



SOFTWARE ENGINEERING

Modern Approaches

Second Edition

Eric J. Braude
Michael E. Bernstein

SOFTWARE ENGINEERING

Modern Approaches

Second Edition

Eric J. Braude

Boston University, Metropolitan College

Michael E. Bernstein

Boston University, Metropolitan College



Long Grove, Illinois

For information about this book, contact:

Waveland Press, Inc.
4180 IL Route 83, Suite 101
Long Grove, IL 60047-9580
(847) 634-0081
info@waveland.com
www.waveland.com

Copyright © 2011 by Eric J. Braude and Michael E. Bernstein
Reissued 2016 by Waveland Press, Inc.

10-digit ISBN 1-4786-3230-5
13-digit ISBN 978-1-4786-3230-6

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without permission in writing from the publisher.

Printed in the United States of America

7 6 5 4 3 2 1

*For Judy (Eric J. Braude)
To Bambi, Garrett and Reid,
for all their love and support (Michael E. Bernstein)*

Brief Contents

Preface xiv

Acknowledgments xvii

Part I Introduction to Software Engineering

- 1 The Goals and Terminology of Software Engineering 1
 - 2 Introduction to Quality and Metrics in Software Engineering 21
-

Part II Software Process

- 3 Software Process 32
 - 4 Agile Software Processes 63
 - 5 Quality in the Software Process 80
 - 6 Software Configuration Management 120
-

Part III Project Management

- 7 Principles of Software Project Management I 140
 - 8 Principles of Software Project Management II 168
 - 9 Quality and Metrics in Project Management 213
-

Part IV Requirement Analysis

- 10 Principles of Requirements Analysis 230
 - 11 Analyzing High-Level Requirements 245
 - 12 Analyzing Detailed Requirements 278
 - 13 Quality and Metrics in Requirements Analysis 331
 - 14 Formal and Emerging Methods in Requirements Analysis (Online chapter) 349
-

Part V Software Design

- 15 Principles of Software Design 350
 - 16 The Unified Modeling Language 361
 - 17 Software Design Patterns 383
 - 18 Software Architecture 438
 - 19 Detailed Design 476
 - 20 Design Quality and Metrics 508
 - 21 Advanced and Emerging Methods in Software Design (Online chapter) 538
-

Part VI Implementation

- 22 Principles of Implementation 539
 - 23 Quality and Metrics in Implementation 584
 - 24 Refactoring 601
-

Part VII Testing and Maintenance

- 25 Introduction to Software Testing 621
 - 26 Unit Testing 630
 - 27 Module and Integration Testing 666
 - 28 Testing at the System Level 694
 - 29 Software Maintenance 730
-

Glossary 759

Index 767

Contents

Preface	xiv
The Issue of Scale	xiv
This Edition Compared with the First	xiv
How Instructors Can Use This Book	xv
Acknowledgments	xvii
PART I Introduction to Software Engineering	
1 The Goals and Terminology of Software Engineering	1
1.1 What is Software Engineering	2
1.2 Why Software Engineering Is Critical: Software Disasters	3
1.3 Why Software Fails or Succeeds	4
1.4 Software Engineering Activities	5
1.5 Software Engineering Principles	10
1.6 Ethics in Software Engineering	12
1.7 Case Studies	14
1.8 Summary	19
1.9 Exercises	19
Bibliography	20
2 Introduction to Quality and Metrics in Software Engineering	21
2.1 The Meaning of Software Quality	22
2.2 Defects in Software	23
2.3 Verification and Validation	25
2.4 Planning for Quality	27
2.5 Metrics	28
2.6 Summary	30
2.7 Exercises	31
Bibliography	31
PART II Software Process	
3 Software Process	32
3.1 The Activities of Software Process	33
3.2 Software Process Models	37
3.3 Case Study: Student Team Guidance	55

3.4	Summary	59
3.5	Exercises	60
	Bibliography	62
4	Agile Software Processes	63
4.1	Agile History and Agile Manifesto	64
4.2	Agile Principles	65
4.3	Agile Methods	66
4.4	Agile Processes	68
4.5	Integrating Agile with Non-Agile Processes	74
4.6	Summary	77
4.7	Exercises	78
	Bibliography	79
5	Quality in the Software Process	80
5.1	Principles of Managing Quality	81
5.2	Managing Quality in Agile Processes	82
5.3	Quality Planning	83
5.4	Inspections	87
5.5	QA Reviews and Audits	92
5.6	Defect Management	93
5.7	Process Improvement and Process Metrics	96
5.8	Organization-Level Quality and the CMMI	100
5.9	Case Study: Software Quality Assurance Plan for Encounter	103
5.10	Summary	118
5.11	Exercises	118
	Bibliography	119
6	Software Configuration Management	120
6.1	Software Configuration Management Goals	121
6.2	SCM Activities	121
6.3	Configuration Management Plans	128
6.4	Configuration Management Systems	128
6.5	Case Study: Encounter Video Game	129
6.6	Case Study: Eclipse	134
6.7	Student Team Guidance: Configuration Management	136
6.8	Summary	137
6.9	Exercises	138
	Bibliography	139
PART III Project Management		
7	Principles of Software Project Management I: Organization, Tools, and Risk Management	140
7.1	Software Project Organization	142
7.2	Team Size	144
7.3	Geographically Distributed Development	146
7.4	The Team Software Process	151
7.5	Software Project Tools and Techniques	153

7.6	Risk Management	159
7.7	Student Team Guidance: Organizing the Software Project's Management	162
7.8	Summary	165
7.9	Exercises	166
	Bibliography	167
8	Principles of Software Project Management II: Estimation, Scheduling, and Planning	168
8.1	Cost Estimation	169
8.2	Scheduling	182
8.3	The Software Project Management Plan	185
8.4	Case Study: Encounter Project Management Plan	187
8.5	Case Study: Project Management in Eclipse	196
8.6	Case Study: Project Management for OpenOffice	205
8.7	Case Study: Student Team Guidance	208
8.8	Summary	210
8.9	Exercises	211
	Bibliography	212
9	Quality and Metrics in Project Management	213
9.1	Cultivating and Planning Internal Quality	214
9.2	Project Metrics	215
9.3	Using Metrics for Improvement	219
9.4	Software Verification and Validation Plan	223
9.5	Case Study: Software Verification and Validation Plan for Encounter	225
9.6	Summary	228
9.7	Exercises	228
	Bibliography	229
PART IV Requirement Analysis		
10	Principles of Requirements Analysis	230
10.1	The Value of Requirements Analysis	231
10.2	Sources of Requirements	231
10.3	<i>High-level</i> vs. <i>Detailed</i> Requirements	232
10.4	Types of Requirements	233
10.5	Nonfunctional Requirements	233
10.6	Documenting Requirements	238
10.7	Traceability	239
10.8	Agile Methods and Requirements	239
10.9	Updating the Project to Reflect Requirements Analysis	241
10.10	Summary	243
10.11	Exercises	244
	Bibliography	244
11	Analyzing High-Level Requirements	245
11.1	Examples of Customer Wants	246
11.2	Stakeholder Vision	247
11.3	The Interview and Documentation Process	248

11.4	Writing an Overview	249
11.5	Describing Main Functions and Use Cases	249
11.6	Agile Methods for High-Level Requirements	252
11.7	Specifying User Interfaces: High Level	254
11.8	Security Requirements	258
11.9	Using Diagrams for High-Level Requirement	260
11.10	Case Study: High-Level Software Requirements Specification (SRS) for the Encounter Video Game	264
11.11	Case Study: High-Level Requirements for Eclipse	268
11.12	Eclipse Platform Subproject (First of three)	269
11.13	Case Study: High-Level Requirements for OpenOffice	273
11.14	Summary	275
11.15	Exercises	275
	Bibliography	276
12	Analyzing Detailed Requirements	278
12.1	The Meaning of Detailed Requirements	279
12.2	Organizing Detailed Requirements	280
12.3	User Interfaces: Detailed Requirements	291
12.4	Detailed Security Requirements	296
12.5	Error Conditions	296
12.6	Traceability of Detailed Requirements	297
12.7	Using Detailed Requirements to Manage Projects	300
12.8	Prioritizing Requirements	301
12.9	Associating Requirements with Tests	302
12.10	Agile Methods for Detailed Requirements	303
12.11	Using Tools and the Web for Requirements Analysis	305
12.12	The Effects on Projects of the Detailed Requirements Process	308
12.13	Student Project Guide: Requirements for the Encounter Case Study	309
12.14	Case Study: Detailed Requirements for the Encounter Video Game	315
12.15	Summary	328
12.16	Exercises	329
	Bibliography	330
13	Quality and Metrics in Requirements Analysis	331
13.1	Quality of Requirements for Agile Projects	332
13.2	Accessibility of Requirements	332
13.3	Comprehensiveness of Requirements	333
13.4	Understandability of Requirements	335
13.5	Unambiguity of Requirements	335
13.6	Consistency of Requirements	336
13.7	Prioritization of Requirements	337
13.8	Security and High-Level Requirements	338
13.9	Self-Completeness of Requirements	339
13.10	Testability of Requirements	340
13.11	Traceability of Requirements	342
13.12	Metrics for Requirements Analysis	343

13.13	Inspecting Detailed Requirements	344
13.14	Summary	347
13.15	Exercises	348
14	Formal and Emerging Methods in Requirements Analysis: An Introduction	
	(Online Chapter)	349
14.1	Provable Requirements Method	
14.2	Introduction to Formal Methods	
14.3	Mathematical Preliminaries	
14.4	The Z-Specification Language	
14.5	The B Language System	
14.6	Trade-offs for Using a B-like system	
14.7	Summary	
14.8	Exercises	
	Bibliography	
PART V	Software Design	
15	Principles of Software Design	350
15.1	The Goals of Software Design	351
15.2	Integrating Design Models	354
15.3	Frameworks	357
15.4	IEEE Standards for Expressing Designs	359
15.5	Summary	359
15.6	Exercises	360
16	The Unified Modeling Language	361
16.1	Classes in UML	362
16.2	Class Relationships in UML	362
16.3	Multiplicity	364
16.4	Inheritance	364
16.5	Sequence Diagrams	368
16.6	State Diagrams	372
16.7	Activity Diagrams	374
16.8	Data Flow Models	376
16.9	A Design Example with UML	377
16.10	Summary	380
16.11	Exercises	381
	Bibliography	382
17	Software Design Patterns	383
17.1	Examples of a Recurring Design Purpose	384
17.2	An Introduction to Design Patterns	386
17.3	Summary of Design Patterns by Type: Creational, Structural, and Behavioral	390
17.4	Characteristics of Design Patterns: Viewpoints, Roles, and Levels	396
17.5	Selected Creational Design Patterns	400
17.6	Selected Structural Design Patterns	408

17.7	Selected Behavioral Design Patterns	417
17.8	Design Pattern Forms: Delegation and Recursion	431
17.9	Summary	435
17.10	Exercises	436
	Bibliography	437
18	Software Architecture	438
18.1	A Categorization of Architectures	439
18.2	Software Architecture Alternatives and Their Class Models	439
18.3	Trading Off Architecture Alternatives	453
18.4	Tools for Architectures	454
18.5	IEEE Standards for Expressing Designs	455
18.6	Effects of Architecture Selection on the Project Plan	455
18.7	Case Study: Preparing to Design Encounter (Student Project Guide continued)	457
18.8	Case Study: Software Design Document for the <i>Role-Playing Video Game</i> Framework	460
18.9	Case Study: Software Design Document for Encounter (Uses the Framework)	462
18.10	Case Study: Architecture of Eclipse	466
18.11	Case Study: OpenOffice Architecture	468
18.12	Summary	473
18.13	Exercises	474
	Bibliography	475
19	Detailed Design	476
19.1	Relating Use Cases, Architecture, and Detailed Design	477
19.2	A Typical Road Map for the "Detailed Design" Process	478
19.3	Object-Oriented Design Principles	479
19.4	Designing against Interfaces	481
19.5	Specifying Classes, Functions, and Algorithms	482
19.6	Reusing Components	485
19.7	Sequence and Data Flow Diagrams for Detailed Design	486
19.8	Detailed Design and Agile Processes	490
19.9	Design in the Unified Development Process	490
19.10	IEEE Standard 890 for Detailed Design	491
19.11	Updating a Project with Detailed Design	491
19.12	Case Study: Detailed Design of Encounter	494
19.13	Case Study: Detailed Design of Eclipse	503
19.14	Summary	505
19.15	Exercises	505
	Bibliography	507
20	Design Quality and Metrics	508
20.1	Degree of Understandability, Cohesion, and Coupling	510
20.2	Degree of Sufficiency as a Quality Goal	510
20.3	Degree of Robustness as a Quality Goal	511
20.4	Degree of Flexibility as a Design Quality Goal	512
20.5	Degree of Reusability as a Design Quality Goal	513
20.6	Degree of Time Efficiency as a Design Quality Measure	517

20.7	Degree of Space Efficiency as a Design Quality Measure	519
20.8	Degree of Reliability as a Design Quality Measure	521
20.9	Degree of Security as a Design Quality Measure	523
20.10	Assessing Quality in Architecture Selection	525
20.11	Assessing the Quality of Detailed Designs	531
20.12	Summary	536
20.13	Exercises	536
	Bibliography	537
21	Advanced and Emerging Methods in Software Design (Online Chapter)	538
21.1	Designing in a Distributed Environment	
21.2	Introduction to Aspect-Oriented Programming	
21.3	Designing for Security with UMLsec	
21.4	Model-Driven Architectures	
21.5	The Formal Design Process in <i>B</i>	
21.6	Summary	
21.7	Exercises	
	Bibliography	
PART VI Implementation		
22	Principles of Implementation	539
22.1	Agile and Non-Agile Approaches to Implementation	540
22.2	Choosing a Programming Language	540
22.3	Identifying Classes	540
22.4	Defining Methods	541
22.5	Implementation Practices	544
22.6	Defensive Programming	548
22.7	Coding Standards	552
22.8	Comments	554
22.9	Tools and Environments for Programming	555
22.10	Case Study: Encounter Implementation	556
22.11	Case Study: Eclipse	559
22.12	Case Study: OpenOffice	559
22.13	Student Team Guidance for Implementation	565
22.14	Summary	566
22.15	Code Listings Referred to in this Chapter	566
22.16	Exercises	581
	Bibliography	583
23	Quality and Metrics in Implementation	584
23.1	Quality of Implementation	585
23.2	Code Inspections and Related Quality Procedures	597
23.3	Summary	599
23.4	Exercises	599

24 Refactoring	601
24.1 Big Refactorings	604
24.2 Composing Methods	606
24.3 Moving Features between Objects	608
24.4 Organizing Data	609
24.5 Generalization	612
24.6 Introducing Modules	616
24.7 Refactoring in Projects	617
24.8 Summary	619
24.9 Exercises	619
Bibliography	620
 PART VII Testing and Maintenance	
25 Introduction to Software Testing	621
25.1 Testing Early and Often, and the Agile Connection	622
25.2 Retesting: Regression Testing	622
25.3 Black Box and White Box Testing	623
25.4 Unit Testing vs. Post-Unit Testing	624
25.5 Testing Object-Oriented Implementations	625
25.6 Documenting Tests	626
25.7 Test Planning	626
25.8 Testing Test Suites by Fault Injection	628
25.9 Summary	628
25.10 Exercises	629
 26 Unit Testing	630
26.1 The Sources of Units for Unit Testing	631
26.2 Unit Test Methods	631
26.3 Testing Methods	642
26.4 Test-Driven Development	647
26.5 Case Study: Encounter Video Game	652
26.6 Summary	662
26.7 Exercises	663
Bibliography	665
 27 Module and Integration Testing	666
27.1 Stubs and Drivers	667
27.2 Testing a Class	668
27.3 Integration	672
27.4 Daily Builds	679
27.5 Interface Testing	680
27.6 Module Integration	682
27.7 Case Study: Class Test for Encounter	683
27.8 Case Study: Encounter Integration Plan	688
27.9 Summary	692
27.10 Exercises	692
Bibliography	693

28 Testing at the System Level	694
28.1 Functional Testing	696
28.2 Nonfunctional Testing	698
28.3 Testing with Lightweight Requirements	708
28.4 Testing Shortly Before Release	713
28.5 Case Study: Encounter Software Test Documentation	714
28.6 Case Study: Eclipse	723
28.7 Case Study: OpenOffice	726
28.8 Summary	728
28.9 Exercises	728
Bibliography	729
29 Software Maintenance	730
29.1 Types of Software Maintenance	731
29.2 Issues of Software Maintenance	734
29.3 Maintenance Process	736
29.4 IEEE Maintenance Standards	741
29.5 Software Evolution	749
29.6 Maintenance Metrics	751
29.7 Case Study	754
29.8 Summary	756
29.9 Exercises	757
Bibliography	758
Glossary	759
Index	767

Preface

Much of the modern world runs on software. As a result, software engineers are entrusted with significant responsibility. Although it is a biomedical engineer, for example, who designs health monitoring systems, it is a software engineer who creates its actual control functions. A marketing professional develops ways to reach customers online but it is a software engineer who makes the system a reality.

Today's software engineer must be able to participate in more than one kind of software process, work in agile teams, deal with customers, express requirements clearly, create modular designs, utilize legacy and open source projects, monitor quality, incorporate security, and apply many types of tests.

THE ISSUE OF SCALE

A software application consists of tens, hundreds, even thousands of classes. This is very different from managing three or four of them, and results in the dragon of complexity suggested by this book's cover. As also suggested there, however, this dragon can be subdued. Indeed, to deal with numerous and complex classes, software engineers have at their disposal a wide variety of tools and techniques. These range from the waterfall process to agile methodologies, from highly integrated tool suites to refactoring and loosely coupled tool sets. Underlying this variety is continuing research into reliable approaches, and an acknowledgment of the fact that one size does not fit all projects.

THIS EDITION COMPARED WITH THE FIRST

The first edition of this book emphasized the object-oriented approach, which has subsequently become widespread. It was also designed to help student teams carry out hands-on term projects through theory, examples, case studies, and practical steps. Object-orientation and hands-on continue to be major features of this edition. However, we have widened the scope of the first edition, especially by including extensive coverage of agile methods and refactoring, together with deeper coverage of quality and software design.

Readers of the first edition made extensive use of the complete video game case study—an example that they could follow “from soup to nuts” but which was significantly more comprehensive than a toy. This edition retains and updates that case study, but it adds the exploration of a simpler example on one hand (a DVD rental store) and large, real, open source case studies on the other. In particular, to provide students a feeling for the scope and complexity of real-world applications, this book leads them through selected requirements, design, implementation, and maintenance of the Eclipse and OpenOffice open source projects. The size, complexity, and transparency of these projects provide students a window into software engineering on a realistic scale.

Every book on software engineering faces a dilemma: how to reconcile the organization of the *topics* with the organization of actual software project *phases*. An organization of chapters into process/project management/requirements analysis/design/implementation/test/maintenance is straightforward but is liable to be misinterpreted as promoting the waterfall development process at the expense others. Our approach has been to use this organization in the seven parts of the book but to demonstrate throughout that each phase

typically belongs to a cycle rather than to a single waterfall sequence. In particular, our approach integrates agile methodologies consistently.

This edition also introduces somewhat advanced influential ideas, including model-driven architectures and aspect-oriented programming. Nowadays, formal methods are mandated by government agencies for the highest levels of security, and this book aims to educate readers in their possibilities. Due to print space limitations, some of this material is to be found in the online extension of this book.

In summary, specific features of this edition compared with the first are as follows:

- A sharpening and standardization of the material from the first edition
- A strong agile thread throughout, including a chapter on agility alone and one devoted to refactoring.
- A separate chapter on quality in six of the book's seven parts
- Real-world case studies, taken from the Eclipse and OpenOffice open source projects
- Greatly expanded coverage of software design and design patterns
- New chapters on advanced, influential software engineering ideas
- An organization of many of the book's seven parts as follows:
 - Principles
 - Details
 - Quality
 - Advanced Methods

HOW INSTRUCTORS CAN USE THIS BOOK

This book has been designed to accommodate multiple approaches to the learning and teaching of software engineering. Most instructors teach the fundamentals of software process, project management, requirements analysis, design, testing, implementation, and maintenance. Beyond this common ground, however, instructors employ a wide variety of styles and emphases. The following are major approaches, together with the sequence of chapters that support each of them.

- A. *Process emphasis*, concentrating on how applications are developed
All of Parts I through IV; and Chapters 15, 22, and 25 (the remaining principles and introduction chapters)
- B. *Design emphasis*, which teaches software engineering primarily as a design activity
Principles and introduction: Chapters 1, 3, 7, and 10; all of Part V; and Chapters 22 and 25 (principles and introduction)
- C. *Programming and agile emphasis*, which emphasizes software engineering as a code-oriented activity that satisfies requirements, emphasizing agile approaches
Principles and introduction: Chapters 1, 3, 7, 10, and 15; all of Part VI; and Chapters 25 and 26
- D. *Two-semester course*, which enables the instructor to cover most topics and assign a substantial hands-on project

D1. All of the chapters in the book, either in sequence from beginning to end
or

D2. In two passes as follows:

- (i) Principles and introduction chapters in the first semester: Chapters 1, 3, 7, 15, 22, and 25
- (ii) The remaining chapters in the second semester

- E. *Emphasis on a hands-on projects and case studies*, which relies mostly on an active team or individual project as the vehicle for learning theory and principles
Principles and introduction chapters: Chapters 1, 3, 7, 15, 22, 25, and 26, and all case study sections in the remaining chapters
- F. *Theory and principles emphasis*, concentrating on what one can learn about software engineering and its underpinnings
Principles and introduction chapters: Chapters 1, 2, 3, 7, 15, 22, and 25, followed, as time allows, by Chapters 14 and 21 (emerging topics)
- G. *Quality assurance and testing emphasis*
Principles and introduction: Chapters 1, 3, 7, and 10; Chapters 2, 5, 9, 13, 20, 23 (quality); and Chapters 25, 26, 27, and 28 (testing).

The web site for this book, including review questions and the Encounter game case study, is waveland.com/Extra_Material/32306/.

Eric Braude
Michael Bernstein
Boston, MA
January 2010

Acknowledgments

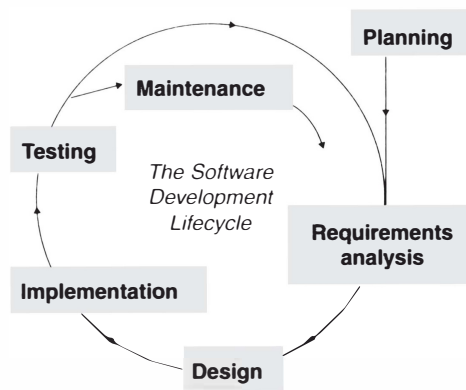
We owe a debt of gratitude to our students at Boston University's Metropolitan College. Working in myriad industries and businesses, they have given us invaluable feedback. The College itself has provided a model place for the teaching and learning of software engineering. Our thanks go to Dick Bostwick and Tom VanCourt, much of whose work in the first edition carries over to this one. We are grateful to the people who worked with us through the painstaking process of writing and publishing this book. We are particularly appreciative of the help from our editors, Dan Sayre and Jonathan Shipley; from Georgia King, Yee Lyn Song, and the indefatigable staff. We thank the reviewers of our manuscript, whose feedback has been invaluable:

Arvin Agah, University of Kansas
Steven C. Shaffer, Pennsylvania State University
Stephen M. Thebaut, University of Florida
Aravinda P. Sistla, University of Illinois, Chicago
James P. Purtilo, University of Maryland
Linda M. Ott, Michigan Technological University
Jianwei Niu, University of Texas, San Antonio
William Lively, Texas A&M University
Chung Lee, California State University, Pomona
Sudipto Ghosh, Colorado State University
Max I. Fomitchev, Pennsylvania State University
Lawrence Bernstein, Stevens Institute of Technology
John Dalbey, California Polytechnic University
Len Fisk, California State University, Chico
Ahmed M. Salem, California State University, Sacramento
Fred Strauss, New York University
Kai H. Chang, Auburn University
Andre van der Hoek, University of California, Irvine
Saeed Monemi, California Polytechnic University
Robert M. Cubert, University of Florida
Chris Tseng, San Jose State University
Michael James Payne, Purdue University
Carol A. Wellington, Shippensburg University
Yifei Dong, University of Oklahoma
Peter Blanchfield, Nottingham University
Desmond Greer, Queen's University Belfast
WeiQi Yan, Queen's University Belfast
Zaigham Mahmood, Derby University
Karel Pieterse, Hogeschool Van Amsterdam

This book would not have been possible without the constant love, patience, and encouragement of our families.

1

The Goals and Terminology of Software Engineering



- Why is software engineering important?
- Who and what does it consist of?
- What are its main activities?
- What are the principles of software engineering?
- What ethics are involved?
- What sorts of case studies will be used to illustrate the subject?

Figure 1.1 The context and learning goals for this chapter

The goal of software engineering, and the theme of this book, is the creation of software systems that meet the needs of customers and are reliable, efficient, and maintainable. In addition, the system should be produced in an economical fashion, meeting project schedules and budgets. This is no easy task, especially for large, complex applications. This chapter introduces the field of software engineering and explains how it addresses these goals. We first explain the term "software engineering," showing that it consists of many parts.

1.1 WHAT IS SOFTWARE ENGINEERING?

Software engineering is an engineering discipline that involves all aspects of developing and maintaining a software product. Engineering disciplines such as civil, mechanical, and electrical involve the design, analysis, and construction of an artifact for some practical purpose. Software engineering is no exception to this—software products certainly have practical purposes.

The IEEE defines Software Engineering [1] as follows:

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).

As this definition suggests, it's not only what is produced that's important but also *how* it is produced. Engineering disciplines employ an established set of *systematic*, *disciplined*, and *quantifiable* approaches to the development of artifacts. By thoroughly applying an analogous set of approaches to the development of software, we can expect the production of software that is highly reliable, is maintainable, and meets specified requirements. A disciplined approach is particularly important as the size of a software project grows. With increased size comes greatly increased complexity, and applying a systematic and disciplined approach is critical.

One of the first uses of the phrase "software engineering" was in 1968, by a NATO Study Group on Computer Science [2]. A conference was organized at that time, motivated by the "rapidly increasing importance of computer systems in many activities of society." The Study Group focused their attention on the problems with software, and held a working conference on Software Engineering that turned out to see far into the future. The following are some quotes from the conference that summarize the cause for their concern:

The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time . . . It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.

Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death.

Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it.

Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes—build the whole thing, push it off the cliff, let it crash, and start over again.

The Study Group discussed possible techniques and methods that might lead to solving these problems. They deliberately and provocatively used the term "software engineering," with an emphasis on engineering, as they wanted to "imply the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering." They believed that if these foundations and discipline were applied to building software systems, the quality of the resulting systems would be vastly improved.

Today, many of the issues they identified are addressed by evolving software engineering techniques and practices even as the scope of applications has increased dramatically. Throughout this book we examine these practices and explain how they contribute to producing high-quality software. Before doing that,

however, it is instructive to begin examining why software fails in the first place, and how some failures can even lead to catastrophic results.

1.2 WHY SOFTWARE ENGINEERING IS CRITICAL: SOFTWARE DISASTERS

Even with the best of intentions, a large number of software projects today are unsuccessful, with a large percentage never completed. Worse, quite a few software projects still end in disaster, causing a loss of money, time, and tragically, even lives. We review some representative samples here as cautionary tales. In all cases, the methods employed were inadequate for the complexity of the required application. Failures such as these motivate us to continually ask: How can we apply software engineering methodologies to ensure the appropriate level of quality in software applications?

1.2.1 The Virtual Case File Project

The FBI's Virtual Case File system was intended to automate the FBI's cumbersome paper-based case system, allow agents to share investigative information, and replace obsolete systems. Instead, after an expenditure of \$170 million, the result did not accomplish these objectives at all. The effect has been to inhibit the FBI from growing its crime-fighting mission despite the growth in terrorism and the increased sophistication of many criminal organizations. All of 700,000 lines of code, costing \$100 million, had to be abandoned. Poorly defined requirements, networking plans, and software development plans were cited by investigators as causes for this disaster.

1.2.2 The Ariane Project

"On 4 June 1996, the maiden flight of the *Ariane 5* launcher ended in failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded." [3] The cost of developing *Ariane* during the preceding decade has been estimated at \$7 billion. A significant fraction of this was wasted on June 4, 1996. *Ariane 5* itself, including its specific development, has been valued at \$500 million.

The source of the problem was described in the official report [3] as follows (italics added):

The internal Inertial Reference System software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating-point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error. . . . *The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose.*

In other words, the data conversion code itself was "correct" but was called upon to execute when it should not have been. The defect lay within controlling code. This kind of problem is easy to describe but not easy to avoid because many people are involved in large projects. Large projects become extraordinarily complex. Development efforts like *Ariane* call for extensive education and coordination within project management, quality assurance, configuration management, architecture, detailed design, programming, and testing organizations. Depending on how the project was organized and designed, any one of these organizations could have been partly responsible for seeing to it that the code in question was not called after liftoff.

1.2.3 Radiation Overdose

As software controls an ever-increasing number of devices, its reliability is coming under increasingly intense scrutiny. In the project management magazine *Baseline*, Debbie Gage, John McCormick, and Berta Ramona wrote

of a lawsuit alleging "massive overdoses of gamma rays partly due to limitations of the computer program that guided use of" a particular radiation-therapy machine. They reported the following: "The International Atomic Energy Agency said in May 2001 that at least five of the deaths were probably from radiation poisoning (from the machine) and at least 15 more patients risked developing 'serious complications' from radiation." [4] The defect did not show up until a significant time after release, and only after certain sequences of operator actions.

The following describes the software defect, and is quoted from [5].

Setting the bending magnets takes about 8 seconds. *Magnet* calls a subroutine called *Ptime* to introduce a time delay. Since several magnets need to be set, *Ptime* is entered and exited several times. A flag to indicate that bending magnets are being set is initialized upon entry to the *Magnet* subroutine and cleared at the end of *Ptime*. Furthermore, *Ptime* checks a shared variable, set by the keyboard handler, that indicates the presence of any editing requests. If there are edits, then *Ptime* clears the bending magnet variable and exits to *Magnet*, which then exits to *Datent*. But the edit change variable is checked by *Ptime* only if the bending magnet flag is set. Since *Ptime* clears it during its first execution, any edits performed during each succeeding pass through *Ptime* will not be recognized. Thus, an edit change of the mode or energy, although reflected on the operator's screen and the mode/energy offset variable, will not be sensed by *Datent* so it can index the appropriate calibration tables for the machine parameters.¹

This is a fairly involved explanation but not at all beyond the complexity of many software systems in existence today. When should this type of error have been found? If sound software engineering discipline had been employed during all phases of the project, there would have been several opportunities in the development process to detect it.

1.2.4 More Software Disasters

Readers who wish to know about more software disasters, big and small, are referred to Neumann [6], who discusses risks, problems, defects, and disasters relating to reliability, safety, security vulnerabilities, integrity, and threats to privacy and well-being. Another source is the ACM publication *Software Engineering Notes* and its Risks Forum [7].

1.3 WHY SOFTWARE FAILS OR SUCCEEDS

Thankfully, not all software projects end in the types of disasters described above, but far too many end in failure. What does it mean for a software project to be unsuccessful? Simply put, an unsuccessful project is one that fails to meet expectations. More specifically, the undesirable outcomes may include the following:

- Over budget
- Exceeds schedule and/or misses market window
- Doesn't meet stated customer requirements
- Lower quality than expected
- Performance doesn't meet expectations
- Too difficult to use

¹ Levenson, Nancy, and Turner C.S., "An Investigation of the Therac-25 Accidents," IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18–41, copyright © 1993 IEEE.

Failing to meet just one of these objectives can cause a project to be deemed unsuccessful. For example, if a project is completed under budget, meets all requirements and functionality, has high quality, good performance and is easy to use, it still may not be successful if the schedule was missed and no customers are willing to purchase it as a result.

Charette [8] notes that there are many underlying reasons software projects are unsuccessful, including:

- Unrealistic or unarticulated project goals
- Poor project management
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Inability to handle the project's complexity

Other contributing factors are:

- Poor software design methodology
- Wrong or inefficient set of development tools
- Poor testing methodology
- Inadequate test coverage
- Inappropriate (or lack of) software process²

Unsuccessful software projects usually fall victim to several of these. To reiterate, the goal of software engineering, and the theme of this book, is the creation of software systems that are reliable, efficient, maintainable, and meet the needs of customers. Software engineering provides the tools and methodologies necessary to accomplish these goals, resulting in the development of successful software systems.

We'll end this section on a positive note. The authors feel that software engineering has improved greatly, when measured fairly. Projects of equal ambition can typically get done far more successfully now than 10 years ago. The issue really is that the ambition and scope of applications have grown enormously. The Eclipse software development platform, which this book uses as a case study, is an excellent example of a successful application. This is largely due to its flexible design, inclusive requirements process, and thorough testing.

1.4 SOFTWARE ENGINEERING ACTIVITIES

The production of software systems can be extremely complex and present many challenges. Systems, especially large ones, require the coordination of many *people*, called stakeholders, who must be organized into teams and whose primary objective is to build a *product* that meets defined requirements. The entire effort must be organized

² Charette, Robert, "Why Software Fails," IEEE Spectrum, Vol. 42, No. 9, September 2005, pp. 42–49, copyright © 2005 IEEE.

-
- **People**
 - Project stakeholders.
 - **Product**
 - The software product plus associated documents.
 - **Project**
 - The activities carried out to produce the product.
 - **Process**
 - Framework within which the team carries out the activities necessary to build the product.
-

Figure 1.2 The four “P’s” that constitute software engineering

into a cohesive *project*, with a solid plan for success. Finally, to successfully develop the product, the activities of the people must be organized through use of an orderly and well-defined *process*. Collectively, these activities are known as the 4 P’s of software engineering: people, product, project, and process. Successful software projects must adequately plan for and address all of them. Sometimes, the needs of each of the P’s conflict with each other, and a proper balance must be achieved for a project to be successful. Concentrating on one P without the others can lead to a project’s failure. For example, if people are organized into efficient teams and given the resources they need to perform their roles, a project can still be unsuccessful if there’s no defined software process to follow, as chaos can ensue. The 4 P’s are summarized in Figure 1.2 and are discussed in the sections that follow.

1.4.1 People

People are the most important resource on a software project. It is through their efforts that software is successfully constructed and delivered. Competent people must be recruited, trained, motivated, and provided with a growth path, which is no easy task. They are the lifeblood of any successful project. Software development is often dictated by tight, market-driven deadlines and demanding lists of required product features. Because of this, only well-organized groups of engineers, educated and experienced in the methods of software engineering, are capable of consistently carrying out these activities to everyone’s satisfaction. The alternative is often chaos and, all too frequently, disaster.

Typically, several groups of people are involved with and have a stake in a project’s outcome. These are called its *stakeholders*. They include business management, project management, the development team, customers, and end users. Although each group is motivated to see the project succeed, given their diverse roles each has a different perspective on the process. This is discussed next, for each of the groups cited.

Business Management

These are people responsible for the business side of the company developing the software. They include senior management (e.g., V.P. Finance), marketing (e.g., Product Manager), and development managers. Their primary focus is on business issues including profit, cost effectiveness, market competitiveness, and customer satisfaction. They are typically not particularly knowledgeable about or involved in the technical aspects of the project.

Project Management

Project managers are responsible for planning and tracking a project. They are involved throughout, managing the people, process, and activities. They continuously monitor progress and proactively implement necessary changes and improvements to keep the project on schedule and within budget.

Development Team

Software engineers are responsible for developing and maintaining the software. Software development includes many tasks such as requirements gathering, software architecture and design, implementation, testing, configuration management, and documentation. This book will have much to say about each of these topics. Software engineers are motivated by many factors including technical innovation, low overhead (e.g., a minimum of business-type meetings), and having the time and support to stay involved in technology.

Customers

Customers are responsible for purchasing the software. They may or may not actually use the software. Customers may be purchasing it for use by others in their organization. They are primarily interested in software that is cost-effective, meets specific business needs, and is of high quality. They are typically involved in some aspect of specifying requirements, and since they are paying for the project, they have the ultimate say in defining the requirements.

End Users

End users are people who interact with and use software after it is finished being developed. End users are motivated by software that's easy to use and helps them perform their jobs as efficiently as possible. For example, once they become accustomed to and are effective using a particular user interface, they are typically reluctant to accept major changes to it.

1.4.2 Product

The products of a software development effort consist of much more than the source and object code. They also include project documentation (e.g., requirements document, design specification), test plans and results, customer documentation (e.g., installation guide, command reference), and productivity measurements. These products are often called *artifacts*, and are summarized in Figure 1.3. This book describes the complete set of artifacts.

Part III, on software management, describes project metrics and how they are collected and used to measure productivity.

-
- **Project documentation**
Documents produced during software definition and development.
 - **Code**
Source and object.
 - **Test documents**
Plans, cases, and results.
 - **Customer documents**
Documents explaining how to use and operate product.
 - **Productivity measurements**
Analyze project productivity.
-

Figure 1.3 The main product artifacts of a software project

Part IV, on requirements analysis, explains how to produce requirements that specify what the product is intended to be.

Part V explains how to specify software designs. Chapter 20 describes software architectures. Chapter 21 describes how to specify the detailed designs. Design patterns, a standard means of communicating intelligently with each other about design, are described in Chapter 19.

Part VI discusses implementation (programming), emphasizing standards and precision. A major goal is to help developers to write programs that are much easier to verify for correctness.

Part VII describes how to test the parts of an application, as well as the whole. It includes test procedures that specify how tests are conducted and the test cases that specify the input data for tests. Part VII also describes the types of customer documentation artifacts that are produced and their purpose.

1.4.3 Project

A software project defines the activities and associated results needed to produce a software product. Every project involves a similar set of activities: planning, determining what's required, determining how the software should be built to meet the requirements, implementing the software, testing the software, and maintaining it once delivered to customers. These major project activities are summarized in Figure 1.4.

In addition to these activities, various development paradigms, techniques, and tools exist and are employed on different projects. A development paradigm is a way of thinking about the process of producing software.

An example of a development paradigm, and one that is in wide use today, is the *object-oriented paradigm*. It was invented to make designs and code match the real world. That is, an object as represented in a software design is patterned after a real-world object. For example, suppose that a banking application is to be built that includes support for customers, bank accounts, and transactions on the accounts. In an object-oriented paradigm, these real-world concepts are represented in the design and implementation by corresponding

-
- **Planning**
 - Plan, monitor, and control the software project.
 - **Requirements analysis**
 - Define what to build.
 - **Design**
 - Describe how to build the software.
 - **Implementation**
 - Program the software.
 - **Testing**
 - Validate that software meets the requirements.
 - **Maintenance**
 - Resolve problems; adapt software to meet new requirements.
-

Figure 1.4 Major activities of a software project

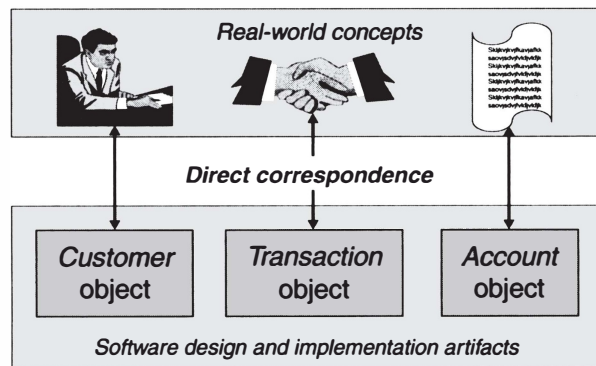


Figure 1.5 A key role of the object-oriented paradigm

Source: Graphics reproduced with permission from Corel.

classes. This greatly facilitates identifying and applying modifications to a design necessitated by changes to real-world requirements. For example, if the steps executed during a particular *transaction* need to change, the design can more easily accommodate this since there's a corresponding *transaction* object in the design. The design representation for transactions is encapsulated within the transaction object, and modifications can be applied more easily. This is illustrated in Figure 1.5. In non-object-oriented languages, the representation of a real-world concept such as a customer may be spread across many disconnected pieces of source code.

1.4.4 Process

A *software process* is a framework for carrying out the activities of a project in an organized and disciplined manner. It imposes structure and helps guide the many people and activities in a coherent manner. A software project progresses through different *phases*, each interrelated and bounded by time. A software process expresses the interrelationship among the phases by defining their order and frequency, as well as defining the deliverables of the project. Figure 1 names the major phases and indicates the order in which they are usually performed.

Specific software process implementations are called *software process models*. There are several such models, but most are based on either the *waterfall* or *iterative development* models. Each of these is briefly described below. Part II covers the evolution of software processes and details these plus several other of the most important process models.

The *waterfall process* is the simplest software process model, and forms the basis for most others. A pure waterfall process dictates that phases are implemented in sequence, with no phase starting before the previous one has almost completed. That is, phases are executed in a strictly sequential order, usually with small overlaps. Once a waterfall phase is finished it's deemed complete for the project and there is no need to return to it. Variations of waterfall exist where already completed phases may be revisited and minor updates applied, as a result of work done on subsequent phases. Waterfall begins with an inception phase, where the product is conceived and business objectives defined. Next is the specification of the requirements, followed by the design phase, the implementation phase, the testing phase, and finally the maintenance phase. Figure 1.6 illustrates the main phases and their sequence. This means that the process goes around the circle of Figure 1.1 just once.

Software development rarely occurs in the strict waterfall sequence. Instead, it skips back and forth somewhat among requirements, design, implementation, and testing. In practice, then, we often use *iterative*