

Software Engineering

A Practitioner's Approach

Seventh Edition

An abstract graphic composed of several overlapping geometric shapes. It includes a large blue square at the bottom, a smaller red square to its right, and a gold circle positioned above and to the left of the blue square. The background is a textured grey.

Roger S. Pressman

Software Engineering

A PRACTITIONER'S APPROACH

Software Engineering

A PRACTITIONER'S APPROACH

SEVENTH EDITION

Roger S. Pressman, Ph.D.



Boston Burr Ridge, IL Dubuque, IA New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto



SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, SEVENTH EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2010 by The McGraw-Hill Companies, Inc. All rights reserved. Previous editions © 2005, 2001, and 1997. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9

ISBN 978-0-07-337597-7
MHID 0-07-337597-7

Global Publisher: *Raghothaman Srinivasan*
Director of Development: *Kristine Tibbetts*
Senior Marketing Manager: *Curt Reynolds*
Senior Managing Editor: *Faye M. Schilling*
Lead Production Supervisor: *Sandy Ludovissy*
Senior Media Project Manager: *Sandra M. Schnee*
Associate Design Coordinator: *Brenda A. Rolwes*
Cover Designer: *Studio Montage, St. Louis, Missouri*
(USE) Cover Image: © The Studio Dog/Getty Images
Compositor: *Macmillan Publishing Solutions*
Typeface: 8.5/13.5 Leawood
Printer: *R. R. Donnelley Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Pressman, Roger S.

Software engineering : a practitioner's approach / Roger S. Pressman. — 7th ed.

p. cm.

Includes index.

ISBN 978-0-07-337597-7 — ISBN 0-07-337597-7 (hard copy : alk. paper)

1. Software engineering. I. Title.

QA76.758.P75 2010

005.1—dc22

2008048802

*In loving memory of my
father who lived 94 years
and taught me, above all,
that honesty and integrity
were the best guides for
my journey through life.*

ABOUT THE AUTHOR

Roger S. Pressman is an internationally recognized authority in software process improvement and software engineering technologies. For almost four decades, he has worked as a software engineer, a manager, a professor, an author, and a consultant, focusing on software engineering issues.

As an industry practitioner and manager, Dr. Pressman worked on the development of CAD/CAM systems for advanced engineering and manufacturing applications. He has also held positions with responsibility for scientific and systems programming.

After receiving a Ph.D. in engineering from the University of Connecticut, Dr. Pressman moved to academia where he became Bullard Associate Professor of Computer Engineering at the University of Bridgeport and director of the university's Computer-Aided Design and Manufacturing Center.

Dr. Pressman is currently president of R.S. Pressman & Associates, Inc., a consulting firm specializing in software engineering methods and training. He serves as principal consultant and has designed and developed *Essential Software Engineering*, a complete video curriculum in software engineering, and *Process Advisor*, a self-directed system for software process improvement. Both products are used by thousands of companies worldwide. More recently, he has worked in collaboration with *EdistaLearning* in India to develop comprehensive Internet-based training in software engineering.

Dr. Pressman has written many technical papers, is a regular contributor to industry periodicals, and is author of seven technical books. In addition to *Software Engineering: A Practitioner's Approach*, he has co-authored *Web Engineering* (McGraw-Hill), one of the first books to apply a tailored set of software engineering principles and practices to the development of Web-based systems and applications. He has also written the award-winning *A Manager's Guide to Software Engineering* (McGraw-Hill); *Making Software Engineering Happen* (Prentice Hall), the first book to address the critical management problems associated with software process improvement; and *Software Shock* (Dorset House), a treatment that focuses on software and its impact on business and society. Dr. Pressman has been on the editorial boards of a number of industry journals, and for many years, was editor of the "Manager" column in *IEEE Software*.

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He is a member of the IEEE, and Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.

On the personal side, Dr. Pressman lives in South Florida with his wife, Barbara. An athlete for most of his life, he remains a serious tennis player (NTRP 4.5) and a single-digit handicap golfer. In his spare time, he has written two novels, *The Aymara Bridge* and *The Puppeteer*, and plans to begin work on another.

CONTENTS AT A GLANCE

CHAPTER 1 Software and Software Engineering 1

PART ONE THE SOFTWARE PROCESS 29

CHAPTER 2 Process Models 30
CHAPTER 3 Agile Development 65

PART TWO MODELING 95

CHAPTER 4 Principles that Guide Practice 96
CHAPTER 5 Understanding Requirements 119
CHAPTER 6 Requirements Modeling: Scenarios, Information, and Analysis Classes 148
CHAPTER 7 Requirements Modeling: Flow, Behavior, Patterns, and WebApps 186
CHAPTER 8 Design Concepts 215
CHAPTER 9 Architectural Design 242
CHAPTER 10 Component-Level Design 276
CHAPTER 11 User Interface Design 312
CHAPTER 12 Pattern-Based Design 347
CHAPTER 13 WebApp Design 373

PART THREE QUALITY MANAGEMENT 397

CHAPTER 14 Quality Concepts 398
CHAPTER 15 Review Techniques 416
CHAPTER 16 Software Quality Assurance 432
CHAPTER 17 Software Testing Strategies 449
CHAPTER 18 Testing Conventional Applications 481
CHAPTER 19 Testing Object-Oriented Applications 511
CHAPTER 20 Testing Web Applications 529
CHAPTER 21 Formal Modeling and Verification 557
CHAPTER 22 Software Configuration Management 584
CHAPTER 23 Product Metrics 613

PART FOUR MANAGING SOFTWARE PROJECTS 645

CHAPTER 24 Project Management Concepts 646
CHAPTER 25 Process and Project Metrics 666

CHAPTER 26	Estimation for Software Projects	691
CHAPTER 27	Project Scheduling	721
CHAPTER 28	Risk Management	744
CHAPTER 29	Maintenance and Reengineering	761

PART FIVE **ADVANCED TOPICS** 785

CHAPTER 30	Software Process Improvement	786
CHAPTER 31	Emerging Trends in Software Engineering	808
CHAPTER 32	Concluding Comments	833
APPENDIX 1	An Introduction to UML	841
APPENDIX 2	Object-Oriented Concepts	863
REFERENCES		871
INDEX		889

TABLE OF CONTENTS

Preface xxv

CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING 1

1.1	The Nature of Software	3
1.1.1	Defining Software	4
1.1.2	Software Application Domains	7
1.1.3	Legacy Software	9
1.2	The Unique Nature of WebApps	10
1.3	Software Engineering	12
1.4	The Software Process	14
1.5	Software Engineering Practice	17
1.5.1	The Essence of Practice	17
1.5.2	General Principles	19
1.6	Software Myths	21
1.7	How It All Starts	24
1.8	Summary	25
PROBLEMS AND POINTS TO PONDER		25
FURTHER READINGS AND INFORMATION SOURCES		26

PART ONE THE SOFTWARE PROCESS 29

CHAPTER 2 PROCESS MODELS 30

2.1	A Generic Process Model	31
2.1.1	Defining a Framework Activity	32
2.1.2	Identifying a Task Set	34
2.1.3	Process Patterns	35
2.2	Process Assessment and Improvement	37
2.3	Prescriptive Process Models	38
2.3.1	The Waterfall Model	39
2.3.2	Incremental Process Models	41
2.3.3	Evolutionary Process Models	42
2.3.4	Concurrent Models	48
2.3.5	A Final Word on Evolutionary Processes	49
2.4	Specialized Process Models	50
2.4.1	Component-Based Development	50
2.4.2	The Formal Methods Model	51
2.4.3	Aspect-Oriented Software Development	52
2.5	The Unified Process	53
2.5.1	A Brief History	54
2.5.2	Phases of the Unified Process	54
2.6	Personal and Team Process Models	56
2.6.1	Personal Software Process (PSP)	57
2.6.2	Team Software Process (TSP)	58
2.7	Process Technology	59
2.8	Product and Process	60

TABLE OF CONTENTS

2.9	Summary	61
PROBLEMS AND POINTS TO PONDER		62
FURTHER READINGS AND INFORMATION SOURCES		63

CHAPTER 3 AGILE DEVELOPMENT 65

3.1	What Is Agility?	67
3.2	Agility and the Cost of Change	67
3.3	What Is an Agile Process?	68
3.3.1	Agility Principles	69
3.3.2	The Politics of Agile Development	70
3.3.3	Human Factors	71
3.4	Extreme Programming (XP)	72
3.4.1	XP Values	72
3.4.2	The XP Process	73
3.4.3	Industrial XP	77
3.4.4	The XP Debate	78
3.5	Other Agile Process Models	80
3.5.1	Adaptive Software Development (ASD)	81
3.5.2	Scrum	82
3.5.3	Dynamic Systems Development Method (DSDM)	84
3.5.4	Crystal	85
3.5.5	Feature Driven Development (FDD)	86
3.5.6	Lean Software Development (LSD)	87
3.5.7	Agile Modeling (AM)	88
3.5.8	Agile Unified Process (AUP)	89
3.6	A Tool Set for the Agile Process	91
3.7	Summary	91
PROBLEMS AND POINTS TO PONDER		92
FURTHER READINGS AND INFORMATION SOURCES		93

PART TWO MODELING 95**CHAPTER 4 PRINCIPLES THAT GUIDE PRACTICE 96**

4.1	Software Engineering Knowledge	97
4.2	Core Principles	98
4.2.1	Principles That Guide Process	98
4.2.2	Principles That Guide Practice	99
4.3	Principles That Guide Each Framework Activity	101
4.3.1	Communication Principles	101
4.3.2	Planning Principles	103
4.3.3	Modeling Principles	105
4.3.4	Construction Principles	111
4.3.5	Deployment Principles	113
4.4	Summary	115
PROBLEMS AND POINTS TO PONDER		116
FURTHER READINGS AND INFORMATION SOURCES		116

CHAPTER 5 UNDERSTANDING REQUIREMENTS 119

5.1	Requirements Engineering	120
5.2	Establishing the Groundwork	125
5.2.1	Identifying Stakeholders	125

5.2.2	Recognizing Multiple Viewpoints	126
5.2.3	Working toward Collaboration	126
5.2.4	Asking the First Questions	127
5.3	Eliciting Requirements	128
5.3.1	Collaborative Requirements Gathering	128
5.3.2	Quality Function Deployment	131
5.3.3	Usage Scenarios	132
5.3.4	Elicitation Work Products	133
5.4	Developing Use Cases	133
5.5	Building the Requirements Model	138
5.5.1	Elements of the Requirements Model	139
5.5.2	Analysis Patterns	142
5.6	Negotiating Requirements	142
5.7	Validating Requirements	144
5.8	Summary	145
PROBLEMS AND POINTS TO PONDER		145
FURTHER READINGS AND INFORMATION SOURCES		146

CHAPTER 6 REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES 148

6.1	Requirements Analysis	149
6.1.1	Overall Objectives and Philosophy	150
6.1.2	Analysis Rules of Thumb	151
6.1.3	Domain Analysis	151
6.1.4	Requirements Modeling Approaches	153
6.2	Scenario-Based Modeling	154
6.2.1	Creating a Preliminary Use Case	155
6.2.2	Refining a Preliminary Use Case	158
6.2.3	Writing a Formal Use Case	159
6.3	UML Models That Supplement the Use Case	161
6.3.1	Developing an Activity Diagram	161
6.3.2	Swimlane Diagrams	162
6.4	Data Modeling Concepts	164
6.4.1	Data Objects	164
6.4.2	Data Attributes	164
6.4.3	Relationships	165
6.5	Class-Based Modeling	167
6.5.1	Identifying Analysis Classes	167
6.5.2	Specifying Attributes	171
6.5.3	Defining Operations	171
6.5.4	Class-Responsibility-Collaborator (CRC) Modeling	173
6.5.5	Associations and Dependencies	180
6.5.6	Analysis Packages	182
6.6	Summary	183
PROBLEMS AND POINTS TO PONDER		183
FURTHER READINGS AND INFORMATION SOURCES		184

CHAPTER 7 REQUIREMENTS MODELING: FLOW, BEHAVIOR, PATTERNS, AND WEBAPPS 186

7.1	Requirements Modeling Strategies	186
7.2	Flow-Oriented Modeling	187

TABLE OF CONTENTS

7.2.1	Creating a Data Flow Model	188
7.2.2	Creating a Control Flow Model	191
7.2.3	The Control Specification	191
7.2.4	The Process Specification	192
7.3	Creating a Behavioral Model	195
7.3.1	Identifying Events with the Use Case	195
7.3.2	State Representations	196
7.4	Patterns for Requirements Modeling	199
7.4.1	Discovering Analysis Patterns	200
7.4.2	A Requirements Pattern Example: Actuator-Sensor	200
7.5	Requirements Modeling for WebApps	205
7.5.1	How Much Analysis Is Enough?	205
7.5.2	Requirements Modeling Input	206
7.5.3	Requirements Modeling Output	207
7.5.4	Content Model for WebApps	207
7.5.5	Interaction Model for WebApps	209
7.5.6	Functional Model for WebApps	210
7.5.7	Configuration Models for WebApps	211
7.5.8	Navigation Modeling	212
7.6	Summary	213
PROBLEMS AND POINTS TO PONDER 213		
FURTHER READINGS AND INFORMATION SOURCES 214		

CHAPTER 8 DESIGN CONCEPTS 215

8.1	Design within the Context of Software Engineering	216
8.2	The Design Process	219
8.2.1	Software Quality Guidelines and Attributes	219
8.2.2	The Evolution of Software Design	221
8.3	Design Concepts	222
8.3.1	Abstraction	223
8.3.2	Architecture	223
8.3.3	Patterns	224
8.3.4	Separation of Concerns	225
8.3.5	Modularity	225
8.3.6	Information Hiding	226
8.3.7	Functional Independence	227
8.3.8	Refinement	228
8.3.9	Aspects	228
8.3.10	Refactoring	229
8.3.11	Object-Oriented Design Concepts	230
8.3.12	Design Classes	230
8.4	The Design Model	233
8.4.1	Data Design Elements	234
8.4.2	Architectural Design Elements	234
8.4.3	Interface Design Elements	235
8.4.4	Component-Level Design Elements	237
8.4.5	Deployment-Level Design Elements	237
8.5	Summary	239
PROBLEMS AND POINTS TO PONDER 240		
FURTHER READINGS AND INFORMATION SOURCES 240		

CHAPTER 9 ARCHITECTURAL DESIGN 242

9.1	Software Architecture	243
9.1.1	What Is Architecture?	243
9.1.2	Why Is Architecture Important?	245
9.1.3	Architectural Descriptions	245
9.1.4	Architectural Decisions	246
9.2	Architectural Genres	246
9.3	Architectural Styles	249
9.3.1	A Brief Taxonomy of Architectural Styles	250
9.3.2	Architectural Patterns	253
9.3.3	Organization and Refinement	255
9.4	Architectural Design	255
9.4.1	Representing the System in Context	256
9.4.2	Defining Archetypes	257
9.4.3	Refining the Architecture into Components	258
9.4.4	Describing Instantiations of the System	260
9.5	Assessing Alternative Architectural Designs	261
9.5.1	An Architecture Trade-Off Analysis Method	262
9.5.2	Architectural Complexity	263
9.5.3	Architectural Description Languages	264
9.6	Architectural Mapping Using Data Flow	265
9.6.1	Transform Mapping	265
9.6.2	Refining the Architectural Design	272
9.7	Summary	273
PROBLEMS AND POINTS TO PONDER 274		
FURTHER READINGS AND INFORMATION SOURCES 274		

CHAPTER 10 COMPONENT-LEVEL DESIGN 276

10.1	What Is a Component?	277
10.1.1	An Object-Oriented View	277
10.1.2	The Traditional View	279
10.1.3	A Process-Related View	281
10.2	Designing Class-Based Components	282
10.2.1	Basic Design Principles	282
10.2.2	Component-Level Design Guidelines	285
10.2.3	Cohesion	286
10.2.4	Coupling	288
10.3	Conducting Component-Level Design	290
10.4	Component-Level Design for WebApps	296
10.4.1	Content Design at the Component Level	297
10.4.2	Functional Design at the Component Level	297
10.5	Designing Traditional Components	298
10.5.1	Graphical Design Notation	299
10.5.2	Tabular Design Notation	300
10.5.3	Program Design Language	301
10.6	Component-Based Development	303
10.6.1	Domain Engineering	303
10.6.2	Component Qualification, Adaptation, and Composition	304
10.6.3	Analysis and Design for Reuse	306
10.6.4	Classifying and Retrieving Components	307

TABLE OF CONTENTS

10.7	Summary	309
PROBLEMS AND POINTS TO PONDER		310
FURTHER READINGS AND INFORMATION SOURCES		311

CHAPTER 11 USER INTERFACE DESIGN 312

11.1	The Golden Rules	313
11.1.1	Place the User in Control	313
11.1.2	Reduce the User's Memory Load	314
11.1.3	Make the Interface Consistent	316
11.2	User Interface Analysis and Design	317
11.2.1	Interface Analysis and Design Models	317
11.2.2	The Process	319
11.3	Interface Analysis	320
11.3.1	User Analysis	321
11.3.2	Task Analysis and Modeling	322
11.3.3	Analysis of Display Content	327
11.3.4	Analysis of the Work Environment	328
11.4	Interface Design Steps	328
11.4.1	Applying Interface Design Steps	329
11.4.2	User Interface Design Patterns	330
11.4.3	Design Issues	331
11.5	WebApp Interface Design	335
11.5.1	Interface Design Principles and Guidelines	336
11.5.2	Interface Design Workflow for WebApps	340
11.6	Design Evaluation	342
11.7	Summary	344
PROBLEMS AND POINTS TO PONDER		345
FURTHER READINGS AND INFORMATION SOURCES		346

CHAPTER 12 PATTERN-BASED DESIGN 347

12.1	Design Patterns	348
12.1.1	Kinds of Patterns	349
12.1.2	Frameworks	352
12.1.3	Describing a Pattern	352
12.1.4	Pattern Languages and Repositories	353
12.2	Pattern-Based Software Design	354
12.2.1	Pattern-Based Design in Context	354
12.2.2	Thinking in Patterns	356
12.2.3	Design Tasks	357
12.2.4	Building a Pattern-Organizing Table	358
12.2.5	Common Design Mistakes	359
12.3	Architectural Patterns	360
12.4	Component-Level Design Patterns	362
12.5	User Interface Design Patterns	364
12.6	WebApp Design Patterns	368
12.6.1	Design Focus	368
12.6.2	Design Granularity	369
12.7	Summary	370
PROBLEMS AND POINTS TO PONDER		371
FURTHER READING AND INFORMATION SOURCES		372

CHAPTER 13 WEBAPP DESIGN 373

13.1	WebApp Design Quality	374
13.2	Design Goals	377
13.3	A Design Pyramid for WebApps	378
13.4	WebApp Interface Design	378
13.5	Aesthetic Design	380
13.5.1	Layout Issues	380
13.5.2	Graphic Design Issues	381
13.6	Content Design	382
13.6.1	Content Objects	382
13.6.2	Content Design Issues	382
13.7	Architecture Design	383
13.7.1	Content Architecture	384
13.7.2	WebApp Architecture	386
13.8	Navigation Design	388
13.8.1	Navigation Semantics	388
13.8.2	Navigation Syntax	389
13.9	Component-Level Design	390
13.10	Object-Oriented Hypermedia Design Method (OOHDM)	390
13.10.1	Conceptual Design for OOHDM	391
13.10.2	Navigational Design for OOHDM	391
13.10.3	Abstract Interface Design and Implementation	392
13.11	Summary	393
PROBLEMS AND POINTS TO PONDER 394		
FURTHER READINGS AND INFORMATION SOURCES 395		

PART THREE QUALITY MANAGEMENT 397**CHAPTER 14 QUALITY CONCEPTS 398**

14.1	What Is Quality?	399
14.2	Software Quality	400
14.2.1	Garvin's Quality Dimensions	401
14.2.2	McCall's Quality Factors	402
14.2.3	ISO 9126 Quality Factors	403
14.2.4	Targeted Quality Factors	404
14.2.5	The Transition to a Quantitative View	405
14.3	The Software Quality Dilemma	406
14.3.1	"Good Enough" Software	406
14.3.2	The Cost of Quality	407
14.3.3	Risks	409
14.3.4	Negligence and Liability	410
14.3.5	Quality and Security	410
14.3.6	The Impact of Management Actions	411
14.4	Achieving Software Quality	412
14.4.1	Software Engineering Methods	412
14.4.2	Project Management Techniques	412
14.4.3	Quality Control	412
14.4.4	Quality Assurance	413
14.5	Summary	413
PROBLEMS AND POINTS TO PONDER 414		
FURTHER READINGS AND INFORMATION SOURCES 414		

TABLE OF CONTENTS**CHAPTER 15 REVIEW TECHNIQUES 416**

- 15.1 Cost Impact of Software Defects 417
 - 15.2 Defect Amplification and Removal 418
 - 15.3 Review Metrics and Their Use 420
 - 15.3.1 Analyzing Metrics 420
 - 15.3.2 Cost Effectiveness of Reviews 421
 - 15.4 Reviews: A Formality Spectrum 423
 - 15.5 Informal Reviews 424
 - 15.6 Formal Technical Reviews 426
 - 15.6.1 The Review Meeting 426
 - 15.6.2 Review Reporting and Record Keeping 427
 - 15.6.3 Review Guidelines 427
 - 15.6.4 Sample-Driven Reviews 429
 - 15.7 Summary 430
- PROBLEMS AND POINTS TO PONDER 431
- FURTHER READINGS AND INFORMATION SOURCES 431

CHAPTER 16 SOFTWARE QUALITY ASSURANCE 432

- 16.1 Background Issues 433
 - 16.2 Elements of Software Quality Assurance 434
 - 16.3 SQA Tasks, Goals, and Metrics 436
 - 16.3.1 SQA Tasks 436
 - 16.3.2 Goals, Attributes, and Metrics 437
 - 16.4 Formal Approaches to SQA 438
 - 16.5 Statistical Software Quality Assurance 439
 - 16.5.1 A Generic Example 439
 - 16.5.2 Six Sigma for Software Engineering 441
 - 16.6 Software Reliability 442
 - 16.6.1 Measures of Reliability and Availability 442
 - 16.6.2 Software Safety 443
 - 16.7 The ISO 9000 Quality Standards 444
 - 16.8 The SQA Plan 445
 - 16.9 Summary 446
- PROBLEMS AND POINTS TO PONDER 447
- FURTHER READINGS AND INFORMATION SOURCES 447

CHAPTER 17 SOFTWARE TESTING STRATEGIES 449

- 17.1 A Strategic Approach to Software Testing 450
 - 17.1.1 Verification and Validation 450
 - 17.1.2 Organizing for Software Testing 451
 - 17.1.3 Software Testing Strategy—The Big Picture 452
 - 17.1.4 Criteria for Completion of Testing 455
- 17.2 Strategic Issues 455
- 17.3 Test Strategies for Conventional Software 456
 - 17.3.1 Unit Testing 456
 - 17.3.2 Integration Testing 459
- 17.4 Test Strategies for Object-Oriented Software 465
 - 17.4.1 Unit Testing in the OO Context 466
 - 17.4.2 Integration Testing in the OO Context 466
- 17.5 Test Strategies for WebApps 467
- 17.6 Validation Testing 467

17.6.1	Validation-Test Criteria	468
17.6.2	Configuration Review	468
17.6.3	Alpha and Beta Testing	468
17.7	System Testing	470
17.7.1	Recovery Testing	470
17.7.2	Security Testing	470
17.7.3	Stress Testing	471
17.7.4	Performance Testing	471
17.7.5	Deployment Testing	472
17.8	The Art of Debugging	473
17.8.1	The Debugging Process	473
17.8.2	Psychological Considerations	474
17.8.3	Debugging Strategies	475
17.8.4	Correcting the Error	477
17.9	Summary	478
PROBLEMS AND POINTS TO PONDER		478
FURTHER READINGS AND INFORMATION SOURCES		479

CHAPTER 18 TESTING CONVENTIONAL APPLICATIONS 481

18.1	Software Testing Fundamentals	482
18.2	Internal and External Views of Testing	484
18.3	White-Box Testing	485
18.4	Basis Path Testing	485
18.4.1	Flow Graph Notation	485
18.4.2	Independent Program Paths	487
18.4.3	Deriving Test Cases	489
18.4.4	Graph Matrices	491
18.5	Control Structure Testing	492
18.5.1	Condition Testing	492
18.5.2	Data Flow Testing	493
18.5.3	Loop Testing	493
18.6	Black-Box Testing	495
18.6.1	Graph-Based Testing Methods	495
18.6.2	Equivalence Partitioning	497
18.6.3	Boundary Value Analysis	498
18.6.4	Orthogonal Array Testing	499
18.7	Model-Based Testing	502
18.8	Testing for Specialized Environments, Architectures, and Applications	503
18.8.1	Testing GUIs	503
18.8.2	Testing of Client-Server Architectures	503
18.8.3	Testing Documentation and Help Facilities	505
18.8.4	Testing for Real-Time Systems	506
18.9	Patterns for Software Testing	507
18.10	Summary	508
PROBLEMS AND POINTS TO PONDER		509
FURTHER READINGS AND INFORMATION SOURCES		510

CHAPTER 19 TESTING OBJECT-ORIENTED APPLICATIONS 511

19.1	Broadening the View of Testing	512
19.2	Testing OOA and OOD Models	513

TABLE OF CONTENTS

19.2.1	Correctness of OOA and OOD Models	513
19.2.2	Consistency of Object-Oriented Models	514
19.3	Object-Oriented Testing Strategies	516
19.3.1	Unit Testing in the OO Context	516
19.3.2	Integration Testing in the OO Context	516
19.3.3	Validation Testing in an OO Context	517
19.4	Object-Oriented Testing Methods	517
19.4.1	The Test-Case Design Implications of OO Concepts	518
19.4.2	Applicability of Conventional Test-Case Design Methods	518
19.4.3	Fault-Based Testing	519
19.4.4	Test Cases and the Class Hierarchy	519
19.4.5	Scenario-Based Test Design	520
19.4.6	Testing Surface Structure and Deep Structure	522
19.5	Testing Methods Applicable at the Class Level	522
19.5.1	Random Testing for OO Classes	522
19.5.2	Partition Testing at the Class Level	524
19.6	Interclass Test-Case Design	524
19.6.1	Multiple Class Testing	524
19.6.2	Tests Derived from Behavior Models	526
19.7	Summary	527
	PROBLEMS AND POINTS TO PONDER	528
	FURTHER READINGS AND INFORMATION SOURCES	528

CHAPTER 20 TESTING WEB APPLICATIONS 529

20.1	Testing Concepts for WebApps	530
20.1.1	Dimensions of Quality	530
20.1.2	Errors within a WebApp Environment	531
20.1.3	Testing Strategy	532
20.1.4	Test Planning	532
20.2	The Testing Process—An Overview	533
20.3	Content Testing	534
20.3.1	Content Testing Objectives	534
20.3.2	Database Testing	535
20.4	User Interface Testing	537
20.4.1	Interface Testing Strategy	537
20.4.2	Testing Interface Mechanisms	538
20.4.3	Testing Interface Semantics	540
20.4.4	Usability Tests	540
20.4.5	Compatibility Tests	542
20.5	Component-Level Testing	543
20.6	Navigation Testing	545
20.6.1	Testing Navigation Syntax	545
20.6.2	Testing Navigation Semantics	546
20.7	Configuration Testing	547
20.7.1	Server-Side Issues	547
20.7.2	Client-Side Issues	548
20.8	Security Testing	548
20.9	Performance Testing	550
20.9.1	Performance Testing Objectives	550
20.9.2	Load Testing	551
20.9.3	Stress Testing	552

20.10	Summary	553
PROBLEMS AND POINTS TO PONDER		554
FURTHER READINGS AND INFORMATION SOURCES		555

CHAPTER 21 FORMAL MODELING AND VERIFICATION 557

21.1	The Cleanroom Strategy	558
21.2	Functional Specification	560
21.2.1	Black-Box Specification	561
21.2.2	State-Box Specification	562
21.2.3	Clear-Box Specification	562
21.3	Cleanroom Design	563
21.3.1	Design Refinement	563
21.3.2	Design Verification	564
21.4	Cleanroom Testing	566
21.4.1	Statistical Use Testing	566
21.4.2	Certification	567
21.5	Formal Methods Concepts	568
21.6	Applying Mathematical Notation for Formal Specification	571
21.7	Formal Specification Languages	573
21.7.1	Object Constraint Language (OCL)	574
21.7.2	The Z Specification Language	577
21.8	Summary	580
PROBLEMS AND POINTS TO PONDER		581
FURTHER READINGS AND INFORMATION SOURCES		582

CHAPTER 22 SOFTWARE CONFIGURATION MANAGEMENT 584

22.1	Software Configuration Management	585
22.1.1	An SCM Scenario	586
22.1.2	Elements of a Configuration Management System	587
22.1.3	Baselines	587
22.1.4	Software Configuration Items	589
22.2	The SCM Repository	590
22.2.1	The Role of the Repository	590
22.2.2	General Features and Content	591
22.2.3	SCM Features	592
22.3	The SCM Process	593
22.3.1	Identification of Objects in the Software Configuration	594
22.3.2	Version Control	595
22.3.3	Change Control	596
22.3.4	Configuration Audit	599
22.3.5	Status Reporting	600
22.4	Configuration Management for WebApps	601
22.4.1	Dominant Issues	601
22.4.2	WebApp Configuration Objects	603
22.4.3	Content Management	603
22.4.4	Change Management	606
22.4.5	Version Control	608
22.4.6	Auditing and Reporting	609
22.5	Summary	610
PROBLEMS AND POINTS TO PONDER		611
FURTHER READINGS AND INFORMATION SOURCES		612

TABLE OF CONTENTS**CHAPTER 23 PRODUCT METRICS 613**

23.1	A Framework for Product Metrics	614
23.1.1	Measures, Metrics, and Indicators	614
23.1.2	The Challenge of Product Metrics	615
23.1.3	Measurement Principles	616
23.1.4	Goal-Oriented Software Measurement	617
23.1.5	The Attributes of Effective Software Metrics	618
23.2	Metrics for the Requirements Model	619
23.2.1	Function-Based Metrics	620
23.2.2	Metrics for Specification Quality	623
23.3	Metrics for the Design Model	624
23.3.1	Architectural Design Metrics	624
23.3.2	Metrics for Object-Oriented Design	627
23.3.3	Class-Oriented Metrics—The CK Metrics Suite	628
23.3.4	Class-Oriented Metrics—The MOOD Metrics Suite	631
23.3.5	OO Metrics Proposed by Lorenz and Kidd	632
23.3.6	Component-Level Design Metrics	632
23.3.7	Operation-Oriented Metrics	634
23.3.8	User Interface Design Metrics	635
23.4	Design Metrics for WebApps	636
23.5	Metrics for Source Code	638
23.6	Metrics for Testing	639
23.6.1	Halstead Metrics Applied to Testing	639
23.6.2	Metrics for Object-Oriented Testing	640
23.7	Metrics for Maintenance	641
23.8	Summary	642
	PROBLEMS AND POINTS TO PONDER	642
	FURTHER READINGS AND INFORMATION SOURCES	643

PART FOUR MANAGING SOFTWARE PROJECTS 645

CHAPTER 24 PROJECT MANAGEMENT CONCEPTS 646

24.1	The Management Spectrum	647
24.1.1	The People	647
24.1.2	The Product	648
24.1.3	The Process	648
24.1.4	The Project	648
24.2	People	649
24.2.1	The Stakeholders	649
24.2.2	Team Leaders	650
24.2.3	The Software Team	651
24.2.4	Agile Teams	654
24.2.5	Coordination and Communication Issues	655
24.3	The Product	656
24.3.1	Software Scope	656
24.3.2	Problem Decomposition	656
24.4	The Process	657
24.4.1	Melding the Product and the Process	657
24.4.2	Process Decomposition	658
24.5	The Project	660
24.6	The W ⁵ HH Principle	661

24.7	Critical Practices	662
24.8	Summary	663
PROBLEMS AND POINTS TO PONDER		663
FURTHER READINGS AND INFORMATION SOURCES		664

CHAPTER 25 PROCESS AND PROJECT METRICS 666

25.1	Metrics in the Process and Project Domains	667
25.1.1	Process Metrics and Software Process Improvement	667
25.1.2	Project Metrics	670
25.2	Software Measurement	671
25.2.1	Size-Oriented Metrics	672
25.2.2	Function-Oriented Metrics	673
25.2.3	Reconciling LOC and FP Metrics	673
25.2.4	Object-Oriented Metrics	675
25.2.5	Use-Case-Oriented Metrics	676
25.2.6	WebApp Project Metrics	677
25.3	Metrics for Software Quality	679
25.3.1	Measuring Quality	680
25.3.2	Defect Removal Efficiency	681
25.4	Integrating Metrics within the Software Process	682
25.4.1	Arguments for Software Metrics	683
25.4.2	Establishing a Baseline	683
25.4.3	Metrics Collection, Computation, and Evaluation	684
25.5	Metrics for Small Organizations	684
25.6	Establishing a Software Metrics Program	686
25.7	Summary	688
PROBLEMS AND POINTS TO PONDER		688
FURTHER READINGS AND INFORMATION SOURCES		689

CHAPTER 26 ESTIMATION FOR SOFTWARE PROJECTS 691

26.1	Observations on Estimation	692
26.2	The Project Planning Process	693
26.3	Software Scope and Feasibility	694
26.4	Resources	695
26.4.1	Human Resources	695
26.4.2	Reusable Software Resources	696
26.4.3	Environmental Resources	696
26.5	Software Project Estimation	697
26.6	Decomposition Techniques	698
26.6.1	Software Sizing	698
26.6.2	Problem-Based Estimation	699
26.6.3	An Example of LOC-Based Estimation	701
26.6.4	An Example of FP-Based Estimation	702
26.6.5	Process-Based Estimation	703
26.6.6	An Example of Process-Based Estimation	704
26.6.7	Estimation with Use Cases	705
26.6.8	An Example of Use-Case-Based Estimation	706
26.6.9	Reconciling Estimates	707
26.7	Empirical Estimation Models	708
26.7.1	The Structure of Estimation Models	709
26.7.2	The COCOMO II Model	709
26.7.3	The Software Equation	711

TABLE OF CONTENTS

26.8	Estimation for Object-Oriented Projects	712
26.9	Specialized Estimation Techniques	713
26.9.1	Estimation for Agile Development	713
26.9.2	Estimation for WebApp Projects	714
26.10	The Make/Buy Decision	715
26.10.1	Creating a Decision Tree	715
26.10.2	Outsourcing	717
26.11	Summary	718
PROBLEMS AND POINTS TO PONDER		719
FURTHER READINGS AND INFORMATION SOURCES		719

CHAPTER 27 PROJECT SCHEDULING 721

27.1	Basic Concepts	722
27.2	Project Scheduling	724
27.2.1	Basic Principles	725
27.2.2	The Relationship Between People and Effort	725
27.2.3	Effort Distribution	727
27.3	Defining a Task Set for the Software Project	728
27.3.1	A Task Set Example	729
27.3.2	Refinement of Software Engineering Actions	730
27.4	Defining a Task Network	731
27.5	Scheduling	732
27.5.1	Time-Line Charts	732
27.5.2	Tracking the Schedule	734
27.5.3	Tracking Progress for an OO Project	735
27.5.4	Scheduling for WebApp Projects	736
27.6	Earned Value Analysis	739
27.7	Summary	741
PROBLEMS AND POINTS TO PONDER		741
FURTHER READINGS AND INFORMATION SOURCES		743

CHAPTER 28 RISK MANAGEMENT 744

28.1	Reactive versus Proactive Risk Strategies	745
28.2	Software Risks	745
28.3	Risk Identification	747
28.3.1	Assessing Overall Project Risk	748
28.3.2	Risk Components and Drivers	749
28.4	Risk Projection	749
28.4.1	Developing a Risk Table	750
28.4.2	Assessing Risk Impact	752
28.5	Risk Refinement	754
28.6	Risk Mitigation, Monitoring, and Management	755
28.7	The RMMM Plan	757
28.8	Summary	759
PROBLEMS AND POINTS TO PONDER		759
FURTHER READINGS AND INFORMATION SOURCES		760

CHAPTER 29 MAINTENANCE AND REENGINEERING 761

29.1	Software Maintenance	762
29.2	Software Supportability	764

29.3	Reengineering	764
29.4	Business Process Reengineering	765
29.4.1	Business Processes	765
29.4.2	A BPR Model	766
29.5	Software Reengineering	768
29.5.1	A Software Reengineering Process Model	768
29.5.2	Software Reengineering Activities	770
29.6	Reverse Engineering	772
29.6.1	Reverse Engineering to Understand Data	773
29.6.2	Reverse Engineering to Understand Processing	774
29.6.3	Reverse Engineering User Interfaces	775
29.7	Restructuring	776
29.7.1	Code Restructuring	776
29.7.2	Data Restructuring	777
29.8	Forward Engineering	778
29.8.1	Forward Engineering for Client-Server Architectures	779
29.8.2	Forward Engineering for Object-Oriented Architectures	780
29.9	The Economics of Reengineering	780
29.10	Summary	781
	PROBLEMS AND POINTS TO PONDER	782
	FURTHER READINGS AND INFORMATION SOURCES	783

PART FIVE ADVANCED TOPICS 785**CHAPTER 30 SOFTWARE PROCESS IMPROVEMENT 786**

30.1	What Is SPI?	787
30.1.1	Approaches to SPI	787
30.1.2	Maturity Models	789
30.1.3	Is SPI for Everyone?	790
30.2	The SPI Process	791
30.2.1	Assessment and Gap Analysis	791
30.2.2	Education and Training	793
30.2.3	Selection and Justification	793
30.2.4	Installation/Migration	794
30.2.5	Evaluation	795
30.2.6	Risk Management for SPI	795
30.2.7	Critical Success Factors	796
30.3	The CMMI	797
30.4	The People CMM	801
30.5	Other SPI Frameworks	802
30.6	SPI Return on Investment	804
30.7	SPI Trends	805
30.8	Summary	806
	PROBLEMS AND POINTS TO PONDER	806
	FURTHER READINGS AND INFORMATION SOURCES	807

CHAPTER 31 EMERGING TRENDS IN SOFTWARE ENGINEERING 808

31.1	Technology Evolution	809
31.2	Observing Software Engineering Trends	811

TABLE OF CONTENTS

31.3	Identifying "Soft Trends"	812
31.3.1	Managing Complexity	814
31.3.2	Open-World Software	815
31.3.3	Emergent Requirements	816
31.3.4	The Talent Mix	816
31.3.5	Software Building Blocks	817
31.3.6	Changing Perceptions of "Value"	818
31.3.7	Open Source	818
31.4	Technology Directions	819
31.4.1	Process Trends	819
31.4.2	The Grand Challenge	821
31.4.3	Collaborative Development	822
31.4.4	Requirements Engineering	824
31.4.5	Model-Driven Software Development	825
31.4.6	Postmodern Design	825
31.4.7	Test-Driven Development	826
31.5	Tools-Related Trends	827
31.5.1	Tools That Respond to Soft Trends	828
31.5.2	Tools That Address Technology Trends	830
31.6	Summary	830
	PROBLEMS AND POINTS TO PONDER	831
	FURTHER READINGS AND INFORMATION SOURCES	831

CHAPTER 32 CONCLUDING COMMENTS 833

32.1	The Importance of Software—Revisited	834
32.2	People and the Way They Build Systems	834
32.3	New Modes for Representing Information	835
32.4	The Long View	837
32.5	The Software Engineer's Responsibility	838
32.6	A Final Comment	839

APPENDIX 1 AN INTRODUCTION TO UML 841**APPENDIX 2 OBJECT-ORIENTED CONCEPTS 863****REFERENCES 871****INDEX 889**

PREFACE

When computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

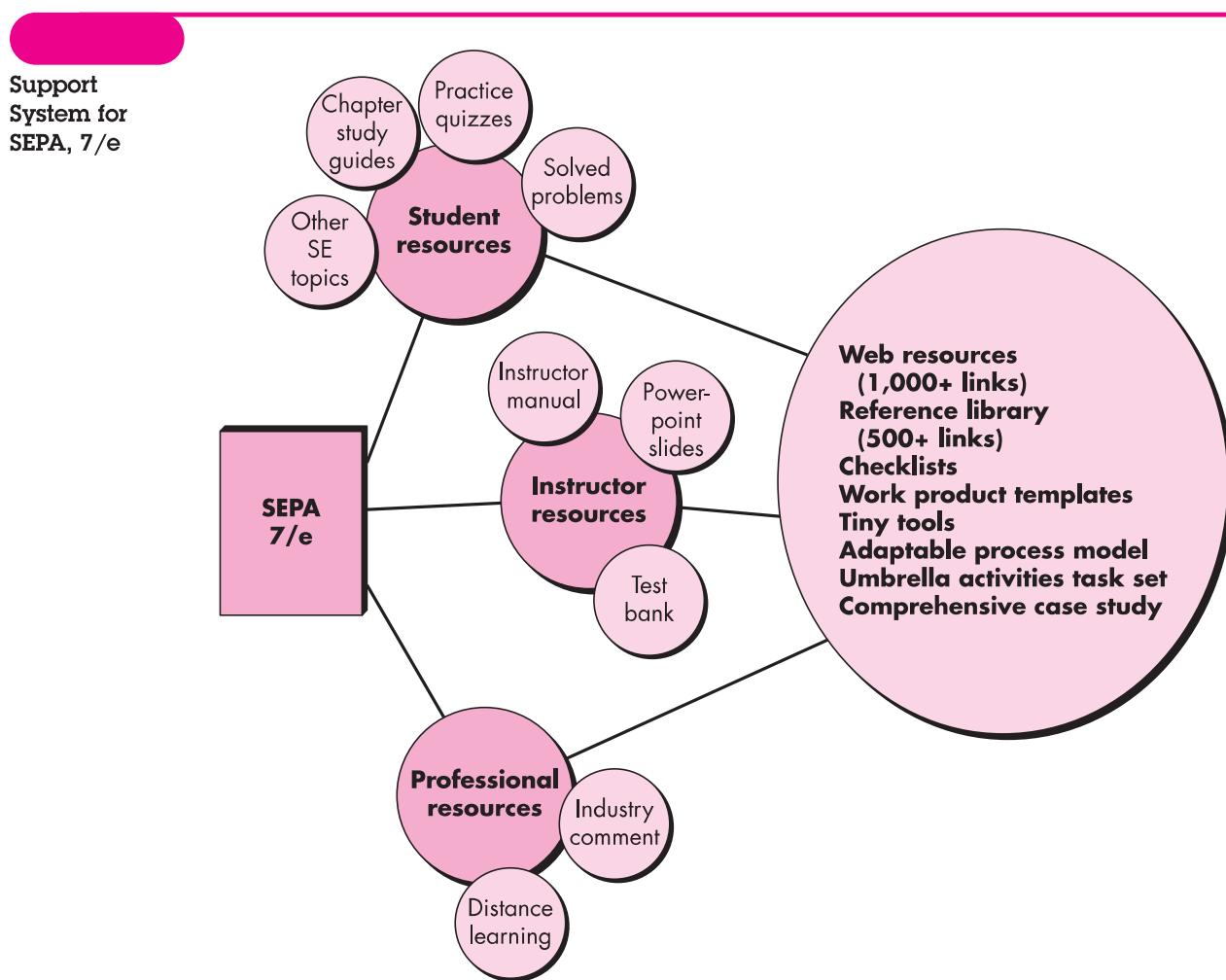
It has been almost three decades since the first edition of this book was written. During that time, software engineering has evolved from an obscure idea practiced by a relatively small number of zealots to a legitimate engineering discipline. Today, it is recognized as a subject worthy of serious research, conscientious study, and tumultuous debate. Throughout the industry, software engineer has replaced programmer as the job title of preference. Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry segments.

Although managers and practitioners alike recognize the need for a more disciplined approach to software, they continue to debate the manner in which discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service today's most advanced technologies. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers, and bad things happen. In addition, debate and controversy about the true nature of the software engineering approach continue. The status of software engineering is a study in contrasts. Attitudes have changed, progress has been made, but much remains to be done before the discipline reaches full maturity.

The seventh edition of *Software Engineering: A Practitioner's Approach* is intended to serve as a guide to a maturing engineering discipline. Like the six editions that preceded it, the seventh edition is intended for both students and practitioners, retaining its appeal as a guide to the industry professional and a comprehensive introduction to the student at the upper-level undergraduate or first-year graduate level.

The seventh edition is considerably more than a simple update. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. In addition, a revised and updated “support system,” illustrated in the figure, provides a comprehensive set of student, instructor, and professional resources to complement the content of the book. These resources are presented as part of a website (www.mhhe.com/pressman) specifically designed for *Software Engineering: A Practitioner's Approach*.

The Seventh Edition. The 32 chapters of the seventh edition have been reorganized into five parts. This organization, which differs considerably from the sixth edition, has been done to better compartmentalize topics and assist instructors who may not have the time to complete the entire book in one term.



Part 1, *The Process*, presents a variety of different views of software process, considering all important process models and addressing the debate between prescriptive and agile process philosophies. Part 2, *Modeling*, presents analysis and design methods with an emphasis on object-oriented techniques and UML modeling. Pattern-based design and design for Web applications are also considered. Part 3, *Quality Management*, presents the concepts, procedures, techniques, and methods that enable a software team to assess software quality, review software engineering work products, conduct SQA procedures, and apply an effective testing strategy and tactics. In addition, formal modeling and verification methods are also considered. Part 4, *Managing Software Projects*, presents topics that are relevant to those who plan, manage, and control a software development project. Part 5, *Advanced Topics*, considers software process improvement and software engineering trends. Continuing in the tradition of past editions, a series of sidebars is used throughout the book to present the trials and tribulations of a (fictional) software team and to provide supplementary materials about methods and tools that are relevant to chapter topics. Two new appendices provide brief tutorials on UML and object-oriented thinking for those who may be unfamiliar with these important topics.

The five-part organization of the seventh edition enables an instructor to “cluster” topics based on available time and student need. An entire one-term course can be built around one or more of the five parts. A software engineering survey course would select chapters from all five parts. A software engineering course that emphasizes analysis and design would select topics from Parts 1 and 2. A testing-oriented software engineering course would select topics from Parts 1 and 3, with a brief foray into Part 2. A “management course” would stress Parts 1 and 4. By organizing the seventh edition in this way, I have attempted to provide an instructor with a number of teaching options. In every case, the content of the seventh edition is complemented by the following elements of the *SEPA, 7/e Support System*.

Student Resources. A wide variety of student resources includes an extensive online learning center encompassing chapter-by-chapter study guides, practice quizzes, problem solutions, and a variety of Web-based resources including software engineering checklists, an evolving collection of “tiny tools,” a comprehensive case study, work product templates, and many other resources. In addition, over 1000 categorized *Web References* allow a student to explore software engineering in greater detail and a *Reference Library* with links to over 500 downloadable papers provides an in-depth source of advanced software engineering information.

Instructor Resources. A broad array of instructor resources has been developed to supplement the seventh edition. These include a complete online *Instructor’s Guide* (also downloadable) and supplementary teaching materials including a complete set of over 700 *PowerPoint Slides* that may be used for lectures, and a test bank. Of course, all resources available for students (e.g., tiny tools, the *Web References*, the downloadable *Reference Library*) and professionals are also available.

The *Instructor’s Guide for Software Engineering: A Practitioner’s Approach* presents suggestions for conducting various types of software engineering courses, recommendations for a variety of software projects to be conducted in conjunction with a course, solutions to selected problems, and a number of useful teaching aids.

Professional Resources. A collection of resources available to industry practitioners (as well as students and faculty) includes outlines and samples of software engineering documents and other work products, a useful set of software engineering checklists, a catalog of software engineering (CASE) tools, a comprehensive collection of Web-based resources, and an “adaptable process model” that provides a detailed task breakdown of the software engineering process.

When coupled with its online support system, the seventh edition of *Software Engineering: A Practitioner’s Approach*, provides flexibility and depth of content that cannot be achieved by a textbook alone.

Acknowledgments. My work on the seven editions of *Software Engineering: A Practitioner’s Approach* has been the longest continuing technical project of my life. Even when the writing stops, information extracted from the technical literature continues to be assimilated and organized, and criticism and suggestions from readers worldwide is evaluated and catalogued. For this reason, my thanks to the many authors of books, papers, and articles (in both hardcopy and electronic media) who have provided me with additional insight, ideas, and commentary over nearly 30 years.

Special thanks go to Tim Lethbridge of the University of Ottawa, who assisted me in the development of UML and OCL examples and developed the case study that accompanies this book, and Dale Skrien of Colby College, who developed the UML tutorial in

Appendix 1. Their assistance and comments were invaluable. Special thanks also go to Bruce Maxim of the University of Michigan–Dearborn, who assisted me in developing much of the pedagogical website content that accompanies this book. Finally, I wish to thank the reviewers of the seventh edition: Their in-depth comments and thoughtful criticism have been invaluable.

Osman Balci,
Virginia Tech University

Max Fomitchev,
Penn State University

Jerry (Zeyu) Gao,
San Jose State University

Guillermo Garcia,
Universidad Alfonso X Madrid

Pablo Gervas,
Universidad Complutense de Madrid

SK Jain,
National Institute of Technology Hamirpur

Saeed Monemi,
Cal Poly Pomona

Ahmed Salem,
California State University

Vasudeva Varma,
IIT Hyderabad

The content of the seventh edition of *Software Engineering: A Practitioner's Approach* has been shaped by industry professionals, university professors, and students who have used earlier editions of the book and have taken the time to communicate their suggestions, criticisms, and ideas. My thanks to each of you. In addition, my personal thanks go to our many industry clients worldwide, who certainly have taught me as much or more than I could ever teach them.

As the editions of this book have evolved, my sons, Mathew and Michael, have grown from boys to men. Their maturity, character, and success in the real world have been an inspiration to me. Nothing has filled me with more pride. And finally, to Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of "the book."

Roger S. Pressman

SOFTWARE AND SOFTWARE ENGINEERING

KEY CONCEPTS

application domains	7
characteristics of software	4
framework activities	15
legacy software ..	9
practice	17
principles	19
software engineering	12
software myths ..	21
software process ..	14
umbrella activities	16
WebApps	10

QUICK LOOK

What is it? Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media. Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high-quality computer software.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

Why is it important? Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

What is the work product? From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software. But from the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

How do I ensure that I've done it right? Read the remainder of this book, select those ideas that are applicable to the software that you build, and apply them to your work.

I had to agree. "So, your life will be much simpler. You guys won't have to worry about five different versions of the same App in use across tens of thousands of users."

He smiled. "Absolutely. Only the most current version residing on our servers. When we make a change or a correction, we supply updated functionality and content to every user. Everyone has it instantly!"

I grimaced. "But if you make a mistake, everyone has that instantly as well."

He chuckled. "True, that's why we're redoubling our efforts to do even better software engineering. Problem is, we have to do it 'fast' because the market has accelerated in every application area."

I leaned back and put my hands behind my head. "You know what they say, . . . you can have it fast, you can have it right, or you can have it cheap. Pick two!"

"I'll take it fast and right," he said as he began to get up.

I stood as well. "Then you really do need software engineering."

"I know that," he said as he began to move away. "The problem is, we've got to convince still another generation of techies that it's true!"

Is software *really* dead? If it was, you wouldn't be reading this book!

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Fifty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the printing industry); that software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than almost all industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

No one could foresee that software would become embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial, entertainment, office machines, . . . the list is almost endless. And if you believe the law of unintended consequences, there are many effects that we cannot yet predict.

No one could predict that millions of computer programs would have to be corrected, adapted, and enhanced as time passed. The burden of performing these "maintenance" activities would absorb more people and more resources than all work applied to the creation of new software.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive

**Quote:**

"Ideas and technological discoveries are the driving engines of economic growth."

Wall Street Journal

to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad-based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering*.

1.1 THE NATURE OF SOFTWARE



Software is both a product and a vehicle that delivers a product.

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information*. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options, have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone

quote:

"Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets."

Brad J. Cox

programmer are the same questions that are asked when modern computer-based systems are built:¹

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has lead to the adoption of software engineering practice.

1.1.1 Defining Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:



Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered.

But a more formal definition probably won't measurably improve your understanding. To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent

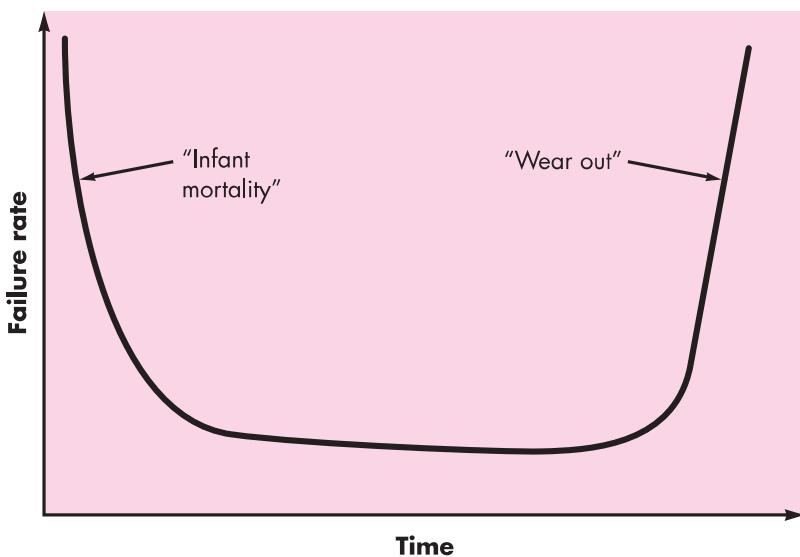


Software is engineered, not manufactured.

¹ In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin asking 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

FIGURE 1.1

**Failure curve
for hardware**



(or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different (see Chapter 24). Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

KEY POINT

Software doesn't wear out, but it does deteriorate.

2. Software doesn't "wear out."

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out*.

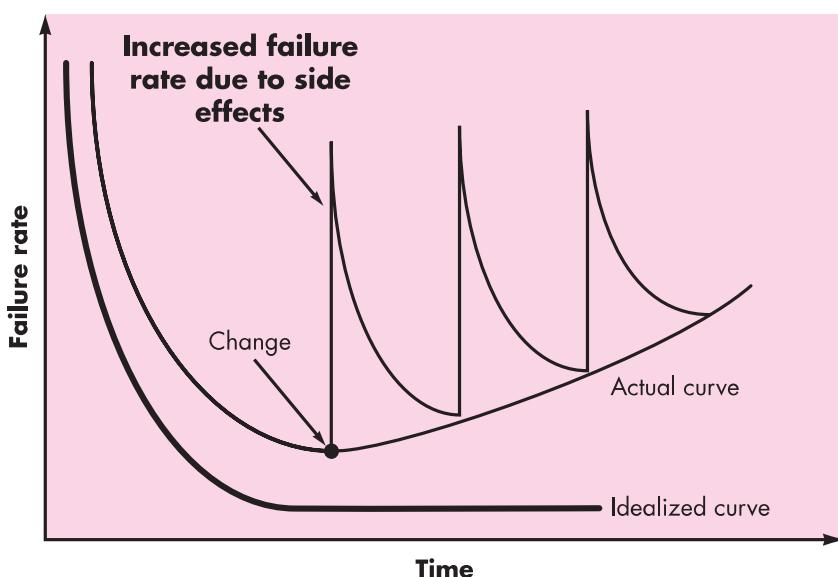
Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!

ADVICE

If you want to reduce software deterioration, you'll have to do better software design (Chapters 8 to 13).

FIGURE 1.2

Failure curves
for software



KEY POINT

Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve in Figure 1.2.

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,² software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve” (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

- 3.** *Although the industry is moving toward component-based construction, most software continues to be custom built.*

“Ideas are the building blocks of ideas.”

Jason Zebekazy

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent

² In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by a variety of different stakeholders.

something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.³ For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,⁴ information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software—has been characterized by “number crunching” algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from

WebRef

One of the most comprehensive libraries of shareware/freeware can be found at shareware.cnet.com

³ Component-based development is discussed in Chapter 10.

⁴ Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

Web applications—called “WebApps,” this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

**note:**

“There is no computer that has common sense.”

Marvin Minsky

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is! Past generations of software people have left a legacy in each of the categories I have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden of future software engineers. And yet, new challenges (Chapter 31) have appeared on the horizon:

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

Netsourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Quote:

"You can't always predict, but you can always prepare."

Anonymous

Each of these new challenges will undoubtedly obey the law of unintended consequences and have effects (for businesspeople, software engineers, and end users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and to business rules that are sure to come over the next decade.

1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that “many legacy systems remain supportive to core business functions and are ‘indispensable’ to the business.” Hence, legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.⁵ Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results

?

What do I do if I encounter a legacy system that exhibits poor quality?

⁵ In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support “core business functions and are indispensable to the business.” What to do?

The only reasonable answer may be: *Do nothing*, at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn’t broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:



- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.



Every software engineer must recognize that change is natural. Don’t try to fight it.

When these modes of evolution occur, a legacy system must be reengineered (Chapter 29) so that it remains viable into the future. The goal of modern software engineering is to “devise methodologies that are founded on the notion of evolution”; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other” [Day99].

1.2 THE UNIQUE NATURE OF WEBAPPS

Quote:

“By the time we see any sort of stabilization, the Web will have turned into something completely different.”

Louis Monier

In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications*⁶ (I refer to these collectively as *WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

As noted in Section 1.1.2, WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different. Powell [Pow98] suggests that Web-based systems and applications “involve a mixture between print publishing and software development, between marketing and computing, between

6 In the context of this book, the term *Web application* (WebApp) encompasses everything from a simple Web page that might help a consumer compute an automobile lease payment to a comprehensive website that provides complete travel services for businesspeople and vacationers. Included within this category are complete websites, specialized functionality within websites, and information processing applications that reside on the Internet or on an Intranet or Extranet.



internal communications and external relations, and between art and technology.” The following attributes are encountered in the vast majority of WebApps.

Network intensiveness. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

Concurrency. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

Performance. If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

Availability. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

Data driven. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

Content sensitive. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

Continuous evolution. Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

Immediacy. Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.⁷

Security. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

⁷ With modern tools, sophisticated Web pages can be produced in only a few hours.

of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

Aesthetics. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

It can be argued that other application categories discussed in Section 1.1.2 can exhibit some of the attributes noted. However, WebApps almost always exhibit all of them.

1.3 SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:



Understand the problem before you build a solution.

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application⁸ has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*



Design is a pivotal software engineering activity.

- The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. *It follows that design becomes a pivotal activity.*



Both quality and maintainability are an outgrowth of good design.

- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*
- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use

⁸ I will call these people “stakeholders” later in this book.

increase, demands for adaptation and enhancement will also grow. *It follows that software should be maintainable.*

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered*. And that leads us to the topic of this book—*software engineering*.

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [Nau69] at the seminal conference on the subject still serves as a basis for discussion:

[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

You will be tempted to add to this definition.⁹ It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of an effective process. And yet, Bauer's definition provides us with a baseline. What are the "sound engineering principles" that can be applied to computer software development? How do we "economically" build software so that it is "reliable"? What is required to create computer programs that work "efficiently" on not one but many different "real machines"? These are the questions that continue to challenge software engineers.

The IEEE [IEE93a] has developed a more comprehensive definition when it states:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies¹⁰ foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework

 **note:**
"More than a discipline or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem."

Scott Whitmire

 **How do we define software engineering?**

KEY POINT
Software engineering encompasses a process, methods for managing and engineering software, and tools.

⁹ For numerous additional definitions of *software engineering*, see www.answers.com/topic/software-engineering#wp_note-13.

¹⁰ Quality management and related approaches are discussed in Chapter 14 and throughout Part 3 of this book.



that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

1.4 THE SOFTWARE PROCESS

? **What are the elements of a software process?**

Quote:

"A process defines who is doing what when and how to reach a certain goal."

**Ivar Jacobson,
Grady Booch,
and James
Rumbaugh**

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

WebRef

CrossTalk is a journal that provides pragmatic information on process, methods, and tools. It can be found at:
www.stsc.hill.af.mil.

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

 **What are the five generic process framework activities?**

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders)¹¹ The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

¹¹ A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. Rob Thomsett jokes that, "a stakeholder is a person holding a large and sharp stake. . . . If you don't look after your stakeholders, you know where the stake will end up.").

 **Quote:**
"Einstein argued that there must be a simplified explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity."

Fred Brooks

For many software projects, framework activities are applied iteratively as a project progresses. That is, **communication**, **planning**, **modeling**, **construction**, and **deployment** are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of *umbrella activities*. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

KEY POINT

Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Each of these umbrella activities is discussed in detail later in this book.

Earlier in this section, I noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and tasks are defined within each framework activity
- Degree to which work products are identified and required

KEY POINT

Software process adaptation is essential for project success.

How do process models differ from one another?

note:

"I feel a recipe is only a theme which an intelligent cook can play each time with a variation."

Madame Benoit

- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor with which the process is described
- Degree to which the customer and other stakeholders are involved with the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

In Part 1 of this book, I'll examine software process in considerable detail. *Prescriptive process models* (Chapter 2) stress detailed definition, identification, and application of process activities and tasks. Their intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system. Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy associated with building computer-based systems and inadvertently create difficulty for all stakeholders.

? **What characterizes an "agile" process?**

Agile process models (Chapter 3) emphasize project "agility" and follow a set of principles that lead to a more informal (but, proponents argue, no less effective) approach to software process. These process models are generally characterized as "agile" because they emphasize maneuverability and adaptability. They are appropriate for many types of projects and are particularly useful when Web applications are engineered.

1.5 SOFTWARE ENGINEERING PRACTICE

WebRef

A variety of thought-provoking quotes on the practice of software engineering can be found at www.literateprogramming.com

In Section 1.4, I introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction, and deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.¹²

ADVICE

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

1.5.1 The Essence of Practice

In a classic book, *How to Solve It*, written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).

12 You should revisit relevant sections within this chapter as specific software engineering methods and umbrella activities are discussed later in this book.

3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing*. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

 **quote:**

"There is a grain of discovery in the solution of any problem."

George Polya

Plan the solution. Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry out the plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

1.5.2 General Principles

The dictionary defines the word *principle* as “an important underlying law or assumption required in a system of thought.” Throughout this book I'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:¹³



Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is

¹³ Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.

note:

"There is a certain majesty in simplicity which is far above all the quaintness of wit."

**Alexander Pope
(1688–1744)**

KEY POINT

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing.* The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner.* Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.¹⁴ This could very possibly lead to the reuse of an entire system.

¹⁴ This advice can be dangerous if it is taken to extremes. Designing for the “general problem” sometimes requires performance compromises and can make specific solutions inefficient.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.¹⁵ Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results.* When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

1.6 SOFTWARE MYTHS

Quote:

"In the absence of meaningful standards, a new industry like software comes to depend instead on folklore."

Tom DeMarco

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

¹⁵ Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

WebRef

The Software Project Managers Network at www.spmn.com can help you dispel these and other myths.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it out-sources software projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.



Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived

iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.¹⁶ However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.



*Whenever you think,
we don't have time for
software engineering,
ask yourself, "Will we
have time to do it over
again?"*

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program “running” I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review. Software reviews (described in Chapter 15) are a “quality filter” that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

¹⁶ Many software engineers have adopted an “agile” approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 3.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a “legacy system” to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

SAFEHOME¹⁷



How a Project Starts

The scene: Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call “home management products.” We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

¹⁷ The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are purely fictitious, but the situations and problems are real.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* product line. The engineering effort will succeed only if *SafeHome* software succeeds. The market will accept the product only if the software embedded within it properly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

1.8 SUMMARY

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these WebApps have unique features and requirements, they are software nonetheless.

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem solving activity that follows a set of core principles.

A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies required to build it grows. As you learn more about software engineering, you'll begin to understand why these myths should be debunked whenever they are encountered.

PROBLEMS AND POINTS TO PONDER

1.1. Provide at least five additional examples of how the law of unintended consequences applies to computer software.

1.2. Provide a number of examples (both positive and negative) that indicate the impact of software on our society.

1.3. Develop your own answers to the five questions asked at the beginning of Section 1.1. Discuss them with your fellow students.

1.4. Many modern applications change frequently—before they are presented to the end user and then after the first version has been put into use. Suggest a few ways to build software to stop deterioration due to change.

1.5. Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.

1.6. Figure 1.3 places the three software engineering layers on top of a layer entitled “a quality focus.” This implies an organizational quality program such as total quality management. Do a bit of research and develop an outline of the key tenets of a total quality management program.

1.7. Is software engineering applicable when WebApps are built? If so, how might it be modified to accommodate the unique characteristics of WebApps?

1.8. As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm (either economic or human).

1.9. Describe a process framework in your own words. When we say that framework activities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.

1.10. Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities?

1.11. Add two additional myths to the list presented in Section 1.6. Also state the reality that accompanies the myth.

FURTHER READINGS AND INFORMATION SOURCES¹⁸

There are literally thousands of books written about computer software. The vast majority discuss programming languages or software applications, but a few discuss software itself. Pressman and Herron (*Software Shock*, Dorset House, 1991) presented an early discussion (directed at the layperson) of software and the way professionals build it. Negroponte’s best-selling book (*Being Digital*, Alfred A. Knopf, Inc., 1995) provides a view of computing and its overall impact in the twenty-first century. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software and the process through which it is developed.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argues that the “modern scourge” of software bugs can be eliminated and suggests ways to accomplish this. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) argues that the “divide” between those who have access to information resources (e.g., the Web) and those that do not is narrowing as we move into the first decade of this century. Books by Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) and Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introduce the concept of “open-world” software and predict a wireless environment in which software must adapt to requirements that emerge in real time.

The current state of the software engineering and the software process can best be determined from publications such as *IEEE Software*, *IEEE Computer*, *CrossTalk*, and *IEEE Transactions on Software Engineering*. Industry periodicals such as *Application Development Trends* and *Cutter*

¹⁸ The *Further Reading and Information Sources* section presented at the conclusion of each chapter presents a brief overview of print sources that can help to expand your understanding of the major topics presented in the chapter. I have created a comprehensive website to support *Software Engineering: A Practitioner’s Approach* at www.mhhe.com/compsci/pressman. Among the many topics addressed within the website are chapter-by-chapter software engineering resources to Web-based information that can complement the material presented in each chapter. An Amazon.com link to every book noted in this section is contained within these resources.

IT Journal often contain articles on software engineering topics. The discipline is “summarized” every year in the *Proceeding of the International Conference on Software Engineering*, sponsored by the IEEE and ACM, and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes*, and *Annals of Software Engineering*. Tens of thousands of websites are dedicated to software engineering and the software process.

Many books addressing the software process and software engineering have been published in recent years. Some present an overview of the entire process, while others delve into a few important topics to the exclusion of others. Among the more popular offerings (in addition to this book!) are

- Abran, A., and J. Moore, *SWEBOk: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Andersson, E., et al., *Software Engineering for Internet Applications*, The MIT Press, 2006.
- Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.
- Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2d ed., Addison-Wesley, 2008.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Pfleeger, S., *Software Engineering: Theory and Practice*, 3d ed., Prentice-Hall, 2005.
- Schach, S., *Object-Oriented and Classical Software Engineering*, 7th ed., McGraw-Hill, 2006.
- Sommerville, I., *Software Engineering*, 8th ed., Addison-Wesley, 2006.
- Tsui, F., and O. Karam, *Essentials of Software Engineering*, Jones & Bartlett Publishers, 2006.

Many software engineering standards have been published by the IEEE, ISO, and their standards organizations over the past few decades. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) provides a useful survey of relevant standards and how they apply to real projects.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.