

## Software Engineering

Software engineering is as much about teamwork as it is about technology. This introductory textbook covers both. For courses featuring a team project, it offers tips and templates for aligning classroom concepts with the needs of the students' projects. Students will learn how software is developed in industry by adopting agile methods, discovering requirements, designing modular systems, selecting effective tests, and using metrics to track progress. The book also covers the why's behind the how-to's, to prepare students for advances in industry practices. The chapters explore ways of eliciting what users really want, how clean architecture divides and conquers the inherent complexity of software systems, how test coverage is essential for detecting the inevitable defects in code, and much more. Ravi Sethi provides real-life case studies and examples to demonstrate practical applications of the concepts. Online resources include sample project materials for students, and lecture slides for instructors.

**Ravi Sethi** is Laureate Professor of Computer Science at the University of Arizona, USA, and is an ACM fellow. He co-authored *Compilers: Principles, Techniques, and Tools*, popularly known as the “dragon” book, and launched Avaya Labs.

Cambridge University Press & Assessment  
978-1-316-51194-7 – Software Engineering  
Ravi Sethi  
Frontmatter  
[More Information](#)

---

# Software Engineering

## Basic Principles and Best Practices

RAVI SETHI

*University of Arizona*



CAMBRIDGE  
UNIVERSITY PRESS



Shaftesbury Road, Cambridge CB2 8EA, United Kingdom  
One Liberty Plaza, 20th Floor, New York, NY 10006, USA  
477 Williamstown Road, Port Melbourne, VIC 3207, Australia  
314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India  
103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment, a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of education, learning and research at the highest international levels of excellence.

[www.cambridge.org](http://www.cambridge.org)

Information on this title: [www.cambridge.org/highereducation/isbn/9781316511947](http://www.cambridge.org/highereducation/isbn/9781316511947)

DOI: 10.1017/9781009051811

© Ravi Sethi 2023

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press & Assessment.

First published 2023

Printed in the United Kingdom by TJ Books Limited, Padstow Cornwall 2023

*A catalogue record for this publication is available from the British Library.*

*Library of Congress Cataloging-in-Publication Data*

Names: Sethi, Ravi Mohan, 1947-author.

Title: Software engineering : basic principles and best practices /  
Ravi Sethi, University of Arizona.

Description: Cambridge, United Kingdom ; New York, NY : Cambridge  
University Press, 2023. | Includes bibliographical references and index.

Identifiers: LCCN 2022034132 (print) | LCCN 2022034133 (ebook) |  
ISBN 9781316511947 (hardback) | ISBN 9781009051811 (epub)

Subjects: LCSH: Software engineering.

Classification: LCC QA76.758 .S45834 2023 (print) | LCC QA76.758 (ebook) |  
DDC 005.1–dc23/eng/20220822

LC record available at <https://lcn.loc.gov/2022034132>

LC ebook record available at <https://lcn.loc.gov/2022034133>

ISBN 978-1-316-51194-7 Hardback

Additional resources for this publication at [www.cambridge.org/sethi](http://www.cambridge.org/sethi).

Cambridge University Press & Assessment has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

## Brief Contents

	<i>Preface</i>	<i>page xv</i>
<b>1</b>	<b>Introduction</b>	1
<b>2</b>	<b>Software Development Processes</b>	24
<b>3</b>	<b>User Requirements</b>	64
<b>4</b>	<b>Requirements Analysis</b>	100
<b>5</b>	<b>Use Cases</b>	125
<b>6</b>	<b>Design and Architecture</b>	142
<b>7</b>	<b>Architectural Patterns</b>	172
<b>8</b>	<b>Static Checking</b>	203
<b>9</b>	<b>Testing</b>	222
<b>10</b>	<b>Quality Metrics</b>	253
<b>Appendix</b>	<b>A Team Project</b>	300
	<i>Notes</i>	314
	<i>References</i>	325
	<i>Index</i>	337

Cambridge University Press & Assessment  
978-1-316-51194-7 – Software Engineering  
Ravi Sethi  
Frontmatter  
[More Information](#)

---

# Contents

	<i>Preface</i>	xv
<b>1</b>	<b>Introduction</b>	<b>1</b>
	1.1 What Is Software Engineering?	1
	1.1.1 Definition of Software Engineering	2
	1.1.2 A Tale of Two Companies	4
	1.2 The Requirements Challenge	5
	1.2.1 Identifying Users and Requirements	6
	1.2.2 Dealing with Requirements Changes	7
	1.3 Software Is Inherently Complex	8
	1.3.1 Sources of Complexity	8
	1.3.2 Architecture: Dealing with Program Complexity	9
	1.4 Defects Are Inevitable	11
	1.4.1 Fix Faults to Avoid Failures	11
	1.4.2 Introduction to Testing	12
	1.4.3 Black-Box and White-Box Testing	13
	1.5 Balancing Constraints: The Iron Triangle	13
	1.5.1 Scope. Cost. Time. Pick Any Two!	14
	1.6 Social Responsibility	15
	1.6.1 Case Study: The Volkswagen Emissions Scandal	15
	1.6.2 The ACM Code	15
	1.6.3 Case Study: The Therac-25 Accidents	16
	1.6.4 Lessons for Software Projects	17
	1.7 Conclusion	19
	Further Reading	20
	Exercises	20
<b>2</b>	<b>Software Development Processes</b>	<b>24</b>
	2.1 Processes and Values Guide Development	24
	2.1.1 What Is a Process?	25
	2.1.2 Two Development Cultures: Plan versus Grow	26
	2.1.3 Role Models: Unix Culture and Agile Values	28
	2.1.4 Selecting a Process Model	30

2.2	Structuring Teamwork: The Scrum Framework	32
2.2.1	Overview of Scrum	32
2.2.2	Scrum Roles	34
2.2.3	Scrum Events	34
2.2.4	Scrum Artifacts	36
2.2.5	Summary	37
2.3	Agile Development: Extreme Programming	37
2.3.1	Listening to Customers: User Stories	38
2.3.2	Testing: Make It Central to Development	39
2.3.3	When to Design?	41
2.3.4	A Scrum+XP Hybrid	43
2.3.5	Summary	44
2.4	Limitations of Waterfall Processes	44
2.4.1	The Perils of Big-Bang Integration and Testing	45
2.4.2	The Waterfall Cost-of-Change Curve	46
2.4.3	Managing the Risks of Waterfall Processes	47
2.4.4	Summary	49
2.5	Levels of Design and Testing: V Processes	49
2.5.1	Overview of V Processes	50
2.5.2	Levels of Testing, from Unit to Acceptance	51
2.5.3	Summary	51
2.6	Additional Project Risks	51
2.6.1	Rough Risk Assessment	53
2.6.2	Netscape 3.0: A Successful Iterative Project	53
2.6.3	Netscape 4.0: A Troubled Project	54
2.7	Risk Reduction: The Spiral Framework	55
2.7.1	Overview of the Spiral Framework	56
2.7.2	Summary	58
2.8	Conclusion	58
	Further Reading	60
	Exercises	60
<b>3</b>	<b>User Requirements</b>	64
3.1	What Is a Requirement?	64
3.1.1	The Basic Requirements Cycle	65
3.1.2	Case Study: Requirements Challenges	66
3.1.3	Kinds of Requirements	67
3.2	Developing Requirements and Software	69
3.2.1	Agile Methods Validate Working Software	69
3.2.2	Case Study: An Agile Emphasis on Requirements	70
3.2.3	Plan-Driven Methods Validate a Specification	72
3.3	Eliciting User Needs	73
3.3.1	A Classification of Needs	73
3.3.2	Accessing User Needs	75



		76
3.3.3	Case Study: Intuit’s Design for Delight	76
3.4	Writing Requirements: Stories and Features	77
3.4.1	Guidelines for Effective User Stories	77
3.4.2	Guidelines for System Features	80
3.4.3	Perspective on User Stories	82
3.5	Writing User-Experience Scenarios	83
3.5.1	Guidelines for User-Experience Scenarios	83
3.5.2	Case Study: A Medical Scenario	84
3.6	Clarifying User Goals	85
3.6.1	Properties of Goals	86
3.6.2	Asking Clarifying Questions	87
3.6.3	Organizing Goals into Hierarchies	89
3.6.4	Contributing and Conflicting Goals	91
3.7	Identifying Security Attacks	93
3.7.1	Attack Trees: Think Like an Attacker	93
3.7.2	Finding Possible Attacks	93
3.8	Conclusion	95
	Further Reading	97
	Exercises	97
<b>4</b>	<b>Requirements Analysis</b>	<b>100</b>
4.1	A Checklist Approach	100
4.2	Relative Estimation: Iteration Planning	103
4.2.1	Anchoring Can Bias Decisions	103
4.2.2	Agile Story Points	104
4.2.3	Velocity of Work	106
4.3	Structured Group Consensus Estimates	106
4.3.1	Wideband Delphi and Planning Poker	107
4.3.2	The Original Delphi Method	108
4.4	Balancing Priorities	110
4.4.1	Must-Should-Could-Won’t (MoSCoW) Prioritization	110
4.4.2	Balancing Value and Cost	111
4.4.3	Balancing Value, Cost, and Risk	112
4.5	Customer Satisfiers and Dissatisfiers	113
4.5.1	Kano Analysis	114
4.5.2	Classification of Features	115
4.5.3	Life Cycles of Attractiveness	117
4.5.4	Degrees of Sufficiency	118
4.6	Plan-Driven Estimation Models	118
4.6.1	How Are Size and Effort Related?	119
4.6.2	The Cocomo Family of Estimation Models	121
4.7	Conclusion	122
	Further Reading	122
	Exercises	123

<b>5</b>	<b>Use Cases</b>	125
5.1	Elements of a Use Case	125
5.1.1	Actors and Goals Outline a System	125
5.1.2	Flows and Basic Flows	127
5.2	Alternative Flows: Conditional Behaviors	127
5.2.1	Specific Alternative Flows	129
5.2.2	Extension Points	130
5.2.3	Bounded Alternative Flows	131
5.3	Writing Use Cases	132
5.3.1	A Template for Use Cases	133
5.3.2	From Actor Intentions to System Interactions	134
5.3.3	How to Build Use Cases	135
5.4	Use-Case Diagrams	136
5.4.1	Diagrams Highlight Goals and Actors	136
5.5	Relationships between Use Cases	137
5.5.1	Subflows	137
5.5.2	Inclusion of Use Cases	137
5.5.3	Extensions of Use Cases	138
5.6	Conclusion	139
	Further Reading	140
	Exercises	140
<b>6</b>	<b>Design and Architecture</b>	142
6.1	The Role of Architecture	142
6.1.1	What Is Architecture?	143
6.1.2	Design Includes Architecture	144
6.1.3	What Is a Good Software Architecture?	144
6.2	Designing Modular Systems	145
6.2.1	The Modularity Principle	145
6.2.2	Coupling and Cohesion	148
6.2.3	Design Guidelines for Modules	149
6.3	Class Diagrams	150
6.3.1	Representing a Class	150
6.3.2	Relationships between Classes	153
6.4	Architectural Views	156
6.4.1	The 4+1 Grouping of Views	156
6.4.2	Structures and Views	160
6.5	Describing System Architecture	162
6.5.1	Outline for an Architecture Description	162
6.5.2	System Overview of a Communications App	164
6.5.3	A Development View: Module Hierarchies	165
6.6	Conclusion	167
	Further Reading	168
	Exercises	168

<b>7</b>	<b>Architectural Patterns</b>	172
	7.1 Software Layering	172
	7.1.1 The Layered Pattern	174
	7.1.2 Design Trade-offs	176
	7.2 Three Building Blocks	178
	7.2.1 The Shared-Data Pattern	179
	7.2.2 Observers and Subscribers	180
	7.3 User Interfaces: Model-View-Controller	181
	7.3.1 Design Decisions	182
	7.3.2 The Basic Model-View-Controller Pattern	183
	7.3.3 Keep Views Simple	185
	7.4 Dataflow Architectures	186
	7.4.1 Dataflow Pipelines	186
	7.4.2 Dataflow Networks	189
	7.4.3 Unbounded Streams	190
	7.4.4 Big Dataflows	191
	7.5 Connecting Clients with Servers	192
	7.5.1 The Client-Server Pattern	193
	7.5.2 Deploying Test Servers	194
	7.5.3 The Broker Pattern	196
	7.6 Families and Product Lines	197
	7.6.1 Commonalities and Variabilities	197
	7.6.2 Software Architecture and Product Lines	198
	7.6.3 Economics of Product-Line Engineering	198
	7.7 Conclusion	199
	Further Reading	199
	Exercises	200
<b>8</b>	<b>Static Checking</b>	203
	8.1 Architecture Reviews	203
	8.1.1 Guiding Principles for Architecture Reviews	204
	8.1.2 Discovery, Deep-Dive, and Retrospective Reviews	206
	8.2 Conducting Software Inspections	207
	8.2.1 The Phases of a Traditional Inspection	207
	8.2.2 Case Study: Using Data to Ensure Effectiveness	209
	8.2.3 Organizing an Inspection	211
	8.3 Code Reviews: Check Intent and Trust	212
	8.3.1 Invested Expert Reviewers	212
	8.3.2 Reviewing Is Done within Hours	213
	8.4 Automated Static Analysis	214
	8.4.1 A Variety of Static Checkers	215
	8.4.2 False Positives and False Negatives	218
	8.5 Conclusion	218

	Further Reading	220
	Exercises	220
<b>9</b>	<b>Testing</b>	222
	9.1 Overview of Testing	222
	9.1.1 Issues during Testing	223
	9.1.2 Test Selection	225
	9.1.3 Test Adequacy: Deciding When to Stop	225
	9.1.4 Test Oracles: Evaluating the Response to a Test	226
	9.2 Levels of Testing	227
	9.2.1 Unit Testing	228
	9.2.2 Integration Testing	229
	9.2.3 Functional, System, and Acceptance Testing	231
	9.2.4 Case Study: Test Early and Often	232
	9.3 Code Coverage I: White-Box Testing	233
	9.3.1 Control-Flow Graphs	233
	9.3.2 Control-Flow Coverage Criteria	235
	9.4 Input Coverage I: Black-Box Testing	238
	9.4.1 Equivalence-Class Coverage	239
	9.4.2 Boundary-Value Coverage	240
	9.5 Code Coverage II: MC/DC	241
	9.5.1 Condition and Decision Coverage Are Independent	242
	9.5.2 MC/DC Pairs of Tests	242
	9.6 Input Coverage II: Combinatorial Testing	245
	9.7 Conclusion	249
	Further Reading	250
	Exercises	250
<b>10</b>	<b>Quality Metrics</b>	253
	10.1 Meaningful Metrics	253
	10.1.1 Metrics Quantify Attributes	254
	10.1.2 Selecting Useful Metrics	255
	10.1.3 Goal-Directed Measurement	257
	10.2 Software Quality	257
	10.2.1 The Many Forms of Software Quality	258
	10.2.2 Measuring Customer Support	260
	10.3 Graphical Displays of Data Sets	261
	10.3.1 Data Sets	261
	10.3.2 Scales of Measurement	262
	10.3.3 Bar Charts Display Data by Category	264
	10.3.4 Gantt Charts Display Schedules	266
	10.4 Product Quality: Measuring Defects	267
	10.4.1 Severity of Defects	268
	10.4.2 Defect-Removal Efficiency	268

	10.4.3 Customer-Found Defects (CFDs)	270
	10.4.4 CFDs Measure Installs, Not Quality	270
10.5	Ops Quality Improvement: A Case Study	272
	10.5.1 How to Improve Software Quality	272
	10.5.2 The Customer Quality Metric	274
	10.5.3 Subgoals: Product and Process Improvement	275
	10.5.4 Measuring Process Improvements	276
10.6	Data Dispersion: Boxplots and Histograms	277
	10.6.1 Medians and Quartiles	277
	10.6.2 Box Plots Summarize Data by Quartile	279
	10.6.3 Histograms of Data Spread	280
10.7	Data Dispersion: Statistics	282
	10.7.1 Variance from the Mean	282
	10.7.2 Discrete Probability Distribution	284
	10.7.3 Continuous Distributions	286
	10.7.4 Introduction to Normal Distributions	286
	10.7.5 Introduction to Student's t-Distributions	287
10.8	Confidence Intervals	287
	10.8.1 Definition of Confidence Interval	288
	10.8.2 If the Population Standard Deviation Is Known	289
	10.8.3 If the Population Standard Deviation Is Unknown	290
10.9	Simple Linear Regression	292
	10.9.1 The Simpler Case: Line through the Origin	293
	10.9.2 Ordinary Least-Squares Fit	294
10.10	Conclusion	296
	Further Reading	298
	Exercises	298
<b>Appendix</b>	<b>A Team Project</b>	<b>300</b>
	A.1 Overview	300
	A.1.1 Goals for a Project	301
	A.1.2 The Team Experience	302
	A.1.3 Coupling the Classroom and Project Experiences	302
	A.2 Project Proposal	304
	A.3 Skeletal System: Status Report 1	306
	A.4 Viable System: Status Report 2	308
	A.5 Comprehensive Final Report	310
	<i>Notes</i>	314
	<i>References</i>	325
	<i>Index</i>	337

Cambridge University Press & Assessment  
978-1-316-51194-7 – Software Engineering  
Ravi Sethi  
Frontmatter  
[More Information](#)

---

## Preface

The selection of content for this book was guided by the following question: What do software engineers really need to know about the subject to be productive today and relevant tomorrow? The discipline continues to evolve, driven by new applications, technologies, and development methods. There is every indication that the evolution will continue during an engineer's career. The IEEE-ACM software engineering curriculum guidelines stress continual learning:

Because so much of what is learned will change over a student's professional career and only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge.<sup>1</sup>

This book therefore focuses on basic principles and best practices. The emphasis is not only on what works, but on why it works. The book includes real-world examples and case studies, where possible. Some classic examples are included for perspective.

Principles endure while practices evolve as the assumptions behind them are reexamined. The principles in the book relate to the intrinsic properties of software and human nature: software is complex, requirements change, defects are inevitable, teams need coordination. Assumptions about how to deal with these intrinsic properties have been tested over the years. Must testing follow coding? Not with test-driven development. The distinction between development and maintenance blurs with an evolving software code base. All assumptions have to be questioned to enable the pace of continuous deployment. What does not change is that design and architecture are the key to managing complexity, iterative agile methods accommodate requirements changes, validation and verification reduce defects, and a healthy balance of structure and flexibility motivates teams and improves performance.

### Content Organization and Coverage

This book is intended for a junior- or senior-level introductory course in software engineering. Students are expected to have enough programming maturity to engage in a team project. They are not expected to have any prior team experience.

The ACM-IEEE guidelines strongly recommend the inclusion of a significant project in a software engineering course. First, systematic engineering methods are intended for problems of complexity and scale. With a significant project, students

get to experience the benefits of engineering concepts and methods. Second, users and teams bring a human dimension to the discipline. Working with a real customer on a project suitable for a team of, say, four provides students with a team experience. Appendix A addresses the challenge of organizing a course with dual tracks for concepts and a project. See also the brief comments in the following paragraphs.

The chapters in this book can be grouped as follows: getting started, what to build, design and architecture, software quality, and metrics.

**Getting Started: Chapters 1–2** Chapter 1 introduces key topics that are explored in the rest of the book: requirements, software architecture, and testing. The chapter also has a section on social responsibility and professional conduct.

Chapter 2 deals with processes, which orchestrate team activities. A team’s culture and values guide activities that are not covered by the rules of a process. Process models tend to focus on specific activities, leaving the rest to the team: Scrum focuses on planning and review events, XP on development practices, V processes on testing, and the Spiral Framework on risk reduction. The chapter discusses how a team can combine best practices from these process models for its project.

**What to Build? Chapters 3–5** Requirements development is iterative, with both agile and plan-driven methods. The difference is that agile methods favor working software to validate what to build, whereas plan-driven methods validate a specification document. Chapter 3 deals with elicitation (discovery) of the wants and goals that users communicate through their words, actions, and emotions. What users say can differ from what they do and feel. To help clarify user goals, the chapter introduces three classes of questions, aimed at identifying user motivations, solution options, and goal quantification. User requirements can be recorded as user stories, system features, or user-experience scenarios. Goal refinement techniques, covered with requirements, also apply to security (attack trees) and measurement (metrics).

The requirements prioritization techniques in Chapter 4 include MoSCoW (must-should-could-won’t), value-cost balancing, value-cost-risk assessment, and Kano analysis. Kano analysis is based not only on what satisfies customers, but on what dissatisfies them. The chapter also includes estimation techniques based on story points for agile methods and on Cocomo formal models for plan-driven methods. Anchoring, which can bias estimates, is a phenomenon to be avoided during both individual and group estimation.

Chapter 5 covers use cases. Use cases can be lightweight if they are developed incrementally, starting with user goals, then adding basic flows, and finally alternative flows as needed.

**Design and Architecture: Chapters 6–7** Architecture is a subset of design, so the following comments carry over to design. Software architecture, Chapter 6, is key to managing the complexity of software. A modular system is built up from units such as classes, so the chapter introduces UML (Unified Modeling Language) class diagrams and includes guidelines for designing modular systems. For system architecture, the chapter introduces views and how to describe a system in terms of key view(s).



A pattern outlines a solution to a problem that occurs over and over again. Chapter 7 covers the following architectural patterns: layering, shared data, observer, publish-subscribe, model-view-controller, client-server, and broker. Client-server architectures enable new software to be deployed to a production system: a load balancer directs some of the incoming traffic to the new software on a trial basis, until the new software is ready to go into production. The ability to add new software to a production system is needed for continuous deployment.

**Software Quality: Chapters 8–10** The combination of reviews (architecture and code), static analysis, and testing is much more effective for defect detection than any of these techniques by themselves. Chapter 8 discusses reviews and static analysis. The focus of the chapter is on static or compile-time techniques.

Chapter 9 is on testing, which is done by running code on specific test inputs. A test set is considered good enough if it meets the desired coverage criteria. For code coverage, the chapter includes statement, branch (decision), and MC/DC coverage. For input-domain coverage, the chapter includes equivalence partitioning and combinatorial testing.

The quality theme continues into Chapter 10, which applies metrics and measurement to the goal of quality assessment and improvement. The chapter introduces six forms of quality: functional, process, product, operations (ops), aesthetics, and customer satisfaction. Ops quality refers to quality after a system is installed at a customer site.

**Metrics: Chapter 10** An alternative long title for this chapter is “the design and use of metrics and measurement, with applications to software quality.” The chapter has roughly three parts. The first part introduces the measurement process, the design of useful metrics, and the graphical display of data sets. The second part deals with metrics for product and ops quality. The third part introduces statistical techniques. Boxplots, histograms, variance, and standard deviation summarize the dispersion of values in data sets. The last two sections on confidence intervals and simple linear regression are mathematical.

**A Team Project: Appendix A** The main challenge in a course with a concepts track and a project track is that the two tracks have somewhat different objectives; for example, the concepts track typically covers multiple process models, while a project embraces a single process. In general, each of the two tracks has its own pace: the concepts track takes many weeks to cover the what, why, and how of processes, requirements, design, and testing, whereas some knowledge of all of these topics is needed early, during the first iteration of a project.

The appendix discusses a hybrid approach, which aligns the tracks initially, while teams are formed and projects are launched. Once the students have enough to start their projects, the coupling between the tracks can be loosened so each track can proceed at its own pace.

## Acknowledgments

My trial-by-software came when Harry Huskey handed me a full box of cards with the assembly code for a compiler for HH-1, his Algol-like language. The code was the only description for either the language or the compiler. Harry then left for a long trip, after giving me free rein to see what I could do. Hello, summer job! The job was at the end of my first year as an undergraduate at IIT Kanpur. At the time, the sum total of my experience was a few short Fortran and assembly programs for the IBM 1620. I spent a formative summer tracing through the code and doing peephole optimization. In retrospect, it would have helped if I had clarified Harry's requirements, tested after every change, asked for a view of the system architecture, and so on.

Fast forward a dozen years to Bell Labs in New Jersey, where I had the good fortune to join Doug McIlroy's department in the Computer Science Research Center. In the Center, Unix was about to be ported from one machine to another for the first time. As a user, I came to appreciate the Unix tools culture, quick iterations, and continual refinement of tools through user feedback. Steve Johnson once lamented "the inability of users to see things" his way and added that "invariably, they were right!"

I also came to appreciate the challenges faced by the Switching business unit in building 99.999 percent reliable telephone switches: each phone call was handled by a concurrent process at each end. Their existing approach had a cast of thousands using variants of waterfall processes. They managed the risks of waterfall methods by freezing requirements and by rigorous continuous validation and verification.

In 2000, Avaya was spun off as a separate company and I went with it to build up Avaya Labs Research. Half of the research staff came from Bell Labs, David Weiss among them. David built up the software technology research department, together with a small group called the Avaya Resource Center (ARC), which served as a bridge to the R&D community in the business units. The mission for the department and the ARC was to "Improve the state of software in Avaya and know it." After David retired, I got a chance to work directly with Randy Hackbarth and John Palframan from the ARC and with Audris Mockus and Jenny Li from the research department. These were the people behind the quality improvement effort described in Section 10.5. I am grateful to them for their briefings and insights about software.

When I joined the University of Arizona in 2014, David generously shared the materials from his software engineering course at Iowa State University. At Arizona, I want to thank Todd Proebsting and David Lowenthal for the opportunity to teach the senior and graduate-level software engineering courses.

For this book, I would like to single out Jon Bentley, Audris Mockus, and Joann Ordille for their insights, encouragement, and comments. Thanks also to the many students in the software engineering courses. The Cambridge University Press team has been most thorough. My thanks to the editors, Emily Watton and Lauren Cowles. Emily's ongoing suggestions have greatly improved the book. Most of all, I want to thank Dianne and Alexandra. Alexandra took over the figures when I could no longer do them.