SOFTWARE DEVELOPMENT PEARLS

Lessons from Fifty Years of Software Experience

KARL WIEGERS

Foreword by Steve McConnell

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for *Software Development Pearls*

"This is a collection of lessons that Karl has learned over his long and, I can say this honestly, distinguished career. It is a retrospective of all the good things (and some of the bad) he picked up along the way. However, this is not a recollection of 'It was like this in my day' aphorisms, but lessons that are relevant and will benefit anybody involved, even tangentially, in software development today. The book is surprising. It is not simply a list of pearls of wisdom—each lesson is carefully argued and explained. Each one carries an explanation of why it is important to you, and importantly, how you might bring the lesson to your reality."

—**James Robertson,** *Author of* Mastering the Requirements Process

"Wouldn't it be great to gain a lifetime's experience early in your career, when it's most useful, without having to pay for the inevitable errors of your own experience? Much of Karl Wiegers's half-century in software and management has been as a consultant, where he's often been called upon to rectify debacles of other people's making. In *Software Development Pearls,* Karl lays out the most common and egregious types of maladies that he's run into. It's valuable to know where the most expensive potholes are and which potholes people keep hitting time and time again.

"Not just a disaster correspondent, Karl is well versed in the best techniques of business analysis, software engineering, and project management. So from Karl's experience and knowledge you'll gain concise but important insights into how to recover from setbacks as well as how to avoid them in the first place.

"Forty-six years ago I was lucky enough to stumble onto Fred Brooks's classic *The Mythical Man-Month*, which gave me tremendous insights into my new career. Karl's book is in a similar vein, but broader in scope and more relevant for today's world. My own half-century of experience confirms that he's right on the money with the lessons that he's chosen for *Software Development Pearls*."

—**Meilir Page-Jones,** *Senior Business Analyst, Wayland Systems Inc.*

"Karl has created yet another wonderful book full of well-rounded advice for software developers. His wisdom will be relatable to all development professionals and students—young and old, new and experienced. Although I've been doing software development for many years, this book brought timely reminders of things my team should do better. I cannot wait to have our new-to-the-job team members read this.

"*Software Development Pearls* is rooted in actual experiences from many years of real projects, with a dose of thorough research to back up the lessons. As with all of Karl's books, he keeps it light and engaging, chock-full of relatable stories and a few funny comments. You can read it from front to back or just dive into a particular section that's relevant to the areas you're looking to improve today. An enjoyable read plus practical advice—you can't go wrong!"

—**Joy Beatty,** *Vice President at Seilevel*

"Karl's *Software Development Pearls* achieves the challenging goal of capturing and explaining many insights that you're unlikely to be exposed to in your training, that most practitioners learn through the school of hard knocks, and yet are critical to developing great software.

"While the book's structure compels you to connect with your experience and identify how to shift your behavior as a result, it's the content that shines: a collection of 59+1 lessons that cover the broad landscape of the software development ecosystem. These insights will help you save time, collaborate more effectively, build better systems, and change your view on common misconceptions. *Software Development Pearls* is an easy read and is backed by a wide range of references to other experts who have discovered these same insights in their travels.

"These lessons truly are Pearls: timelessly valuable elements of wisdom to make you better at developing great software, regardless of your role. Consider getting two copies of the book: one for yourself, and one to leave where others on the team can pick it up and discover their own pearls."

—**Jim Brosseau,** *Clarrus*

"This is an excellent book for anyone involved in software development. One of the brilliant (and unusual) aspects of the book is the way it is organized into self-contained lessons. Once you read them, they work like memes—memorable chunks of distilled knowledge that spring to mind when you need them. This happened to me recently when I was discussing the need for a requirements competency on agile projects with a senior executive and immediately thought of Lesson 8, 'The overarching objective of requirements development is clear and effective communication.'

"From personal experience, I can attest to the value of lessons like #22, 'Many system problems take place at interfaces,' but only because I was burned badly by not paying enough attention to them. Anyone in software development eventually accumulates hard-won lessons like these about what to do—and not do—in the future. This book will get you there with much less pain. As Karl says in Lesson 7, 'The cost of recording knowledge is small compared to the cost of acquiring knowledge.' Not only is that good advice for practitioners, it also neatly captures why you should buy this book."

—**Howard Podeswa,** *Author of* The Agile Guide to Business Analysis and Planning: From Strategic Plan to Continuous Value Delivery

Software Development Pearls

This page intentionally left blank

Software Development Pearls

Lessons from Fifty Years of Software Experience

Karl Wiegers

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021942545

Copyright © 2022 Karl E. Wiegers

Cover image: Philipp Tur/Shutterstock

Key icon: LDDesign/Shutterstock

Person reading book icon: VoodooDot/Shutterstock

Stairs icon: FOS_ICON/Shutterstock

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-748777-6
ISBN-10: 0-13-748777-0

ScoutAutomatedPrintCode

For Chris, as always

This page intentionally left blank

Contents

Foreword . . . xix

Acknowledgments . . . xxi

About the Author . . . xxiii

Chapter 1: Learning from Painful Experience . . . 1

- My Perspective . . . 1
- About the Book . . . 2
- A Note on Terminology . . . 4
- Your Opportunity . . . 5

Chapter 2: Lessons About Requirements . . . 7

- Introduction to Requirements . . . 7
  - Many Types of Requirements . . . 7
  - Subdomains of Requirements Engineering . . . 9
  - The Business Analyst Role . . . 10
  - Requirements Are Foundational . . . 11
- First Steps: Requirements . . . 11
- Lesson 1: Get the requirements right or the project will fail . . . 12
  - The Right Requirements—But When? . . . 13
  - The Right Requirements—But How? . . . 14
- Lesson 2: Requirements development delivers shared understanding . . . 15
- Lesson 3: Stakeholder interests intersect at the requirements . . . 17
  - Stakeholder Analysis . . . 18
  - Who Makes the Call? . . . 21
  - We're All on the Same Side Here . . . 21
- Lesson 4: Favor a usage-centric approach to requirements . . . 21
  - Why the Excess Functionality? . . . 22
  - Putting Usage First . . . 22
  - A User Story Concern . . . 24
  - Usage Rules! . . . 25

Lesson 5: Requirements development demands iteration 25
- Progressive Refinement of Detail 26
- Emergent Functional Requirements 27
- Emergent Nonfunctional Requirements 28

Lesson 6: Agile requirements aren't different from other requirements 28
- Roles and Responsibilities 29
- Terminology 29
- Documentation Detail 30
- Activity Timing 30
- Deliverable Forms 31
- When to Prioritize 32
- Is There Really a Difference? 32

Lesson 7: Recording knowledge is cheaper than acquiring it 33
- Fear of Writing 34
- Benefits of Written Communication 34
- A Sensible Balance 36

Lesson 8: Requirements are about clear communication 37
- Multiple Audiences, Multiple Needs 38
- Choosing Representation Techniques 40
- Can We Talk? 41

Lesson 9: Requirements quality is in the eye of the beholder 41
- Many Requirements Beholders 42
- Requirements Quality Checklist 42

Lesson 10: Requirements must be good enough to reduce risk 44
- Dimensions of Detail 45
- How Much Is Enough? 46

Lesson 11: People don't simply gather requirements 46
- Gathering versus Elicitation 47
- When to Elicit Requirements 48
- The Elicitation Context 48
- Elicitation Techniques 49
- Laying the Foundation 51

Lesson 12: Elicitation brings the customer's voice to the developer 51
- Communication Pathways 52
- Product Champions 53

Other Requirements Communication Pathways 53
Bridging the Gap 54
Lesson 13: Telepathy and clairvoyance don't work 55
Guess That Requirement! 55
Being Explicit 55
Telepathy Fails 57
Lesson 14: Large groups have difficulty agreeing on requirements 57
Pay Attention! 58
Facilitator to the Rescue 59
Focus, Focus, Focus 59
Reaching Outside the Group 60
Lesson 15: Avoid decibel prioritization 61
Prioritization Techniques 62
Prioritization Criteria 62
Analysis over Volume 64
Lesson 16: Define scope to know whether your scope is creeping . . . 64
The Specter of Scope Creep 64
How to Document Scope 65
Is It in Scope? 66
Fuzzy Requirements = Fuzzy Scope 67
Next Steps: Requirements 69

Chapter 3: Lessons About Design 71

Introduction to Design 71
Different Aspects of Design 72
Do You Have a Good Design? 74
First Steps: Design 75
Lesson 17: Design demands iteration 76
The Power of Prototypes 77
Proofs of Concept 78
Mock-ups 78
Lesson 18: It's cheaper to iterate at higher levels of abstraction 79
Stepping Back from the Details 81
Rapid Visual Iteration 81
Iteration Made Easy 83
Lesson 19: Make products easy to use correctly, hard to use incorrectly 84
Make It Impossible for the User to Make a Mistake 86

Make It Difficult for the User to Make a Mistake 86
Make It Easy to Recover from an Error 86
Just Let It Happen 87
Lesson 20: You can't optimize all desirable quality attributes 87
Dimensions of Quality 88
Specifying Quality Attributes 90
Designing for Quality 90
Architecture and Quality Attributes 92
Lesson 21: An ounce of design is worth a pound of recoding 92
Technical Debt and Refactoring 93
Architectural Shortcomings 94
Lesson 22: Many system problems take place at interfaces 94
Technical Interface Issues 96
Input Data Validation 98
User Interface Issues 99
Interface Wars 100
Next Steps: Design 100

Chapter 4: Lessons About Project Management 103

Introduction to Project Management 103
People Management 104
Requirements Management 104
Expectation Management 104
Task Management 105
Commitment Management 105
Risk Management 105
Communication Management 106
Change Management 106
Resource Management 106
Dependency Management 106
Contract Management 107
Supplier Management 107
Managing Away the Barriers 107
First Steps: Project Management 108
Lesson 23: Work plans must account for friction 109
Task Switching and Flow 109
Effective Hours 111

Other Sources of Project Friction 112
Planning Implications 113
Lesson 24: Don't give anyone an estimate off the top of your head 114
Hasty Predictions 114
Fear of Fuzz 115
Lesson 25: Icebergs are always larger than they first appear 116
Contingency Buffers 117
Risky Assumptions 119
Contracting on Icebergs 120
The Beauty of Buffers 121
Lesson 26: Data strengthens your negotiating position 121
Where Did You Get That Number? 122
Principled Negotiation 123
Lesson 27: Use historical data to improve estimates 124
Multiple Sources of Historical Data 125
Software Metrics 126
Lesson 28: Don't change an estimate just to make someone happy 127
Goals versus Estimates 128
When to Adjust 129
Lesson 29: Stay off the critical path 129
Critical Path Defined 129
Keeping Out of the Way 131
Lesson 30: Incomplete tasks get no partial credit 132
What Does "Done" Mean? 132
No Partial Credit 134
Tracking by Requirements Status 135
Doneness Leads to Value 136
Lesson 31: A project team needs flexibility to adapt to change ... 136
Five Project Dimensions 136
Negotiating Priorities 138
The Flexibility Diagram 138
Applying the Five Dimensions 140
Lesson 32: Uncontrolled project risks will control you 140
What Is Risk Management? 141
Identifying Software Risks 141

Risk Management Activities 143
There's Always Something to Worry About 145
Lesson 33: The customer is not always right 145
Being "Not Right" 146
Respecting the Point 148
Lesson 34: We do too much pretending in software 149
Living in Fantasyland 149
Irrational Exuberance 150
Games People Play 150
Next Steps: Project Management 151

Chapter 5: Lessons About Culture and Teamwork 153

Introduction to Culture and Teamwork 153
Keeping the Faith 154
Cultural Congruence 155
Crystallizing the Culture 156
Growing the Group 157
First Steps: Culture and Teamwork 158
Lesson 35: Knowledge is not zero-sum 159
The Knowledge Hog 160
Rectifying Ignorance 160
Scaling Up Knowledge Transfer 161
A Healthy Information Culture 163
Lesson 36: Don't make commitments you know you can't fulfill 163
Promises, Promises 164
Life Happens 165
Lesson 37: Higher productivity requires training and better practices 166
What's the Problem? 167
Some Possible Solutions 167
Tools and Training 169
Individual Developer Variation 169
Lesson 38: The flip side of every right is a responsibility 171
Some Customer Rights and Responsibilities 172
Some Developer Rights and Responsibilities 172
Some Project Manager or Sponsor Rights and Responsibilities 172

Some Autonomous Team Rights and Responsibilities 173
Concerns before Crises 173
Lesson 39: Surprisingly little separation can inhibit communication 173
Barriers of Space and Time 174
Virtual Teams: The Ultimate in Separation 175
A Door, a Door, My Kingdom for a Door! 176
Lesson 40: Small-team approaches don't scale to large projects 177
Processes and Tools 178
The Need for Specialization 179
Communication Clashes 180
Lesson 41: Address culture change during a change initiative 180
Values, Behaviors, and Practices 181
Agile and Culture Change 182
Internalization 184
Lesson 42: Engineering techniques don't work with unreasonable people 185
Try a Little Teaching 186
Who's Out of Line Here? 186
In Favor of Flexibility 187
Next Steps: Culture and Teamwork 187

Chapter 6: Lessons About Quality 189
Introduction to Quality 189
Definitions of Quality 189
Planning for Quality 190
Multiple Views of Quality 192
Building Quality In 192
First Steps: Quality 194
Lesson 43: Pay for quality now or pay more later 195
The Cost-of-Repair Growth Curve 195
Harder to Find 197
Early Quality Actions 198
Lesson 44: High quality naturally leads to higher productivity . . . 200
A Tale of Two Projects 200
The Scourge of Rework 202
The Cost of Quality 203

Lesson 45: Organizations somehow find time to fix bad software . . . 205
Why Not the First Time? . . . 205
The $100 Million Syndrome . . . 206
Striking the Balance . . . 207
Lesson 46: Beware the crap gap . . . 207
The Crap Gap Illustrated . . . 207
Crap-Gap Scenarios in Software . . . 208
Lesson 47: Never let anyone talk you into doing a bad job . . . 209
Power Plays . . . 210
Rushing to Code . . . 211
Lack of Knowledge . . . 211
Shady Ethics . . . 212
Circumventing Processes . . . 212
Lesson 48: Strive to have peers find defects . . . 213
Benefits of Peer Reviews . . . 214
Varieties of Software Reviews . . . 215
The Soft Side: Cultural Implications of Reviews . . . 216
Lesson 49: A fool with a tool is an amplified fool . . . 217
A Tool Must Add Value . . . 218
A Tool Must Be Used Sensibly . . . 219
A Tool Is Not a Process . . . 219
Lesson 50: Rushed development leads to maintenance nightmares . . . 221
Technical Debt and Preventive Maintenance . . . 221
Conscious Technical Debt . . . 222
Designing for Quality, Now or Later . . . 223
Next Steps: Quality . . . 224

Chapter 7: Lessons About Process Improvement . . . 225
Introduction to Process Improvement . . . 225
Software Process Improvement: What and Why . . . 225
Don't Fear the Process . . . 226
Making SPI Stick . . . 227
First Steps: Software Process Improvement . . . 228

Lesson 51: Watch out for "Management by Businessweek" 229
First Problem, Then Solution 230
A Root Cause Example 230
Diagnosis Leads to Cure 232
Lesson 52: Ask not, "What's in it for me?" Ask,
"What's in it for us?" 233
The Team Payoff 233
The Personal Payoff 235
Take One for the Team 235
Lesson 53: The best motivation for changing how people
work is pain 236
Pain Hurts! 236
Invisible Pain 237
Lesson 54: Steer change with gentle pressure,
relentlessly applied 238
Steering 239
Managing Upward 240
Lesson 55: Don't make all the mistakes other people
already have 241
The Learning Curve 242
Good Practices 243
Lesson 56: Good judgment and experience can trump
a process 244
Processes and Rhythms 245
Being Nondogmatic 246
Lesson 57: Shrink templates to fit your project 247
Lesson 58: Learn and improve so the next project goes better 252
Looking Back 252
Retrospective Structure 254
After the Retrospective 255
Lesson 59: Don't do ineffective things repeatedly 256
The Merits of Learning 257
The Merits of Thinking 258
Next Steps: Software Process Improvement 259

Chapter 8: What to Do Next **261**
Lesson 60: You can't change everything at once 262
Prioritizing Changes 264
Reality Check 265
Action Planning 266
Your Own Lessons 267

Appendix: Summary of Lessons **269**

References **273**

Index **285**

Foreword

After Karl Wiegers received his PhD in organic chemistry, he took a job as a research scientist at Kodak in Rochester, New York. Karl had interviewed at Kodak before he accepted the job, and he thought he understood the nature of the work. He would be doing research related to photographic film, photo development, and related projects.

When Karl arrived at Kodak, he was escorted through a light lock, into the laboratory. A light lock is like an air lock in a submarine, except that it ensures that no light leaks into a room that is kept completely dark. After Karl went through the light lock, his eyes took several minutes to adjust to the barely lit laboratory. No one had told Karl that his research lab would be a photographic dark room.

Karl quickly realized he did not want to spend his career literally working in the dark, and so he transitioned to roles of software developer, then software manager, and eventually software process and quality improvement leader. Later, he founded his own company, Process Impact.

This practical book is Karl's attempt to lead others out of software darkness and into the light. As with his other books, there is more pragmatism than theory. The book concentrates on the areas in which Karl has direct experience: especially requirements, process improvement, quality, culture, and teamwork.

Karl doesn't explain why he titled the book *Software Development Pearls*. The process of growing a pearl begins when an irritant such as a grain of sand gets trapped in an oyster. In response, the oyster gradually accretes a substance to protect itself from the irritant. It takes a long time, but eventually the irritant results in a valuable pearl.

Karl is one of the most thoughtful software people I know. He has reflected deeply on the software development irritants he has encountered over his career, and this book contains 60 of his most valuable responses.

—*Steve McConnell, Construx Software and author of* Code Complete

This page intentionally left blank

Acknowledgments

Over a span of more than fifty years, I've learned how to do software development, project management, and process improvement from many sources. I've read countless books and articles, taken many professional training courses, and attended a multitude of conference presentations. I'm grateful to all the educators who passed along anything from one helpful nugget of knowledge to a whole new understanding of some part of our discipline. Two great trainers particularly stand out: Steve Bodenheimer and Dr. Joyce Statz. How many names of your professional instructors stick in your mind decades later?

The vast body of software engineering literature is a virtually endless source of enlightenment. Authors whose work I've found especially illuminating include Mike Cohn, Larry Constantine, Alan Davis, Tom DeMarco, Tom Gilb, Robert Glass, Ellen Gottesdiener, Capers Jones, Norm Kerth, Tim Lister, Steve McConnell, Roxanne Miller, James Robertson, Suzanne Robertson, Johanna Rothman, and Ed Yourdon. If you haven't read their work, you should. It's been a rare privilege to become friends with so many wise authors and consultants over the years.

I've had the good fortune to work with some talented software engineers. You can learn a lot from seeing how others work, because we all have complementary experiences. As the Principal Consultant at my company, Process Impact, I've provided about 150 companies and government agencies with training and consulting services. I appreciate all of the clients and students in my training courses who shared their horror—and success—stories with me. Those reports helped me learn what techniques do and don't work well in a multitude of real-world situations. I've distilled all that I learned from these numerous sources into the lessons in this book.

In preparing this book, I had valuable discussions with Jim Brosseau, Tanya Charbury, Mike Cohn, David Hickerson, Tony Higgins, Norm Kerth, Ramsay Miller, Howard Podeswa, Holly Lee Sefton, and particularly Meilir Page-Jones, Ken Pugh, and Kathy Reynolds. I thank them sincerely for their patience with my questions and the experience stories they shared. Thanks also to those people who provided insightful quotations to augment my personal observations.

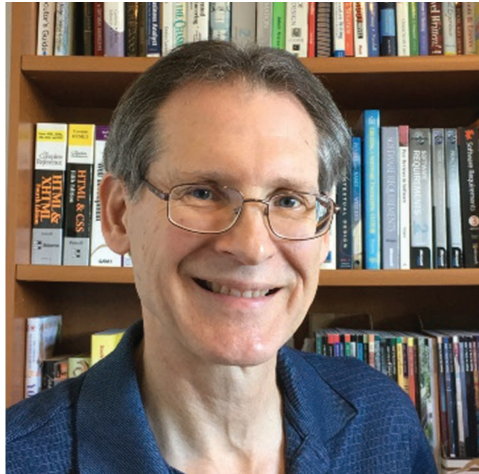
I appreciate the helpful manuscript review input provided by Joy Beatty, Jim Brosseau, Mike Cohn, Gary K. Evans, Lonnie Franks, David Hickerson, Kathy Iberle, Norm Kerth, Darryl Logsdon, Jeannine McConnell, Marco Negri, Meilir Page-Jones, Neil Potter, Ken Pugh, Gina Schmidt, James Shields, John Siegrist, Jeneil

Stephen, Tom Tomasovic, and Sebastian Watzinger. Review comments from Tanya Charbury, Kathy Reynolds, Maud Schlich, and Holly Lee Sefton were particularly valuable. Thanks also to Gary K. Evans for permission to modify a useful figure on design interfaces.

I'm grateful to Haze Humbert, Menka Mehta, and the entire editorial and production team at Pearson for their fine work on my manuscript.

As always, I'm indebted to my wife, Chris, for patiently tolerating another book project.

About the Author



Since 1997, Karl Wiegers has been Principal Consultant with Process Impact, a software development consulting and training company in Happy Valley, Oregon. Previously, he spent eighteen years at Kodak, where he held positions as a photographic research scientist, software developer, software manager, and software process and quality improvement leader. Karl received a PhD in organic chemistry from the University of Illinois.

Karl is the author of twelve previous books, including *The Thoughtless Design of Everyday Things, Software Requirements, More About Software Requirements, Practical Project Initiation, Peer Reviews in Software, Successful Business Analysis Consulting,* and a forensic mystery novel titled *The Reconstruction*. He has written many articles on software development, management, design, consulting, chemistry, and military history. Several of Karl's books have won awards, most recently the Society for Technical Communication's Award of Excellence for *Software Requirements, 3rd Edition* (co-authored with Joy Beatty). Karl has served on the Editorial Board for *IEEE Software* magazine and as a contributing editor for *Software Development* magazine.

When he's not at the keyboard, Karl enjoys wine tasting, volunteering at the public library, delivering Meals on Wheels, playing guitar, writing and recording songs, reading military history, and traveling. You can reach him through www.processimpact.com or www.karlwiegers.com.

This page intentionally left blank

Chapter 1

Learning from Painful Experience

I've never known anyone who could truthfully say, "I am building software today as well as software could ever be built." Anyone who can't say that would benefit from learning better ways to work. This book offers some shortcuts for that quest.

Experience is the form of learning that sticks with us the best. It's also the most painful way to learn. Our initial attempts to try new approaches often stumble and sometimes fail. We all must climb learning curves, accepting short-term productivity hits as we struggle to master new methods and understand when and how to use them adeptly.

Fortunately, an alternative learning mechanism is available. We can compress our learning curves by absorbing lessons, tips, and tricks from people who have already acquired and applied the knowledge. This book is a collection of such pearls of wisdom about software engineering and project management—useful insights I've gained through my personal experiences and observed from other people's work. Your own experiences and lessons might differ, and you might not agree with everything I present. That's fine; everyone's experience is unique. These are all things I've found valuable in my software career, though.

My Perspective

Let me start with a bit of my background to show how I accumulated these lessons. I took my first computer programming class in college in 1970—FORTRAN, of course. My very first job—the next summer—involved automating some operations of the financial aid office at my college, all on my own. I'd had two credits of programming, so I was a software engineer, right? The project was surprisingly

successful, considering my limited background. I took two more programming courses in college. Everything else I've learned about software engineering I've picked up on my own from reading, training courses, experience, and colleagues. That unofficial career path wasn't unusual some time ago, as people were drawn to software development from many backgrounds but had little formal education in computing.

Since that early start, I spent a lot of time doing a diverse range of software work: requirements development, application design, user interface design, programming, testing, project management, writing documentation, quality engineering, and process improvement leadership. I took some side trips along the way, like getting a PhD in organic chemistry. Even then, one-third of my doctoral thesis consisted of software code for analyzing experimental data and simulating chemical reactions.

Early in my career as a research scientist at Eastman Kodak Company, then a huge and highly successful corporation, I used computers to design and analyze experiments. I soon transitioned into full-time software development, building applications for use in the Kodak Research Laboratories and managing a small software group for a few years. I found that my scientific background and inclination guided me to take a more systematic approach to software development than I might have otherwise.

I wrote my first article about software in 1983. Since then, I've written many articles and eight books on numerous aspects of the discipline. As an independent consultant and trainer since 1997, I have provided services to nearly 150 companies and government agencies in many business domains. These interactions let me observe techniques that work effectively on software projects—and techniques that don't.

Many of my insights about software development and management came from my personal project experiences, some rewarding, but also some disappointing. I gained other knowledge from my consulting clients' experiences, generally on projects that did not go well. No one calls a consultant when everything is going swimmingly. I wrote this book so that you don't need to accumulate all those same lessons slowly and painfully through your personal project struggles. One highly experienced software engineer who read this list of lessons commented, "Every one of those items has a scar (or several) associated with it."

About the Book

This book presents fifty-nine lessons about software development and management, grouped into six domains, with one chapter on each domain:

Chapter 2. Requirements

Chapter 3. Design

Chapter 4. Project management

Chapter 5. Culture and teamwork

Chapter 6. Quality

Chapter 7. Process improvement

Chapter 8 provides one final, general lesson to keep in mind as you move forward. For easy reference, all sixty lessons are collected in the Appendix.

I haven't attempted to provide a complete list of lessons in those domains. There's so much knowledge in each category that no one could create an exhaustive compilation. Nor do I address other essential aspects of software development, most obviously programming, testing, and configuration management. Other authors have compiled comprehensive wisdom in those areas in books like these:

- *Programming Pearls* by Jon Bentley (2000)
- *Lessons Learned in Software Testing* by Cem Kaner, James Bach, and Bret Pettichord (2002)
- *Code Complete* by Steve McConnell (2004)
- *Software Engineering at Google* by Titus Winters, Tom Manshreck, and Hyrum Wright (2020)

The topics and lessons in this book are largely independent, so you can read them in any order you like without any loss of continuity. Each chapter begins with a brief overview of the pertinent software domain. Then several First Steps encourage you to reflect on your previous experiences with that domain before you dive into the chapter's lessons. The First Steps invite you to think about problems your teams have experienced in that area, the impacts of those problems, and possible contributing root causes.

Each lesson concisely states a core insight, followed by a discussion and suggested practices that teams can adopt based on the lesson. As you read through each chapter, think about how those practices might relate to your situation. A book icon in the margin, as shown here, indicates a true story drawn from my personal experiences, interactions with my consulting clients, or experiences that colleagues have shared with me. All the stories are real, though names have been changed to preserve privacy. In addition to the true-story icons, key points in each lesson description are flagged with a key icon in the margin, like the one shown here. Some of the lessons contain cross-references to other lessons, which are indicated with a margin icon like the one you see here.

The Next Steps section at the end of each chapter will help you plan how to apply the chapter's material to your project, team, or organization. No matter what sort of project you work on, what life cycle it follows, or what kind of product you build, look for the idea behind each lesson and see how you might adapt it to help your project be more successful.

Consider going through the First Steps and Next Steps with a group of your colleagues rather than doing them alone. At the beginning of the hundreds of training courses I've taught, I have small groups discuss problems their teams have experienced related to the course topic (the First Steps). At the end of the course, the same groups explore solutions to those problems, brainstorming ways to apply the course contents right away (the Next Steps). My students find it valuable to include a variety of stakeholders in those discussion groups. Different stakeholders bring diverse perspectives on how certain aspects of the project are going. Combining their perspectives provides a rich understanding of their current practices and a creative opportunity to choose practical solutions.

I hope many of my lessons resonate with you and motivate you to try something different on your projects. However, you can't change everything you do at once. Individuals, teams, and organizations can absorb change only at a certain rate as they strive to get their project work done concurrently. The final chapter in the book, "What to Do Next," will help you chart a path to translate the lessons into actions. That chapter offers suggestions about prioritizing the changes you want to put into motion and crafting an action plan to help you get from where you are today to where you want to be.

A Note on Terminology

I use the terms *system, product, solution,* and *application* more or less interchangeably in this book. In each instance, I'm simply referring to whatever ultimate deliverable your project creates, so please don't read any significance into whichever term I use in a particular place. Whether you work on corporate or government information systems, websites, commercial software applications, or hardware devices with embedded software, the lessons and their associated practices will be broadly applicable.

Your Opportunity

Unless you are indeed among that rare class of practitioners who are already building software as well as software could ever be built, you have some improvement opportunities. We all need to continuously enhance our capabilities: as individual practitioners, as project teams, and as organizations. We all want fewer scars.

A junior developer named Zachary Minott (2020) made some thoughtful observations about how he outperformed more experienced developers. Minott described an ethic of acknowledging what he didn't know, systematically going about learning, and putting the new knowledge into practice. He said, "If there is any superpower that I do have, it's the ability to learn fast and immediately apply what I learn to what I'm doing." Minott discovered the critical mechanism for continuously wending his way toward mastery of his discipline.

We all need to continuously enhance our capabilities. We all want fewer scars.

Perhaps you decide to take a class to learn a new skill or enhance your current way of working. While you take the class, the work continues to pile up. It's easy to ignore what you've learned and continue to work as you always have in the rush to get caught up. That's comfortable, as your current approach has sufficed so far. But that's not the way to improve.

I adopted the approach of identifying two areas on each project to get better at. I would set aside some time to learn about those topics and try to apply my new understanding. Not every technique worked out, but my approach allowed me to gradually accumulate skills that have served me well.

I encourage you to do the same. Don't merely read the book; take the next step. Decide how you and your colleagues can apply the practices you read about and what you hope they'll do for you. Build a list of practices that you want to learn more about and then put them into use. That way, you'll come out ahead in the long run.

Lesson 1	If you don't get the requirements right, it doesn't matter how well you execute the rest of the project.

A business analyst at one of my consulting clients related an unfortunate project experience. Their IT department was building a replacement information system for use within their company. The development team believed that they already understood the system's requirements without obtaining any additional user input. They weren't arrogant, just confident. However, when the developers presented the completed system to the users, their reaction was, "But seriously, folks, where's our application?" The users rejected the system as completely unacceptable.

The development team was shocked; they thought they were working in good faith to build the right product. However, neglecting to engage with the users to ensure that the development team understood the requirements was a serious oversight.

When you proudly present your new baby to the world, you do not want to be told, "Your baby is ugly." But that's what happened in this case. So, what did the company do? They rebuilt the system, this time with adequate user input. (See Lesson 45, "Organizations never have time to build software right, yet they find the resources to fix it later.") That was an expensive lesson in the importance of customer involvement in getting the requirements right.

Whether you're building a new product or enhancing an existing one, requirements are the basis for all subsequent project work. Design, construction, testing, documentation, training, and migration from one system or operating environment to another all depend on having the right requirements. Numerous studies have found that effectively developing and communicating requirements are critical success factors for any project. Conversely, common contributors to troubled projects include inadequate project vision, incomplete and inaccurate requirements, and changing requirements and project objectives (PMI 2017). Getting the requirements right is core to ensuring that the solution aligns with the developing organization's product vision and business strategy (Stretton 2018). If you don't get the requirements right, you will fail.

Without high-quality requirements, stakeholders can be surprised at what the development team delivers. Software surprises are usually bad news.

The Right Requirements—But When?

I'm not saying that you need a complete set of requirements before you commence implementation. That's not realistic for any but the smallest and most stable prod-

ucts. New ideas, changes, and corrections will always come along that you must fold into your development plans. But for any portion of the system that you're building—whether it's a single development iteration, a specific release, or the full product—you need to have the requirements as nearly correct as possible. Otherwise, plan on performing rework after you think you're done. Agile projects use development iterations to validate the requirements that fed into the iteration. The further away those initial requirements are from what customers actually need, the more rework that will be needed.

Some people claim that you'll never get the requirements right. They say customers always think of more to add, there are always worthwhile changes to make, and the environment evolves continuously. That may be true, but I counter with, "In that case, you might never finish the project." From the perspective that there's always something you could add, you might never nail the requirements perfectly. But for the agreed-upon scope of a given development portion, you've got to get them right, or success will elude you.

The situation is a little different if you're building a highly innovative product. If no one has ever made anything like it before, you're unlikely to get it right on the first try. Your first attempt is essentially a plan to test hypotheses and determine the requirements through experimentation. Ultimately, though, your explorations will lead to an understanding of your novel product's capabilities and characteristics—its requirements.

The Right Requirements—But How?

There's no substitute for ongoing customer engagement to develop a set of accurate, clear, and timely requirements. (See Lesson 12, "Requirements elicitation must bring the customer's voice close to the developer's ear.") You can't just hold a workshop early on and then tell the customers, "We'll call you when we're done." Ideally, the team will have customer representatives available throughout the project. The team will have many questions to ask and points that require clarification. They'll need to elaborate high-level requirements from early explorations into appropriate detail at the right time. The team needs frequent feedback from users and other stakeholders to validate their understanding of requirements and the solutions they conceive.

It can be challenging to get customers to sign up for this extensive level of participation. They have their own work to do; their managers might not want some of their best people to spend a lot of time on the project. "You can go to a workshop or two," the manager might say, "but I don't want those software people interrupting you all the time with questions."

One way to sell the case for ongoing engagement is to point out problems the organization has experienced because of inadequate customer participation. Even better, cite local experiences where customer engagement paid off. Another persuasion technique is to propose a structured framework for the requirements engagement instead of leaving it completely open-ended. This framework might include some combination of informal discussions, elicitation workshops, requirements reviews, and working with screen sketches, prototypes, and incremental releases.

Customers are more likely to be excited about the project and willing to contribute if they see signs of tangible progress, such as through periodic releases of working software. They'll also be more enthusiastic if they see that their input truly influences the project's direction. It's sometimes a struggle to persuade users to accept new and replacement software systems. User representatives who worked with the IT team and understand the new system and its rationale can greatly smooth the transition.

I've worked with several customer representatives who had an outsize impact on the project's success. Besides providing input on requirements, some of them also provided user interface sketches and tests to verify that portions of the software were implemented properly. I can't overstate the contribution such committed customers made to help the development team get the requirements right and deliver the right solution.

Without high-quality requirements, stakeholders can be surprised at what the development team delivers. In my experience, software surprises are usually bad news. When they see the product, the reaction I want from my customers is, "Wow, Karl, this is better than I ever imagined. Thank you!" That's the kind of software surprise we can all live with.

Lesson 17 Design demands iteration.

In his classic book *The Mythical Man-Month,* Frederick P. Brooks, Jr. (1995) advises, "Plan to throw one away; you will, anyhow." Brooks is referring to the idea that, on large projects, it's advisable to create a pilot or preproduction system to figure out how best to build the complete system. That's an expensive prospect, particularly if the system includes hardware components. However, a pilot system is valuable if you have technical feasibility questions or if a suitable design strategy isn't clear initially.

A pilot system also reveals the unknown unknowns, factors you hadn't yet realized were significant.

While you're unlikely to build and then discard a preliminary version of most products, you do need to iterate on potential designs before the team gets very far into construction. Creating the simplest possible design sounds attractive, and it does accelerate solution delivery. Rapid delivery might meet a customer's short-term perception of value, but it may not be the best long-term strategy as the product grows over time.

There's always more than one design solution for a software problem and seldom a single best solution (Glass 2003). The first design approach you conceive won't be the best option. Norman Kerth, a highly experienced designer of software, furniture, and other items, explained it to me nicely:

> You haven't done your design job if you haven't thought of at least three solutions, discarded all of them because they weren't good enough, and then combined the best parts of all of them into a superior fourth solution. Sometimes, after considering three options, you realize that you don't really understand the problem. After reflection, you might discover a simple solution when you generalize the problem.

Software design isn't a linear, orderly, systematic, or predictable process. Top designers often focus first on the hard parts where a solution might not be obvious or perhaps even feasible (Glass 2003). Several methods facilitate iteration as a designer moves from an initial concept to an effective solution. One method is to create and refine graphical models—diagrams—of the proposed designs. This technique is addressed in Lesson 18, "It's cheaper to iterate at higher levels of abstraction." Prototyping is another valuable technique for iterating on both technical and UX designs.

The Power of Prototypes

A prototype is a partial, preliminary, or possible solution. You build a piece of the system as an experiment, testing the hypothesis that you understand how to design the system well. If the experiment fails, you redesign it and try again. A prototype is valuable for assessing and reducing risk, particularly if you're employing a novel architectural or design pattern that you want to validate before committing to it.

If you intend a prototype to grow into the product, you must build it with production-level quality from the beginning.

Before you construct a prototype, determine whether you intend to discard it and then develop the real thing, or grow the preliminary solution into the product. A key point is that if you intend a prototype to grow into the product, you must build it with production-level quality from the beginning. That takes more effort than building something temporary that you'll discard after it has served its purpose. The more work you put into a prototype, the more reluctant you become to change it significantly or throw it away, which impedes the iteration mindset. Your prototyping approach should encourage cyclical refinement and even starting over if necessary.

Agile teams sometimes create stories called *spikes* to research technical approaches, resolve uncertainty, and reduce risk before committing to a specific solution (Leffingwell 2011). Unlike other user stories, a spike's prime deliverable is not working code, but rather knowledge. Spikes could involve technical prototypes, UI prototypes, or both, depending on the information sought. A spike should have a clear goal, just like a scientific experiment. The developer has a hypothesis to test. The spike should be designed to provide evidence for or against the hypothesis, test and confirm the validity of some approach, or allow the team to make an informed technical decision quickly.

Proofs of Concept

Proof-of-concept prototypes, also called *vertical prototypes,* are valuable for validating a proposed architecture. I once worked on a project that envisioned an unconventional client–server approach. The architecture made sense in our computing environment, but we wanted to make sure we weren't painting ourselves into a technical corner. We built a proof-of-concept prototype with a vertical slice of functionality from the UI through the communication layers and the computational engine. It worked, so we felt confident this design was workable.

Experimenting on a proof-of-concept prototype is a way to iterate at a relatively low cost, although you do need to build some executable software. Such prototypes are valuable for assessing the proposed design's technical aspects: architecture, algorithms, database structure, system interfaces, and communications. You can evaluate architectures against their needed properties—such as performance, security, safety, and reliability—and then refine them progressively.

Mock-ups

User interface designs always require iteration. Even if you're following established UI conventions, you should perform at least informal usability testing to choose appropriate controls and layouts to meet your ease-of-learning, ease-of-use, and accessibility goals. For instance, A/B testing is an approach in which you present

users with two UI alternatives for a given operation so that they can choose which one makes the most sense to them. The people who conduct an A/B test can observe user behaviors with the different approaches to determine which option is more intuitive or leads to more successful outcomes. It's simpler, faster, and cheaper to conduct such experiments while you're still exploring the design than to react to post-delivery customer complaints or lower-than-expected click-through rates on a web page.

As with requirements, UX designs benefit from the progressive refinement of detail through prototyping. You can create *mock-ups,* also called *horizontal prototypes* because they consist of just a thin layer of user interface with no functional substance below it. Mock-ups range from basic screen sketches to executable interfaces that look authentic but don't do real work (Coleman and Goodwin 2017). Even simple paper prototypes are valuable and are quick to create and modify. You can use a word processing document or even index cards to lay out the data elements in boxes representing potential screens, see how the elements relate to each other, and note which elements are user input and which are displayed results. Watch out for these traps with user interface prototyping.

- Spending too much time perfecting the UI's cosmetics ("How about a darker red for this text?") before you've mastered the screen flow and functional layouts. Get the broad strokes right first.
- Customers or managers thinking the software must be nearly done because the UI looks good, even if there's nothing behind it but simulated functions. A less polished prototype shows that it isn't yet finished.
- Coaching prototype evaluators as they attempt to perform a task that isn't obvious to them. You can't judge usability if you're helping the users learn and use the prototype.

If you don't invest in repeatedly exploring both user experience and technical designs before committing to them, you risk delivering products that customers don't like. Thoughtlessly designed products annoy customers, waste their time, erode their goodwill toward your product and company, and generate bad reviews (Wiegers 2021). A few more iteration cycles will get you much closer to useful and enjoyable designs.

Lesson 23 Work plans must account for friction.

I overheard this conversation at work one day:

Manager Shannon: "Jamie, I know you're doing the usability assessments on the Canary project right now. Several other projects are also interested in usability assessments. How much time do you spend on that?"

Team Member Jamie: "About eight hours a week."

Manager Shannon: "Okay, so you could work with five projects at a time then."

Do you see any flaws in Shannon's thinking? Five times eight is forty, the nominal hours in a work week, so this discussion seems reasonable on the surface. But Shannon hasn't considered the many factors that reduce the time that individuals have available each day for project work: project friction (as opposed to interpersonal friction, which I'm not discussing here).

There's a difference between elapsed hours on the job and effective available hours. This difference is just one factor that both project planners and individual team members must keep in mind as they translate size or effort estimates into calendar time. If people don't incorporate these friction factors into their planning, they'll forever underestimate how long it will take to get work done.

Task Switching and Flow

People do not multitask—they task switch. When multitasking computers switch from one job to another, there's a period of unproductive time during the switch. The same is true of people, only it's far worse. It takes a little while to gather all the materials you need to work on a different activity, access the right files, and reload your brain with the pertinent information. You need to change your mental context to focus on the new problem and remember where you were the last time you worked on it. That's the slow part.

Some people are better at task switching than others. Maybe I have a short attention span, but I'm pretty good at diverting my focus to something different and then resuming the original activity right where I left off. For many people, though, excessive task switching destroys productivity. Programmers are particularly susceptible to the time-sucking impact of multitasking, as Joel Spolsky (2001) explains:

> When you manage programmers, specifically, task switches take a really, really, really long time. That's because programming is the kind of task where you have to keep a lot of things in your head at once. The more things you remember at once, the more productive you are at programming. A programmer coding at full throttle is keeping zillions of things in their head at once.

People do not multitask—they task switch.

When I was a manager, a developer named Jordan said he was flailing. Jordan didn't understand the priorities of items in his work backlog. He would work on task A for a while, then feel guilty that he was neglecting task B, so he'd switch to that one. He was getting little done as a result. Jordan and I worked out his priorities and a plan for allocating time to tasks in turn. He stopped flailing, his productivity went up, and Jordan felt much better about his progress. Jordan's task-switching overhead and priority confusion affected both his productivity and his state of mind.

When you're deeply immersed in some work, focused on the activity and free from distractions, you enter a mental state called *flow*. Creative knowledge work like software development (or writing a book) requires flow to be productive (DeMarco and Lister 2013). You understand what you're working on, the information you need is in your working memory, and you know where you're headed. You can tell you've been in a state of flow when you lose track of time as you're making great progress and having fun. Then your phone pings with a text message, an e-mail notification pops up, your computer reminds you that a meeting starts in five minutes, or someone stops by to talk. Boom—there goes your flow.

Interruptions are flow killers. It takes several minutes to get your brain back into that highly productive state and pick up where you were before the interruption. Some reports suggest that interruptions and task switching can impose a penalty of at least 15 percent of a knowledge worker's time, amounting to more than one hour per day (DeMarco 2001). A realistic measure of your effective work capacity is based not on how many hours you're at work or even how many hours you're on task, but how many *uninterrupted* hours you're on task (DeMarco and Lister 2013).

To achieve the high productivity and satisfaction that come from an extended state of flow, you need to actively manage your work time. The potential for distractions and interruptions is ever-present unless you take steps to block them out. Jory MacKay (2021) offers several recommendations for reducing context switching and its accompanying productivity destruction.

- **Timeblock your schedule to create clearer focus boundaries.** Planning how you will spend your day, with dedicated blocks of time allocated to specific activities, carves out opportunities for extended deep concentration. If the nature of your work permits, devote certain full days each week to focus on your most important individual tasks, to more actively collaborate with others, or to catch up on busywork.

Build a habit of single tasking throughout the day. One of my talented but less productive team members was able to get more work done when we agreed that he would set aside half-day blocks of time during which he didn't answer the phone, texts, or e-mails—at all.

- **Employ routines to remove attention residue as you migrate from one task to the next.** Physically moving on to the next activity doesn't immediately unplug your brain from the previous one, which can be a distraction. A small transition ritual or distraction—a cup of coffee, an amusing video—can help you make that mental break into a new work mode.
- **Take regular breaks to recharge.** The intense concentration of a state of flow is great—up to a point. You must come up for air occasionally. Stretch your tired neck, arms, and shoulders. To minimize eyestrain, periodically focus your eyes on something in the distance for a few seconds instead of staring at the screen endlessly. Short mental breaks are refreshing before you dive back into that productive flow state.

Effective Hours

At-work hours seep away through many channels. You attend meetings and video chats, respond to e-mails, look things up on the web, participate in retrospectives, and review your teammates' code. Time gets lost to unexpected bug fixes, kicking around ideas with your coworkers, administrative activities, and the usual healthy socializing. Working from home offers myriad other distractions, many of them more fun than project work. Even if you work forty hours a week, you don't spend anywhere near that many on your project.

One software group of mine measured how we devoted our time on projects for several years (Wiegers 1996). Individuals tracked the time (to half-hour resolution) they spent working on each project in ten activity categories: project planning, requirements, design, implementation, testing, documentation, and four types of maintenance. We didn't try to make the weekly numbers add up to any total. We just wanted to know how we really spent our time, compared to how we thought we spent our time, compared to how we were supposed to spend our time.

The results were eye-opening. In the first year we collected data, we devoted an average of just 26 hours per week to project work. The tracking made us all more conscious of finding ways to focus our time more productively. However, we never exceeded an average of 31 hours of project time per week.

Several of my colleagues have obtained similar results, averaging five to six hours per day on project work. Other sources also suggest that a typical average of ideal work hours—"uninterrupted attentive time for project tasks"—is about five hours per day (Larman 2004). Rather than relying on published figures to estimate your effective project time, collect your own data. Recording how you work for a few typical weeks will provide a good idea of how many hours per week you can expect to devote to project tasks, which affects the team's projected productivity or velocity.

The intent of this time tracking is not so that managers can see who's working hard. Managers shouldn't even see the data for individuals, just aggregated team or organizational data. Knowing the team's average effective weekly work hours helps everyone make more realistic estimates, plans, and commitments.

Other Sources of Project Friction

Besides the daily frittering away of time on myriad activities, project teams lose time to other sources of friction. For instance, most corporate IT organizations are responsible for both new development and enhancing and repairing current production systems. Since you can't predict when something will break or a change request will come along, these sporadic, interruptive maintenance demands usurp team members' time with unplanned work.

Distance between project participants can retard information exchanges and decision-making. (See Lesson 39, "It takes little physical separation to inhibit communication and collaboration.") Even with the many collaboration tools available, projects with people in multiple locations and time zones should expect some slowdown from communication friction. Sometimes you can't reach an individual, such as a key customer representative who has the answer you need. You have to either wait until they're available or make your best guess so that you can proceed. That slows you down, especially when an incorrect guess leads to rework.

The team composition can further impose friction if project participants speak different native languages and work in diverse cultures. Unclear and volatile requirement priorities can chew up hours as people spend time researching, debating, and adjusting priorities. The team might have to temporarily shelve some incomplete work if a new, higher-priority task inserts itself into the schedule. Unplanned rework is yet another time diversion.

I know of a contract project that involved a customer in the eastern United States and a vendor in western Canada (Wiegers 2003). Their project plan included some peer reviews of certain deliverables. However, the long-distance reviews took longer than expected, as did follow-up to verify the corrections made. Slow decision-making across the distance further reduced the project's pace. Sluggish iteration to resolve requirements questions and ambiguity about who the right contact people were for each issue were further impediments. These—and other—factors put the project behind schedule after just the first week and eventually contributed to its failure.

Planning Implications

Project friction has a profound impact on estimation, so both individuals and teams must keep it in mind. I estimate how long individual tasks will take as though I will have no distractions or interruptions, just focused and productive time. Next, I convert that ideal effort estimate into calendar time based on my effective work-hour percentage. I also consider whether any of the other aforementioned sources of friction could affect my estimates. Then I try to arrange my work so that I can focus on a single task at a time until it's complete or I hit a blocking point.

My colleague Dave described what happens on his current project, whose manager doesn't consider the impacts of time lost to excessive multitasking:

> The manager likes to split people up between teams, 50 percent here and 50 percent there, or 50, 25, and 25. But when this happens, it seems like they forget the percentages and think the team has all full-time people. Then they seem surprised at how long things take. Also, being on multiple teams means more overhead in meetings and less coding time.

If people always create estimates without accounting for the many ways that time splitting and project conditions can slow down the work, they're destined to overrun their estimates every time.

Lesson 35 Knowledge is not zero-sum.

I used to work with a software developer named Stephan. Stephan was territorial about his knowledge. If I asked him a question, I could almost see the wheels turning in his head: "If I give Karl the full answer to his question, he'll know as much as I do about that topic. That's not acceptable, so I'll give him half of the answer and see if he goes away." If I came back later seeking a more complete response, I might get half of the remaining answer. In this way, I asymptotically approached getting the complete answer to my question.

Extracting information bit by bit from Stephan was annoying. The information I sought was never confidential. We both worked at the same company, so we should have been aligned toward our joint success. Stephan apparently didn't agree with me that freely sharing knowledge with your colleagues is a characteristic of a healthy software culture.

Knowledge isn't like other commodities. If I have $3 and give you one of them, now I have only $2. Money is zero-sum in the sense that I must lose some of it for you to gain something in this transaction. In contrast, if I give you some of my knowledge, I still possess all the knowledge myself. I can share it with other people, as can you. Everyone touched by this expanding circle of knowledge benefits.

A healthy organization fosters a culture of free knowledge exchange and continuous learning.

The Knowledge Hog

Some people hoard information out of insecurity. They fear that if they share some of their precious hard-won knowledge with others, those other people become more competitive with them. Perhaps that's true, but it's flattering for someone to ask for your help. The requester is acknowledging your superior experience and insights. Rarely, someone might ask you for information because they don't want to take the time to figure it out themselves. You don't have to do someone else's work for them, but you should remember that you and your teammates are all working toward the same ends.

Other people carefully protect their knowledge as a form of job security. If no one else knows what they know, the company can't possibly fire them, because too much institutional knowledge would go out the door. Maybe they think they should get a raise because they're the sole holder of so much important information.

People who conceal organizational knowledge pose a risk. They create an informational bottleneck that can impede other people's work. My colleague Jim Brosseau aptly refers to knowledge hoarding as *technical ransom* (Brosseau 2008). Information hiding is an excellent practice for software design; for software teams, not so much.

Rectifying Ignorance

A healthy organization fosters a culture of knowledge exchange and continuous learning. Sharing knowledge enhances everyone's performance, so management rewards people who freely pass along what they know, not those who keep it to themselves. In a learning organization, team members feel that it's psychologically safe to ask questions (Winters et al. 2020). We're all ignorant about most of the knowledge in the universe, so let's learn from our colleagues when the opportunity arises.

Experienced members of an organization can share their expertise in many ways. The most obvious method is simply to answer questions. But more than just answering questions, experts should invite the questions. They should appear approachable to fellow employees, particularly novices, and be thoughtful and patient when someone picks their brains. Beyond simply transferring information, experts can convey insights about how to apply the knowledge to specific situations.

Some organizations use formal mentoring programs to get new team members up to speed quickly (Winters et al. 2020). Pairing new employees with experienced colleagues greatly accelerates learning. When I began my professional career as a research chemist, I was the first guinea pig for a new mentoring program. My mentor, Seth, was a scientist in the group I had joined, but he wasn't in my reporting chain. I felt comfortable asking Seth questions that otherwise might have awkwardly revealed my ignorance to my manager. Seth helped me get rolling in an unfamiliar technology area. A mentoring or "buddy" program reduces the learning curve for new team members and starts building relationships with them immediately.

Scaling Up Knowledge Transfer

One-on-one communications are effective, but they don't scale well. Experienced, talented people are much in demand, both for their project work and for the expertise they can share. To cultivate a knowledge-sharing culture, consider techniques to leverage information more efficiently than one-on-one time spent together. Here are several possibilities.

Technical Talks

My software development team once decided to work through the many excellent programming examples in Steve McConnell's classic book *Code Complete* (McConnell 2004). We took turns studying a particular section and then describing it to the group in a lunch-and-learn session. These informal group learning experiences efficiently disseminated good programming practices across the group. They facilitated a shared understanding of techniques and a common vocabulary.

Presentations and Training

Formal technical presentations and training courses are good ways to communicate institutional knowledge across an organization. I developed several training courses when I worked at Kodak and delivered them many times. If you set up an internal training program, line up enough qualified instructors so that one individual doesn't have to teach the same courses all the time.

Documentation

Written knowledge spans the spectrum from specific project or application documentation to broadly applicable technical guides, tutorials, FAQs, wikis, and tip sheets. Someone must write this documentation, which means they aren't spending that time creating other project deliverables. Written documentation is a highly leverageable organizational asset, provided that team members turn to it as a useful resource.

I've known people who preferred to rediscover knowledge on their own rather than seek it from existing documentation. Those people didn't heed Lesson 7, "The cost of recording knowledge is small compared to the cost of acquiring knowledge." Once someone has invested the time to create relevant and useful documentation, it's a lot quicker to read it than to reconstruct the same information. All organization members should be able to update such documents to keep them valuable as current sources of pooled experience.

Deliverable Templates and Examples

When I worked in a large product development organization, our process improvement group built an extensive online catalog containing high-quality templates and examples of many project deliverables (Wiegers 1998b). We scoured company software departments for good requirements specifications, design documents, project plans, process descriptions, and other items. This "good practices" collection provided a valuable jump-start whenever any software practitioner in the company needed to create some new project artifact.

Technical Peer Reviews

Peer reviews of software work products serve as an informal mechanism for exchanging technical knowledge. They're a great way to look over other people's shoulders and to let them peek over yours. I've learned something from every review I've participated in, whether as a reviewer or as the author of the item being reviewed. The technique of pair programming, in which two people write code together, provides a form of instantaneous peer review, as well as exchanging knowledge between the programmers. See Lesson 48, "Strive to have a peer, rather than a customer, find a defect," for more about reviews.

Discussion Groups

Rather than trying to locate exactly the right person when you have a question, you might post the question to a discussion group or group chat tool within your company. Exposing your lack of knowledge to a large community can be awkward. That's why it's valuable to grow a culture that invites questioning and rewards those who assist. Ignorance is not a tragedy, but an unwillingness to solicit help is.

Discussion participants can offer multiple perspectives on your question quickly. The posted responses are available to everyone in the discussion, which further disseminates the knowledge. You probably weren't the only person who didn't know the answer to that specific question, so good for you for asking. I have a friend who's the most curious person I know. He's willing to ask anyone he encounters in daily life about things he sees them do that are unfamiliar to him. He's learned a lot that way, and the people he asks are always happy to share what they know.

A Healthy Information Culture

Everyone has something to teach—and to learn. When I managed a software development group, I hired a graduate student in computer science as a temporary summer employee. I confess that at first I was skeptical about his understanding of practical software engineering. Similarly, he had some disdain for the process-driven approach our group advocated. After just a few weeks, though, I gained respect for his contemporary programming knowledge, which far exceeded mine. And he acquired an appreciation for how sensible processes can help teams be more effective. We both grew by being open to what the other had to share.

You don't need to be the world's expert on some topic to be helpful. You just need some useful block of knowledge and the willingness to share it. In the world of technology, if you're one week ahead of the next person in some area, you're a wizard. Someone else will doubtless be ahead of you in other areas, so take advantage of their trailblazing. People in a healthy learning culture share what they know and also acknowledge that someone else might know a better way.

Lesson 43	When it comes to software quality, you can pay now or pay more later.

Suppose I'm a BA and I have a conversation with a customer to flesh out some requirements details. I go back to my office and write up what I learned in whatever form my project uses for requirements. The customer e-mails me the next day and says, "I just talked to one of my coworkers and learned that I had something wrong in that requirement we talked about yesterday." How much work must I do to correct that error? Very little; I simply update the requirement to match the customer's current request. Let's say that making that correction cost ten dollars' worth of company time.

Alternatively, suppose the customer contacts me a month or six after we had the conversation to point out the same problem. Now how much does it cost to correct that error? It depends on how much work the team has done based on the original, incorrect requirement. Not only does my company still have to pay ten dollars to fix the requirement, but a developer might have to redo some portion of the design. Maybe that costs another thirty or forty dollars. If the developers already implemented the original requirement, they'll have to modify or recode it. They'll need to update tests, verify the newly implemented requirement, and run regression tests to see whether the code changes broke anything. All that could cost perhaps a hundred dollars more. Maybe someone must revise a web page or a help screen as well. The bill keeps increasing.

Software's malleability lets us make changes and corrections whenever warranted. But every change has a price. Even discussing the possibility of adding some functionality or fixing a bug and then deciding not to do it takes time. The longer a requirement defect lingers undetected and the more rework you have to do to correct it, the higher the price tag.

The Cost-of-Repair Growth Curve

The cost of correcting a defect depends on when it was introduced into the product and when someone fixed it. The curve in Figure 6.2 shows that the cost increases significantly for late-discovered requirements errors. I omitted a numeric scale on the y-axis because various sources cite different data, and software people debate the exact numbers. The cost ratio and steepness of the curve depend on the product type, the development life cycle being followed, and other factors.

For instance, data from Hewlett-Packard indicated that the cost ratio could be as high as 110:1 if a customer discovered a requirement defect in production versus someone finding it during requirements development (Grady 1999). Another analysis

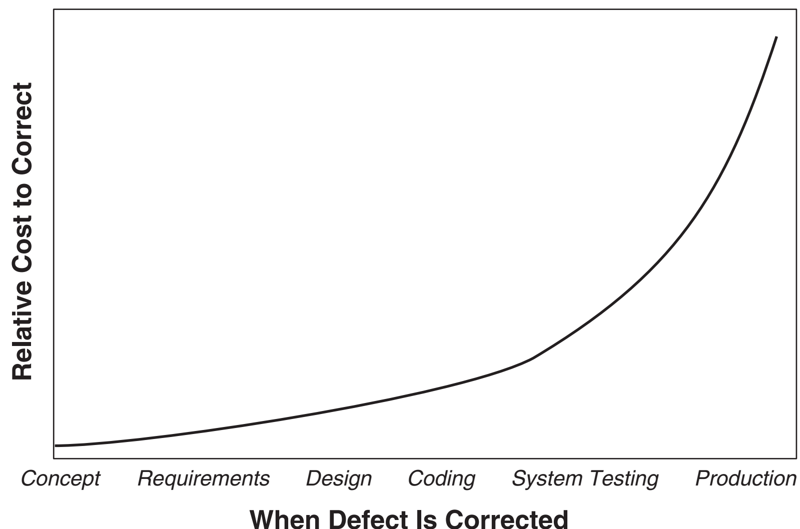


Figure 6.2 *The cost to correct a defect increases rapidly with time.*

suggested a relative cost factor of 30:1 to correct errors introduced during requirements development or architectural design that were discovered post-release (NIST 2002). For highly complex hardware/software systems, the cost amplification factor from discovery in the requirements stage versus the operational stage can range from 29x to more than 1,500x (Haskins et al. 2004).

The cost of correcting a defect depends on when it was introduced into the product and when someone fixed it.

Regardless of the exact numbers, there's broad agreement that early defect correction is far cheaper than fixing defects following release (Sanket 2019, Winters et al. 2020). It's a bit like paying your credit card bill. You can pay the balance due on time, or you can pay a smaller amount now plus the remaining balance along with substantial interest charges and late fees in the future. Johanna Rothman (2000) compared how three hypothetical companies could employ different strategies to deal with defects and consequently experience different relative defect-repair costs. However, in all three scenarios, the later in the project the team fixes a defect, the more it costs.

Some people have argued that agile development greatly flattens the cost-of-change curve (Beck and Andres 2005). I haven't yet located any actual project data to support this contention. However, this lesson isn't about the cost of making a change like adding new functionality—it's about the price you pay to correct defects. A requirements defect that is discovered before a user story is coded is still less expensive to repair than the same defect identified during acceptance testing. Scott Ambler (2006) suggests that the relative defect-correction cost is lower on agile projects because of agile's quick feedback cycles that shorten the time between when some work is done and when its quality is assessed. That sounds plausible, but it only partially addresses the fundamental issue with defect-repair costs.

The issue with cost-to-repair is not only the days, weeks, or months between when the defect was injected into the product and when someone discovers it. The amplification factor depends heavily on how much work was done based on the defective bit that now must be redone. It costs very little to fix a coding error if your pair-programming partner finds it moments after you typed the wrong thing, when knowledge about the defective work is fresh in your brain. However, if the customer calls to report the same type of error when the software is in production, it certainly will be far more difficult to rectify. As an example, a developer friend of mine related this recent experience:

> This week I missed one comma (literally) in a ColdFusion script on a custom website for a client. It caused a crash, which caused him a delay and hassle. Plus, then there were the back-and-forth e-mails, and then me opening up all the tools and source code and finding the bug, adding the comma, retesting, and so on. One darn comma.

Besides the cost-to-repair issue, my friend alluded to another important aspect of late defect detection we should keep in mind: the negative impact on the users.

Harder to Find

Diagnosing a system failure takes longer if the underlying fault was introduced long ago. If you review some requirements and spot an error, you know exactly where the problem lies. However, if a customer reports a malfunction—whether that's one month or five years after someone wrote the requirement—the detective work is more challenging. Is the failure due to an erroneous requirement, a design problem, a coding bug, or an error in a third-party component? Therein lies Ambler's argument for lower defect-correction costs on agile projects: when defects are revealed shortly after they're introduced, it's easier to locate the fault that caused a failure.

After you uncover the root cause—the fault—for a customer-reported system failure, you have to recognize all of the affected work products, repair them, retest the system, write release notes, redeploy the corrected product, and reassure the customer that the problem is fixed. That's a lot of expensive *re-* stuff to do. Plus, at that point, the problem has affected many more stakeholders than if someone had found it much earlier.

Early Quality Actions

Serious defects discovered during system testing can lead to a lot of repair work. Those found after release can disrupt user operations and trigger emergency fixes that divert team members from new development work. This reality leads us to several thoughts about how to pay less for high-quality software.

Prevent Defects Instead of Correcting Them

Quality *control* activities, such as testing, code static analysis, and code reviews, look for defects. Quality *assurance* activities seek to prevent defects in the first place. Improved processes, better technical practices, more proficient practitioners, and taking a little more time to do our work carefully are all ways to prevent errors and avoid their associated correction costs.

Push Quality Practices to the Left

Regardless of the project's development life cycle, the earlier you find a defect, the cheaper it is to resolve. Each piece of software work involves a micro-sequence of requirements, design, and coding, moving from left to right on a timescale axis. We've seen that eradicating requirement errors provides the greatest leverage for time savings down the road. Therefore, we should use all the tools at our disposal to find errors in requirements and designs before they're translated into erroneous code.

Peer reviews and prototyping are effective ways to detect requirement errors. Pushing testing from its traditional position late in the development sequence—on the right side of the timeline—far to the left is particularly powerful. Strategy options include following a test-driven development process (Beck 2003), writing acceptance tests to flesh out requirements details (Agile Alliance 2021b), and—my preference—concurrently writing functional requirements and their corresponding tests (Wiegers and Beatty 2013).

Every time I write tests shortly after writing requirements, I discover errors in both the requirements and the tests. The thought processes involved with writing requirements and tests are complementary, which is why I find that doing both yields the highest-quality outcome. Writing tests through a collaboration between the BA and the tester leverages both the idea of doing it earlier in the process and having multiple sets of eyes looking at the same thing from different perspectives. Writing tests early in the development cycle doesn't add time to the project; it just reallocates time to a point where it provides greater quality leverage. Those conceptual tests can be elaborated into detailed test scenarios and procedures as development progresses.

During implementation, developers can use static and dynamic code analysis tools to reveal many problems far faster than humans can review code manually. These tools can find run-time errors that code reviewers struggle to spot, such as memory corruption bugs and memory leaks (Briski et al. 2008). On the other hand, human reviewers can spot code logic errors and omissions that automated tools won't detect.

The timing of quality control activities is important. I once worked with a developer who wouldn't let anyone review her code until it was fully implemented, tested, formatted, and documented—that is, clear on the right side of her development timescale. At that point, she was psychologically resistant to hearing that she wasn't done after all. Each issue that someone raised in a code review triggered a defensive response and rationalization about why it was fine the way it was. You're much better off starting with preliminary reviews on just a portion of a work item—be it requirements, design, code, or tests—to get input from others on how to craft the rest of the item better. Push quality to the left by reviewing early and often.

Track Defects to Understand Them

The most efficient way to control defects is to contain them to the life cycle activity—requirements, design, coding—in which they originated. Record some information about your bugs instead of simply swatting them and moving on. Ask yourself questions to identify the origin of each defect so that you can learn what types of errors are the most common. Did this problem happen because I didn't understand what the customer wants? Did I understand the need accurately but make an incorrect assumption about other system components or interfaces? Did I simply make a mistake while coding? Was a customer change request not communicated to everyone who needed to know about it?

Note the life cycle activity (not necessarily a discrete project phase) in which each defect originated and how it was discovered. You can calculate your defect containment percentage from that data to see how many problems are leaking from their creation stage into later development activities, thereby amplifying their cost-to-repair factors. That information will show you which practices are the best quality filters and where your improvement opportunities lie.

Minimizing defect creation and finding them early reduces your overall development costs. Strive to bring your full arsenal of quality weapons to bear from the earliest project stages.

Lesson 51 Watch out for "Management by Businessweek."

Frustration with disappointing results is a powerful motivation for trying a different approach. However, you need to be confident that any new strategy you adopt has a good chance of solving your problem. Organizations sometimes turn to the latest buzzword solution, the hot new thing in software development, as the magic elixir for their software challenges.

A manager might read about a promising—but possibly overhyped—methodology and insist that his organization adopt it immediately to cure their ills. I've heard this phenomenon called "Management by Businessweek." Perhaps a developer is enthused after hearing a conference presentation about a new way of working and wants his team to try it. The drive to improve is laudable, but you need to direct that energy toward the right problem and assess how well a potential solution fits your culture before adopting it.

Over the years, people have leaped onto the bandwagons of countless new software engineering and management paradigms, methodologies, and frameworks. Among others, we've gone through

- Structured systems analysis and design
- Object-oriented programming
- Information engineering
- Rapid application development
- Spiral model
- Test-driven development
- Rational Unified Process
- DevOps

More recently, agile software development in numerous variations—Extreme Programming, Adaptive Software Development, Feature-Driven Development, Scrum, Lean, Kanban, Scaled Agile Framework, and others—has exemplified this pursuit of ideal solutions.

Alas, as Frederick P. Brooks, Jr. (1995) eloquently informs us, there are no silver bullets: "There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity." All of the approaches in the preceding list have their merits and limitations; all must be applied to appropriate problems by properly prepared teams and managers. I'll use a hypothetical new software development approach called Method-9 as an example for this discussion.

Before you settle on any new development approach, ask yourself, "What's preventing us from achieving those better results today?"

First Problem, Then Solution

The articles and books that its inventors and early adopters wrote about Method-9 praised its benefits. Some companies are drawn to Method-9 because they want their products to satisfy customer needs better. Maybe you want to deliver useful software faster (and who doesn't?). Method-9 can get you there. Perhaps you wish to reduce the defects that annoy customers and drain the team's time with rework (again, who doesn't?). Method-9 to the rescue! This is the essence of process improvement: setting goals, identifying barriers, and choosing techniques you believe will address them.

Before you settle on any new development approach, though, ask yourself, ***"What's preventing us from achieving those better results today?"*** (Wiegers 2019f). If you want to deliver useful products faster, what's slowing you down? If your goal is fewer defects and less rework, why do your products contain too many bugs today? If your ambition is to respond faster to changing needs, what's standing in your way?

In other words, if Method-9 is the answer—at least according to that article you read—what was the question?

I suspect that not all organizations perform a careful root cause analysis before they latch on to what sounds like a promising solution. Setting improvement goals is a great start, but you must also understand the current obstacles to achieving those goals. You need to treat real causes, not symptoms. If you don't understand those problems, choosing any new approach is just a hopeful shot in the dark.

A Root Cause Example

Suppose you want to deliver software products that meet customers' needs better than in the past. You've read that Method-9 teams include a role called the Vision Guru, who's responsible for ensuring the product achieves the desired outcome. "Perfect!" you think. "The Vision Guru will make sure we build the right thing. Happy customers are guaranteed." Problem solved, right? Maybe, but I suggest that, before making any wholesale process changes, your team should understand why your products don't thrill your customers already.

Root cause analysis is a process of thinking backward, asking "why" several times until you get to issues that you can target with thoughtfully selected improvement actions. The first contributing cause suggested might not be directly actionable; nor might it be the ultimate root cause. Therefore, addressing that initial cause won't solve the problem. You need to ask "why" another time or two to ensure that you're getting to the tips of the analysis tree.

Figure 7.1 shows a portion of a fishbone diagram—also called an Ishikawa or cause-and-effect diagram—which is a convenient way to work through a root cause analysis. The only tools you need are a few interested stakeholders, a whiteboard, and some markers. Let's walk through that diagram.

Your goal is to better meet customer needs with the initial release of your software products. Write that goal on a long horizontal line. Alternatively, you could phrase it as a problem statement: "Initial product release does not meet customer needs." In either case, that long horizontal line—the backbone in the fishbone diagram—represents your target issue.

Next, ask your group, "Why are we not already meeting our customer needs?" Now the analysis begins. One possible answer is that the team doesn't get adequate input to the requirements from end users—a common situation. Write that cause on a diagonal line coming off the goal statement line. That's a good start, but you need a deeper understanding to know how to solve the problem. You ask, "Why not?"

One member of the group says, "We've tried to talk to real users, but their managers say they're too busy to work with the software team." Someone else complains that the surrogate customer representatives who work with the team don't do a good job of presenting the ultimate users' real needs. Write those second-level causes on horizontal lines coming off the parent problem's diagonal line.

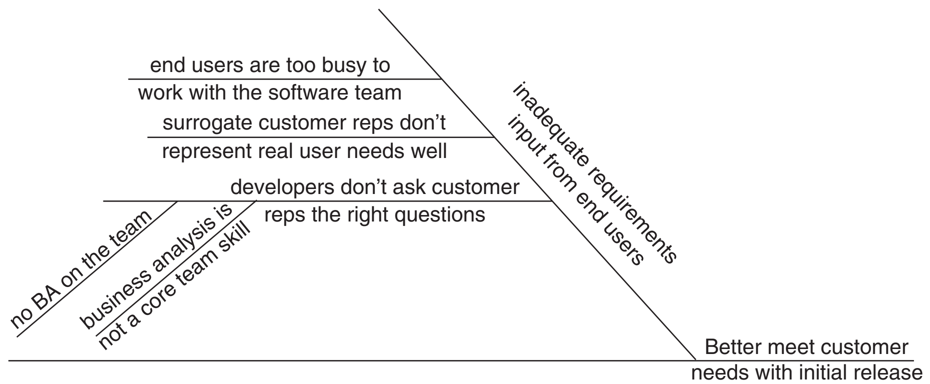


Figure 7.1 *Root cause analysis often is depicted as a fishbone diagram.*

A third participant points out that the developers who attempt to elicit requirements aren't skilled at asking customer reps the right questions. Then comes the usual follow-up question: "Why not?" There could be multiple reasons, including a lack of education or interest in requirements on the developers' part. It might be that business analysis is neither a core team skill nor a dedicated team role. Each cause goes onto a new diagonal line attached to its parent.

Now you're getting to actionable barriers that stand between your team's current performance and where you all want to be. Continue this layered analysis until the participants agree that they understand why they aren't already achieving the desired results. I've found this technique to be remarkably efficient at focusing the participants' thinking and quickly reaching a clear understanding of the situation. The diagram might get messy; consider writing the causes on sticky notes so that you can shuffle them around as the exploration continues.

Diagnosis Leads to Cure

In subsequent brainstorming sessions, team members can explore practical solutions to address those root causes. Then you are well on your way toward achieving superior performance. You might conclude that adding experienced business analysts (BAs) to your teams could be more valuable than adopting Method-9 with its Vision Guru. Or maybe the combination of the two will prove to be the secret sauce. You just don't know until you think it through.

As you contemplate whether a new development method will work for you, look past the hype and the fads. Understand the prerequisites and risks associated with the new approach, and then balance those against a realistic appraisal of the potential payoff. Good questions to explore include these.

- Will your team need training, tools, or consulting assistance to get started and sustain progress?
- Would the solution's cost yield a high return on investment?
- What possible cultural impacts would the transition have on your team, your customers, and their respective organizations and businesses?
- How ugly could the learning curve be?

The insights from a root cause analysis can point you toward better practices to address each problem you discover. Without exploring the barriers between where you are today and your goals, don't be surprised if the problems are still there after you switch to a different development strategy. Try a root cause analysis instead of chasing after the Hottest New Thing someone read in a headline.

Root cause analysis takes less time than you might fear. It's a sound investment in focusing your improvement efforts effectively. Any doctor will tell you that it's a good idea to understand the disease before prescribing a treatment.

Appendix

Summary of Lessons

Requirements

#1. If you don't get the requirements right, it doesn't matter how well you execute the rest of the project.

#2. The key deliverables from requirements development are a shared vision and understanding.

#3. Nowhere more than in the requirements do the interests of all the project stakeholders intersect.

#4. A usage-centric approach to requirements will meet customer needs better than a feature-centric approach.

#5. Requirements development demands iteration.

#6. Agile requirements aren't different from other requirements.

#7. The cost of recording knowledge is small compared to the cost of acquiring knowledge.

#8. The overarching objective of requirements development is clear and effective communication.

#9. Requirements quality is in the eye of the beholder.

#10. Requirements must be good enough to let construction proceed at an acceptable level of risk.

#11. People don't simply gather requirements.

#12. Requirements elicitation must bring the customer's voice close to the developer's ear.

#13. Two commonly used requirements elicitation practices are telepathy and clairvoyance. They don't work.

#14. A large group of people can't agree to leave a burning room, let alone agree on exactly how to word some requirement.

#15. Avoid decibel prioritization when deciding which features to include.

#16. Without a documented and agreed-to project scope, how do you know whether your scope is creeping?

Design

#17. Design demands iteration.

#18. It's cheaper to iterate at higher levels of abstraction.

#19. Make products easy to use correctly and hard to use incorrectly.

#20. You can't optimize all desirable quality attributes.

#21. An ounce of design is worth a pound of recoding.

#22. Many system problems take place at interfaces.

Project Management

#23. Work plans must account for friction.

#24. Don't give anyone an estimate off the top of your head.

#25. Icebergs are always larger than they first appear.

#26. You're in a stronger negotiating position when you have data to build your case.

#27. Unless you record estimates and compare them to what actually happened, you will forever be guessing, not estimating.

#28. Don't change an estimate based on what the recipient wants to hear.

#29. Stay off the critical path.

#30. A task is either entirely done or it is not done: no partial credit.

#31. The project team needs flexibility around at least one of the five dimensions of scope, schedule, budget, staff, and quality.

#32. If you don't control your project's risks, they will control you.

#33. The customer is not always right.

#34. We do too much pretending in software.

Culture and Teamwork

#35. Knowledge is not zero-sum.

#36. No matter how much pressure others exert, never make a commitment you know you can't fulfill.

#37. Without training and better practices, don't expect higher productivity to happen by magic.

#38. People talk a lot about their rights, but the flip side of every right is a responsibility.

#39. It takes little physical separation to inhibit communication and collaboration.

#40. Informal approaches that work for a small colocated team don't scale up well.

#41. Don't underestimate the challenge of changing an organization's culture as it moves toward new ways of working.

#42. No engineering or management technique will work if you're dealing with unreasonable people.

Quality

#43. When it comes to software quality, you can pay now or pay more later.

#44. High quality naturally leads to higher productivity.

#45. Organizations never have time to build software right, yet they find the resources to fix it later.

#46. Beware the crap gap.

#47. Never let your boss or your customer talk you into doing a bad job.

#48. Strive to have a peer, rather than a customer, find a defect.

#49. Software people love tools, but a fool with a tool is an amplified fool.

#50. Today's "gotta get it out right away" development project is tomorrow's maintenance nightmare.

Process Improvement

#51. Watch out for "Management by Businessweek."

#52. Ask not, "What's in it for me?" Ask, "What's in it for us?"

#53. The best motivation for changing how people work is pain.

#54. When steering an organization toward new ways of working, use gentle pressure, relentlessly applied.

#55. You don't have time to make each mistake that every practitioner before you has already made.

#56. Good judgment and experience sometimes trump a defined process.

#57. Adopt a shrink-to-fit philosophy with document templates.

#58. Unless you take the time to learn and improve, don't expect the next project to go any better than the last one.

#59. The most conspicuous repeatability the software industry has achieved is doing the same ineffective things over and over.

General

#60. You can't change everything at once.

Index

Numbers

10x developer performance ratio, 170
$100 million syndrome, 206–207
100 percent done, 133

A

A/B testing, 78
abandoned projects, 201–207, 227
About Face (Cooper et al.), 100
absorbing risk, 144
abstraction
  advantages of design, 75
  iterating at high level of, 79–84
acceptance criteria, 31
acceptance tests, 15, 29, 32, 40, 48, 197, 198–199
ACM (Association for Computing Machinery), 171
action planning for change, 262
  template for, 266–267
activity network diagrams, 129–131
adaptive maintenance, 221
Advanced Automation System, 206
Affordable Care Act (ACA), 206–207
after-action reviews. *See* retrospectives
agile requirements, 28–33, 269
  activity timing, 30–31
  deliverable forms, 31–32
  documentation detail, 30
  representation of, 31–32
  prioritization of, 32
  roles and responsibilities for, 29
  terminology for, 29
agile software development. *See also* Scrum
  accommodating change in, 106
  architecture in, 226, 229, 246–247
  business analyst on, 29, 33, 54
  communication pathways for requirements, 54
  contingency buffers in, 118–119
  cost of change in, 197
  cultural change for, 182–184
  definition of done in, 133
  frameworks for, 229
  iteration backlog, 66
  job stories in, 25
  lightweight and just-in-time documentation, 30–31
  Manifesto for Agile Software Development, 36, 226, 253
  planning game, 62
  product backlog, 29, 30, 53, 54, 65, 66, 106, 119, 192
  product owners, 10, 28, 36–37, 43, 53–54, 114, 170, 183
  requirements in, 28–33, 269. *See also* agile requirements, requirements
  retrospectives in, 253
  scope management in, 65, 137–138
  spikes, 77–78
  technical debt in, 93
  user stories in, 24–25, 29, 30
air traffic control (ATC) system, 206
algorithm development, 73
alignment, of teams, 157–158
allocating requirements to design components, 71–72

ambiguous requirements, 31, 40, 44, 67–68
Ambler, Scott, 197
American Society for Quality, 189
analysis
  paralysis, 34, 84, 257
  requirements, 9
  risk, 143–145
  stakeholder, 18–20
analyst, business. *See* business analyst (BA)
architectural design, 72, 82, 92, 195–196
  interfaces, 75, 94–100, 270
architecture
  agile projects and, 226, 229, 246–247
  quality attributes and, 92
assessment, of current processes, 237, 262
Association for Computing Machinery (ACM), 171
assumed requirements, 55–57
assumptions
  defined, 56
  and contingency buffers, 119–120
  in estimates, 115, 117, 119–120, 129, 141–142
  in requirements, 55–57, 248, 249–250
assurance, quality, 198
attributes, quality. *See* quality attributes
audiences for requirements, 38–39
authentication, multifactor, 91
automated testing tools, 220
availability, 88, 91

B

BA. *See* business analyst (BA)
BABOK (Business Analysis Body of Knowledge), 46
Bach, James, 3
backlog
  iteration, 66
  product, 29, 30, 53, 54, 65, 66, 106, 119, 192
bad-job pressure, 207–213, 272
  circumventing processes, 212–213
  illustration of, 207–208
  lack of knowledge, 211–212
  power plays, 210
  rushing to code, 211
  scenarios for, 208–210
  shady ethics, 212
barriers, removing, 107–108, 266
Beatty, Joy, 244, 247
Bentley, Jon, 3
best practices, 243–244
bottom-up change, 262
brainstorming, 4, 141–142, 215, 232
breaks, importance of, 111
Bright Hub PM, 142
Brooks, Frederick P. Jr. 76, 137, 229
Brooks's Law, 137
Brosseau, Jim, 160
budget, as a project dimension, 136–140, 271
buffers, contingency, 116–121
  advantages of, 68, 121
  on agile projects, 118–119
  assumptions in, 119–120
  contracting and, 120–121
  example of, 117–119
  plans for risks in, 119–120
building in quality, 192–193, 200–202, 205–206
burning room, leaving a, 57–58, 270
business
  analysis. *See* requirements
  analyst. *See* business analyst (BA)
  objectives, 63
  requirements, 8, 15, 21, 26, 63
  rules, 20, 33, 35–36, 38, 45, 50, 179
business analysis. *See* requirements
Business Analysis Body of Knowledge (BABOK), 46
business analyst (BA), 10, 90
  on agile projects, 29, 33, 54
  as facilitator, 29–33

modeling for, 81
as project role, 10
project management and, 114
business requirements document (BRD), 15
Businessweek, management by, 228–233, 272
buy-in, process improvement and, 234

C

Capability Maturity Model for Software (CMM), 163, 187, 226
Capability Maturity Model Integration (CMMI), 226, 246
cause-and-effect diagram, 231–232
change, 265. *See also* process improvement
on agile projects,106
bottom-up, 262
cost of, 195–198
criteria for success of, 265
cultural, 180–184, 271
as cycle, 263
gentle pressure and, 239–240, 272
management, 106, 240–241
metrics for, 239
objectives of, 239
pain as motivation for, 236–238, 272
process of, 262–263
prioritizing, 264
rate of, 262–263
requirements, 116–117
scope, 107, 129
treating as a project, 240, 266–267
CHAOS report, 257–258
charter, project, 21
charts. *See also* diagrams; modeling
Gantt, 117–119
PERT, 129–131
clairvoyance, 55–57, 270
clashes, communication, 180
CMM. *See* Capability Maturity Model for Software (CMM)
CMMI. *See* Capability Maturity Model Integration (CMMI)
Code Complete (McConnell), 3, 161
code of ethics, 171
code
patches, 221
reuse, 94
reviews, 233. *See also* peer reviews
cohesion, 74
Cohn, Mike, 24, 27, 114
collaboration, 173–177, 271
barriers of space and time in, 174–175
by virtual teams, 175–176
collective versus individual results, 233–235
personal payoff, 235
team return on investment, 233–234
commitment versus support, 182
commitments, 105, 163–166, 271
chain of, 164–165
versus estimates, 105, 116
ethics and professionalism of, 163–164
inability to fulfill, 163–164
for process improvement, 240
resisting pressure to make, 163–166, 271
commitment management, 105
communication
barriers to, 112–113, 173–177, 271
benefits of written, 34–38
clashes, 180
collaboration and, 173–177, 271
explicit versus implicit, 249
friction from, 112–113
importance of, 170, 238
for knowledge transfer, 161–163
management, 106
pathways for, 52–54
public, 156

representation techniques for, 40–41
requirements, 41, 48, 269
with stakeholders, 19–20
standards for, 88
verbal, 30
visual models for, 84
compliance, regulatory, 63
conflicting requirements, 146
conformance to standards, 88, 190
congruence, cultural, 155
Constantine, Larry, 74
constraints, 72, 187, 192
design and implementation, 45, 72, 248
interface, 96
from nonfunctional requirements, 88, 248
project dimension, 123–124, 129, 137–138, 139–471
schedule and budget, 68
from stakeholders, 17–18, 19–20
triple, 136, 190–191
context diagram, 66
contingency buffers, 116–121
advantages of, 68, 121
on agile projects, 118–119
assumptions in, 119–120
contracting and, 120–121
example of, 117–119
plans for risks in, 119–120
contingency plans, 106, 119–120, 144
continuous learning, 160–161
contract
management, 107
team, 35
contracting, 68, 120–121
Cooper, Alan, 100
corporate culture, 154. *See also* culture
corrective maintenance, 221–223
cost
of change, 195–198
of quality, 89, 203–205
of quality goals, 92
to repair defects, 195–197
coupling, 74
Cover Oregon project, 206–207
crap gap, 207–209, 272. *See also* bad-job pressure
Creating a Software Engineering Culture (Wiegers), 153
creativity and process, 226
critical chain project management, 117
critical path, 129–132, 271
defined, 129–131
staying clear of, 131–132
criticism, giving and accepting, 213–214
Crosby, Philip B., 190, 203
CRUD, 73
culture, 153–158. *See also* teamwork
changing, 180–184, 265, 271
changing to agile development, 182–184
congruence and, 155
corporate, 154
crystallizing, 156–157
defined, 153–154
first steps for, 158–159
fragility of, 154
healthy software engineering, 153–155, 158, 159, 160, 163
internalization of, 162, 184
knowledge and, 159–163, 271
management and, 154, 265
next steps for, 187–188
peer reviews and, 215–217
values, behaviors, and technical practices, 181–182
Cunningham, Ward, 222
customers. *See also* stakeholders
are not always right, 145–148
back door processes of, 147
communication pathways for, 52–54
conflicting requests from, 146
ear of the developer and, 51–54, 270

positional power of, 148
as product champions, 52–54
respecting viewpoint of, 148–149
rights and responsibilities of, 172
as source of requirements and constraints, 13–14
value to, 76, 91, 92, 93
voice of, 145–148
cycle, process improvement, 262–263

D

daily scrum, 59
DAR (display-action-response) model, 81
data
for estimation, 114–116
historical, 120, 124–127
as a negotiation aid, 121–123
software metrics, 126–127
validation of interface, 98–99
data flow diagrams, 244
database design, 73
Davis, Alan, 45
debt, technical. *See* technical debt
decision makers, 21
defects
containment, 199
cost to repair, 195–197
difficulty of finding, 197–198
effect on productivity, 200–202
finding versus fixing, 203
preventing, 141–142, 168, 198
tracking, 199–200
zero, 190
deferred requirements, 135
defined processes
judgment and experience versus, 244–247, 272
nondogmatism and, 246–247
rhythms versus, 245–246
degrees of freedom, 137–138, 139
deleted requirements, 135
deliverable forms, 31–32
deliverable templates, 162
Deming, Dr. W. Edwards, 262
dependencies, 106–107, 120, 128, 130, 164–165, 248, 249–250
contingency plans for, 106
design, 71–75
architectural, 72, 82
avoiding redesign and recoding with, 92–94, 270
database, 73
defined, 71
detailed, 73, 74, 81–82
first steps for, 75–76
good, 74–77
interfaces, 75, 94–100. *See also* interfaces
iteration on, 76–84, 270
modeling, 81
next steps for, 100–101
prefactoring, 94
principles for, 74–75
prototypes in, 77–78. *See also* prototypes
for quality, 223, 270
quality attributes in, 87–92. *See also* quality attributes
refactoring, 93, 202, 222
requirements and, 71–72
user experience, 73–74, 99–100
user interface, *See* user interface design
value of, 270
for X, 88
Design for X (Design for Excellence), 88
designer, 43
detail
documentation on agile projects, 30
progressive refinement of, 26–27
requirements, 45–46
detailed design, 73, 74, 81–82
developers, 43
performance variation of, 169–170
rights and responsibilities of, 172

development, requirements, 9, 16–17, 269. *See also* elicitation, requirements
- on agile projects, 28–32, 269. *See also* agile requirements
- business analyst role in, 13–14
- communication, 41, 269
- documentation for, 33–37. *See also* documenting requirements
- emergent functional and nonfunctional requirements, 27–29
- feature bloat in, 22
- objective of, 37–41, 269
- purpose of, 11
- subdomains of, 9–10
- timing of, 13–14
- usage-centric strategy for, 22–23
- vision and understanding in, 15–17

DFD (data flow diagrams), 244
DfX, 88
diagrams. *See also* charts
- activity network, 129–131
- cause-and-effect, 231–232
- context, 66
- data flow, 244
- dialog map, 82–83
- ecosystem, 66
- flexibility, 138–140
- Kiviat, 138–140
- in requirements, 40–41
- state-transition, 82
- statechart, 82
- UML, 84
- use case, 66

dialog maps, 82–83
Dijkstra, Edsger, 74
direct users, 19
discussion groups, 162
display-action-response (DAR) model, 81
distance, as communication barrier, 112–113, 175–176
document analysis, 50
document templates, shrinking to fit, 247–252, 272
documentation, 161–162
- benefits of, 30, 34–36
- cost of recording, 33, 269
- lightweight and just-in-time, 30–31
- project charter, 21
- vision and scope, 21, 65–66

documenting requirements, 33–37
- balance in, 36–37
- benefits of, 34–36
- cost of recording, 33–37, 269
- detail in, 30
- fear of writing, 34
- limitations of, 36–37, 269

dogmatic, not being, 170, 187, 210, 246–247
doing more with less, 167
done, definition of, 122–136, 271
- no partial credit, 132, 134–135
- one hundred percent done, 133
- planning checklist for, 132–134
- requirements status tracking, 135
- value and, 136

doors, office, 176–177
durability, 89

E

ear of the developer, 51–54, 270
Eastman Kodak Company. *See* Kodak
ecosystem diagram, 66
effective hours, 109–111
efficiency, 88, 91
effort, metrics for, 112, 126, 203, 255
elicitation, requirements, 46–51, 270
- clairvoyance and, 55–57, 270
- communication pathways for, 52–54
- conflicting requests in, 146
- context of, 48–49
- defined, 47
- document analysis, 50
- gathering versus, 46–47
- interviews, 49
- observations, 50

practices for, 55–57, 270
prototypes, 51
requirements for, 270
scope creep, 64–68
solutions versus needs in, 146
surveys, 50
techniques for, 49–51
telepathy and, 55–57, 270
voice of customer in, 145–148
wikis, 50
workshops, 49, 57–61
emergent requirements
functional, 27–28
nonfunctional, 28–29
emotions
buy-in for process improvement, 234
retrospectives and, 254
epics, 29
errors, user, 86–87
estimates, 114–116
assumptions in, 115, 117, 119–120, 129, 141–142
best response to a request for, 114–116, 271
changing, 127–128
versus commitment, 105, 116
versus goals, 127–129
presenting as a range, 115
when to adjust, 129
estimation, 114–116
from historical data, 124–127
process, 115
ethics
code of, 171
shady, 212
Evans, Gary K. 99
event lists, 66
expectation management, 15, 104–105
exposure, risk, 144–145
extensibility, 89
external interface requirements, 8
external quality, 192
external stakeholders, 18
exuberance, irrational, 150

F

facilitation, 29–33, 59
fault handling, 89
fear of writing, 34
Fearless Change (Manns and Rising), 262
feasibility, of requirements, 43
feature road map, 66
feature tree, 66
feature-centric requirements approach, 21–25, 269
features
bloat, 22
prioritization of, 61–64
unused, 21–22
versus usage, 22–23
Federal Aviation Administration's Advanced Automation Program, 206
feeding buffer, 118
First Steps
requirements, 11–12
design, 75–76
project management, 108–109
culture and teamwork, 158–159
quality, 194–195
process improvement, 228
fishbone diagram, 231–232
fitness for use, 190
five-dimensional analysis, 136–140, 271
flexibility diagrams, 138–140
float time, 130
flow, 109–111
flow, task, 22, 73–74, 83
focus, change efforts and, 264
fool with a tool, 217–220
foundational functionality, 64
frameworks for agile software development, 229, 241–253
freedom, degrees of, 137–138, 139
friction, project, 109–113
additional sources of, 112–113
effective hours and, 109–112
example of, 109

planning implications of, 113
project management and, 109–113, 270
task switching and flow, 109–111
functional requirements, 8, 29, 31, 198–199

G

Gantt chart, 117–119
garbage in, garbage out (GIGO), 220
gathering requirements. *See* elicitation, requirements
gentle pressure, relentlessly applied, 239–240, 272
GIGO (garbage in, garbage out), 220
glossary, 39
going slow to go fast, 168
goal versus estimate, 128
good design, 74–77
good enough, 44–46
quality, 45, 193
requirements, 44–46, 269
good practices, 162, 243–244, 246
Gottesdiener, Ellen, 59
grammar checker, 219
group memory, 179
group workshops, 49, 57–61
groups, discussion, 162
Guiney, Eamonn, 23, 244

H

hardware engineers, 37, 38, 43
Harris, Sidney, 72
HCI, 73–74
healthy software engineering culture, 153–155, 158, 159, 160, 163
HealthCare.gov, 206–207
heuristics for user interface design, 100
hiding information, in design, 74
historical data, 120
for estimation, 124–127
hoarding of information, 160
horizontal prototypes, 78–79
horizontal scope, 59
hours, effective, 109–111
human–computer interaction (HCI), 73–74

I

IBM, inspection program at, 234
icebergs, 116–121, 271
IDEF0, 41
IEC (International Electrotechnical Commission), 250
IEEE (Institute of Electrical and Electronics Engineers), 171, 247, 250
IEEE Computer Society, 171, 247
IIBA (International Institute of Business Analysis), 10, 46
IIBA Business Analysis Body of Knowledge, 46
implied requirements, 55–57
inch-pebbles, 134–135
incremental development, 98, 121, 257. *See also* agile software development
indirect users, 19
individual versus collective results, 233–235
personal payoff, 235
team return on investment, 233–234
information culture
healthy, 163
information hoarding, 160
knowledge exchange and continuous learning, 160–161
knowledge transfer techniques, 161–162
lack of knowledge, 211–212
information hiding, in design, 74
information hoarding, 160
inputs, validating, 98–99
inspection, 42, 215, 234. *See also* peer reviews

installability, 88
Institute of Electrical and Electronics Engineers (IEEE), 171, 247, 250
institutionalization, of practices, 184
integrity, 88, 91
interfaces, 75, 94–100, 270
  input data validation in, 98–99
  making easy to use, 84–87
  system problems at, 94–95
  technical issues with, 96–98
internal quality, 192
internalization
  of practices, 154, 167, 184, 205
  of process, 246
International Electrotechnical Commission (IEC), 250
International Institute of Business Analysis (IIBA), 10, 46
International Organization for Standardization (ISO), 250
interoperability, 88
interruptions, 175
  impact of, 110–111
  reducing, 111
interviews, 49
invisible pain, 237–238
iron triangle, project management, 136
irrational exuberance, 150
Ishikawa diagram, 231–232
ISO (International Organization for Standardization), 250
iteration
  contingency buffers in agile, 118
  on designs, 76–84, 269–270
  at high level of abstraction, 79–84
  rapid visual, 81–83
  on requirements, 25–28, 84, 269–270
  on user interface architecture, 92
iteration backlog, 66
iterative development, 117. *See also* agile software development

J

job stories, 25
Jones, Capers, 116–117, 142
Juran, Joseph M., 190
just-in-time documentation, 30–31

K

Kaner, Cem, 3
Kerth, Norman L., 76–77, 252
key performance indicators, 239
Kiviat diagram, 138–140
knowledge. *See also* information culture
  acquisition of, 241–242, 269
  cost of recording, 33–37, 269
  is not zero-sum, 159–163, 271
  lack of, 211–212
  learning curve, 242–243
  sharing of, 159–162
  transfer, 161–162
Kodak, 2, 4, 161, 254–255
Kulak, Daryl, 23, 244

L

large workshop groups, problems with, 57–61, 270
large-scale software failures, 206
late defect detection, 197–198
learning. *See also* knowledge
  continuous, 160–161. *See also* information culture
  merits of, 257–258
  from peer reviews, 162, 235
  sessions, 242
learning curve, 169, 242–243
left, shifting quality to the, 198–199
Leonard, Andrew, 179
lessons learned, 254–255, 267
Lessons Learned in Software Testing (Kaner, Bach, and Pettichord), 3

life cycle stage, containing defects to, 199–200
lightweight documentation, 30–31
lint, 219
Liskov, Barbara, 74
Lister, Tim, 17

M

MacKay, Jory, 111
maintainability, 89
maintenance, software, 221–223
  adaptive, 221
  categories of, 221
  corrective, 221
  perfective, 221
  preventive, 221
  technical debt and, 221–222
Making Process Improvement Work (Potter and Sakry), 266
management, 244, 245, 272
  behaviors, 240–241
  by Businessweek, 228–233, 272
  congruence, 155
  expectation, 15, 104–105
  influence on culture, 154, 265
  product, 43, 53–54
  project. *See* project management
  requirements, 9, 220
  rights and responsibilities of, 172
  senior, 183, 184
  steering by gentle pressure, 239–240, 272
managing upward, 240–241
Manifesto for Agile Software Development, 36, 226, 253
Manns, Mary Lynn, 262
Manshreck, Tom, 3
manufacturability, 89
manuscript reviewers, 131, 166
maps
  dialog, 82–83
  feature road, 66
  user story, 66
market requirements document (MRD), 15
marketing, 15, 38, 43, 45, 53, 54
maturity levels, process, 163, 187, 226
McConnell, Steve, 3, 127, 142, 161
meetings
  daily scrum, 59
  peer review, 215, 233–234
  virtual, 175, 176
mentoring, 161
methodology, 229, 245, 246–247
lightweight, 226
metrics
  process improvement, 239
  software, 126–127, 239
Microsoft, 179
work effort, 112, 126, 203, 255
Microsoft PowerPoint, 83
Microsoft Project, 218
Miller, Roxanne, 90
Minott, Zachary, 5
mistakes, software engineering
  avoiding, 241–244, 272
  danger of repeating, 256–258, 272
mistakes, user, 86–87
mitigation of risk, 144–145
mockup, 78–79
modeling
  abstraction in, 79–84
  designs, 81
  dialog map, 82–83
  iteration during, 79–84
  notations for, 41, 83–94
  requirements, 84
  tools, 83–84, 218
  visual, 40–41, 84
more with less, doing, 167
multifactor authentication, 91
multitasking, 109–111
Myers, Glenford, 74
Mythical Man-Month, The (Brooks), 76

N

Nagappan, Raj, 24
natural language, 40
needs versus solutions, 146
negotiation, 121–123, 138
  commitments, 105
  data and, 121–123
  principled, 123–125
  of project priorities,138
Next Steps
  requirements, 69
  design, 100–101
  project management, 151
  culture and teamwork, 187–188
  quality, 224
  process improvement, 259
Nichols, Bill, 170
Nielsen, Jakob, 100
no silver bullets, 229
nondogmatic, being, 246–252
nonfunctional requirements, 8, 88–92. *See also* quality attributes
  architecture and, 92
  as constraints, 88
  optimizing, 90–91
  prioritizing, 91
  trade-offs between, 88, 91, 93

O

Obamacare, 206–207
observation, of users, 50
offices, 174–175, 176–177
one hundred million dollar syndrome, 206–207
one hundred percent done, 133
optimizing nonfunctional requirements, 90–91
Oregon healthcare system, 206–207
owner, product, 10, 28, 36–37, 43, 54, 114, 170, 183

P

Page-Jones, Meilir, 200
pain, points of, 228, 236–238, 272
  as change motivator, 236
  examples of, 236–237
  invisible, 237–238
paper prototypes, 78–79
Parnas, David, 74
partial credit, 132, 134–135
passaround, 215
patches, code, 221
path, critical, 129–132
  defined, 129–131
  staying clear of, 131–132
Pearls from Sand, 16
peer deskcheck, 215
peer reviews, 198, 213–217, 233–235, 244
  benefits of, 162, 214, 235, 244
  best practices for code, 216
  cultural aspects of, 215–217
  guidelines for, 216–217
  inspections, 42, 215, 234
  knowledge exchange through, 162, 235
  personal payoff from, 235
  of requirements, 42–43
  return on investment from, 233–234
  ways to perform, 215–216
  when to perform, 205–206
people management, 104
perfective maintenance, 221
performance, 89, 91
  of individual software developers, 169–170
personas, 20
PERT charts, 129–131
Pettichord, Bret, 3
pilot
  activities for process improvement, 240, 263
  system, 76
Plan-Do-Check-Act cycle, 262–263
Planguage, 90

planning
- done, definition of, 132–134
- for process improvement, 262
- for quality, 189–190
- worksheets, 125, 133

plans, action, 262
- template for, 266–267

Podeswa, Howard, 33
points of pain. *See* pain, points of
portability, 89, 91
positional power, 123, 128, 148
positives, false, 219
postmortems. *See* retrospectives
post-project reviews. *See* retrospectives
power plays, 210
prefactoring, 94
Prefactoring (Pugh), 72
presentations, 161
pressure, gentle, 239–240, 272
pretending, in software, 149–151, 271
- fantasy thinking, 149
- game playing, 150–151
- irrational exuberance, 150

preventive maintenance, 221–223
Prime Directive for retrospectives, 253
principled negotiation, 123–125
priorities, 106, 110, 113, 131, 140, 148
- negotiating project, 138

prioritization, requirements, 32, 44, 138, 269
- criteria for, 62–64
- decibel, 61–64, 270
- techniques for, 62

prioritizing
- changes, 264
- risks, 141, 143, 144, 145

problems
- requirements, 12
- design, 75
- project management, 108
- culture and teamwork, 158
- quality, 194
- process improvement, 228

process
- assessment of, 262
- benefits of, 226–227
- choosing an appropriate, 229–233
- circumventing, 212–213
- defined, 245–246
- fear of, 226–227
- internalizing, 246
- judgment and experience versus, 244
- maturity levels, 163, 187, 226
- negative connotations surrounding, 226
- nondogmatism and, 246–247
- prioritizing changes, 264
- rhythm versus, 245–246
- scaling up, 177–180, 271
- tools versus, 219–221
- verification of, 264

process improvement, 225–228
- action plan for, 262
- challenges of, 226–227
- culture change for, 265, 272
- cycle for, 262–263
- defined, 225–226
- first steps for, 228
- gentle pressure, relentlessly applied, 239–240, 272
- key performance indicators, 239
- management behaviors, 265, 272
- next steps for, 259
- objective of, 225
- policies, 181
- retrospectives and, 254, 255–256
- tracking progress of, 240
- treating as a project, 240, 266–267

product
- backlog, 29, 30, 53, 54, 65, 66, 106, 119, 192
- champion, 53–54
- defined, 4
- making easy to use, 84–87
- manager, 53–54

owner, 10, 28, 36–37, 43, 54, 114, 170, 183
requirements, 18. *See also* requirements
productivity, 170
increasing, 166–168, 271
individual developer, 169–170
quality and, 200–202, 271–272
tools, 169
Programming Pearls (Bentley), 3
progressive refinement of requirements detail, 26–27
project
buffer, 117
charter, 21
defined, 103
dimensions, 136–140, 271
estimation. *See* estimation
friction, 109–113. *See also* friction, project
management. *See* project management
manager. *See* project manager
planning, 132–134, 189–190, 262
retrospectives. *See* retrospectives
priorities, 32, 44, 61–64, 138, 269
sponsor, 43, 122–123
treating change as, 240, 266–267
project management, 103–108
change management in, 106
commitment management in, 105
communication management in, 106
contingency buffers in, 116–121
contract management in, 107
critical path, 129–132, 271
defined, 103–104
dependency management in, 106–107
done, definition of, 122–136, 271
estimates. *See* estimates
expectation management in, 104–105
first steps for, 108–109
five-dimensional analysis in, 136–140, 271
iron triangle, 136
negotiation, 121–123. *See also* negotiation
next steps for, 151
people management in, 104
pretending and wishful thinking in, 149–151, 271
removing barriers in, 107–108
requirements management in, 104
resource management in, 106
risk. *See* risk; risk management
supplier management in, 107
iron triangle, 136
task management in, 105
triple constraint, 136, 190–191
uncertainty in, 116–121, 271
voice of customer in, 145–148
Project Management Institute, 151
project management iron triangle, 136
project manager, 105–108, 137–138
crap-gap scenarios and, 209
defined, 103
five-dimensional analysis by, 137–141
process assessment activity and, 238
requirements planning and, 36–37, 43, 67–68
rights and responsibilities of, 172
unreasonable, 186
use of experience-based knowledge by, 258
project retrospectives. *See* retrospectives
project priorities, negotiating, 138
Project Retrospectives (Kerth), 252
project tracking, 135–136, 240
proof-of-concept prototype, 78
prototypes, 51, 77–78, 198
defined, 77
design and, 71, 77–79
horizontal, 78–79
paper, 78–79
power of, 77–78
proof-of-concept, 78
requirements and, 14, 15, 28, 40, 51, 57

user interface, 73–74, 78–79
vertical, 78
Pugh, Ken, 73, 93, 94

Q

quality, 136–138, 189–193
as a project dimension, 271
assurance, 198
attributes. *See* quality attributes
building in, 192–193, 200–202, 205–206
control, 198
cost of, 195–197, 203–205
defined, 189–190
effect on productivity, 200–202, 209, 271–272
external, 192
first steps for, 194–195
internal, 192
metrics for, 126–128, 255
next steps for, 224
of requirements, 41–44, 269
requirements quality checklist, 42–45
shifting to the left, 198–199
quality attributes, 270
architecture and, 92
designing for, 90–91
lists of, 88–92
optimizing, 88–92
prioritizing, 90–91
specifying, 90
trade-offs between, 88, 91, 93
quality factors. *See* quality attributes
"Quality is free" (Crosby), 203
quality of service requirements. *See* quality attributes

R

rapid visual iteration, 81–83
recoding, avoiding, 92–94, 270
recording knowledge, cost of, 33–37, 269
redesign, avoiding, 92–94, 270
refactoring, 93, 222
refining requirements details, 26–27
regulatory compliance, 63
reliability, 89, 91
repair, cost of, 195–197
reports, CHAOS, 257–258
representation, of requirements, 31, 40–41
requests, from customers, 145–148
requirements, 7–11, 269
agile. *See* agile requirements
ambiguity in, 31, 40, 44, 67–68
analysis, 9
assumed, 55–57
audiences for, 38–39
business, 8, 15, 21, 26, 63
categories of, 8–9
characteristics of high-quality, 42–45
communication of, 41, 269
communication pathways, 52–54
complete, 13–18, 269
conflicting, 146
conformance to, 88, 190
defined, 8
detail in, 26–27, 30, 45–46
development. *See* development, requirements
documenting, 33–37. *See also* documenting requirements
elicitation. *See* elicitation, requirements
emergent, 27–29
external interface, 8
first steps for, 11–12
functional, 8, 27–28, 31
good enough, 44–46, 269
growth, 68, 116–121
implied, 55–57
iteration on, 25–28, 84, 269
management, 9, 104
management tools, 218, 220
modeling of, 40–41, 84

- next steps for, 69
- nonfunctional, 8, 28–29. *See also* nonfunctional requirements; quality attributes
- peer reviews of, 42–43
- perfect, 44
- prioritization. *See* prioritization, requirements
- and prototypes, 14, 15, 28, 40, 51, 57
- quality checklist, 42–45
- quality of, 41–44, 269
- representation of, 31, 40–41
- specification. *See* specification, requirements
- solution, 9, 17, 46, 54
- status tracking, 135
- testing, 198–199, 220
- transition, 8
- types of, 8
- usage-centric approach to, 22–23, 269
- user, 8, 20, 25, 26, 49, 54, 60, 147
- validation, 9
- ways to represent, 40–41

Requirements by Collaboration (Gottesdiener), 59
requirements management (RM) tools, 218, 220
Requirements Modeling Language (RML), 41
resource management, 106
resource usage, 89
responsibilities and rights. *See* rights and responsibilities
retrospectives, 127, 252–256
- on agile projects, 253
- characteristics of, 252–253
- emotional component of, 254
- post-retrospective activities, 255–257
- Prime Directive, 253
- risks gathered from, 142, 252, 254, 255
- structure of, 254–255

return on investment (ROI)
- from peer reviews, 233–234
- from training, 240

reusability, 89, 91
reuse, 94
reverse engineering, 33
reviewers, manuscript, 131, 166
reviews. *See* peer reviews
rework
- cost of, 203
- design to reduce, 92–94, 270
- requirements development to reduce, 12, 13, 44, 56–57
- quality problems and, 200, 202–203
- time required for, 112–113, 125, 126–127
- software process improvement to reduce, 225, 230, 234

rhythms, 245–246
Rice, David, 223
rights and responsibilities, 171–173, 235, 271
- of autonomous teams, 173
- of customers, 172
- of developers, 172
- of project managers or sponsors, 172

Rising, Linda, 262
risk
- absorbing, 144
- analysis, 143–145
- avoidance, 144
- condition–consequence format, 143–144
- contingency plans for, 144
- and estimation, 115
- exposure, 144–145
- gathered from retrospectives, 142, 252, 254, 255
- identifying, 141–143
- lists, 142–145
- management. *See* risk management
- mitigation, 144–145
- prioritizing, 144
- transferring, 144

risk analysis and contingency buffers, 117
risk management, 105, 140–141, 143–145, 271

RML (Requirements Modeling Language), 41
Robertson, James, 47, 50
Robertson, Suzanne, 47, 50
robustness, 89, 91
root cause analysis, 228–233
  benefits of, 228–230
  cause-and-effect diagram, 231–232
  example of, 230–232
root causes of problems
  requirements, 12
  design, 76
  project management, 109
  culture and teamwork, 159
  quality, 194
  process improvement, 228
Rothman, Johanna, 196
rules, business, 20, 33, 35–36, 38, 45, 50, 179
rushing to code, 211

S

safety, 89, 91
Santayana, George, 256–257
scalability, 89
scaling up processes and methods, 177–180, 271
  communication clashes, 180
  processes and tools, 178–179
  specialization, 179
schedule, as a project dimension, 136–140, 271
schedule estimation, 111, 136–140
scope, 26–68, 270
  ambiguous requirements and, 31, 40, 44, 67–68
  as a project dimension, 136–140, 271
  changing, 67–68, 136–140
  on contracted projects, 68, 107
  creep, 56–57, 64–68, 270
  defined, 65
  horizontal, 59
  project, 66–67
  representing, 65–66
  vertical, 59
Scrum, 54, 59, 182–184, 246
scrum, daily, 59
"Scrum Guide, The" (Schwaber and Sutherland), 246
security, 89, 91
senior management, 183, 184
serviceability, 89
shared vision, 15–17
Shewhart Cycle, 262–263
shrink-to-fit philosophy, 247–252, 272
silver bullets, no, 229
simple matter of programming (SMOP), 212
simulation, 83
size, metrics for, 126, 255
slack time, 119, 120, 130–131
Software Engineering Code of Ethics and Professional Practice, The, 171
software engineering culture, healthy, 153–155, 158, 159, 160, 163
Software Engineering at Google (Winters, Manshreck, and Wright), 3
Software Engineering Institute, 170
Software Estimation (McConnell), 127
software metrics, 126–127, 239
software process improvement. *See* process improvement
software requirements specification (SRS), 11, 15, 30, 39, 46
  template for, 247–249
Software Requirements (Wiegers and Beatty), 244
solution
  defined, 4
  versus needs, 146
  requirements, 9, 17, 46, 54
  specialization, 179
specification, requirements, 9, 17, 29
  goal of, 15, 39, 45
  template for, 247–249
SPI (software process improvement). *See* process improvement

spike, 77–78
Spolsky, Joel, 110
sponsor, project, 43, 122–123
sprint. *See also* iteration
  backlog, 66
  goal, 173
SRS. *See* software requirements specification (SRS)
staff, as a project dimension, 136–140, 271
stakeholders, 42–43, 148. *See also* customers
  analysis of, 18–20
  conflicting requirements from, 146
  defined, 17
  examples of, 18, 19
  interests of, 4, 17–21, 145–149
  as source of requirements and constraints, 13–14
standards, conformance to, 88, 190
Standish Group, The, 170, 257
static code analysis tools, 219
steering
  gentle pressure, relentlessly applied, 239–240, 272
  managing upward, 240–241
stories, user, 24–25, 29–31, 48, 62, 77–78
story maps, 66
story points, 118–119
structured analysis and design, 41
subdomains, of requirements engineering, 9–10
supplier management, 107
support, versus commitment, 182
surrogate representatives, 146–147
surveys, 50
sustainabiity, 89

T

task
  dependencies, 129–130
  management, 105
  planning, 133–134
  sequence of, 130
  switching, 109–111
  tracking completion, 134
team
  alignment of, 157–158
  contract, 107
  review, 215
  virtual, 175–176
teamwork, 153–158. *See also* culture
  alignment and, 157–158
  commitment in, 163–166, 271
  communication and collaboration in, 173–177, 271
  culture and, 271
  first steps for, 158–159
  mentoring, 161
  next steps for, 187–188
  peer reviews and, 233–235, 244
technical debt, 93, 221–223
  conscious, 222–223
  refactoring and, 93, 222
  repaying, 168, 222
  software maintenance and, 221–222
technical ransom, 160
technical reviews. *See* peer reviews
technical talks, 161
telepathy, 55–57, 270
templates, 39, 266–267
  action planning, 266–267
  collection of examples, 162
  job story, 25
  populating, 249
  shrinking to fit, 247–252, 272
  software requirements specification, 247–249
  use cases, 23, 25, 39
  user story, 24
  vision and scope document, 39, 65–66
  vision statement, 16–17
test-driven development, 27, 198–199
tester, 43
testing, 115, 191, 198, 202, 203

acceptance, 15, 29, 32, 40, 48, 197, 198–199
automated, 220
A/B, 78
pushing to the left, 198–199
regression, 115, 209, 223
requirements, 78, 198–199, 220
unit, 209
usability, 78, 85
tests, acceptance, 198–199
thinking, merits of, 258
time
metrics for, 126, 255
needed for process improvement, 227, 239, 240, 255
waste of, 62, 79, 92, 99, 114–115, 167–168, 193, 208, 236, 255–256
time zones, 174–175
tools, 217–220, 251, 272
for modeling, 83–84, 218
processes and, 178–179
for quality, 217–220
requirements management, 218, 220
for security, 98
traceability, of requirements, 44
tracking
defects, 199–200
project, 135–136, 240
requirements status, 135
task completion, 134
trade-offs
project dimensions,137
quality attributes, 88, 91, 93
training, 4, 161, 271
versus achievement, 240
cost of quality and, 204
development of, 125, 133
to improve capability, 169, 271
limitations of, 181, 183
presentations and, 161
for process improvement, 240
and transition requirements, 9
return on investment from, 240
transition requirements, 8
triple constraint, 136, 190–191

U

UI design. *See* user interface (UI) design
UML (Unified Modeling Language), 41, 84
understanding, shared, 15–17
Unified Modeling Language (UML), 41, 84
unreasonable people, 185–187, 271
upgradability, 89
usability, 89, 91
usage, product, 8, 21–23, 63
versus features, 22–23
usage-centric
design, 200
requirements approach, 21–25, 269
use cases, 22–23, 29, 31, 39, 66, 244
user classes, 63
disfavored, 19
favored, 20, 63
user experience (UX) design, 73–74, 99–100, 270
user interface (UI) design, 73–74, 84–87, 270
architecture of, 92
detailed, 73, 74, 81–82
guidelines for, 99–100
iterating on, 84–87
making easy to use, 84–87
optimizing, 88–92
and prototypes, 73–74, 78–79
user mistakes, 86–87
user representatives, 43, 52–54
user requirements, 8, 20, 25, 26, 49, 54, 60, 147. *See also* use cases; user stories
user stories, 29, 30, 48, 62
spikes, 77–78
templates for, 24–25
user story maps, 66

users
classes of, 63
direct, 19
indirect, 19
observation of, 50
as product champions, 52–54

V

value, to customers, 76, 91, 92, 93
variation of developer performance, 169–170
velocity, 112, 118–119, 126
verifiability, 89
verification
of new processes, 263
of requirements, 44, 135
vertical prototypes, 78
vertical scope, 59
virtual teams, 175–176
vision
product, 16–17, 21
shared, 15–17, 269
vision and scope document, 16–17, 21, 39, 65–66
vision statement, 16–17, 21
template for, 16
visual models, 40–41, 81–84
voice of the customer, 145–148

W

walkthrough, 215
waste of time, 62, 79, 92, 99, 114–115, 167–168, 193, 208, 236, 255–256
Weinberg, Gerald, 190
Wideband Delphi, 105
wikis, 50
Winters, Titus, 3
wishful thinking, 149
work effort, 126–127
metrics, 112, 126, 203, 255
worksheets, planning, 125, 133
workshops, elicitation, 49, 57–61
Wright, Hyrum, 3
written documentation. *See* documentation

X-Y-Z

zero defects, 190
zero-sum, knowledge is not, 159–163, 271