foundation. They really only know how to develop one specific product in one specific way. When they are confronted with major changes in personnel, development tools, development environment, or product concept, that fragile development capability breaks down. Suddenly they find themselves back at square 1. That's what happened on SBP 4 when you had to rewrite the product from scratch with new developers. That's why your experiences on version 1 and version 4 were so similar."

(continued)

Case Study 4-1. Lack of Fundamentals, *continued*

"I hadn't thought about it that way before, but maybe you're right," Bill said quietly. "That sounds like a lot of work, though. I don't know if we can justify it."

"If you don't master the fundamentals, you'll do OK on the easy projects, but your hard projects will fall apart," Charles said, "and those are usually the ones you really care about."

4.1 Management Fundamentals

**FURTHER READING**

This chapter's description of development fundamentals is similar to what the Software Engineering Institute calls a "repeatable" process. For details, see *The Capability Maturity Model: Guidelines for Improving the Software Process* (Carnegie Mellon University/Software Engineering Institute, 1995).

Management fundamentals have at least as large an influence on development schedules as technical fundamentals do. The Software Engineering Institute has repeatedly observed that organizations that attempt to put software-engineering discipline in place before putting project-management discipline in place are doomed to fail (Burlton 1992). Management often controls all three corners of the classic trade-off triangle—schedule, cost, and product—although sometimes the marketing department controls the product specification and sometimes the development department controls the schedule. (Actually, development always controls the real schedule; sometimes development also controls the planned schedule.)

Management fundamentals consist of determining the size of the product (which includes functionality, complexity, and other product characteristics), allocating resources appropriate for a product of that size, creating a plan for applying the resources, and then monitoring and directing the resources to keep the project from heading into the weeds. In many cases, upper management delegates these management tasks to technical leads explicitly, and in other cases it simply leaves a vacuum that a motivated lead or developer can fill.

Estimation and Scheduling

CROSS-REFERENCE

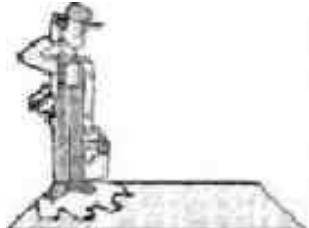
For more on estimation, see Chapter 8, "Estimation." For more on scheduling, see Chapter 9, "Scheduling."

Well-run projects go through three basic steps to create a software schedule. They first estimate the size of the project, then they estimate the effort needed to build a product of that size, and then they estimate a schedule based on the effort estimate.

Estimation and scheduling are development fundamentals because creating an inaccurate estimate reduces development efficiency. Accurate estimation is essential input for effective planning, which is essential for efficient development.

Planning

As Philip W. Metzger points out in his classic *Managing a Programming Project*, poor planning boils to the surface as a source of problems more often than any other problem (Metzger 1981). His list of software-development problems looks like this:



CLASSIC MISTAKE

- Poor planning
- Ill-defined contract
- Poor planning
- Unstable problem definition
- Poor planning
- Inexperienced management
- Poor planning
- Political pressures
- Poor planning
- Ineffective change control
- Poor planning
- Unrealistic deadlines
- Poor planning

In his review of best projects, Bill Hetzel found, that the industry's best projects are characterized by strong up-front planning to define tasks and schedules (Hetzel 1993). Planning a software project includes these activities:

CROSS-REFERENCE

For more on these topics, see
Chapter 12, "Teamwork";
Chapter 13, "Team Structure";
Chapter 7, "Lifecycle
Planning"; Chapter 5, "Risk
Management"; and Chapter 14,
"Feature-SetControl."

- Estimation and scheduling
- Determining how many people to have on the project team, what technical skills are needed, when to add people, and who the people will be
- Deciding how to organize the team
- Choosing which lifecycle model to use
- Managing risks
- Making strategic decisions such as how to control the product's feature set and whether to buy or build pieces of the product

Tracking

CROSS-REFERENCE

For details on one project-tracking practice, see Chapter 27, "Miniature Milestones."

Once you've planned a project, you track it to check that it's following the plan—that it's meeting its schedule, cost, and quality targets. Typical management-level tracking controls include task lists, status meetings, status reports, milestone reviews, budget reports, and management by walking around. Typical technical-level tracking controls include technical audits,

technical reviews, and quality gates that control whether you consider milestones to be complete.

Bill Hetzel found that strong measurement and tracking of project status was evident in every "best project." Status measurement to support project management appears as a natural by-product of the good planning work and is a critical success factor (Hetzel 1993).

As Figure 4-1 suggests, on a typical project, project management is almost a black-box function. You rarely know what's going on during the project, and you just have to take whatever comes out at the end. On an ideal project, you have 100 percent visibility at all times. On an efficient project, you always have at least some visibility and you have good visibility more often than not.

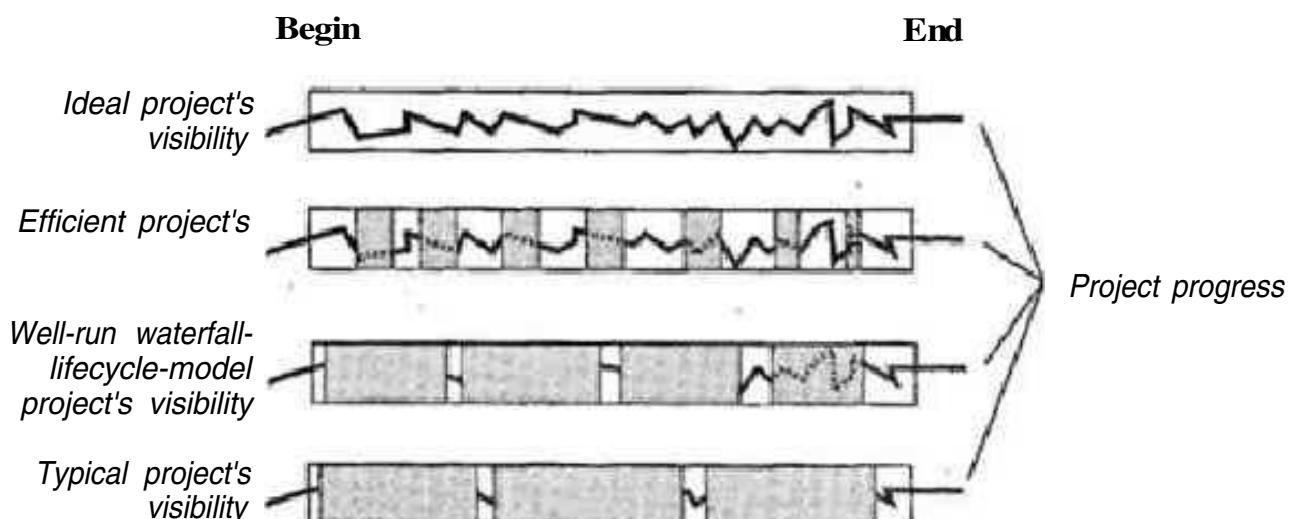


Figure 4-1. Progress visibility for different kinds of projects. Efficient development provides much better visibility than typical development.



Capers Jones reports that "software progress monitoring is so poor that several well-known software disasters were not anticipated until the very day of expected deployment" (Jones 1995b). After assessing 59 sites between 1987 and 1993, the Software Engineering Institute found that 75 percent of the sites needed to improve their project tracking and oversight (Kitson and Masters 1993). When organizations have been assessed, tried to improve, and then been reassessed, the biggest problems for the organizations that failed to improve lay in the project planning and tracking-and-oversight areas (Baumert 1995).

Tracking is a fundamental software management activity. If you don't track a project, you can't manage it. You have no way of knowing whether your plans are being carried out and no way of knowing what you should do next. You have no way of monitoring risks to your project. Effective tracking enables you to detect schedule problems early, while there's still time to do

something about them. If you don't track a project, you can't do rapid development.

Measurement

CROSS-REFERENCE

For more on measurement, see Chapter 26, "Measurement."

One key to long-term progress in a software organization is collecting metrics data to analyze software quality and productivity. Virtually all projects collect data on costs and schedules. But this limited data doesn't provide much insight into how to reduce the costs or shorten the schedules.

Collecting a little more data can go a long way. If, in addition to cost and schedule data, you collect historical data on how large your programs are in lines of code or some other measurement, you will have a basis for planning future projects that's better than gut instinct. When your boss says, "Can we develop this product in 9 months?" You can say, "Our organization has never developed a product of this size in less than 11 months, and the average time for such a product is 13 months."

You need to have a basic knowledge of software measurement to develop efficiently. You need to understand the issues involved in collecting metrics, including how much or how little data to collect and how to collect it. You should have a knowledge of specific metrics you can use to analyze status, quality, and productivity. An organization that wants to develop rapidly needs to collect basic metrics in order to know what its development speed is and whether it's improving or degrading over time.

Further Reading on Management Fundamentals

The first four volumes listed below discuss far-ranging software topics, including pragmatic issues such as what to do with a problem team member, theoretical issues such as how to model a software project as a system, and esoteric issues such as the importance of observation to software development. Weinberg's writing is entertaining and full of insights.

Weinberg, Gerald M. *Quality Software Management, Vol. 1: Systems Thinking*. New York: Dorset House, 1992.

Weinberg, Gerald M. *Quality Software Management, Vol. 2: First-Order Measurement*. New York: Dorset House, 1993.

Weinberg, Gerald M. *Quality Software Management, Vol. 3: Congruent Action*. New York: Dorset House, 1994.

Weinberg, Gerald M. *Quality Software Management, Vol. 4: Anticipating Change*. New York: Dorset House, 1996.

Pressman, Roger S. *A Manager's Guide to Software Engineering*. New York: McGraw-Hill, 1993. This might be the best overview available on general aspects of software project management. It includes introductory sections on estimation, risk analysis, scheduling and tracking, and the human element. Its only drawback is its use of a question-and-answer format that might come across as disjointed to some readers. (It does to me.)

Carnegie Mellon University/Software Engineering Institute. *Tfye Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley, 1995. This book describes a management-level framework for understanding, managing, and improving software development:

Thayer, Richard H., ed. *Tutorial: Software Engineering Project Management*. Los Alamitos, Calif.: IEEE Computer Society Press, 1990. This is a collection of about 45 papers on the topic of managing software projects. The papers are some of the best discussions available on the topics of planning, organizing, staffing, directing, and controlling a software project. Thayer provides an introduction to the topics and comments briefly on each paper.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Gilb's thesis is that project managers generally do not want to predict what will happen on their projects; they want to control it. Gilb's focus is on development practices that contribute to controlling software schedules, and several of the practices he describes in his book have been included as best practices in this book.

DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982. Although now in its second decade, DeMarco's book doesn't seem the least bit dated. He deals with problems that are the same today as they were in 1982—managers who want it all and customers who want it all *now*. He lays out project-management strategies, with a heavy emphasis on measurement.

Metzger, Philip W. *Managing a Programming Project, 2d Ed.* Englewood Cliffs, N.J.: Prentice Hall, 1981. This little book is the classic introductory project-management textbook. It's fairly dated now because of its emphasis on the waterfall lifecycle model and on document-driven development practices. But anyone who's willing to read it critically will find that Metzger still has some important things to say about today's projects and says them well.

The following book is not specifically about software projects, but it is applicable nonetheless.

Grove, Andrew S. *High Output Management*. New York: Random House, 1983. Andy Grove is one of the founders of Intel Corporation and has strong opinions about how to manage a company in a competitive technical industry. Grove takes a strongly quantitative approach to management.

4.2 Technical Fundamentals

A 1984 study of "modern programming practices"—technical fundamentals—found that you can't achieve high productivity without using them. Figure 4-2 illustrates the study's results.

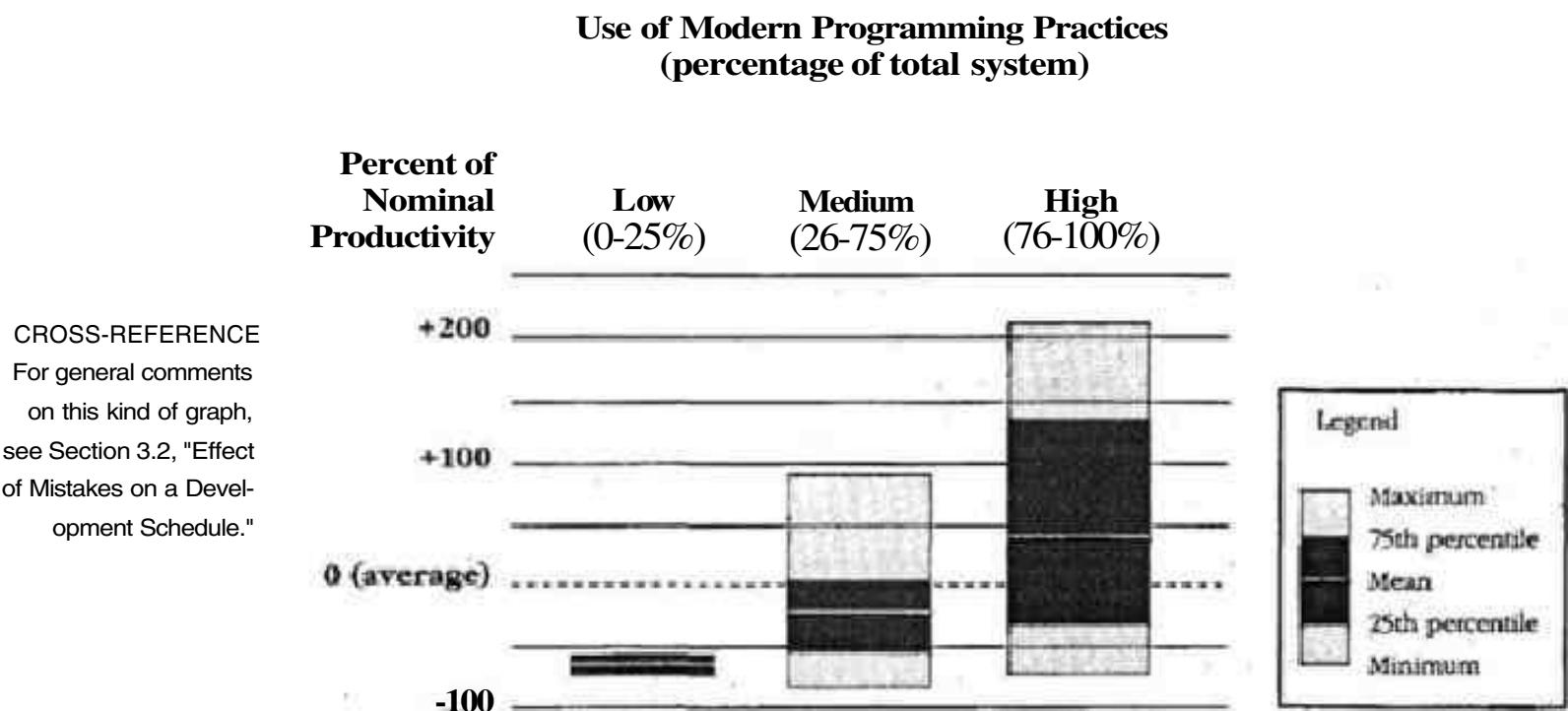


Figure 4-2. Findings for "Use of Modern Programming Practices" factor (Vosburgh et al. 1984). You can't achieve top productivity without making extensive use of "modern programming practices"—what this chapter calls "technical fundamentals."

This is the same chart that I presented in the "Classic Mistakes" chapter, and it provides a specific example of the general classic mistakes lesson. Application of technical fundamentals, by itself, is hot enough to create high productivity. Some projects used modern programming practices a great deal and still had productivity about as low as other projects that didn't use them at all. Thus attention to development fundamentals is necessary but not sufficient for achieving rapid development.

Larry Constantine tells a story about the Australian Computer Society Software Challenge (Constantine 1995b). The challenge called for three-person teams to develop and deliver a 200 function-point application in 6 hours.

The team from Ernst and Young decided to follow a formal development methodology—a scaled-down version of their regular methodology—complete with staged activities and intermediate deliverables. Their approach included careful analysis and design—part of what this chapter describes as technical fundamentals. Many of their competitors dived straight into coding, and for the first few hours, the team from Ernst and Young lagged behind.

By midday, however, the Ernst and Young team had developed a commanding lead. At the end of the day, the team from Ernst and Young lost, but not because of their formal methodology. They lost because they accidentally overwrote some of their working files, delivering less functionality at the end of the day than they had demonstrated at lunchtime. Ironically, what would have saved their bacon was not less formality, but more—namely, formal configuration management including periodic backups. They got bitten by the classic mistake of not using effective source-code control.

The moral of this story seems clear enough, but some skeptics, including me, were left wondering: Would the team from Ernst and Young really have won without the configuration-management snafu? The answer is "yes." They reappeared a few months later at another rapid-development face-off—this time with version control and backup—and they won (Constantine 1996).

In this case, formal methodologies paid off within a single day. If attention to technical fundamentals can make this much difference in that amount of time, imagine how much of a difference they can make over a 6- to 12-month project.

Requirements Management

CROSS-REFERENCE

For more on traditional requirements-management practices, see Chapter 14, "Feature-Set Control."

Requirements management is the process of gathering requirements; recording them in a document, email, user-interface storyboard, executable prototype, or some other form; tracking the design and code against them; and then managing changes to them for the rest of the project.



HARD DATA

It's not uncommon for developers to complain about the problems associated with traditional requirements-management practices, the most common being that they are too rigid. Some practices can be overly rigid, but the alternative is often worse. A survey of more than 8000 projects found that the top three reasons that projects were delivered late, over budget, and with less functionality than desired all had to do with requirements-management practices: lack of user input, incomplete requirements, and changing requirements (Staridish Group 1994). A survey of projects by the Software Engineering

CROSS-REFERENCE
For more on controlling feature creep, see Section 14.2, "Mid-Project: Feature-Creep Control."

Institute reached essentially the same conclusion: more than half the projects surveyed suffered from inadequate requirements management (Kitson and Masters 1993).

Success at requirements management depends on knowing enough different practices to be able to choose the ones that are appropriate for a specific project. Here are the fundamentals of requirements management:

- Requirements-analysis methodologies including structured analysis, data structured analysis, and object-oriented analysis
- System-modeling practices such as class diagrams, dataflow diagrams, entity-relationship diagrams, data-dictionary notation, and state-transition diagrams
- Communication practices such as Joint Application Development (JAD), user-interface prototyping, and general interview practices
- The relationships between requirements management and the different lifecycle models including evolutionary prototyping, staged releases, spiral, waterfall, and code-and-fix

CROSS-REFERENCE
For more on speeding up requirements gathering, see "Requirements specification" in Section 6.5.



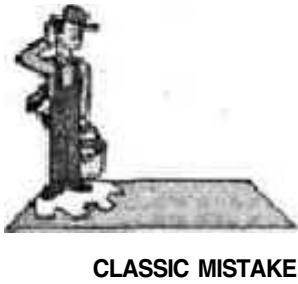
Requirements management provides great development-speed leverage in two ways. First, requirements gathering tends to be done at a leisurely pace compared with other software-development activities. If you can pick up the pace without hurting quality, you can shorten overall development time.

Second, getting a requirement right in the first place typically costs 50 to 200 times less than waiting until construction or maintenance to get it right (Boehm and Papaccio 1988). The typical project experiences a 25-percent change in requirements. Some fundamental requirements-management practices allow you to reduce the number of requirements changes. Other fundamental development practices allow you to reduce the cost of each requirements change. Imagine what the combined effect would be if you could reduce the number of changes from 25 percent to 10 percent and simultaneously reduce the cost of each change by a factor of 5 or 10. Rapid development would be within your grasp.

Design



Just as it makes sense to create a set of blueprints before you begin building a house, it makes sense to create an architecture and design before you begin building a software system. A design error left undetected until system testing typically takes 10 times as long to fix as it would if it were detected at design time (Dunn 1984).



CLASSIC MISTAKE

Doesn't everyone already do good design? No. My impression is that good design receives more lip service than any other activity in software development and that few developers really do design at all. A design architect who works for Microsoft said that in 6 years of interviewing more than 200 candidates for software-development positions, he had interviewed only 5 who could accurately describe the concepts of "modularity" and "information hiding" (Kohen 1995).

The ideas of modularity and information hiding are design fundamentals. They are part of the foundation of both structured design and object design. A developer who can't discuss modularity and information hiding is like a basketball player who can't dribble. "When you consider that Microsoft screens its candidates rigorously before they are even interviewed, you come to the somewhat frightening conclusion that the situation throughout most of the software-development world is considerably worse than 195 out of 200 developers who have major gaps in their knowledge of design fundamentals.

Here are the fundamental topics in architecture and design:

- Major design styles (such as object design, structured design, and data-structure design)
- Foundational design concepts (such as information hiding, modularity, abstraction, encapsulation, cohesion, coupling, hierarchy, inheritance, polymorphism, basic algorithms, and basic data structures)
- Standard design approaches to typically challenging areas (including exception handling, internationalization and localization, portability, string storage, input/output, memory management, data storage, floating-point arithmetic, database design, performance, and reuse)
- Design considerations unique to the application domain you're working in (financial applications, scientific applications, embedded systems, real-time systems, safety-critical software, or something else)
- Architectural schemes (such as subsystem organization, layering, subsystem communication styles, and typical system architectures)
- Use of design tools

CROSS-REFERENCE

For details on a kind of design well suited to rapid-development projects, see Chapter 19, "Designing for Change."

It is possible to develop a system without designing it first. Major systems have been implemented through sheer ^pding and debugging prowess, high enthusiasm, and massive overtime—and without systematic design. However, design serves as the foundation for construction, project scheduling, project tracking, and project control, and as such effective design is essential to achieving maximum development speed.

By the time you get to construction, most of the groundwork for your project's success or failure has already been laid. Both requirements management and design offer greater leverage on your development schedule than construction does. In those activities, small changes can make a big difference in your schedule.

Construction might not offer many opportunities for large reductions in schedule, but construction work is so detailed and labor intensive that it's important to do a good job of it. If your code quality isn't good to start with, it's nearly impossible to go back and make it better. It certainly isn't time-effective to do it twice.

Although construction is a low-level activity, it does present many occasions to use time inefficiently or to become sidetracked on noncritical but time-consuming tasks. You can, for example, waste time gold-plating functions that do not need to be gold-plated, debugging needlessly sloppy code, or performance-tuning small sections of the system before you know whether they need to be tuned.

Poor design practices can force you to rewrite major parts of your system; poor construction practices won't force you to do that. Poor construction practices can, however, introduce subtle errors that take days or weeks to find and fix. It can sometimes take as long to find an off-by-one array declaration error as it can to redesign and reimplement a poorly designed module. The total work you have to show for your debugging is a "+1" rather than several pages of new code, but the schedule penalty is just as real.

Construction fundamentals include the following topics:

- Coding practices (including variable and function naming, layout, and documentation)
- Data-related concepts (including scope, persistence, and binding time)
- Guidelines for using specific types of data (including numbers in general, integers, floating-point numbers, characters, strings, Booleans, enumerated types, named constants, arrays, and pointers)
- Control-related concepts (including organizing straight-line code, using conditionals, controlling loops, using Boolean expressions, controlling complexity, and using unusual control structures such as *goto*, *return*, and recursive procedures)
- Assertions and other code-centered error-detection practices
- Rules for packaging code into routines, modules, classes, and files
- Unit-testing and debugging practices

- Integration strategies (such as incremental integration, big-bang integration, and evolutionary development)
- Code-tuning strategies and practices
- The ins and outs of the particular programming language you're using
- Use of construction tools (including programming environments, groupwork support such as email and source-code control, code libraries, and code generators)

Adherence to some of these fundamentals takes time, but it saves time over the life of a project. Toward the end of a project, a product manager told a friend of mine, "You're slower than some of the other programmers on the team, but you're more careful. There's a place for that on this team because we have a lot of modules that have too many bugs and will need to be re-written." That statement reveals a person who doesn't yet understand what makes software projects take as long as they do.

When all is said and done, paying attention to construction fundamentals is as much a risk-management practice as a time-savings practice. Good construction practices prevent the creation of a rat's nest of indecipherable code that causes your project to grind to a halt when a key person gets sick, when a critical bug is discovered, or when a simple change needs to be made. Such practices improve the predictability and control you have over your project and increase the chance of delivering on time.

Software Configuration Management

Software configuration management (SCM) is the practice of managing project artifacts so that the project stays in a consistent state over time. SCM includes practices for evaluating proposed changes, tracking changes, handling multiple versions, and keeping copies of project artifacts as they existed at various times. The project artifact managed the most often is source code, but you can apply SCM to requirements, plans, designs, test cases, problem reports, user documentation, data, and any other work you use to build your product. I even used SCM in writing this book because not using it on my last book caused too many problems.

Most software-development books treat SCM as a quality-assurance practice—and it does have a strong effect on quality. But treating it as a QA practice could imply that it has either a neutral or negative effect on the development schedule. SCM is sometimes implemented in a way that hurts project efficiency, but it is critical if you want to achieve maximum development speed. Without configuration management, your teammates can change part of the design and forget to tell you. You can then implement code that's incompatible with the design changes, which eventually will require either you or your teammates to redo your work.



Lack of automated source-code control is a common and irksome inefficiency. Of sites surveyed between 1987 and 1993, the Software Engineering Institute found that more than 50 percent needed to improve their software configuration management (Kitson and Masters 1993). On small projects, lack of configuration management adds a few percentage points to the overall project cost. On large projects, configuration management is a critical-path item (Jones 1994).

Further Reading on Development Fundamentals

Many training organizations offer workshops on requirements analysis and design. Workshops on construction and configuration management may be more difficult to find. The most readily available source of information on any of the topics will probably be books, so I've listed the best books on each topic here.

Requirements management

Yourdon, Edward. *Modern Structured Analysis*, New York: Yourdon Press, 1989. Yourdon's book contains a survey of requirements specification and analysis circa 1989 including modeling tools, the requirements-gathering process, and related issues. Note that one of the most useful sections is hidden in an appendix: "Interviewing and Data Gathering Techniques."

Hatley, Derek J., and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. New York: Dorset House Publishing, 1988. Hatley and Pirbhai emphasize real-time systems and extend the graphical notation used by Yourdon to real-time environments.

Gause, Donald C., and Gerald Weinberg. *Exploring Requirements: Quality Before Design*. New York: Dorset House, 1989. Gause and Weinberg chart an untraditional course through the requirements-management terrain. They discuss ambiguity, meetings, conflict resolution, constraints, expectations, reasons that methodologies aren't enough, and quite a few other topics. They mostly avoid the topics that other requirements books include and include the topics that the other books leave out.

Design

Plauger, P. J. *Programming on Purpose: Essays on Software Design*. Englewood Cliffs, N.J.: PTR Prentice Hall, 1993- This is a refreshing collection of essays that were originally published in *Computer Language* magazine. Plauger is a master designer and takes up a variety of topics having as much to do with being a designer as with design in the abstract. What makes the essays refreshing is that Plauger ranges freely

over the entire landscape of design topics rather than restricting himself to a discussion of any one design style. The result is uniquely insightful and thought provoking.

McConnell, Steve. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993. This book contains several sections about design, particularly design as it relates to construction. Like the Plauger book, it describes several design styles.

Yourdon, Edward, and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs, N.J.: Yourdon Press, 1979. This is the classic text on structured design by one of the co-authors (Constantine) of the original paper on structured design. The book is written with obvious care. It contains full discussions of coupling, cohesion, graphical notations, and other relevant concepts. Some people have characterized the book as "technically difficult," but it's hard to beat learning about a practice from its original inventor.

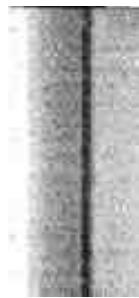
Page-Jones, Meilir. *The Practical Guide to Structured Systems Design, 2d Ed.* Englewood Cliffs, N.J.: Yourdon Press, 1988. This is a popular textbook presentation of the same basic structured-design content as Yourdon and Constantine's book and is written with considerable enthusiasm. Some people have found Page-Jones's book to be more accessible than Yourdon and Constantine's.

Booch, Grady. *Object Oriented Analysis and Design: With Applications, 2d Ed.* Redwood City, Calif.: Benjamin/Cummings, 1994. Booch's book discusses the theoretical and practical foundations of object-oriented design for about 300 pages and then has 175 more pages of object-oriented application development in C++. No one has been a more active advocate of object-oriented design than Grady Booch, and this is the definitive volume on the topic.

Goad, Peter, and Edward Yourdon. *Object-Oriented Design*. Englewood Cliffs, N.J.: Yourdon Press, 1991. This is a slimmer alternative to Booch's book, and some readers might find it to be an easier introduction to object-oriented design.

Construction

McConnell, Steve. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993. This is the only book I know of that contains thorough discussions of all the key construction issues identified in the "Construction" section. It contains useful checklists on many aspects of construction as well as hard data on the most effective construction practices. The book contains several hundred coding examples in C, Pascal, Basic, Fortran, and Ada.



Marcotty, Michael. *Software Implementation*. New York; Prentice Hall, 1991.

Marcotty discusses the general issues involved in constructing software by focusing on abstraction, complexity, readability, and correctness. The first part of the book discusses the history of programming, programming subculture, programming teams, and how typical programmers spend their time. The book is written with wit and style, and the first 100 pages on the "business of programming" are especially well done.

The two Bentley books below discuss programming energetically, and they clearly articulate the reasons that some of us find programming so interesting. The fact that the information isn't comprehensive or rigidly organized doesn't prevent the books from conveying powerful insights that you'll read in a few minutes and use for years.

Bentley, Jon. *Programming Pearls*. Reading, Mass.: Addison-Wesley, 1986.

Bentley, Jon. *More Programming Pearls: Confessions of a Coder*. Reading, Mass.: Addison-Wesley, 1988.

Maguire, Steve. *Writing Solid Code*. Redmond, Wash.: Microsoft Press, 1993.

This book describes key software-construction practices used at Microsoft. It explains how to minimize defects by using compiler warnings, protecting your code with assertion statements, fortifying subsystems with integrity checks, designing unambiguous function interfaces, checking code in a debugger, and avoiding risky programming practices.

Software configuration management (SCM)

These Bersoff and Babich books thoroughly cover the SCM topic.

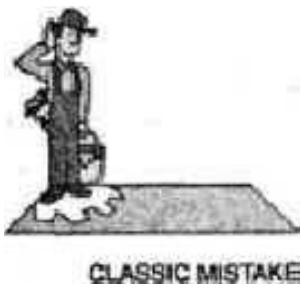
Bersoff, Edward H. et al. *Software Configuration Management*. Englewood Cliffs, N.J.: Prentice Hall, 1980.

Babich, W. *Software Configuration Management*. Reading, Mass.: Addison-Wesley, 1986.

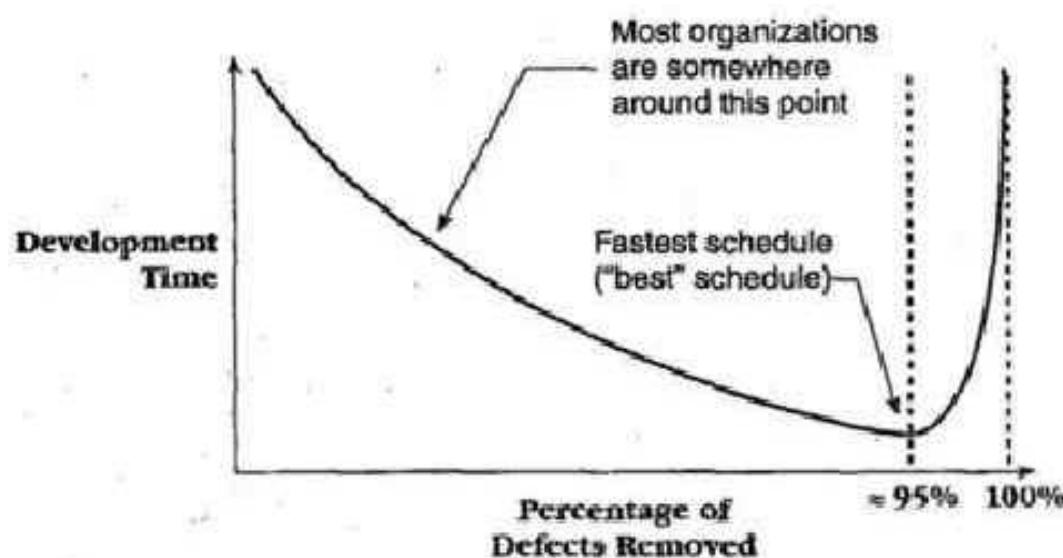
Bersoff, Edward H., and Alan M. Davis. "Impacts of Life Cycle Models on Software Configuration Management," *Communications of the ACM* 34, no. 8 (August 1991): 104-118. This article describes how SCM is affected by newer approaches to software development, especially by prototyping approaches.

4.3 Quality-Assurance Fundamentals

Like management and technical fundamentals, quality-assurance fundamentals provide critical support for maximum development speed. When a software product has too many defects, developers spend more time fixing the software than they spend writing it. Most organizations have found that they are better off not installing the defects in the first place. The key to not installing defects is to pay attention to quality-assurance fundamentals from Day 1 on.



Some projects try to save time by reducing the time spent on quality-assurance practices such as design and code reviews. Other projects—running late—try to make up for lost time by compressing the testing schedule, which is vulnerable to reduction because it's usually the critical-path item at the end of the project. These are some of the worst decisions a person who wants to maximize development speed can make because higher quality (in the form of lower defect rates) and reduced development time go hand in hand. Figure 4-3 illustrates the relationship between defect rate and development time.



Source: Derived from data in *Applied Software Measurement* (Jones 1991).

Figure 4-3. Relationship between defect rate and development time. In most cases, the projects that achieve the lowest defect rates also achieve the shortest schedules.

A few organizations have achieved extremely low defect rates (shown on the far right of the curve in Figure 4-3), at which point, further reducing the number of defects will increase the amount of development time. It's worth the extra time when it's applied to life-critical systems such as the life-support systems on the Space Shuttle—but not when it applies to non-life-critical software development.

On a rapid-development project, the project manager and the project manager's boss should review the Top-10 list once a week. The most useful aspects of the Top-10 list are that it forces you to look at risks regularly, to think about them regularly, and to be alert to changes in importance.

Interim Postmortems

Although the Top-10 list is probably the most useful risk-monitoring practice, a fast-track project should also include postmortems conducted throughout the project. Many project managers wait until the end to do a postmortem. That produces a nice benefit for the next project, but it doesn't help you when you really need it—on your current project! For maximum effectiveness, conduct a small-scale postmortem after you complete each major milestone.

Risk Officer

Some organizations have found appointing a risk officer to be useful. The job of the risk officer is to be alert to risks to the project and to keep managers and developers from ignoring them in their planning. As with testing and peer reviews, it turns out that for psychological reasons it's beneficial to have a person whose job it is to play devil's advocate—to look for all of the reasons that the project might fail. On large projects (50 persons or more), the job of risk officer might be full time. On smaller projects, you can assign someone with other project duties to play the role when needed. For the psychological reasons mentioned, the person assigned shouldn't be the project manager.

5.6 Risk, High Risk, and Gambling

It is not only the magnitude of the risk that we need to be able to appraise in entrepreneurial decisions.

It is above all the character of the risk. Is it, for instance, the kind of risk we can afford to take, or the kind of risk we cannot afford to take? Or is it that rare but singularly important risk, the risk we cannot afford not to take—sometimes regardless of the odds?

Peter Drucker

For purposes of rapid development, some projects are risks, some are high risks, and some are gambles. It's hard to find a software project that doesn't involve some risk, and projects that are merely "risks" are the kind best-suited to achieving maximum development speed. They allow you to move in an efficient, straight line from the beginning of the project to the end. Fast development, while not necessarily easy to achieve, is well within the grasp of the person who understands the strategies and practices described in this book.

High risk and rapid development make a less compatible combination. Risks tend to extend development schedules, and high risks tend to extend them a lot. But business realities sometimes require you to commit to an ambitious development schedule even when a project involves many risks—vague requirements, untrained personnel, unfamiliar product areas, strong research elements, or all of the above.

If you find yourself forced to commit to an ambitious schedule in such circumstances, be aware of the nature of the commitment. With two or three high-risk areas, even your best schedule projections will be nearly meaningless. Be careful to explain to the people who depend on you that you are willing to take calculated risks, even high risks, but more likely than not you won't be able to deliver what everyone is hoping for. In such cases, not just active but vigorous risk management will help you to make the best of a difficult situation.

CROSS-REFERENCE

For more on projects that have their schedules, feature sets, and resources dictated to them, see Section 6.6, "Development-Speed Trade-Offs." For more on code and fix, see Section 7.2, "Code-and-Fix,"

"Development-Speed Trade-Offs." For more on code and fix, see Section 7.2, "Code-and-Fix,"

At the extreme end of the risk scale, some projects are scheduled so aggressively that they become out-and-out gambles—they are more like the purchase of a lottery ticket than a calculated business decision. About one-third of all projects have an impossible combination of schedules, feature sets, and resources dictated to them before they start. In such circumstances, there is no incentive to practice risk management because the project starts out with a 100-percent chance of failure. With no chance of meeting their schedules through the use of any known development practices, it becomes rational to gamble on 1000-to-1 long shots such as code-and-fix development. These projects, which know they need maximum development speed, ironically become the projects that are most likely to throw effective, proven speed-oriented practices out the window.

The results are inevitable. The long shot doesn't pay off, and the project is delivered late—much later than it would have been if the project had been set up to take calculated risks rather than desperate gambles.

Do one-third of the projects in the industry really need to take desperate gambles? Do one-third of the projects have business cases for taking 1000-to-1 long shots? I don't think so. Beware of the risk level on your project, and try to keep it in the "risk" or "high-risk" category.



CLASSIC MISTAKE

Case Study 5-2. Systematic Risk Management

Square-Calc version 3.0 had been a disaster, overrunning its schedule by 50 percent. Eddie agreed to take over the project with version 3.5, and he hoped to do better than the last manager had done.

"As you know, Square-Calc 3.0 didn't do very well compared to its planned schedule," he told the team at the first planning meeting. "It was scheduled to take 10 months, and it took 15. We need to do better. We've got 4 months to complete a medium-sized upgrade, and I think that's a reasonable amount of time for this work. Risk management is going to be a top priority. The first thing I want to do is appoint a risk officer, someone who will look for all of the things that might go wrong on this project. Is anyone interested?"

(continued)

Case Study 5-2. Systematic Risk Management, *continued*

Jill had worked a lot harder than she wanted to on the last project and was willing to help prevent that from happening again so she said, "Sure, I'm interested. What do I need to do?"

"The first thing you need to do is create a list of known risks," replied Eddie. "I want you to get together with each of the developers this morning and find out what risks they're aware of, and then I want to get together with you this afternoon. We'll look at the risks and go from there."

That afternoon, Eddie and Jill met in Eddie's office. "Everyone, including me, thinks that the biggest risk is the Square-Calc 3.0 code base. It really sucks," Jill said. "None of us wants to make any major changes to it, and there are certain modules we don't even want to touch."

"The next most important risk is the user-documentation schedule. We shipped late last time partly because we didn't coordinate well enough with user documentation. We've got to make sure that doesn't happen again."

"The last big risk is creeping requirements. There were a lot of features that didn't make the cutoff for version 3.5, and I'm afraid that marketing will try to shoehorn them in." Jill went on to note a dozen smaller risks, but the first three were the big ones.

"OK, I want us to come up with a risk-management plan for the major risks," Eddie said. He explained the idea of the Top-10 Risks list to her and said that he wanted to review the top 10 risks with the team every week.

To manage the code-base risk, they decided to analyze their bug database to see whether any modules in the system were truly error prone. They allocated 1 month of their 4-month schedule to focus on rewriting the most error-prone modules.

For the user-documentation risk, they decided to develop a throwaway user-interface prototype that exactly matched the appearance of the new code they'd be writing. They would not allow visual deviations from the prototype. They would also include a rep from user documentation in their weekly risk-management meetings, which would help them to stay in sync with the documentation effort.

For the feature-creep risk, Eddie promised to talk with someone in the marketing department. "I know their number-one goal is to get the product out on time," he said. "We need to restore customer confidence after the schedule problems on 3.0. I'll explain the importance of setting crystal-clear goals to them, and I think that will cut down on the feature requests." They also

(continued)

Case Study 5-2. Systematic Risk Management, *continued*

invited Carlos from marketing to attend their risk meetings, reasoning that if someone from marketing understood all the other risks they faced, marketing might be less inclined to pile on new risks themselves.

During the next 4 weeks, identification and replacement of the error-prone modules went about as planned. It turned out that about 5 percent of the modules accounted for about 50 percent of the errors. They were able to redesign and reimplementation those at a careful pace. They subjected each module to a thorough review at each stage, and by the time they were done, they felt comfortable that the code base could sustain the rest of the modifications they needed to make for version 3.5.

At the weekly risk meeting 6 weeks into the schedule, Jill raised a new issue. "As you know, I've been monitoring some lower-priority risks in addition to the bigger ones, and one of them has become more important. Bob has been working on speeding up some of the scientific functions, and he told me a few weeks ago that he wasn't sure he could meet the revised spec. Apparently he researched the best available algorithms, implemented them, and that only made the functions about 50 percent faster. The spec calls for a 100-percent speedup, so Bob's been trying to come up with faster algorithms. I told him that I wanted to set a red-alert point in the schedule so that if he wasn't done at that point I could raise a warning flag. Yesterday we hit the red-alert point, and Bob says he is nowhere near being done. Basically, I think he's doing software research, and there's no way to predict how long he's going to take."

Carlos from marketing spoke up. "I was one of the people who pushed for the speed improvement, and I think that '100 percent' number is flexible. It's more important to me to get the product out on time than it is to meet that performance requirement to the letter. At least we'll be able to show our customers that we're responsive."

"That sounds good," Eddie said. "I think the 50-percent improvement we've already reached is good enough, so I'll redirect Bob to some other tasks. That's one less risk to worry about."

After that, there were few surprises. Minor issues arose and were addressed while they were still minor. Compared to the last project, this one seemed a little boring, but no one minded. A few marketing people tried to add features, but Carlos understood the importance of the schedule goal, and he fended off most of the requests before development even heard about them. The team performed well, and they delivered Square-Calc 3.5 by the 4-month due date.

Further Reading

Boehm, Barry W., ed. *Software Risk Management*. Washington, DC: IEEE Computer Society Press, 1989. This collection of papers is based on the premise that successful project managers are good risk managers. Boehm has collected a nicely balanced set of papers from both technical and business sources. One of the best features of the book is that Boehm contributed about 70 pages of original writing himself. That 70 pages is a good introduction to software risk management. You might think that a tutorial published in 1989 would seem dated by now, but it actually presents a more forward-looking view of software project management than you'll find most other places.

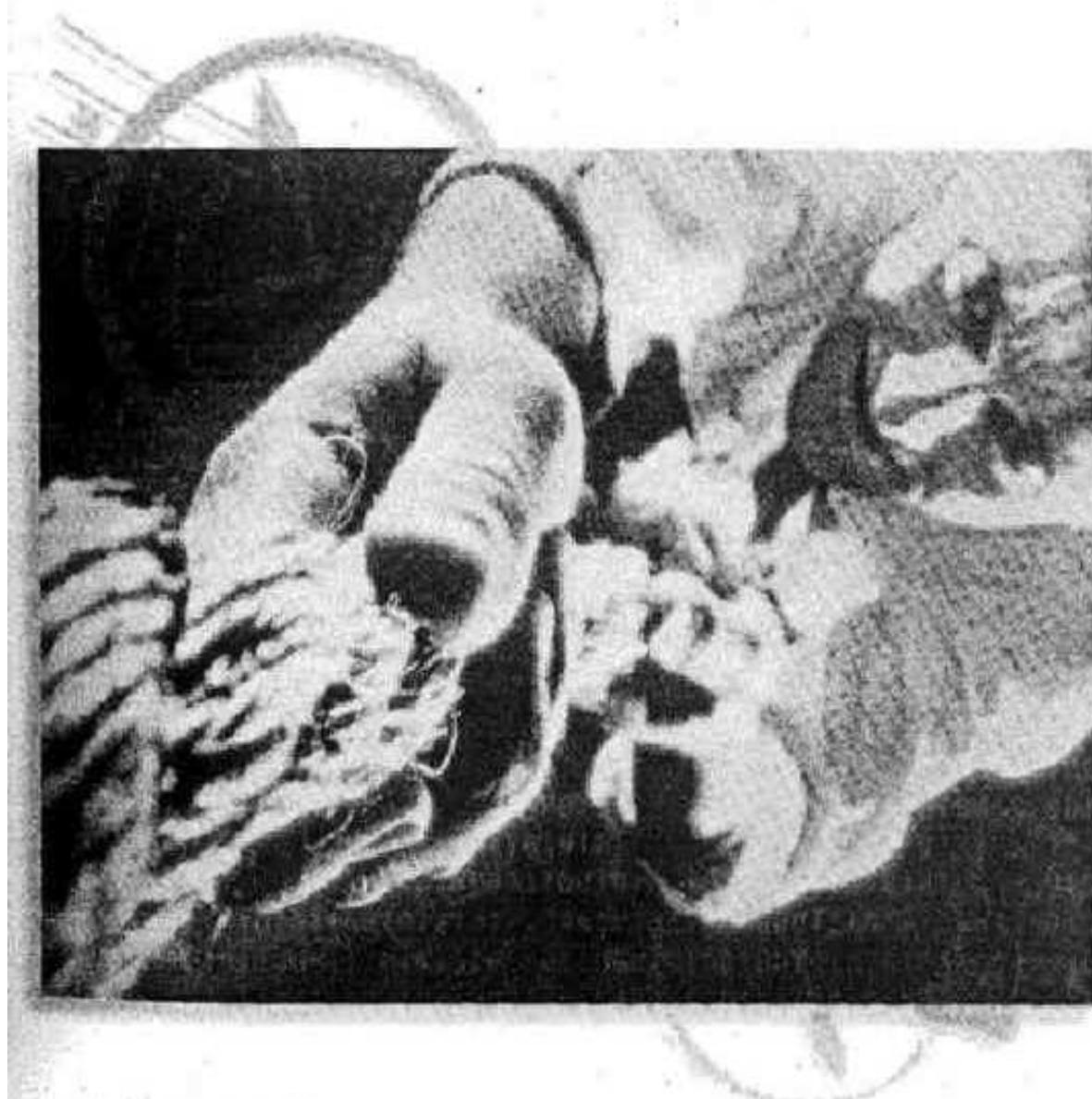
Boehm, Barry W. "Software Risk Management: Principles and Practices." *IEEE Software*, January 1991, pp. 32-41. This article hits the high points of the comments Boehm wrote for *Software Risk Management* and contains many practical suggestions.

Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Yourdon Press, 1994. Jones's book is an excellent complement to Boehm's risk-management tutorial. It says almost nothing about how to manage software risks in general; instead, it describes 60 of the most common and most serious software risks in detail. Jones uses a standard format for each risk that describes the severity of the risk, frequency of occurrence, root causes, associated problems, methods of prevention and control, and support available through education, books, periodicals, consulting, and professional associations. Much of the risk analysis in the book is supported by information from Jones's database of more than 4000 software projects.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. This book has one chapter that is specifically devoted to the topic of risk estimation. The software development method that Gilb describes in the rest of the book puts a strong emphasis on risk management.

Thomsett, Rob. *Third Wave Project Management*. Englewood Cliffs, N.J.: Yourdon Press, 1993. The book contains a 43-question risk-assessment questionnaire that you can use to obtain a rough idea of whether your project's risk level is low, medium, or high.

RAPID DEVELOPMENT



P
A
R
T
I

6

Core Issues in Rapid Development

Contents

- 6.1 Does One Size Fit All?
- 6.2 What Kind of Rapid Development Do You Need?
- 6.3 Odds of Completing on Time
- 6.4 Perception and Reality
- 6.5 Where the Time Goes
- 6.6 Development-Speed Trade-Offs
- 6.7 Typical Schedule-Improvement Pattern
- 6.8 Onward to Rapid Development

Related Topics

Rapid-development strategy: Chapter 2

ONCE YOU'VE LEARNED HOW TO AVOID the classic mistakes and mastered development fundamentals and risk management, you're ready to focus on schedule-oriented development practices. The first step in that direction is to understand several issues that lie at the heart of maximum development speed.

6.1 Does One Size Fit All?

You will use different practices to develop a heart-pacemaker control than you will to develop an inventory tracking system that tracks videotapes. If a software malfunction causes you to lose 1 video out of 1000, it might affect your profits by a fraction of a percent, but it doesn't really matter. But if a malfunction causes you to lose 1 pacemaker out of 1000, you've got real problems.

Different projects have different rapid-development needs, even when they all need to be developed "as fast as possible." Generally speaking, products that are widely distributed need to be developed more carefully than products that are narrowly distributed. Products whose reliability is important need to be developed more carefully than products whose reliability doesn't much matter. Figure 6-1 illustrates some of the variations in distribution and reliability.

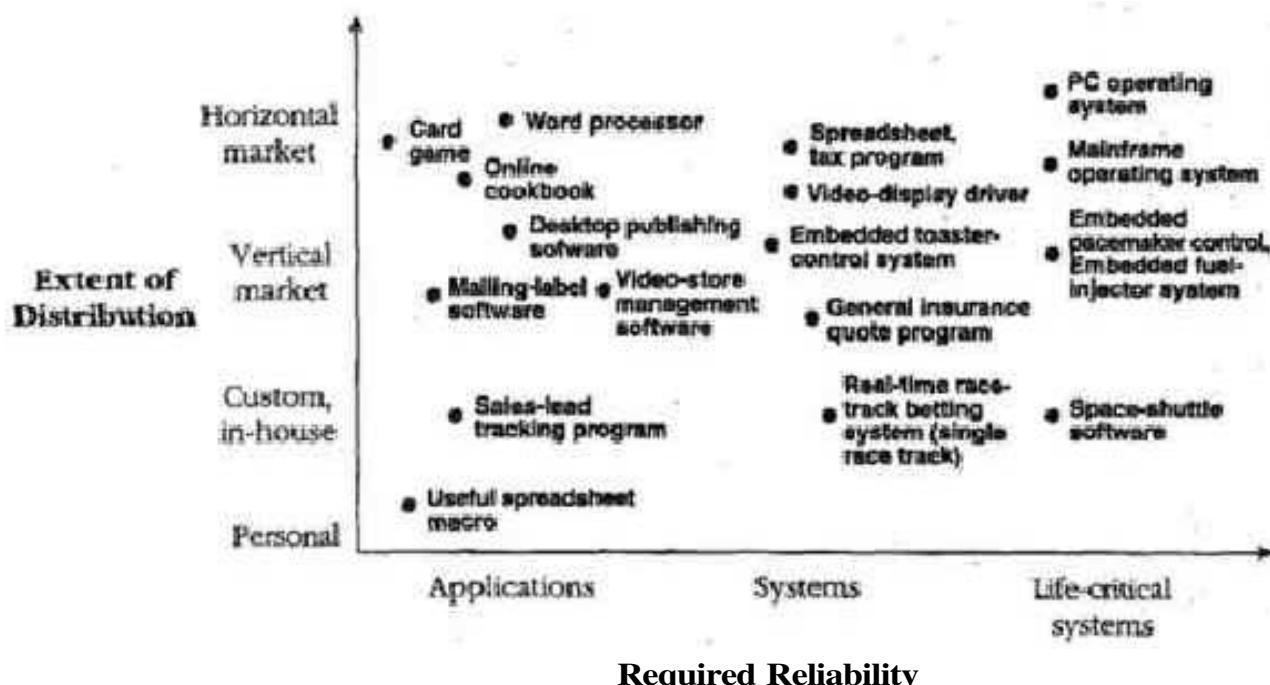


Figure 6-1. *Different kinds of software require different kinds of solutions. Practices that would be considered to be quick and dirty for an embedded bean-pacemaker control might be overly rigorous for an online cookbook,*

CROSS-REFERENCE
For more on customizing software processes to the needs of specific projects, see Section 2.4, "Which Dimension Matters the Most?"

The specific entries in the grid are intended to serve only as illustrations. You could argue about whether a video-display driver or a tax program needs to be more reliable or whether desktop-publishing software or spreadsheet programs are more widely distributed. The point is that both the extent of distribution and the required reliability vary greatly among different kinds of software. A software failure can cause a loss of time, work, money, or human life. Some schedule-oriented development practices that are perfectly acceptable when only time is at stake would be unconscionably reckless when human life is at stake.

On the other hand, practices that would be considered quick and dirty in a life-critical system might be overly rigorous for a custom business-software application. Rapid development of limited-distribution custom software could conceivably consist of what we think of as "workarounds" in more widely distributed software. Glue together some pieces that solve today's problem today, not tomorrow. Tomorrow might be too late—a late solution might be worthless.

As a result of this tremendous variation in development objectives, it's impossible to say, "Here's the rapid-development solution for you" without knowing your specific circumstances. The right solution for you depends on where you would place yourself on Figure 6-1's grid. Many products don't fit neatly into the grid's categories, and products vary in many ways other than degree of reliability and extent of distribution. That means that most people will need to customize a solution for their situation. As Figure 6-2 suggests, one size does not fit all.

»



Figure 6-2. *One size does not fit all.*

6.2 What Kind of Rapid Development

The most central issue to the topic of rapid development is determining what kind of rapid development you need. Do you need a slight speed edge, more predictability, better progress visibility, lower costs, or more speed at all costs?

One of the most surprising things I've discovered while doing the background research for this book is that many people who initially say they need

faster development find that what they really need is lower cost or more predictability—or simply a way to avoid a catastrophic failure.

You can ask several questions to help determine what kind of rapid development you need:

- How strong is the product's schedule constraint?
- Does the project's emphasis on schedule arise because it is really one of the common "rapid-development look-alikes"?
- Is your project limited by any weaknesses that would prevent a rapid-development success?

The following sections describe how to answer these questions.

Products with Strong Schedule Constraints

Products that truly need to focus on all-out development speed rather than cost or predictability have a different time-value curve than typical products have. As the value line for a typical product in Figure 6-3 shows, the value of a typical product declines gradually as time goes by. But with a product that has a strong schedule constraint, there is a point at which the value of the product declines precipitously.

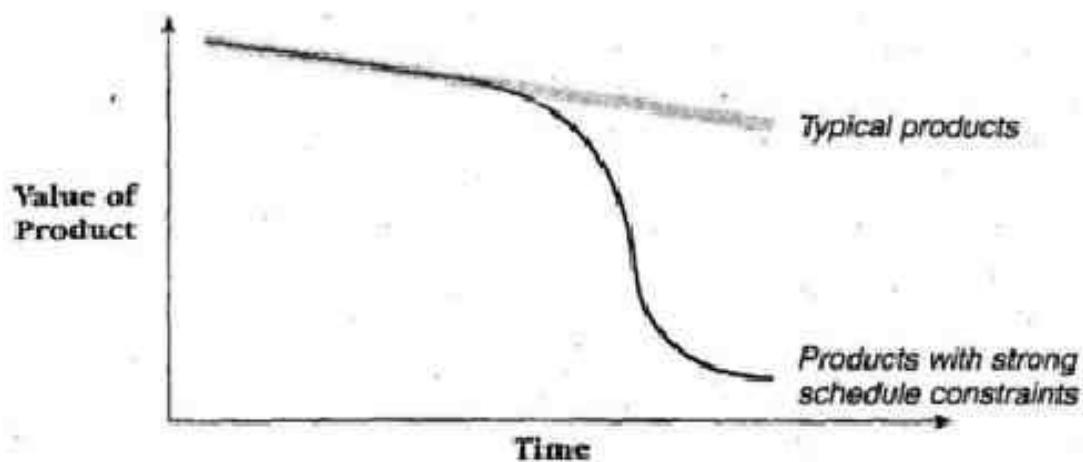


Figure 6-3. Depiction of value over time for typical products and products with strong schedule constraints. There isn't as much urgency to complete a typical product by any particular date as there is for a product that has a strong schedule constraint.

For a typical product, efficient development usually provides the best combination of development cost and schedule performance. But maybe your product must be ready in time for the Christmas sales season or you'll have to wait another year. Maybe you need to make a payroll-system change in time to comply with a new tax law. Maybe your company is about to go under financially, and you need the revenue from the product to save the

company. Or maybe you need to leapfrog a competitive product, and you stand to double your revenue if you can beat your competitor to market by 6 weeks instead of releasing your product 2 weeks after they do.

As the graph suggests, on these projects there may be a point by which, if you haven't released your product, you might as well not have developed it at all. In these cases, a focus on all-out development speed can be appropriate.

Rapid-Development Look-Alikes

In some instances the demand for "rapid development" comes via a circuitous path from users or customers or upper management. They can apply an incredible amount of pressure to get a product done fast, but sometimes they really want lower cost or less risk instead. They just don't know how to ask for those things—or don't know that those things do not go hand in hand with all-out development speed.

Before you orient your project toward the shortest schedule rather than the least cost, lowest risk, or best functionality, find out what's really needed. Several rapid-development look-alikes appear to call for all-out development speed but really call for something else; these are discussed in the following subsections.

Runaway prevention. If the software organization has a history of overshooting its planned schedules and budgets, the customer might ask for "rapid development." But in this case, what the customer really wants is assurance that the project will be completed close to its target schedule and budget.

You can distinguish this rapid-development look-alike from a need for all-out development speed either by realizing that there's no specific schedule goal other than "as soon as possible" or, if there is a specific goal, by finding that no one can explain why it matters. A history of runaway projects can be another tip-off. The solution in this case is not the use of schedule-oriented practices but rather the use of better risk management, project estimation, and management control.

Predictability. In many instances, customers want to coordinate the software-development part of a project with revenue projections, marketing, personnel planning, and other software projects. Although they might call for "rapid development," they're really calling for predictability good enough to let them coordinate related efforts. If your customers emphasize the need to complete the software "on time" and don't have an external constraint such as a trade show, they are probably more concerned about predictability than out-and-out development speed. In that case, focus on efficient development, and emphasize practices that reduce schedule risk.

Lowest cost. It isn't uncommon for customers to want to minimize the cost of a software-development project. In such cases, they will talk about getting the software done quickly, but they will emphasize their budget concerns more than their schedule concerns.

CROSS-REFERENCE

For more on schedule compression, see "Costs increase rapidly when you shorten the schedule below nominal" in Section 8.6.

If the customers' primary concern is the cost of a project, a focus on development schedule is particularly unfortunate. Although it's logical to assume that the shortest development schedule is also the cheapest, in actuality the practices that minimize cost and schedule are different. Lengthening the schedule somewhat beyond the nominal schedule and shrinking the team size can actually reduce the total cost of a project. Some rapid-development practices increase the total cost.

Fixed drop-dead date. As shown in Figure 6-3, sometimes the value of a product declines steadily over time, and sometimes it declines precipitously after a certain point. If there is a point at which it declines precipitously, it seems logical to say that: "We need all-out development speed so that we can be sure to release the product by that point."

But whether you need rapid development really depends on how much time you have to do the project and how much time it would take to develop the project using efficient-development methods. Figure 6-4 shows two possibilities.

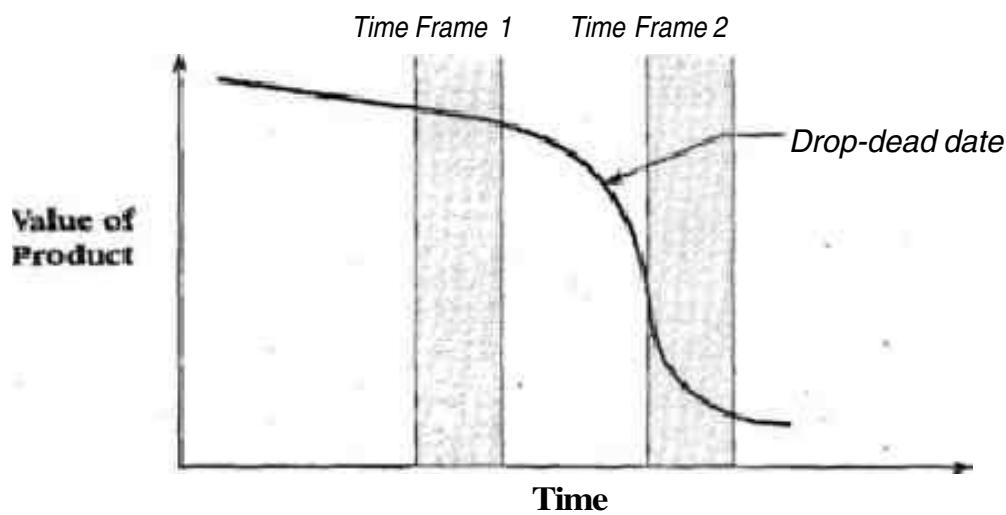


Figure 6-4. Whether you need to use rapid-development practices depends on how soon you need the software. If you can develop it in Time Frame 1 by using efficient-development practices, you should do that and keep risks low instead of focusing on speed-oriented practices that can increase risk.

If you can complete the project in Time Frame 1 (before the drop-dead date) by using efficient-development practices, then do that—and focus on risk-reduction rather than development speed. That will provide the greatest likelihood of completing the project on time. Some rapid-development practices

reduce development time but also increase schedule uncertainty, and it would be a mistake to use practices that increase schedule risk in this situation.

If efficient-development practices alone aren't capable of completing the project before the drop-dead date—for example if they are capable only of completing the project in Time Frame 2—then you'll need to use speed-oriented practices to have any chance of completing the project on time.

Desire for free overtime. In a few instances, the customer's (or manager's) interest in rapid development masks a desire to improve rapid development's bottom line by eking out as much unpaid overtime as possible. The sense of urgency created by an ambitious schedule helps to do that.

This look-alike is easy to distinguish from true rapid development because the customer will stress the importance of the schedule and simultaneously refuse to provide the support needed to improve development speed through any means other than unpaid overtime. The customer won't kick in for more developers, improved hardware tools, improved software tools, or other kinds of support. The customer won't be willing to make feature-set trade-offs to achieve schedule goals. On a true rapid-development project, the customer will be eager to consider any and all means of shortening the schedule.



FURTHER READING

For additional moral support in this situation, see "Spanish Theory Management" in *Peopleware* (DeMarco and Lister 1987).

If meeting the project's schedule is important enough to put pressure on you, it is important enough for the customer to increase the level of support for the project. If the company asks its developers to work harder, it must be willing to work harder, too. If you find yourself in a situation in which your customer is simply trying to get your team to work for free, there is probably almost nothing that you can do to improve it. Customers who practice this style of software development do not have your best interests in mind. Your most sensible options are to refuse to work on such projects or to change jobs.

So, Is All-Out Rapid Development Really What You Need?

It's a fact of life that customers—including end-users, marketers, managers, and others—will always clamor for new features and new releases. But customers are also aware of the disruption that a product upgrade can cause. Be aware that customers expect you to balance product, cost, and schedule for them. Of course, they will request that you provide a great product at low cost on a short schedule, but you usually get to pick only two out of these three desires. Releasing a low-quality product on a short schedule is usually the wrong combination. If you release a low-quality product on time, people will remember that it was low-quality—not that it was on time. If you release a late product that knocks their socks off, your customers will remember that

you released a knockout product; in retrospect, the late delivery won't matter as much as it seems to now.

To determine whether customer requests justify an all-out rapid-development effort, try to determine whether the value line of your product looks more like the typical product or like products with strong schedule constraint shown in Figure 6-3- Find out whether an external date is driving the schedule or whether the date is really just "as soon as possible." Finally, find out whether top management will provide the level of support you'll need for a rapid-development effort. There's little point in going all out if you have to do it on your own.

If you're not sure that development speed occupies top priority, take your time and develop software you can be proud of. Develop a program that's worth waiting for; high-quality products are harder to compete with than are quickly delivered mediocre products.

CROSS-REFERENCE
For more on the development of Word for Windows, see "An Example of Overly Optimistic Scheduling" in Section 9.1.

The history of the microcomputer software industry is replete with examples of products that were delivered late but went on to achieve immense popularity. The development of Microsoft Word for Windows 1.0 was originally scheduled to take one year and took five (lansiti 1994). Microsoft Windows 95 was delivered iVa years later than originally announced (Cusumano and Selby 1995) and became one of the fastest-selling products in software history. One financial product that I worked on was delivered 50 percent later than originally scheduled by its company but went on to become the most popular software product in that company's 25-year history. For each of these products, timely release (as originally defined) was not a key factor, even though everyone thought that the development schedule was critically important at the time.

6.3 Odds of Completing on Time

Many projects are perceived to be slow; however, not all projects are slow in the same way. Some development efforts really are slow, and others merely appear slow because of unreachable effort estimates.

One view of software-project estimation holds that every project has one exact time at which it should be completed. This view holds that if the project is run well, there is a 100-percent chance that it will be completed on a particular date. Figure 6-5 shows a graphical representation of that view.

Most developers' experience doesn't support this view. Many unknowns contribute to software schedules. Circumstances change. Developers learn more about the product they are building as they build it. Some practices work better than expected, others worse.

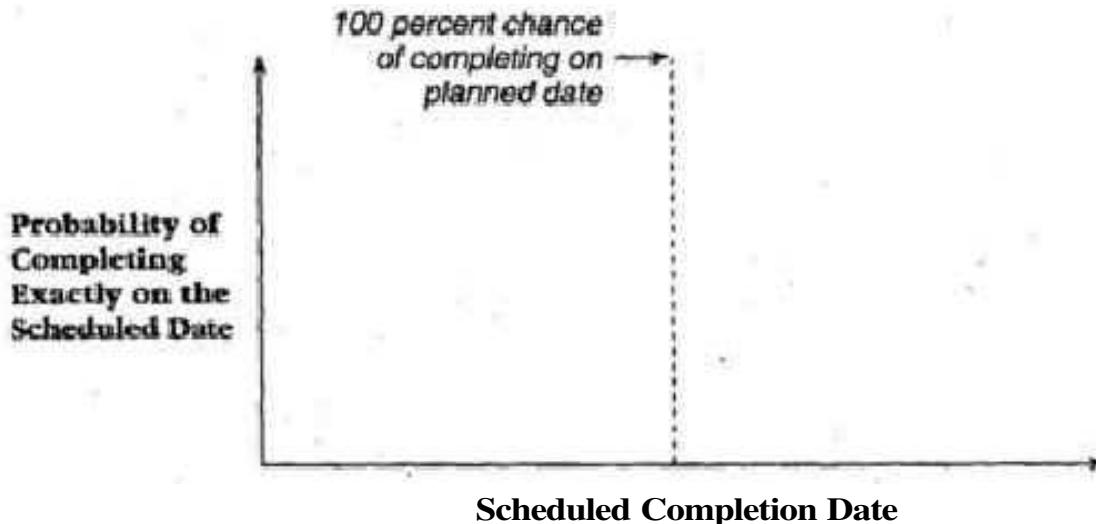


Figure 6-5. One view of software scheduling. The project is thought to have a 100-percent chance of being completed on a specific date.

Software projects contain too many variables to be able to set schedules with 100-percent accuracy. Far from having one particular date when a project would finish, for any given project there is a range of completion dates, of which some are more likely and some are less. The probability distribution of that range of dates looks like the curve shown in Figure 6-6.

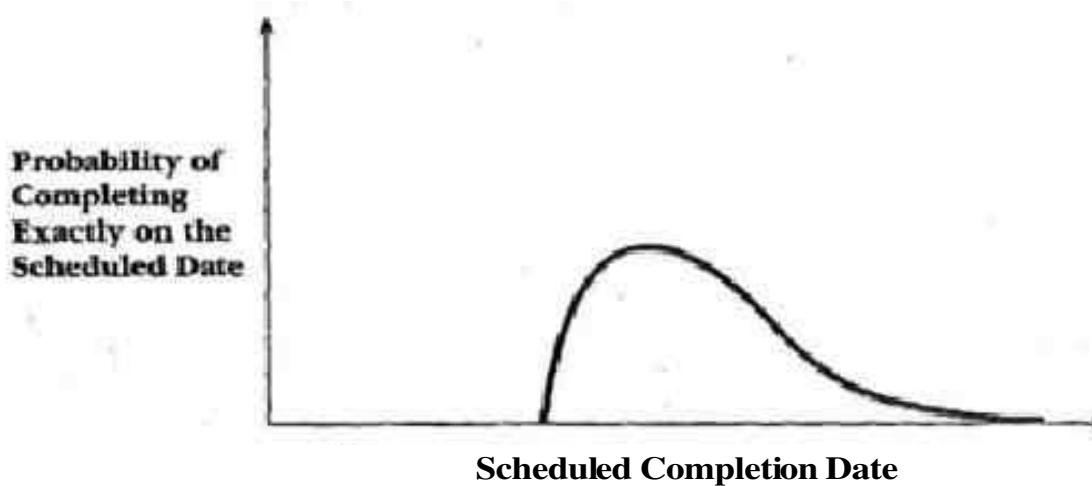


Figure 6-6. The shape of a software schedule. Because of the unknowns that feed into a software project's schedule, some completion dates are more likely and some are less, but none are certain.

CROSS-REFERENCE
For more on the shortest possible schedules, see "Shortest Possible Schedules" in Section 8.6.

The shape of this probability curve expresses several assumptions. One is that there is an absolute limit on how quickly you can complete any particular project. Completion in a shorter amount of time isn't just difficult; it's impossible. Another assumption is that the shape of the curve on the "early" side isn't the same as the shape on the "late" side. Even though there is a sharp limit to how quickly you can complete a project, there is no sharp limit to how slowly you can complete one. Since there are more ways to make a project late than there are to make it early, the slope of the curve on the late side is more gradual than it is on the early side.

The area on the far left side of the graph is the "impossible-development zone." This zone represents a level of productivity that no project has ever achieved. Projects that are scheduled in this zone are guaranteed to overrun their planned schedules.

The area on the left side of the curve is the "rapid-development zone." A project that is completed in this zone is considered to be rapid because it had less than a 50-percent chance of being completed in the scheduled time. A development team that completes a project in this zone has beaten the odds.

The area in the middle of the curve is the "efficient-development zone." A project that is completed in this zone is considered to be efficient because it has neither beaten the odds nor been beaten by them. Most likely, the project has come in close to its estimated completion date. Effective software-development organizations consistently schedule and complete their projects in this zone, which represents a good combination of schedule and cost.

The area on the right side of the curve is the "slow-development zone." A project that is completed in this zone is considered to be slow because it actually had a better than 50-percent chance of coming in earlier. As far as the schedule is concerned, a project that has finished in this zone without intending to has blown it. Success at 50/50 scheduling depends both on accurate estimation and on getting an accurate estimate accepted, topics discussed in detail in Chapters 8 and 9.

6.4 Perception and Reality

Suppose that six months from now you plan to move to a new town 100 miles away and build a new house. You arrange with Honest Abe's Construction Company to construct the house for you. You agree to pay Abe \$100,000, and Abe agrees to have the house ready in six months so that you can move into it when you move to town. You've already bought the site, and Abe agrees that it's suitable. So you shake hands, pay him half the money as a down payment, and wait for your house.

After a few weeks, you get curious about how work on the house is going, so one weekend you drive the 100 miles to the site to take a look at it. To your surprise, all you see at the site is an area where the dirt has been leveled. No foundation, no framing, no other work at all. You call Honest Abe and ask, "How's the work on my house going?" Abe says, "We're getting kind of a slow start because of another house that we've been finishing. I built some slack into my estimate for your house, though, so there's nothing to worry about."

Your job gets busy again, and by the time you have a chance to look at your house's progress again, three months have passed. You drive to the site again, and this time the foundation has been poured, but no other work is visible. You call the contractor, who says, "No problem. We're right on schedule." You're still nervous, but you decide to take Abe's word for it.

The fourth and fifth months go by. You call Abe a few times to check on progress, and each time he says it's going great. At the beginning of the sixth month, you decide to drive to look at the house one more time before it's finished. You're excited to see it. But when you get to the site, the only thing you see is the frame for the house. There's no roofing, siding, plumbing, wiring, heating, or cooling. You're nervous and tense, and you decide to check some of Honest Abe's references. You find that once in a while Abe has managed to complete a house when promised, but most of the time his houses are anywhere from 25 percent to 100 percent late. You're irate. "We're five months into a six-month schedule, and you hardly have anything done," you growl at him. "When am I going to get my house? I need to move into it a month from now." Abe says, "My crew is working as hard as possible. You'll get your house on time. Trust me."

Do you decide to trust Abe? Of course not! Not with his track record and the progress you've seen so far.

Yet in similar cycles in software development—with similar time frames and even larger amounts of money—we expect our customers to want less in the way of signs of progress than we would expect from someone building a house for us. We expect them to sign off on a set of requirements and then just sit tight for weeks or months or even years before we have anything tangible to show them. No wonder customers get nervous! No wonder customers think that software development takes a long time!

Even if you're completely on schedule, be aware that the perception of slow development can affect your project as much as the reality. Even though we do it all the time, it's unreasonable to expect customers to sit tight for months on end, and it's part of our job to provide them with steady signs of progress.

Unrealistic Customer Expectations

CROSS-REFERENCE
For details on problems associated with unrealistic schedules, see Section 9.1, "Overly Optimistic Scheduling."

Sometimes overcoming the perception of slow development requires more than providing steady signs of progress. Most of today's projects are scheduled in the rapid or impossible zones. Most projects lack the planning and resource commitments needed to meet their aggressive schedules. The project planners often do not even realize just how ambitious their schedules are, and, as Figure 6-9 suggests, they are usually completed in the slow zone.

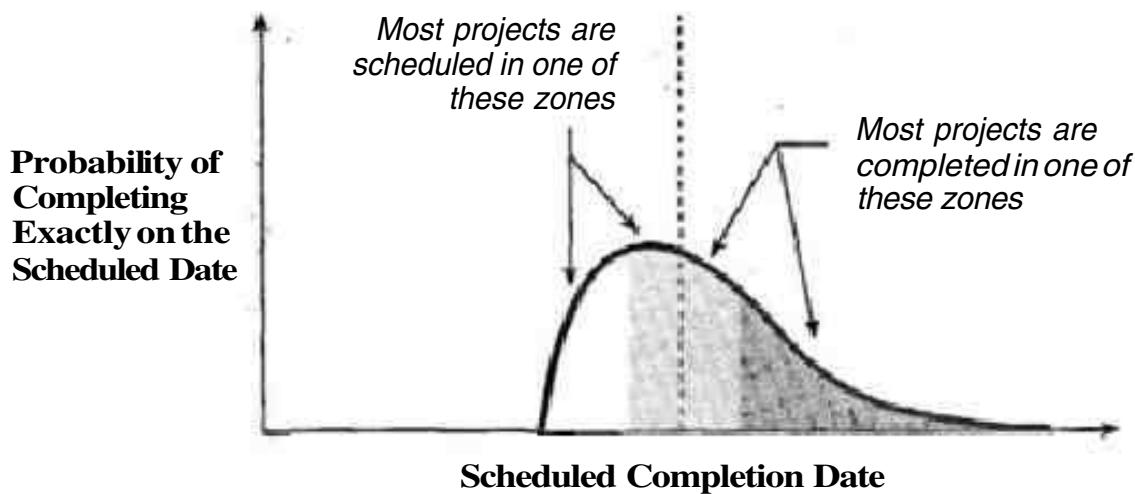


Figure 6-9. Typical planning in relation to the software-schedule curve. Because of unrealistic expectations, most projects will be perceived as slow even if they are completed in the efficient or rapid zones.

The gap between the scheduled completion date and the actual completion date accounts for much of the perception that software projects are slow. If the average project is scheduled in the impossible zone but staffed and completed in the efficient zone, people will consider it to have failed even though its developers have completed it efficiently and might actually have completed it as quickly as possible given the resources provided.

Overcoming the Perception of Slow Development

In general terms, you can overcome the problem of slow development in either of two ways:

- *Address the reality of slow development.* Make the actual schedules shorter by moving from slow development to efficient development or by moving from efficient development to rapid development.
- *Address the perception of slow development.* Eliminate wishful thinking, and make planned schedules more realistic by lengthening them to close the gap between planned and actual completion dates. Use practices that highlight progress visibility. Sometimes customers don't want increased development speed as much as they just want you to keep them informed.

The development speed zone you're in currently will determine whether you should focus on slow development itself or on the perception of slow development. Most often, you'll need to address the problem on both levels.

6.5 Where the Time Goes

CROSS-REFERENCE

For details on figuring out where the time goes on your own projects, see Chapter 26, "Measurement."

One strategy for achieving rapid development is to determine the area in which most of the time is spent on a typical project and then try to reduce that time. You can shrink some areas more readily than others, and attempting to reduce some areas can inadvertently lengthen the schedule.

You can view time on a software project from many angles, and the different views produce different insights into where the time goes. The next subsection presents the classic (phase-by-phase) view, and the subsections after that present other views.

The Classic View

Most projects start out in an ill-defined, pre-requirements phase that can last for quite a long time. At some point, requirements gathering starts in earnest, and at some point after that, software development officially begins. The post-requirements activities tend to be the better defined part of the project. Table 6-1 provides a rough idea of where the time is spent on post-requirements activities on efficiently run small and large projects.

Table 6-1. Approximate Activity Breakdown by Size of Project

Activity	Small Project (2,500 lines of code)	Large Project (500,000 lines of code)
Architecture/design	10%	30%
Detailed design	20%	20%
Code/debug	25%	10%
Unit test	20%	5%
Integration	15%	20%
System test	10%	15%

Source: Adapted from *Code Complete* (McConnell 1993).

The most time-expensive activities of a small project are the construction activities of detailed design, code/debug, and unit test. If you were able to magically eliminate them, you could cut your project's effort by 65 percent. On a large project, construction activities take up less of the total effort. Magically eliminating them would reduce your project's effort by only about 35 percent.



CLASSIC MISTAKE

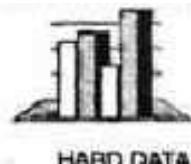


HARD DATA

Most of us have learned through hard experience not to arbitrarily abbreviate the upstream activities of architecture and design. Cutting design time by 5 percent might seem as though it would reduce the development schedule by 5 percent, but what is more likely to happen is that any time saved by shortchanging design will be paid back with interest during the later stages of the project, (Actually, the amount you'll pay back will seem more like usury than interest.) Put another way, a design defect that takes an hour-and-a-half to fix at design time will take anywhere from two days to a month to fix if it isn't detected until system testing (Pagan 1976).

A more effective strategy than trying to abbreviate the earlier stages arbitrarily is to perform them as efficiently as possible or to pick practices that require less design. (An example would be using a code library for part of the system.) You're more likely to reduce total development time by spending more time in upstream activities, not less.

Soft Spots



HARD DATA

After surveying more than 4000 projects, Capers Jones reported that the software industry on the whole is probably about 35 percent efficient (Jones 1994). The other 65 percent of the time is spent on harmful or unproductive activities such as use of productivity tools that don't work, repair of carelessly developed modules, work lost because of lack of configuration control, and so on. Where might time be saved? The next few subsections describe some of those areas.

CROSS-REFERENCE
For more on the importance of avoiding rework, see Section 4.3, "Quality-Assurance Fundamentals,"

CROSS-REFERENCE
For more on feature creep, see Section 14.2, "Mid-Project: Feature-Creep Control."

Rework. Reworking defective requirements, design, and code typically consumes 40 to 50 percent of the total cost of software development (Jones 1986b; Boehm 1987a). Correcting defects early, when they're cheapest to correct and when such corrections preempt later rework, represents a powerful opportunity to shorten your projects.

Feature creep. Feature creep can arise from requirements changes or developer gold-plating. The typical project experiences about a 25-percent change in requirements throughout its development, which adds more than 25-percent effort to the project (Boehm 1981, Jones 1994). Failing to limit changes to those that are absolutely essential is a classic development-speed mistake, and eliminating feature creep goes a long way toward eliminating schedule overruns.

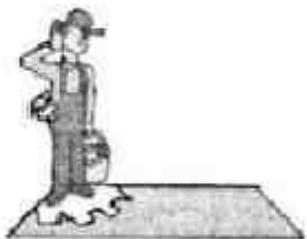
CROSS-REFERENCE

For more on requirements analysis, see Section 14.1, "Early Project: Feature-Set Reduction."

Requirements specification. An activity that isn't shown in Table 6-1 is requirements specification. Whereas the activities listed in Table 6-1 are concerned with specifying the solution to a problem, requirements specification is concerned with specifying the problem itself. It is more open-ended than other development activities, and the amount of time you spend gathering requirements doesn't bear any particular relationship to the total time you'll spend building the program. You could spend 12 months amassing requirements for a system that will take 36 months to build. Or you could spend the same 12 months mediating among several groups to define a system that will ultimately take only 6 months to build. Typically, requirements specification takes between 10 percent and 30 percent of the elapsed time on a project (Boehm 1981).

Because requirements gathering is such an open-ended activity, it's possible to burn huge amounts of time unnecessarily. A moderate sense of urgency during requirements gathering can help to prevent a full-scale sense of panic at the end of a project.

Rapid-development practices that have been developed to combat wasted time during requirements specification include Joint Application Development (JAD), evolutionary prototyping, staged releases, and various risk management approaches. These practices are described elsewhere in this book.



CLASSIC MISTAKE

The "fuzzy front end." Another kind of activity that isn't described in Table 6-1 is the "fuzzy front end." The total time needed to develop a software product extends from the point at which the product is just a glimmer in someone's eye to the point at which the working software is put into the customer's hands. At some time after the software is a glimmer in someone's eye, a "go" decision is made, and the software project officially commences. The time between the initial glimmer and the "go" decision can be a long time indeed. That "fuzzy front end" can consume much of the time available for getting a product to market. A typical pattern is described in Case Study 6-1.

Case Study 6-1. Wandering in the Fuzzy Front End

Bill is a manager at Giga Safe Insurance Company. Here are the notes he took about the approval process for Giga-Quote 1.0, an insurance quote program:

October 1. We want to develop a new quote program for our field agents. We want the program to upload the day's quotes to the head office each night. It will take about 12 months to complete its development, so we can't get it done in time for this January's rate increase, but we should be able to get it done for the rate increase after that (15 months from now). We should aim to complete it by November 1 (13 months from now) so that we have time

(continued)

Case Study 6-1. Wandering in the Fuzzy Front End, *continued*

to train the field agents before the new rates go into effect. I'll propose the project at the executive committee meeting at the end of the month.

January 2. The Giga-Quote proposal got bumped off the executive committee agenda two months in a row. I finally brought it up at the end of December and got the approval to draw up a business-case analysis.

February 1. Business-case analysis is complete; it just needs to be reviewed.

March 1. Two key sales managers are on vacation. The business-case analysis can't be approved until they've reviewed it.

April 15. All reviews are complete, and the project is a "go." The executive committee still wants the project completed by November 1, though, so the team had better start coding now.

Because no formal management controls are in place—no schedule, no budget, no goals, and no objectives—progress during this period can be hard to track. Moreover, managers tend to give this phase a low priority because the financial impact is distant. You can lose time during the front end in several ways:

- No one has been assigned responsibility for the product's development.
- No sense of urgency exists for making a go/no-go decision about developing the product.
- No mechanism exists to keep the product from slipping into dormancy.
- No mechanism exists to revive the product once it has slipped into dormancy.
- Key aspects of the product's viability—technical feasibility and market appeal—can't be explored until the product receives budget approval.
- The product must wait for an annual product-approval cycle or budgeting cycle before it can receive budget approval.
- The team that develops the product isn't built from the team that worked for the product's approval. Time and momentum are lost assembling the development team, familiarizing it with the product, and handing off the product to it.

**FURTHER READING**

For more on the fuzzy front end, see *Developing Products in Half the Time* (Smith and Reinertsen 1991).

The effort expended in the front end of a project is usually low, but the cost resulting from delayed time-to-market can be high. The only way to recapture a month wasted on the front end is to shorten the product-development cycle by a month on the back end. Shortening the full-scale development effort by a month costs far more than shortening the front end by the same amount. The front end presents one of the cheapest and most effective rapid-development opportunities available.

6.6 Development-Speed Trade-Offs

With rare exceptions, initial resource estimates and schedules are unacceptable.

This is not because the programmers are unresponsive, but because the users generally want more than they can afford.

If the job doesn't fit the available schedule

and resources, it must either be pared down or the time and resources increased.

Watts Humphrey

One of the philosophies undergirding this book is that it is better to make trade-off decisions with your eyes open than closed. If development speed is truly your top priority, then go ahead and increase the cost of the project and compromise the product's feature set in order to deliver it on time. But understand the implications of the decisions you make. Don't close your eyes and hope that somehow you'll be able to optimize your project for schedule, cost, and features all at the same time. You won't. Instead, you'll end up optimizing it for none of them; you'll waste time and money and deliver a product with less functionality than you otherwise could have.

Schedule, Cost, and Product Trade-Offs

A trade-off triangle with schedule, cost, and quality at its corners is a general management fundamental. In software, however, having a "quality" corner on a trade-off triangle doesn't make much sense. A focus on some kinds of quality reduces cost and schedule, and on other kinds increases them. In the software arena, a better way to think of trade-offs is among schedule, cost, *and product*. The product corner includes quality and all other product-related attributes including features, complexity, usability, modifiability, maintainability, defect rate, and so on. Figure 6-10 illustrates the software trade-off triangle.



Figure 6-10. Software trade-off triangle. You have to keep schedule, cost, and product in balance for the project to succeed.

To keep the triangle balanced, you have to balance schedule, cost, and product. If you want to load up the product corner of the triangle, you also have to load up cost or schedule or both. The same goes for the other combinations. If you want to change one of the corners of the triangle, you have to change at least one of the others to keep it in balance.

To help me think about which option to manipulate, during planning discussions I like to visualize a large cardboard triangle with the corners labeled "schedule," "cost," and "product." The customers hold the corner or corners

that they want to control. Our job as software developers is to let customers show us which corners they are holding and then to tell them what has to be done to balance the triangle. If a customer is holding the "product" and "cost" corners, we tell them what the "schedule" corner has to be. If they are holding only the "product" corner, we can give them a variety of cost-and-schedule combinations. But we developers absolutely must have at least one corner to hold on to. If your customer won't give you a corner of the triangle, you usually can't do the project.



CLASSIC MISTAKE

CROSS-REFERENCE
For more on negotiating in difficult environments, see
Section 9.2, "Beating Schedule Pressure."

Jim McCarthy reports that in informal polling he has found that about 30 to 40 percent of all development projects suffer from simultaneously dictated features, resources, and schedules (McCarthy 1995a). If schedule, cost, and product aren't initially in balance—and they rarely are—that suggests that 30 to 40 percent of all development projects start out with no ability to balance their project characteristics for success. When a customer hands you a product definition, a fixed cost, and a fixed schedule, they are usually *trying* to put a 10-pound product into a 5-pound sack. You can try to force the 10-pounder into the sack, stretch the sack, and tear the sack, but in the end all you'll do is wear yourself out—because it just won't fit. And you'll still have to decide whether you want to get a bigger sack or put less into the sack you have.

Quality Trade-Offs

CROSS-REFERENCE
For details on the relationship between defect rate and development time, see Section 4.3, "Quality-Assurance Fundamentals."

Software products have two kinds of quality, which affect the schedule in different ways. One kind of quality is a low defect rate. To a point, low defects and short development times go together, so there is no way to trade off that kind of quality for schedule. The road to the shortest possible schedule lies in getting the product right the first time so that you don't waste time reworking design and code.

CROSS-REFERENCE
For details on how to use this kind of quality to reduce development time, see Chapter 14, "Feature-Set Control."

The other kind of quality includes all the other characteristics that you think of when you think of a high-quality software product—usability, efficiency, robustness, and so on. Attention to this kind of quality lengthens the development schedule, so there is an opportunity for trading off this kind of quality against the schedule.

Per-Person-Efficiency Trade-Off

Is there a conflict between trying to achieve the greatest per-person productivity and the greatest schedule efficiency? Yes, there is. The easiest way to maximize per-person productivity is to keep the team size small. One of the easiest ways to shorten a software schedule is to increase team size, which increases total productivity but usually makes each person less efficient. Rapid development isn't always efficient.

6.7 Typical Schedule-Improvement Pattern

Organizations that try to improve their development speed by moving toward efficient development follow a predictable pattern. If you take 100 typical projects, you'd find that their chances of coming in on time would look like Figure 6-11.

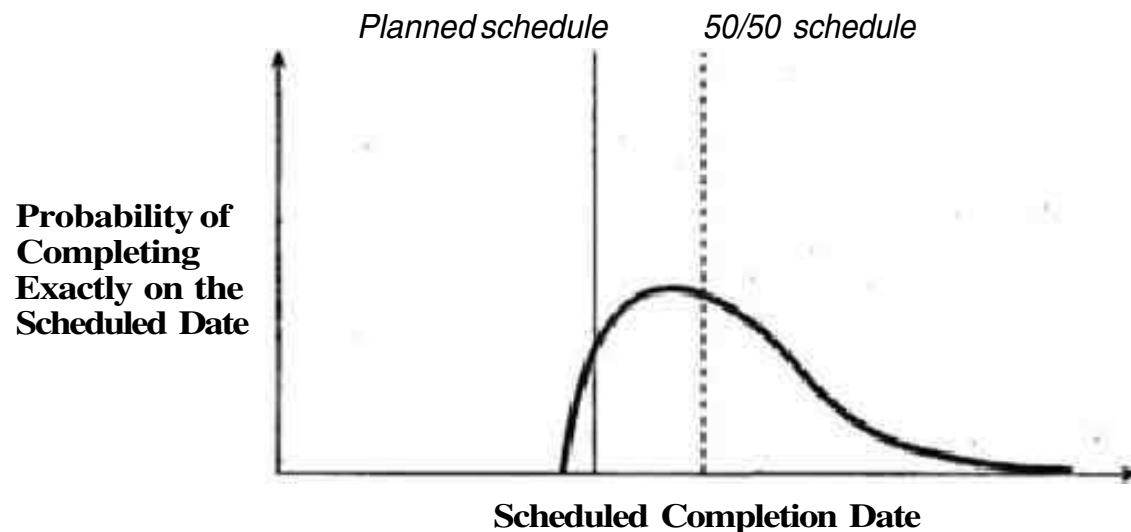


Figure 6-11. Typical-development schedule curve. Typical projects make schedule plans that they have almost no chance of meeting.

Among typical projects, the spread of project performance is wide, and many of the projects have severe overruns. Look at how much of the curve in Figure 6-11 is to the right of the planned-schedule line on typical projects. Few typical projects come anywhere close to meeting thek cost or schedule goals.



FURTHER READING

For a similar discussion, see "Capability Maturity Model for Software, Version 1.1° (Paulketal.1993).

As Figure 6-12 shows, among efficient-development projects, the schedule spread is narrower, with most projects coming in close to their cost and schedule targets. About half the projects finish earlier than the target date, and about half finish later. The planned schedules are longer than they are in typical development, but the actual schedules are shorter. This is partly a result of learning how to set targets more realistically and partly a result of learning how to develop software faster. The move from wishful thinking to meaningful project planning is a big part of what it takes to move from typical development to efficient development.

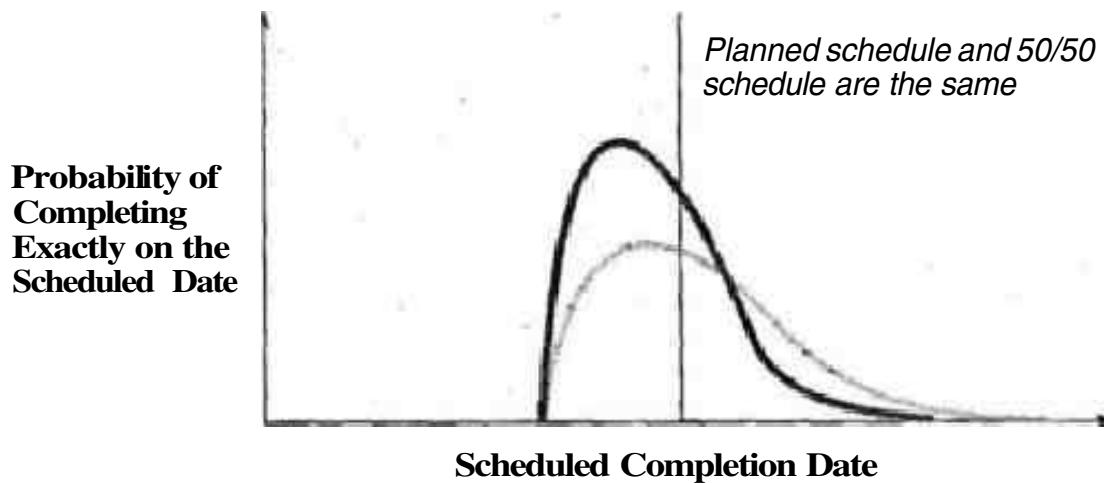


Figure 6-12. Efficient-development schedule curve. Planned schedules in efficient projects are longer (than planned schedules in typical projects, but "actual schedules are shorter,

Once you've achieved efficient development, the improvement pattern depends on whether you want to improve raw development speed or schedule predictability or both. Ideally, you could employ practices that would give you the tall, skinny curve shown in Figure 6-13-

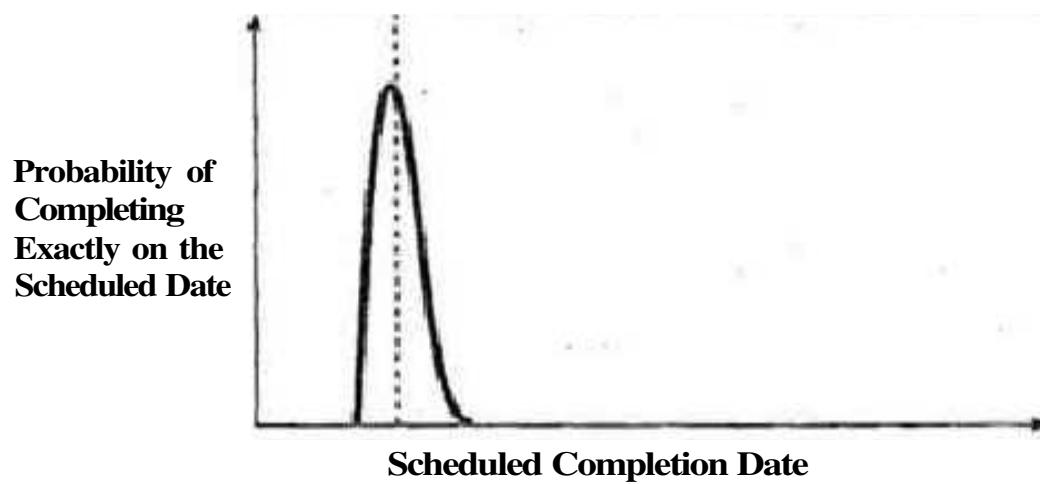


Figure 6-13. Ideal rapid-development schedule curve. If every single thing goes as planned, the result is great speed and predictability.

Unfortunately for all of us, the ideal curve in Figure 6-13 is as elusive in software development as it is in fad dieting. As shown in Figure 6-14, that means that most rapid-development practices are tilted either toward increasing development speed or reducing schedule risk, but not both.

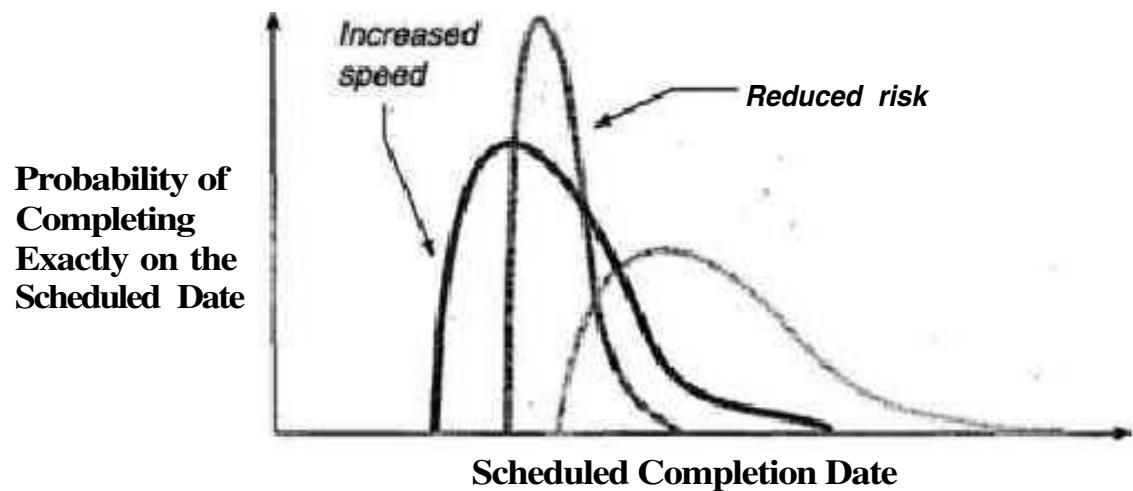


Figure 6-14. *Scheduling options. Rapid development can focus on either increasing development speed or reducing schedule-related risks.*

When you choose rapid-development practices, you need to decide whether you would rather improve the chance of delivering a product earlier or reduce the risk that the product will slip past a certain date. The rest of the book describes such practices.

6.8 Onward to Rapid Development

The remaining chapters in this part of the book describe approaches that contribute to rapid development. Here are the practices:

- Lifecycle Planning
- Estimation
- Scheduling
- Customer-Oriented Development
- Motivation
- Teamwork
- Team Structure
- Feature-Set Control
- Productivity Tools
- Project Recovery

Some of these topics could be considered as part of what I have described as "development fundamentals" or "efficient development." Because these approaches are critical to achieving maximum development speed, though, they are discussed in this part of the book.

Further Reading

DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982. This book contains much of the inspiration for this chapter's discussion of the shape of software schedules. DeMarco paints a humorous and sometimes painfully vivid picture of current estimating practices—which as far as I can tell haven't changed since he published his book in 1982. He lays out one approach for improving estimation and scheduling.

Martin, James. *Rapid Application Development*. New York: Macmillan Publishing Company, 1991. This book presents a different perspective on the core issues of rapid development for IS applications.

Smith, P.O., and D.G. Reinertseiv. *Developing Products in Half the Time*. New York: Van Nostrand Reinhold, 1991. Although not about software development specifically, this book contains many insights that relate to developing software products more rapidly. Chapter 3 contains a full discussion of the "fuzzy front end."

Lifecycle Planning

Contents

- 7.1 Pure Waterfall
- 7.2 Code-and-Fix
- 7.3 Spiral
- 7.4 Modified Waterfalls
- 7.5 Evolutionary Prototyping
- 7.6 Staged Delivery
- 7.7 Design-to-Schedule
- 7.8 Evolutionary Delivery
- 7.9 Design-to-Tools
- 7.10 Commercial Off-the-Shelf Software
- 7.11 Choosing the Most Rapid Lifecycle for Your Project

Related Topics

- Evolutionary delivery: Chapter 20
- Evolutionary prototyping: Chapter 21
- Staged delivery: Chapter 36
- Summary of spiral lifecycle model: Chapter 35
- Summary of lifecycle model selection: Chapter 25

EVERY SOFTWARE-DEVELOPMENT EFFORT goes through a "lifecycle," which consists of all the activities between the time that version 1.0 of a system begins life as a gleam in someone's eye and the time that version 6.74b finally takes its last breath on the last customer's machine. A lifecycle model is a prescriptive model of what should happen between first glimmer and last breath.

For our purposes, the main function of a lifecycle model is to establish the order in which a project specifies, prototypes, designs, implements, reviews, tests, and performs its other activities. It establishes the criteria that you use to determine whether to proceed from one task to the next. This chapter

focuses on a limited part of the full lifecycle, *the period between the first glimmer and initial release*. You can direct this focus either to new product development or to maintenance updates of existing software.

The most familiar lifecycle model is the well-known waterfall lifecycle model, which has some equally well-known weaknesses. Other lifecycle models are available, and in many cases they are better choices for rapid development than the waterfall model is. (The waterfall model is described in the next section, "Pure Waterfall.")

By defining the master plan for the project, the lifecycle model you choose has as much influence over your project's success as any other planning decision you make. The appropriate lifecycle model can streamline your project and help ensure that each step moves you closer to your goal. Depending on the lifecycle model you choose, you can improve development speed, improve quality, improve project tracking and control, minimize overhead, minimize risk exposure, or improve client relations. The wrong lifecycle model can be a constant source of slow work, repeated work, unnecessary work, and frustration. Not choosing a lifecycle model can produce the same effects.

Many lifecycle models are available. In the next several sections, I'll describe the models; and in the final section, I'll describe how to pick the one that will work best for your project.

Case Study 7-1. Ineffective Lifecycle Model Selection

The field agents at Giga-Safe were clamoring for an update to Giga-Quote 1.0, both to correct defects and to fix some annoying user-interface glitches. Bill had been reinstated as the project manager for Giga-Quote 1.1 after being removed at the end of Giga-Quote 1.0, and he brought in Randy for some advice. Randy was a high-priced consultant he had met at a sports bar.

"Here's what you should do," Randy said. "You had a lot of schedule problems last time, so this time you need to organize your project for all-out development speed. Prototyping is the fastest approach, so have your team use that." Bill thought that sounded good, so when he met with the team later that day, he told them to use prototyping.

Mike was the technical lead on the project, and he was surprised. "Bill, I don't follow your reasoning," he said. "We've got 6 weeks to fix a bunch of bugs and make some minor changes to the UI. What do you want a prototype for?"

"We need a prototype to speed up the project," Bill said testily. "Prototyping is the newest, fastest approach, and that's what I want you to use. Is there some kind of problem with that?"

(continued)

In evolutionary delivery, your initial emphasis is on the core of the system, which consists of lower level system functions that are unlikely to be changed by customer feedback.

Incremental Development Practices

The phrase "incremental development practices" refers to development practices that allow a program to be developed and delivered in stages. Incremental practices reduce risk by breaking the project into a series of small subprojects. Completing small subprojects tends to be easier than completing a single monolithic project. Incremental development practices increase progress visibility by providing finished, operational pieces of a system long before you could make the complete system operational. These practices provide a greater ability to make midcourse changes in direction because the system is brought to a shippable state several times during its development—you can use any of the shippable versions as a jumping-off point rather than needing to wait until the very end.

Lifecycle models that support incremental development include the spiral, evolutionary-prototyping, staged-delivery, and evolutionary-delivery models (discussed earlier in this chapter).

7.9 Design-to-Tools

The design-to-tools lifecycle model is a radical approach that historically has been used only within exceptionally time-sensitive environments. As tools have become more flexible and powerful—complete applications frameworks, visual programming environments, full-featured database programming environments—the number of projects that can consider using design-to-tools has increased.

CROSS-REFERENCE
For more on productivity tools, see Chapter 15, "Productivity Tools," and Chapter 31, "Rapid-Development Languages (RDLs)."

The idea behind the design-to-tools model is that you put a capability into your product only if it's directly supported by existing software tools. If it isn't supported, you leave it out. By "tools," I mean code and class libraries, code generators, rapid-development languages, and other software tools that dramatically reduce implementation time.

As Figure 7-12 suggests, the result of using this model is inevitably that you won't be able to implement all the functionality you ideally would like to include. But if you choose your tools carefully, you can implement most of the functionality you would like. When time is a constraint, you might actually be able to implement more total functionality than you would have been able to implement with another approach—but it will be the functionality that the tools make easiest to implement, not the functionality you ideally would like.

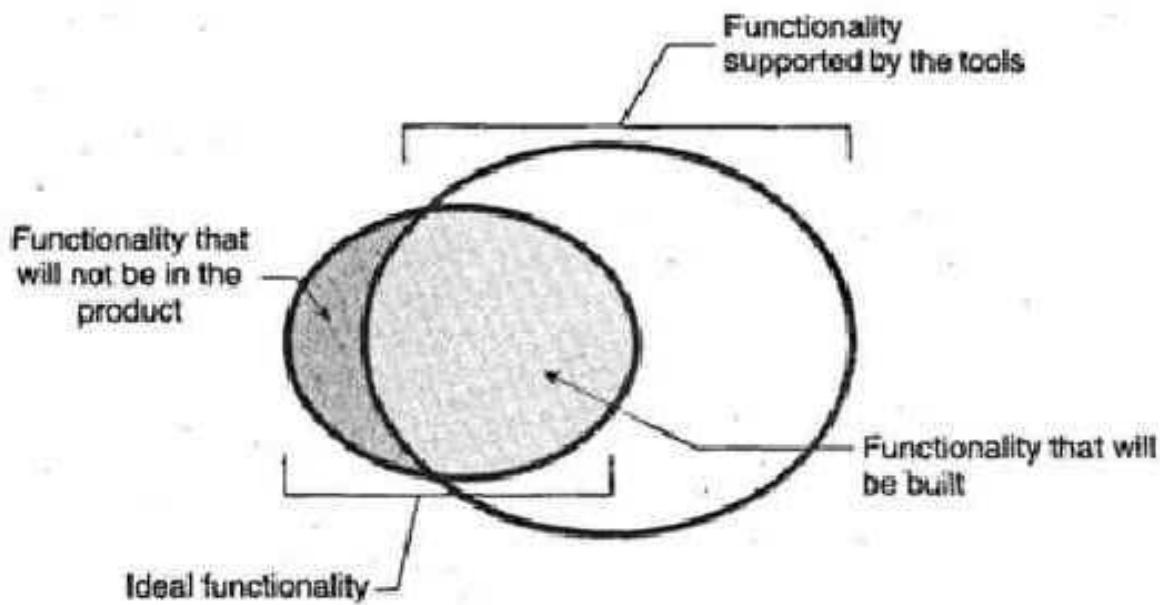


Figure 7-12. *Design-to-tools product concept. Design-to-tools can provide exceptional development speed, but it typically provides less control over your product's functionality than other lifecycle models would provide.*

This model can be combined with the other flexible lifecycle models. You might conduct an initial spiral to identify the capabilities of existing software tools, to identify core requirements, and to determine if the design-to-tools approach is workable. You can use a design-to-tools approach to implement a throwaway prototype, prototyping only the capabilities that can be implemented easily with tools. Then implement the real software using one of the other lifecycle models. You can also combine this model with staged delivery, evolutionary delivery, and design-to-schedule.

The design-to-tools model has a few main disadvantages. You lose a lot of control over your product. You might not be able to implement all the features that you want, and you might not be able to implement other features exactly the way you want. You become more dependent on commercial software producers—on both their product strategies and their financial stabilities. If you're writing small, mostly disposable programs, that might not be much of a problem; but if you're writing programs that you intend to support for a few years, each vendor whose products you use potentially becomes a weak link in the product chain.

7.10 Commercial Off-the-Shelf Software

One alternative that is sometimes overlooked in the excitement surrounding a new system is the option to buy software off the shelf. Off-the-shelf software will rarely satisfy all your needs, but consider the following points.

CROSS-REFERENCE
For details on problems associated with relying on outside vendors for technical products, see Chapter 28, "Outsourcing."

off-the-shelf software is available immediately. In the intervening time between when you can buy off-the-shelf software and when you could release software of your own creation, your users will be provided with at least some valuable capabilities. They can learn to work around the products' limitations by the time you could have provided them with custom software. As time goes by, the commercial software might be revised to suit your needs even more closely.

Custom software probably won't turn out to match your mental vision of the ideal software. Comparisons between custom-built software and off-the-shelf software tend to compare the actual off-the-shelf software to the idealized custom-built software. However, when you actually build your own software, you have to make design, cost, and schedule concessions, and the actual custom-built product will fall short of the ideal you envisioned. If you were to deliver only 75 percent of the ideal product, how would that compare to the off-the-shelf software? (This argument applies to the design-to-tools model, too.)

7.11 Choosing the Most Rapid Lifecycle for Your Project

Different projects have different needs, even if they all need to be developed as quickly as possible. This chapter has described 10 software lifecycle models, which, along with all their variations and combinations, provide you with a full range of choices. Which one is fastest?

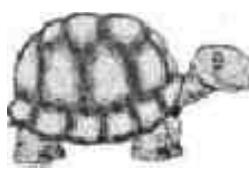
There is no such thing as a "rapid-development lifecycle model" because the most effective model depends on the context in which it's used. (See Figure 7-13.) Certain lifecycle models are sometimes touted as being more rapid than others, but each one will be fastest in some situations, slowest in others. A lifecycle model that often works well can work poorly if misapplied (as prototyping was in Case Study 7-1).

To choose the most effective lifecycle model for your project, examine your project and answer several questions:

- How well do my customer and I understand the requirements at the beginning of the project? Is our understanding likely to change significantly as we move through the project?
- How well do I understand the system architecture? Am I likely to need to make major architectural changes midway through the project?
- How much reliability do I need?
- How much do I need to plan ahead and design ahead during this project for future versions?

(continued on page 156)





Dinner Menu

Welcome to le Cafe de Lifecycle Rapide. Bon Appetit!

Entrees

Spiral

Handmade rotini finished with a risk-reduction sauce.
\$15.95

Evolutionary Delivery

Mouth-watering melange of staged delivery and evolutionary prototyping.
\$15.95

Staged Delivery

A five-course feast. Ask your server for details.
\$14.95

Design-to-Schedule

Methodology medley, ideal for quick executive lunches.
\$11.95

Pure Waterfall

A classic, still made from the original recipe.
\$14.95

Salads

Design-to-Toois

Roast canard generously stuffed with juienned multi-color beans.
Market Price

Commercial Off-the-Shelf Software

Chef's alchemic fusion of technology du jour. Selection varies daily.
\$4.95

Code-and-Fix

Bottomless bowl of spaghetti lightly sprinkled with smoked design
and served with reckless abandon.
\$5.95

Figure 7-13. Choosing a lifecycle model. No one lifecycle model is best for all projects. The best lifecycle model for any particular project depends on that project's needs.

- How much risk does this project entail?
- Am I constrained to a predefined schedule?
- Do I need to be able to make midcourse corrections?
- Do I need to provide my customers with visible progress throughout the project?
- Do I need to provide management with visible progress throughout the project?
- How much sophistication do I need to use this lifecycle model successfully?

CROSS-REFERENCE
For more on why a linear, waterfall-like approach is most efficient, see "Wisdom of Stopping Changes Altogether" in Section 14.2.

After you have answered these questions, Table 7-1 should help you decide which lifecycle model to use. In general, the more closely you can stick to a linear, waterfall-like approach—and do it effectively—the more rapid your development will be. Much of what I say throughout this book is based on this premise. But if you have reasons to think that a linear approach won't work, it's safer to choose an approach that's more flexible.

Table 7-1. Lifecycle Model Strengths and Weaknesses

Lifecycle Model Capability	Pure Waterfall	Code-and-Fix	Spiral	Modified Waterfalls	Evolutionary Prototyping
Works with poorly understood requirements	Poor	Poor	Excellent	Fair to excellent	Excellent
Works with poorly understood architecture	Poor	Poor	Excellent	Fair to excellent	Poor to fair
Produces highly reliable system	Excellent	Poor	Excellent	Excellent	Fair
Produces system with large growth envelope	Excellent	Poor to fair	Excellent	Excellent	Excellent
Manages risks	Poor	Poor	Excellent	Fair	Fair
Can be constrained to a predefined schedule	Fair	Poor	Fair	Fair	Poor
Has low overhead	Poor	Excellent	Fair	Excellent	Fair
Allows for midcourse corrections	Poor	Poor to excellent	Fair	Fair	Excellent
Provides customer with progress visibility	Poor	Fair	Excellent	Fair	Excellent
Provides management with progress visibility	Fair	Poor	Excellent	Fair to excellent	Fair
Requires little manager or developer sophistication	Fair	Excellent	'Poor	Poor to fair	Poor

Each rating is either "Poor," "Fair," or "Excellent." Finer distinctions than that wouldn't be meaningful at this level. The ratings in the table are based on the model's best potential. The actual effectiveness of any lifecycle model will depend on how you implement it. It is usually possible to do worse than the table indicates. On the other hand, if you know the model is weak in a particular area, you can address that weakness early in your planning and compensate for it—perhaps by creating a hybrid of one or more of the models described. Of course, many of the table's criteria will also be strongly influenced by development considerations other than your choice of lifecycle models.

Here are detailed descriptions of the lifecycle-model criteria described in Table 7-1:

Works with poorly understood requirements refers to how well the lifecycle model works when either you or your customer understand the system's requirements poorly or when your customer is prone to change requirements. It indicates how well-suited the model is to exploratory software development.

Lifecycle Model Capability	Staged Delivery	Evolutionary Delivery	Design-to-Schedule	Design-to-Tools	Commercial Off-the-Shelf Software
Works with poorly understood requirements	Poor	Fair to excellent	Poor to fair	Fair	Excellent
Works with poorly understood architecture	Poor	Poor	Poor	Poor to excellent	Poor to excellent
Produces highly reliable system	Excellent	Fair to excellent	Fair	Poor to excellent	Poor to excellent
Produces system with large growth envelope	Excellent	Excellent	Fair to excellent	Poor	N/A
Manages risks	Fair	Fair	Fair to excellent	Poor to fair	N/A
Can be constrained to a predefined schedule	Fair	Fair	Excellent	Excellent	Excellent
Has low overhead	Fair	Fair	Fair	Fair to excellent	Excellent
Allows for midcourse corrections	Poor	Fair to excellent	Poor to fair	Excellent	Poor
Provides customer with progress visibility	Fair	Excellent	Fair	Excellent	N/A
Provides management with progress visibility	Excellent	Excellent	Excellent	Excellent	N/A
Requires little manager or developer sophistication	Fair	Fair	Poor	Fair	Fair

Works with poorly understood architecture refers to how well the lifecycle model works when you're developing in a new application area or when you're developing in a familiar applications area but are developing unfamiliar capabilities.

Produces highly reliable system, refers to how many defects a system developed with the lifecycle model is likely to have when put into operation.

Produces system with large growth envelope refers to how easily you're likely to be able to modify the system in size and diversity over its lifetime. This includes modifying the system in ways that were not anticipated by the original designers.

Manages risks refers to the model's support for identifying and controlling risks to the schedule, risks to the product, and other risks.

Can be constrained to a predefined schedule refers to how well the lifecycle model supports delivery of software by an immovable drop-dead date.

Has low overhead refers to the amount of management and technical overhead required to use the model effectively. Overhead includes planning, status tracking, document production, package acquisition, and other activities that aren't directly involved in producing software itself.

Allows for midcourse corrections refers to the ability to change significant aspects of the product midway through the development schedule. This does not include changing the product's basic mission but does include significantly extending it.

Provides customer with progress visibility refers to the extent to which the model automatically generates signs of progress that the customer can use to track the project's status.

Provides management with progress visibility refers to the extent to which the model automatically generates signs of progress that management can use to track the project's status.

Requires little manager or developer sophistication refers to the level of education and training that you need to use the model successfully. That includes the level of sophistication you need to track progress using the model, to avoid risks inherent in the model, to avoid wasting time using the model, and to realize the benefits that led you to use the model in the first place.

Case Study 7-2. Effective Lifecycle Model Selection

Eddie had volunteered to oversee Square-Tech's development of a new product code named "Cube-It," a scientific graphics package. Rex, the CEO, felt that Square-Calc had given them a foot in the door they could use to become a market leader in scientific graphics.

Eddie met with George and Jill, both developers, to plan the project. "This is a new area for us, so I want to minimize the company's risk on this project. Rex told me that he wanted the preliminary product spec implemented within a year. I don't know whether that's possible, so I want you to use a spiral lifecycle model. For the first iteration of the spiral, we need to find out whether this preliminary spec is pure fantasy or whether we can make it a reality."

George and Jill worked for two weeks and then met with Eddie to evaluate the alternatives they had identified. "Here's what we found out. If the objective is to build the market-leading scientific-graphics package, there are two basic alternatives: beat the competition in features or beat them in ease of use. Right now, the easier niche to fill seems to be ease of use.

"We analyzed the risks for each alternative. If we go the full-feature route, we're looking at a minimum of about 200 staff-months to develop a market-leading product. We have the constraints of a maximum of 1 year to ship a product and a maximum team size of 8 people. We can't meet those constraints with the full-featured product. If we go the usability route, we're looking at more like 75 staff-months. That fits with our constraints, and there will be more room in the market for us."

"That's good work," Eddie said. "I think Rex will like that." Eddie met with Rex later that day, and then he got back together with George and Jill the following morning.

"Rex pointed out that we need to develop some in-house usability experts. He thought that developing a product that emphasizes usability was a good strategic move, so he gave us the thumbs-up."

"Now we need to plan the next iteration of the spiral. Our goal for this iteration is to refine the product spec in ways that minimize our development time and maximize usability."

George and Jill spent 4 weeks on the iteration, and then they met with Eddie to review their findings. "We've created a prioritized list of preliminary requirements," George reported. "The list is sorted by usability and then by estimated

(continued)

Case Study 7-2. Effective Lifecycle Model Selection, *continued*

implementation time. We've made both best-case and worst-case effort estimates for each feature. You can see that there's a lot of variation, and a lot of that variation just has to do with how we define the specifics of each feature. In other words, we have a lot of control over how much time this product takes to implement.

"Having maximum usability as our clear, primary objective really makes some decisions easy for us. Some of the most time-consuming features to implement would also be the least usable. I recommend that we just eliminate some of them because it will be a win for both the schedule and the product."

"That's interesting," Eddie responded. "What high-level alternatives have you come up with?"

"We recommend either of two possibilities," Jill said. "We've got the 'safe' version, which puts a strong emphasis on usability but uses proven technology. And we've got the 'risky' version, which pushes the usability state of the art. Either one should be a lot more usable than anything else on the market. The risky version will make it harder for the competition to catch up with us, but it will also nominally take about 60 staff-months compared with the safe version's 40 staff-months. That's not all that much of a difference, but the worst case for the risky version is 120 staff-months compared with the safe version's 55."

"Wow!" Eddie said. "That's good information. Is it possible to implement the safe version but design ahead so that we can push the state of the art in version 2?"

"I'm glad you asked that," Jill said. "We estimated that the safe version with design-ahead for version 2 would nominally take 45 staff-months, with a worst-case of 60."

"That makes it pretty clear, doesn't it?" Eddie said. "We've got 10½ months left, so let's do the safe version with design-ahead for version 2. While you all were focusing on the technical schedule risk, I've been focusing on the personnel schedule risk, and I've got three developers lined up. We'll add them to the team now and start the next iteration."

"George, you mentioned that a lot of the variation in schedule has to do with how each feature is ultimately defined, right? For the next spiral iteration, we need to focus on minimizing our design and implementation risk, and that means defining as many of those features to take as little implementation time as possible while staying consistent with our usability goal. I also want to have the new developers review your estimates to reduce the risk of any estimation error." George and Jill agreed.

The next iteration, which focused on design, took 3 months, bringing the project to the 4½-month mark. Their reviews had convinced them that their

(continued)

Case Study 7-2. Effective Lifecycle Model Selection, *continued*

design was solid—including the design-ahead for version 2. The design work had allowed them to refine their estimates, and they now estimated that the remaining implementation would take 30 staff-months, with a worst case of 40. Eddie thought that was exceptional because it meant that the worst case had them delivering the software only 2 weeks late.

At the beginning of the coding iteration, the developers identified low code quality and poor status visibility as their primary risks. To minimize those risks, they established code reviews to detect and correct coding errors, and they used miniature milestones to provide excellent status visibility.

Their estimates hadn't been perfect, and the final iteration took 2 weeks longer than nominal. They delivered the first release candidate to system testing at 11 months instead of at 10½. But the product's quality was excellent, and it took only two release candidates to declare a winner. Cube-It 1.0 was released on time.

Further Reading

DeGrace, Peter, and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions*. Englewood Cliffs, N.J.: Yourdon Press, 1990. The subtitle of this book is "A Catalog of Modern Software Engineering Paradigms," and it is by far the most complete description of software lifecycle models available. The book was produced through an unusual collaboration in which Peter DeGrace provided the useful technical content and Leslie Hulet Stahl provided the exceptionally readable and entertaining writing style.

Boehm, Barry W., ed. *Software Risk Management*. Washington, DC: IEEE Computer Society Press, 1989. This tutorial is interesting for the introduction to Section 4, "Implementing Risk Management." Boehm describes how to use the spiral model to decide which software lifecycle model to use. The volume also includes Boehm's papers, "A Spiral Model of Software Development and Enhancement" and "Applying Process Programming to the Spiral Model," which introduce the spiral lifecycle model and describe an extension to it (Boehm 1988; Boehm and Belz 1988).

Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Yourdon Press, 1994. Chapter 57, "Partial Life-Cycle Definitions" describes the hazards of not breaking down your lifecycle description into enough detail. It provides a summary of the 25 activities that Jones says make up most of the work on a successful software project.

8

Estimation

Contents

- 8.1 The Software-Estimation Story
- 8.2 Estimation-Process Overview
- 8.3 Size Estimation
- 8.4 Effort Estimation
- 8.5 Schedule Estimation
- 8.6 Ballpark Schedule Estimates
- 8.7 Estimate Refinement

Related Topics

- Scheduling: Chapter 9
- Measurement: Chapter 26
- 50/50 scheduling: Section 6.3



SOME ESTIMATES ARE CREATED CAREFULLY, and others are created by seat-of-the-pants guesses. Most projects overshoot their estimated schedules by anywhere from 25 to 100 percent, but a few organizations have achieved schedule-prediction accuracies to within 10 percent, and 5 percent is not unheard of (Jones 1994).

An accurate schedule estimate is part of the foundation of maximum development speed. Without an accurate schedule estimate, there is no foundation for effective planning and no support for rapid development. (See Case Study 8-1.)

This chapter provides an introduction to software project estimation. It describes how to come up with a useful estimate—how to crunch the numbers and create something reasonably accurate. Coming up with a perfect estimate doesn't do any good if you can't get the estimate accepted, so the next chapter describes how to handle the interpersonal elements involved in scheduling software projects.

Case Study 8-1. Seat-of-the-Pants Project Estimation

Carl had been put in charge of version 1 of Giga-Safe's inventory control system (ICS). He had a general idea of the capabilities desired when he attended the first meeting of the oversight committee for the project. Bill was the head of the oversight committee. "Carl, how long is ICS 1.0 going to take?" he asked.

"I think it will take about 9 months, but that's just a rough estimate at this point," Carl said.

"That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. We absolutely need to bring that system in within 6 months. Can you do it in 6?"

"I'm not sure," Carl said honestly. "I'd have to look at the project more carefully, but I can try to find a way to get it done in 6."

"Treat 6 months as a goal then," Bill said. "That's what it's got to be, anyway." The rest of the committee agreed.

By week 5, additional work on the product concept had convinced Carl that the project would take closer to his original 9-month guess than to 6 months, but he thought that with some luck he still might be able to complete it in 6. He didn't want to be branded a troublemaker, so he decided to sit tight.

Carl's team made steady progress, but requirements analysis took longer than they had hoped. They were now almost 4 months into what was supposed to be a 6-month project. "There's no way we can do the rest of the work we have to do in 2 months," he told Bill. He told Bill he needed a 2-month schedule slip and rescheduled the project to take 8 months.

A few weeks later, Carl realized that design wasn't proceeding as quickly as he had hoped either. "Implement the parts you can do easily," he told the team. "We'll worry about the rest of the parts when we get to them."

Carl met with the oversight committee. "We're now 7 months into our 8-month project. Detailed design is almost complete, and we're making good progress. But we can't complete the project in 8 months." Carl announced his second schedule slip, this time to 10 months. Bill grumbled and asked Carl to look for ways to bring the schedule back to around 8 months.

At the 9-month mark, the team had completed detailed design, but coding still hadn't begun on some modules. It was clear that Carl couldn't make the 10-month schedule either. He announced the third schedule slip number—to 12 months. Bill's face turned red when Carl announced the slip, and the pressure from him became more intense. Carl began to feel that his job was on the line.

Coding proceeded fairly well, but a few areas needed redesign and reimplementation. The team hadn't coordinated design details in those areas well,

(continued)

Case Study 8-1 . Seat-of-the-Pants Project Estimation, *continued*

and some of their implementations conflicted. At the 11-month oversight-committee meeting, Carl announced the fourth schedule slip—to 13 months. Bill became livid. “Do you have any idea what you’re doing?” he yelled. “You obviously don’t have any idea! You obviously don’t have any idea when the project is going to be done! I’ll tell you when it’s going to be done! It’s going to be done by the 13-month mark, or you’re going to be out of a job! I’m tired of being jerked around by you software guys! You and your team are going to work 60 hours a week until you deliver!” Carl felt his blood pressure rise, especially since Bill had backed him into an unrealistic schedule in the first place. But he knew that with four schedule slips under his belt, he had no credibility left. He felt that he had to knuckle under to the mandatory overtime or he would lose his job.

Carl told his team about the meeting. They worked hard and managed to deliver the software in just over 13 months. Additional implementation uncovered additional design flaws, but with everyone working 60 hours a week, they delivered the product through sweat and sheer willpower.

8.1 The Software-Estimation Story

Software estimation is difficult, and what some people try to do with software estimation isn't even theoretically possible. Upper management, lower management, customers, and some developers don't seem to understand why estimation is so hard. People who don't understand software estimation's inherent difficulties can play an unwitting role in making estimation even harder than it already is.

People remember stories better than they remember isolated facts, and there is a story to be told about why software estimation is hard. I think that we as developers need to make telling that story a priority. We need to be sure that customers and managers at all levels of our organizations have heard and understood it.

The basic software-estimation story is that software development is a process of gradual refinement. You begin with a fuzzy picture of what you want to build and then spend the rest of the project trying to bring that picture into clearer focus. Because the picture of the software you're trying to build is fuzzy, the estimate of the time and effort needed to build it is fuzzy, too. The estimate can come into focus only along with the software itself, which means that software-project estimation is also a process of gradual refinement.

The next several subsections describe the story in more detail.

Software and Construction

Suppose you go to your friend Stan, who's an architect, and say that you want to build a house. You start by asking Stan whether he can build a three-bedroom home for under \$100,000. He'll say yes, but he'll also say that the specific cost will vary depending on the detailed characteristics you want. (See Figure 8-1.)

If you're willing to accept whatever Stan designs, it will be possible for him to deliver on his estimate. But if you have specific ideas about what kind of house you want—if you later insist on a three-car garage, gourmet kitchen, sunroom, sauna, swimming pool, den, two fireplaces, gold-plated fixtures, floor-to-ceiling Italian marble, and a building site with the best view in the state—your home could cost several times \$100,000, even though the architect told you it was possible to build a three-bedroom home for under \$100,000.



*"A whole year to build
a house here?
No problem."*

*"Good. Let's get
started. I'm in
a hurry."*

Figure 8-1. It is difficult to know whether you can build the product that the customer wants in the desired time frame until you have a detailed understanding of what the customer wants.

It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of a subject admits, and not to seek exactness when only an approximation of the truth is possible...

Aristotle

How much does a new house cost? It depends on the house. How much does a new billing system cost? It depends on the billing system. Some organizations want cost estimates to within \pm 10 percent before they'll fund work on requirements definition. Although that degree of precision would be nice to have that early in the project, it isn't even theoretically possible. That early, you'll do well to estimate within a factor of 2.

Until each feature is understood in detail, you can't estimate the cost of a program precisely. Software development is a process of making increasingly detailed decisions. You refine the product concept into a statement of requirements, the requirements into a preliminary design, the preliminary design into a detailed design, and the detailed design into working code. At each of these stages, you make decisions that affect the project's ultimate cost and schedule. Because you can't know how each of these decisions will be made until you actually make them, uncertainty about the nature of the product contributes to uncertainty in the estimate.

Here are some examples of the kinds of questions that contribute to estimation uncertainty:

- Will the customer want Feature X?
- "Will the customer want the cheap or expensive version of Feature X? There is typically at least a factor of 10 difference in the implementation difficulty of different versions of the same feature.
- If you implement the cheap version of Feature X, will the customer later want the expensive version after all?
- How will Feature X be designed? There is typically at least a factor of 10 difference in the design complexity of different designs for the same feature.
- What will be the quality level of Feature X? Depending on the care taken during implementation, there can be a factor of 10 difference in the number of defects contained in the original implementation.
- How long will it take to debug and correct mistakes made in the implementation of Feature X? Individual performance among different programmers with the same level of experience has been found to vary by at least a factor of 10 in debugging and correcting the same problems.
- How long will it take to integrate Feature X with all the other features?

- Removing or turning off partially implemented features that can't be completed in time to ship the product
- Implementing quick-and-dirty versions of features that absolutely must be completed in time to ship the product
- Fixing low-priority defects
- Polishing help files and user documents by checking spelling, coordinating page numbers between different source files, inserting exact cross-references and online help jumps, creating indexes, taking final screen shots, and so on
- Performing end-to-end system tests of the entire product and formally entering defects into the defect-reporting system

I think of these activities as forcing the product to *converge*. When a project tries to force convergence too early, it will fail to converge, and then it has to do all of those time-consuming activities again later.

Doing activities twice when they could be done once is inefficient. But there are other time-wasting aspects of premature convergence, too. If software is released to testing before it is ready, testers will find many more defects than they would find if it were not released until it was ready. When testers find more defects, they enter the defects into a formal bug-tracking system, which adds overhead that takes both testing and development time. Debugging aids have to be turned back on. Removed features have to be put back in. Quick-and-dirty "ship mode" changes that aren't reliable or maintainable come back to haunt the developers. To repeat, premature convergence is a waste of time.

Perhaps worse for the project is the effect that premature convergence has on developer morale. If you're running a foot race, when the gun goes off for the last lap, you'll give it your all. You want to arrive at the finish line with nothing left. With premature convergence, the gun goes off, you give it your all, and just before you reach the finish line someone moves it. That wasn't the last lap after all, but you're left with nothing more to give. In the long run, pacing is important, and premature convergence burns out developers too early.

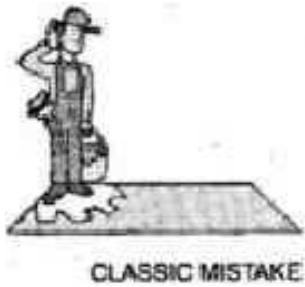
Poorly managed projects often discover their schedule problems for the first time when their developers aren't able to force convergence. Better managed projects detect schedule problems much earlier. Symptoms of premature convergence include:

- Developers can't seem to fix defects without tearing the system apart; small changes take longer than they should.

- Developers have long lists of "trivial" changes, which they know they need to make but which they haven't made yet.
- Testers find defects faster than developers can correct them.
- Defect fixes generate as many new defects as they correct.
- Tech writers have trouble completing user documentation because the software is changing too fast to be able to document it accurately.
- Project estimates are adjusted many times by similar amounts; the estimated release date stays 3 "weeks away for 6 months.

After you can't force convergence the first time, you'll need to back up, continue working, and try again to converge later. Optimistic schedules lead to premature and multiple attempts to converge, both of which lengthen schedules.

Excessive Schedule Pressure



Customers' and managers' first response when they discover they aren't meeting their optimistic schedule is to heap more schedule pressure onto the developers and to insist on more overtime. 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).

Schedule pressure has become so ingrained in the software-development landscape that many developers have accepted it as an unalterable fact of life. Some developers are no longer even aware that the extreme schedule pressure they experience could be otherwise. That is unfortunate. Overly optimistic scheduling hurts the real development schedule in many ways, but excessive schedule pressure hurts it the most, so I want to explore this particular problem in detail.



CROSS-REFERENCE

For more on error-prone modules, see "Error-prone modules" in Section 4.3.

Quality. About 40 percent of all software errors have been found to be caused by stress; those errors could have been avoided by scheduling appropriately and by placing no stress on the developers (Glass 1994c). When schedule pressure is extreme, about four times as many defects are reported in the released product as are reported for a product developed under less extreme pressure (Jones 1994). Excessive schedule pressure has also been found to be the most significant causative factor in the creation of extremely costly error-prone modules (Jones 1991).

With extreme schedule pressure, developers also increase the subtle pressure they put on themselves to focus on their own work rather than on

quality-assurance activities. Developers might still hold code reviews, for example, but when they are faced with a choice between spending an extra hour reviewing someone else's code or working on their own routines, the developers usually will choose to spend the extra hour on their own code. They'll promise themselves to do better on the code review next time. Thus, quality starts its downward slide.

CROSS-REFERENCE

For more on the relationship between defect-level and schedule, see Section 4.3, "Quality-Assurance Fundamentals."

CROSS-REFERENCE

For details on the occurrence of gambling on rapid-development projects, see Section 5.6, "Risk, High Risk, and Gambling."

CROSS-REFERENCE

For more on schedule pressure and motivation, see Chapter 11, "Motivation," and Section 43.1, "Using Voluntary Overtime."

Projects that aim from the beginning at having the lowest number of defects usually also have the shortest schedules. Projects that apply excessive schedule pressure and shortchange quality are rudely awakened when they discover that what they have really shortchanged is the schedule.

Gambling. Since an overly optimistic schedule is impossible to achieve through normal, efficient development practices, project managers and developers are provided with an incentive to gamble rather than to take calculated risks. "I doubt that the Gig-O-Matic CASE tool will really improve my productivity by a factor of 100, but I have absolutely no chance of meeting my schedule without it, so what do I have to lose?"

On a rapid-development project, you should be doing everything possible to reduce risk. Software projects require you to take calculated risks but not close-your-eyes-and-hope-that-it-works risks. Schedule pressure contributes to poor risk management and mistakes that slow development.

Motivation. Software developers like to work. A little bit of schedule pressure resulting from a slightly optimistic but achievable schedule can be motivating. But at some point the optimistic schedule crosses the threshold of believability, and at that point motivation drops—fast.

An overly optimistic schedule sets up developers to put in Herculean efforts only to be treated as failures for not achieving an impossible schedule—even when they have achieved, a schedule that is nothing short of remarkable. The developers know this, and unless they are very young or very naive, they will not work hard—they will not commit, they will not "sign up"—to achieve a schedule that is out of reach. Anyone who tries to motivate by forcing commitment to an unachievable schedule will achieve exactly the opposite of what is desired.

Creativity. Many aspects of software development—including product specification, design, and construction—require creative thought. Creativity requires hard thinking and persistence when the sought-after solution doesn't immediately spring to mind. The drive to think hard and persist requires internal motivation. Excessive external motivation (aka stress) reduces internal motivation and in turn reduces creativity (Glass 1994a).

Aside from reducing the incentive to be creative, a pressure-cooker environment is simply the wrong kind of environment for creative thought. The cogitation required for a breakthrough solution requires a relaxed, contemplative state of mind.



Given the same set of requirements, developers will create solutions that vary by as much as a factor of 10 in the amounts of code they require (Sackman, Erikson, and Grant 1968; Weinberg and Schulman 1974; Boehm, Gray, and Seewaldt 1984; De Marco and Lister 1989). If you're on a rapid-development schedule, you can't afford to create a pressure-cooker environment in which people are too rushed to find the solution that is one-tenth as expensive to implement as the others.

Burnout. If you use too much overtime in one project, your developers will more than make up for it on the next project. Programmers will putter around for months after putting in a big push on a major project—cleaning up their file systems, slowly commenting their source code, fixing low-priority bugs that they find interesting but which were not important enough to fix for the release (and may not be important enough to fix now), playing ping-pong, organizing their email, fine-tuning design documents, reading industry publications, and so on. If your schedule pushes developers too hard (perhaps by trying to force a premature convergence), you can experience that burnout on your current project rather than the next one.

Turnover. Overly optimistic schedules and the accompanying schedule pressure tend to cause extraordinarily high voluntary turnover, and the people who leave the project tend to be the most capable people with the highest performance appraisals (Jones 1991). Finding and training their replacements lengthens the schedule.

Long-term rapid development. Excessive overtime eliminates the free time that developers would otherwise spend on professional development. Developers who don't continue to grow don't learn about new practices, and that hurts your organization's long-term rapid-development capacity.

Relationship between developers and managers. Schedule pressure widens the gap between developers and managers. It feeds the existing tendency developers have to believe that management doesn't respect them, management doesn't care about them, and management doesn't know enough about software development to know when they're asking for something that's impossible. (See Figure 9-4.) Poor relationships lead to low morale, miscommunication, and other productivity sinkholes.



"If the book says that the shortest possible schedule is 6 months, you'll just have to work extra hard to finish in 4 months!"

Figure 9-4. Unreasonable schedule pressure can cause developers to lose respect for their managers.

The Bottom Line

Some people seem to think that software projects should be scheduled optimistically because software development should be more an adventure than a dreary engineering exercise. These people say that schedule pressure adds excitement.

How much sense does that make? If you were going on a *real* adventure, say a trip to the south pole by dogsled, would you let someone talk you into planning for it to take only 30 days when your best estimate was that it would take 60? Would you carry only 30 days' worth of food? Only 30 days' worth of fuel? Would you plan for your sled dogs to be worn out at the end of 30 days rather than 60? Doing any of those things would be self-destructive, and underscoping and underpinning a software project is similarly self-destructive, albeit usually without any life-threatening consequences.

In *Quality Software Management*, Gerald Weinberg suggests thinking about software projects as systems (Weinberg 1992). Each system takes in inputs and produces outputs. The system diagram for a project that's been accurately scheduled might look like the picture in Figure 9-5.

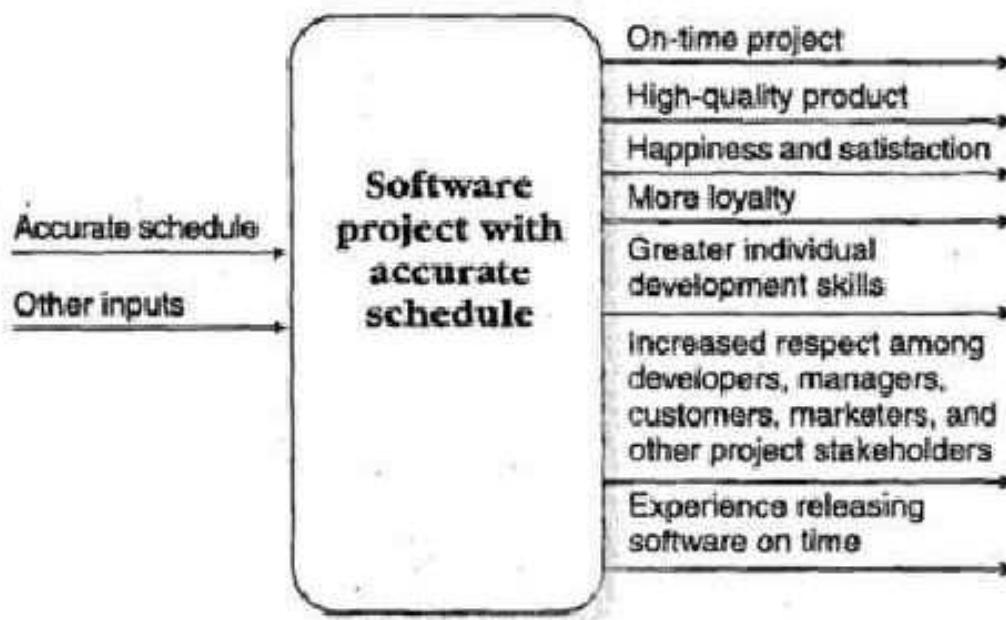


Figure 9-5. System diagram for a project with an accurate schedule. Most people will be happy with the outputs from an accurately-scheduled project.

The system diagram for a project that's been scheduled overly optimistically will, unfortunately, look more like the picture in Figure 9-6.

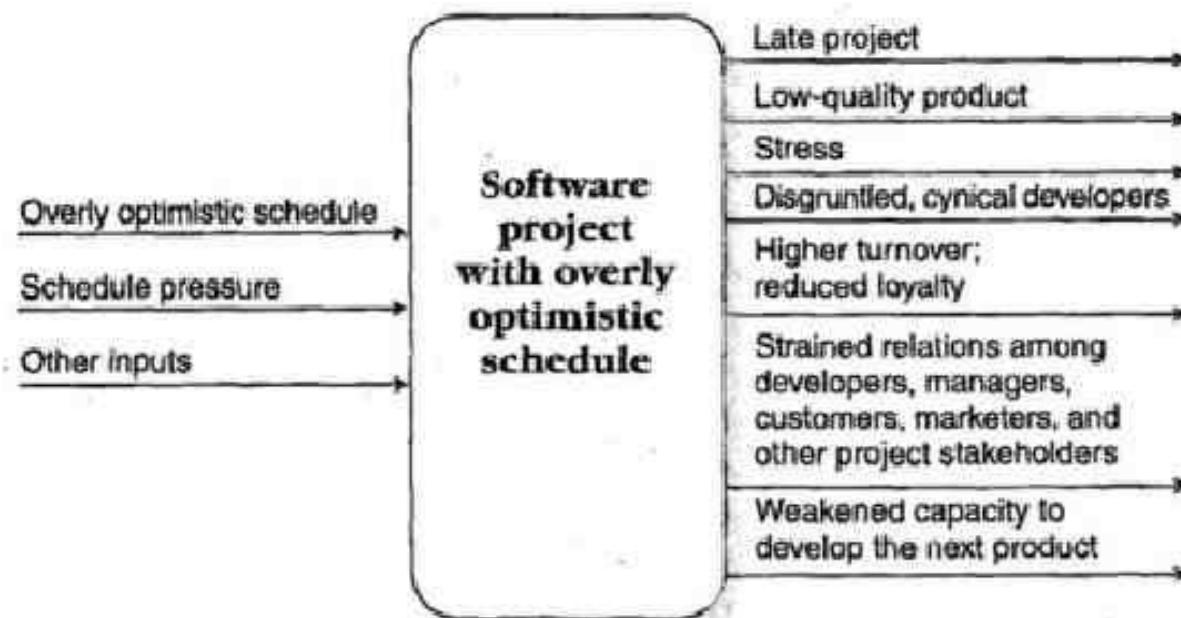


Figure 9-6. System diagram for a project with an overly optimistic schedule. Most people won't like the outputs from a project with an overly optimistic schedule.

When you compare the two systems, you can see that one is markedly healthier than the other.

In the end, I am opposed to the practice of overly optimistic scheduling because it undermines effective planning, eats into developers' personal lives, erodes relationships with customers, contributes to high turnover, contributes to low quality, and hurts the industry by stunting professional growth and creating the perception that software developers can't deliver what they promise.

If you are not outraged, you are not paying attention.

Bumper sticker

But I am most opposed to overly optimistic scheduling because it doesn't work, It doesn't result in shorter actual schedules; it results in longer ones. That's what happened with WinWord 1.0. That's what happened with the EAA's Advanced Automation System. That's what happened with every other project I know of that has tried to work to an impossible schedule.

The shortest actual schedule results from the most accurate planned schedule. A project that has schedule problems needs to look no further for the source of its problems than an unachievable initial schedule.

9.2 Beating Schedule Pressure

Schedule pressure appears to be endemic to software development, and it has produced damaging, short-term thinking on two levels. At a local level, it has encouraged shortcut-taking on specific projects, which damages those specific projects. At a more global level, it has encouraged a fire-fighting mentality about schedule pressure itself. People view schedule pressure as a problem unique to their current project, even though they've felt schedule pressure on every project they've ever worked on and even though it has been one of the defining characteristics of the software industry for at least 30 years.

Ironically, we can't solve the problem of rapid development until we solve the problem of schedule pressure. As Figure 9-7 shows, schedule pressure creates a vicious circle of more stress, more mistakes, more schedule slips, and ever more schedule pressure.

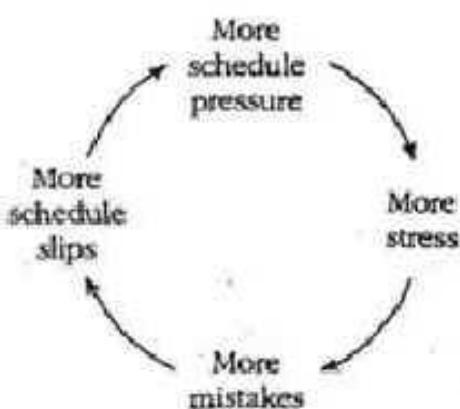


Figure 9-7. Vicious circle of schedule-pressure and schedule-slips. Anyone who wants to solve the problem of rapid development must first solve the problem of excessive schedule pressure. •

We as an industry need to learn how to beat schedule pressure. There cannot be a long-term solution to the schedule-pressure problem until we take time out to learn how to do our jobs better.

Three factors converge to make up the bulk of the problems associated with setting software schedules:

- *Wishful thinking*—Customers, managers, and end-users naturally and rationally want to get as much as they can for their money, and they want to get it as soon as possible. Most software project schedules are ambitious. Think about that. Most aren't average; most are ambitious. The previous section should provide all the reasons anyone needs to abandon their wishful thinking about software schedules.
- *Little awareness of the software estimation story or the real effects of overly optimistic scheduling*—Software can't be reliably estimated in its early stages. It's logically impossible. Yet we let people force us into unrealistic estimates. The estimation story described in Section 8.1 should help with this problem.
- *Poor negotiating skills*—Philip Metzger observed 15 years ago that developers were fairly good at estimating but were poor at defending their estimates (Metzger 1981). I haven't seen any evidence that developers have gotten any better at defending their estimates in recent years.

Developers tend to be bad negotiators for a few reasons.

CROSS-REFERENCE

For more on the profile of the

Section 11.1, "Typical Developer Motivations."

First, developers are, as a rule, introverts. About three-quarters of developers are introverts whereas only one-third of the general population would be described as such. Most developers get along with other people just fine, but challenging social interactions are not their strong suit.

Second, software schedules are typically set in negotiations between development and management or development and marketing. Gerald Weinberg points out that marketers tend to be 10 years older and negotiate for a living—that is, they tend to be seasoned, professional negotiators (Weinberg 1994). The deck is stacked against developers during schedule negotiations.

Third, developers tend to be temperamentally opposed to negotiating tricks. Such tricks offend their sense of technical accuracy and fairness. Developers won't offer lopsidedly high initial estimates even when they know that customers, marketers, or managers will start with lopsidedly low initial bargaining positions.

I have become convinced that developers need to become better negotiators, and I'll spend the rest of this chapter describing how to negotiate schedules effectively.

Why Schedule Negotiations Are Difficult

"False scheduling to match the patron's desired date is much more common in our discipline than elsewhere in engineering. It is very difficult to make a vigorous, plausible, and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers."

—Fred Brooks

Principled Negotiation



A good place to start improving your negotiating skills is the principled negotiation method described in *Getting to Yes* (Fisher and Ury 1981). This method has several characteristics that I find appealing. It doesn't rely on negotiating tricks, but it explains how to respond to tricks when others use them. It's based on the idea of creating win-win alternatives. You don't try to beat the person you're negotiating with; you try to cooperate so that both of you can win. It's an open strategy. You don't have to fear that the person you're negotiating with has read the same negotiating book and knows the same tricks. The method works best when all the parties involved know about it and use it.

CROSS-REFERENCE

For a related win-win strategy, see Chapter 37, "Theory-W Management."

The principled-negotiation strategy consists of four parts that deal with people, interests, options, and criteria:

- Separate the people from the problem
- Focus on interests, not positions
- Invent options for mutual gain
- Insist on using objective criteria

Each of these is described in the following sections.

Separate the People from the Problem

All negotiations involve people first, interests and positions second. When the negotiators' personalities are at odds—as, for example, developers' and marketers' personalities often are—negotiations can get hung up on personality differences.

Begin by understanding the other side's position. I've had cases in which a non-technical manager had good business reasons for wanting a specific deadline. In one case, a manager felt pressure from the marketing organization and his boss to produce what was probably a 15-month project in 6

CROSS-REFERENCE

People's expectations can affect negotiations. For more on expectations, see Section 10.3, "Managing Customer Expectations."

months. He told me that he had to have the software in 6 months. I told him that the best I could do was 15 months. He said, "I'm not giving you a choice. Our customers are expecting the software in 6 months." I said, "I'm sorry. I wish I could. But 15 months is the best I can do." He just froze and stared at me for 2 or 3 minutes.

Why did he freeze? Was he using silence as a negotiating maneuver? Maybe. But I think it was because he felt trapped and powerless. He had promised his boss a 6-month development schedule, and now the person who was supposed to head the project was telling him he couldn't keep his promise.

Understand that managers can be trapped by their organization's outdated policies. Some organizations fund software projects in ways that are essentially incompatible with the way software is developed. They don't allow managers to ask for funding just to develop the product concept and come up with a good cost estimate. To get enough funding to do a decent estimate, managers have to get funding for the whole project. By the time they get a decent estimate, it can be embarrassing or even career-threatening to go back and ask for the right amount of funding. People at the highest levels of such organizations need to hear the software-estimation story so that they can institute sensible funding practices.

Most middle managers aren't stupid or irrational when they insist on a deadline that you know is impossible. They simply don't know enough about the technical work to know that it's impossible, or they know all too well how much pressure they feel from their own bosses, customers, or people higher up in the organization.

CROSS-REFERENCE
For more on the software estimation story, see Section 8.1, 'The Software-Estimation Story.'

What can you do? Work to improve the relationship with your manager or customer. Be cooperative. Work to set realistic expectations. Be sure that everyone understands the software-estimation story. Be an advisor on schedule matters, and avoid slipping into the role of adversary. Suggest ways to change the project that will reduce the schedule, but hold firm to not just writing down a different date.

It's also useful to try to take emotions out of the negotiating equation. Sometimes the easiest way to do that is to let the other people blow off steam. Don't react emotionally to their emotions. Invite them to express themselves fully. Say something like, "I can see that those are all serious concerns, and I want to be sure I understand your position. What else can you tell me about your situation?" When they are done, acknowledge their emotions and reiterate your commitment to find a win-win solution. The other parts of principled negotiation will help you to follow through on that commitment.

Focus on Interests, Not Positions

Suppose you're selling your car in order to buy a new boat, and you've figured that you need to get \$5000 for your car in order to buy the boat you want. A prospective buyer approaches you and offers \$4500. You say, "There's no way I can part with this car for less than \$5000." The buyer says, "\$4500 is my final offer."

When you negotiate in this way, you focus on positions rather than interests. Positions are bargaining statements that are so narrow that in order for one person to win, the other person has to lose.

Now suppose that the car buyer says, "I really can't go over \$4500, but I happen to know that you're in the market for a new boat, and I happen to be the regional distributor for a big boat company. I can get the boat you want for \$1000 less than you can get it from any dealer. Now what do you think about my offer?" Well, now the offer sounds pretty good because it will leave you with \$500 more than you would have gotten if the buyer had just agreed to your price.

Underlying interests are broader than bargaining positions, and focusing on them opens up a world of negotiating possibilities. Your boss might start out by saying, "I need Giga-Blat 4.0 in 6 months," and you might know immediately that you can't deliver it in less than 9 months. Your boss's interest might be keeping a promise made to the sales organization, and your interest might be working less than 60 hours a week for the next 6 months. Between the two of you, you might be able to create a product that would satisfy the sales organization and would be deliverable within 6 months. If you focus on interests, you're more likely to find a win-win solution than if you dig into bargaining positions.

One of the major problems with schedule negotiations is that they tend to become one-dimensional, focusing only on the schedule. Don't get dug into a position. Make it clear that you're willing to consider a full-range of alternatives—just not pie-in-the-sky options. If other people have dug themselves into demanding a specific schedule, here are some points you can use to dislodge them:

Appeal to true development speed. Point out that the worst fault of overly optimistic schedules is that they undermine actual development speed. Explain the negative effects of overly optimistic scheduling that were described in Section 9.1. True rapid development requires that you be firmly connected to reality, including to a realistic schedule.

Appeal to increasing the chance of success. Point out that you have estimated the most likely completion date and that you already have only a 50/50 chance of meeting that. Shortening the schedule will further reduce your chances of completing on time. ."

Invoke your organization's track record. Point to your organization's history of underestimating projects, coming in late, and all the problems that lateness has caused. Appeal to the other person's good sense not to do the same thing again.

Invent Options for Mutual Gain

CROSS-REFERENCE

For more on the value of cooperation, see "Cooperation" in Section 8.1.

Rather than thinking of negotiating as a zero-sum game in which one person wins at the other's expense, think of it as an exercise in creative problem-solving; the truly clever negotiator will find a way for both parties to win.

Your most powerful negotiating ally in schedule negotiations is your ability to generate options that the other person has no way of knowing about. You hold the key to a vault of technical knowledge, and that puts the responsibility for generating creative solutions more on your shoulders than on the nontechnical person you're negotiating with. It's your role to explain the full range of possibilities and trade-offs.

CROSS-REFERENCE

For details on the schedule, cost, and product triangle, see "Schedule, Cost, and Product Trade-offs" in Section 6.6.

I find it useful to think about how many degrees of freedom there are in planning a software project. The basic degrees of freedom are defined by the schedule, cost, and product triangle. You have to keep the three corners in balance for the project to succeed. But there are infinite variations on that triangle, and the person you're negotiating with might find some of those variations to be a lot more appealing than others. Here are some of the degrees of freedom you might suggest related to the product itself:

- Move some of the desired functionality into version 2. Few people need all of what they asked for exactly when they asked for it.
- Deliver the product in stages—for example, versions 0.7, 0.8, 0.9, and 1.0—with the most important functionality coming first.
- Cut features altogether. Features that are time-consuming to implement and often negotiable include the level of integration with other systems, level of compatibility with previous systems, and performance.
- Polish some features less—implement them to some degree, but make them less fancy.
- Relax the detailed requirements for each feature. Define your mission as getting as close as possible to the requirements through the use of prebuilt commercial components.

Here are some degrees of freedom related to project resources:

- Add more developers, if it's early in the schedule.
- Add higher-output developers (for example, subject-area experts).
- Add more testers.
- Add more administrative support.

- Increase the degree of developer support. Get quieter, more private offices, faster computers, on-site technicians for network and machine support, approval to use higher priced developer-support services, and so on.
- Eliminate company red tape. Set your project up as a skunkworks project.
- Increase the level of end-user involvement. Devote a full-time end-user to the project who is authorized to make binding decisions about the product's feature set.
- Increase the level of executive involvement. If you've been trying to introduce JAD sessions to your organization but haven't been able to get the executive sponsorship you need, this is a good time to ask for it.

Here are some degrees of freedom you can suggest related to the project's schedule:

- Set a schedule *goal* but not an ultimate deadline for the whole project until you've completed the detailed design, product design, or at least the requirements specification.
- If it's early in the project, agree to look for ways to reduce the development time as you refine the product concept, specification, and design.
- Agree to use estimation ranges or coarse estimates and to refine them as the project progresses.

You can propose a few additional degrees of freedom in certain circumstances. They can make a difference in development time, but they also tend to be political hot potatoes. Don't bring them up unless you know the person on the other side of the table is already sympathetic to your cause.

- Provide exceptional developer support so that developers can focus more easily on the project—shopping service, catered meals, laundry, housecleaning, lawn care, and so on.
- Provide increased developer motivation—paid overtime, guarantee of comp time, profit sharing, all-expenses-paid trips to Hawaii, and so on.

Whatever you do, don't agree to a lopsided schedule-cost-product triangle. Remember that once you've settled on a feature set, the size of the triangle is practically a law of physics—the three corners have to be in balance.

Throughout the negotiations, focus on what you can do and avoid getting stuck on what you can't. If you're given an impossible combination of feature set, resources, and schedule, say, "I can deliver the -whole feature set with my current team 4 weeks later than you want it. Or I could add a

person to the team and deliver the whole feature set when you want it. Or I can cut features X, Y, and Z and deliver the rest with my current team by the time you want it."

The key is to take attention away from a shouting match like this one: *I can't do it.* "Yes you can." *No I can't.* "Can!" *Can't!* Lay out a set of options, and focus your discussion on what you can do.

One warning: In the cooperative, brainstorming atmosphere that arises from this kind of free-wheeling discussion, it's easy to agree to a solution that seems like a good idea at the time but by the next morning seems like a bad deal. Don't make any hard commitments to new options until you've had enough time to analyze them quietly by yourself.

Insist on Using Objective Criteria

The ultimate act of disempowerment is to take away the responsibility for the schedule from those who must live by It.

Jim McCarthy

One of the oddest aspects of our business is that when careful estimation produces estimates that are notably longer than desired, the customer or manager will often simply disregard the estimate (Jones 1994). They'll do that even when the estimate comes from an estimation tool or an outside estimation expert—and even when the organization has a history of overrunning its estimates. Questioning an estimate is a valid and useful practice. Throwing it out the window and replacing it with wishful thinking is not.

A key to breaking deadlocks with principled negotiations is the use of objective criteria. The alternative is to break negotiation deadlocks based on whoever has the most willpower. I've seen schedules for million-dollar projects decided on the basis of which person could stare the longest without blinking. Most organizations will be better off using a more principled approach.

In principled negotiation, when you reach a deadlock, you search for objective criteria you can use to break the deadlock. You reason with the other people about which criteria are most appropriate, and you keep an open mind about criteria they suggest. Most important, you don't yield to pressure, only to principle.

Here are some guidelines for keeping schedule negotiations focused on principles and not just desires.

Don't negotiate the estimate itself. You can negotiate the inputs to the estimate—the degrees of freedom described in the previous section—but not the estimate itself. As Figure 9-8 suggests, treat the estimate as something that's determined from those inputs almost by a law of nature. Be extremely open to changing the inputs and be ready to offer alternatives, but match your flexibility in those areas with a firm stance on the estimate itself.

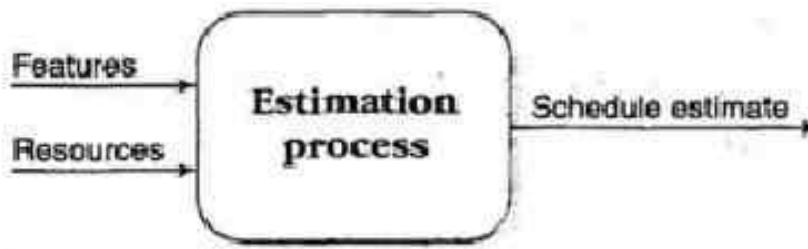


Figure 9-8. *Treating an estimate as something determined by a law of nature. You can negotiate the inputs, but you can't change the output without changing the inputs.*

Suppose you're negotiating with an in-house customer. You could say something like this: "This is my best estimate. I can write a different date on a piece of paper, but that won't be a valid estimate. I can write a bigger number in my checkbook, too, but that doesn't mean that I'll have any more money. I've already given the team only about a 50-percent chance of delivering the software on time. Planning for a shorter schedule won't shorten the amount of time it will take them to create the product you want. It will just increase the risk of being late."

Point out that by refusing to accept an impossible deadline you're really looking out for your customer's best interests. Point to your organization's history of schedule overruns, and tell the in-house customer that you're unwilling to set either of you up for failure. It's easy to make this argument once you've demonstrated your willingness to look for win-win solutions.



FURTHER READING

For an illustration of using a software-estimating tool as an impartial expert, see Section 2.3.2 of "Theory-W Software Project Management: Principles and Examples" (Boehm and Ross 1989).

Insist that the estimate be prepared by a qualified party. Sometimes negotiations produce the absurd situation in which customers who have no idea how to build a software system nevertheless claim to know how long it will take to build it. Insist that the estimate be prepared by someone with appropriate qualifications. That will often be you.

Some organizations have had good success creating an independent estimation group. Those groups are effective because they do not have a vested interest either in delivering the product in the shortest possible time or in avoiding hard work. If negotiations become deadlocked on the topic of the estimate, propose submitting the estimate to a third party and pledge to accept their results. Ask your negotiation foe to pledge to do the same.

A variation on this theme is to bring in a consultant or noted authority to review your schedule. (An unfamiliar expert sometimes has more credibility than a familiar one.) Some organizations have also had success using software-estimation tools. They've found that once developers calibrate the estimation tool for a specific project; the tool allows them to easily and objectively explore the effects of different options in an unbiased way.

Insist on a rational estimation procedure. Chapter 8, "Estimation," explains the basis for creating software project estimates. Insist that the procedure that is used to create the estimate comply with the following guidelines:

- *Nails down features before it nails down the estimate.* You can't know how much a new house costs until you know quite a lot about the house. You can't tell how much a new software system will cost until you know quite a lot about the system.
- *Doesn't provide unrealistic precision.* Provide your estimates in ranges that become increasingly refined as the project progresses.
- *Reestimates after changes.* The estimate should be the last step in the process. It's irrational to create an estimate, pile on more features, and keep the estimate the same.

Don't bow to the pressure to commit to impossible deadlines. That short-term fix damages your long-term credibility. No one really benefits from pretending that you can meet an impossible schedule, even though sometimes people think they do. Improve your credibility by pushing for solutions that respond to the real business needs of your bosses and customers.

In reality, you don't need permission to do your job well.
Larry Constantine

Weather the storm. Although people have different tolerances for withstanding pressure, if your customers, managers, or marketers want you to change your estimate without also changing the feature set or resources, I think the best approach is to politely and firmly stand by your estimate. Batten down the hatches and endure the thunderstorm of an unwelcome estimate early in the project rather than the hurricane of schedule slips and cost overruns later on.

Case Study 9-1. A Successful Schedule Negotiation

Tina's team had put a lot of work into their estimate for the Giga-Bill 1.0 project, which they had estimated would probably take 12 months. Her boss, Bill, wasn't happy with the estimate the team came up with. He said it needed to be shorter. Tina found herself sitting across from Bill at the oversight committee meeting.

"The team has estimated it can deliver the product in 6 months," Bill said.

"Err—ahem." Tina cleared her throat. "What Bill means is that we estimated an ideal-world, best case of 6 months, but in order to achieve that best case, every single thing on the project has to go perfectly. And you know software projects—nothing ever goes perfectly. Our most likely estimate is 12 months, with a realistic range of 10 to 15 months." Tina was sweating and wished she had a handkerchief to wipe her forehead.

(continued)

CROSS-REFERENCE

For more on team structures,
see Chapter 13, "Team
Structure."

Results-Driven Structure

You can structure teams for optimal output, or you can structure them in such a way that it is almost impossible for them to produce anything at all.

For rapid development, you need to structure the team with maximum development speed in mind. You don't put John in charge just because he's the owner's cousin, and you don't use a chief-programmer team structure on a three-person project when the three people have roughly equal skills.

Here are four essential characteristics of a results-driven team structure:

- Roles must be clear, and everyone must be accountable for their work at all times. Accountability is critical to effective decision making and to rapid execution after the decisions have been made.
- The team must have an effective communication system that supports the free flow of information among team members. Communication must flow freely both from and to the team's management.
- The team must have some means of monitoring individual performance and providing feedback. The team should know whom to reward, who needs individual development, and who can assume more responsibilities in the future.
- Decisions must be made based on facts rather than on subjective opinions whenever possible. The team needs to be sure that the facts are interpreted without biases that undercut the functioning of the team.

There are practically an infinite number of team structures that can satisfy these essential characteristics. It is amazing that so many team structures do not.

Competent Team Members

Just as team structures are chosen for the wrong reasons, team members are often chosen for the wrong reasons: for example, they are often chosen because they have an interest in the project, because they are cheap, or most often simply because they are available. They are not chosen with rapid development in mind. Case Study 12-3 describes the way in which team members are typically selected.

Case Study 12-3. Typical Team-Member Selection

Bill had a new application to build, and he needed to put a team together fast. The project was supposed to take about 6 months and was going to involve a lot of custom graphics, and the team would have to work closely with the

(continued)

Case Study 12-3. Typical Team-Member Selection, *continued*

customer. It should take about four developers. Ideally, Bill thought, he'd like to get Juan, who had worked on GUI custom graphics on that platform before, and Sue, who was a database guru and great with customers. But they were both busy on other projects for the next 2 or 3 weeks.

At the manager's meeting Bill found out that Tomas, Jennifer, Carl, and Angela would be available at the end of the week. "They'll do OK," he said. "That will let us get started right away."

He planned the project this way. "Tomas can work on the graphics. He hasn't worked on this platform before, but he's done some graphics work. Jennifer would be good for the database side. She said she was tired of working on databases, but she agreed to work on them again if we really needed her to. Carl's done some work on this platform before, so he could lend Tomas a hand with the graphics. And Angela is an expert in the programming language. Carl, Angela, and Tomas have had a few problems working together before, but I think they've put their differences behind them. None of them are particularly strong in working with customers, but I can fill in that gap myself."

Case Study 12-3 describes a team that's selected based on who's available at exactly the right time without much concern for the long-term performance consequences. It's almost certain that the team would do better on its 6-month project if it waited the 3 weeks until Juan and Sue were available.

For rapid development, team members need to be chosen based on who has the competencies that are currently needed. Three kinds of competencies are important:

- Specific technical skills—application area, platform, methodologies, and programming language
- A strong desire to contribute
- Specific collaboration skills required to work effectively with others

Mix of Roles

On an effective team, the team members have a mix of skills and they play several different roles. It obviously doesn't make sense to have a team of seven people who are all experts in assembly language if your project is in C++. Likewise, it doesn't make sense to have seven people who are all experts in C++ if no one of them knows the applications area. Less obviously, you need team members who have a blend of technical, business, management, and interpersonal skills. In rapid development, you need interpersonal leaders as much as technical leaders.



FURTHER READING

These labels are not Belbin's but are taken from Constantino on Peopleware (Constantine 1995a).

Dr. Meredith Belbin identified the following leadership roles:

- *Driver*'—Controls team direction at a detailed, tactical level. Defines things, steers and shapes group discussions and activities.
- *Coordinator*—Controls team direction at the highest, strategic level. Moves the problem-solving forward by recognizing strengths and weaknesses and making the best use of human and other resources.
- *Originator*—Provides leadership in ideas, innovating and inventing ideas and strategies, especially on major issues.
- *Monitor*—Analyzes problems from a practical point of view and evaluates ideas and suggestions so that the team can make balanced decisions.
- *Implementer*—Converts concepts and plans into work procedures and carries out group plans efficiently and as agreed.
- *Supporter*—Builds on team members' strengths and underpins their shortcomings. Provides emotional leadership and fosters team spirit. Improves communications among team members.
- *Investigator*—Explores and reports on ideas, developments, and resources outside the group. Creates external contacts that may be useful to the group.
- *Finisher*—Ensures that all necessary work is completed in all details. Seeks work that needs greater than average attention to detail, and maintains the group's focus and sense of urgency.

Even on a rapid-development project, it's best not to staff a project with nothing but high-performance individuals. You also need people who will look out for the organization's larger interests, people who will keep the high-performance individuals from clashing, people who will provide technical vision, and people who will do all the detail work necessary to carry out the vision.

One symptom of a team that isn't working is that people are rigid about the roles they will and won't play. One person will do database programming only and won't work on report formatting. Or another person will program only in C++ and won't have anything to do with Visual Basic.

On a well-oiled team, different people will be willing to play different roles at different times, depending on what the team needs. A person who normally concentrates on user-interface work might switch to database work if there are two other user-interface experts on the team. Or a person who usually plays a technical-lead role may volunteer to play a participant role if there are too many leaders on a particular project.

CROSS-REFERENCE

For more on commitment to a project, see Chapter 34, "Signing Up."

Commitment to the Team

The characteristics of vision, challenge, and team identity coalesce in the area of commitment. On an effective team, team members commit to the team. They make personal sacrifices for the team that they would not make for the larger organization. In some instances, they may make sacrifices to the team to spite the larger organization, to prove that they know something that the larger organization doesn't. In any case, the minimum requirement for team success is that the team members contribute their time and energy—their effort—and that calls for commitment.

When team members commit, there must be something for them to commit to. You can't commit to unstated goals. You can't commit at any deep level to "doing whatever management wants." Vision, challenge, and team identity provide the things to which team members commit.

Getting project members to commit to a project is not as hard as it might sound. IBM found that many developers were eager for the opportunity to do something extraordinary in their work. They found that simply by asking and giving people the option to accept or decline, they got project members to make extraordinary commitments (Scherr 1989).

Mutual Trust

Larson and LaFasto found that trust consisted of four components:

- Honesty
- Openness
- Consistency
- Respect

If any one of these elements is breached, even once, trust is broken.

Trust is less a cause than an effect of an effective team. You can't force the members of a team to trust each other. You can't set a goal of "Trust your teammates." But once project members commit to a common vision and start to identify with the team, they learn to be accountable and to hold each other accountable. When team members see that other team members truly have the team's interests at heart—and realize that they have a track record of being honest, open, consistent, and respectful with each other—trust will arise from that.

Interdependence Among Members

Team members rely on each other's individual strengths, and they all do what's best for the team. Everybody feels that they have a chance to contribute and that their contributions matter. Everybody participates in decisions. In short, the team members become interdependent. Members of healthy teams sometimes look for ways they can become dependent on other team members. "I could do this myself, but Joe is especially good at debugging assembly language code. I'll wait until he comes back from lunch and then ask him for help."

On the most effective one-project teams that I've been on, the beginning of the project is characterized by an unusual measure of tentativeness. Team members might feel that they have specific strengths to offer the team, but they are not pushy about asserting their rights to occupy specific roles. Through a series of tacit negotiations, team members gradually take on roles that are not just best for them individually but that are best for the team as a whole. In this way, everyone gravitates toward productive positions, and no one feels left out.

Effective Communication

CROSS-REFERENCE

For more on the role of communication in teamwork, see "Effective communication" in Section 13.1.

Members of cohesive teams stay in touch with each other constantly. They are careful to see that everyone understands when they speak, and their communication is aided by the fact that they share a common vision and sense of identity. Amish barn raisers communicate efficiently during a barn raising because they live in a tight-knit community and nearly all of them have been through barn raisings before. They are able to communicate precise meanings with a few words or gestures because they have already established a baseline of mutual understanding.

Team members express what they are truly feeling, even when it's uncomfortable. Sometimes team members have to present bad news. "My part of the project is going to take 2 weeks longer than I originally estimated." In an environment characterized by interdependence and trust, project members can broach uncomfortable subjects when they first notice them, when there's still time to take effective corrective action. The alternative is covering up mistakes until they become too serious to overlook, which is deadly to a rapid-development effort.

Sense of Autonomy

Effective teams have a sense that they are free to do whatever is necessary

without interference. The team might make a few mistakes—but the motivational benefit will more than offset the mistakes.

This sense of autonomy is related to the level of trust they feel from their manager. It is imperative that the manager trust the team. That means not micromanaging the team, second-guessing it, or overriding it on tough decisions. Any manager will support a team when the team is clearly right—but that's not trust. When a manager supports the team when it looks like it might be wrong—*that's trust*.

Sense of Empowerment

An effective team needs to feel empowered to take whatever actions are needed to succeed. The organization doesn't merely allow them to do what they think is right, it supports them in doing it. An empowered team knows that it can, as they say at Apple Computer, *push back* against the organization when it feels the organization is asking for something unreasonable or is headed in the wrong direction.

One common way that teams are denied empowerment is in the purchase of minor items they need to be effective. I worked on an aerospace project in which it took 6 months to get the approval to buy two scientific hand-held calculators. This was on a project whose mission was to analyze scientific data!

As Robert Townsend says, "Don't underestimate the morale value of letting your people 'waste' some money" (Townsend 1970). The most extreme example I know of was an episode during the development of Windows 95. To ensure that Windows 95 worked well with every program, the project manager and the rest of the team headed over to the local software store and loaded up a pickup truck with one of every kind of program available. The total tab was about \$15,000, but the project manager said that the benefit to morale was unbelievable. (The benefit to morale at the software store wasn't bad, either.)

Small Team Size

Some experts say that you must have fewer than 8 to 10 people for a team to jell (Emery and Emery 1975, Bayer and Highsriith 1994). If you can keep the group to that size, do so. If your project requires you to have more than 10 project members, try to break the project into multiple teams, each of which has 10 or fewer members.

The 10-person limitation applies mainly to single-project teams. If you can keep a team together across several projects, you can expand the size of the team as long as the team shares a deep-rooted culture. The Amish farmers formed a cohesive team of several dozen people, but they had been together for generations.

On the other end of the scale, it is possible for a group to be too small to form a team. Emery and Emery point out that with less than four members, a group has a hard time forming a group identity, and the group will be dominated by interpersonal relationships rather than a sense of group responsibility (Emery and Emery 1975).

High Level of Enjoyment

CROSS-REFERENCE

For more on what motivates developers, see Section 11.1, "Typical Developer Motivations."

Not every enjoyable team is productive, but most productive teams are enjoyable. There are several reasons for this. First, developers like to be productive. If their team supports their desire to be productive, they enjoy that. Second, people naturally spend more time doing things that they enjoy than doing things that they don't enjoy, and if they spend more time at it, they'll get more done. Third, part of what makes a team jell is adopting a group sense of humor. DeMarco and Lister describe a jelled group in which all the members thought that chickens and lips were funny (DeMarco and Lister 1987). Chickens with lips were especially funny. The group actually rejected a well-qualified candidate because they didn't think he would find chickens with lips amusing. I don't happen to think that chickens with lips are funny, but I know what DeMarco and Lister are talking about. One group I was a part of thought that cream soda was hilarious and another thought that grape Lifesavers were a riot. There's nothing intrinsically comical about cream soda or grape Lifesavers, but those jokes were part of what gave those teams their identities. I haven't personally seen a cohesive team that didn't have a keen sense of humor. That might just be a quirk of my specific experience, but I don't think so.

How to Manage a High-Performance Team

CROSS-REFERENCE

For the difference between managers and team leaders, see Section 13.3, "Managers and Technical Leads."



FURTHER READING

For excellent discussions of each of these points, see *Quality Software Management, Volume 3: Congruent Action* (Weinberg 1994).

A cohesive team creates an "us" and the manager is in the sticky position of being not completely "us" and not completely "them." Some managers find that kind of team unity threatening. Other managers find it exhilarating. By taking on a great deal of autonomy and responsibility, a high-performance team can relieve a manager of many of the usual management duties.

Here are some keys to success in managing a cohesive team:

- Establish a vision. The vision is all-important, and it is up to the manager and team leader to put it into play.
- Create change. The manager recognizes that there is a difference between the way things should be and the way they are now. Realize that the vision requires change, and make the change happen.
- Manage the team as a team. Make the team responsible for its actions rather than making individuals on the team responsible for their individual actions. Team members often set higher standards for themselves than their leaders do (Larson and LaFasto 1989).

- Delegate tasks to the team in a way that is challenging, clear, and supportive. Unleash the energy and talents of the team members.
- Leave details of how to do the task to the team, possibly including the assignment of individual work responsibilities.
- When a team isn't functioning well, think about the MOI model, which states that most team problems arise from Motivation, Organization, or Information. Try to remove roadblocks related to these three factors.

12.4 Why Teams Fail



FURTHER READING

For an excellent discussion of team failure, see Chapter 20, "Teamicide," in /%op/eware (DeMarco and Lister 1987).

The cohesiveness of a group depends on the total field of forces that act on that group. As with other aspects of rapid development, you have to do a lot of things right to succeed, but you only have to do one thing wrong to fail. Teams don't need to have all the characteristics described in the previous section, but they do need to have most of them.

Teams can fail for any of the reasons listed in Section 11.4, "Morale Killers." Those morale killers can keep a team from jelling just as easily as they can undercut individual morale.

Here are some other reasons that teams fail.

Lack of common vision. Teams rarely form without a common vision. Organizations sometimes prevent teams from forming by undercutting their visions. A team might form around the vision of producing "the best word processor in the world." That vision takes a beating if the organization later decides that the word processor doesn't have to be world class, but it does have to be completed within the next 3 months. When the vision takes a beating, the team takes a beating too.

Lack of identity. Teams can fail because they don't establish a team identity. The team members might be willing, but no one plays the role of Supporter, and without anyone to look after the team, the team doesn't form. This risk is particularly strong on rapid-development projects because of pressure not to "waste time" on "nonproductive" activities such as developing a team logo or shared sense of humor. Each team needs to have someone who will take responsibility for maintaining the health of the team.

Teams can also lack identity because one or more members would rather work alone than be part of a team. Some people aren't joiners, and some people think the whole idea of teams is silly. Sometimes a group is composed of 9-to-5ers who don't want to make the commitment to their jobs that team membership entails. There are lots of appropriate places for people who work like this, but their presence can be deadly to team formation.

Lack of recognition. Sometimes project members have been part of a project team that gave its heart and soul—only to find that its efforts weren't appreciated. One young woman I know worked practically nonstop for 3 months to meet a deadline. When her product shipped, the manager thanked her in a fatherly way and gave her a stuffed animal. She thought the gesture was patronizing, and she was livid. I wouldn't blame her for not signing up for another all-out team project. If an organization wants to create a high-performance team more than once, it should be sure to recognize the extraordinary efforts of the first team appropriately. If group members' previous experience has conditioned them to ask, "What's in it for me?" you'll have an uphill battle in getting a high-performance team to form.

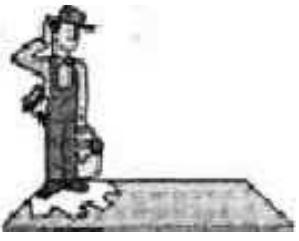
Productivity roadblocks. Sometimes teams fail because they feel that they can't be productive. People can't survive if the environment doesn't contain enough oxygen, and teams can't survive if they're prevented from getting their work done. Some experts say that the primary function of a software-project manager is to remove barriers to productivity so that the naturally self-motivated developers can be productive (DeMarco and Lister 1987).

Ineffective communication. Teams won't form if they can't communicate regularly. Common barriers to communication include lack of voicemail, lack of email, insufficient number of conference rooms, and separation of the team members into geographically dispersed sites. Bill Gates has pointed out that doing all of Microsoft's new-product development on one site is a major advantage because whenever interdependencies exist, you can talk about them face to face (Cusumano and Selby 1995).

Lack of trust. Lack of trust can kill a team's morale as quickly as any other factor. One reason that teams usually don't form within bureaucratic organizations is that the organizations (to varying extents) are based on lack of trust. You've probably heard something similar to this: "We caught someone buying an extra pack of 3-by-5 cards in August of 1952, so now all purchases have to go through central purchasing." The lack of trust for employees is often institutionalized.

Managers who pay more attention to how their teams go about administrative details than to the results they achieve are demonstrating a lack of trust. Managers who micromanage their team's activities, who don't allow them to meet with their customers, or who give them phony deadlines are giving a clear signal that they don't trust them.

Instead of micromanaging a project team, set up a high-level project charter. Let the team run within that charter. Set it up so that management can't overrule the team unless they've gone against their charter.



CLASSIC MISTAKE

Problem personnel. The software field is littered with stories of developers who are uncooperative in legendary proportions. I worked with one belligerent developer who said things like, "OK, Mr. Smarty Pants Programmer, if you're so great, how come I just found a bug in your code?" Some programmers browbeat their co-workers into using their design approaches. Their nonconfrontational co-workers would rather acquiesce to their design demands than prolong their interactions with them. I know of one developer who was so difficult to work with that the human resources department had to be brought in to resolve module-design disputes.

If you tolerate even one developer whom the other developers think is a problem, you'll hurt the morale of the good developers. You are implying that not only do you expect your team members to give their all; you expect them to do it when their co-workers are 'working against them.'



HARD DATA

In a review of 32 management teams, Larson and LaFasto found that the most consistent and intense complaint from team members was that their team leaders were unwilling to confront and resolve problems associated with poor performance by individual team members (Larson and LaFasto 1989). They report that, "[m]ore than any other single aspect of team leadership, members are disturbed by leaders who are unwilling to deal directly and effectively with self-serving or noncontributing team members." They go on to say that this is a significant management blind spot because managers nearly always think their teams are running more smoothly than their team members do.

Problem personnel are easy to identify if you know what to look for:

- They cover up their ignorance rather than trying to learn from their teammates. "I don't know how to explain my design; I just know that it works" or "My code is too complicated to test." (These are both actual quotes.)
- They have an excessive desire for privacy. "I don't need anyone to review my code."
- They are territorial. "No one else can fix the bugs in my code. I'm too busy to fix them now, but I'll get to them next week."
- They grumble about team decisions and continue to revisit old discussions after the team has moved on. "I still think we ought to go back and change the design we were talking about last month. The one we picked isn't going to work."
- Other team members all make wisecracks or complain about the same person. Software developers often won't complain directly, so you have to ask if there's a problem when you hear many wisecracks.

- They don't pitch in on team activities. On one project I worked on, 2 days before our first major deadline a developer asked for the next day off. The reason? He wanted to spend the day at a men's-clothing sale in a nearby city—a clear sign that he hadn't integrated with the team.

Coaching the problem person on how to work as part of a team sometimes works, but it's usually better to leave the coaching to the team than to try to do it as the team leader or manager. You might have to coach the team on how to coach the problem team member.

If coaching doesn't produce results quickly, don't be afraid to fire a person who doesn't have the best interests of the team at heart. Here are three solid reasons:

- It's rare to see a major problem caused by lack of skill. It's nearly always attitude, and attitudes are hard to change.
- The longer you keep a disruptive person around, the more legitimacy that person will gain through casual contacts with other groups and managers, a growing base of code that person has to maintain, and so on.
- Some managers say that they have never regretted firing anyone. They've only regretted not doing it sooner.

You might worry about losing ground if you replace a team member, but on a project of almost any size, you'll more than make up for the lost ground by eliminating a person who's working against the rest of the team. Cut your losses, and improve the rest of your team's morale.

12.5 Long-Term Teambuilding

The team of Amish farmers is a good model of the perfect, jelled team. But that team didn't form overnight. Those farmers had been together for years, and their families had been together for years before that. You can't expect performance as dramatic as raising a barn in a single day from a temporary team. That kind of productivity comes only from permanent teams.

Here are some reasons to keep teams together permanently.

Higher productivity. With a permanent-team strategy, you keep a group together if it jells into a team, and you disband it if it doesn't. Rather than breaking up *every* team and rolling the dice on every new project to see whether new teams jell or not, you roll the dice only after you've lost. You stockpile your winnings by keeping the productive teams together. The net effect is an "averaging up" of the level of performance in your organization.

Lower startup costs. The startup costs for building a team are unavoidable, so why not try to reuse the team and avoid additional startup costs? By keeping the effective teams together, you preserve some of the vision, team identity, communication, trust, and reservoir of good will built up from completing an enjoyable project together. You're also likely to preserve specific technical practices and knowledge of specific tools within a group.

Lower risk of personnel problems. Personnel issues arising from people who work poorly together cost your projects time and money. You can avoid these problems altogether by keeping teams together when they jell.



HARD DATA

Less turnover. The current annual turnover rate is about 35 percent among computer people' (Thomsett 1990). DeMarco and Lister estimate that 20 percent of the average company's total labor expense is turnover cost (DeMarco and Lister 1987). An internal Australian Bureau of Statistics estimate placed the average time lost by a project team member's resignation at 6 weeks (Thomsett 1990). Studies by M. Cherlin and by the Butler Cox Foundation estimate the cost of replacing an experienced computer person at anywhere from \$20,000 to \$100,000 (Thomsett 1990).



HARD DATA

Costs are not limited simply to the loss of the employee. Productivity suffers generally. A study of 41 projects at Dupont found that projects with low turnover had 65 percent higher productivity than projects with high turnover (Martin 1991).

Not surprisingly, people who have formed into cohesive teams are less likely to leave a company than people who have not (Lakhanpal 1993). Why should they leave? They have found an environment they enjoy and can feel productive in.

The idleness question. Organizations are sometimes leery of keeping teams together because they might have to pay a team to sit idle until a project comes along that's appropriate for them to work on. That's a valid objection, but I think that in most organizations it isn't ultimately a strong objection.

Organizations that look exclusively at the cost of idle time overlook the costs of rebuilding teams for each new project. The cost of building a new team includes the cost of assembling the team and of training the team to work together.

Organizations tend to overlook how much they lose by breaking up a high-performance team. They take a chance that individuals who could be working as part of a high-performance team will instead become part of an average team or a poor one.

Some organizations worry that if they keep teams together, they won't be able to get any teams to work on certain projects. But others have found that if you give people the chance to work with other people they like, they'll work on just about any project (DeMarco and Lister 1987).

Finally, I have yet to see a software organization that has long idle periods. To the contrary, every project that I've ever worked on has started late because personnel weren't available to staff it until their previous projects were completed.

Peopleware issues tend to lose out in the bean-counter calculations because in the past they have lacked the quantitative support that staff-days-spent-idle has. But the situation has changed. Australian software consultant Rob Thomsett has shown that there is a tremendous return on investment from teambuilding—for example, it is an order of magnitude better than for CASE tools (Constantine 1995a). We now know that among groups of people with equivalent skills, the most productive teams are 2 to 3 times as productive as the least productive teams. They're 1½ to 2 times as productive as the average teams. If you have a team that you know is on the top end of that range, you would be smart to allow them to sit idle for up to one-third or even one-half of their work lives just to avoid the risk of breaking them up and substituting a merely average team in their place.

12.6 Summary of Teamwork Guidelines

Larson and LaFasto distilled the results of their research into a set of practical guidelines for team leaders and team members. If your team wants to adopt a set of rules, the guidelines in Table 12-1 on the facing page are a good place to start.

Case Study 12-4. A Second High-Performance Team

Frank O'Grady captured the intense efficiency that a jelled team can have:

"I would sit in on design meetings, amazed at what I was seeing. When they were on a roll, it was as if they were all in some kind of high-energy trance during which they could see in their mind's eye how the program would unfold through time. They spoke in rapid-fire shorthand, often accompanied by vivid hand gestures when they wanted to emphasize a point. After 15 minutes or so, a consensus was reached as to what had to be done. Everyone knew which programs had to be changed and recompiled. The meeting adjourned." (O'Grady 1990)

Table 12-1. Practical Guidelines for Team Members and Leaders

Team Leader	Team Members
<p>As a team leader, I will:</p> <ol style="list-style-type: none"> 1. Avoid compromising the team's objective with political issues. 2. Exhibit personal commitment to the team's goal. 3. Not dilute the team's efforts with too many priorities. 4. Be fair and impartial toward all team members. 5. Be willing to confront and resolve issues associated with inadequate performance by team members. 6. Be open to new ideas and information from team members. 	<p>As a team member, I will:</p> <ol style="list-style-type: none"> 1. Demonstrate a realistic understanding of my role and accountabilities. 2. Demonstrate objective and fact-based judgments. 3. Collaborate effectively with other team members. 4. Make the team goal a higher priority than any personal objective. 5. Demonstrate a willingness to devote whatever effort is necessary to achieve team success. 6. Be willing to share information, perceptions, and feedback appropriately. 7. Provide help to other team members when needed and appropriate. 8. Demonstrate high standards of excellence. 9. Stand behind and support team decisions. 10. Demonstrate courage of conviction by directly confronting important issues. 11. Demonstrate leadership in ways that contribute to the team's success. 12. Respond constructively to feedback from others.

Source: Adapted from *TeamWork* (Larson and LaFasto 1989).

Further Reading

Here are three books and articles about teambuilding in software:

DeMarco, Tom, and Timothy Lister. *Peopleware-. Productive Projects and Teams*. New York: Dorset House, 1987. Part IV of this book focuses on growing productive software teams. It's entertaining reading, and it provides memorable stories about teams that worked and teams that didn't.

Weinberg, Gerald M. *Quality Software Management, Volume 3: Congruent Action*. New York: Dorset House, 1994. Part IV of this book is on managing software teams. Weinberg's treatment of the topic is a little more systematic, a little more thorough, and just as entertaining as *Peopleware*'s. Parts I through III of his book lay the foundation for managing yourself and people who work in teams.

Constantine, Larry L. *Constantine on Peopleware*. Englewood Cliffs, N.J.: Yourdon Press, 1995. Constantine brings his expertise in software development and family counseling to bear on topics related to effective software teams.

Here are some sources of information on teambuilding in general:

Larson, Carl E., and Frank M. J. LaFasto. *Teamwork: What Must Go Right; What Can Go Wrong*. Newbury Park, Calif: Sage, 1989. This remarkable book describes what makes effective teams work. The authors conducted a 3-year study of 75 effective teams and distilled the results into eight principles, each of which is described in its own chapter. At 140 pages, this book is short, practical, easy to read, and informative.

Katzenbach, Jon, and Douglas Smith. *The Wisdom of Teams*. Boston: Harvard Business School Press, 1993. This is a full-scale treatment of teams in general rather than just software teams. It's a good alternative to Larson and LaFasto's book.

Dyer, William G. *Teambuilding*. Reading, Mass: Addison-Wesley, 1987. This book describes more of the nuts and bolts of teambuilding than Larson and LaFasto's book does. Whereas Larson and LaFasto's intended audience seems to be the leader of the team, this book's intended audience seems to be the leader of a teambuilding workshop. It makes a nice complement to either Larson and LaFasto's book or Katzenbach and Smith's.

Witness. Paramount Pictures. Produced by Edward S. Feldman and directed by Peter Weir, 1985. The Amish barn-raising scene is about 70 minutes into this love-story/thriller, which received Oscars for best original screenplay and best editing and was nominated for best picture, best direction, best actor, best cinematography, best art direction, and best original score.

Team Structure

Contents

- 13.1 Team-Structure Considerations
- 13.2 Team Models
- 13.3 Managers and Technical Leads

Related Topics

Teamwork: Chapter 12



EVEN WHEN YOU HAVE SKILLED, MOTIVATED, hard-working people, the wrong team structure can undercut their efforts instead of catapulting them to success. A poor team structure can increase development time, reduce quality, damage morale, increase turnover, and ultimately lead to project cancellation. Currently, about one-third of all team projects are organized in ineffective ways (Jones 1994).

This chapter describes the primary considerations in organizing a rapid-development team and lays out several models of team structure. It concludes with a discussion of one of the most nettlesome team-structure issues: the relationship between project managers and technical leads.

Case Study 13-1. Mismatch Between Project Objectives and Team Structure

After several failed projects, Bill was determined to bring the Giga-Bill 4.0 project in on time and within budget, so he brought in Randy, a high-priced consultant, to help him set up the project.

Randy talked to Bill about his project for a while and then recommended that he set up the project as a skunkworks team. "Software people are creative, and they need lots of flexibility. You should set them up with an off-site office and give them lots of autonomy so they can create. If you do that, they'll work day and night, and they can't help but complete the project on time."

(continued')

Case Study 13-1. Mismatch Between Project Objectives and Team Structure,*continued*

Bill was uncomfortable with the idea of setting up an off-site office, but the project was important, and he decided to take Randy's advice. He put the developer he considered to be his best lead, Carl, in charge of the project.

"Carl, we need to get this project finished as fast as possible. The end-users are clamoring for an upgrade to solve all of the problems with Giga-Bill 4.0. We've got to hit a home run with this project. We've got to make the users happy. The users are so eager to get a new product that they have already drafted a set of requirements. I've looked at the requirements, and it looks to me like their requirements tell us exactly what we need to build. We need to get this next version out in as close to 6 months as we can."

Bill continued. "Randy recommended that I not interfere with your day-to-day activities, so you're in charge. I'll give you whatever flexibility you want. You do whatever it takes to get this project done now!"

Carl was excited about the idea of working off-site, and he knew he would enjoy working with the other people on the team. He met with Juan and Jennifer later that day. "I've got good news and better news," he told them. "The good news is that Bill has gotten the word that the users are fed up with Giga-Bill 4.0, and we need to hit a home run with this project. He's willing to give us all the flexibility we need to hit the ball out of the park. The better news is that we're going to be set up with an off-site office and no interference from Bill or anyone else. Good-bye dress code!"

Juan and Jennifer were as excited about the project as Carl, and when they started the project officially two weeks later, their morale was sky-high. Carl went to work early the first day thinking that he would get some work done before the others arrived, and Juan and Jennifer were already there. They stayed late every day the first week, and had no trouble staying focused on the project. It felt great to have a chance to develop a truly great product.

At the end of the first week, Carl had a status meeting with Bill. "We've had some great ideas that are going to make a truly great product. We've come up with some ways to meet the users' requirements that will knock their socks off." Bill thought it was great to see Carl so charged about the project, and he decided not to risk squashing his blossoming morale by asking him about the schedule.

Carl continued to report status semimonthly, and Bill continued to be impressed with the team's extraordinary morale. Carl reported that the team was working at least 9 or 10 hours a day and most Saturdays without being asked. After the first meeting, Bill always asked whether the project was on schedule and Carl always reported that they were making great progress. Bill wanted more details, but Randy had emphasized that pushing too hard could hurt morale. So he didn't push.

At the 5½-month mark, Bill couldn't wait any more and asked: "How are you doing on the schedule?"

(continued)

Case Study 13-1. Mismatch Between Project Objectives and Team Structure, *continued*

"We're doing great," Carl answered. "We've been working night and day, and this program is really coming together."

"OK, but are you going to be able to deliver the software in 2 weeks?" Bill asked.

"Two weeks? No, we can't deliver in 2 weeks. This is a complicated program, and it will probably be more like 8 weeks," Carl said. "But it is really going to blow you away."

"Wait a minute!" Bill said. "I thought you told me that you were on schedule. The users are expecting to get this software in 2 weeks!"

"You said we needed to hit a home run. That's what we're doing. We needed more than exactly 6 months to do that. It should take about 7½ months. Don't worry. The users are going to love this software."

"Holy cow!" Bill said. "This is a disaster! You guys need to get that software done now! I've had users breathing down my neck for 6 months while I waited for you to get this done. They told us exactly what a home run would be 6 months ago. All you needed to do was follow their instructions. I spent a lot of personal chits to get approval to put the project off-site so that you could get it done fast. They're going to pin my ears back."

"Gee, I'm sorry Bill. I didn't realize that the schedule was the main thing here. I thought hitting a home run was. I'll talk with everyone on the team and see what we can do."

Carl went back to the team and told them about the change in direction. By that time, they had already done all of the design and a lot of the implementation for the home-run project, so they decided that the fastest way to finish would be to continue as planned. They had only 8 weeks to go. Changing course now would introduce all kinds of unpredictable side effects that would probably just prolong the schedule.

They continued to work as hard as possible, but their estimates had not been very good. At the 8-month mark, Bill decided that he had let this team goof off enough, and moved them back on-site. The team's morale plummeted, and they quit working voluntary nights and weekends. Bill responded by ordering them to work 10-hour days and mandatory Saturdays until they were done. The team remained enthusiastic about their product, but they had lost their enthusiasm for the project. They finally finished after 9½ months. The users loved the new software, but they said they wished they could have gotten it 4 months earlier.

13.1 Team-Structure Considerations

The first consideration in organizing a team is to determine the team's broad objective. Here are the broad objectives according to Larson and LaFasto (1989)

- Problem resolution
- Creativity
- Tactical execution

Once you've established the broad objective, you choose a team structure that matches it. Larson and LaFasto define three general team structures for effective, high-performance teams. The structure that's most appropriate depends on the team's objective.

The users in Case Study 13-1 had described exactly what they wanted, and the team really just needed to create and execute a plan for implementing it. Bill got some bad advice from Randy, who should have identified the primary goal of the team as tactical execution. The users were more interested in getting a timely upgrade than they were in creative solutions. The effect of the bad advice was predictable: organized for creativity, the team came up with a highly creative solution but didn't execute their plan efficiently. They weren't organized for that.

Kinds of Teams

Once you've identified the team's broadest objective—problem resolution, creativity, or tactical execution—then you set up a team structure that emphasizes the characteristic that is most important for that kind of team. For a problem-resolution team, you emphasize trust. For a creativity team, autonomy. And for a tactical-execution team, clarity.

Problem-resolution team. The problem-resolution team focuses on solving a complex, poorly defined problem. A group of epidemiologists working for the Centers for Disease Control who are trying to diagnose the cause of a cholera outbreak would be a problem-resolution team. A group of maintenance programmers trying to diagnose a new showstopper defect is too. The people on a problem-resolution team need to be trustworthy, intelligent, and pragmatic. Problem-resolution teams are chiefly occupied with one or more specific issues, and their team structure should support that focus.

Creativity team. The creativity team's charter is to explore possibilities and alternatives. A group of McDonald's food scientists trying to invent a new kind of McFood would be a creativity team. A group of programmers who are breaking new ground in a multimedia application would be another kind of creativity team. The creativity team's members need to be self-motivated, independent, creative, and persistent. The team's structure needs to support the team members' individual and collective autonomy.

Tactical-execution team. The tactical-execution team focuses on carrying out a well-defined plan. A team of commandos conducting a raid, a surgical team,

and a baseball team would all be tactical-execution teams. So would a software team working on a well-defined product upgrade in which the purpose of the upgrade is not to break new ground but to put well-understood functionality into users' hands as quickly as possible. This kind of team is characterized by having highly focused tasks and clearly defined roles. Success criteria tend to be black-and-white, so it is often easy to tell whether the team succeeds or fails. Tactical-execution team members need to have a sense of urgency about their mission, be more interested in action than esoteric intellectualizing, and be loyal to the team.

Table 13-1 summarizes the different team objectives and the team structures that support those objectives.

Table 13-1. Team Objectives and Team Structures

	Broad Objective		
	Problem Resolution	Creativity	Tactical Execution
<i>Dominant feature</i>	Trust	Autonomy	Clarity
<i>Typical software example</i>	Corrective maintenance on live systems	New product development	Product-upgrade development
<i>Process emphasis</i>	Focus on issues	Explore possibilities and alternatives	Highly focused tasks with clear roles, often marked by clear success or failure
<i>Appropriate lifecycle models</i>	Code-and-fix, spiral	Evolutionary prototyping, evolutionary delivery, spiral, design-to-schedule, design-to-tools	Waterfall, modified waterfalls, staged delivery, spiral, design-to-schedule, design-to-tools
<i>Team selection criteria</i>	Intelligent, street smart, people sensitive, high integrity	Cerebral, independent thinkers, self-starters, tenacious	Loyal, committed, action-oriented, sense of urgency, responsive
<i>Appropriate software-team models</i>	Business team, search-and-rescue team, SWAT team	Business team, chief-programmer team, skunkworks team, feature team, theater team	Business team, chief-programmer team, feature team, S'WAT team, professional athletic team

Source: Adapted from *Team Work* (Larson and LaFasto 1989).

Additional Team-Design Features

Beyond the three basic kinds of teams, there are four team-structure features that seem to characterize all kinds of effectively functioning teams:

Clear roles and accountabilities. On a high-performance team, every person counts, and everyone knows what they are supposed to do. As Larson and LaFasto say, "EVERYONE IS ACCOUNTABLE ALL THE TIME on successful teams" [*authors' emphasis*] (Larson and LaFasto 1989).

Monitoring of individual performance and providing feedback. The flip side of accountability is that team members need some way of knowing whether they are living up to the team's expectations. The team needs to have mechanisms in place to let team members know in what ways their performance is acceptable and in what ways it needs improvement.

Effective communication. Effective communication depends on several project characteristics.

Information must be easily accessible. Meting out information on a "need to know" basis is bad for morale on a rapid-development project. Put all relevant information including documents, spreadsheets, and project-planning materials into version control and make them available on-line.

Information must originate from credible sources. The team's confidence in its decision making—the extent to which it's willing to make decisions actively or boldly—depends on how confident it is in the information on which it bases its decisions.

There must be opportunities for team members to raise issues not on the formal agenda. The word "formal" is key. Team members need informal opportunities to raise issues in an environment where titles, positions, office sizes, and power ties are not part of the equation. This is part of the underlying reason for the success of informal management approaches such as Management By Walking Around.

The communication system must provide for documenting issues raised and decisions made. Keeping accurate records prevents the team from retracing its steps through old decisions.

Fact-based decision making. Subjective judgments can undercut team morale. High-performance team members need to understand the bases for all decisions that affect them. If they find that decisions are made for arbitrary, subjective, or self-serving reasons, their performance will suffer.

Which Kind of Team Is Best for Rapid Development?

CROSS-REFERENCE

For more on the need to tailor the development approach to the project, see Section 2.4, "Which Dimension Matters the Most?" and Section 6.1, "Does One Size Fit All?"

A key to organizing a team for rapid development is understanding that there is no single team structure that achieves the maximum development speed on every project.

Suppose you're working on a brand new word-processing product and your goal is to create the best word processor in the world. You don't know at the beginning of the project exactly what the world's best word processor looks like. Part of your job will be to discover the characteristics that make up an exceptional product. For the most rapid development within that context, you should choose a team structure that supports creativity.

Now suppose that you're working on version 2 of that same word-processing product. You learned on version 1 what it would take to create a world-class product, and you don't view version 2 as exploratory. You have a detailed list of features that need to be implemented, and your goal is to implement them as fast as possible so that you stay ahead of the competition. For the most rapid development within that context, you should choose a team structure that supports tactical execution.

There's no such thing as a single best "rapid-development team structure" because the most effective structure depends on the context (See Figure 13-1.)

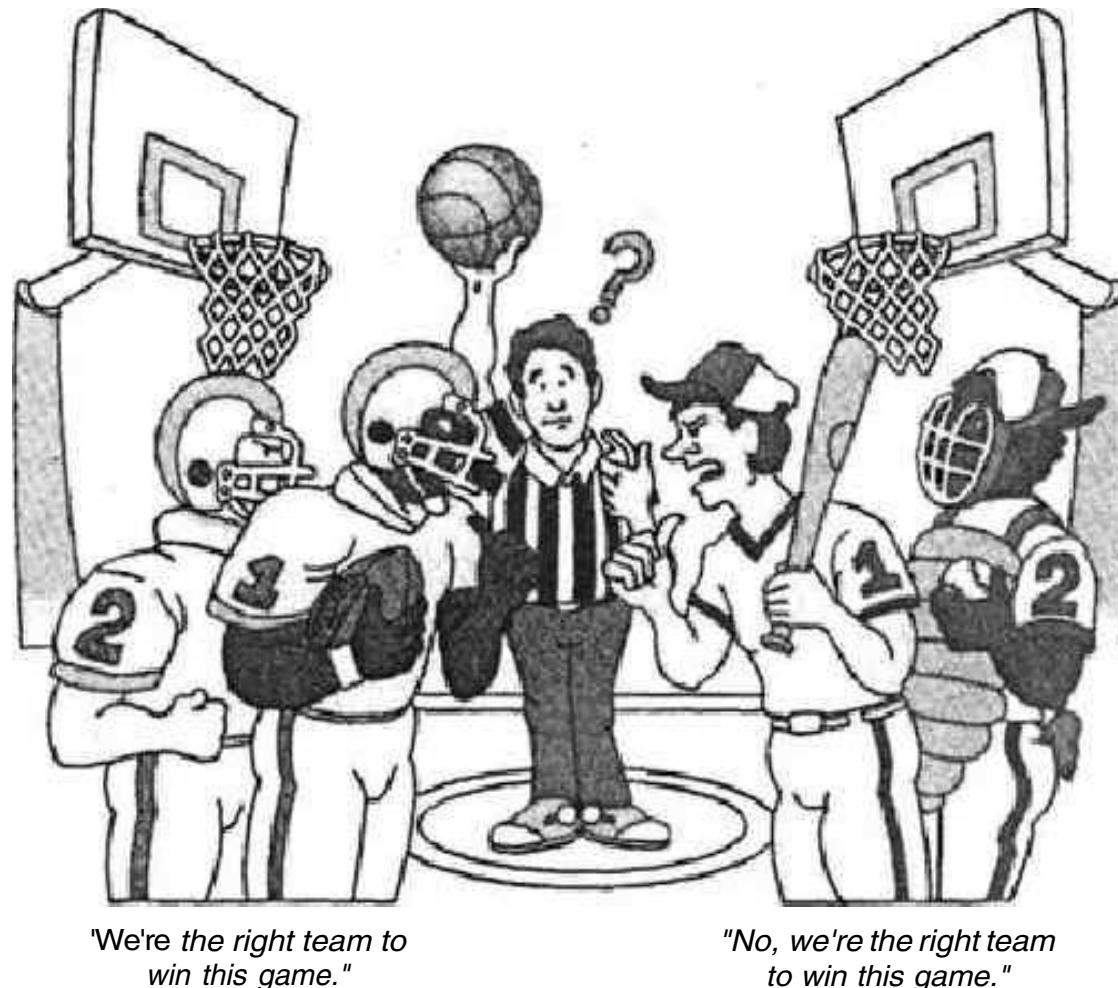


Figure 13-1. No single team structure is best for all projects.

13.2 Team Models

CROSS-REFERENCE
For more on the roles that people play in effective teams, see "Mix of Roles" in Section 12.3.

Team leads, project managers, writers, and researchers have come up with many team models over the years, and this section catalogs a few of them. Some of the models affect only how the team operates on the inside and thus could be implemented by the technical lead or the team itself. Others affect how the team looks to management and would ordinarily require management approval.

The models in this section don't make up an orthogonal set. You will find overlaps and contradictions among the models, and you could combine elements from several different models to make up your own model. This section is intended more to generate ideas about different ways to structure a team than to be a systematic presentation of all possible team structures.

Business Team

The most common team structure is probably the peer group headed by a technical lead. Aside from the technical lead, the team members all have equal status, and they are differentiated by area of expertise: database, graphics, user interface, and various programming languages. The technical lead is an active technical contributor and is thought of as the first among equals. The lead is usually chosen on the basis of technical expertise rather than management proficiency.

Most commonly, the lead is responsible for making final decisions on tough technical issues. Sometimes the lead is a regular team member who merely has the extra duty of being the team's link to management. In other cases, the lead occupies a first-level management position. The specific amount of management responsibility the technical lead has varies from one organization to another, and I'll discuss that topic more later in the chapter.

From the outside, the business-team structure looks like a typical hierarchical structure. It streamlines communication with management by identifying, one person as principally responsible for technical work on the project. It allows each team member to work in his or her area of expertise, and it allows the team itself to sort out who should work on what. It works well with small groups and with long-standing groups that can sort out their relationships over time.

It is adaptable enough that it can work on all kinds of projects—problem resolution, creativity, and tactical execution. But its generality is also its weakness, and in many cases a different structure can work better.

Chief-Programmer Team

The idea of the chief-programmer team was originally developed at IBM during the late 1960s and early 1970s (Baker 1972, Baker and Mills 1973). It was popularized by Fred Brooks in the *Mythical Man-Month* (Brooks 1975, 1995), in which Brooks referred to it as a surgical team. The two terms are interchangeable.

CROSS-REFERENCE
For more on variations in individual performance, see "People" in Section 2.2.

The chief-programmer team takes advantage of the phenomenon that some developers are 10 times as productive as others. Ordinary team structures put mediocre programmers and superstars on equal footing. You take advantage of the high productivity of the superstars, but you're also penalized by the lower productivity of other team members. In the surgical-team concept, a programming superstar is identified as the surgeon, or chief programmer. That person then drafts the entire specification, completes all of the design, writes the vast majority of the production code, and is ultimately responsible for virtually all of the decisions on a project.

With the surgeon handling the bulk of the design and code, other team members are free to specialize. They are arrayed about the surgeon in support roles, and the chief-programmer team takes advantage of the fact that specialists tend to outperform generalists (Jones 1991).

A "backup programmer" serves as the chief programmer's alter ego. The backup programmer supports the surgeon as critic, research assistant, technical contact for outside groups, and backup surgeon.

The "administrator" handles administrative matters such as money, people, space, and machines. Although the surgeon has ultimate say about these matters, the administrator frees the surgeon from having to deal with them on a daily basis.

The "toolsmith" is responsible for creating custom tools requested by the surgeon. In today's terminology, the toolsmith would be in charge of creating command scripts and make files, of crafting macros for use in the programming editor, and of running the daily build.

The team is rounded out by a "language lawyer" who supports the surgeon by answering esoteric questions about the programming language the surgeon is using.

Several of the support roles suggested in the original chief-programmer proposal are now regularly performed by nonprogrammers—by documentation specialists, test specialists, and program managers. Other tasks such as word processing and version control have been simplified so much by modern software tools that they no longer need to be performed by support personnel.

When it was first used more than 20 years ago, the chief-programmer team achieved a level of productivity unheard of in its time (Baker and Mills 1973). In the years since, many organizations have attempted to implement chief-programmer teams, and most have not been able to repeat the initial stunning success. It turns out that true superstars capable of serving as chief programmers are rare. When individuals with such exceptional capabilities are found, they want to work on state-of-the-art projects, which is not what most organizations have to offer.

In spite of 20 years worth of changes and the rarity of superstar programmers, I think this structure can still be appropriate when used opportunistically. You can't start out by saying, "I need to get this project done fast, and I want to use a chief-programmer team structure." But what if you do happen to have a superstar who's willing to work exceptionally hard, who has few other interests, and who is willing to put in 16 hours a day? In that case, I think the chief-programmer team can be the answer.

The chief-programmer team is appropriate for creative projects, in which having one mind at the top will help to protect the system's conceptual integrity. It's also well suited to tactical-execution projects, in which the chief programmer can serve as near dictator in plotting out the most expeditious means of reaching project completion.

Skunkworks Team

The skunkworks team is an integral part of the lore of the engineering world. A skunkworks project takes a group of talented, creative product developers, puts them in a facility where they will be freed of the organization's normal bureaucratic restrictions, and turns them loose to develop and innovate.

Skunkworks teams are typically treated as black-boxes by their management. The management doesn't want to know the details of how they do their job; they just want to know that they're doing it. The team is thus free to organize itself as it sees fit. A natural leader might emerge over time, or the team might designate a leader from the outset.

Skunkworks projects have the advantage of creating a feeling of intense ownership and extraordinary buy-in from the developers involved. The motivational effect can be astounding. They have the disadvantage of not providing much visibility into the team's progress. Some of this is probably an inevitable effect of the unpredictability involved in any highly creative work. Some of it is an explicit trade-off—trading a loss in visibility for an increase in motivation.

Increased developer attachment to specific features. The increased ownership that developers feel with a minimal-spec approach can be a double-edged sword. You get extra motivation, but you can also encounter more resistance when you want to change one of the developer's features. Keep in mind that when product changes are suggested, they are no longer implicitly critical of only the product; they are implicitly critical of the developer's work.

CROSS-REFERENCE
For details on incremental delivery strategies, see Chapter 7, "Lifecycle Planning."

Use of minimal specification for the wrong reason. Don't use a minimal specification to reduce time on the requirements-specification activity itself. If you use this practice as a lazy substitute for doing a good job of requirements specification, you'll end up with a lot of rework—designing and implementing features twice, the second time at a point in the project when it's expensive to change your mind. But if you use this practice to avoid doing work that would be wasted, you follow through with clear goals, and you give developers latitude in how they implement their features, then you can develop more efficiently.



CLASSIC MISTAKE

If you use this approach, both you and your customers have to be prepared to accept a version of the product that doesn't match your vision of what the product should look like. Sometimes people start out saying they want to use this approach—and then, when the product departs from their mental image, they try to bring it into line. Worse, some customers will try to use the looseness of this specification approach to their advantage, shoehorning in extra features late in the project and interpreting every feature in the most elaborate way possible. If you allow much of that, you'll waste time by making late, costly changes; you would have done better specifying your intent up front and then designing and implementing the product efficiently the first time.

Know yourself and your customers well enough to know whether you'll accept the results of giving developers this much discretion. If you're uncomfortable with ceding such control, you'll be better off using a traditional specification approach or an incremental-delivery strategy.

Keys to success in using minimal specifications

There are several keys to success in using minimal specifications.

Use a minimum specification only when requirements are flexible. The success of this approach depends on the flexibility of the requirements. If you think the requirements are really less flexible than they appear, take steps to be sure that they are flexible before you commit to use a minimal specification. Flexible initial requirements have been identified by some people as a key to success with any rapid-development approach (Millington and Stapleton 1995).

Keep the spec to a minimum. With any of these approaches, you'll need to make it clear that one of the objectives is to specify only the minimum detail necessary. For items that users might or might not care about, the default option should be to leave further specification to the developers. When in doubt, leave it out!

Capture the important requirements. Although you should try not to capture requirements that the users don't care about, you must be careful to capture all of the requirements that they do care about. Doing a good minimal spec requires a special sensitivity to what users really care about.

CROSS-REFERENCE
For more on flexible development approaches, see Chapter 7, "Lifecycle Planning," and Chapter 19, "Designing for Change."

Use flexible development approaches. Use development approaches that allow mistakes to be corrected. With a minimal-spec approach, more mistakes will occur than with a traditional-spec approach. The use of flexible development approaches is a means of hedging your bet that you can save more time than you waste.

Involve key users. Find people who understand the business need or organizational need for the software, and involve them in product specification and development. This helps to avoid the problem of omitted requirements.

Focus on graphically oriented documentation. Graphics in the form of diagrams, sample outputs, and live prototypes tend to be easier to create than written specifications and more meaningful to users. For the graphically oriented parts of your system, focus your documentation efforts on creating graphically oriented materials.

Requirements Scrubbing

Entirely removing ("scrubbing") a feature from a product is one of the most powerful ways to shorten a software schedule because you remove every ounce of effort associated with that feature: specification, design, testing, documentation—everything. The earlier in the project you remove a feature, the more time you save. Requirements scrubbing is less risk¹ than minimal specification. Because it reduces the size and complexity of the product, it also reduces the overall risk level of the project. (See Figure 14-3 on the next page.)

The idea behind requirements scrubbing is simple: After you create a product specification, go over the specification with a fine-tooth comb and with the following aims:

- Eliminate all requirements that are not absolutely necessary.
- Simplify all requirements that are more complicated than necessary.
- Substitute cheaper options for all requirements that have cheaper options.

As with minimal specification, the ultimate success of this practice depends on follow-through. If you begin with 100 requirements and the requirements-scrubbing activity pares that number down to 70, you might well be able to complete the project with 70 percent of the original effort. But if you pare the list down to 70 only to reinstate the deleted requirements later, the project will likely cost more than it would have if you had retained the entire 100 requirements the whole time.

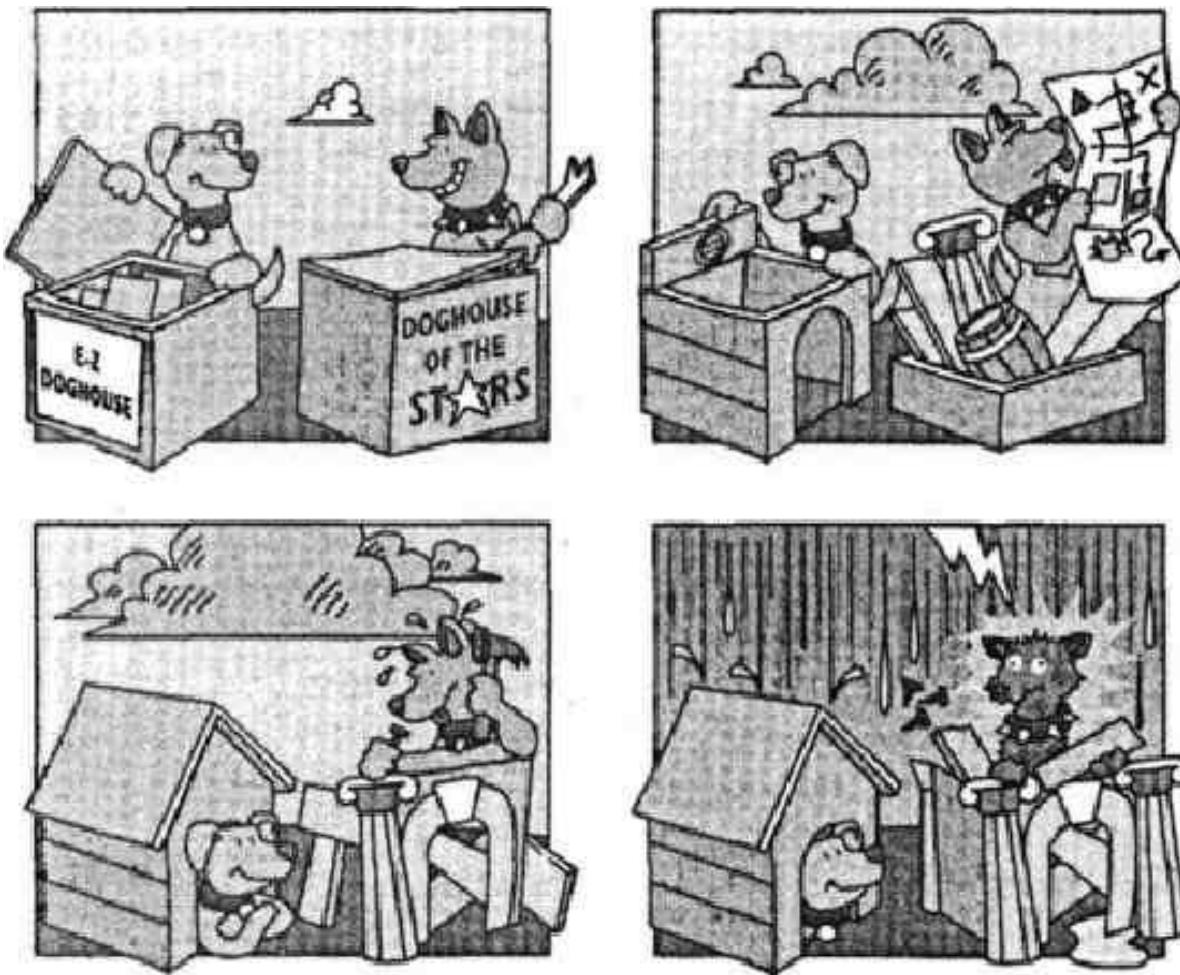


Figure 14-3. Smaller projects take less time to build.

Versioned Development

gROSS-REFERENCE
For more on versioned development, see "Define Families of Programs" in Section 19.1; Chapter 20, "Evolutionary Delivery"; and Chapter 36, "Staged Delivery."

An alternative to eliminating requirements altogether is eliminating them from the current version. You can plan out a set of requirements for a robust, complete, ideal project but then implement the project in pieces. Put in any hooks you'll need to support the later pieces, but don't implement the pieces themselves. The development practices of evolutionary delivery and staged delivery¹⁴ can help in this area.

The inevitable effect of using these practices is that by the time you finish version 1 and begin work on version 2, you scrap some of the features you had originally planned to have in the version 2 and add others. When that

happens, you become especially glad that you didn't put the scrapped features into version 1.

14.2 Mid-Project: Feature-Creep Control

If you do a good job of specifying a lean product, you might think you have your feature set under control. Most projects aren't that lucky. For many years, the holy grail of requirements management has been to collect a set of requirements—scrubbed, minimally specified, or otherwise—encase them in permafrost, and then build a complete product design, implementation, documentation, and quality assurance atop them. Unfortunately for developers and their ulcers, projects that have successfully frozen their requirements have proven to be almost as hard to find as the Holy Grail itself. A typical project experiences about a 25-percent change in requirements during development (Boehm 1981, Jones 1994).

Sources of Change

Mid-project changes arise from many sources. End-users want changes because they need additional functionality or different functionality or because they gain a better understanding of the system as it's being built.

Marketers want changes because they see the market as feature-driven. Software reviews contain long lists of features and check marks. If new products with new features come out during a product's development, marketers naturally want their products to stack up well against the competition, and they want the new features added to their product.

Developers want changes because they have a great emotional and intellectual investment in all of the system's details. If they're building the second version of a system, they want to correct the first version's deficiencies, whether such changes are required or not. Each developer has an area of special interest. It doesn't matter whether a user interface that's 100 percent compliant with user-interface standards is required, or whether lightning-fast response time, perfectly commented code, or a fully normalized database are specified: developers will do whatever work is needed to satisfy their special interests.

All these groups—end-users, marketers, and developers—will try to put their favorite features into the requirements spec even if they didn't make it during the formal requirements-specification activity. Users sometimes try to end-run the requirements process and coax specific developers into implementing their favorite features. Marketers build a marketing case and insist later that their favorite features be added. Developers implement unrequired features on their own time or when the boss is looking the other way.

CROSS-REFERENCE

For tips on resisting pressure, including pressure to add requirements, see Section 9.2, "Beating Schedule Pressure."

All in all, projects tend to experience about a 1-percent change in requirements per month. On average, the longer your project takes, the more your product will change before it's complete (Jones 1994). A few factors can make that figure considerably worse.

Killer-app syndrome

Shrink-wrap products are particularly susceptible to "killer-app syndrome." The development group at Company A sets developing "the best application in its class" as its design goal. The group designs an application that meets that criteria and then begins to implement it. A few weeks before the software is scheduled to ship, Company B's application enters the market. Their application has some features that Company A never thought of and others that are superior to Company A's. The development group at Company A decides to slide its schedule a few months so that it can redesign its application and truly clobber Company B. It works until a few weeks before its revised ship date, and then Company C releases its software, which is again superior in some areas. The cycle begins again.

Unclear or impossible goals

It's difficult to resist setting ambitious goals for a project: "We want to develop a world-class product in the shortest possible time at the lowest possible cost." Because it isn't possible to meet that entire set of goals or because the goals are unclear, the most likely result is meeting *none* of the goals. If developers can't meet the project's goals, they will meet their own goals instead, and you will lose much of your influence over the project's outcome.

To illustrate the way in which clear goals can have a significant effect on a development schedule, consider the design and construction of a charting program. There is a tiny part of the charting program that deals with "polymarkers" — squares, circles, triangles, and stars that designate specific points on a graph. Figure 14-4 shows an example. Suppose that the specification is silent on the question of whether to provide the user with the ability to control the polymarkers' sizes. In such a case, the developer who implements the polymarkers can provide such control in any of many ways:

1. Do not provide any control at all.
2. Set up the source code to be modified in one place for the whole set of polymarkers (that is, sizes of all polymarkers are set by a single named constant or preprocessor macro).
3. Set up the source code to be modified in one place, on a polymarker-by-polymarker basis, for a fixed number of polymarkers (that is, size of each polymarker — square, triangle, and so on — is set by its own named constant or preprocessor macro).

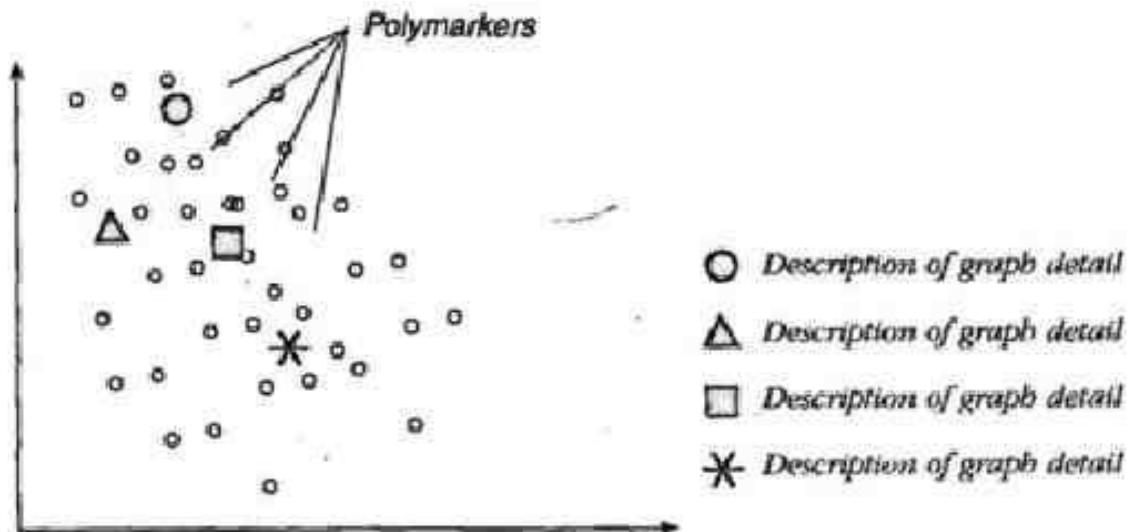


Figure 14-4. Examples of polymarkers. There can be at least a 10-to-1 difference in size and implementation time of even seemingly trivial features.

4. Set up the source code to be modified in one place, on a polymarker-by-polymarker basis, for a dynamic number of polymarkers (for example, you might later want to add cross-hairs, diamonds, and bull's-eyes to the original set of polymarkers).
5. Allow for modification of an external file that the program reads at startup, one setting for the whole set of polymarkers (for example, an .ini file or other external file).
6. Allow for modification of an external file that the program reads at startup, different settings for each polymarker, for a fixed number of polymarkers.
7. Allow for modification of an external file that the program reads at startup, different settings for each polymarker, for a dynamic number of polymarkers.
8. Allow for interactive, end-user modification of a single polymarker-size specification.
9. Allow for interactive, end-user modification of polymarker-size specifications, with one setting per polymarker, for a fixed number of polymarkers.
10. Allow for interactive, end-user modification of polymarker-size specifications, with one setting per polymarker, for a dynamic number of polymarkers.

These options represent huge differences in design and implementation times. At the low end, a fixed number of polymarkers have their sizes hard-coded into the program. This requires only that polymarkers be represented in an array of fixed size. The amount of work required to implement that

beyond the base work required to implement polymarkers would be negligible, probably on the order of a few minutes.

At the high end, a variable maximum number of polymarkers have their sizes set at runtime, interactively, by the user. That calls for dynamically allocated data, a flexible dialog box, persistent storage of the polymarker sizes set by the user, extra source-code files put under version control, extra test cases, and documentation of all the above in paper documents and online help. The amount of work required to implement this would probably be measured in weeks.

The amazing thing about this example is that it represents weeks of potential difference in schedule arising from a *single, trivial* characteristic of a charting program—the size of the polymarkers. We haven't even gotten to the possibility that polymarkers might also be allowed to have different outline colors, outline thicknesses, fill colors, orderings, and so on. Even worse, this seemingly trivial issue is likely to interact with other seemingly trivial issues in combination, meaning that you multiply their difficulties together rather than merely add them.

The point of this is that the devil really is in the details, and implementation time can vary tremendously based on how developers interpret seemingly trivial details. *No specification can hope to cover every one of these trivial details.*



Without any guidelines to the contrary, developers will pursue flexible approaches that tend more toward option #10 than #1. Most conscientious developers will intentionally try to design some flexibility into their code, and as the example illustrates, the amount of flexibility that a good developer will put into code can vary tremendously. As my friend and colleague Hank Meuret says, the programmer ideal is to be able to change one compiler switch and compile a program as a spreadsheet instead of a word processor. When you multiply the tendency to choose flexibility rather than development speed across dozens of developers and hundreds of detailed decisions on a project, it's easy to see why some programs are vastly larger than expected and take vastly longer than expected to complete. Some studies have found up to 10-to-1 differences in the sizes of programs written to the same specification (DeMarco and Lister 1989).

CROSS-REFERENCE
For more on goal setting, see
"Goal Setting" in Section 11.2.

If you were to proceed through implementation with the assumption that whenever you encountered an ambiguity in the specification you would tend toward the #1 end of the set of options rather than toward the #10 end, you could easily implement your whole program an order of magnitude faster than someone who took the opposite approach. If you want to leverage your

product's feature set to achieve maximum development speed, you must make it clear that you want your team to tend toward #1. You must make it clear that development speed is the top design-and-implementation goal, and you must not confuse that goal by piling many other goals on top of it.

Effects of Change

People are far too casual about the effects that late changes in a project have. They underestimate the ripple effects that changes have on the project's design, code, testing, documentation, customer support, training, configuration management, personnel assignments, management and staff communications, planning and tracking, and ultimately on the schedule, budget, and product quality (Boehm 1989). When all these factors are considered, changes typically cost anywhere from 50 to 200 times less if you make them at requirements time than if you wait until construction or maintenance (Boehm and Papaccio 1988).

As I said at the beginning of the chapter, several studies have found that feature creep is the most common source of cost and schedule overruns. A study at ITT produced some interesting results in this area (Vosburgh et al. 1984). It found that projects that experienced enough change to need their specifications to be rewritten were significantly less productive than projects that didn't need to rewrite their specs. Figure 14-5 illustrates the difference.

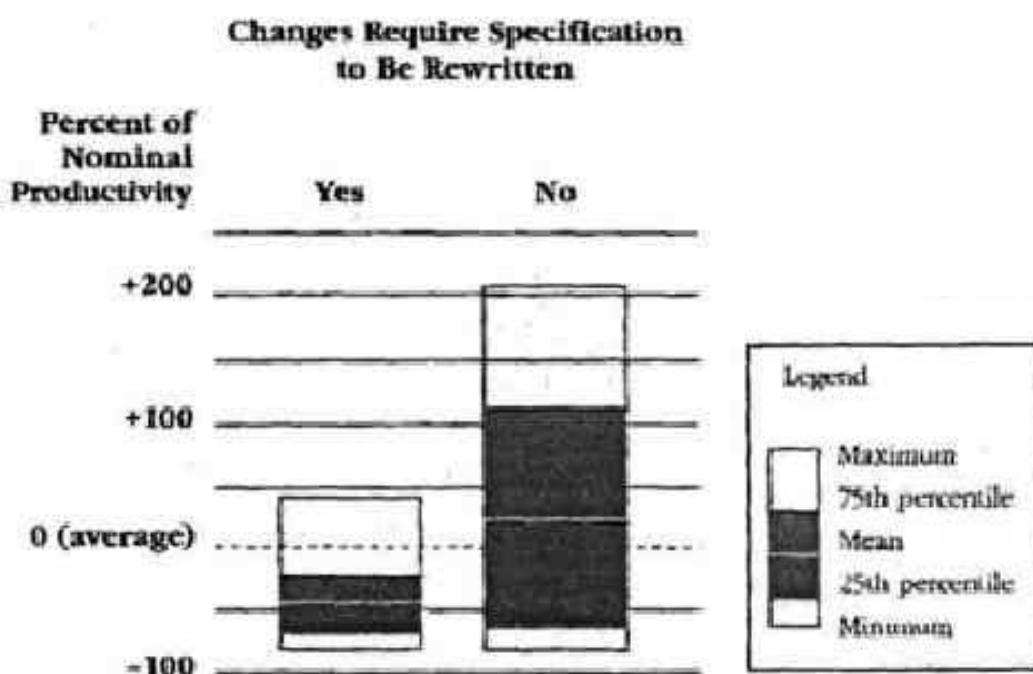


Figure 14-5. Findings for "Changes Require Specification to Be Rewritten" factor (Vosburgh et al. 1984). Controlling changes can produce dramatic improvements in productivity, but that control does not by itself guarantee success.

CROSS-REFERENCE
For details on this point, see
Section 3.2, "Effect of
Mistakes on a Development
Schedule."

As you can see in Figure 14-5, both the average and maximum productivities were higher when changes were controlled, and this study suggests that it is hardly ever possible to achieve high productivity unless you control changes. At the bottom ends of the ranges, some projects had the same low productivity regardless of whether they did a good job of controlling changes. As with other effective development practices, controlling changes is not by itself sufficient to guarantee high productivity. Even when you do a good job of controlling changes, there are other ways to torpedo a project.

Wisdom of Stopping Changes Altogether

Requirements that don't change are great. If you can develop software to an unchanging set of requirements, you can plan with almost 100-percent accuracy. You can design and code without wasting time to make late changes inefficiently. You can use any of the unstable-requirements practices and any of the stable-requirements practices too. The whole universe of speed-oriented development practices is open to you. Such projects are a joy to work on and a joy to manage. You can reach project completion faster and more economically than you can in any other circumstances.

That's nice work if you can get it. When you do need some flexibility, however, pretending that no changes are allowable or desirable is just a way to lose control of the change process. Here are some circumstances in which it is unwise to try to stop changes altogether.

When your customers don't know what they want. Refusing to allow changes assumes that your customers know what they want at requirements time. It assumes that you can truly know requirements at the outset, before design and coding begin. On most projects, that's not possible. Part of the software developer's job is to help customers figure out what they want, and customers often can't figure out what they want until they have working software in their hands. You can use the various incremental development practices to respond to this need.

When you want to be responsive to your customer. Even when you think your customers know what they want, you might want to keep the software flexible to keep your options open. If you follow a frozen-requirements plan, you might deliver the product on time, but you might seem unresponsive, and that can be just as bad as late delivery. If you're a contract software developer, you might need to stay flexible to stay competitive. If you're an in-house developer, your company's well-being might depend on your providing that flexibility.

In-house development organizations are especially susceptible to being unresponsive to their users. They quit looking at their users as true customers because they have an exclusive engagement. Friction results over time, and then the developers use the requirements specification as a weapon to force their users to behave. No one likes working in the face of a steady stream of arbitrary changes, but there are more constructive ways to respond than beating the users on the head with a requirements document. Developers who do that are finding with increasing frequency that their engagements aren't as exclusive as they had once thought (Yourdon 1992).

When the market is changing rapidly. In the 1960s, and earlier, when business needs changed more slowly than they do today, it might have been wise to plan a product in detail two years before you released it. Today, the most successful products are often those that had the most change implemented the latest in the development cycle. Rather than automatically trying to eliminate requirements changes, the software developer's job today is to strike a balance between chaos and rigidity—rejecting low-priority changes and accepting changes that represent prudent responses to changing market conditions.

When you want to give latitude to the developers. One big change associated with the PC revolution has been that project sponsors are more willing to leave large parts of the specification to the developers' discretion (for the reasons described in the "Minimal Specification" section early in this chapter). If you want to leave part of the product concept to the developers, you can't freeze the product concept as soon as requirements specification is complete; you have to leave at least part of it open for the developers to interpret.

Stable or not stable?

How stable your requirements are has a huge impact on how you go about software development, and particularly on what you need to do to develop rapidly. If your requirements are stable, you develop one way. If they are unstable, you develop another way. From a rapid-development point of view, one of the worst mistakes you can make is to think your requirements are stable when they aren't.

If your project doesn't have stable requirements, it isn't sufficient any more to throw up your hands and shout "Yikes! We don't have stable requirements!"—and proceed the same as you would have if you did have stable requirements. You can take certain steps to compensate for unstable requirements, and you must take those steps if you want to develop rapidly.

Methods of Change Control

Because stopping changes altogether is rarely in a project's best interest, the question for most projects turns to how to manage change most effectively. Any change-management plan should aim for several goals:

- Allow changes that help to produce the best possible product in the time available. Disallow all other changes.
- Allow all parties that would be affected by a proposed change to assess the schedule, resource, and product impacts of the change.
- Notify parties on the periphery of the project of each proposed change, its assessed impact, and whether it was accepted or rejected.
- Provide an audit trail of decisions related to the product content.

The change process should be structured to perform these jobs as efficiently as possible. Here are some options for accomplishing that objective.

Customer-oriented requirements practices

One change-control strategy is to try to minimize the number of changes needed. Customer-oriented requirements-gathering practices do a better job of eliciting the real requirements than traditional practices. For example, one study found that the combination of JAD and prototyping can drop creeping requirements to below 5 percent (Jones 1994). Prototyping helps to minimize changes because it strikes at what is often the root of the problem—the fact that customers don't know what they want until they see it. Throwaway prototypes are generally the most effective in this regard and provide the most resistance to requirements creep (Jones 1994).

CROSS-REFERENCE
For more on customer-oriented requirements practices, see Chapter 10, "Customer-Oriented Development."™ For details on throwaway prototypes, see Chapter 38, "Throwaway Prototyping."

Change analysis

In most cases, it's not the developer's or technical lead's job to say "No" to changes. But you can use the change-analysis process to screen out superfluous changes. Rather than saying "No" to each change, you provide cost and schedule impacts. Explain that you have to adjust the schedule for the time spent analyzing new feature requests. That explanation will screen out most of the frivolous change requests.

You can also screen out changes by making it harder to submit a change request. You can insist on a complete written specification of the change, a business-case analysis, a sample of the inputs and outputs that will be affected, and so on.

John Boddie tells the story of being called to the main office for an emergency schedule meeting on a rush project. When he and his team arrived, his boss asked "Is everything on schedule for delivery on the 18th?" Boddie said it was. Then the boss asked, "So we'll have the product on the 18th?"

Boddie said, "No, you'll have to wait until the 19th because this meeting has taken a day from our schedule" (Boddie 1987).

In the last few weeks of a project, you might say that the minimum schedule slide for any feature change is an entire day or more. You could even say that the minimum slide for *considering* a feature change is an entire day or more. This is appropriate since late-project changes tend to impact all aspects of the project, often in ways that are hard to predict without careful analysis.

You would probably want to make it a little easier to get defect-related change requests accepted. But even minor defects can have far-reaching consequences, so, depending on the kind of program and the stage of the project, you might not.

Version 2

One great help in saying "No" to changing the current product is being able to say "Yes" to putting those changes into some future product. Create a list of future enhancements. People should understand that features won't necessarily make it into version 2 just because they didn't make it into version 1, but you don't need to emphasize that point. What you do need to emphasize is that you're listening to people's concerns and plan to address them at the appropriate time. On a rapid-development project, the appropriate time is often "Next project."

A useful adjunct to the version-2 strategy is to create a "multi-release technology plan," which maps out a multi-year strategy for your product. That helps people to relax and see that the feature they want will be more appropriate for some later release (McCarthy 1995a),

Short release cycles

One of the keys to users and customers agreeing to the version-2 approach is that they have some assurance that there will in fact be a version 2. If they fear that the current version will be the last version ever built, they'll try harder to put all their pet features into it. Short release cycles help to build the user's confidence that their favorite feature will eventually make it into the product. The incremental development approaches of evolutionary delivery, evolutionary prototyping, and staged delivery can help.

Changeboard



Formal change-control boards have proven effective against creeping requirements for large projects (Jones 1994). They can also be effective for small projects.

Structure. The change board typically consists of representatives from each party that has a stake in the product's development. Concerned parties

options that are more subtle and often more effective than "you're fired!" are available.

- Change the manager's boss. Sometimes a manager needs different leadership.
- Move the manager into a participatory role. Sometimes a technically-oriented manager can make a technical contribution that will help the project succeed more than his or her leadership contribution can.
- Provide the manager with an assistant. Depending on what's needed, the assistant either can focus on technical details, freeing up the manager to concentrate on big-picture issues, or can handle administrative issues, freeing up the manager to focus on technical matters. In the extreme case, sometimes, the "assistant" can take over nearly all of the manager's responsibilities, leaving the manager in place to handle administrative duties and reports to upper management.

These points focus on management changes, but they apply just as well to changes in the project's technical leadership.



Add people carefully, if at all. Remember Brooks's law that adding people to a late project is like pouring gasoline on a fire (Brooks 1975). Don't add people to a late project willy-nilly.

But remember the whole law. If you can partition your project's work in such a way that an additional person can contribute without interacting with the other people on the project, it's OK to add a person. Think about whether it makes sense to add someone who will spend 8 hours doing what an existing developer could do in 1 hour. If your project is that desperate, go ahead and add someone. But stick to the plan. Some people can't abide watching another person spend 8 hours on a 1-hour job regardless of their original intentions. Know what kind of person you are. If you think you might err, err on the side of not adding anyone.

Focus people's time. When you're in project-recovery mode, you need to make the best possible use of the people who are already familiar with the project. Consider taking the money you would have spent adding people and use it instead to focus the efforts of your existing people. You'll come out ahead.

You can focus existing people in a variety of ways. Give them private offices. Move them off-site. Be sure that they are not distracted by other projects within your organization, so relieve them of tech-support duty, maintenance of other systems, proposal work, and all of the other responsibilities that eat into a developer's time. The point is not to hold their noses to the grindstone, but to relieve them of all nonessential tasks.

If you must hire additional people, consider not hiring developers. Hire administrative people -who can take care of clerical work and help your developers minimize personal downtime (for example, laundry, shopping, bill paying, yard work, and so on).

CROSS-REFERENCE

Allowing different levels of commitment is different at the beginning of a project. For details, see Chapter 34, "Signing Up."

Allow team members to be different. Some people will rise to the challenge of project recovery and become heroes. Others will be too burned out and will refuse to give their all. That's fine. Some people want to be heroes, and other people don't. In the late stages of a project, you have room for quiet, steady contributors who don't rise to heroic heights but who know their way around the product. What you don't have room for are loud naysayers who chide their heroic teammates for being heroic. Morale during project recovery is fragile, and you can't tolerate people who bring the rest of the team down.

CROSS-REFERENCE

For more on seeing that developers pace themselves, see Section 43.1, "Using Voluntary Overtime."

See that developers pace themselves. Runners run at different speeds depending on the distance to the finish line. Runners run faster toward a nearby finish line than they do toward a finish line that's miles away. The best runners learn to pace themselves.

Allow your team to break the vicious circle of schedule pressure leading to stress leading to more defects leading to more work leading back to more schedule pressure. Ease the schedule pressure, give the developers time to focus on quality, and the schedule will follow.

Process

Although you'll find your greatest leverage in the area of people, you must also clean up your process if you want to rescue a project that's in trouble.

Identify and fix classic mistakes. Survey your project to see whether you're falling victim to any of the classic mistakes. Here are the most important questions to ask:

- Is the product definition still changing?
- Is your project suffering from an inadequate design?
- Are there too few management controls in place to accurately track the project's status?
- Have you shortchanged quality in the rush to meet your deadline?
- Do you have a realistic deadline? (If you've slipped your schedule two or more times already, you probably don't.)
- Have people been working so hard that you risk losing them at the end of the project or earlier? (If you've already lost people, they're 'working too hard.)
- Have you lost time by using new, unproved technology?

CROSS-REFERENCE
For a list of many more classic mistakes, see Section 3.3, "Classic Mistakes Enumerated."

- Is a problem developer dragging the rest of the group down?
- Is team morale high enough to finish the project?
- Do you have accountability leaks? People or groups who might mean well but who have not been accountable for the results of their work?

Fix the parts of your development processes that are obviously broken.

When a project is in trouble, everyone usually knows that a few parts of the process are broken. This is where back-to-basics really comes into play—the broken parts are often broken because the project has consciously or unconsciously been ignoring the software fundamentals.

If the team is tripping over itself because you haven't set up version control, set up version control. If you're losing track of the defects being reported, set up a defect tracking system. If end-users or the customer have been adding changes uncontrollably, set up a change-control board. If the team hasn't been able to concentrate because of a steady stream of interruptions, move them off-site, have the facilities group physically wall-off their area, or put up your own floor-to-ceiling boundary with empty computer boxes. If people haven't been getting the timely decisions they need, set up a war room: meet at 5:00 p.m. every day and promise that anybody who needs a decision will get one.

CROSS-REFERENCE
For details, see Chapter 27,
"Miniature Milestones,"

Create detailed miniature milestones. In rescuing a drowning project, it is absolutely essential that you set up a tracking mechanism that allows you to monitor progress accurately. This is your key to controlling the rest of the project. If the project is in trouble, you have all the justification you need to set up miniature milestones.

Miniature milestones allow you to know on a day-by-day basis whether your project is on schedule. The milestones should be miniature, binary, and exhaustive. They're *miniature* because each of them can be completed in one or two days, no longer. They're *binary* because either they're done or they're not—they're not "90 percent done." They're *exhaustive* because when you check off the last milestone, you're done with the project. If you have tasks that aren't on the milestone schedule, add them to the schedule. No work is done "off schedule."

Setting and meeting even trivial milestones provides a boost to morale. It shows that you can make progress and that there's a possibility of regaining control.

One of the biggest problems with setting up mini milestones at the beginning of a project is that you don't know enough to identify all the work in detail. In project-recovery mode, the situation is different. At that late stage in the project, developers have learned enough about the product to be able to say in detail what needs to be-done. Thus mini milestones are particularly appropriate for use in project recovery;

Set up a schedule linked to milestone completion. Plan completion dates for each mini milestone. Don't plan on massive overtime: that hasn't worked so far, and it won't work going forward. If you plan massive overtime into your schedule, developers can't catch up by working more overtime when they get behind. Set the schedule so that if developers get behind on their miniature milestones, they can catch up by working overtime the same day. That allows them to stay on schedule on a day-by-day basis. If you stay on schedule day by day, you stay on schedule week by week and month by month, and that's the only way it's possible to stay on schedule for a whole project.

Track schedule progress meticulously. If you don't track progress after you set up the mini milestones, the schedule-creation process will have been just an exercise in wasting time. Check with developers daily to assess their progress against the mini milestones. Be sure that when a milestone is marked "done" it is truly, 100 percent done. Ask the developer, "If I take the source code for this module that's 'done' and lock it in a vault for the rest of the project, can we ship it? Do you still have some tweaking or polishing to do, or is it 100 percent done?" If the developer says, "It's 99 percent done," then it's *not done*, and the milestone has not been met.

Do not allow developers to get off track on their mini-milestone schedules. The easiest way to get off track is to miss one milestone and then to stop keeping track. A 1-day slip turns into a 2-day slip, which turns into 3 days, and then into a week or more. Soon there is no correspondence between the developer's work and the milestone schedule. Once a schedule has been calibrated, do not take schedule slips lightly. If a single developer falls behind on a single milestone, expect him or her to work overtime that day to catch up. (If a developer meets a single milestone early, it's OK to allow him or her to go home early that day.) Daily milestones must be met consistently or the schedule must be recalibrated so that they can be met consistently.

Record the reasons for missed milestones. Having a record of the reasons that each milestone was missed can help to detect the underlying causes. A record might point to an individual developer's need for training or highlight organizational dynamics that make it hard for any developers to make good on their estimates. It can help to distinguish between estimate-related problems and other schedule-related problems.

Recalibrate after a short time—one or two weeks. If a developer consistently misses milestones and falls more than $\frac{1}{2}$ day behind, it's time to recalibrate that developer's schedule. Recalibrate by increasing the current schedule by the percentage of the slip so far. If the developer has needed 7 days to do 4 days' work, multiply the rest of the work by⁷A. Don't play games by thinking that you'll make up the lost time later. If you're in project-recovery mode, that game has already been lost.

Never trade a bad date for an equally bad date. That's a bad deal. You're just hemorrhaging credibility if you do that.

Jim McCarthy

Don't commit to a new schedule until you can create a meaningful one. Do not give a new schedule to upper management until after you have created a mini-milestone schedule, worked to it for at least a week or two, recalibrated, and worked a week or two more to check your recalibration. Giving a new schedule to management any other way is tantamount to replacing one bad schedule with a different but equally bad schedule. If you follow these steps first, you will have a more solid basis for your future schedule commitments.

Manage risks painstakingly. Focus on risk management using the guidelines spelled out in Chapter 5, "Risk Management." Create a top-10 risks list, and hold daily risk-monitoring meetings. You can expand the 5:00 p.m. war-room meetings to review risks and address new issues that have arisen as well as to provide timely decisions.

Product

It's often not possible to recover a project until you rein in the product's feature set.

CROSS-REFERENCE
For details on change control,
see Section 14.2, "Mid-Project:
Feature-Creep Control."¹¹

Stabilize the requirements. If requirements have been changing throughout the project, you don't need to look any further for the source of your problems. You must stabilize requirements before you can bring your project to a successful conclusion. A system with significantly changing requirements cannot be developed quickly and often cannot be developed at all.

It is not uncommon to need to do a nearly complete requirements specification at this stage. Some products change so much during their development that, by the time the crisis hits, no one knows for sure what features the product is supposed to contain. Developers and testers are working on features that might or might not need to be in the product.

Some projects will resist the work involved with formalizing a statement of requirements this late in the project, but keep in mind that *the other approach is the one that's causing all the problems*. You have to do something different, and you have to know what your feature set is before you can finish the product, before you can even be sure that the development team is working on the product you want.

If the project has been running for some time, formalizing requirements will be a painful step because it will involve eliminating some people's pet features. That's too bad, but it should have been done early on, and it has to be done before you can complete the project. If you can't get the parties involved to commit to a set of requirements when the project is hanging on by its fingernails in recovery mode, then you might as well give up. You're fighting a battle you can't win.

After you do get a set of requirements, set the bar for accepting changes very high. Require a full day even to consider a change. Require more than a day as the minimum time needed to implement a change. (This is for feature changes, not defect corrections.)

CROSS-REFERENCE

For more on trimming requirements, see "Requirements Scrubbing" in Section 14.1.

Trim the feature set. Recovery mode presents an opportunity to reduce the requirements to the minimal acceptable set. Cut low-priority features ruthlessly. You don't need to fix everything, and you don't need to implement every feature. Prioritize. Remember, the real problem at this stage is not developing the product in the shortest possible time or creating the best possible product: it's completing the product at all. Worry about low-priority features on the next version.

People should be ready and willing to define a minimal feature set at this point. If they aren't willing to sacrifice pet features when the project is in recovery mode, they probably 'won't ever be willing to.

Assess your political position. If people aren't willing to freeze requirements or fall back to minimal requirements, this is a good time to take a step back and look at what's really happening on your project. Think about why the other parties are still not focused on the product. What are they focused on? What is more important to them than the product? Are they focused on a power struggle? Are they focused on making you or your boss look bad? As ugly as organizational politics can be, they do exist, and an unwillingness to make crucial compromises when there's no other choice is a telltale sign. If you're caught in the middle of a political skirmish rather than a product development, the project-recovery plan in this chapter won't be of much help, and you should choose your actions accordingly.

CROSS-REFERENCE

For details, see "Error-prone modules" in Section 4.3.

Take out the garbage. Find out if there are any parts of the product that everyone knows are extremely low quality. When you've found a lot of defects in a particular piece of code, it's tempting to think that you've found the last one. But buggy modules tend to produce an amazing, continuing stream of defects. Error-prone modules are responsible for a disproportionate amount of the work on a project, and it is better to throw them out and start over than to continue working with them.

Throw them out. Go through a design cycle. Review the design. Implement the design. Review the implementation. This will seem like work that you can't afford when you're in recovery mode, but what will really kill you in recovery mode is getting nickeled-and-dimed to death by an uncontrollable number of defects. Systematic redesign and implementation reduces your risk.

Reduce the number of defects, and keep them reduced. Projects that are in schedule trouble often start to focus on schedule and expedient shortcuts to

the exclusion of quality. Those compromises invariably return to haunt the developers before the product is released. If you've been in 3-weeks-to-ship mode for awhile, you've almost certainly been making quality compromises and shortcuts during that time that will make your project take longer rather than shorter.

Start using an "open-defects" graph, and update it daily. Figure 16-2 is an example.

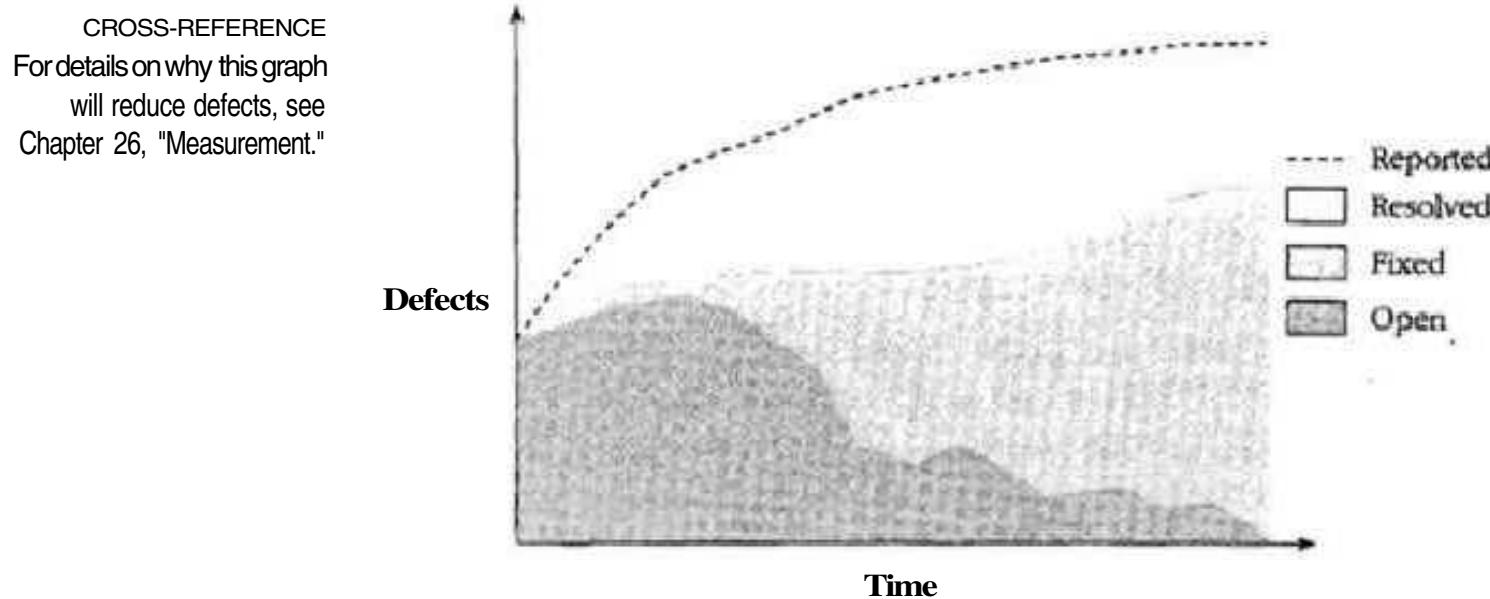


Figure 16-2. Example of an "open defects" graph. Publishing this graph emphasizes that reducing defects is a high priority and helps to gain control on projects with quality problems.

The goal of the open-defects graph is to emphasize how many open defects there are and to highlight the priority of reducing them. Bring the number of open defects down to a manageable level, and keep it there. Start using design and code reviews to maintain a consistent level of low defects. Development time is wasted working and reworking low-quality software. Focusing on quality is one way to reduce development time, and doing so is essential to project recovery.

CROSS-REFERENCE
For details, see Chapter 18, "Daily Build and Smoke Test."

Get to a known good state, and build on that. Plot out the shortest possible course from wherever your product is now to a state at which you can build and test some subset of it. When you get the product to that point, use daily builds to keep the product in a buildable, testable state every day. Add code to the build, and make sure that the code you add doesn't break the build. Make maintaining the build a top priority. Some projects have developers wear beepers and require them to fix the build day or night if they are responsible for breaking it.

Timing

Surprisingly, the best time to launch a project-recovery plan might not be the first moment you notice your project is in trouble. You need to be sure that your management and the development team are ready to hear the message and ready to take the steps needed to truly recover the project. This is something that you need to do right the first time.

You have to walk a line between two considerations. If you launch your recovery plan too early, people won't yet believe that there's a need for it. It is not in your best interest to cry "Wolf!" before other people can see that the wolf is there. If you launch it too late, it will likely follow on the heels of a series of small corrections or mini-recovery attempts that didn't completely work, and you will have undermined your credibility in leading a more effective, larger-scale recovery effort. You will have cried "Wolf!" too many times.

I don't recommend letting your project run into the weeds merely so that you can do a better job of rescuing it. However, if your project has already run into the weeds, I do recommend that you time the presentation of your recovery plan so that everyone else on the project will be receptive enough for it to succeed.

Case Study 16-2. A Successful Project Recovery

End-user reaction to canceling the new inventory-tracking system had been fierce, and a few weeks after Bill canceled the project, he reconsidered. He concluded that they should finish it after all. By that time, Keiko, the contractor, had moved on to a different project. Kip had been reassigned to a short-term project but could be brought back to the team. Jennifer and Joe were just returning from vacation, and Bill thought they might be ready to try again. He called Carl, who had been team lead on the canceled project, into his office. Carl saw a stranger in his boss's office.

"Carl, I've decided to resuscitate ICS 2.0, and I'm going to give you another chance. Meet Charles. He's a project-recovery expert, and I've hired him to help you bring this sucker in. He's already told me that you can't come up with a new schedule right away. He said it might take a few weeks before we know exactly how long it will take to fix the project. I really got my butt kicked for canceling this thing, so now we've got to finish it no matter what. Let me know as soon as you have a new schedule."

Carl was glad to get another chance at rescuing the project. He had thought of some things he could do better, and he knew that Jennifer and Joe had been depressed about the project being canceled. He and Charles left Bill's office together.

(continued)

Case Study 16-2. A Successful Project Recovery, *continued*

Charles started talking. "From what I've heard, I think the main task here is just to finish the project. I'd like to identify each group's win conditions, and then manage the rest of the project so that those are met. Based on the end-users I've talked to, getting a replacement for the old system by the end of the year would be a win as long as it fixes a few long-standing problems. I've got a list of the major problems, and I got the end-users to agree that the rest could wait. Of course they'd like the next release sooner, but they really just want a guarantee that they'll get it eventually."

"Bill's win condition is the same. He wants to follow through with the user group. What do you need to make this a win for you?"

Carl thought a minute. "I need to show that I can rescue this project and meet everybody else's win conditions. I've had a rest, and I can work as hard as I need to." Later that day, Carl talked to Jennifer, Joe, and Kip. Their win conditions were that they wanted to finish the job they'd started, and they wanted to lead normal lives outside work while they were doing that.

"I can't sacrifice the rest of my life to this project anymore," Jennifer said. "Even after 3 weeks of vacation I'm too burned out to do that. It would be nice to finish this project, but I'd rather work on a different project and never finish this one than get that burned out again." Kip said he was willing to work hard, but Joe said he felt the same as Jennifer.

Charles asked the team what they thought needed to be done to save the project, and Jennifer and Joe were in complete agreement. "We were in such a hurry last time that we took all kinds of low-quality shortcuts. We need to go back and clean up some of the product. We shouldn't add any new people this time, either." Carl agreed. He didn't want to make the same mistake twice.

Charles stepped in. "What I'd like you all to do is create a detailed list of every task that needs to be done to release the product, and I really mean everything. Rewriting bad modules, fixing the build script, setting up automated version control, documenting old code, duplicating diskettes, talking to end-users on the phone, everything. And I want you to estimate how long each task will take. If you have any tasks that take more than 2 days, I want you to break them up into smaller tasks that take less time. Then we're going to sit down and plan out the rest of the project."

"I want you to know that your estimates aren't commitments. They're just estimates, and nobody outside of this room will know about them until we're confident that they're right. I know I'm asking for a lot of detail, and it will take time to do all those estimates. I wouldn't be surprised if it takes you at least a day to come up with them. But this project is broken, and this is what we need to do to get it back on track."

The developers spent the next 2 days coming up with incredibly detailed task lists. Joe was surprised at some of his estimates. He took tasks that he had

(continued)

Case Study 16-2. A Successful Project Recovery, continued

originally estimated would take 3 days, broke them down into more detail, and found that the sum of the parts for a few of them was more like 5 or 6 days. Charles said he wasn't surprised. The whole group put together a schedule based on the detailed task lists, and Charles told Bill that they would have a revised completion date in about 15 days.

Carl and Charles checked the team's progress every day for the next week. Kip completed his tasks consistently on time. Jennifer found that one day she finished all of her work by mid afternoon, and one day she had to work until 9:00 in the evening. She was getting the work done, but by the end of the week she had logged almost 50 hours. She told Carl that that was too much. Joe had had trouble completing his tasks on time, and by the end of the week he had completed only half of what he had planned.

The team met to look over their progress. Charles insisted that they recalibrate Joe's schedule by multiplying all of his estimates by 2.0. Even though Jennifer was meeting her deadlines, he reminded them that Jennifer's win condition included leading a normal life outside of work, and they recalibrated her schedule by multiplying her estimates by 1.25. The recalibration made everyone's schedules come out uneven, so they reshuffled the work so that everybody on the team had about the same amount of work.

Carl was surprised at the result. If their estimates were right, they would finish the project in 10 weeks, which wasn't nearly as bad as he had feared. "Should I give Bill the good news?" he asked Charles.

"No, we'll work another week to the recalibrated schedule, and if we're hitting the mini milestones consistently, then we'll tell Bill. But we will let Bill know that we'll have a revised schedule for him a week from Monday."

The next week went surprisingly smoothly. Carl continued to check with each developer every day to be sure that each milestone task was getting done, and each one was. Jennifer stayed late one night, but she told Carl that was mainly because she had goofed off part of the day, not because she had too much work to do. By the end of the week, everybody was on-schedule. More important, everybody was happy. Jennifer had originally thought she would be annoyed by the mini milestones' micro management, but it actually felt good to be able to check off a task every day and to tell someone that she was making progress. Morale had improved.

Carl and Charles told Bill that they would be done in 9 weeks. Bill said that was good news and had been worth the wait. For the remainder of the project, Charles and Carl continued to check progress daily. Each person put in a few late nights to keep to their mini-milestone schedules, but by the end of the 9 weeks they were really and truly finished. They delivered the software to their end-users and notified Bill. Everyone considered the project a win.

Further Reading

McCarthy, Jim. *Dynamics of Software Development*. Redmond, Wash.: Microsoft Press, 1995. This is an entertaining set of lessons that McCarthy learned from his experiences working on Microsoft's Visual C++ and other products. McCarthy describes an enthusiastic but essentially grim vision of software development at Microsoft. He presents Microsoft projects as spending nearly all their time doing what this chapter has called "project recovery." If you recognize that that is what McCarthy is writing about and read his book on that level, he has some valuable things to say.

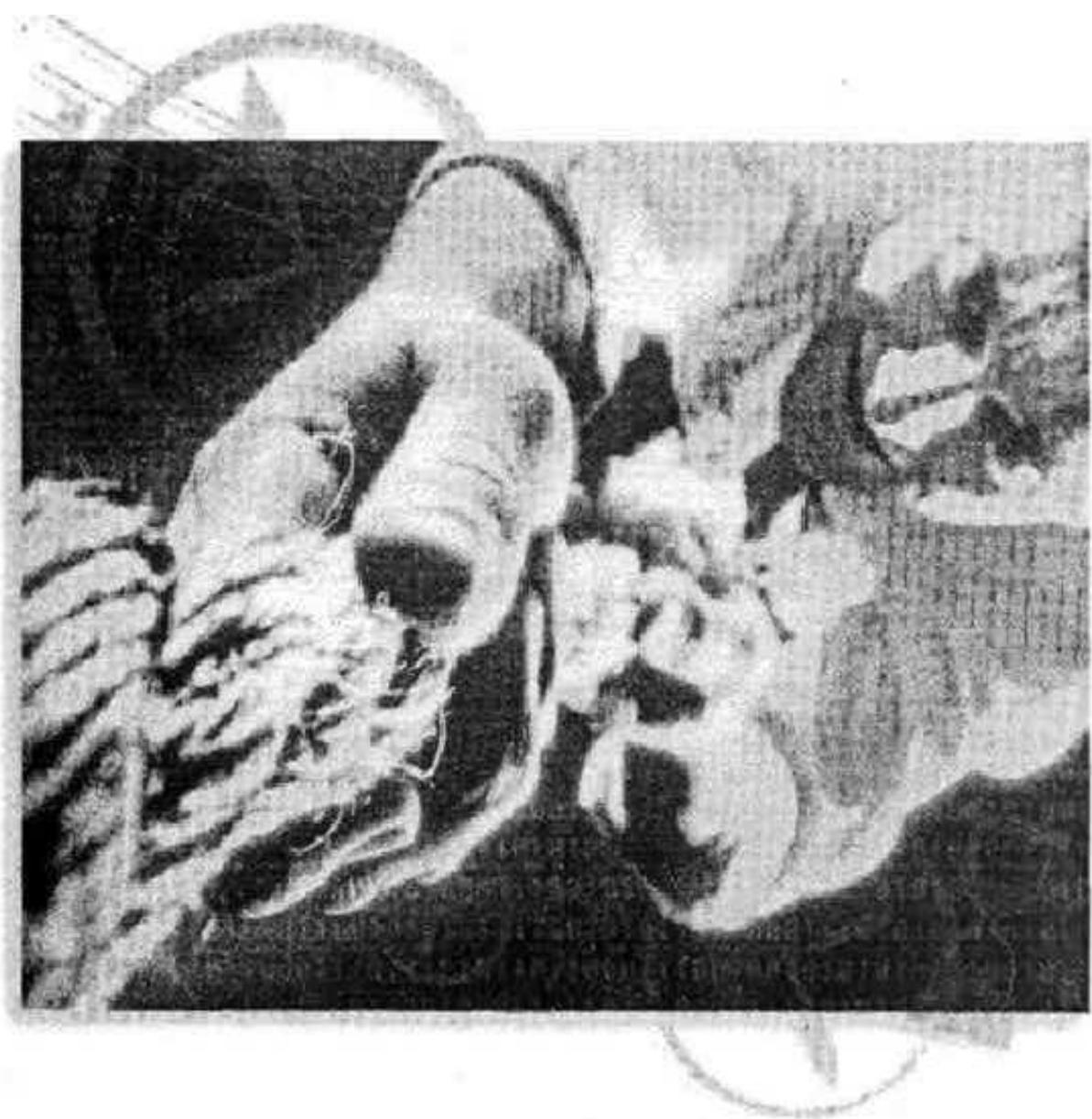
Zachary, Pascal. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*, New York: Free Press, 1994. This is a description of the development of Microsoft Windows NT 3.0. According to the author's description, the NT project spent more time in recovery mode than in normal development. Like McCarthy's book, if you read this book as a project-recovery fable, you can gather some valuable lessons. Once you've read the book, you'll be glad that you didn't learn these particular lessons firsthand! ,

Boddie, John. *Crunch Mode*. New York: Yourdon Press, 1987. This book is not specifically about project recovery, but it is about how to develop software under tight schedules. You can apply a lot of Boddie's approach to a project in recovery mode.

Weinberg, Gerald M. *Quality Software Management, Volume 1: Systems Thinking*. New York: Dorset House, 1992. Pressure is a constant companion during project recovery. Chapters 16 and 17 of this book discuss what Weinberg calls "pressure patterns." He describes what happens to developers and leaders under stress as well as what to do about it.

Thomsett, Rob. "Project Pathology: A Study of Project Failures," *American Programmer*, July 1995, 8-16. Thomsett provides an insightful review of the factors that get projects into trouble in the first place and of early warning signs that they're in trouble.

BEST PRACTICES



P
A
R
T
II

On smaller projects you might not need to devote a whole person to the daily build. In that case, put someone from quality assurance in charge of it.

Add code to the build only when it makes sense to do so... Although daily builds require that you build the system daily, they do not require that developers add every new line of code they write to the system every day. Individual developers usually don't write code fast enough to add meaningful increments to the system every day. They work on a chunk of code and then integrate it when they have a collection of code in a consistent state. Developers should maintain private versions of the source files they're working on. Once they have created a complete set of modifications, they make a private build of the system using their modifications, test it, and check it in.

...but don't wait too long to add code to the build. Although few developers will check in code every day, beware of developers who check in code infrequently. It's possible for a developer to become so embroiled in a set of revisions that every file in the system seems to be involved. That undermines the value of the daily build. The rest of the team will continue to realize the benefit of incremental integration but that particular developer will not. If a developer goes more than a few days without checking in a set of changes, consider that developer's work to be at risk.



- **FURTHER READING**
- For many details on developer-level testing practice, see *Writing Solid Code* (Maguire 1993).

Require developers to smoke test their code before adding it to the system. Developers need to test their own code before they add it to the build. A developer can do this by creating a private build of the system on a personal machine, which the developer then tests individually. Or the developer can release a private build to a "testing buddy," a tester who focuses on that developer's code. The goal in either case is to be sure that the new code passes the smoke test before it's allowed to influence other parts of the system.

Create a holding area for code that's to be added to the build. Part of the success of the daily-build process depends on knowing which builds are good and which are not. In testing their own code, developers need to be able to rely on a known good system.

Most groups solve this problem by creating a holding area for code that developers think is ready to be added to the build. New code goes into the holding area, the new build is built, and if the build is acceptable, the new code is migrated into the master sources.

On small- and medium-sized projects, a version-control system can serve this function. Developers check new code into the version-control system. Developers who want to use a known good build simply set a date flag in their version-control—options file that tells the system to retrieve files based on the date of the last-known good build.,

On large projects or projects that use unsophisticated version-control software, the holding-area function has to be handled manually. The author of a set of new code sends email to the build group to tell them where to find the new files to be checked in. Or the group establishes a "check-in" area on a file server where developers put new versions of their source files. The build group then assumes responsibility for checking new code into version control after they have verified that the new code doesn't break the build.

Create a penalty for breaking the build. Most groups that use daily builds create a penalty for breaking the build. If the build is broken too often, it's hard for developers to take the job of not breaking the build seriously. Breaking the build should be the exception, not the rule. Make it clear from the beginning that keeping the build healthy is the project's top priority. Refuse to take a broken build casually. Insist that developers who have broken the build stop their other work until they've fixed the build.

A fun penalty can help to emphasize the priority of keeping the build healthy. Some groups give out suckers to developers who break the build. The sucker who broke the build has to tape a sucker to his office door until he fixes the build. Other projects make the person who broke the build responsible for running the build until someone else breaks it. On one project I worked on, the person who broke the build had to wear a hat made out of an umbrella until he fixed the build. Other groups have guilty developers wear goat horns or contribute \$5 to a morale fund.

Some projects establish a penalty with more bite. Microsoft developers on high-profile projects such as NT, Windows 95, and Excel have taken to wearing beepers in the late stages of their projects. If they break the build, they are called in to fix the build regardless of the time of day or night. The success of this practice depends on the build group being able to determine precisely who broke the build. If they aren't careful, a developer who's mistakenly called to fix the build at 3:00 in the morning will quickly become an enemy of the daily-build process.

Release builds in the morning. Some groups have found that they prefer to build overnight, smoke test in the early morning, and release new builds in the morning rather than the afternoon. There are several advantages to smoke testing and releasing builds in the morning.

First, if you release a build in the morning, testers can test with a fresh build that day. If you generally release builds in the afternoon, testers feel compelled to launch their automated tests before they leave for the day. When the build is delayed, which it often is, the testers have to stay late to launch their tests. Because it's not their fault that they have to stay late, the build process becomes demoralizing.

When you complete the build in the morning, you have more reliable access to developers when there are problems with the build. During the day, developers are down the hall. During the evening, developers can be anywhere. Even when developers are given beepers, they're not always easy to locate.

It might be more macho to start smoke testing at the end of the day and call people in the middle of the night when you find problems, but it's harder on the team, it wastes time, and in the end you lose more than you gain.



CLASSIC MISTAKE

Build and smoke even under pressure. When schedule pressure becomes intense, the work required to maintain the daily build can seem like extravagant overhead. The opposite is true. Under pressure, developers lose some of their discipline. They feel pressure to take design and implementation shortcuts that they would not take under less stressful circumstances. They review and unit test their own code less carefully than usual. The code tends toward a state of entropy more quickly than it would during less stressful times.

Against this backdrop, the daily-build-and-smoke-test process enforces discipline and keeps pressure-cooker projects on track. The code still tends toward a state of entropy, but you bring that tendency to heel every day. If you build daily and the build is broken, identifying the developer responsible and insisting on an immediate code fix is a manageable task. Bringing the code back from its state of entropy is a relatively small undertaking.

If you wait two days—until twice as many defects are inserted into the code—then you have to deal both with twice as many defects and with multiplicative interaction effects of those defects. It will take more than twice as much effort to diagnose and correct them. The longer you wait between builds, the harder it is to bring the build back into line.

What Kinds of Projects Can Use the Daily-Build-and-Smoke-Test Process?

Virtually any kind of project can use daily builds—large projects, small projects, operating systems, shrink-wrap software, and business systems.

Fred Brooks reports that software builders in some organizations are surprised or even shocked by the daily-build process. They report that they build every week, but not every day (Brooks 1995). The problem with weekly builds is that you tend not to build every week. If the build is broken one week, you might go for several weeks before the next good build. When that happens, you lose virtually all the benefit of frequent builds.

Some developers protest that it is impractical to build every day because their projects are too large. But the project that was perhaps the most complex software-development effort in recent history used daily builds successfully. By the time it was first released, Microsoft NT consisted of 5.6 million lines of code spread across 40,000 source files. A complete build took as many as 19 hours on several machines, but the NT development team still managed to build every day (Zachary 1994). Far from being a nuisance, the NT team attributed much of their success on that huge project to their daily builds. Those of us who work on projects of less staggering proportions will have a hard time explaining why we don't use daily builds.

For technical leads, daily builds are particularly useful because you can implement them at a purely technical level. You don't have to have management authority to insist that your team build successfully every day.

CROSS-REFERENCE

For more on project recovery, see Chapter 16, "Project Recovery."

Daily builds are especially valuable in project-recovery mode. If you can't get to a good-build state, you have no hope of ever shipping the product, so you might as well work toward a good build as one of your first project-recovery objectives. Once you get to a known good state, you make incremental additions and stay in a known good state. It's a morale boost during project recovery to have a working product at all times, and it makes for clear progress indications.

18.2 Managing the Risks of the Daily Build and Smoke Test

The daily-build process has few drawbacks. Here is the main one.

CROSS-REFERENCE

The problems of premature releases are related to the problems of premature convergence discussed in "Premature convergence" in Section 9.1.

Tendency toward premature releases. When people outside the development group see that the product is being built every day, pressure mounts to create frequent releases for outside groups. Creating external releases can look easy to a product manager, and it is easier than when you're not using daily builds, but it still sucks up developers' time in subtle ways:

- Developers spend time preparing materials that are not needed for the final product but that are needed to support the release. These materials include documentation, interim versions of features still under development, stubbing out hazardous areas of the product, hiding debugging aids, and so on.
- Developers make quick fixes so that features will work for a particular release rather than waiting until they can make more careful changes. These quick fixes eventually break, and the developers then have to make the more careful changes that they should have made in the first place. The net effect is that the developers waste time fixing the same code twice.

- Developers spend more time responding to minor problems on an ad hoc basis early in the development cycle, problems that could be taken care of more efficiently as part of the developers' normal work cycle.

You wouldn't want to eliminate interim releases entirely. What you can do is to plan for a specific number of interim releases and then try not to increase that number.

18.3 Side Effects of the Daily Build and Smoke Test

Some developers who have used daily builds claim that it improves overall product quality (Zachary 1994). Other than that general claim, the daily build's effect is limited to its improvements in integration risk, quality risk, and progress visibility.

18.4 The Daily Build and Smoke Test's Interactions with Other Practices

Daily builds combine nicely with the use of miniature milestones (Chapter 27). As Chris Peters says, "Scheduling rule #1 is constant vigilance" (Peters 1995). If you have defined a complete set of mini milestones and you know that your daily build is not broken, then you will have exceptional visibility into your progress. You can check the build every single day to determine whether your project is meeting its mini milestones. If it is meeting its mini milestones, it will finish on time. If it is falling behind, you will detect that immediately, and you can adjust your plans accordingly. The only kind of scheduling error you can make is to leave tasks off the schedule.

CROSS-REFERENCE
For more on incremental development practices, see Chapter 7, "Lifecycle Planning."

Daily builds also provide support for incremental-development practices (Chapter 7). Those practices depend on being able to release interim versions of the software externally. The effort required to prepare a good build for release is relatively small compared to the effort needed to convert an average, infrequently built program into a good build.

18.5 The Bottom Line on the Daily Build and Smoke Test

CROSS-REFERENCE
For more on the importance of making progress visible see Section 6.3, "Perception and Reality."

The daily-build-and-smoke-test process is at heart a risk-management practice. Because the risks it addresses are schedule risks, it has a powerful ability to make schedules more predictable and to eliminate some of the risks that can cause extreme delays—integration problems, low quality, and lack of progress-visibility.

I don't know of any quantitative data on the schedule efficacy of the daily-build process, but the anecdotal reports on its value are impressive. Jim McCarthy has said that if Microsoft could evangelize only one idea from its development process, the daily-build-and-smoke-test process would be the one (McCarthy 1995c).

18.6 Keys to Success in Using the Daily Build and Smoke Test

Here are the keys to success in using daily builds:

- Build every day.
- Smoke test every day.
- Grow the smoke test with the product. Be sure that the test remains meaningful as the product evolves.
- Make a healthy build the project's top priority.
- Take steps to ensure that broken builds are the exception rather than the rule.
- Don't abandon the process under pressure.

Further Reading

Cusumano, Michael, and Richard Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York: Free Press, 1995. Chapter 5 describes Microsoft's daily-build process in detail, including a detailed listing of the steps that "an individual developer goes through on applications products such as Microsoft Excel.

McCarthy, Jim. *Dynamics of Software Development*. Redmond, Wash.: Microsoft Press, 1995. McCarthy describes the daily-build process as a practice he has found to be useful in developing software at Microsoft. His viewpoint provides a complement to the one described in *Microsoft Secrets* and in this chapter.

McConnell, Steve. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993. Chapter 27 of this book discusses integration approaches. It provides additional background on the reasons that incremental integration practices such as daily builds are effective.

Designing for Change



Designing for Change is a broad label that encompasses several change-oriented design practices. These practices need to be employed early in the software lifecycle to be effective. The success of Designing for Change depends on identifying likely changes, developing a change plan, and hiding design decisions so that changes do not ripple through a program. Some of the change-oriented design practices are more difficult than people think, but when they are done well they lay the groundwork for long-lived programs and for flexibility that can help to minimize the schedule impacts of late-breaking change requests.

Efficacy

Potential reduction from nominal schedule:	Fair
Improvement in progress visibility:	None
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Good
Chance of long-term success:	Excellent

S
U
M
M
A
R
Y

Major Risks

- Overreliance on the use of programming languages to solve design problems rather than on change-oriented design practices

Major Interactions and Trade-Offs

- Provides necessary support for incremental-development practices
- Design methods involved work hand-in-hand with software reuse

Many developers recognize designing for change as a "good" software engineering practice, but few recognize it as a valuable contributor to rapid development.

Some of the most damaging influences on a development schedule are late, unexpected changes to a product—changes that occur after initial design is complete and after implementation is underway. Projects that deal poorly with such changes are often perceived as slow even if they were on schedule up to the point where the changes were introduced.

Current development practices place an increasing emphasis on responding to changes—changes in market conditions, changes in customer understanding of the problem, changes in underlying technology, and so on.

Against this backdrop, Designing for Change does not produce direct schedule savings. The care needed to design for change might actually lengthen the nominal development schedule. But the schedule equation is seldom so simple as to contain the nominal schedule as its only variable. You must factor in the likelihood of changes, including changes in future versions. Then you must weigh the small, known increase in effort needed to design for change against the large potential risk of not designing for change. On balance, it is possible to save time. If you use incremental-development practices such as Evolutionary Delivery and Evolutionary Prototyping, you build a likelihood of change into the development process, so your design had better account for it.

19.1 Using Designing for Change

"Designing for change" does not refer to any single design methodology, but to a panoply of design practices that contribute to flexible software designs. Here are some of the things you can do:

- Identify areas likely to change.
- Use information hiding.
- Develop a change plan.
- Define families of programs.
- Use object-oriented design.

The rest of this section lays out each of these practices. Some of the practices overlap, but I think that each of them has distinctive heuristic value.

Identify Areas Likely to Change

The first key to success in Designing for Change is to identify the potential changes. Begin design work by listing design decisions that are likely to

change. Robert L. Glass has pointed out that one characteristic of great designers is that they are able to anticipate more kinds of possible change than average designers can (Glass 1994a). Here are some frequent sources of change:

- Hardware dependencies
- File formats
- Inputs and outputs
- Nonstandard language features
- Difficult design and implementation areas
- Global variables
- Implementations of specific data structures
- Implementations of abstract data types
- Business rules
- Sequences in which items will be processed
- Requirements that were barely excluded from the current version
- Requirements that were summarily excluded from the current version
- Features planned for the next version

Identifying changes needs to be done at design time or earlier. Identifying possible requirements changes should be a part of identifying requirements.

Use Information Hiding

CROSS-REFERENCE
For more on information hiding, see "FurtherReading" at the end of the chapter.



Once you have created your list of potential changes, isolate the design decisions related to each of those changes inside its own module. By "module" I am not necessarily referring to a single routine. A module in this context could be a routine or a collection of routines and data. It could be a "module" in Modula-2, a "class" in C++, a "package" in Ada, a "unit" in Turbo, Pascal, or Delphi, and so on.

The practice of hiding changeable design decisions inside their own modules is known as "information hiding," which is one of the few theoretical techniques that has indisputably proven its usefulness in practice (Boehm 1987a). In the time since David Parnas first introduced the technique, large programs that use information hiding have been found to be easier to modify—by a factor of four—than programs that don't (Korson and Vaishnavi 1986). Moreover, information hiding is part of the foundation of both structured design and object-oriented design. In structured design, the notion of black boxes comes from information hiding. In object-oriented design, information hiding gives rise to the notions of encapsulation and visibility.

In the 20th anniversary edition of *The Mythical Man-Month*, Fred Brooks concludes that his criticism of information hiding was one of the few shortcomings of the first edition of his book. "Parnas "was right, and I was wrong about information hiding," he proclaims (Brooks 1995).

To use information hiding, begin design by listing design decisions that are likely to change (as described above) or especially difficult design decisions. Then design each module to hide the effects of changes to one of those design decisions. Design the interface to the module to be insensitive to changes inside the module. That way, if the change occurs, it will affect only one module. The goal should be to create black boxes—modules that have well-defined, narrow interfaces and that keep their implementation details to themselves.



CLASSIC MISTAKE

Suppose you have a program in which each object is supposed to have a unique ID stored in a member variable called *ID*. One design approach would be to use integers for the IDs and to store the highest ID assigned so far in a global variable called *MaxID*. Each place a new object is allocated, perhaps in each object's constructor, you could simply use the statement *ID = ++MaxID*. (This is a C-language statement that increments the value of *MaxID* by 1 and assigns the new value to *ZD*.) That would guarantee a unique ID, and it would add the absolute minimum of code in each place an object is created. What could go wrong with that?

A lot of things could go wrong. What if you want to reserve ranges of IDs for special purposes? What if you want to be able to reuse the IDs of objects that have been destroyed? What if you want to add an assertion that fires when you allocate more IDs than the maximum number you've anticipated? If you allocated IDs by spreading *ID = ++MaxID* statements throughout your program, you would have to change the code associated with every one of those statements.

The way that new IDs are created is a design decision that you should hide. If you use the phrase *++MaxID* throughout your program, you expose the information that the way a new ID is created is simply by incrementing *MaxID*. If, instead, you put the statement *ID = NewID()* throughout your program, you hide the information about how new IDs are created.

Inside the *NewID()* function you might still have just one line of code, *return. (++MaxID)* or its equivalent, but if you later decide to reserve certain ranges of IDs for special purposes, to reuse old IDs, or to add assertions, you could make those changes within the *NewID()* function itself—without touching dozens or hundreds of *ID = NewID()* statements. No matter how complicated the revisions inside *NewID()* might become, they wouldn't affect any other part of the program.

Now suppose that you further discover you need to change the type of the ID from an integer to a string. If you've spread variable declarations like *int ID* throughout your program, your use of the *NewID()* function •won't help. You'll still have to go through your program and make dozens or hundreds of changes.

In this case, the design decision to hide is the ID's type. You could simply declare your IDs to be *ofIDTYPE*—a user-defined type that resolves to *int*—rather than directly declaring them to be of type *int*. Once again, hiding a design decision makes a huge difference in the amount of code affected by a change.

Develop a Change Plan

For the areas that are likely to change, develop change plans. Change plans can prescribe the use of any of the following practices:



FURTHER READING
For details on all these practices, see *Code Complete* (McConnell 1993).

- Use abstract interfaces to modules rather than interfaces that expose the implementation details.
- Use named constants for data-structure sizes rather than hard-coded literals.
- Use late-binding strategies. Look up data-structure sizes in an external file or registry in the Windows environment. Allocate data structures dynamically based on those sizes.
- Use table-driven techniques in which the operation of the program changes based on the data in the table. Decide whether to store the data table inside the program (which will require recompilation to change) or outside the program in a data file, initialization file, Windows registry, or resource file.
- Use routines rather than duplicating lines of code—even if it's only one or two lines.
- Use simple routines that perform single, small functions. If you keep routines simple, they'll be easier to use in ways you didn't originally anticipate.
- Keep unrelated operations separate. Don't combine unrelated operations into a single routine just because they seem too simple to put into separate routines.
- Separate code for general functionality from code for specialized functionality. Distinguish between code for use throughout your organization, for use in a specific application, and for use in a specific version of an application.

These practices are all good software-engineering practices, which, among other things, help to support change.

Define Families of Programs

David Parnas pointed out as early as 1976 that the developer's job had changed from designing individual programs to designing families of programs (Parnas 1976, 1979). In the 20 years since he wrote that, the developer's job has shifted even more in that direction. Developers today use the same code base to produce programs for different languages, different platforms, and different customers. Figure 19-1 illustrates how this is done.

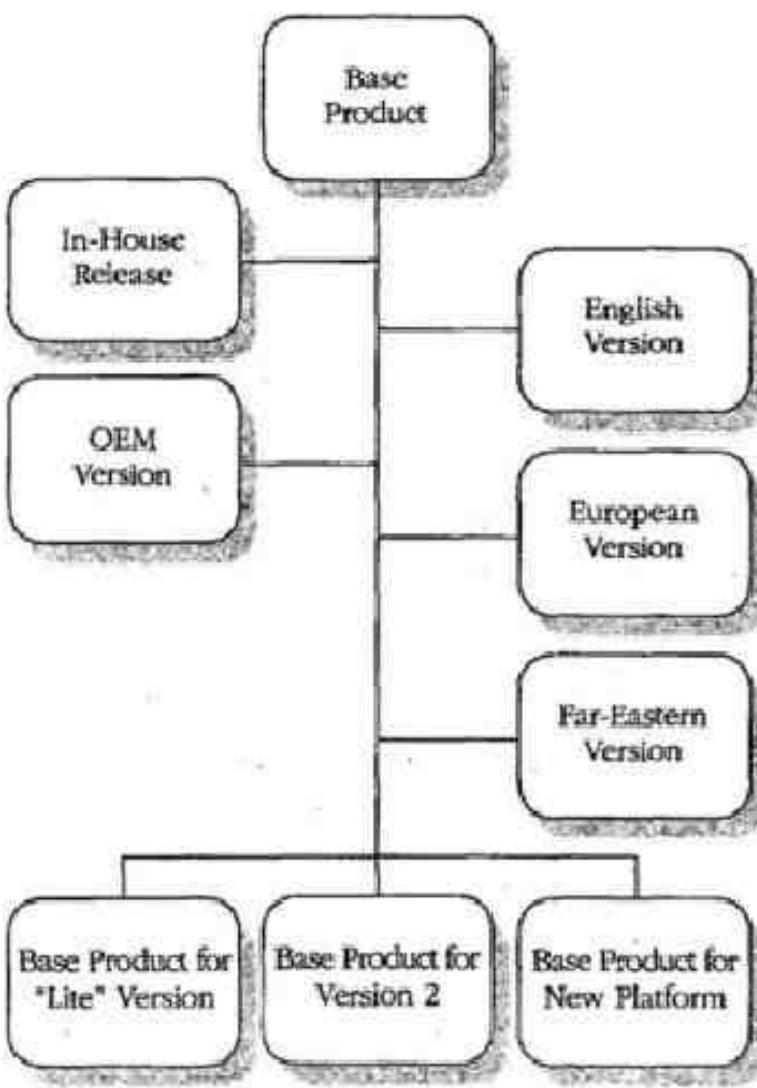


Figure 19-1. Family of products. Since most products eventually turn into a family of products, most design efforts should concentrate on designing a family of programs instead of just one.

In this environment, Parnas argues that a designer should try to anticipate the needs of the family of programs in developing the base product. The designer should anticipate lateral versions such as in-house releases, English, European, and Far-Eastern versions, and the designer should also anticipate follow-on versions. The designer should design the product so that the decisions that are least likely to change among versions are placed closest to the root of the tree. This holds true whether you're consciously designing a

CROSS-REFERENCE

For more on defining minimal feature sets and on versioned-release practices, see Chapter 20, "Evolutionary Delivery"; Chapter 36, "Staged Delivery"; and "Versioned Development" in Section 14.1.

family of programs or just a single program that you'd like to prepare for change.

A good practice is first to identify a minimal subset of functionality that might be of use to the end-user and then to define minimal increments beyond that. The minimal subset usually won't be large enough to make up a program that anyone would want to use; it is useful for the purpose of preparing for change, but it usually isn't worth building for its own sake. The increments you define beyond that should also be so small as to seem trivial. The point of keeping them small is to avoid creating components that perform more than one function. The smaller and sharper you make the components, the more adaptable to change the system will be. Designing minimal components that add minimal incremental functionality also leads to systems in which you can easily trim features when needed.

Use Object-Oriented Design

CROSS-REFERENCE

For more comments on object-oriented programming, see "Identifying Silver Bullets" in Section 15.5.

One of the outgrowths of information hiding and modularity has been object-oriented design. In object-oriented design, you divide a system into objects. Sometimes the objects model real-world entities; sometimes they model computer-science structures or more abstract entities.

A study at NASA's Software Engineering Laboratory found that object-oriented development practices increased reusability, reconfigurability, and productivity, and they reduced development schedules (Scholtz, et al. 1994).

Use object-oriented design, but don't expect it to be a cure-all. The success of object-oriented design in a high-change environment depends on the same factors that information hiding does. You still need to identify the most likely sources of change, and you still need to hide those changes behind narrow interfaces that insulate the rest of the program from potential changes.

19.2 Managing the Risks of Designing for Change



CLASSIC MISTAKE

Using the best practice of Designing for Change poses no risks to the rest of the project. The main risk associated with Designing for Change is simply the risk of failing to use the practice to its full benefit.

Overreliance on languages and pictures rather than on design. The mere act of putting objects into classes does not create an object-oriented design, does not provide information hiding, and does not protect a program from changes. The mere act of drawing a module-hierarchy chart does not create a change-tolerant design. Good designs come from good design work, not from pictures of design.

CROSS-REFERENCE

For another broad view of object-oriented technology, see "Identifying Silver Bullets" in Section 15.5.

Doing object-oriented design effectively is harder than people have made it out to be. As discussed in Section 15.5, it is an expert's technology. David Parnas writes that object-oriented (O-O) programming has caught on slowly for the following reason:

[O-O] has been tied to a variety of complex languages. Instead of teaching people that O-O is a type of design, and giving them design principles, people have been taught that O-O is the use of a particular tool. We can write good or bad programs with any tool. Unless we teach people how to design, the languages matter very little. The result is that people do bad designs with these languages and get very little value from them. (Parnas in Brooks 1995)

To design for change, you must actually *design*. Focusing on areas that are likely to change, information hiding, and families of programs make up the strategic backbone of object-oriented design. If you don't follow those design steps, you might as well be working in Fortran, plain old C, or assembler.

19.3 Side Effects of Designing for Change



Programs that are designed for change continue to yield benefits long after their initial construction. Capers Jones reports that programs that are well-structured and developed with high quality are nearly guaranteed to have long and useful service lives. Programs that are poorly structured and developed with low quality are nearly always taken out of service or become catastrophically expensive to maintain within 3 to 5 years (Jones 1994).

19.4 Designing for Change's Interactions with Other Practices

The flexibility provided by Designing for Change is an important part of the support needed for incremental-development practices such as Evolutionary Delivery (Chapter 20) and Evolutionary Prototyping (Chapter 21). The change-oriented design practices also provide moderate support for Reuse (Chapter 33).

19.5 The Bottom Line on Designing for Change

The bottom line is that Designing for Change is a risk-reduction practice. If the system is stable, it doesn't produce immediate schedule reductions, but it helps to prevent the massive schedule slips that can occur when unanticipated changes cause widespread ripple effects through the design and code.

19.6 Keys to Success in Using Designing for Change

Here are the keys to success in designing for change:

- Identify the most likely changes.
- Use information hiding to insulate the system from the effects of the most likely changes.
- Define families of programs rather than considering only one program at a time.
- Don't count on the mere use of an object-oriented programming language to do the design job automatically.

Further Reading

The three Parnas papers below are the seminal presentations of the ideas of information hiding and designing for change. They are still some of the best sources of information available on these ideas. They might be difficult to find in their original sources, but the 1972 and 1979 papers have been reproduced in *Tutorial on Software Design Techniques* (Freeman and Wasserman 1983), and the 1972 paper has also been reproduced in *Writings of the Revolution* (Yourdon 1982).

Parnas, David L. "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, v. 5, no. 12, December 1972, 1053-58 (also in Yourdon 1979, Freeman and Wasserman 1983).

Parnas, David L. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, v. SE-5, March 1979, 128-138 (also in Freeman and Wasserman 1983).

Parnas, David Lorge, Paul C. Clements, and David M. Weiss. "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, March 1985, 259-266.

McConnell, Steve. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993. Section 6.1 of this book discusses information hiding, and Section 12.3 discusses the related topic of abstract data types. Chapter 30, "Software Evolution," describes how to prepare for software changes at the implementation level.

20

Evolutionary Delivery



Evolutionary Delivery is a lifecycle model that strikes a balance between Staged Delivery's control and Evolutionary Prototyping's flexibility. It provides its rapid-development benefit by delivering selected portions of the software earlier than would otherwise be possible, but it does not necessarily deliver the final software product any faster. It provides some ability to change product direction mid-course in response to customer requests. Evolutionary Delivery has been used successfully on in-house business software and shrink-wrap software. Used thoughtfully, it can lead to improved product quality, reduced code size, and more even distribution of development and testing resources. As with other lifecycle models, Evolutionary Delivery is a whole-project practice: if you want to use it, you need to start planning to use it early in the project.

S
U
M
M
A
R
Y

Efficacy

Potential reduction from nominal schedule:	Good
Improvement in progress visibility:	Excellent
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Very Good
Chance of long-term success:	Excellent

Major Risks

- Feature creep
- Diminished project control
- Unrealistic schedule and budget expectations
- Inefficient use of development time by developers

Major Interactions and Trade-Offs

- Draws from both Staged Delivery and Evolutionary Prototyping
- Success depends on use of designing for change

Some people go to the grocery store carrying a complete list of the groceries they'll need for the week: "2 pounds of bananas, 3 pounds of apples, 1 bunch of carrots," and so on. Other people go to the store with no list at all and buy whatever looks best when they get there: "These melons smell good. I'll get a couple of those. These snow peas look fresh. I'll put them together with some onions and water chestnuts and make a stir-fry. Oh, and these porterhouse steaks look terrific. I haven't had a steak in a long time. I'll get a couple of those for tomorrow." Most people are somewhere in between. They take a list to the store, but they improvise to greater and lesser degrees when they get there.

In the world of software lifecycle models, Staged Delivery is a lot like going to the store with a complete list. Evolutionary Prototyping is like going to the store with no list at all. Evolutionary Delivery is like starting with a list but improvising some as you go along.

The Staged Delivery lifecycle model provides highly visible signs of progress to the customer and a high degree of control to management, but not much flexibility. Evolutionary Prototyping is nearly the opposite: like Staged Delivery, it provides highly visible signs of progress to the customer—but unlike Staged Delivery, it provides a high degree of flexibility in responding to customer feedback and little control to management. Sometimes you want to combine the control of Staged Delivery with the flexibility of Evolutionary Prototyping. Evolutionary Delivery straddles the ground between those two lifecycle models and draws its advantages and disadvantages from whichever it leans toward the most.

CROSS-REFERENCE

For details on these kinds of support for rapid development, see the introductions to Chapter 21, "Evolutionary Prototyping," and Chapter 36, "Staged Delivery."

Evolutionary Delivery supports rapid development in several ways:

- It reduces the risk of delivering a product that the customer doesn't want, avoiding time-consuming rework.
- For custom software, it makes progress visible by delivering software early and often.
- For shrink-wrap software, it supports more frequent product releases.
- It reduces estimation error by allowing for recalibration and reestimation after each evolutionary delivery.
- It reduces the risk of integration problems by integrating early and often—whenever a delivery occurs.
- It improves morale because the project is seen as a success from the first time the product says, "Hello World" until the final version is ultimately delivered.

As with other aspects of Evolutionary Delivery, the extent to which it supports rapid development in each of these ways depends on whether it leans more toward Staged Delivery or Evolutionary Prototyping.

20.1 Using Evolutionary Delivery

To use Evolutionary Delivery, you need to have a fundamental idea of the kind of system you're building at the outset of the project. As Figure 20-1 suggests, in the evolutionary-delivery approach, you start with a preliminary idea of what your customer wants, and you create a system architecture and core based on that. That architecture and core serve as the basis for further development.

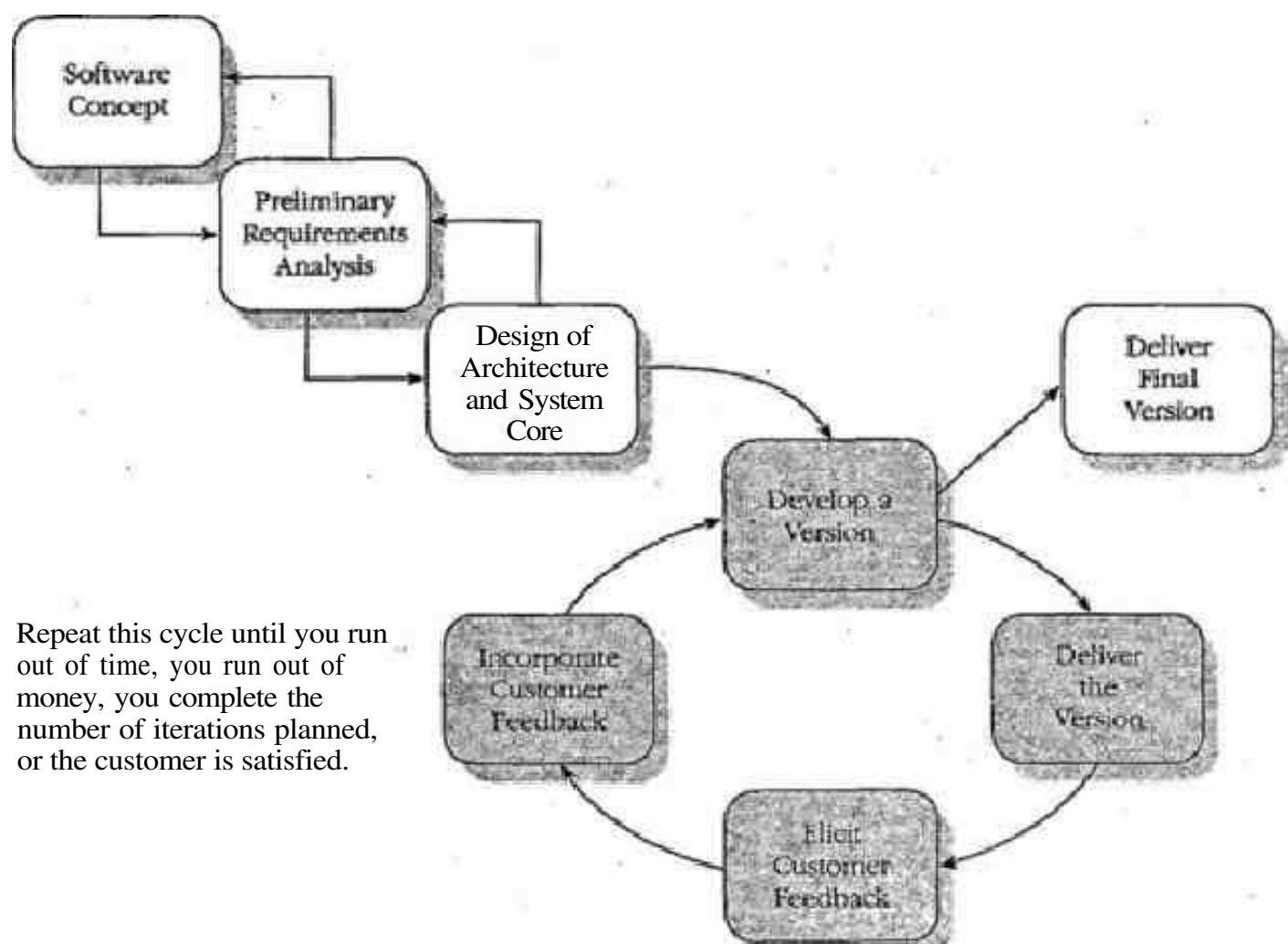


Figure 20-1. *The Evolutionary Delivery lifecycle model draws from Staged Delivery's control and Evolutionary Prototyping's flexibility. You can tailor it to provide as much control or flexibility as you need.*

The architecture should anticipate as many of the possible directions the system could go as it can. The core should consist of lower-level system functions that are unlikely to change as a result of customer feedback. It's fine to be uncertain about the details of what you will ultimately build on top of the core, but you should be confident in the core itself.

Properly identifying the system core is one key to success in using Evolutionary Delivery. Aside from that, the most critical choice you make in Evolutionary Delivery is whether to lean more toward Evolutionary Prototyping or Staged Delivery.

In Evolutionary Prototyping, you tend to iterate until you and the customer agree that you have produced an acceptable product. How many iterations will that take? You can't know for sure. Usually you can't afford for a project to be that open-ended.

In Staged Delivery, on the other hand, you plan during architectural design how many stages to have and exactly what you want to build during each stage. What if you change your mind? Well, pure Staged Delivery doesn't allow for that.

With Evolutionary Delivery, you can start from a base of Evolutionary Prototyping and slant the project toward Staged Delivery to provide more control. You can decide at the outset that you will deliver the product in four evolutionary deliveries. You invite the customer to provide feedback at each delivery, which you can then account for in the next delivery. But the process will not continue indefinitely: it will stop after four deliveries. Deciding on the number of iterations in advance and sticking to it is one of the critical factors to success in this kind of rapid development (Burlton 1992).

With Evolutionary Delivery, another option is to start from a base of Staged Delivery and slant the project toward Evolutionary Prototyping to provide more flexibility. You can decide at the outset what you will deliver in stages 1, 2, and 3, but you can be more tentative about stages 4 and 5, thus giving your project a direction but not an exact road map.

Whether you slant more toward Evolutionary Prototyping or Staged Delivery should depend on the extent to which you need to accommodate customer requests. If you need to accommodate most requests, set up Evolutionary Delivery to be more like prototyping. Some experts recommend delivering the software in increments as small as 1 to 5 percent (Gilb 1988). If you need to accommodate few change requests, plan it to be more like Staged Delivery, with just a handful of larger releases.

Release Order

CROSS-REFERENCE
For details on the value of mapping out possible changes to a system at design time, see "Define Families of Programs" in Section 19.1.

You use Evolutionary Delivery when you're not exactly sure what you want to build. But unlike Evolutionary Prototyping, you do have at least some idea, so you should map out a preliminary set of deliveries at the beginning of your project, while you're developing the system architecture and system core.

Measurement



Measurement is a practice that has both short-term motivational benefits and long-term cost, quality, and schedule benefits. Measurement provides an antidote to the common problems of poor estimates, poor scheduling, and poor progress visibility. Companies that have active measurement programs tend to dominate their industries. Virtually any organization or project can benefit from applying Measurement at some level. For greatest effect, Measurement should have high-level management commitment and be enacted through a permanent measurement group. Measurement can also be implemented to a lesser degree on individual projects by the project team or individual team members.

S
U
M
M
A
R
Y

Efficacy

Potential reduction from nominal schedule:	Very Good
Improvement in progress visibility:	Good
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Good
Chance of long-term success:	Excellent

Major Risks

- Overoptimization of single-factor measurements
- Misuse of measurements for employee evaluations
- Misleading information from lines-of-code measurements

Major Interactions and Trade-Offs

- Provides the foundation for improvements in estimation, scheduling, productivity-tool evaluation, and programming-practice evaluation

Software products and projects can be measured in dozens of ways: size in lines of code or function points; defects per thousand lines of code or function point; hours spent designing, coding, and debugging; developer satisfaction—these are just the tip of the iceberg.

Measurement programs support rapid development in several ways.

Measurement provides status visibility. The only thing worse than being late is not knowing that you're late. Measuring your progress can help you know exactly what your status is.

CROSS-REFERENCE

For more on the importance of objectives, see "Goal Setting" in Section 11.2.

Measurement focuses people's activities. As I've mentioned elsewhere, people respond to the objectives you set for them. When you measure a characteristic of your development process and feed it back to the people involved, you're implicitly telling them that they should work to improve their performance against that characteristic. If you measure the program's bug count and feed that back, they'll reduce the bug count. If you measure the percentage of modules marked as done, they'll increase the percentage of modules marked as done.

What gets measured gets optimized. If you measure developer-speed related characteristics of the project, those will get optimized.

Measurement improves morale. Properly implemented, a measurement program can improve developer morale by bringing attention to chronic problems such as excessive schedule pressure, inadequate office space, and inadequate computing resources.

CROSS-REFERENCE

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

Measurement can help to set realistic expectations. You'll be in a much stronger position to make and defend schedule estimates if you have measurements to support you. When your customer asks you to work to an impossible deadline, you can say something like this: "I will work hard to deliver the system by the time you want it, but historically, based on the figures I have just described to you, it will take longer than your imposed deadline. We might set a new record on this project, but I recommend that you modify your plans in light of our history" (Rifkin and Cox 1991).

Measurement lays the groundwork for long-term process improvement. The most significant benefit of Measurement can't be realized in the short-term on a single project, but it will pay off over two or three years. By measuring your projects on a consistent basis, you lay a groundwork for comparing projects and analyzing which practices work and which don't. A measurement program helps you avoid wasting time on practices that aren't paying off. It helps you identify silver-bullet technologies that aren't living up to their claims. It helps you to accumulate a base of experience that will support more accurate project estimation and more meaningful planning. Measurement is the cornerstone of any long-term process-improvement program.

26.1 Using Measurement

There are several keys to using Measurement effectively.

Goals, Questions, Metrics

Some organizations waste both time and money by measuring more things than they need to. A good way to avoid that problem is to be sure that you're collecting data for a reason. The Goals, Questions, Metrics practice can help (Basil! and Weiss 1984):

- *Set goals.* Determine how you want to improve your projects and products. Your goal, for example, might be to reduce the number of defects put into the software in the first place so that you don't spend so much time debugging and correcting the software downstream.
- *Ask questions.* Determine what questions you need to ask in order to meet your goals. A question might be, "What kinds of defects are costing us the most to fix?"
- *Establish metrics.* Set up metrics (measurements) that will answer your questions. You might start collecting data on defect types, creation times, detection times, cost to detect, and cost to correct.

A review of data collection at NASA's Software Engineering Laboratory concluded that the most important lesson learned in 15 years was that you need to define measurement goals before you measure (Valett and McGarry 1989).

Measurement Group

Some organizations set up a separate measurement group, and that is usually a good idea because effective measurement requires a specialized set of skills. The group can consist of typical developers, but the ideal measurement group would have knowledge of the following areas (Jones 1991):

- Statistics and multivariate analysis
- Literature of software engineering
- Literature of software project management
- Software planning and estimating methods
- Software planning and estimating tools
- Design of data-collection forms
- Survey design
- Quality-control methods, including reviews
- Walk-throughs, inspections, and all standard forms of testing

- Pros and cons of specific software metrics
- Accounting principles

This is the skill-set that experienced measurement groups at AT&T, DuPont, Hewlett-Packard, IBM, and ITT have.

You don't necessarily need to have an organization-level, full-time measurement group to measure aspects of specific projects. A team leader or an individual team member can introduce specific measures at the project level to take advantage of Measurement's short-term motivational benefits.

What to Measure

Each organization needs to decide what to measure based on its own priorities—its own goals and questions. But, at a minimum, most organizations will want to keep historical data on project sizes, schedules[^] resource requirements, and quality characteristics. Table 26-1 lists some of the data elements that different organizations collect.

Table 26-1. Examples of Kinds of Measurement Data

Cost and resource data

Effort by activity, phase, and type of personnel (see also Table 26-2)
 Computer resources
 Calendartime

Change and defect data

Defects by classification (severity, subsystem, time of insertion, source of error, resolution)
 Problem-report status
 Defect detection method (review, inspection, test, etc.)
 Effort to detect and correct each defect

Process data

Process definition (design method, programming language, review method, etc.)
 Process conformance (is code reviewed when it's supposed to be, etc.)
 Estimated time to complete
 Milestone progress -
 Code growth over time
 Code changes over: time
 Requirements changes over time

Product data

Development dates
 Total effort

(continued)

Table 26-1. Examples of Kinds of Measurement Data, *continued****Product data, continued***

Kind of project (business, shrink-wrap, systems, etc.)
 Functions or objects included in project
 Size in lines of code and function points
 Size of documents produced
 Programming language

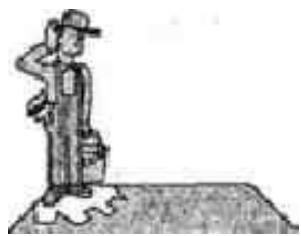
Once you've started collecting even a few data elements, you can gain insight from the raw data, and you can also combine data elements to gain other insights. Here are some examples of combined data elements you can create:

- Number of open defects vs. total defects reported (to help predict project release dates)
- Number of defects found by inspection vs. by execution testing (to help plan quality-assurance activities)
- History of estimated vs.. actual days remaining in a project, as a percentage (to help track and improve estimation accuracy)
- Average lines of code per staff month, broken down by programming language (to help estimate and plan programming activity)
- Average function points per staff month, broken down by programming language (to help estimate and plan programming activity)
- Percentage of total defects removed before product release (to help assess product quality)
- Average time to fix a defect, by severity, by subsystem, and by time of defect insertion (to help plan defect-correction activity)
- Average hours per page of documentation (to help plan documentation activity on a project)

Most projects don't collect the raw data that would allow them to create this information, but as you can see, this information requires the collection of only a few measurements.

Granularity

One of the problems with the data that most organizations collect is that it's collected at too large a granularity to be useful. For example, an organization might accumulate data "on the total number of hours spent on project FooBar but not distinguish between how much time was spent on specification, prototyping, architecture, design, implementation, and so on. Such large-grain data might be useful to accountants, but it's not useful to someone who wants to analyze planning and estimation data or data that will be used to improve the software-development process.



CLASSIC MISTAKE

Table 26-2 lists a set of categories and activities for time accounting that will provide as much granularity, as most projects need to start with.

**FURTHER READING**

For a different list of activities that you can use as a basis for time accounting, see Table 1.1 in *Applied Software Measurement* (Jones 1991).

Table 26-2. Example of Time-Accounting Activities

Category	Activity
Management	Plan Manage customer or end-user relations Manage developers Manage quality assurance Manage change Downtime
Administration	Development lab setup
Process development	Create development process Review or inspect development process Rework/fix development process Educate customer or team members about development process
Requirements specification	Create specification Review or inspect specification Rework/fix specification Report defects detected during specification
Prototyping	Create prototype Review or inspect prototype Rework/fix prototype Report defects detected during prototyping
Architecture	Create architecture Review or inspect architecture Rework/fix architecture Report defects detected during architecture
Design	Create design Review or inspect design Rework/fix design Report defects detected during design
Implementation	Create implementation Review or inspect implementation Rework/fix implementation Report defects detected during implementation
Component acquisition	Investigate or acquire components, either commercial or in-house Manage component acquisition

(continued)

Table 26-2. Example of Time-Accounting Activities, *continued*

Category	Activity
Component acquisition, continued	Test acquired components Maintain acquired components Report defects in acquired components
Integration	Create and maintain build Test build Distribute build
System testing	Plan system testing Create manual for system testing Create automated system test Run manual system test Run automated system test Report defects detected during system test
Product release	Prepare and support intermediate release Prepare and support alpha release Prepare and support beta release Prepare and support final release
Metrics	Collect measurement data Analyze measurement data

Using the Data You Collect

Collecting the data doesn't do much good if you don't use it. The following sections explain what to do with the data you collect.

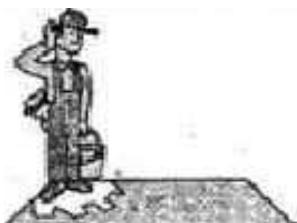
CROSS-REFERENCE
For details on how average projects spend their time, see Section 6.5, "Where the Time Goes."

Pareto analysis

If you're concerned about development speed, one of the most powerful things you can do with the data you collect is to perform a Pareto analysis—look for the 20 percent of activities that consume 80 percent of the time. Optimizing a software project for speed is similar to optimizing a software program for speed. Measure where you spend your time, and then look for ways to make the most time-consuming areas more efficient.

Analysis vs. measurement

On the opposite end of the scale from organizations that collect data that's too coarse are organizations that are so excited about software measurement that they collect data about everything. Collecting too much data is actually not much better than collecting too little. It's easy to bury yourself in data that's so unreliable you can't know what any of it means. To be meaningful,



CLASSIC MISTAKE

ful, metrics need to be defined and standardized so that you can compare them across projects.



NASA's Software Engineering Laboratory (SEL) has had an active measurement program for almost 20 years. One of the lessons SEL has learned is to spend more effort on analysis and less on data collection. In the early years of the program, SEL spent about twice as much on data collection as on analysis. Since then, data collection effort has dropped by half, and SEL now spends about three times as much on analysis and packaging as on collection (NASA 1994).

After the data is analyzed, SEL has found that packaging the lessons learned is key to making the data collection and analysis useful. You might package data and analysis in any of the following forms:

- Equations, such as the amount of effort or computing resources typically needed to develop a program of a particular size
- Pie charts, such as distributions of errors by severity
- Graphs defining ranges of normal, such as growth in lines of code checked into version control over time
- Tables, such as expected time-to-ship for various numbers of open defects
- Defined processes, such as code-inspection or code-reading process
- Rules of thumb, such as, "Code-reading finds more interface defects than execution testing"
- Software tools that embody any or all of the above

If you're just beginning a measurement program, use a simple set of measurements, such as number of defects, effort, dollars, and lines of code. (The time-accounting data described in Table 26-2 would make up a single "effort" measurement.) Standardize the measurements across your projects, and then refine them and add to them as you gain experience (Pietrasanta 1990, Rifkin and Cox 1991).



Organizations that are just starting to collect software data tend on average to collect about a dozen different measurements. Even organizations that have a great deal of experience with measurement programs still tend to collect only about two-dozen measurements (Brodman and Johnson 1995).

Feedback



Once you set up a measurement program, it's important to update developers and managers on the results of the measurements. Providing feedback to developers can be an organizational blind spot; organizations tend to feed the data back to managers but overlook developers. In one survey, 37 percent of managers thought that feedback was given on measurement data, but only 11 percent of developers thought so (Hall and Fenton 1994).

Developers tend to be leery of how measurement data will be used. Both managers and developers think that managers manipulate measurement data at least one-third of the time (Hall and Fenton 1994). When developers can't see what's being done with the data, they lose confidence in the measurement program. A poorly implemented metrics program can actually damage developer morale.



When an organization does provide feedback on measurement data, developers are enthusiastic about the measurement program. With feedback, they say that the measurement program is "quite useful" or "very useful" about 90 percent of the time. When an organization doesn't provide feedback, they say the measurement program is "quite useful" or "very useful" only about 60 percent of the time (Hall and Fenton 1994).

Another way to increase developer enthusiasm is to ask developers to participate in the design of the data-collection forms. If you do, the data you collect will be better, and your invitation "will improve the likelihood of their buy-in (Basili and McGarry 1995).

Baseline report

One specialized kind of feedback that measurement organizations provide is an annual software-baseline report. The baseline report is similar to an annual financial report, but it describes the state of the organization's software-development capability. It includes summaries of the projects conducted that year; strengths and weaknesses in the areas of people, process, product, and technology; staffing levels; schedules; productivity levels; and quality levels. It describes non-software personnel's perceptions of the software-development organization and the development organization's perceptions of itself. It also includes a description of the organization's existing software inventory.

The baseline is built on the basis of historical data, surveys, roundtable discussions, and interviews. It isn't evaluative; it doesn't tell you whether your software-development capability is good or bad. It's purely descriptive. As such, it provides a critical foundation for comparing your status year-to-year and for future improvements.

Limitations

**Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?**

T.S. Eliot

Measurement is useful, but it is not a panacea. Keep these limitations in mind.

Overreliance on statistics. One of the mistakes that NASA's Software Engineering Laboratory (SEL) made initially was that it assumed it would gain the most insight through statistical analysis of the data it collected. As SEL's measurement program matured, SEL discovered that it did get some insight from statistics but that it got more from talking about the statistics with the people involved (Basili and McGarry 1995).



HARD DATA

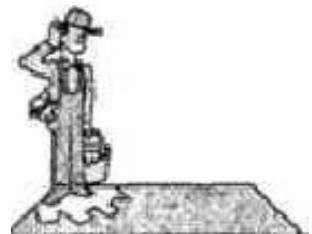
Data accuracy. The fact that you measure something doesn't mean the measurement is accurate. Measurements of the software process can contain a lot of error. Sources of errors include unpaid and unrecorded overtime, charging time to the wrong project, unrecorded user effort, unrecorded management effort, unrecorded specialist effort on projects, unreported defects, unrecorded effort spent prior to activating the project-tracking system, and inclusion of non-project tasks. Capers Jones reports that most corporate tracking systems tend to omit 30 to 70 percent of the real effort on a software project (Jones 1991). Keep these sources of error in mind as you design your measurement program.

26.2 Managing the Risks of Measurement

In general, Measurement is an effective risk-reduction practice. The more you measure, the fewer places there are for risks to hide. Measurement, however, has risks of its own. Here are a few specific problems to watch for.

Over-optimization of single-factor measurements. What you measure gets optimized, and that means you need to be careful when you define what to measure. If you measure only lines of code produced, some developers will alter their coding style to be more verbose. Some will completely forget about code quality and focus only on quantity. If you measure only defects, you might find that development speed drops through the floor.

It's risky to try to use too many measurements when you're setting up a new measurement program, but it's also risky not to measure enough of the project's key characteristics. Be sure to set up enough different measurements that the team doesn't overoptimize for just one.



CLASSIC MISTAKE

Measurements misused for employee evaluations. Measurement can be a loaded subject. Many people have had bad experiences with measurement in SAT scores, school grades, work performance evaluations, and so on. A tempting mistake to make with a software-measurement program is to use it to evaluate specific people. A successful measurement program depends on the buy-in of the people whose work is being measured, and it's important that a measurement program track projects, not specific people.

Perry, Staudenmayer, and Votta set up a software research project that illustrated exemplary use of measurement data. They entered all data under an ID code known only to them. They gave each person being measured a "bill of rights," including the right to temporarily discontinue being measured at any time, to withdraw from the measurement program entirely, to examine the measurement data, and to ask the measurement group not to record something. They reported that not one of their research subjects exercised these rights, but it made their subjects more comfortable knowing they were there (Perry, Staudenmayer; and Votta 1994).

**FURTHER READING**

For an excellent discussion of problems with lines-of-code measurements, see *Programming Productivity* (Jones 1986a).

Misleading information from lines-of-code measurements. Most measurement programs will measure code size in lines of code, and there are some anomalies with that measurement. Here are some of them:

- Productivity measurements based on lines of code can make high-level languages look less productive than they are. High-level languages implement more functionality per line of code than low-level languages. A developer might write fewer lines of code per month in a high-level language and still accomplish far more than would be possible with more lines of code in a low-level language.
- Quality measurements based on lines of code can make high-level languages look as if they promote lower quality than they do. Suppose you have two equivalent applications with the same number of defects, one written in a high-level language and one in a low-level language. To the end-user, the applications will appear to have exactly the same quality levels. But the one written in the low-level language will have fewer defects per line of code simply because the lower-level language requires more code to implement the same functionality. The fact that one application has fewer defects per line of code creates a misleading impression about the applications' quality levels.

To avoid such problems, beware of anomalies in comparing metrics across different programming languages. Smarter, quicker ways of doing things may result in less code. Also consider using function points for some measurements. They provide a universal language that is better suited for some kinds of productivity and quality measurements.

26.3 Side Effects of Measurement

The main side effect of a measurement program is that what you measure gets optimized. Depending on what you measure, you might end up optimizing defect rates, usability, execution efficiency, schedule, or some other factor.

26.4 Measurement's Interactions with Other Practices

A measurement program provides the foundation for improvement in areas including estimation (Chapter 8), scheduling (Chapter 9), and productivity-tool evaluation (Chapter 15). Although it is possible to design a measurement program so that it undercuts a rapid-development project, there is no reason that a well-designed measurement program should interact negatively with any other practice.

26.5 The Bottom Line on Measurement



HARD DATA

Measurement programs naturally have some of the best data available to support their efficacy. Metrics guru Capers Jones reports that organizations that have established full software-measurement programs have often improved quality by about 40 percent per year and productivity by about 15 percent per year for 4 to 5 years consecutively (Jones 1991, 1994). He points out that only a handful of U.S. organizations currently have accurate measures of software defect rates and defect removal and that those organizations tend to dominate their industries (Jones 1991). The cost for this level of improvement is typically from 4 to 5 percent of the total software budget.

26.6 Keys to Success in Using Measurement

Here are the keys to success in using Measurement:

- Set up a measurement group. Put it in charge of identifying useful measurements and helping projects to measure themselves.
- Track time-accounting data at a fine level of granularity.
- Start with a small set of measurements. Select what you want to measure by using the Goals, Questions, Metrics approach.
- Don't just collect the data. Analyze it and provide feedback about it to the people whose work it describes.

Further Reading

Software Measurement Guidebook. Document number SEL-94-002. Greenbelt, Md.: Goddard Space Flight Center, NASA, 1994. This is an excellent introductory book that describes the basics of how and why to establish a measurement program. Among other highlights, it includes a chapter of experience-based guidelines, lots of sample data from NASA projects, and an extensive set of sample data-collection forms. You can obtain a single copy for free by writing to Software Engineering Branch, Code 552, Goddard Space Flight Center, Greenbelt, Maryland 20771.

Grady, Robert B., and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, N.J.: Prentice Hall, 1987. Grady and Caswell describe their experiences in establishing a software-metrics program at Hewlett-Packard and how to establish one in your organization.

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, N.J.: PTR Prentice Hall, 1992. This book is the follow-on to Grady and Caswell's earlier book and extends the discussion of lessons learned at Hewlett-Packard. It contains a particularly nice presentation of a set of software business-management graphs, each of which is annotated with the goals and questions that the graph was developed in response to.

Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality*. New York: McGraw-Hill, 1991. This book contains Jones's recommendations for setting up an organization-wide measurement program. It is a good source of information on functional metrics (the alternative to lines-of-code metrics). It describes problems of measuring software, various approaches to measurement, and the mechanics of building a measurement baseline. It also contains excellent general discussions of the factors that contribute to quality and productivity.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Menlo Park, Calif.: Benjamin/Cummings, 1986. This book catalogs software-measurement knowledge, including commonly used measurements, experimental techniques, and criteria for evaluating experimental results. It is a useful, complementary reference to either of Grady's books or to Jones's book.

IEEE Software, July 1994. This issue focuses on measurement-based process improvement. The issue contains articles that discuss the various process-rating scales and industrial experience reports in measurement-based process improvement.

Miniature Milestones



S
U
M
M
A
R
Y

The Miniature Milestones practice is a fine-grain approach to project tracking and control that provides exceptional visibility into a project's status. It produces its rapid-development benefit by virtually eliminating the risk of uncontrolled, undetected schedule slippage. It can be used on business, shrink-wrap, and systems software projects, and it can be used throughout the development cycle. Keys to success include overcoming resistance of the people whose work will be managed with the practice and staying true to the practice's "miniature" nature.

Efficacy

Potential reduction from nominal schedule:	Fair
Improvement in progress visibility:	Very Good
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Good
Chance of long-term success:	Excellent

Major Risks

None

Major Interactions and Trade-Offs

- Especially well-suited to project recovery
- Especially effective when combined with the Daily Build and Smoke Test practice
- Works well with Evolutionary Prototyping, User-Interface Prototyping, Requirements Specification, and other hard-to-manage project activities
- Trades increase in project-tracking effort for much greater status visibility and control

Imagine that you're a pioneer heading from the east coast to the west. Your journey is much too long to be completed in a single day, so you define a set of points that will mark the significant milestones on your journey. It's a 2500 mile journey, so you mark five milestones, each about 500 miles apart.

Major milestones 500 miles apart are great for setting long-term direction, but they are lousy for figuring out where to go each day—especially when you're traveling only, say, 25 miles per day. For that, you need finer-grain control. If you know that your big milestone is 500 miles away, north-by-northwest, you can take a compass reading, find a closer landmark that's roughly north-by-northwest, and then strike out toward that. Once you reach that closer landmark, you take another compass reading, find another landmark, and strike out *again*.

The close landmarks that you pick—the tree, rock formation, river, or hill-top—serve as your miniature milestones. Reaching the miniature milestones provides you with a steady sense of accomplishment. Since you pick only milestones that are between you and your next big milestone, reaching the miniature milestone also gives you confidence that you will eventually reach your larger objective.

Miniature Milestones' support for rapid development boils down to four factors: improved status visibility, fine-grain control, improved motivation, and reduced schedule risk.

Improved status visibility. One of the most common problems on software-development projects is that neither developers, project leaders, managers, nor customers are able to assess the project's status accurately. Say nothing about whether they can predict when the project will be done, they don't even know how much they've already completed!



' CLASSIC MISTAKE

Jim McCarthy cautions against letting a developer "go dark" (McCarthy 1995a). You believe that everything's going along OK. Why? Because every day you ask your developers, "How's it going?" They say, "Fine." And then one day you ask, "How's it going?" And they say, "Urn, we're going to be about 6 months late." Wow! They slipped 6 months in 1 day! How did that happen? It happened because they were "working in the dark"—neither you nor they had enough light on their work to know that they had been slipping all along.

With Miniature Milestones, you define a set of targets that you have to meet on a near-daily basis. If you start missing milestones, your schedule isn't

**How does a project
get to be a year late?
One day at a time.**

Frederick P. Brooks, Jr.

realistic. Since your milestones are fine-grained, you will find out early that you have a problem. That gives you an early opportunity to recalibrate your schedule, adjust your plan, and move on. .

Fine-grain control. In *Roger's Version*, John Updike describes a diet plan in which a woman weighs herself every Monday morning. She is a small woman, and she wants to weigh less than 100 pounds. If on Monday morning she finds that she weighs more than 100 pounds, she eats only carrots and celery until she again weighs less than 100 pounds. She reasons that she can't gain more than 1 or 2 pounds in a week, and if she doesn't gain more than 1 or 2 pounds, she certainly won't gain 10 or 20. With her approach, her weight will always stay close to where she wants it to be.

The Miniature Milestone practice applies this same idea to software development, and it's based on the idea that if your project never gets behind schedule by more than a day or so, it is logically impossible for it to get behind schedule by a week or a month or more.

Milestones also help to keep people on track. Without short-term milestones, it is too easy to lose sight of the big picture. People spend time on detours that seem interesting or productive in some vague sense but that fail to move the project forward. With larger-grain tracking, developers get off schedule by a few days or a week, and they stop paying attention to it.

With Miniature Milestones, everyone has to meet their targets every day or two. If you meet most of your milestones just by working a full day—and meet the rest by working an extra full day—you will meet the overall, big milestones as well as the little ones. There's no opportunity for error to creep in.

CROSS-REFERENCE
For more on developer motivations, see Chapter 11, "Motivation."

CROSS-REFERENCE
For more on detailed estimation, see "Estimate at a low level of detail" in Section 8.3.

Improved motivation. Achievement is the strongest motivator for software developers, and anything that supports achievement or makes progress more palpable will improve motivation. Miniature Milestones make progress exceptionally tangible.

Reduced schedule risk. One of the best ways to reduce schedule risk is to break large, poorly defined tasks into smaller ones. When creating an estimate, developers and managers tend to concentrate on the tasks they understand best and to shrug off the tasks they understand least. The frequent result is that a 1-week "DBMS interface" job can turn out to take an unexpected 6 weeks because no one ever looked at the job carefully. Miniature Milestones address the risk by eliminating large schedule blobs entirely.

Major Milestones

Major milestones are the far-apart milestones that help you set general directions for your projects. Such milestones are typically months apart. On traditional, waterfall projects, the major milestones typically consist of product definition complete, requirements specification complete, architecture complete, detailed design complete, code complete, system testing complete, and product acceptance.

On modern shrink-wrap projects, a typical set of major milestones might consist of project approval, visual freeze, feature complete, code complete, and release to manufacturing.

Major milestones are useful for setting direction, but they are too far apart to provide good project control. The combination of major and miniature milestones can work well, with major milestones providing the project's strategic objectives and miniature milestones providing the tactical means to achieve each objective.

27.1 Using Miniature Milestones

You can apply the Miniature Milestones practice throughout the life of a project. You can apply it to early activities such as Requirements Specification and Evolutionary Prototyping; in fact, it is particularly useful in focusing those hard-to-direct activities.

For maximum benefit, the Miniature Milestones practice will be implemented at the project level by the technical lead or manager, whichever is appropriate. But individual contributors can implement it on a personal level even if their leaders don't.

The amount of detail required when implementing Miniature Milestones will give pause to whoever has responsibility for tracking those details, especially on large projects. But large projects are the projects that most commonly spin out of control, and it is on those projects that this kind of detailed tracking is especially needed.

CROSS-REFERENCE
For more on initiating new measures in response to a crisis, see "Timing" in Section 16.2.

Initiate Miniature Milestones early or in response to a crisis. Miniature Milestones provide a high degree of project control. Set them up early in the project or in response to an acknowledged crisis. If you set them up at other times, you are at the risk of seeming Draconian. As with other aspects of project control, it's easier to overcontrol in the beginning and relax control as the project progresses than it is the other way around. As Barry Boehm and Rony Ross say, "Hard-soft works better than soft-hard" (Boehm and Ross 1989).

Have developers create their own mini milestones. Some developers will view Miniature Milestones as micro-management, and, actually, they'll be right. It *is* micro-management. More specifically, it's micro project-tracking. However, not all micro-management is bad. The micro-management that developers resist is micro-management of the details of how they do their jobs.

If you let people define their own miniature milestones, you allow them to control the details of their jobs. All you're asking is that they tell you what the details are, which improves buy-in and avoids seeming like micro-management. Some people don't understand the details of their jobs, and those people will feel threatened by this practice. If you handle their objections diplomatically, learning to work to a miniature-milestone schedule will serve as an educational experience for them.

Keep milestones miniature. Make mini milestones that are achievable in 1 or 2 days. There's nothing magical about this size limit, but it's important that anyone who misses a milestone can catch up quickly. If people have done generally good jobs of estimating their work, they should be able to catch up on any particular missed milestone by working overtime for 1 or 2 days.

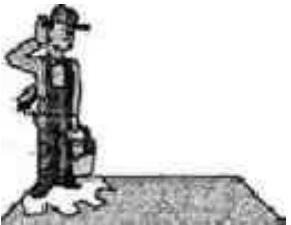
Another reason to keep milestones small is to reduce the number of places that unforeseen work can hide. Developers tend to view a week or weekend as an infinite amount of time—they can accomplish anything. They don't think about exactly what's involved in creating the "data conversion module," and that's why the job takes 2 weeks instead of the estimated one weekend. But most developers won't commit to tackling a problem in 1 or 2 days unless they understand what it involves.

To be sure you're basing your schedule on meaningful estimates, insist on further decomposing tasks that are above the "infinite amount of time" threshold for your environment.

Make milestones binary. Define milestones so that they are either done or not. The only two statuses are "done" and "not done." Percentages are not used. As soon as people are allowed to report that they are "90 percent done," the milestones lose their ability to contribute to a clear view of project progress.

Some people can't resist the temptation to fudge their status reporting with Miniature Milestones. "Are you done?" you ask. "Sure!" they say. "Are you 100 percent done?" you ask. "Well, uh, I'm 99 percent done!" they say. "What do you mean, '99 percent done?'" you ask. And they say, "Uh, I mean that I still need to compile and test and debug the module, but I've got it written!"

Be fanatic about interpreting milestones strictly.



CLASSIC MISTAKE

CROSS-REFERENCE
For another example of this, see "Track schedule progress meticulously" in Section 16.2.

- Do at least as good a job of specifying requirements as you would do for an in-house project (unless requirements specification is one of the vendor's strengths).
- Be sure that Outsourcing of the rapid-development project is in your organization's long-term best interest.

Further Reading

Marciniak, JohnJ., and DonaldJ. Reifer. *Software Acquisition Management*. New York: John Wiley & Sons, 1990. Marciniak and Reifer fully explore the considerations on both the buying and selling sides of outsourced software relationships. The book has a strong engineering bent, and it discusses how to put work out for bid, write contracts, and manage outsourced projects from start to completion.

Humphrey, W. S., and W. L. Sweet. *A Method for Assessing the Software Engineering Capability of Contractors*. SEI Technical Report CMU/SEI-87-TR-23, Pittsburgh: Software Engineering Institute, 1987. This report contains more than 100 detailed questions that you can use to assess a vendor's software-development capability. The questions are divided into categories of organizational structure; resources, personnel, and training; technology management; documented standards and procedures; process metrics; data management and analysis; process control; and tools and technology. Vendor evaluation has been a major emphasis of work at the Software Engineering Institute, and this report also describes an overarching framework that you can use to evaluate the vendor's general development capabilities.

Humphrey, Watts S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989- Chapter 19 is devoted to contracting for software. It contains insightful guidelines for establishing a relationship with a vendor, developing a set of requirements, tracking progress, monitoring quality, and managing vendors at different competency levels.

Dedene, Guido, and Jean-Pierre De Vreese. "Realities of Off-Shore Reengineering," *IEEE Software*, January 1995, 35-45. This is an interesting case study in outsourcing two software projects overseas.

Principled Negotiation



Principled Negotiation is a negotiating strategy that relies on improving communications and the creation of win-win options rather than on negotiating tricks. It can be used during requirements analysis, schedule creation, feature-change discussions, and at other times throughout a project. It produces its rapid-development benefit by clarifying expectations and identifying exactly what is needed to set the project up for success. Effective use of Principled Negotiation depends on separating people from the problem; focusing on interests, not positions; inventing options for mutual gain; and insisting on the use of objective criteria. It can be used on any kind of project.

S
U
M
M
A
R
N

Efficacy

Potential reduction from nominal schedule:	None
Improvement in progress visibility:	Very Good
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Very Good
Chance of long-term success:	Excellent

Major Risks

None

Major Interactions and Trade-Offs

None

For more on principled negotiation, see Section 9.2, 'Beating Schedule Pressure.'

Productivity Environments



Software development is a highly intellectual activity that requires long periods of uninterrupted concentration. Productivity Environments provide developers with the freedom from noise and interruptions they need in order to work effectively. The use of Productivity Environments can benefit any kind of project—business, shrink-wrap, or systems. In addition to productivity improvements, some organizations have experienced improvements in developer morale and retention rates after establishing Productivity Environments.

Efficacy

Potential reduction from nominal schedule:	Good
Improvement in progress visibility:	None
Effect on schedule risk:	No Effect
Chance of first-time success:	Good
Chance of long-term success:	Very Good

Major Risks

- Lost productivity from status-oriented office improvements
- Transition downtime
- Political repercussions of preferential treatment for software professionals

Major interactions and Trade-Offs

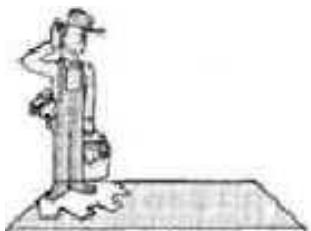
- Trades small increase in cost for large increase in productivity

S
U
M
M
A
R
Y

If you were in the oil business, before you could make a single dollar you would need to locate a source of oil, drill a hole in the ground, pump the oil out of the ground, refine the oil, pump it into ships or trucks or barrels, and sell it. Part of the profitability of your business would be determined by how efficiently you could pump the oil out of the ground. If your pumping technique left 50 percent of the oil underground, you would be leaving 50 percent of your potential revenues underground, too.

You can't pump software out of a hole in the ground. Our nation's software reserves are located predominately inside the brains of our nation's software developers, and extracting that software from those brains requires every bit as much finesse as extracting oil from an oil well.

Paradoxically, the majority of developers today work under conditions that almost seem designed to prevent the extraction of working software from their brains. More than 70 percent of all software organizations have crowded office conditions, and the average time between interruptions under those conditions is only about 11 minutes (Jones 1994).



CLASSIC MISTAKE

Unlike most management tasks, which are interrupt based and which survive and thrive on frequent interruptions, software development tasks require long periods of uninterrupted concentration. Because managers generally do not need long uninterrupted periods to do their work, developers' requests for peace and quiet can seem like requests for preferential treatment. But developers are usually highly self-motivated, and what they're really asking for is to be provided with conditions that will allow them to work efficiently.

Flow time. During the analysis and design stages, software development is an ephemeral, conceptual activity. Like any conceptual activity, the quality of the 'work' is dependent on the worker's ability to sustain a "flow state"—a relaxed state of total immersion in a problem that facilitates understanding of it and the generation of solutions for it (DeMarco and Lister 1987). Converting brain waves to computer software is a delicate process, and developers work best during the hours they spend in this state of effortless concentration. Developers require 15 minutes'or more to enter a state of flow, which can then last many hours, until fatigue or interruption terminates it. If developers are interrupted every 11 minutes, they will likely never enter a flow state and will therefore be unlikely to ever reach their highest levels of productivity.

CROSS-REFERENCE

For more on hygiene needs, see "Hygiene Factors" in Section 11.4.

Hygiene needs. In addition to the enhanced ability to enter a flow state, the Productivity Environments practice addresses a major motivational factor for software developers. Office space appropriate for development work is a

"hygiene" motivational factor: that is, adequate office space does not increase motivation or productivity, but below a certain threshold inadequate office space can seriously erode motivation and productivity.

For developers, the need for productivity environments is obvious and basic. Productivity environments are in the same motivational category as adequate lighting, reliable power, and accessible bathroom facilities. An organization that doesn't provide an environment in which developers can work effectively is not providing the basics that developers need to be productive, and developers tend to view such organizations as irrationally working against their own interests. Good developers tend to gravitate toward organizations that provide work environments in which they can be productive. Developers who stay in less productive environments tend to lose their motivation and morale, and productivity suffers. (See Figure 30-1.)

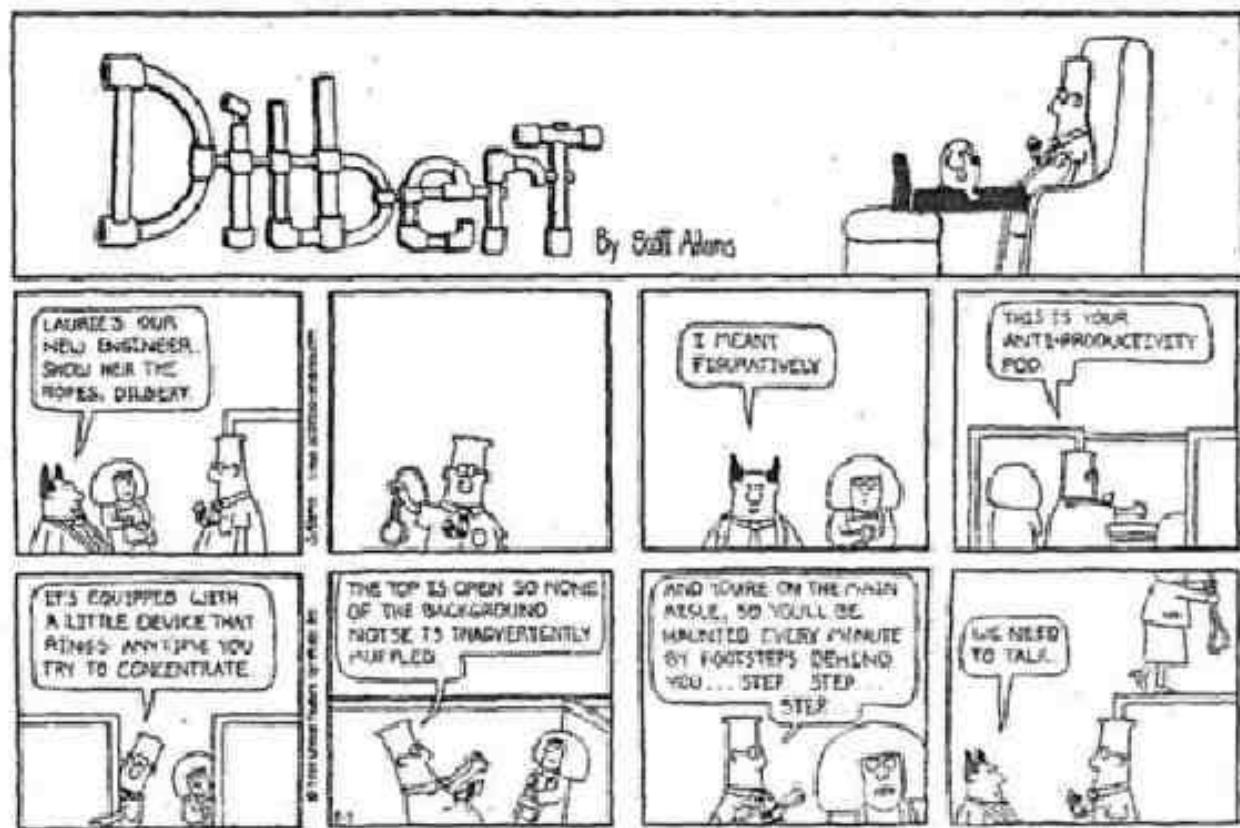


Figure 30-1. *Djlbert* reprinted by permission of United Feature Syndicate, Inc.

Developers, team leads, and lower-level managers don't usually have the latitude to move a team to more productive offices. But if your project is under schedule pressure and your management is serious about productivity improvements, you might try negotiating for quieter, more private office space in return for promises of higher productivity.

31.3 Side Effects of RDLs

Specific RDLs can have influences on quality, usability, functionality, and other product characteristics, and you should consider those factors when you evaluate specific RDL products.

31.4 RDLs' Interactions with Other Practices

Nearly all of the general guidelines for using productivity tools (Chapter 15) apply to RDLs.

RDLs provide schedule-reduction leverage because of their ability to reduce construction time. Because they reduce construction time so much, they also make new kinds of lifecycle models possible. If switching from a 3GL to an RDL cuts detailed design and coding effort by 75 percent (which is a good rule of thumb), that will significantly reduce the amount of time between iterations in an iterative lifecycle model. An iteration that takes 2 months in C might be cut to as little as 2 weeks in Visual Basic. The difference in those two time periods is the difference between customers or end-users seeing you as responsive and unresponsive. To derive maximum benefit from an RDL, employ it within an incremental, iterative lifecycle model (Chapter 7).

If you're working on shrink-wrap, real-time, or other software that is not well-suited to implementation in an RDL, you can still use RDLs for User-Interface Prototyping (Chapter 42) and Throwaway Prototyping (Chapter 38). One of the goals of prototyping is to use the minimum effort possible, and RDLs can contribute to that goal.

31.5 The Bottom Line on RDLs



As suggested by Tables 31-1 and 31-2, the bottom line on RDLs is that the specific savings you can expect depends both on the specific language you're using now and on the specific RDL you switch to. It also depends on whether the kind of program you need to build is the kind the RDL is good at building. If you're currently using a 3GL, you can probably expect to cut construction effort by about 75 percent when you switch to an RDL. That amount will continue to improve as new and improved languages become available—but it probably won't improve as quickly as tool vendors will want you to believe! :

If you can't switch completely to an RDL, you might still be able to implement some of your project in an RDL. The 75 percent rule of thumb applies: you can expect to cut your design and construction effort by about 75 percent for the part of the code you implement in the RDL (Klepper and Bock 1995).

**FURTHER READING**

For more on the effects of project size on project activities, see Chapter 21, "Project Size," in *Code Complete* (McDonnell 1993).

As project size increases, the savings you realize from switching to an RDL decreases. RDLs yield their savings by shortening the construction part of a project. On small projects, construction activities can make up as much as 80 percent of the project's total effort. On larger projects, however, detailed design, coding, and debugging shrink to something like 25 percent of the total effort, and the savings from streamlining those activities will shrink accordingly (McCohnell 1993).

31.6 Keys to Success in Using RDLs

Here are the keys to success in using RDLs:

- All other things being equal, for maximum development speed use the language with the highest language level in Table 31-2 (keeping in mind the limitations of that table's data).
- Select specific RDLs using the selection criteria listed in Section 15.3, "Productivity-Tool Acquisition."
- Put specific RDLs into use using the guidelines described in Section 15.4, "Productivity-Tool Use."
- Estimate the savings you expect to realize from RDL usage conservatively. Consider your project's size and the part of the lifecycle you expect to compress by using the RDL. On all but the smallest and simplest projects, schedule in time for working around the limitations of the RDL.
- Be careful about using RDLs on large projects. Bear in mind that as project size increases, the limitations of RDLs become more severe and the savings potential decreases.
- Err on the side of overdesign and overly careful coding standards when using an RDL.
- When you switch to an RDL, look for opportunities to use new lifecycle models that allow you to be increasingly responsive to your customers.

Further Reading

Jones, Capers. "Software Productivity Research Programming Languages Table," 7th Edition, March 1995, Burlington, Mass.: Software Productivity Research, 1995. This table provides language levels and statements per function point for several hundred languages. You can access the full table on the Internet at <http://www.spr.com/library/langtbl.htm>.

McConnell, Steve. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993. Much of the book describes how to work around programming-language limitations, advice which applies to RDLs as well as any other language. Chapter 21 describes the effect that program size has on project activities and therefore on the potential that an RDL has to reduce overall development time.



Requirements Scrubbing

Requirements Scrubbing is a practice in which a product specification is carefully examined for unnecessary or overly complex requirements, which are then removed. Since product size is the largest contributor to a project's cost and duration, reducing the size of the product produces a commensurately less expensive project and shorter schedule. Requirements Scrubbing can be used on virtually any project.

S
U
M
M
A
R
Y

Efficacy

Potential reduction from nominal schedule:	Very Good
Improvement in progress visibility:	None
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Very Good
Chance of long-term success:	Excellent

Major Risks

- Elimination of requirements that are later reinstated

Major Interactions and Trade-Offs

None

For more on requirements scrubbing, see "Requirements Scrubbing" in Section 14.1.

Reuse



Reuse is a long-term strategy in which an organization builds a library of frequently used components, allowing new programs to be assembled quickly from existing components. When backed by long-term management commitment, Reuse can produce greater schedule and effort savings than any other rapid-development practice. What's more, it can be used by virtually any kind of organization for any kind of software. Reuse can also be implemented opportunistically, as a short-term practice, by salvaging code for a new program from existing programs. The short-term approach can also produce significant schedule and effort savings, but the savings potential is far less dramatic than with Planned Reuse.

Efficacy

Potential reduction from nominal schedule:	Excellent
Improvement in progress visibility:	None
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Poor
Chance of long-term success:	Very Good

Major Risks

- Wasted effort if the components that are prepared for reuse are not selected carefully

Major Interactions and Trade-Offs

- Reuse needs to be coordinated with productivity-tool use.
- Planned Reuse must be built on a foundation of software-development fundamentals.

With the Signing Up practice, a manager or team leader asks potential team members to make an unconditional commitment to seeing that a project succeeds. The team is then allowed to complete the project in its own way. Signing Up derives its rapid-development benefit from its tremendous motivational ability. Developers who sign up make a voluntary, personal commitment to the project, and they often go to extraordinary lengths to make the project succeed. Teams that have signed up work at such a hectic pace that they are bound to make some mistakes, but the sheer amount of effort they put in can swamp the effects of those mistakes.

34.1 Using Signing Up

Kerr and Hunter point out that the Antarctic explorer Shackleton found his crew by advertising for men to perform hard work under dangerous conditions for low pay, *with the possibility of tremendous glory if successful* (Ken and Hunter 1994). That, in a nutshell, is how you use signing up. You offer developers little reward except those intrinsic in the work itself: the chance to work on something important, to stretch their capabilities, to surmount a seemingly impossible goal, and to achieve something that no one in the organization has achieved before.

In *The Soul of a New Machine*, Tracy Kidder describes a signed-up team in which the main benefit of signing up is something called "pinball" (Kidder 1981). In pinball, the only benefit of winning the game is that you get to play *again*. If you sign up for a development project and succeed, you get to sign up again, on the next exciting project. That's the reward, and that's all the reward that many developers need.

Some people don't like playing pinball, and some people don't like signing up. To them, it seems like an exercise in illogic and masochism. To some, it seems like management manipulation. But to others, it represents a long-awaited opportunity. IBM found that it had no problem getting people to sign up. They found that people wanted to commit to producing extraordinary results at work; all that was missing was the opportunity (Scherr 1989).

CROSS-REFERENCE
For details on motivation and teamwork, see Chapter 11, "Motivation," and Chapter 12, "Teamwork." For details on creating visions, see "Goal Setting" in Section 11.2 and "Shared, Elevating Vision or Goal" in Section 12.3.

Frame a challenge and a vision. The keys to success with Signing Up are similar to the keys to success in motivation in general and in building high-performance teams. At the top of all the lists is providing a clear vision of what the project is supposed to accomplish. To get people to sign up, the vision needs to be of an extraordinary accomplishment. Merely completing a project is not good enough. Here are some examples of extraordinary visions:

- Be the first group of explorers to reach the south pole

- Be *the first* country to put an astronaut on the moon
- Design and build a totally new computer without the company's support
- Design the best computer operating system in the world
- Be the first team in the organization to develop a complete shrink-wrap software product in less than a year
- Create a DBMS package that places number one in its first *InfoWorld* comparative test and beats all its competitors by at least 0,5 points
- Decisively leapfrog the competition in the same software category

Give people a choice about signing up. The Signing Up practice doesn't work if people don't have a choice about whether they sign up. You have to accept the possibility that some of the people you'd like to have on your team won't make the extraordinary commitment that Signing Up requires. The fact that some qualified people don't make the team works partly in your favor. The team members who do sign up can see that some people don't have what it takes to be on it, and that helps to foster their sense of team identity.

If you've already put the team together, you can't use Signing Up unless you're prepared to kick people off the team who won't sign up midway through the project. Signing Up needs to be implemented at the beginning of the project or in response to a crisis. It's not a practice you can initiate midstream.

Once developers have made their choice, however, the commitment must be unequivocal. They must commit to make the project succeed no matter what.

Sign up at the team level. The Signing Up practice seems to work best on teams that are small enough to have a team identity. It's hard for someone to sign up with a large organization. Some companies have used this practice successfully on large projects by creating small teams within the large projects and having people sign up for those.

People need to identify with a group that's small enough so that they know their contribution matters. When people are signed up, each and every one of them will feel personally responsible for completing the product. When the product is done, each person will feel that he or she was the key person that the project could not have survived without. And each of those people will be right.

Because Signing Up is best done at the team level, it doesn't have to be initiated by management. It can be initiated by the team lead or even by a de facto leader—someone who happens to be **exceptionally** self-motivated and wants to pull the rest of the team along.

Followthrough by empowering the team. Signing Up won't work unless you let the team run with the ball. Point the direction, but don't specify how to get to the end.

Don't use Signing Up on a project with shaky requirements. Your highly motivated team needs to be able to charge full-steam ahead, not full-steam right, then full-steam left, then full-steam backwards. The only exception to this rule is when the team itself is responsible for defining requirements. Then sometimes they can tolerate numerous changes in direction without losing motivation.

Signing Up in Different Environments

You can use Signing Up on virtually any kind of project—business, shrink-wrap, systems, and so on. Signing Up always requires an "extraordinary commitment," but the exact nature of that extraordinary commitment means different things in different environments.

In the world of RAD, James Kerr wrote about a dedicated RAD team that signed up, which meant that they were willing to sweep aside their normal mix of responsibilities and work a hard, focused 8 hours a day on one project (Kerr and Hunter 1994). Kerr reports that this team sometimes had to work at home in the evenings or for a few hours on the weekend. The high point of this project for Kerr was the night that the team of four people stayed late to implement a set of help screens. Kerr talks at length about what a grueling day it was. The team worked a full day, took a half-hour break for pizza and beer, and kept going almost until midnight. Kerr describes this as a "breakneck schedule."

In contrast, on the Microsoft Windows NT project, signing up meant foregoing *everything* to be able to work on the project: evenings, weekends, holidays, normal sleeping hours, you name it (Zachary 1994). When they weren't sleeping, they were working. Developers sacrificed hobbies, health, and families to work on the project. One team member answered email from the hospital while his wife was in labor. The NT development team wore beepers so that they could be called at 3:00 in the morning if their code had broken the build. People kept cots in their offices, and many would go several days without going home. Tracy Kidder describes a similar level of commitment in *The Soul of a New Machine* (Kidder 1981).

Some organizations, like Microsoft, don't mind if the signing-up commitment results in a lot of overtime.¹ Other organizations, like IBM, have found that part of the commitment can be not to work any overtime (Scherr 1989²). They've found that placing a set of severe, seemingly impossible constraints on a project forces the team to consider and implement radically productive solutions that' they would normally not even consider.

The level of commitment will vary with the degree of the challenge. The Windows NT team was faced with the challenge of developing the best operating system in the world, Kerr's RAD team was faced with the relatively mundane challenge of developing a business system for in-house use. Not surprisingly, one team was willing to sacrifice much more than the other.

34.2 Managing the Risks of Signing Up

The Signing Up practice is a double-edged sword. It offers tremendous motivational potential, but it also offers as many hazards as any other practice described in this book.

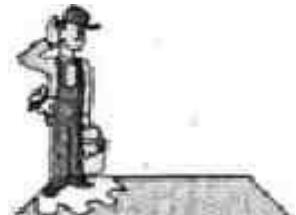
I think of that statistic that the best programmers are 25 times as productive as the worst programmers, and it seems that I am both of those guys.

Al Corwin

Increased inefficiency. Teams that are signed up have a tendency to work hard rather than to work smart. Although it's theoretically possible to work hard *and* work smart, most people seem to be able to do one or the other, but not both. A focus on working hard almost guarantees that they'll make mistakes that they'll live to regret, mistakes that will take more project time rather than less. Some experts have even argued that people who work more than 40 hours per week don't get any more done than people who work about 40 hours (Metzger 1981, Mills in Metzger 1981, DeMarco and Lister 1987, Brady and DeMarco 1994, Maguire 1994).

If you're working on a project in which people are signed up, watch for an increase in the number of time-consuming mistakes and other signs that people are not working as smart as they should.

Decreased status visibility. People who sign up make a personal commitment to deliver a product in the shortest possible time. In some cases, that can-do mentality makes it hard to assess the real status of the project.



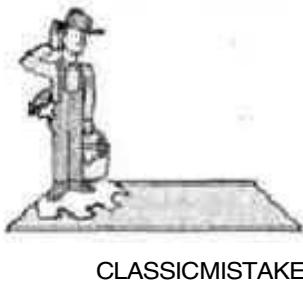
CLASSIC MISTAKE

CROSS-REFERENCE

All the problems discussed here are characteristic of this kind of scheduling. For details, see "Commitment-Based Scheduling" in Section B.5.

- | | |
|------------|---|
| Monday: | "Will you be done by Friday?" - 'You bet!' |
| Wednesday: | "Will you be done by Friday?" "You bet!" |
| Friday: | "Are you done?" "Um, no, but I will be really soon, I'll be done by Monday." |
| Monday: | "Are you done?" "Um, no. I should be done in a few hours." |
| Friday: | "Are you done?" "/ 'in getting really close. I'll be done any time now." |
| Monday: | "Are you done?" "No. I ran into some setbacks, but I'm on top of them, I should be done by Friday." |

Multiply this phenomenon across an entire project and you have a project whose status is virtually impossible to determine. Some organizations are willing to trade this kind of loss of management visibility for higher morale, and some aren't. Be aware of the trade-off your project is making.



Loss of control. The signed-up team takes on a life of its own, and the life it takes on is sometimes not the life that the company wants it to have. The team (and the product) might be headed in a different direction than management wants them to go. Forcing the team to change direction can give the team the impression that they aren't as empowered as they thought they were, and that can be fatal to morale.

Addressing this risk requires that you make a judgment about the trade-off between morale and efficiency as well as between morale and control. Are you getting enough of a morale boost from having a signed-up team to justify letting them go their own direction?

Smaller available talent pool for the project. Enthusiasm can work wonders, but it has its limits too. Some older, more-experienced developers who have signed up before simply won't sign up for Windows NT-like projects. The result can be an averaging down of the experience level on a project in which Signing Up is used. Such projects can be characterized more by their exceptional energy levels than by their exceptional results.

Burnout. Even when developers work overtime voluntarily, long hours can take a heavy toll. The anecdotal reports of developers who sign up for projects and work lots of overtime also include lengthy lists of developers who leave their companies at the ends of their projects (Kidder 1981; Zachary 1994).



34.3 Side Effects of Signing Up

Signing Up does not have any consistent, predictable effect on a product's characteristics. Side effects arise from the fact that the development team will have more control over the character of the product than it might otherwise, and that can mean that the product's quality, usability, functionality, or other attributes turn out better, worse, or simply different than you expect.

34.4 Signing Up's Interactions with Other Practices

Signing Up has a close relationship with teamwork (Chapter 12). People sign up for a team, and it's hard to have a team in which half the people are signed up and half are not. Usually, if the Signing Up practice is going to work, everyone on the team needs to be signed up.

You can expect teams that are signed up to work some Voluntary Overtime (Chapter 43). In exchange, the team will expect the organization to support

their efforts, at least in the form of providing them with a productive work environment (Chapter 30). The team might also resist more active, hands-on management practices such as Miniature Milestones (Chapter 27). You'll probably have to use commitment-based scheduling ("Commitment-Based Scheduling" in Section 8.5) and accept all of the trade-offs it involves.

34.5 The Bottom Line on Signing Up

The bottom line with Signing Up is that the degree of commitment you elicit will vary depending on the excitement that you're able to generate about the project. When you have an extraordinary project, you can see extraordinary commitment, extraordinary morale, and extraordinary productivity. On more mundane projects, you can expect less.

34.6 Keys to Success in Using Signing Up

Here are the keys to success in using Signing Up:

- Create a compelling vision for the project so that team members will have something that's worth signing up for.
- Make Signing Up voluntary.
- Empower the team so that it can succeed at the challenge it has been motivated to respond to.
- Be prepared to address or accept the inefficiencies that result from people working hard rather than smart.

Further Reading

Kidder, Tracy. *The Soul of a New Machine*. Boston: Atlantic Monthly-'Little Brown, 1981. This book describes how computer hardware developers signed up to work on Data General's Eagle computer. It lays out the signing-up process in detail and illustrates the motivational benefit of the process—Eagle's developers worked practically 24 hours a day throughout the project. It is also an object lesson in the risks associated with signing up: much of the development team quit the company when the project was finished.

version 2c and defer release to version 2d. But if you don't use Staged Delivery, you won't have the option.

CROSS-REFERENCE

For more on refining estimates based on experience, see "Recalibration" in Section 8.7.

Reduces estimation error. Staged Delivery sidesteps the problem of bad estimates by delivering early and often. Instead of making one large estimate for the whole project, you can make several smaller estimates for several smaller releases. With each release, you can learn from the mistakes in your estimates, recalibrate your approach, and improve the accuracy of future estimates.

CROSS-REFERENCE

For more on frequent integration, see Chapter 18, "Daily Build and Smoke Test."

Integration problems minimized. A common risk to software projects is difficulty in integrating components that were developed separately. The likelihood of serious integration problems is related to the time between successive integration attempts. If the time between attempts is long, the chance of problems is large. When you deliver software early and often, as you do with Staged Delivery, you must also perform integration early and often. That minimizes potential integration problems.

36.1 Using Staged Delivery

In Staged Delivery, you start with a clear idea of the product you will ultimately deliver. Staged Delivery is a late-in-the-project development practice. If you're following a traditional waterfall lifecycle model, you don't need to start planning for it until after you've completed requirements analysis and architectural design.

Once you've completed architectural design, to use Staged Delivery you plan a series of deliveries. As Figure 36-1 on page 550 suggests, within each stage you do a complete detailed design, construction, and test cycle, and at the end of each stage you deliver a working product. For example, if you were developing a word processing program, you might create the following delivery plan:

Table 36-1. Example of a Staged-Delivery Schedule for a Word Processor

Stage 1	Text editor is available, including editing, saving, and printing.
Stage 2	Character and basic paragraph formatting is available.
Stage 3	Advanced formatting is available, including WYSWYG page layout and on-screen formatting tools.
Stage 4	Utilities are available, including spell checking, thesaurus, grammar checking, hyphenation, and mail merge.
Stage 5	Full integration with other products is complete.

The first delivery should be the germ of the product you will ultimately deliver. Subsequent releases add more capabilities in a carefully planned way. You deliver the final product in the last stage.

Planning for the first release is unique in that it needs to include some global architectural thinking, which raises questions such as: "Is the software architecture open enough to support modifications, including many that we haven't fully anticipated?" It's also a good idea to plot a general direction for the software at the beginning of the project—although, depending on whether you intend to use a pure Staged Delivery approach or an Evolutionary Delivery approach, that general direction might just be a best guess that you'll override later.

You don't have to deliver each release to a customer to use the Staged Delivery practice, and you can implement it on a technical-lead level. In the case of the word processor, you might not even release a version to your customer until delivery 3 or 4 or even 5. But you could use Staged Delivery as an aid to track progress, coordinate drops to quality assurance, or reduce the risk of integration problems.

For the approach to work well, each stage should include size, performance, and quality targets in addition to functionality targets. Too much hidden work accumulates as you go along if you don't deliver a truly releasable product at the end of each stage.

As a general goal, try to deliver the software's capabilities from most important to least important. Defining the deliveries in this way forces people to prioritize and helps to eliminate gold-plating. If you do a good job of this, by the time you've delivered 80 percent of the product your customer will be wondering what could possibly be left in that last 20 percent.

Technical Dependencies

In a single large release, the order of component delivery doesn't matter much. Multiple small releases of a product require more planning than a single large release does, and you have to be sure that you haven't overlooked any technical dependencies in your planning. If you plan to implement autosave in delivery 3, you'd better be sure that manual save isn't planned for delivery 4. Be sure that the development team reviews the delivery plan with an eye toward technical dependencies before you promise specific features at specific times to your customers.

Developer Focus

Staged Delivery requires that each developer meet the deadline for each stage. If one developer misses a deadline, the whole release can slide, some

developers are used to working solo, performing their assignments in whatever order they choose. Some developers might resent the restrictions that a Staged Delivery plan imposes. If you allow developers the amount of freedom they are used to, you'll miss your delivery dates and lose much of the value of staged deliveries.

With Staged Delivery, developers can't follow their noses as much as they're used to doing. You need to be sure that the developers have bought into the delivery plan and agreed to work to it. The best way to ensure developer buy-in is to involve developers intimately in the creation of the plan. If the delivery plan is *their* delivery plan—and if there was no heavy hand influencing their work—you won't have to worry about getting their buy-in.

Theme Releases

CROSS-REFERENCE
For details of another kind of theme, see "Shared, Elevating Vision or Goal" in Section 12.3.

A good way to define the stages is to use themes for each incremental release (Whitaker 1994). Defining releases can give rise to feature-by-feature negotiations that can take up a lot of time. The use of themes raises those negotiations to a higher level.

In the delivery schedule mapped out in Table 36-1 on page 552, the themes are text editing, basic formatting, advanced formatting, utilities, and integration. These themes make it easier to decide into which release to put a particular capability. Even if the feature straddles a gray area—you could classify automatic list numbering as either advanced formatting or a utility, for example—your job will be easier because you only have to decide which of the two themes is most appropriate. You don't have to consider every feature for every release.

When you use themes, you probably won't be able to deliver features in exact priority order. Instead, plan to prioritize the themes in order of importance, and then deliver the themes in priority order.

The use of themes, shouldn't be taken as an invitation to abbreviate release planning. Map out exactly which features you plan to have in each release. If you don't, you won't know exactly what to expect at each delivery stage, and you'll lose much of the project-tracking benefit of this lifecycle model.

Kinds Of Projects

Staged Delivery works best for well-understood systems. If you're not sure what features your product should have, then the Staged Delivery practice isn't a good choice. You have to understand the product well enough to plan the stages by the time you're done with architectural design.

Staged Delivery works especially well when your customers are eager to begin using a relatively small portion of the product's functionality. You can

CROSS-REFERENCE
You can use the Miniature Milestones practice to track progress during each stage. For details, see Chapter 27, "Miniature Milestones."

provide a valuable service to your customers if you can provide the 20 percent-of-the-product they most need in a fraction of the development time needed for the complete product.

Staged Delivery is also an appropriate practice for very large projects. Planning a series of four 9-month projects is considerably less risky than planning a single 3-year project. You would probably have stages within stages for a project of that size; even a 9-month project is too large to provide good progress visibility- to your customer, and you should break it up into several incremental releases.

Staged Delivery works well only for systems in which you can develop useful subsets of the product independently. Most end-user products can be defined in such a way that you can make meaningful intermediate deliveries before you deliver the final product. But operating systems, some kinds of embedded systems, and some other kinds of products might not be usable without complete or nearly complete functionality-; for those kinds of systems. Staged Delivery is not appropriate. So if you can't figure out how to break the delivery of your product up into stages, Staged Delivery is not the right approach for you.

36.2 Managing the Risks of Staged Delivery

The preceding discussion might give you the idea that Staged Delivery works almost every time, but keep this limitation in mind.

Feature creep. The main risk associated with Staged Delivery is the risk of feature creep. When customers begin to use the first release of your product, they are likely to want to change what has been planned for the other releases.

CROSS-REFERENCE
For a variation of Staged Delivery that works better when requirements aren't stable, see Chapter 20, "Evolutionary Delivery."

The best way to manage this risk is not to use Staged Delivery if you're uncertain what features need to be developed. Pure Staged Delivery does not provide much flexibility to respond to customer requests. Staged Delivery works best when you have a broad and deep consensus about what should be in the product.

If you decide to use Staged Delivery, you can still build time into your schedule to accommodate unknown features. You might define the last stage as the stage for making late-breaking changes. By allocating that time, you make it clear to your customers that you intend to be flexible, but you also make it clear that you expect to limit the number of unknown features that you implement. Of course, when you get to the last stage you can renegotiate the schedule if your customers want more features than you have time for. By then your customers will have working software in their hands, and **they**

might well find that their initial schedule goal is no longer as important as it once seemed.

In addition to these practices specific to Staged Delivery, you can use any of the general means of managing feature creep. Those are described in Chapter 14, "Feature-Set Control."

36.3 Side Effects of Staged Delivery

In addition to its positive effect on project scheduling, Staged Delivery can benefit several other project characteristics.

More even distribution of development and testing resources. Projects using Staged Delivery consist of several minicycles of planning, requirements analysis, design, code, and test. Design isn't bunched up at the beginning, programming isn't bunched up in the middle, and testing isn't bunched up at the end. You can distribute analysis, programming, and testing resources more uniformly than you can with approaches that are closer to the pure waterfall model.

Improved code quality. In traditional approaches, you know that "someone" will have to read your code and maintain it. That provides an abstract motivation to write good code. With Staged Delivery, you know that *you* will need to read and modify the code many times. That provides a concrete motivation to write good code, which is a more compelling incentive (Basi and Turner 1975).

More likely project completion. A staged-delivery project won't be abandoned as easily as a waterfall-model project. If the project runs into funding trouble, it's less likely to be canceled if the system is 50 percent complete, 50 percent operational, and in the users' hands than if it's 90 percent complete, doesn't work at all, and has never been touched by anyone outside the development team.

Support for build-to-budget. The premise of Staged Delivery is that you deliver something useful as early as possible. The project will be partially complete even if your customers run out of money. At the end of each stage you and your customers can examine the budget and determine whether they can afford the next stage. Thus, even if the well runs dry, the product still might be largely usable. In many cases, the last 1.0 or 20 percent of the product consists of optional capabilities that aren't part of the product's core. Even if a few frills are missing, the customers will get most of the necessary capabilities. If funding runs out at the 90-percent mark, imagine how much happier your customers will be with a mostly functioning product than if you

had used a pure waterfall lifecycle model—and "90 percent complete" meant "0 percent operational."

36.4 Staged Delivery's Interactions with Other Practices

Although there are a few similarities, Staged Delivery is not a form of prototyping. Prototyping is exploratory, and Staged Delivery is not. Its goal is to make progress visible or to put useful software into the customers' hands more quickly. Unlike prototyping, you know the end result when you begin the process.

If the Staged Delivery practice provides less flexibility than you need, you can probably use Evolutionary Delivery (Chapter 20) or Evolutionary Prototyping (Chapter 21) instead. If your customers get the stage 1 release and tell you that what you're planning to deliver for stage 2 won't suit them, you'd have to be pigheaded not to change course. If you know the general nature of the system you're building but still have doubts about significant aspects of it, don't use Staged Delivery.

Staged Delivery combines well with Miniature Milestones (Chapter 27). By the time you get to each stage, you should know enough about what you're building to map out the milestones in detail.

Success at developing a set of staged deliveries depends on designing a family of programs ("Define Families of Programs" in Section 19.1). The more you follow that design practice, the better able you'll be to avoid disaster if the requirements turn out to be less stable than you thought.

Why Is Staged Delivery So Rigid?

Staged Delivery doesn't always have to be as rigid as it's described in this chapter. Sometimes it's desirable to map out an entire product and deliver it in stages, as described here. Sometimes you'll want more flexibility—rigidly defining the early stages and allowing for more flexibility in later stages. Sometimes you'll want to evolve the product throughout its development, as you do with Evolutionary Prototyping (Chapter 21).

This chapter describes the most rigid version of Staged Delivery in order to provide a clear contrast between it (at one end of the scale) and Evolutionary Prototyping (at the other). For your own use, you can adapt Staged Delivery (or Evolutionary Delivery or Evolutionary Prototyping) to the specific needs of your project and call it anything you like.

As for the objection that throwing away the prototype costs too much, done right, the reason you create a throwaway prototype is that it is cheaper to develop a throwaway prototype, learn lessons the cheap way, and then implement the real code with fewer mistakes than it is not to create the throwaway prototype in the first place. If you can think of some other method that will be more cost effective in a specific situation, use that instead. Otherwise, far from creating extra costs, Throwaway Prototyping is the most cost-effective practice available.

CROSS-REFERENCE

For more on using prototyping time effectively, see "Inefficient use of prototyping time" in Section 20.2.

Inefficient use of prototyping time. As with Evolutionary Prototyping, projects often waste time during Throwaway Prototyping and unnecessarily lengthen the development schedule. Although prototyping is by nature an exploratory process, that does not mean that it has to be an open-ended process.

Monitor prototyping activities carefully. Treat each throwaway prototype as an experiment. Develop a hypothesis such as, "A disk-based merge-sort will sort 10,000 records in less than 30 seconds." Then be sure that the prototyping activity stays focused on proving or disproving the hypothesis. Don't let it stray off into related areas, and make sure that the prototyping stops as soon as the hypothesis has been proved or disproved.

CROSS-REFERENCE

For more on creating realistic schedule and budget expectations, see "Unrealistic schedule and budget expectations" in Section 20.2.

Unrealistic schedule and budget expectations. As with other kinds of prototyping, when users, managers, or marketers see rapid progress on a prototype, they sometimes make unrealistic assumptions about how quickly the final product can be developed. The time required to move from a throwaway-prototype implementation to implementation in the target language is sometimes grossly underestimated.

The best way to combat this risk is to estimate the development of the prototype and the development of the final product as separate projects.

38.3

Side Effects of Throwaway Prototyping

In addition to its rapid-development benefits, Throwaway Prototyping produces many side effects, most of which are beneficial. Prototyping tends to:

- Reduce project risk (since you explore risky implementation areas early)
- Improve maintainability
- Provide resistance to creeping requirements
- Provide a good opportunity to train inexperienced programmers since the code they write will be thrown away anyway

38.4 Throwaway Prototyping's Interactions with Other Practices

You can use prototyping in one form or another on most kinds of projects regardless of what other practices are used. Even in projects in which you can't use full-scale Evolutionary Prototyping (Chapter 21), you can still often use Throwaway Prototyping to explore key risk areas.

38.5 The Bottom Line on Throwaway Prototyping

The greatest schedule benefit of Throwaway Prototyping arises from its risk-reduction potency. It might not shorten a schedule at all, but by exploring high-risk areas early, it reduces schedule volatility. Any direct schedule benefits from Throwaway Prototyping depend on what specific area of a product is prototyped.

Steven J. Andriole has run his own requirements-modeling and prototyping business since 1980 and is the author of *Rapid Application Prototyping* (Andriole 1992). He says that the main lesson he's learned in his business is that the Throwaway Prototyping practice, when used to clarify requirements, "is *always* cost effective and *always* improves specifications" (his emphasis) (Andriole 1994).

38.6 Keys to Success in Using Throwaway Prototyping

Here are the keys to using the Throwaway Prototyping practice successfully:

- Choose your prototyping language or environment based on how quickly it will allow you to create throwaway code.
- Be sure that both management and technical staffs are committed to throwing away the throwaway prototype.
- Focus prototyping efforts on areas that are poorly understood.
- Treat prototyping activities as scientific experiments, and monitor and control them carefully.

Further Reading

For further reading on prototyping, see Chapter 11. "Evolutionary Prototyping."

Timebox Development



Timebox Development is a construction-time practice that helps to infuse a development team with a sense of urgency and helps to keep the project's focus on the most important features. The Timebox Development practice produces its schedule savings through redefining the product to fit the schedule rather than redefining the schedule to fit the project. It is most applicable to in-house business software, but it can be adapted for use on specific parts of custom and shrink-wrap projects. The success of timeboxing depends on using it only on appropriate kinds of projects and on management's and end-users' willingness to cut features rather than stretch the schedule.

S
U
M
M
A
R
Y

Efficacy

Potential reduction from nominal schedule:	Excellent
Improvement in progress visibility:	None
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Good
Chance of long-term success:	Excellent

Major Risks

- Attempting to timebox unsuitable work products
- Sacrificing quality instead of features

Major interactions and Trade-Offs

- Depends on the use of Evolutionary Prototyping
- Trades feature-set control for development-time control
- Often combined with JAD, CASE tools, and Evolutionary Prototyping on RAD projects
- Can be combined with Evolutionary Delivery when timing of deliveries matters more than contents

It's amazing how much you can get done the day before you leave for a vacation. You can pick up the dry cleaning, pay the bills, stop the mail and the newspaper, buy new travel clothes, pack, buy film, and drop off a key with the neighbors. Just as amazing are all the things that you don't do the day before you leave. You don't seem to need to spend quite as long in the shower that morning or as long reading the newspaper that night. You might have many other things that you would like to do that day, but suddenly some of those day-to-day priorities slip down a notch.

Timebox Development is a means of harnessing the same sense of urgency that accompanies preparing for a vacation except that it usually accompanies preparing to work hard instead! When you follow the Timebox Development practice, you specify a maximum amount of time that you will spend constructing a software system. You can develop as much as you want or whatever kind of system you want, but you have to constrain that development to a fixed amount of time. This sense of urgency produces several results that support rapid development.

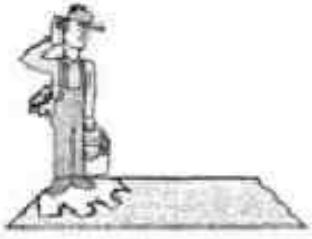
CROSS-REFERENCE

For more on trading off resources and product attributes for schedule gains, see Section 6.6, "Development-Speed Trade-Offs."

It emphasizes the priority of the schedule. By making the schedule *absolutely fixed*, you stress that the schedule is of utmost importance. The time limit, or "timebox," is so important that it overrides all other considerations. If the project scope conflicts with the time limit, you reduce the scope to fit the time limit. The time limit itself is not allowed to change.

It avoids the 90-90 problem. Many projects get to the point where they are 90 percent complete and then stay at that point for months or even years. Many projects spend an undue amount of time in sinkholes that don't move the project forward but that consume huge amounts of resources. You build a small version first, and you build it quickly so that you can get on to version 2. Rather than building a feature-rich first version, it's often more efficient to get a basic version working, learn from the experience, and build a second version after that.

It clarifies feature priorities. Projects can expend a disproportionate amount of time quibbling about features that make little difference in a product's utility. "Should we spend an extra 4 weeks implementing full-color print-preview, or is black-and-white good enough? Should the 3D sculpting on our buttons be one pixel wide or two? Should our code editor reopen text files in the exact location they were last used or at the top of the file?" Rather than spending time arguing about whether to include features of "very low" priority or only "moderately low priority," tight time constraints focus attention on the top end of the priority list.



CLASSIC MISTAKE

CROSS-REFERENCE

For details on the way that gold-plating can occur unintentionally, see "Unclear or Impossible Goals" in Section 14.2.



HARD DATA

It limits developer gold-plating. Within the bounds of what was specified, you can often implement a particular feature in several ways. There is often a 2-day, 2-week, and 2-month version of the same feature. In the absence of the clarifying presence of a development timebox, developers are left to choose an implementation based on their own goals for quality, usability, or level of interest in the feature's design and implementation. Timeboxing makes it clear that if there is a 2-day version of a feature, that is what you want.

It controls feature creep. Feature creep is generally a function of time and averages about 1 percent per month on most projects (Jones 1994). Timebox Development helps to control feature creep in two ways. First, if you shorten the development cycle, you reduce the amount of time people have to lobby for new features. Second, some feature creep on long projects arises from changing market conditions or changes in the operational environment in which the computer system will be deployed. By cutting development time, you reduce the amount that the market or the operational environment can change and thus the need for corresponding changes in your software.

CROSS-REFERENCE

For more on motivation, see Chapter 11, "Motivation."

It helps to motivate developers and end-users. People like to feel that the work they're doing is important, and a sense of urgency can contribute to that feeling of importance. A sense of urgency can be a strong motivator.

39.1 Using Timebox Development

Timebox Development is a construction-phase practice. Developers implement the most-essential features first and the less-essential features as time permits. The system grows like an onion with the essential features at the core and the other features in the outer layers. Figure 39-1 on the next page illustrates the timebox process.

CROSS-REFERENCE

For details on this kind of prototyping, see Chapter 21, "Evolutionary Prototyping."

Construction in Timebox Development consists of developing a prototype and evolving it into the final working system. Timeboxing usually includes significant end-user involvement and ongoing reviews of the developing system.

Timeboxes usually last from 60 to 120 days. Shorter periods are usually not sufficient to develop significant systems. Longer periods do not create the sense of urgency that creates the timebox's clear focus. Projects that are too big for development in a 120-day timebox can sometimes be divided into multiple timebox projects, each of which can be developed within 60 to 120 days.

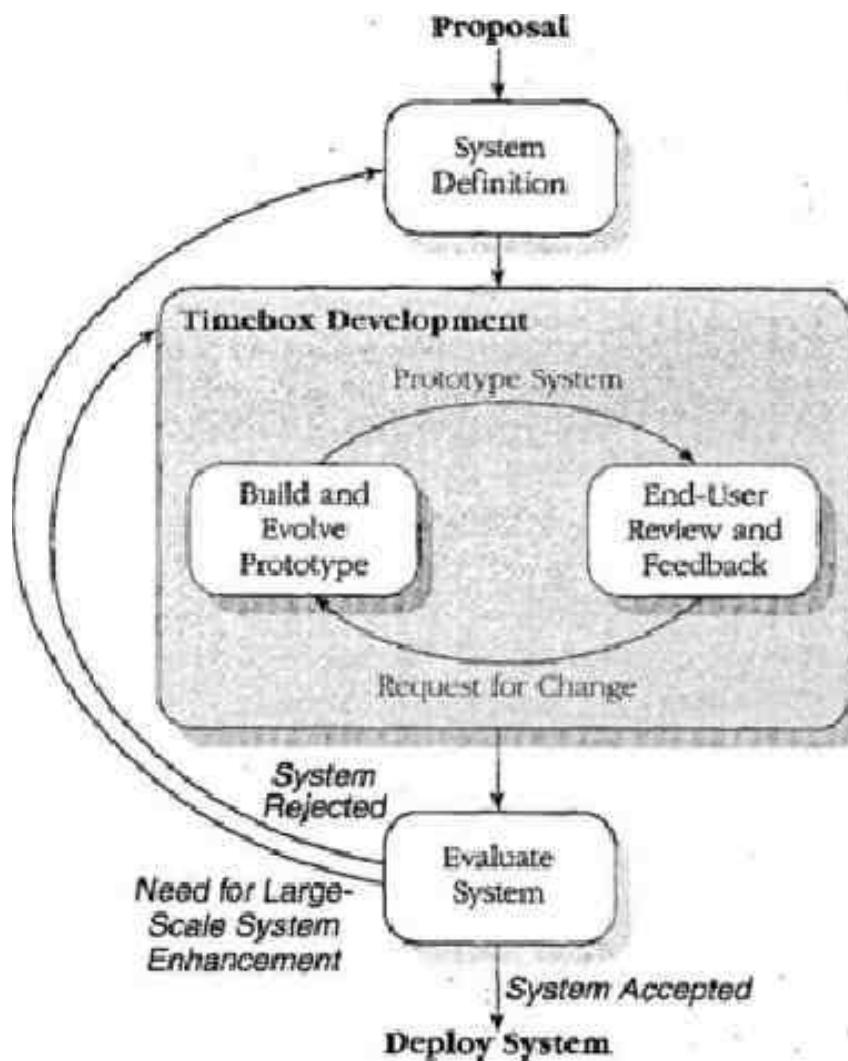


Figure 39-1. *Timebox Development cycle. Timebox Development consists of constructing and evolving an Evolutionary Prototype with frequent end-user interaction,*

After the construction phase, the system is evaluated and you choose from three options:

- Accept the system and put it into production.
- Reject the system because of a construction failure. It might have insufficient quality, or the development team might not have been able to implement the minimum amount of functionality needed for the core system. If that happens, the organization can launch a new Timebox Development effort.
- Reject the system because it does not meet the needs of the organization that built it. A perfectly legitimate outcome of a timebox development is for the team to develop the core system that was identified before Timebox Development began, but for end-users to conclude that the system is not what they wanted. In that case, work begins anew at the system-definition stage, as shown in Figure 39-1.

The people who evaluate the system include the executive sponsor, one or more key users, and a QA or maintenance representative. Technical support and auditing personnel can also be involved in the evaluation.

Regardless of the outcome, it is critical to the long-term success of timeboxing that the timebox not be extended. The end-date for the limebox is not a *due date*. It's a *deadline*. It needs to be clear to the timebox team that whatever they have completed at the end of the timebox is what will be either put into operation or rejected. If the organization has a history of extending its timebox deadlines, developers won't take the deadline seriously and the practice will lose much of its value.

Entrance Criteria for Timebox Development

Timeboxing is not suited for all kinds of development. Here are some guidelines you can use to be sure it is suitable for your project.

Prioritized list of features. Before timebox construction can begin, the functions and design framework of the system need to be defined. The end-users or customers need to have prioritized the system's features so that developers know which are essential and which are optional. They should have defined a minimal core feature set that you are sure you can implement within the timebox time frame. If this prioritization cannot be done, the system is not well-suited to timebox development.

Realistic schedule estimate created by the timebox team. An estimate for the timebox construction should be created by the development team. The construction team needs to estimate both how much time they need (usually 60 to 120 days) and how much functionality they think they can implement within that period. From a motivation point of view, it is essential that the team create its own estimate. Timeboxing is an ambitious practice, and it won't succeed if developers are simply presented with an impossible combination of schedule and functionality goals.

Right kind of project. Timeboxing is best suited for in-house business software (IS software). Timeboxing is an evolutionary prototyping practice and should be built with rapid-development languages, CASE tools, or other tools that support extremely rapid code generation. Highly custom **applications** that require hand-coding are usually not appropriate projects for timebox development. Before beginning a timebox project, verify that you can build the project with the available tool set and staff.

Sufficient end-user involvement. As with other prototyping-based activities, the success of Timebox Development depends on good feedback from end-users. If you can't get adequate end-user involvement, don't use a limebox,

CROSS-REFERENCE
For more on motivation and setting realistic goals, see "Goal Setting" in Section 11.2 and Section 43.1, "Using Voluntary Overtime."

CROSS-REFERENCE
For more on languages that support rapid code generation, see Chapter 31, "Rapid-Development Languages (RDLs)."

CROSS-REFERENCE
For more on effective teamwork, see Chapter 12, 'Teamwork.'

The Timebox Team

A timebox-development team can consist of from one to five people. The full "timebox team" also includes end-users who have been designated to assist the construction team. These end-users are often dedicated full-time to their role of supporting the construction team.

The timebox team needs to be skilled in developing systems with the rapid-development software that will be used. There is no time to learn new software on a timebox project.

The timebox team needs to be highly motivated. The urgency created by the timebox development practice itself will provide some of the motivation. The ability to achieve a level of productivity rare within the organization should provide the rest.

Although my understanding of developer motivation makes me wary of his advice, James Martin recommends that motivation on a timebox project also include the following (Martin 1991):

- Tell developers that they will be judged by whether they create a system that is in fact accepted. Point out that most timebox efforts succeed and that they shouldn't distinguish themselves by being one of the rare failures.
- Tell developers that success will be rewarded and that their efforts are visible to upper management. Follow through on the rewards.
- Tell developers that if they succeed, you'll hold a major victory celebration. Follow through on the celebration.

Don't use Martin's advice verbatim without first thinking through how the issues discussed in Chapter 11, "Motivation," apply within your environment.

Variations on Timebox Development

Timeboxing is usually applied to the design and construction phase of entire business systems. It is generally not well-suited to the development of shrink-wrap software products because of the long development times needed. But timeboxing can be quite useful as a strategy for developing parts of software systems—live user-interface prototypes or throwaway prototypes on a shrink-wrap software project. Timeboxes for prototypes are much shorter than the time recommended for information systems, perhaps on the order of 6 to 12 days rather than 60 to 120. The development team will have to define a timebox that makes sense for the specific prototype they're building.

You can use timeboxing on a variety of implementation activities—software construction, help-screen generation, user-documentation, throwaway prototypes, training-course development, and so on.

39.2 Managing the Risks of Timebox Development

Here are some of the problems with timeboxing.



HARD DATA



FURTHER HEADING

For a different point of view on timeboxing upstream activities, see "Timeboxing" (ZahniseM995).

CROSS-REFERENCE

For more on the effects of conflicting goals, see "Goal Setting" in Section 11.2. For more on the effects of low quality, see Section 4.3, "Quality-Assurance Fundamentals."

Attempting to timebox unsuitable work products. I don't recommend using timeboxes for upstream activities (or beginning-of-the-food-chain activities) such as project planning, requirements analysis, or design—because work on those activities has large downstream implications. A \$100 mistake in requirements analysis can cost as much as \$20,000 to correct later (Boehm and Papaccio 1988). The software-project graveyard is filled with the bones of project managers who tried to shorten upstream activities and wound up delivering software late because small upstream defects produced large downstream costs. Time "saved" early in the project is usually a false economy.

Timeboxing is effective on activities at the end of the development food-chain because the penalty for poor-quality work is limited to throwing away the timeboxed work and doing it over. Other work isn't affected.

Sacrificing quality instead of features. If your customer isn't committed to the timebox practice of cutting features instead of quality, don't use a timebox. Developers have a hard time meeting conflicting goals, and if the customer insists on a tight schedule, high quality, and lots of features, developers won't be able to meet all three objectives at once. Quality will suffer.

Once quality begins to suffer, the schedule will suffer too. The team will produce feature-complete software by the timebox deadline, but the quality will be so poor that it will take several more weeks to bring the product up to an acceptable level of quality.

With true timeboxing, the software is either accepted or thrown away at the timebox deadline. That makes it clear that the quality level must be acceptable at all times. The success of timeboxing depends on being able to meet tight schedules by limiting the product's scope, not its quality.

39.3 Side Effects of Timebox Development

Timebox Development's influence is limited to shortening development schedules. It does not typically have any influence—positive or negative—on product quality, usability, functionality, or other product attributes.

39.4 Timebox Development's Interactions with Other Practices

Timebox Development is a specific kind of design-to-schedule practice (Section 7.7). It is an essential part of RAD, which means that it is often combined with JAD (Chapter 24), CASE tools, and Evolutionary Prototyping (Chapter 21). Because Timebox Development calls for an unusual degree of commitment on the part of the development team, it is also important that each team member be Signed Up (Chapter 34) for the project.



FURTHER READING

The milestone process that Microsoft uses could be considered to be a modified timebox approach. For details, see *Microsoft Secrets* (Cusumano and Seiby 1995).

Timeboxes can also be combined with Evolutionary Delivery (Chapter 20) if you need to define each delivery cycle more by the time you complete it than by the exact functionality you deliver. Similarly, shrink-wrap and other kinds of projects can use timeboxes as part of a staged, internal-delivery approach. Delivering the software at well-defined intervals helps to track the progress and quality of the evolving product. Most projects that use timeboxes in this way won't be willing to throw away work that isn't completed by the deadline, so they won't be using pure timeboxes. But they can still realize some of timebox development's motivational, prioritization, and feature-creep benefits.

39.5 The Bottom Line on Timebox Development



HARD DATA

Timebox Development has been found to produce extraordinary productivity at DuPont, where it was initially developed. DuPont averages about 80 function points per person month with timeboxing, compared to 15 to 25 with other methodologies (Martin 1991). Moreover, timebox development entails little risk. System evaluation and possible rejection is an explicit part of the practice, but after its first few years of use, DuPont had not rejected a single system developed with timeboxing. Scott Shultz, who created the methodology at DuPont, says that "[a]ll applications were completed in less time than it would have taken just to write the specifications for [an application in] Cobol or Fortran" (Shultz in Martin 1991).

39.6 Keys to Success in Using Timebox Development

Here are the keys to success in using timeboxing:

- Use timeboxing only with projects that you can complete within the timebox time frame (usually from 60 to 120 days).
- Be sure that end-users and management have agreed to a core feature set that the timebox construction team believes it can implement within the timebox time frame. Be sure that features have been prioritized, and that you can drop some of them from the product if needed to meet the schedule.
- Be sure that the timebox team is signed up for the ambitious timebox project. Provide any motivational support needed.
- Keep the quality of the software high throughout the timebox.
- If you need to, cut features to make the timebox deadline. Don't extend a timebox deadline.

Further Reading

Martin, James. *Rapid Application Development*. New York: Macmillan Publishing Company, 1991. Chapter 11 discusses timebox development specifically. The rest of the book explains the *RAD* context within which Martin suggests using timeboxes.

Tools Group



The Tools Group practice sets up a group that's responsible for gathering intelligence about, evaluating, coordinating the use of, and disseminating new tools within an organization. A Tools Group allows for a reduced amount of trial and error and minimizes the number of development groups that will be handicapped by the use of inadequate software tools. Virtually any organization that has more than one software project running at a time can set up a Tools Group, though in some cases the "group" might consist of a single person working only part time.

S
U
M
M
A
R
Y

Efficacy

Potential reduction from nominal schedule:	Good
Improvement in progress visibility:	None
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Good
Chance of long-term success:	Very Good

Major Risks

- Bureaucratic overcontrol of information about and deployment of new tools.

Major Interactions and Trade-Offs

- The same basic structure can be used by software-reuse and software-engineering process groups.

For more on tools groups, see "Tools Group" in Section 15-3-

Top-10 Risks List



The Top-10 Risks List is a simple tool that helps to monitor a software project's risks. The list consists of the 10 most serious risks to a project ranked from 1 to 10, each risk's status, and the plan for addressing each risk. The act of updating and reviewing the Top-10 Risks List each week raises awareness of risks and contributes to timely resolution of them.

Efficacy

Potential reduction from nominal schedule:	None
Improvement in progress visibility:	Very Good
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Excellent
Chance of long-term success:	Excellent

Major Risks

None

Major Interactions and Trade-Offs

- Can be used in combination with virtually any other practice.

For more on top-10 risks lists, see "Top-10 Risks List" in Section 5.5.

S
U
M
M
A
R
Y

- Pressman, Roger S. 1988. *Making Software Engineering Happen: A Guide for Instituting the Technology*. Englewood Cliffs, N.J.: Prentice Hall.
- Pressman, Roger S.' 1992. *Software Engineering: A Practitioner's Approach*, 3d ed. New York: McGraw-Hill.
- Pressman, Roger S. 1993. *A Manager's Guide to Software Engineering*. New York:McGraw-Hill.
- Putnam, Lawrence H. 1994. "The Economic Value of Moving Up the SEI Scale." *Managing System Development*, July: 1-6.
- Putnam, Lawrence H., and Ware Myers. 1992. *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, N.J.: Yourdon Press.
- Raytheon Electronic Systems. 1995. *Advertisement, IEEE Software*, September: back cover.
- Rich, Charles, and Richard C. Waters. 1988. "Automatic Programming: Myths and Prospects." *IEEE Computer*, August.
- Rifkin, Stan, and Charles Cox. 1991. "Measurement in Practice." Report CMU/SEI-91-TR-16, Pittsburgh: Software Engineering Institute.
- Rothfeder, Jeffrey. 1988. "It's Late, Costly, Incompetent—But Try Firing a Computer System." *Business Week*, November 7: 164-165.
- Rush, Gary. 1985. "The Fast Way to Define System Requirements." *Computerworld*, October 7.
- Russell, Glen W. 1991. "Experience with Inspection in Ultralarge-Scale Developments." *IEEE Software*, vol. 8, no. 1 (January): 25-31.
- Sackman, H., W. J. Erikson, and E. E. Grant. 1968. "Exploratory Experimental Studies Comparing Online and Offline Programming Performance." *Communications of the ACM*, vol. 11, no. 1 (January): 3—11.
- Saiedian, Hossein, and Scott Hamilton. 1995. "Case Studies of Hughes and Raytheon's CMM Efforts." *IEEE Computer*, January: 20-21.
- Scherr, Allen. 1989. "Managing for Breakthroughs in Productivity." *Human Resources Management*, vol. 28, no. 3 (Fall): 403-424.
- Scholtz, et al. 1994. "Object-Oriented Programming: the Promise and the Reality." *Software Practitioner*, January: 1, 4-7.
- Sherman, Roger. 1995a. "Balancing Product-Unit Autonomy and Corporate Uniformity." *IEEE Software*, January: .110-111.
- Sims, James. 1995. "A Blend of Technical and Mediation'Skills Sparks Creative Problem-Solving." *IEEE Software*, September: 92-95.

- Smith, P.O., and D.G. Reinertsen. 1991. *Developing Products in Half the Time*. New York: Van Nostrand Reinhold.
- Sommerville, Ian. 1996. *Software Engineering*, 6th ed. Reading, Mass.: Addison-Wesley.
- Standish Group, The. 1994. "Charting the Seas of Information Technology." Dennis, Mass.: The Standish Group.
- Symons, Charles. 1991. *Software Sizing and Estimating: Mk IIFPA (Function Point Analysis)*. Chichester: John Wiley & Sons.
- Tesch, Deborah B., Gary Klein, and Marion G. Sobol. 1995. "Information System Professionals' Attitudes. "Journal of Systems and Software, January: 39-47.
- Thayer, Richard H., ed. 1990. *Tutorial: Software Engineering Project Management*. Los Alamitos, Calif.: IEEE Computer Society Press.
- Thomsett, Rob. 1990. "Effective Project Teams: A Dilemma, A Model, A Solution." *American Programmer*, July-August: 25-35.
- Thomsett, Rob. 1993. *Third Wave Project Management*. Englewood Cliffs. NJ.: Yourdon Press.
- Thomsett, Rob. 1994. "When the Rubber Hits the Road: A Guide to Implementing Self-Managing Teams." *American Programmer*, December: 37-45.
- Thomsett, Rob. 1995. "Project Pathology: A Study of Project Failures." *American Programmer*, July: 8-16.
- Townsend, Robert. 1970, *Up the Organization*. New York: Alfred A. Knopf,
- Tracz, Will. 1995. *Confessions of a Used Program Salesman*. Reading, Mass.: Addison-Wesley.
- Udell, John. 1994. "Component Software." *Byte* magazine, May.- 45-55.
- Valett, J., and F. E. McGarry. 1989. "A Summary of Software Measurement Experiences in the Software Engineering Laboratory." *Journal of Systems and Software*, 9 (2): 137-148.
- van Genuchten, Michiel. 1991. "Why is Software Late? An Empirical Study of Reasons for Delay in Software Development." *IEEE Transactions on Software Engineering*, vol. 17, no. 6 (June): 582-590.
- Vosburgh, J. B., et al. 1984. "Productivity Factors and Programming Environments." *Proceedings of/be 7th International Conference on Software Engineering*. Los Alamitos, Calif.- IEEE Computer Society:' 143-152.
- Weinberg, Gerald M. 1971. *the Psychology of Computer Programming*. New York: Van-NostrandReinhold.

Bibliography

- Weinberg, Gerald. 1982. *Becoming a Technical Leader*. New York: Dorset House.
- Weinberg, Gerald M. 1992. *Quality Software Management, Volume 1: Systems Thinking*. New York: Dorset House.
- Weinberg, Gerald M. 1993. *Quality Software Management, Volume 2: First-Order Measurement*. New York: Dorset House.
- Weinberg, Gerald M. 1994. *Quality Software Management, Volume 3: Congruent Action*. New York: Dorset House.
- Weinberg, Gerald M., and Edward L. Schulman. 1974. "Goals and Performance in Computer Programming." *Human Factors*, vol. 16, no. 1 (February): 70-77.
- Whitaker, Ken. 1994. *Managing Software Maniacs*. New York: John Wiley & Sons.
- Wiener, Lauren Ruth. 1993. *Digital Woes: Why We Should Not Depend on Software*. Reading, Mass.: Addison-Wesley.
- Wirth, Niklaus. 1995. "A Plea for Lean Software." *IEEE Computer*, February: 64-68.
- Witness*. 1985. Paramount Pictures. Produced by Edward S. Feldman and directed by Peter Weir.
- Wood, Jane, and Denise Silver. 1995. *Joint Application Development*, 2nd ed. New York: John Wiley & Sons.
- Yourdon, Edward. 1982, ed. *Writings of the Revolution: Selected Readings on Software Engineering*. New York: Yourdon Press.
- Yourdon, Edward. 1989a. *Modem Structured Analysis*. New York: Yourdon Press.
- Yourdon, Edward. 1989b. *Structured Walk-Throughs*, 4th ed. New York: Yourdon Press.
- Yourdon, Edward. 1992. *Decline & Fall of the American Programmer*. Englewood Cliffs, N.J.: Yourdon Press.
- Yourdon, Edward, and Constantine, Larry L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, N.J.: Yourdon Press.
- Yourdon, Edward, ed. 1979. *Classics in Software Engineering*. Englewood Cliffs, N.J.: Yourdon Press.

..:

- Zachary, Pascal. 1994. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. New York: Free Press.
- Zahniser, Rick. 1995. "Controlling Software Projects with Timeboxing." *Software Development*, March.
- Zawacki, Robert A. 1993. "Key Issues in Human Resources Management." *Information Systems Management*, Winter: 72-75.
- Zelkowitz, et al. 1984. "Software Engineering Practice in the US and Japan." *IEEE Computer*, June: 57-66.

morale budget at Microsoft (case study), 270
More Programming Pearls (Bentley), 68
motivation(s), 14
 achievement as, 255-57
 case studies
 disheartening lunch with the boss, 250-51
 highly motivational environment, 270-71
 commitment-based approach and, 23
 defined, 254
 of developers, 251-54
 further reading on, 271-72
 heavy-handed campaigns to improve, as morale killer, 269
 miniature milestones and, 483
 minimal-specification approach and, 325-26
 morale killers, 265-69
 excessive schedule pressure, 267
 heavy-handed motivation campaigns, 269
 hygiene factors, 265-66
 inappropriate involvement of technically inept management, 268
 lack of appreciation for developers, 267
 low quality, 269
 management manipulation, 266-67
 not involving developers, 268—69
 productivity barriers, 269
 overly optimistic scheduling and, 216
 performance reviews and, 265
 personal life as, 260-61
 pilot projects and, 263-65
 possibility for growth as, 257-58
 rewards and incentives and, 262-63
 technical-supervision opportunity as, 261
 timebox development and, 581
 undermined, as classic mistake, 40
 work itself as, 258-60
motivation (motivation factors), 249—72
mutual trust, teamwork and, 285
Myers, Glenford, 73, 74, 76
Myers, Ware, 14, 182, 192, 518
Myers-Briggs Type Indicator (MBTI) test, 253
Mythical Man-Month, 7&e (Brooks), 305, 317, 369, 418

N

Naisbitt, John, 258, 508
NASA, 478
 Software Engineering Laboratory, 74, 277, 352, 421, 469, 474, 475, 530, 533
negotiation (negotiating skills)
 estimates and, 221
 principled (*see* principled negotiation method)
 push-me, pull-me, 47
 separating the people from the problem, 222-23
noisy, crowded offices, 41
nominal schedules, 194—96
"No Silver Bullets-Essence and Accidents of Software Engineering" (Brooks), 348

O

Object Oriented Analysis and Design (Booch), 67
object-oriented design
 designing for change and, 421
 information hiding, 417-19
 planned reuse and, 534
Object-Oriented Design (Goad and Yourdon), 67
object-oriented programming, 367
Object-Oriented Rapid Prototyping (Connell and Shafer), 443
O'Brien, Larry, 358, 370
office space. *See* productivity environments
offshore outsourcing, 496
off-the-shelf software, 153-54
 strengths and weaknesses of, 157
O'Grady, Frank, 294
Oldham, Greg R., 258, 272
Olsen, Neil, 18
"One More Time: How Do You Motivate Employees?" (Herzberg), 272
"On the Criteria to Be Used in Decomposing Systems into Modules" (Parnas), 423
opportunistic efficiency, minimal-specification approach and, 326
optimistic schedules, overly, as classic mistake, 44
optimistic scheduling. *See* scheduling (schedules), overly optimistic

organization of teams, 13
 organization risks, 87-88
 outsourcing, 491-502
 bottom line on, 501
 feature creep and, 492
 further reading on, 502
 interactions with other practices, 501
 managing the risks of, 499-500
 planning and, 493
 requirements specification and, 492
 reusable components and, 492
 side effects of, 501
 staffing flexibility and, 492
 summary of, 491
 time savings with, 492
 using, 493-99
 communication with the vendor, 493
 contract considerations, 498-99
 contract management, 493
 double standards for outsourced work, 495
 keys to success in, 501-2
 kinds of arrangements, 495-96
 legacy-systems reengineering, 494
 management plan including risk
 management, 493
 offshore outsourcing, 496-97
 technical resources, 493-94
 unstable requirements, 494
 vendor evaluation, 497-98
 overrun, project, 94
 overtime
 desire for free, as rapid-development look-alike, 115
 voluntary (.see voluntary overtime)
 ownership, achievement motivation and, 255

P

padding of estimates, 184
 Page-Jones, Meilir, 67 > '
 Papaccio, 45, 62, 71, 212, 335, 585
 Pareto analysis, 473
 Parnas, David L, 417, 419, 420, 422, 423
 Peat Mai-wick, 82

penalty for breaking the build, 410
 people (peopleware issues) *See also* personnel
 (employees; staff)
 classic mistakes related to, 40
 as dimension of development speed, 11, 12-14
 recovery of projects and, 376-79
Peopleware (DeMarco and Lister), 28, 231, 271, 513, 608
 perception of development speed, 119-21
 overcoming the perception of slow
 development, 121
 performance
 product, evolutionary prototyping and, 437
 team structure and monitoring of, 302
 performance reviews, motivation and, 265
 per-person-efficiency trade-offs. 127
 Perry, Dewayne E., 476
 personal life, as motivation, 260-61
 personnel (employees; staff). *See also* people
 (peopleware issues.)
 evaluations of, measurements misused for, 476
 long-term teambuilding and. 293
 outsourcing and staffing flexibility, 492
 problem, 40-41
 failure of teams and, 291-92
 recovery of projects and, 377
 risks associated with, 90
 weak, 40
 Peters, Chris, 255, 324, 413, 536
 Peters, Tomas J., 247, 258, 203, 265, 271, 272, 317
 Pfleeger, Shari Lawrence, 441, 533-36
 philosophy, best practices and. 395
 Pietrasanta, Alfred M., 14, 474
 Pirbhai, Imliaz A., 66
 planned reuse, 531-35
 planning (plans), 486
 abandonment of, under pressure, as classic
 mistake, 44
 change, 419
 customer oriented, 239
 as development fundamental, 56
 insufficient, as classic misiukc, 44
 outsourcing and, 493

- planning (plans), *continued*
- overly optimistic scheduling and adherence to, 211
 - overly optimistic scheduling and quality of, 211
 - risk-management, 85, 96-97
- Plauger, P.J., 28, 66-67
- point-of-departure spec, 324
- politics placed over substance, 43
- postmortems, interim, 102
- "potential reduction from nominal schedule" entry, 392
- Practical Guide to Structured Systems Design, The* (Page-Jones), 67
- Practical Software Metrics for Project Management and Process Improvement* (Grady), 479
- Prasad, Jayesh, xiii, 178, 319
- precision, accuracy vs., 173
- predictability, as rapid-development look-alike, 113
- premature convergence, overly optimistic scheduling and, 213-15
- premature releases, daily build and smoke test and, 412-13
- presentation styles, for estimates, 179-82
- Pressman, Roger S., 59, 75, 79
- principled negotiation method, 222-29, 503
- degrees of freedom in planning a software project and, 225-26
 - focusing on interests, not positions, 224—25
 - insisting on using objective criteria, 227-29
 - inventing options for mutual gain, 225-27
 - separating the people from the problem, 222—23
- Principles of Software Engineering Management* (Gilb), 59, 71, 106, 204, 231, 428, 430, 432, 489, 558
- prioritization of risks, 85, 94—96
- problem-resolution** team, 300
- process
- abuse of focus on, 14—15
 - classic mistakes related to, 44-46
 - as dimension of development speed, 11, 14—16
 - recovery of projects and, 379—82
 - risks associated with, 91
- product. *See also* features (feature set)
- as dimension of development speed, 11, 17
 - risks associated with, 89-90
 - trade-offs among schedule, cost, and, 126-27
- product characteristics, 17
- product families, 420-21
- productivity
- barriers to, 269
 - failure of teams and, 290
 - classic mistakes and (, *see* classic mistakes)
 - factors that affect, 37-38
 - lines-of-code measurement of, 477
 - motivation and, 40
 - peopleware issues and, 12, *Id*
 - per-person-efficiency trade-offs and, 127
- team
- permanent-team strategy, 292
 - variations in, 276-77
- variations in, 14
- among groups (teams), 12
 - among individuals, 12
- productivity environments, 505-13
- bottom line on, 511
 - further reading on, 513
 - interactions with other practices, 511
 - managing the risks of, 510—11
 - side effects of, 511
 - summary of, 505
 - using, 508-9
- keys to success in, 513
- productivity tools, 345-70
- acquisition of, 353-58
 - criteria for tool selection, 356-58
 - risks of setting up a tools group, 355
 - tools person or group, 354—55
- case studies
- effective tool use, 368-69
 - ineffective tool use, 346-47
- defined, 346
- further reading on, 369-70
- maturity of, ;356-57

- rapid prototyping, use of term, 434
"Rapid Prototyping: Lessons Learned" (Gordon and Bieman), 443
Raytheon, 14, 15, 21
RDLs. *See* rapid-development languages
"Realities of Off-Shore Reengineering" (Dedene and De Vreese), 502
real-time systems, 22
recalibration of schedules, 199-200
recognition. *See* appreciation (recognition)
recovery of projects, 371-88. *See also* features (feature set), late-project feature cuts *and* mid-project changes
case studies
 successful project recovery, 385-87
 unsuccessful project recovery, 372-73
characteristics of projects in trouble, 371-72
fundamental approaches to, 373—74
philosophy of this book's approach to, 374
plan for, 375-85
 first steps, 376
 people issues, 376-79
 process, 379-82
 product, 382
 timing, 385
reestimation mistakes, 46
Reifer, Donald J., 493, 502
Reinertsen, D. G., 131
Reinventing the Corporation (Naisbitt and Aburdene), 258
release cycles, short, 339
requirements (requirements specifications or analysis), 61-62, 124
case studies, 234-36, 246
creeping (feature creep) *{see* features (feature set), creeping (creeping requirements))
customer-oriented, 239-42, 338
detailed, 321-23
further reading on, 66
gold-plating, 46
JAD and, 450, 461-62
minimal specification, 323-29
 benefits of, 325-26
 keys to success in using, 328-29
requirements (requirements specifications or analysis), *continued*
 lack of support for parallel activities, 327
 omission of key requirements, 326
 risks of, 326-28
 unclear or impossible goals, 326
 wrong reason for using, 328
outsourcing and, 492, 494
recover)' of projects and, 382-84
risks associated with, 89
scrubbing (entirely removing), 329-30, 525
research-oriented development, 47
resources, targeting, 16
reuse (reusable components), 527-38
 bottom line on, 537
 further reading on, 538
 interactions with other practices, 536-37
 keys to success in using, 537-38
 managing the risks of, 535-36
 outsourcing and, 492
 side effects of, 536
 summary of, 527
 using, 528-35
 opportunistic reuse, 528—31
 planned reuse, 531—35
 , ,
rewards
 case study: at Microsoft, 270-71
 motivation and, 262
rework
 amount of time spent on, 123
 avoidance of, 15
 customer relationships and, 237
Rich, Charles, 366, 367
Rifkin, Stan, 468, 474
risk analysis, 85, 91-94
risk assessment, elements of, 85
risk, assuming, 98
risk control, 85, 96-102
risk exposure, 92
risk identification, 85-91
 complete list of schedule risks, 86-91
 contractor risks, 89
 customer risks, 88-89
 'design and implementation risks, 91

- risk identification, *continued*
- development environment risks, 88
 - end-user risks, 88
 - external environment risks, 90
 - organization and management risks, 87-88
 - personnel risks, 90
 - process risks, 91
 - product risks, 89-90
 - requirements risks, 89
 - schedule creation risks, 87
 - general risks, 85-86
 - most common schedule risks, 86
- risk management, 8, 15, 81-106. *See also* *managing-risks sections of best-practice Chapters 17-43*
- case studies, 82-83
 - systematic risk management, 103-5
 - disadvantages of, 82
 - elements of, 84
 - further reading on, 106
 - insufficient, as classic mistake, 44
 - levels of, 84
 - outsourcing and, 493
 - risk-management planning, 85, 96-97
 - risk monitoring, 85, 100-102
 - risk officer, 102
 - risk prioritization, 85, 94-96
 - risk quantification, estimation and, 180-81
 - risk resolution, 85, 97-100
 - risks, 102
 - customers-related, 237
 - roles, clear, team structure and, 302
- Ross, Rony, 484, 559, 568
- Rothfeder, Jeffrey, 82
- runaway prevention, as rapid-development look-alike, 113
- Rush, Gary W., 461
- Russell, Glen \\", 74
- S**
- Sackman, H., 12, 217, 249
- Saiedian, Hossein, 14
- sashimi model, 143-44
- schedule compression, 191-92
- schedule constraints, 112-13
- schedule creation risks, 87
- schedule estimation, 183-85
- schedule-oriented practices, 3, 9, 10. *See also* speed of development; visibility-oriented practices
- focusing only on, 10
 - focusing too much on single, 5
 - kinds of, 4
- schedule pressure
- beating, 220-29
 - build and smoke testing even under, 411
 - excessive, as morale killer, 267
- schedule recalibration, 199-200
- schedule risk, miniature milestones and, 483
- schedule-risk-oriented practices. *See* fundamentals of software development
- scheduling (schedules). 205-31
- beating schedule pressure, 220-29
 - principled negotiation, 222-29
 - case study: .a successful schedule negotiation, 229-30
 - commitment-based, 184
 - as development fundamental, 55
 - further reading on, 231
 - overly optimistic, 207-20
 - accuracy of schedules and, 210
 - adherence to plan and, 211
 - bottom line and, 218-20
 - burnout and, 217
 - creativity and, 216-17
 - customer relations and, 213
 - developer-manager relationships and, 217
 - effects of, 210-15
 - example of, 207
 - gambling and, 216
 - long-term rapid development and, 217
 - motivation and, 216
 - premature convergence, 213-15
 - project focus and, 213
 - quality and, 215 In
 - quality of project **planning** and, 211
 - root causes of, 2U^C>-10
 - underscoping the project and, 212

- scheduling (schedules), *continued*
 overly optimistic, as classic mistake, 44
 principled negotiation method and, 222-29
 probability of meeting, 116-19
 timebox development and, 580
 trade-offs among product, cost, and, 126
- Scherr, Allen, 285, 540, 542
- Scholtz, 367, 421, 534
- Schulman, Edward, 12, 217, 255, 276
- scientific software, 22
- scribe, in JAD sessions, 455
- scrubbing (entirely removing) features, 329-30, 525
- search-and-rescue team model, 307-8
- Seewaldt, T, 12, 217, 277
- Selby, Richard, 74, 116, 255, 270, 272, 290, 324, 327, 341, 407, 414, 432, 536
- Shafer, Linda, 443
- Shaw, 436
- Shen, V. Y., 479
- Sherman, Roger, 138
- shortest possible schedules, 188-92
 efficient schedules and, 193
- Showstopper!* (Zachary), 272, 388
- shrink-wrap software, defined, 22
- Shultz, Scott, 586
- signing up, 539-45
 bottom line on, 545
 in different environments, 542
 further reading on, 545
 giving people a choice about, 541
 interactions with other practices, 544-45
 keys to success in using, 545
 managing the risks of, 543-44
 shaky requirements and, 542
 side effects of, 544
 at the team level, 541
 vision and, 540
- Silver, Denise, 463
- silver bullets (silver-bullet syndrome), 47-48, 363-69, 520-21
 biting the bullet, 367-68
 identifying, 365-67
- Sims, James, 452
- size estimation, 174-82
 definition of size, 175
 function-point estimation, 174
- size of product, 17
- skill variety, work itself as motivation and, 258
- skunkworks-team structure, 306—7
- Smith, Douglas, 296, 275
- Smith, P. G., 131
- smoke test, fee daily build and smoke test
- Sobol, Marion G., 366
- software, types of, 22
- Software Acquisition Management* (Marciniak and Reifer), 493, 502
- Software Configuration Management* (Babich), 68
- Software Configuration Management* (Bersoff), 68
- software configuration management (SCM)
 as development fundamental, 65-66
 further reading on, 68
- software development. *See* fundamentals of software development; rapid development
- software engineering, further reading on, 79
- Software Engineering* (Pressman), 75, 79
- Software Engineering* (Sommerville), 75, 79
- Software Engineering Economics* (Boehm), 203
- Software Engineering Institute, 14, 28, 55, 57, 59, 66, 70, 374, 502
- Software Engineering Laboratory (NASA), 74, 277, 352, 421, 469, 474, 475, 530, 533
- Software Engineering Metrics and Models* (Conte et al.), 479
- "Software Engineering Under Deadline Pressure" (Costello), 231
- Software Implementation* (Marcotty), 68
- Software Inspection* (Gilb and Graham), 76
- Software Measurement Guidebook* (NASA), 478
- Software Metrics* (Grady and Caswell), 478
- "Software Productivity Research Programming Languages Table" (Jones), 524
- Software Risk Management* (Boehm), 106, 161
- "Software Risk iVImanagement: Principles and Practices" (Boehm), 106
- "Software's Chronic Crisis" (Gibbs), 343

- Theory-W management, *continued*
 using, 561-66
 keys to success in, 568
 kinds of projects that can use Theory-W, 566
 manager's role, 566
 step 1: establish win-win preconditions, 562-64
 step 2: structure a win-win software process, 564
 step 3: structure a win-win software • product, 565
- "Theory-W Software Project Management: Principles and Examples" (Boehm and Ross), 568
- third-generation languages. *See 3GLs*
- Third Wave Project Management (Thomseti)*, 106
- Thomsett, Rob, 42, 106, 253, 293, 294, 317, 388
- Thriving on Chaos* (Peters), 258
- throwaway prototyping, 433, 569-73
 evolutionary prototyping and, 441
 user-interface prototyping and, 591-92
- timebox development, 575-83
 bottom line on, 582
 further reading on, 583 .
 interactions with other practices, 582
 managing the risks of, 581
 side effects of, 581
 using, 577-80
 entrance criteria, 579
 keys to success in, 583
 timebox team, 580
 variations, 580
- tools group, 585
 acquisition of productivity tools and, 354—56
 summary of, 585
- Top-10 Risks List, 100-102, 587
- Townsend, Robert, 287
- tracking, as development fundamental, 56—58
- Tracz, Will, 534-36, 538
- trade-offs
 per-person-efficiency, 127
 quality, 127
 among schedule, cost, and product, 120—27'
- training, in productivity tool use, 357
- transferring risks, 97
- trust
 failure of teams and lack of, 290
 among team members, 285
- Turner, Albert J., 556
- turnover
 long-term teambuilding and, 293
 overly optimistic scheduling and, 217
- Tutorial on Software Design Techniques* (Parnas). 423
- Tutorial: Software Engineering Project Management* (Thayer), 59
- Tutorial: Software Quality Assurance* (Chow, ed.). 76
-  **U**
- Udell, John, 538
- Umphress, David, 49 '
- underscoping, overly optimistic scheduling and, 212
- unrealistic expectations, 42
 of customers, 120—21
 after JAD session, 460
 about performance, evolutionary prototyping and, 437-38
 about schedule and budget
 evolutionary prototyping and, 436
 throwaway prototyping and, 572
- upstream activities, shortchanged, 45
- Ury, William, 231, 562
- user-interface prototyping, 324, 589-97
 bottom line on, 597
 interactions with other practices, 596
 managing the risks of, 595 -96
 side effects of, 596
 summary of, 589
 using, 591-95
 choosing a prototyping language, 592
 end-user feedback and involvement, 593-94
 evolutionary vs. throwaway prototyping, 591-92
 finished product, 595

user-interface prototyping, using, *continued*
 keys to success in, 597
 prototype as a Hollywood movie facade, 593
user manual, as specification, 324

V

Vaishnavi, Vijay K., 417
Valett, J., 12, 249, 277, 469
van Genuchten, Michiel, 46, 184, 199, 210, 486
vendors. *See* outsourcing
version 1 of productivity tools, 356
version 2
 features, 339
 of productivity tools, 357
version 3 of productivity tools, 357
versioned development, 330-31
visibility-oriented practices, 10
vision
 failure of teams and lack of, 289
 managing a high-performance team and, 288
 sharing a, in high-performance teams, 279
 signing up and, 540
vision statement, minimal specification and, 324
Visual C++, Microsoft, 242
voluntary overtime, 599-608
 bottom line on, 606-7
 further reading on, 608
 interactions with other practices, 606
 managing the risks of, 605-6
 side effects of, 606
 summary of, 599
 using, 600-605
 caution about too much overtime, 604—5
 developer-pull vs. leader-push approach, , 600-601
 how much overtime to ask for, 603
 keys to success in, 607
 out-of-control projects, 603
 overtime should not be mandatory, 601-3
Vosburgh, J. B., 37, 38, 60, 240, 241, 256, 319, 335
Votta, Lawrence G., 476

W

Waligora, Sharon, 530, 534
walkthroughs, 73
 estimating by, 178
wasted effort, minimal-specification approach and avoidance of, 326
wasted time, during fuzzy front end, 44
waterfall lifecycle model, 136-39
 modified, 143-47
 with risk reduction, 146
 sashimi model, 143
 with subprojects, 145
 strengths and weaknesses of, 156
Waterman, Robert H., Jr., 247, 263, 271, 272, 317
Waters, Richard, 366, 367
Weinberg, Gerald M., 12, 40-41, 58, 66, 77, 217, 218, 221, 255, 271, 272, 276, 296, 388, 600, 601, 608
Weiss, David M., 423, 469
"What Are the Realities of Software Productivity/ Quality Improvements" (Glass), 369
"When the Rubber Hits the Road: A Guide to Implementing Self-Managing Teams" (Thomsett), 317
Whitaker, Ken, 247, 267, 554
Why Does Software Cost So Much? (DeMarco), 231, 317
"Why Is Technology Transfer So Hard?" (Jones), 370
Wicked Problems, Righteous Solutions (DeGrace and Stahl), 161
Wiener, Lauren Ruth, 321
Windows NT 3.0, development of, 388, 408, 542-44
win-win projects, 239
WinWord (Microsoft Word for Windows) 1.0
 optimistic scheduling practices arid, 207-9
Wirth, Niklaus, 319
Wisdom of Teams, Tie (Katzenbach and-Smith), 275

wishful thinking, 43
schedules and, 221
Witness (film), 273, 296
Wood, Jane, 463
Word for Windows 1.0, optimistic scheduling
practices and, 207-9
work environments. *See also* productivity environments
noisy, crowded, 41
Work Redesign (Hackman and Oldham), 272
Wosser, 531, 534
Writing Solid Code (Maguire), 68

Y

Yourdon, Edward, 66-67, 73, 337, 423

Z

Zachary, Pascal, 23, 271, 272, 388, 408, 412, 413,
542, 544
Zawacki, Robert, xviii, 259, 351
Zelkowitz, 351

About the Author

Steve McConnell *is* chief software engineer at Construx Software Builders, Inc., a Seattle-area software-construction firm. He is the author of *Code Complete*, editor of *IEEE Software*'s "Best Practices" column, and an active developer. His primary focus has been on the development of mass-distribution shrink-wrap software, which has led him to consulting engagements with many companies in the Puget Sound area, including Microsoft Corporation.

Steve earned a Bachelor's degree from Whitman College in Walla Walla, Washington, and a master's degree in software engineering from Seattle University. He is a member of the IEEE Computer Society and the ACM.

Steve lives in Bellevue, Washington, with his English Springer Spaniel, Odie; Bearded Collie, Daisy; and elementary-school-teacher wife, Tammy.

If you have any comments or questions about this book, please contact Steve care of Microsoft Press, on the Internet at stevemcc@construx.com, or at his website at <http://www.construx.com/stevemcc>.

