

This International Student Edition is for use outside of the U.S.

NINTH EDITION

# Software Engineering

A PRACTITIONER'S APPROACH

Mc  
Graw  
Hill

ROGER S. PRESSMAN

BRUCE R. MAXIM

# Software Engineering

A PRACTITIONER'S APPROACH





# Software Engineering

A PRACTITIONER'S APPROACH

NINTH EDITION

**Roger S. Pressman, Ph.D.**  
**Bruce R. Maxim, Ph.D.**

**Mc  
Graw  
Hill**



## SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright © 2020 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. 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 McGraw-Hill Education, 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 LCR 24 23 22 21 20 19

ISBN 978-1-260-54800-6

MHID 1-260-54800-7

Cover Image: ©R.L. Hausdorf/Shutterstock

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw-Hill Education, and McGraw-Hill Education does not guarantee the accuracy of the information presented at these sites.



*To Barbara, Matt, Mike,  
Shiri, Adam, Lily,  
and Maya.*

—Roger S. Pressman

*To my family who  
support me in all  
that I do.*

—Bruce R. Maxim

## ABOUT THE AUTHORS



Courtesy of Roger Pressman

**Roger S. Pressman** is an internationally recognized consultant and author in software engineering. For almost five decades, he has worked as a software engineer, a manager, a professor, an author, a consultant, and an entrepreneur.

Dr. Pressman was president of R. S. Pressman & Associates, Inc., a consulting firm that specialized in helping companies establish effective software engineering strategies. Over the years he developed a set of techniques and tools that improved software engineering practice. He is also the founder and chief technology officer of EVANNEX<sup>®</sup>, an automotive aftermarket company that specializes in the design and manufacture of accessories for the Tesla line of electric vehicles.

Dr. Pressman is the author of ten books, including two novels, and many technical and management papers. He has been on the editorial boards of *IEEE Software* and *The Cutter IT Journal* and was editor of the “Manager” column in *IEEE Software*.

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He has presented tutorials at the International Conference on Software Engineering and at many other industry meetings. He has been a member of the ACM, IEEE, Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.



Michigan Creative/  
UM-Dearborn

**Bruce R. Maxim** has worked as a software engineer, project manager, professor, author, and consultant for more than thirty years. His research interests include software engineering, user experience design, serious game development, artificial intelligence, and engineering education.

Dr. Maxim is professor of computer and information science and collegiate professor of engineering at the University of Michigan–Dearborn. He established the GAME Lab in the College of Engineering and Computer Science. He has published papers on computer algorithm animation, game development, and engineering education. He is coauthor of a best-selling introductory computer science text and two edited collections of software engineering research papers. Dr. Maxim has supervised several hundred industry-based software development projects as part of his work at the University of Michigan–Dearborn.

Dr. Maxim’s professional experience includes managing research information systems at a medical school, directing instructional computing for a medical campus, and working as a statistical programmer. Dr. Maxim served as the chief technology officer for a game development company.

Dr. Maxim was the recipient of several distinguished teaching awards, a distinguished community service award, and a distinguished faculty governance award. He is a member of Sigma Xi, Upsilon Pi Epsilon, Pi Mu Epsilon, Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers, and International Game Developers Association.

# CONTENTS AT A GLANCE

CHAPTER 1 Software and Software Engineering 1

## **PART ONE THE SOFTWARE PROCESS 19**

---

CHAPTER 2 Process Models 20  
CHAPTER 3 Agility and Process 37  
CHAPTER 4 Recommended Process Model 54  
CHAPTER 5 Human Aspects of Software Engineering 74

## **PART TWO MODELING 83**

---

CHAPTER 6 Principles That Guide Practice 84  
CHAPTER 7 Understanding Requirements 102  
CHAPTER 8 Requirements Modeling—A Recommended Approach 126  
CHAPTER 9 Design Concepts 156  
CHAPTER 10 Architectural Design—A Recommended Approach 181  
CHAPTER 11 Component-Level Design 206  
CHAPTER 12 User Experience Design 233  
CHAPTER 13 Design for Mobility 264  
CHAPTER 14 Pattern-Based Design 289

## **PART THREE QUALITY AND SECURITY 309**

---

CHAPTER 15 Quality Concepts 310  
CHAPTER 16 Reviews—A Recommended Approach 325  
CHAPTER 17 Software Quality Assurance 339  
CHAPTER 18 Software Security Engineering 356  
CHAPTER 19 Software Testing—Component Level 372  
CHAPTER 20 Software Testing—Integration Level 395  
CHAPTER 21 Software Testing—Specialized Testing for Mobility 412  
CHAPTER 22 Software Configuration Management 437  
CHAPTER 23 Software Metrics and Analytics 460



**PART FOUR    MANAGING SOFTWARE PROJECTS    489**

---

- CHAPTER 24 Project Management Concepts    490  
CHAPTER 25 Creating a Viable Software Plan    504  
CHAPTER 26 Risk Management    532  
CHAPTER 27 A Strategy for Software Support    549

**PART FIVE    ADVANCED TOPICS    567**

---

- CHAPTER 28 Software Process Improvement    568  
CHAPTER 29 Emerging Trends in Software Engineering    583  
CHAPTER 30 Concluding Comments    602
- APPENDIX 1 An Introduction to UML    611  
APPENDIX 2 Data Science for Software Engineers    629  
REFERENCES    639  
INDEX    659

# TABLE OF CONTENTS

*Preface* xxvii

## **CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING 1**

---

- 1.1 The Nature of Software 4
  - 1.1.1 Defining Software 5
  - 1.1.2 Software Application Domains 7
  - 1.1.3 Legacy Software 8
- 1.2 Defining the Discipline 8
- 1.3 The Software Process 9
  - 1.3.1 The Process Framework 10
  - 1.3.2 Umbrella Activities 11
  - 1.3.3 Process Adaptation 11
- 1.4 Software Engineering Practice 12
  - 1.4.1 The Essence of Practice 12
  - 1.4.2 General Principles 14
- 1.5 How It All Starts 15
- 1.6 Summary 17

## **PART ONE THE SOFTWARE PROCESS 19**

---

### **CHAPTER 2 PROCESS MODELS 20**

---

- 2.1 A Generic Process Model 21
- 2.2 Defining a Framework Activity 23
- 2.3 Identifying a Task Set 23
- 2.4 Process Assessment and Improvement 24
- 2.5 Prescriptive Process Models 25
  - 2.5.1 The Waterfall Model 25
  - 2.5.2 Prototyping Process Model 26
  - 2.5.3 Evolutionary Process Model 29
  - 2.5.4 Unified Process Model 31
- 2.6 Product and Process 33
- 2.7 Summary 35

**CHAPTER 3 AGILITY AND PROCESS 37**

---

- 3.1 What Is Agility? 38
- 3.2 Agility and the Cost of Change 39
- 3.3 What Is an Agile Process? 40
  - 3.3.1 Agility Principles 40
  - 3.3.2 The Politics of Agile Development 41
- 3.4 Scrum 42
  - 3.4.1 Scrum Teams and Artifacts 43
  - 3.4.2 Sprint Planning Meeting 44
  - 3.4.3 Daily Scrum Meeting 44
  - 3.4.4 Sprint Review Meeting 45
  - 3.4.5 Sprint Retrospective 45
- 3.5 Other Agile Frameworks 46
  - 3.5.1 The XP Framework 46
  - 3.5.2 Kanban 48
  - 3.5.3 DevOps 50
- 3.6 Summary 51

**CHAPTER 4 RECOMMENDED PROCESS MODEL 54**

---

- 4.1 Requirements Definition 57
- 4.2 Preliminary Architectural Design 59
- 4.3 Resource Estimation 60
- 4.4 First Prototype Construction 61
- 4.5 Prototype Evaluation 64
- 4.6 Go, No-Go Decision 65
- 4.7 Prototype Evolution 67
  - 4.7.1 New Prototype Scope 67
  - 4.7.2 Constructing New Prototypes 68
  - 4.7.3 Testing New Prototypes 68
- 4.8 Prototype Release 68
- 4.9 Maintain Release Software 69
- 4.10 Summary 72

**CHAPTER 5 HUMAN ASPECTS OF SOFTWARE ENGINEERING 74**

---

- 5.1 Characteristics of a Software Engineer 75
- 5.2 The Psychology of Software Engineering 75



5.3	The Software Team	76
5.4	Team Structures	78
5.5	The Impact of Social Media	79
5.6	Global Teams	80
5.7	Summary	81

---

**PART TWO      MODELING    83**

---

**CHAPTER 6    PRINCIPLES THAT  
GUIDE PRACTICE    84**

---

6.1	Core Principles	85
6.1.1	Principles That Guide Process	85
6.1.2	Principles That Guide Practice	86
6.2	Principles That Guide Each Framework Activity	88
6.2.1	Communication Principles	88
6.2.2	Planning Principles	91
6.2.3	Modeling Principles	92
6.2.4	Construction Principles	95
6.2.5	Deployment Principles	98
6.3	Summary	100

---

**CHAPTER 7    UNDERSTANDING REQUIREMENTS    102**

---

7.1	Requirements Engineering	103
7.1.1	Inception	104
7.1.2	Elicitation	104
7.1.3	Elaboration	104
7.1.4	Negotiation	105
7.1.5	Specification	105
7.1.6	Validation	105
7.1.7	Requirements Management	106
7.2	Establishing the Groundwork	107
7.2.1	Identifying Stakeholders	107
7.2.2	Recognizing Multiple Viewpoints	107
7.2.3	Working Toward Collaboration	108
7.2.4	Asking the First Questions	108
7.2.5	Nonfunctional Requirements	109
7.2.6	Traceability	109

7.3	Requirements Gathering	110
7.3.1	Collaborative Requirements Gathering	110
7.3.2	Usage Scenarios	113
7.3.3	Elicitation Work Products	114
7.4	Developing Use Cases	114
7.5	Building the Analysis Model	118
7.5.1	Elements of the Analysis Model	119
7.5.2	Analysis Patterns	122
7.6	Negotiating Requirements	122
7.7	Requirements Monitoring	123
7.8	Validating Requirements	123
7.9	Summary	124

## **CHAPTER 8    REQUIREMENTS MODELING— A RECOMMENDED APPROACH    126**

---

8.1	Requirements Analysis	127
8.1.1	Overall Objectives and Philosophy	128
8.1.2	Analysis Rules of Thumb	128
8.1.3	Requirements Modeling Principles	129
8.2	Scenario-Based Modeling	130
8.2.1	Actors and User Profiles	131
8.2.2	Creating Use Cases	131
8.2.3	Documenting Use Cases	135
8.3	Class-Based Modeling	137
8.3.1	Identifying Analysis Classes	137
8.3.2	Defining Attributes and Operations	140
8.3.3	UML Class Models	141
8.3.4	Class-Responsibility-Collaborator Modeling	144
8.4	Functional Modeling	146
8.4.1	A Procedural View	146
8.4.2	UML Sequence Diagrams	148
8.5	Behavioral Modeling	149
8.5.1	Identifying Events with the Use Case	149
8.5.2	UML State Diagrams	150
8.5.3	UML Activity Diagrams	151
8.6	Summary	154

**CHAPTER 9 DESIGN CONCEPTS 156**

---

- 9.1 Design Within the Context of Software Engineering 157
- 9.2 The Design Process 159
  - 9.2.1 Software Quality Guidelines and Attributes 160
  - 9.2.2 The Evolution of Software Design 161
- 9.3 Design Concepts 163
  - 9.3.1 Abstraction 163
  - 9.3.2 Architecture 163
  - 9.3.3 Patterns 164
  - 9.3.4 Separation of Concerns 165
  - 9.3.5 Modularity 165
  - 9.3.6 Information Hiding 166
  - 9.3.7 Functional Independence 167
  - 9.3.8 Stepwise Refinement 167
  - 9.3.9 Refactoring 168
  - 9.3.10 Design Classes 169
- 9.4 The Design Model 171
  - 9.4.1 Design Modeling Principles 173
  - 9.4.2 Data Design Elements 174
  - 9.4.3 Architectural Design Elements 175
  - 9.4.4 Interface Design Elements 175
  - 9.4.5 Component-Level Design Elements 176
  - 9.4.6 Deployment-Level Design Elements 177
- 9.5 Summary 178

**CHAPTER 10 ARCHITECTURAL DESIGN—  
A RECOMMENDED APPROACH 181**

---

- 10.1 Software Architecture 182
  - 10.1.1 What Is Architecture? 182
  - 10.1.2 Why Is Architecture Important? 183
  - 10.1.3 Architectural Descriptions 183
  - 10.1.4 Architectural Decisions 184
- 10.2 Agility and Architecture 185
- 10.3 Architectural Styles 186
  - 10.3.1 A Brief Taxonomy of Architectural Styles 187
  - 10.3.2 Architectural Patterns 192
  - 10.3.3 Organization and Refinement 193



10.4	Architectural Considerations	193
10.5	Architectural Decisions	195
10.6	Architectural Design	196
10.6.1	Representing the System in Context	196
10.6.2	Defining Archetypes	197
10.6.3	Refining the Architecture into Components	198
10.6.4	Describing Instantiations of the System	200
10.7	Assessing Alternative Architectural Designs	201
10.7.1	Architectural Reviews	202
10.7.2	Pattern-Based Architecture Review	203
10.7.3	Architecture Conformance Checking	204
10.8	Summary	204

## **CHAPTER 11 COMPONENT-LEVEL DESIGN 206**

---

11.1	What Is a Component?	207
11.1.1	An Object-Oriented View	207
11.1.2	The Traditional View	209
11.1.3	A Process-Related View	211
11.2	Designing Class-Based Components	212
11.2.1	Basic Design Principles	212
11.2.2	Component-Level Design Guidelines	215
11.2.3	Cohesion	216
11.2.4	Coupling	218
11.3	Conducting Component-Level Design	219
11.4	Specialized Component-Level Design	225
11.4.1	Component-Level Design for WebApps	226
11.4.2	Component-Level Design for Mobile Apps	226
11.4.3	Designing Traditional Components	227
11.4.4	Component-Based Development	228
11.5	Component Refactoring	230
11.6	Summary	231

## **CHAPTER 12 USER EXPERIENCE DESIGN 233**

---

12.1	User Experience Design Elements	234
12.1.1	Information Architecture	235
12.1.2	User Interaction Design	236
12.1.3	Usability Engineering	236
12.1.4	Visual Design	237

12.2	The Golden Rules	238
12.2.1	Place the User in Control	238
12.2.2	Reduce the User's Memory Load	239
12.2.3	Make the Interface Consistent	240
12.3	User Interface Analysis and Design	241
12.3.1	Interface Analysis and Design Models	241
12.3.2	The Process	242
12.4	User Experience Analysis	243
12.4.1	User Research	244
12.4.2	User Modeling	245
12.4.3	Task Analysis	247
12.4.4	Work Environment Analysis	248
12.5	User Experience Design	249
12.6	User Interface Design	250
12.6.1	Applying Interface Design Steps	251
12.6.2	User Interface Design Patterns	252
12.7	Design Evaluation	253
12.7.1	Prototype Review	253
12.7.2	User Testing	255
12.8	Usability and Accessibility	255
12.8.1	Usability Guidelines	257
12.8.2	Accessibility Guidelines	259
12.9	Conventional Software UX and Mobility	261
12.10	Summary	261

## **CHAPTER 13 DESIGN FOR MOBILITY 264**

---

13.1	The Challenges	265
13.1.1	Development Considerations	265
13.1.2	Technical Considerations	266
13.2	Mobile Development Life Cycle	268
13.2.1	User Interface Design	270
13.2.2	Lessons Learned	271
13.3	Mobile Architectures	273
13.4	Context-Aware Apps	274
13.5	Web Design Pyramid	275
13.5.1	WebApp Interface Design	275
13.5.2	Aesthetic Design	277
13.5.3	Content Design	277
13.5.4	Architecture Design	278
13.5.5	Navigation Design	280

- 13.6 Component-Level Design 282
- 13.7 Mobility and Design Quality 282
- 13.8 Mobility Design Best Practices 285
- 13.9 Summary 287

## **CHAPTER 14 PATTERN-BASED DESIGN 289**

---

- 14.1 Design Patterns 290
  - 14.1.1 Kinds of Patterns 291
  - 14.1.2 Frameworks 293
  - 14.1.3 Describing a Pattern 293
  - 14.1.4 Machine Learning and Pattern Discovery 294
- 14.2 Pattern-Based Software Design 295
  - 14.2.1 Pattern-Based Design in Context 295
  - 14.2.2 Thinking in Patterns 296
  - 14.2.3 Design Tasks 297
  - 14.2.4 Building a Pattern-Organizing Table 298
  - 14.2.5 Common Design Mistakes 298
- 14.3 Architectural Patterns 299
- 14.4 Component-Level Design Patterns 300
- 14.5 Anti-Patterns 302
- 14.6 User Interface Design Patterns 304
- 14.7 Mobility Design Patterns 305
- 14.8 Summary 306

## **PART THREE QUALITY AND SECURITY 309**

---

### **CHAPTER 15 QUALITY CONCEPTS 310**

---

- 15.1 What Is Quality? 311
- 15.2 Software Quality 312
  - 15.2.1 Quality Factors 312
  - 15.2.2 Qualitative Quality Assessment 314
  - 15.2.3 Quantitative Quality Assessment 315
- 15.3 The Software Quality Dilemma 315
  - 15.3.1 “Good Enough” Software 316
  - 15.3.2 The Cost of Quality 317
  - 15.3.3 Risks 319
  - 15.3.4 Negligence and Liability 320

15.3.5	Quality and Security	320
15.3.6	The Impact of Management Actions	321
15.4	Achieving Software Quality	321
15.4.1	Software Engineering Methods	322
15.4.2	Project Management Techniques	322
15.4.3	Machine Learning and Defect Prediction	322
15.4.4	Quality Control	322
15.4.5	Quality Assurance	323
15.5	Summary	323

## **CHAPTER 16 REVIEWS—A RECOMMENDED APPROACH 325**

---

16.1	Cost Impact of Software Defects	326
16.2	Defect Amplification and Removal	327
16.3	Review Metrics and Their Use	327
16.4	Criteria for Types of Reviews	330
16.5	Informal Reviews	331
16.6	Formal Technical Reviews	332
16.6.1	The Review Meeting	332
16.6.2	Review Reporting and Record Keeping	333
16.6.3	Review Guidelines	334
16.7	Postmortem Evaluations	336
16.8	Agile Reviews	336
16.9	Summary	337

## **CHAPTER 17 SOFTWARE QUALITY ASSURANCE 339**

---

17.1	Background Issues	341
17.2	Elements of Software Quality Assurance	341
17.3	SQA Processes and Product Characteristics	343
17.4	SQA Tasks, Goals, and Metrics	343
17.4.1	SQA Tasks	343
17.4.2	Goals, Attributes, and Metrics	345
17.5	Formal Approaches to SQA	347
17.6	Statistical Software Quality Assurance	347

17.6.1	A Generic Example	347
17.6.2	Six Sigma for Software Engineering	349
17.7	Software Reliability	350
17.7.1	Measures of Reliability and Availability	350
17.7.2	Use of AI to Model Reliability	351
17.7.3	Software Safety	352
17.8	The ISO 9000 Quality Standards	353
17.9	The SQA Plan	354
17.10	Summary	355

## **CHAPTER 18 SOFTWARE SECURITY ENGINEERING 356**

---

18.1	Why Software Security Information Is Important	357
18.2	Security Life-Cycle Models	357
18.3	Secure Development Life-Cycle Activities	359
18.4	Security Requirements Engineering	360
18.4.1	SQUARE	360
18.4.2	The SQUARE Process	360
18.5	Misuse and Abuse Cases and Attack Patterns	363
18.6	Security Risk Analysis	364
18.7	Threat Modeling, Prioritization, and Mitigation	365
18.8	Attack Surface	366
18.9	Secure Coding	367
18.10	Measurement	368
18.11	Security Process Improvement and Maturity Models	370
18.12	Summary	370

## **CHAPTER 19 SOFTWARE TESTING—COMPONENT LEVEL 372**

---

19.1	A Strategic Approach to Software Testing	373
19.1.1	Verification and Validation	373
19.1.2	Organizing for Software Testing	374
19.1.3	The Big Picture	375
19.1.4	Criteria for “Done”	377

19.2	Planning and Recordkeeping	378
19.2.1	Role of Scaffolding	379
19.2.2	Cost-Effective Testing	379
19.3	Test-Case Design	381
19.3.1	Requirements and Use Cases	382
19.3.2	Traceability	383
19.4	White-Box Testing	383
19.4.1	Basis Path Testing	384
19.4.2	Control Structure Testing	386
19.5	Black-Box Testing	388
19.5.1	Interface Testing	388
19.5.2	Equivalence Partitioning	389
19.5.3	Boundary Value Analysis	389
19.6	Object-Oriented Testing	390
19.6.1	Class Testing	390
19.6.2	Behavioral Testing	392
19.7	Summary	393

## **CHAPTER 20 SOFTWARE TESTING— INTEGRATION LEVEL 395**

---

20.1	Software Testing Fundamentals	396
20.1.1	Black-Box Testing	397
20.1.2	White-Box Testing	397
20.2	Integration Testing	398
20.2.1	Top-Down Integration	398
20.2.2	Bottom-Up Integration	399
20.2.3	Continuous Integration	400
20.2.4	Integration Test Work Products	402
20.3	Artificial Intelligence and Regression Testing	402
20.4	Integration Testing in the OO Context	404
20.4.1	Fault-Based Test-Case Design	405
20.4.2	Scenario-Based Test-Case Design	406
20.5	Validation Testing	407
20.6	Testing Patterns	409
20.7	Summary	409

## **CHAPTER 21 SOFTWARE TESTING—SPECIALIZED TESTING FOR MOBILITY 412**

---

- 21.1 Mobile Testing Guidelines 413
- 21.2 The Testing Strategies 414
- 21.3 User Experience Testing Issues 415
  - 21.3.1 Gesture Testing 415
  - 21.3.2 Virtual Keyboard Input 416
  - 21.3.3 Voice Input and Recognition 416
  - 21.3.4 Alerts and Extraordinary Conditions 417
- 21.4 Web Application Testing 418
- 21.5 Web Testing Strategies 418
  - 21.5.1 Content Testing 420
  - 21.5.2 Interface Testing 421
  - 21.5.3 Navigation Testing 421
- 21.6 Internationalization 423
- 21.7 Security Testing 423
- 21.8 Performance Testing 424
- 21.9 Real-Time Testing 426
- 21.10 Testing AI Systems 428
  - 21.10.1 Static and Dynamic Testing 429
  - 21.10.2 Model-Based Testing 429
- 21.11 Testing Virtual Environments 430
  - 21.11.1 Usability Testing 430
  - 21.11.2 Accessibility Testing 433
  - 21.11.3 Playability Testing 433
- 21.12 Testing Documentation and Help Facilities 434
- 21.13 Summary 435

## **CHAPTER 22 SOFTWARE CONFIGURATION MANAGEMENT 437**

---

- 22.1 Software Configuration Management 438
  - 22.1.1 An SCM Scenario 439
  - 22.1.2 Elements of a Configuration Management System 440
  - 22.1.3 Baselines 441

22.1.4	Software Configuration Items	441
22.1.5	Management of Dependencies and Changes	442
22.2	The SCM Repository	443
22.2.1	General Features and Content	444
22.2.2	SCM Features	444
22.3	Version Control Systems	445
22.4	Continuous Integration	446
22.5	The Change Management Process	447
22.5.1	Change Control	448
22.5.2	Impact Management	451
22.5.3	Configuration Audit	452
22.5.4	Status Reporting	452
22.6	Mobility and Agile Change Management	453
22.6.1	e-Change Control	453
22.6.2	Content Management	455
22.6.3	Integration and Publishing	455
22.6.4	Version Control	457
22.6.5	Auditing and Reporting	458
22.7	Summary	458

## **CHAPTER 23 SOFTWARE METRICS AND ANALYTICS 460**

---

23.1	Software Measurement	461
23.1.1	Measures, Metrics, and Indicators	461
23.1.2	Attributes of Effective Software Metrics	462
23.2	Software Analytics	462
23.3	Product Metrics	463
23.3.1	Metrics for the Requirements Model	464
23.3.2	Design Metrics for Conventional Software	466
23.3.3	Design Metrics for Object-Oriented Software	468
23.3.4	User Interface Design Metrics	471
23.3.5	Metrics for Source Code	473
23.4	Metrics for Testing	474
23.5	Metrics for Maintenance	476
23.6	Process and Project Metrics	476



23.7	Software Measurement	479
23.8	Metrics for Software Quality	482
23.9	Establishing Software Metrics Programs	485
23.10	Summary	487

---

## **PART FOUR    MANAGING SOFTWARE PROJECTS    489**

---

### **CHAPTER 24    PROJECT MANAGEMENT CONCEPTS    490**

---

24.1	The Management Spectrum	491
24.1.1	The People	491
24.1.2	The Product	491
24.1.3	The Process	492
24.1.4	The Project	492
24.2	People	493
24.2.1	The Stakeholders	493
24.2.2	Team Leaders	493
24.2.3	The Software Team	494
24.2.4	Coordination and Communications Issues	496
24.3	Product	497
24.3.1	Software Scope	497
24.3.2	Problem Decomposition	497
24.4	Process	498
24.4.1	Melding the Product and the Process	498
24.4.2	Process Decomposition	498
24.5	Project	500
24.6	The W <sup>5</sup> HH Principle	501
24.7	Critical Practices	502
24.8	Summary	502

---

### **CHAPTER 25    CREATING A VIABLE SOFTWARE PLAN    504**

---

25.1	Comments on Estimation	505
25.2	The Project Planning Process	506

25.3	Software Scope and Feasibility	507
25.4	Resources	507
25.4.1	Human Resources	508
25.4.2	Reusable Software Resources	509
25.4.3	Environmental Resources	509
25.5	Data Analytics and Software Project Estimation	509
25.6	Decomposition and Estimation Techniques	511
25.6.1	Software Sizing	511
25.6.2	Problem-Based Estimation	512
25.6.3	An Example of LOC-Based Estimation	512
25.6.4	An Example of FP-Based Estimation	514
25.6.5	An Example of Process-Based Estimation	515
25.6.6	An Example of Estimation Using Use Case Points	517
25.6.7	Reconciling Estimates	518
25.6.8	Estimation for Agile Development	519
25.7	Project Scheduling	520
25.7.1	Basic Principles	521
25.7.2	The Relationship Between People and Effort	522
25.8	Defining a Project Task Set	523
25.8.1	A Task Set Example	524
25.8.2	Refinement of Major Tasks	524
25.9	Defining a Task Network	525
25.10	Scheduling	226
25.10.1	Time-Line Charts	526
25.10.2	Tracking the Schedule	528
25.11	Summary	530

## **CHAPTER 26 RISK MANAGEMENT 532**

---

26.1	Reactive Versus Proactive Risk Strategies	533
26.2	Software Risks	534
26.3	Risk Identification	535
26.3.1	Assessing Overall Project Risk	536
26.3.2	Risk Components and Drivers	537
26.4	Risk Projection	538

26.4.1	Developing a Risk Table	538
26.4.2	Assessing Risk Impact	540
26.5	Risk Refinement	542
26.6	Risk Mitigation, Monitoring, and Management	543
26.7	The RMMM Plan	546
26.8	Summary	547

## **CHAPTER 27 A STRATEGY FOR SOFTWARE SUPPORT 549**

---

27.1	Software Support	550
27.2	Software Maintenance	552
27.2.1	Maintenance Types	553
27.2.2	Maintenance Tasks	554
27.2.3	Reverse Engineering	555
27.3	Proactive Software Support	557
27.3.1	Use of Software Analytics	558
27.3.2	Role of Social Media	559
27.3.3	Cost of Support	559
27.4	Refactoring	560
27.4.1	Data Refactoring	561
27.4.2	Code Refactoring	561
27.4.3	Architecture Refactoring	561
27.5	Software Evolution	562
27.5.1	Inventory Analysis	563
27.5.2	Document Restructuring	564
27.5.3	Reverse Engineering	564
27.5.4	Code Refactoring	564
27.5.5	Data Refactoring	564
27.5.6	Forward Engineering	565
27.6	Summary	565

## **PART FIVE ADVANCED TOPICS 567**

---

### **CHAPTER 28 SOFTWARE PROCESS IMPROVEMENT 568**

---

28.1	What Is SPI?	569
28.1.1	Approaches to SPI	569
28.1.2	Maturity Models	570
28.1.3	Is SPI for Everyone?	571

28.2	The SPI Process	571
28.2.1	Assessment and GAP Analysis	572
28.2.2	Education and Training	573
28.2.3	Selection and Justification	573
28.2.4	Installation/Migration	574
28.2.5	Evaluation	575
28.2.6	Risk Management for SPI	575
28.3	The CMMI	576
28.4	Other SPI Frameworks	579
28.4.1	SPICE	579
28.4.2	TickIT Plus	579
28.5	SPI Return on Investment	580
28.6	SPI Trends	580
28.7	Summary	581

## **CHAPTER 29 EMERGING TRENDS IN SOFTWARE ENGINEERING 583**

---

29.1	Technology Evolution	584
29.2	Software Engineering as a Discipline	585
29.3	Observing Software Engineering Trends	586
29.4	Identifying “Soft Trends”	587
29.4.1	Managing Complexity	588
29.4.2	Open-World Software	589
29.4.3	Emergent Requirements	590
29.4.4	The Talent Mix	591
29.4.5	Software Building Blocks	591
29.4.6	Changing Perceptions of “Value”	592
29.4.7	Open Source	592
29.5	Technology Directions	593
29.5.1	Process Trends	593
29.5.2	The Grand Challenge	594
29.5.3	Collaborative Development	595
29.5.4	Requirements Engineering	596
29.5.5	Model-Driven Software Development	596
29.5.6	Search-Based Software Engineering	597
29.5.7	Test-Driven Development	598
29.6	Tools-Related Trends	599
29.7	Summary	600

**CHAPTER 30 CONCLUDING COMMENTS 602**

---

30.1	The Importance of Software—Revisited	603
30.2	People and the Way They Build Systems	603
30.3	Knowledge Discovery	605
30.4	The Long View	606
30.5	The Software Engineer’s Responsibility	607
30.6	A Final Comment from RSP	609
APPENDIX 1	An Introduction to UML	611
APPENDIX 2	Data Science for Software Engineers	629
REFERENCES		639
INDEX		659

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 nearly four 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 or coder 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.

## NEW TO THE NINTH EDITION

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

The ninth 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, we have further enhanced the popular “support system” for the book, providing a comprehensive set of student, instructor, and professional resources to complement the content of the book.

Readers of the past few editions of *Software Engineering: A Practitioner’s Approach* will note that the ninth edition has actually been reduced in page length. Our goal was concision, making the book stronger from a pedagogical viewpoint and less daunting for the reader who desires to journey through the entire book. An anecdote, attributed to Blaise Pascal, the famous mathematician and physicist, goes like this: In writing a overly long letter to a friend, Pascal ended with this sentence. “I wanted to write you a shorter letter, but I didn’t have the time.” As we worked on concision for the ninth edition, we came to appreciate Pascal’s words.

The 30 chapters of the ninth edition are organized into five parts. This organization better compartmentalizes topics and assists instructors who may not have the time to complete the entire book in one term.

Part 1, *The Software Process*, presents a variety of different views of software process, considering several important process models and frameworks that allow us to address 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 mobility computing applications are also considered. The coverage of user experience design has been expanded in this section. Part 3, *Quality and Security*, 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, we present software security practices that can be inserted into incremental software development models. 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. Boxed features are included 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.

The five-part organization of the ninth 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 ninth edition in this way,

we have attempted to provide an instructor with a number of teaching options. In every case the content of the ninth edition is complemented by the following elements of the *SEPA, 9/e Support System*.

### Additional Resources

A wide variety of resources can be accessed through the instructor website, including an extensive online learning center encompassing problem solutions, a variety of Web-based resources with software engineering checklists, an evolving collection of “tiny tools,” and comprehensive case studies. *Professional Resources* provide several hundred categorized web references which allow students to explore software engineering in greater detail, along with a reference library with links to several hundred downloadable references providing an in-depth source of advanced software engineering information. Additionally, a complete online *Instructor’s Guide* and supplementary teaching materials along with several hundred PowerPoint slides that may be used for lectures are included.

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.

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

Bruce Maxim has taken the lead in developing new content for the ninth edition of *Software Engineering: A Practitioner’s Approach*, while Roger Pressman has served as editor-in-chief as well as providing contributions in select circumstances.

**Acknowledgments** Special thanks go to Nancy Mead from Software Engineering Institute at Carnegie Mellon University who wrote the chapter on software security engineering; Tim Lethbridge of the University of Ottawa who assisted us in the development of UML and OCL examples and developed the case study that accompanies this book; Dale Skrien of Colby College who developed the UML tutorial in Appendix 1; William Grosky of the University of Michigan–Dearborn who developed the overview of data science in Appendix 2 with his student Terry Ruas; and our Australian colleague Margaret Kellow for updating the pedagogical Web materials that accompany this book. In addition, we would like to thank Austin Krauss for providing insight into software development in the video game industry from his perspective as a senior software engineer.

**Special Thanks** BRM: I am grateful to have had the opportunity to work with Roger on the ninth edition of this book. During the time I have been working on this book, my son, Benjamin, has become a software engineering manager and my daughter, Katherine, used her art background to create the figures that appear in the book chapters. I am quite pleased to see the adults they have become and enjoy my time with their children



(Isla, Emma, and Thelma). I am very grateful to my wife, Norma, for her enthusiastic support as I filled my free time working on this book.

RSP: 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. After many years of following our own professional paths, the three of us now work together in a business that we founded in 2012. Nothing has filled me with more pride. Both of my sons now have children of their own, Maya and Lily, who start still another generation. Finally, to my wife, Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of “the book.”

*Bruce R. Maxim*  
*Roger S. Pressman*

# Affordability & Outcomes = Academic Freedom!

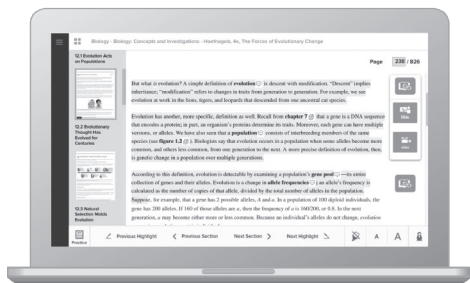
You deserve choice, flexibility and control. You know what's best for your students and selecting the course materials that will help them succeed should be in your hands.

That's why providing you with a wide range of options that lower costs and drive better outcomes is our highest priority.



# connect

Students—study more efficiently, retain more and achieve better outcomes. Instructors—focus on what you love—teaching.



Laptop: McGraw-Hill Education

## They'll thank you for it.

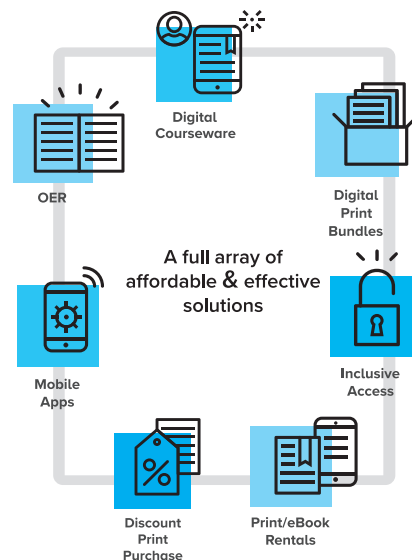
Study resources in Connect help your students be better prepared in less time. You can transform your class time from dull definitions to dynamic discussion. Hear from your peers about the benefits of Connect at [www.mheducation.com/highered/connect/smartbook](http://www.mheducation.com/highered/connect/smartbook)

## Make it simple, make it affordable.

Connect makes it easy with seamless integration using any of the major Learning Management Systems—Blackboard®, Canvas, and D2L, among others—to let you organize your course in one convenient location. Give your students access to digital materials at a discount with our inclusive access program. Ask your McGraw-Hill representative for more information.

## Learning for everyone.

McGraw-Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services office and ask them to email [accessibility@mheducation.com](mailto:accessibility@mheducation.com), or visit [www.mheducation.com/about/accessibility.html](http://www.mheducation.com/about/accessibility.html) for more information.



Learn more at: [www.mheducation.com/realvalue](http://www.mheducation.com/realvalue)



### Rent It

Affordable print and digital rental options through our partnerships with leading textbook distributors including Amazon, Barnes & Noble, Chegg, Follett, and more.



### Go Digital

A full and flexible range of affordable digital solutions ranging from Connect, ALEKS, inclusive access, mobile apps, OER and more.



### Get Print

Students who purchase digital materials can get a loose-leaf print version at a significantly reduced rate to meet their individual preferences and budget.



# SOFTWARE AND SOFTWARE ENGINEERING

As he finished showing me the latest build of one of the world's most popular first-person shooter video games, the young developer laughed.

“You’re not a gamer, are you?” he asked.

I smiled. “How’d you guess?”

The young man was dressed in shorts and a tee shirt. His leg bounced up and down like a piston, burning the nervous energy that seemed to be commonplace among his co-workers.

## KEY CONCEPTS

application domains . . . . .	7	process . . . . .	9
failure curves . . . . .	5	questions about . . . . .	4
framework activities . . . . .	10	software engineering, definition . . . . .	3
general principles . . . . .	14	layers . . . . .	9
legacy software . . . . .	8	practice . . . . .	12
principles . . . . .	14	umbrella activities . . . . .	11
problem solving . . . . .	12	wear . . . . .	5
<i>SafeHome</i> . . . . .	16		
software, definition . . . . .	5		
nature of . . . . .	4		

## QUICK LOOK

**What is it?** Computer software is a work product that software professionals build and then support over many years. These work products include programs that execute within computers of any size and architecture. 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. Software engineers apply the software engineering process.

**Why is it important?** Software engineering is important because it enables us to build complex systems in a timely manner and with high quality. It imposes discipline to work that can become quite chaotic, but it also allows the

people who build computer software to adapt their approach in a manner that best suits their needs.

**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.

**What is the work product?** From the software engineer’s point of view, the work product is the set of programs, content (data), and other work products that support computer software. But from the user’s point of view, the work product is a tool or product 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.

“Because if you were,” he said, “you’d be a lot more excited. You’ve gotten a peek at our next generation product and that’s something that our customers would kill for . . . no pun intended.”

We sat in a development area at one of the most successful game developers on the planet. Over the years, earlier generations of the game he demoed sold over 50 million copies and generated billions of dollars in revenue.

“So, when will this version be on the market?” I asked.

He shrugged. “In about five months, and we’ve still got a lot of work to do.”

He had responsibility for game play and artificial intelligence functionality in an application that encompassed more than three million lines of code.

“Do you guys use any software engineering techniques?” I asked, half-expecting that he’d laugh and shake his head.

He paused and thought for a moment. Then he slowly nodded. “We adapt them to our needs, but sure, we use them.”

“Where?” I asked, probing.

“Our problem is often translating the requirements the creatives give us.”

“The creatives?” I interrupted.

“You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game.”

“And after the requirements are established?”

He shrugged. “We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends.”

“You know my book?” I was honestly surprised.

“Sure, used it in school. There’s a lot there.”

“I’ve talked to some of your buddies here, and they’re more skeptical about the stuff in my book.”

He frowned. “Look, we’re not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques.”

“And how will your subset change as the years pass?”

He paused as if to ponder the future. “Games will become bigger and more complex, that’s for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don’t, we’re dead.”

\*\*\*\*\*

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. Sixty 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 media); that software would be the driving force behind the personal computer revolution; that software applications would be purchased by

consumers using their mobile devices; 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 all industrial-era companies; or that a vast software-driven network 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.

As software’s importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and support 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*.

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

- Software has become deeply embedded in virtually every aspect of our lives. The number of people who have an interest in the features and functions provided by a specific application<sup>1</sup> has grown dramatically. *A concerted effort should be made to understand the problem before a software solution is developed.*
- The information technology requirements demanded by individuals, businesses, and governments grow increasingly complex with each passing year. Large teams of people now create computer programs. 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 autonomous vehicles. *Design has become a pivotal activity.*
- 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 consequences. *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 increase, demands for adaptation and enhancement will also grow. *Software should be maintainable.*

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

---

<sup>1</sup> We will call these people “stakeholders” later in this book.

## 1.1 THE NATURE OF SOFTWARE

Today, software takes on a dual role. It is a product, and 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 device, on the desktop, in the cloud, or within a mainframe computer or autonomous machine, 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 an augmented-reality representation derived from data acquired from dozens of independent sources and then overlaid on the real world. As the vehicle used to deliver a 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 its forms. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

The role of computer software has undergone significant change over the last 60 years. 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 and protect 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 programmer are the same questions that are asked when modern computer-based systems are built:<sup>2</sup>

- 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?

---

<sup>2</sup> 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.”

These, and many other questions, are a manifestation of the concern about software and how it is developed—a concern that has led 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:

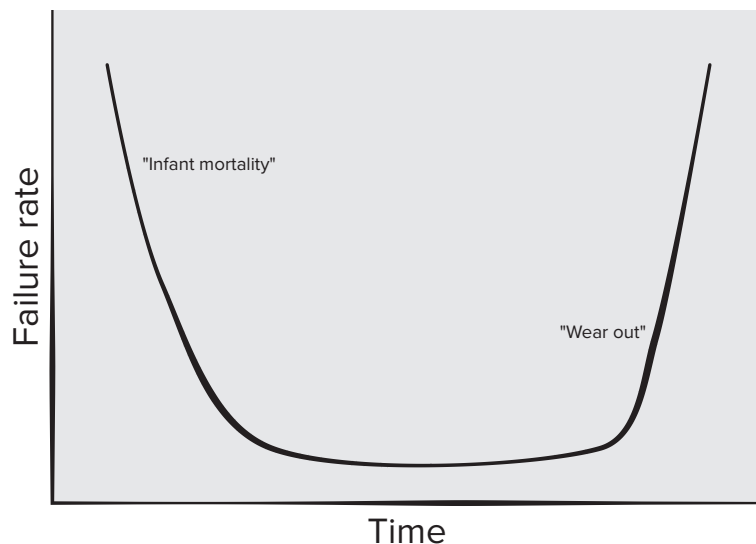
Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information; and (3) descriptive information in both hard copy and virtual forms that describes the operation 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 one fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*

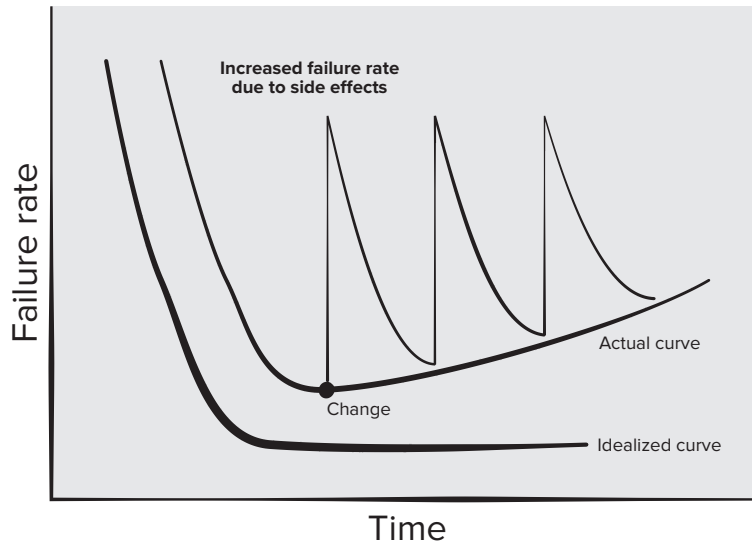
Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected, and 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,

**FIGURE 1.1**

**Failure curve  
for hardware**





**FIGURE 1.2****Failure curves for software**

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!*

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,<sup>3</sup> 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.

<sup>3</sup> 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.

## 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,<sup>4</sup> information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

**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.

**Engineering/scientific software.** A broad array of “number-crunching” or data science programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, from computer-aided design to consumer spending habits, and from genetic analysis to meteorology.

**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.** Composed of reusable components and designed to provide specific capabilities for use by many different customers. It may focus on a limited and esoteric marketplace (e.g., inventory control products) or attempt to address the mass consumer market.

**Web/mobile applications.** This network-centric software category spans a wide array of applications and encompasses browser-based apps, cloud computing, service-based computing, and software that resides on mobile devices.

**Artificial intelligence software.** Makes use of heuristics<sup>5</sup> to solve complex problems that are not amenable to regular computation or straightforward analysis. Applications within this area include robotics, decision-making systems, pattern recognition (image and voice), machine learning, theorem proving, and game playing.

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 on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.

---

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.

5 The use of heuristics is an approach to problem solving that employs a practical method or “rule of thumb” not guaranteed to be perfect, but sufficient for the task at hand.

### 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. 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.

These changes may create an additional side effect that is often present in legacy software—*poor quality*.<sup>6</sup> Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, and a poorly managed change history. The list can be quite long. And yet, these systems often support “core 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 work with other more modern systems or databases.
- The software must be re-architected to make it viable within an evolving computing environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable in 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 change continually, new software systems can be built from the old ones, and . . . all must interact and cooperate with each other” [Day99].

## 1.2 DEFINING THE DISCIPLINE

The IEEE [IEE17] has developed the following definition for software engineering:

Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

<sup>6</sup> 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.

FIGURE 1.3

Software engineering layers



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. You may have heard of total quality management (TQM) or Six Sigma, and similar philosophies<sup>7</sup> that foster a culture of continuous process improvement. It is this culture that ultimately leads to 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 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 semi-automated 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.

## 1.3 THE SOFTWARE PROCESS

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

<sup>7</sup> Quality management and related approaches are discussed throughout Part Three of this book.