ESSENTIALS OF

software engineering

FOURTH EDITION

Frank Tsui
Orlando Karam
Barbara Bernal

ESSENTIALS OF

software engineering

FOURTH EDITION

Frank Tsui
Orlando Karam
Barbara Bernal

All associated with Kennesaw State University

JONES & BARTLETT
LEARNING

World Headquarters
Jones & Bartlett Learning
5 Wall Street
Burlington, MA 01803
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

13278-6

Production Credits
VP, Executive Publisher: David D. Cella
Executive Editor: Matt Kane
Acquisitions Editor: Laura Pagluica
Editorial Assistant: Taylor Ferracane
Editorial Assistant: Mary Menzemer
Director of Vendor Management: Amy Rose
Director of Marketing: Andrea DeFronzo
Marketing Manager: Amy Langlais
VP, Manufacturing and Inventory Control: Therese Connell
Project Management and Composition: S4Carlisle Publishing Services
Cover Design: Michael O'Donnell
Rights & Media Specialist: Merideth Tumasz
Media Development Editor: Shannon Sheehan
Cover Image: © Eladora/Dreamstime.com
Printing and Binding: Edwards Brothers Malloy
Cover Printing: Edwards Brothers Malloy

Library of Congress Cataloging-in-Publication Data
Names: Tsui, Frank F., author. | Karam, Orlando, author. | Bernal, Barbara, author.
Title: Essentials of software engineering / Frank Tsui, Orlando Karam, Barbara Bernal.
Description: Fourth edition. | Burlington, Massachusetts : Jones & Bartlett Learning, [2017] | Includes bibliographical references and index.
Identifiers: LCCN 2016036199 | ISBN 9781284106008
Subjects: LCSH: Software engineering.
Classification: LCC QA76.758 .T78 2014 | DDC 005.1--dc23 LC record available at https://lccn.loc.gov/2016036199

6048
Printed in the United States of America
20 19 18 17 16 10 9 8 7 6 5 4 3 2 1

Contents

Preface xiii

Chapter 1 Creating a Program 1

1.1 A Simple Problem 2
1.1.1 Decisions, Decisions 2
1.1.2 Functional Requirements 3
1.1.3 Nonfunctional Requirements 4
1.1.4 Design Constraints 5
1.1.5 Design Decisions 6
1.2 Testing 6
1.3 Estimating Effort 7
1.4 Implementations 9
1.4.1 A Few Pointers on Implementation 9
1.4.2 Basic Design 10
1.4.3 Unit Testing with `JUnit` 10
1.4.4 Implementation of `StringSorter` 11
1.4.5 User Interfaces 16
1.5 Summary 20
1.6 Review Questions 20

1.7 Exercises 20
1.8 References and Suggested Readings 21

Chapter 2 Building a System 23
2.1 Characteristics of Building a System 24
2.1.1 Size and Complexity 24
2.1.2 Technical Considerations of Development and Support 25
2.1.3 Nontechnical Considerations of Development and Support 29
2.2 Building a Hypothetical System 30
2.2.1 Requirements of the Payroll System 30
2.2.2 Designing the Payroll System 32
2.2.3 Code and Unit Testing the Payroll System 34
2.2.4 Integration and Functionally Testing the Payroll System 35
2.2.5 Release of the Payroll System 35
2.2.6 Support and Maintenance 36
2.3 Coordination Efforts 37
2.3.1 Process 37
2.3.2 Product 38
2.3.3 People 38
2.4 Summary 38
2.5 Review Questions 39
2.6 Exercises 39
2.7 References and Suggested Readings 39

Chapter 3 Engineering of Software 41
3.1 Examples and Characteristics of Software Failures 42
3.1.1 Project Failures 42
3.1.2 Software Product Failures 43
3.1.3 Coordination and Other Concerns 44
3.2 Software Engineering 45
3.2.1 What Is Software Engineering? 45
3.2.2 Definitions of Software Engineering 45
3.2.3 Relevancy of Software Engineering and Software 46
3.3 Software Engineering Profession and Ethics 47

3.3.1 Software Engineering Code of Ethics 47
3.3.2 Professional Behavior 48
3.4 Principles of Software Engineering 49
3.4.1 Davis's Early Principles of Software Engineering 50
3.4.2 Royce's More Modern Principles 51
3.4.3 Wasserman's Fundamental Software Engineering Concepts 52
3.5 Summary 53
3.6 Review Questions 54
3.7 Exercises 54
3.8 References and Suggested Readings 54

Chapter 4 Software Process Models 57
4.1 Software Processes 58
4.1.1 Goal of Software Process Models 58
4.1.2 The "Simplest" Process Model 58
4.2 Traditional Process Models 59
4.2.1 Waterfall Model 59
4.2.2 Chief Programmer Team Approach 61
4.2.3 Incremental Model 61
4.2.4 Spiral Model 63
4.3 A More Modern Process 65
4.3.1 General Foundations of Rational Unified Process Framework 65
4.3.2 The Phases of RUP 66
4.4 Entry and Exit Criteria 68
4.4.1 Entry Criteria 69
4.4.2 Exit Criteria 69
4.5 Process Assessment Models 70
4.5.1 SEI's Capability Maturity Model 70
4.5.2 SEI's Capability Maturity Model Integrated 72
4.6 Process Definition and Communication 78
4.7 Summary 79
4.8 Review Questions 80
4.9 Exercises 80
4.10 References and Suggested Readings 81

Chapter 5 New and Emerging Process Methodologies 83
5.1 What Are Agile Processes? 84
5.2 Why Agile Processes? 85
5.3 Some Process Methodologies 86
5.3.1 Extreme Programming (XP) 86
5.3.2 The Crystal Family of Methodologies 90
5.3.3 The Unified Process as Agile 94
5.3.4 Scrum 94
5.3.5 Kanban Method: A New Addition to Agile 96
5.3.6 Open Source Software Development 97
5.3.7 Summary of Processes 98
5.4 Choosing a Process 100
5.4.1 Projects and Environments Better Suited for Each Kind of Process 100
5.4.2 Main Risks and Disadvantages of Agile Processes 101
5.4.3 Main Advantages of Agile Processes 101
5.5 Summary 102
5.6 Review Questions 102
5.7 Exercises 102
5.8 References and Suggested Readings 103

Chapter 6 Requirements Engineering 105
6.1 Requirements Processing 106
6.1.1 Preparing for Requirements Processing 106
6.1.2 Requirements Engineering Process 107
6.2 Requirements Elicitation and Gathering 109
6.2.1 Eliciting High-Level Requirements 110
6.2.2 Eliciting Detailed Requirements 112
6.3 Requirements Analysis 114
6.3.1 Requirements Analysis and Clustering by Business Flow 114
6.3.2 Requirements Analysis and Clustering with Object-Oriented Use Cases 116
6.3.3 Requirements Analysis and Clustering by Viewpoint-Oriented Requirements Definition 118
6.3.4 Requirements Analysis and Prioritization 119
6.3.5 Requirements Traceability 121

6.4 Requirements Definition, Prototyping, and Reviews 122
6.5 Requirements Specification and Requirements Agreement 126
6.6 Summary 127
6.7 Review Questions 127
6.8 Exercises 128
6.9 References and Suggested Readings 129

Chapter 7 Design: Architecture and Methodology 131
7.1 Introduction to Design 132
7.2 Architectural Design 133
7.2.1 What Is Software Architecture? 133
7.2.2 Views and Viewpoints 133
7.2.3 Meta-Architectural Knowledge: Styles, Patterns, Tactics, and Reference Architectures 135
7.2.4 A Network-based Web Reference Architecture—REST 141
7.3 Detailed Design 142
7.3.1 Functional Decomposition 143
7.3.2 Relational Database Design 144
7.3.3 Designing for Big Data 149
7.3.4 Object-Oriented Design and UML 151
7.3.5 User-Interface Design 155
7.3.6 Some Further Design Concerns 162
7.4 HTML-Script-SQL Design Example 162
7.5 Summary 165
7.6 Review Questions 165
7.7 Exercises 166
7.8 References and Suggested Readings 166

Chapter 8 Design Characteristics and Metrics 169
8.1 Characterizing Design 170
8.2 Some Legacy Characterizations of Design Attributes 170
8.2.1 Halstead Complexity Metric 170
8.2.2 McCabe's Cyclomatic Complexity 171
8.2.3 Henry-Kafura Information Flow 173
8.2.4 A Higher-Level Complexity Measure 174

8.3 "Good" Design Attributes 175
8.3.1 Cohesion 175
8.3.2 Coupling 178
8.4 OO Design Metrics 181
8.4.1 Aspect-Oriented Programming 182
8.4.2 The Law of Demeter 183
8.5 User-Interface Design 184
8.5.1 Good UI Characteristics 184
8.5.2 Usability Evaluation and Testing 185
8.6 Summary 186
8.7 Review Questions 186
8.8 Exercises 187
8.9 References and Suggested Readings 188

Chapter 9 Implementation 191
9.1 Introduction to Implementation 192
9.2 Characteristics of a Good Implementation 192
9.2.1 Programming Style and Coding Guidelines 193
9.2.2 Comments 196
9.3 Implementation Practices 197
9.3.1 Debugging 197
9.3.2 Assertions and Defensive Programming 199
9.3.3 Performance Optimization 199
9.3.4 Refactoring 200
9.3.5 Code Reuse 201
9.4 Developing for the Cloud 202
9.4.1 Infrastructure as a Service 202
9.4.2 Platform as a Service 203
9.4.3 Cloud Application Services 203
9.4.4 Cloud Services for Developers 204
9.4.5 Advantages and Disadvantages of the Cloud 205
9.5 Summary 205
9.6 Review Questions 206
9.7 Exercises 206
9.8 References and Suggested Readings 206

Chapter 10 Testing and Quality Assurance 209
10.1 Introduction to Testing and Quality Assurance 210
10.2 Testing 212
10.2.1 The Purposes of Testing 212
10.3 Testing Techniques 213
10.3.1 Equivalence-Class Partitioning 215
10.3.2 Boundary Value Analysis 217
10.3.3 Path Analysis 218
10.3.4 Combinations of Conditions 222
10.3.5 Automated Unit Testing and Test-Driven Development 223
10.3.6 An Example of Test-Driven Development 224
10.4 When to Stop Testing 228
10.5 Inspections and Reviews 229
10.6 Formal Methods 231
10.7 Static Analysis 232
10.8 Summary 233
10.9 Review Questions 234
10.10 Exercises 235
10.11 References and Suggested Readings 235

Chapter 11 Configuration Management, Integration, and Builds 237
11.1 Software Configuration Management 238
11.2 Policy, Process, and Artifacts 238
11.2.1 Business Policy Impact on Configuration Management 241
11.2.2 Process Influence on Configuration Management 241
11.3 Configuration Management Framework 243
11.3.1 Naming Model 243
11.3.2 Storage and Access Model 245
11.4 Build and Integration and Build 247
11.5 Tools for Configuration Management 248
11.6 Managing the Configuration Management Framework 250

11.7 Summary 251
11.8 Review Questions 252
11.9 Exercises 252
11.10 References and Suggested Readings 253

Chapter 12 Software Support and Maintenance 255
12.1 Customer Support 256
12.1.1 User Problem Arrival Rate 256
12.1.2 Customer Interface and Call Management 258
12.1.3 Technical Problem/Fix 260
12.1.4 Fix Delivery and Fix Installs 262
12.2 Product Maintenance Updates and Release Cycles 263
12.3 Change Control 265
12.4 Summary 267
12.5 Review Questions 267
12.6 Exercises 267
12.7 References and Suggested Readings 268

Chapter 13 Software Project Management 269
13.1 Project Management 270
13.1.1 The Need for Project Management 270
13.1.2 The Project Management Process 270
13.1.3 The Planning Phase of Project Management 271
13.1.4 The Organizing Phase of Project Management 274
13.1.5 The Monitoring Phase of Project Management 275
13.1.6 The Adjusting Phase of Project Management 277
13.2 Some Project Management Techniques 279
13.2.1 Project Effort Estimation 279
13.2.2 Work Breakdown Structure 286
13.2.3 Project Status Tracking with Earned Value 289
13.2.4 Measuring Project Properties and GQM 291
13.3 Summary 293
13.4 Review Questions 294
13.5 Exercises 294
13.6 References and Suggested Readings 296

Chapter 14 Epilogue and Some Contemporary Issues 299
14.1 Security and Software Engineering 301
14.2 Reverse Engineering and Software Obfuscation 301
14.3 Software Validation and Verification Methodologies and Tools 302
14.4 References and Suggested Readings 304

Appendix A 307
Essential Software Development Plan (SDP) 307

Appendix B 309
Essential Software Requirements Specifications (SRS) 309
Example 1: Essential SRS—Descriptive 309
Example 2: Essential SRS—Object Oriented 311
Example 3: Essential SRS—IEEE Standard 312
Example 4: Essential SRS—Narrative Approach 313

Appendix C 315
Essential Software Design 315
Example 1: Essential Software Design—UML 315
Example 2: Essential Software Design—Structural 316

Appendix D 319
Essential Test Plan 319

Glossary 321

Index 325

Preface

Essentials of Software Engineering was born from our experiences in teaching introductory material on software engineering. Although there are many books on this topic available in the market, few serve the purpose of introducing only the core material for a 1-semester course that meets approximately 3 hours a week for 16 weeks. With the proliferation of small web applications, many new information technology personnel have entered the field of software engineering without fully understanding what it entails. This book is intended to serve both new students with limited experience as well as experienced information technology professionals who are contemplating a new career in the software engineering discipline. The complete life cycle of a software system is covered in this book, from inception to release and through support.

The content of this book has also been shaped by our personal experiences and backgrounds—one author has more than 25 years in building, supporting, and managing large and complex mission-critical software with companies such as IBM, Blue Cross Blue Shield, MARCAM, and RCA; another author has experience involving extensive expertise in constructing smaller software with Agile methods at companies such as Microsoft and Amazon; and the third author is bilingual and has broad software engineering teaching experiences with both U.S. college students and non-U.S. Spanish-speaking students.

Although new ideas and technology will continue to emerge and some of the principles introduced in this book may have to be updated, we believe that the underlying and fundamental concepts we present here will remain.

Preface to the Fourth Edition

Since the publication of the third edition, the computing industry has moved faster toward service applications and social media. While the software engineering fundamentals have stayed relatively stable, we decided to make a few modifications to reflect some of the movements in software engineering, including the enhancements of teaching aids to this book. It is our goal to continue keeping the content of the book concise enough to be taught in a 16-week, 1-semester course.

The following is a list of major enhancements made for the fourth edition.

- Discussion on kanban methodology in Chapter 5
- REST distributed processing architecture in Chapter 7
- Data design, analysis and "big data" in Chapter 7
- Code reuse in Chapter 9
- Cloud computing in Chapter 9
- Sample team projects available online for students and instructors
- Updated and enhanced instructor resources

In addition, we have made small modifications to some sentences throughout the book to improve the expression, emphasis, and comprehension. We have also received input from those who used our first, second, and third editions of the book from different universities and have corrected the grammatical and spelling errors. Any remaining error is totally ours.

The first, second, and third editions of this book have been used by numerous colleges and universities, and we thank them for their patience and input. We have learned a lot in the process. We hope the fourth edition will prove to be a better one for all future readers.

Organization of the Book

Chapters 1 and 2 demonstrate the difference between a small programming project and the effort required to construct a mission-critical software system. We purposely took two chapters to demonstrate this concept, highlighting the difference between a single-person "garage" operation and a team project required to construct a large "professional" system. The discussion in these two chapters delineates the rationale for studying and understanding software engineering. Chapter 3 is the first place where software engineering is discussed more formally. Included in this chapter is an introduction to the profession of software engineering and its code of ethics.

The traditional topics of software processes, process models, and methodologies are covered in Chapters 4 and 5. Reflecting the vast amount of progress made in this area, these chapters explain in extensive detail how to evaluate the processes through the Capability Maturity Models from the Software Engineering Institute (SEI).

Chapters 6, 7, 9, 10, and 11 cover the sequence of development activities from requirements through product release at a macro level. Chapter 7 includes an expanded UI design discussion with an example of HTML-Script-SQL design and implementation. Chapter 8,

following the chapter on software design, steps back and discusses design characteristics and metrics utilized in evaluating high-level and detail designs. Chapter 11 discusses not only product release, but the general concept of configuration management.

Chapter 12 explores the support and maintenance activities related to a software system after it is released to customers and users. Topics covered include call management, problem fixes, and feature releases. The need for configuration management is further emphasized in this chapter. Chapter 13 summarizes the phases of project management, along with some specific project planning and monitoring techniques. It is only a summary, and some topics, such as team building and leadership qualities, are not included. The software project management process is contrasted from the development and support processes. Chapter 14 concludes the book and provides a view of the current issues within software engineering and the future topics in our field.

The appendices give readers and students insight into possible results from major activities in software development with the "essential samples" for a Team Plan, Software Development Plan, Requirements Specification, Design Plan, and Test Plan. An often asked question is what a requirements document or a test plan should look like. To help answer this question and provide a starting point, we have included sample formats of possible documents resulting from the four activities of Planning, Requirements, Design, and Test Plan. These are provided as follows:

- Appendix A: Essential Software Development Plan (SDP)
- Appendix B: Essential Software Requirements Specifications (SRS)
  - Example 1: Essential SRS—Descriptive
  - Example 2: Essential SRS—Object Oriented
  - Example 3: Essential SRS—IEEE Standard
  - Example 4: Essential SRS— Narrative Approach
- Appendix C: Essential Software Design
  - Example 1: Essential Software Design—UML
  - Example 2: Essential Software Design—Structural
- Appendix D: Essential Test Plan

Many times in the development of team projects by novice software engineers there is a need for specific direction on how to document the process. The four appendices were developed to give the reader concrete examples of the possible essential outlines. Each of the appendices gives an outline with explanations. This provides the instructor with concrete material to supplement class activities, team project assignments, and/or independent work.

The topical coverage in this book reflects those emphasized by the IEEE Computer Society–sponsored *Software Engineering Body of Knowledge* (SWEBOK) and by the *Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Program in Software Engineering*. The one topic that is not highlighted but is discussed throughout the book concerns quality—a topic that needs to be addressed and integrated into all activities. It is not just a concern of the testers. Quality is discussed in multiple chapters to reflect its broad implications and cross activities.

Suggested Teaching Plan

All the chapters in this book can be covered within 1 semester. However, some instructors may prefer a different emphasis:

- Those who want to focus on direct development activities should spend more time on Chapters 6 through 11.
- Those who want to focus more on indirect and general activities should spend more time on Chapters 1, 12, and 13.

It should be pointed out that both the direct development and the indirect support activities are important. The combined set forms the software engineering discipline.

There are two sets of questions at the end of each chapter. For the Review Questions, students can find answers directly in the chapter. The Exercises are meant to be used for potential class discussion, homework, or small projects.

Supplements

Slides in PowerPoint format, Answers to End-of-Chapter Exercises, Sourcecode, and sample Test Questions are available for free instructor download. To request access, please visit go.jblearning.com/Tsui4e or contact your account representative.

Acknowledgments

We would first like to thank our families, especially our wives, Lina Colli and Teresa Tsui. They provided constant encouragement and understanding when we spent more time with the manuscript than with them. Our children—Colleen and Nicholas; Orlando and Michelle; and Victoria, Liz, and Alex—enthusiastically supported our efforts as well.

In addition, we would like to thank the reviewers who have improved the book in many ways. We would like to specifically thank the following individuals for work on our third edition:

- Brent Auernheimer, California State University, Fresno
- Ayad Boudiab, Georgia Perimeter College
- Kai Chang, Auburn University
- David Gustafson, Kansas State University
- Theresa Jefferson, George Washington University
- Dar-Biau Liu, California State University, Long Beach
- Bruce Logan, Lesley University
- Jeanna Matthews, Clarkson University
- Michael Oudshoorn, Montana State University
- Frank Ackerman, Montana Tech
- Mark Hall, Hastings College
- Dr. Dimitris Papamichail, The College of New Jersey
- Dr. Jody Paul, Metro State Denver
- Dr. David A. Cook, Stephen F. Austin State University
- Dr. Reza Eftekari, George Washington University, University of Maryland at College Park
- Dr. Joe Hoffert, Indiana Wesleyan University

- Dr. Sofya Poger, Felician College
- Dr. Stephen Hughes, Coe College
- Ian Cottingham, Jeffrey S. Raikes School at The University of Nebraska, Lincoln
- Dr. John Dalbey, California Polytechnic State University
- Dr. Michael Murphy, Concordia University Texas
- Dr. Edward G. Nava, University of New Mexico
- Dr. Yenumula B. Reddy, Grambling State University
- Alan C. Verbit, Delaware County Community College
- Dr. David Burris, Sam Houston State University

We would also like to thank the following individuals for their work on our fourth edition:

- Savador Almanza-Garcia, Vector CANtech, Inc.
- Dr. Ronand Finkbine, Indiana University Southeast
- Dr. Christopher Fox, James Madison University
- Paul G. Garland, Johns Hopkins University
- Dr. Emily Navarro, University of California, Irvine
- Benjamin Sweet, Lawrence Technological University
- Ben Geisler, University of Wisconsin, Green Bay

We continue to appreciate the help from Taylor Ferracane, Laura Pagluica, Bharathi Sanjeev, Amy Rose, Mary Menzemer, and others at Jones & Bartlett Learning. Any remaining error is solely the mistake of the authors.

—Frank Tsui
—Orlando Karam
—Barbara Bernal

CHAPTER 1

Creating a Program

OBJECTIVES

- Analyze some of the issues involved in producing a simple program:
  - Requirements (functional, nonfunctional)
  - Design constraints and design decisions
  - Testing
  - Effort estimation
  - Implementation details
- Understand the activities involved in writing even a simple program.
- Preview many additional software engineering topics found in the later chapters.

1.1 A Simple Problem

In this chapter we will analyze the tasks involved in writing a relatively simple program. This will serve as a contrast to what is involved in developing a large system, which is described in Chapter 2.

Assume that you have been given the following simple problem: "Given a collection of lines of text (strings) stored in a file, sort them in alphabetical order, and write them to another file." This is probably one of the simplest problems you will be involved with. You have probably done similar assignments for some of your introduction to programming classes.

1.1.1 Decisions, Decisions

A problem statement such as the one mentioned in the above simple problem does not completely specify the problem. You need to clarify the requirements in order to produce a program that better satisfies the real problem. You need to understand all the **program requirements** and the **design constraints** imposed by the client on the design, and you need to make important technical decisions. A complete problem statement would include the requirements, which state and qualify what the program does, and the design constraints, which depict the ways in which you can design and implement it.

Program requirements Statements that define and qualify what the program needs to do.

Design constraints Statements that constrain the ways in which the software can be designed and implemented.

The most important thing to realize is that the word *requirements* is not used as it is in colloquial English. In many business transactions, a requirement is something that absolutely must happen. However, in software engineering many items are negotiable. Given that every requirement will have a cost, the clients may decide that they do not really need it after they understand the related cost. Requirements are often grouped into those that are "needed" and those that are "nice to have."

It is also useful to distinguish between **functional requirements**—what the program does—and **nonfunctional requirements**—the manner in which the program must behave. In a way, a function is similar to that of a direct and indirect object in grammar. Thus the functional requirements for our problem will describe what it does: sort a file (with all the detail required); the nonfunctional requirements will describe items such as performance, usability, and maintainability. Functional requirements tend to have a Boolean measurement where the requirement is either satisfied or not satisfied, but nonfunctional requirements tend to apply to things measured on a linear scale where the measurements can vary much more. Performance and maintainability requirements, as examples, may be measured in degrees of satisfaction.

Functional requirements What a program needs to do.

Nonfunctional requirements The manner in which the functional requirements need to be achieved.

Nonfunctional requirements are informally referred as the "ilities," because the words describing most of them will end in *-ility*. Some of the typical characteristics defined as nonfunctional requirements are performance, modifiability, usability, configurability, reliability, availability, security, and scalability.

Besides requirements, you will also be given design constraints, such as the choice of programming language, platforms the system runs on, and other systems it interfaces with. These design constraints are sometimes considered nonfunctional requirements. This is not a very crisp or easy-to-define distinction (similar to where requirement analysis ends

and design starts); and in borderline cases, it is defined mainly by consensus. Most developers will include usability as a nonfunctional requirement, and the choice of a specific user interface such as graphical user interface (GUI) or web-based as a design constraint. However, it can also be defined as a functional requirement as follows: "the program displays a dialog box 60 by 80 pixels, and then . . ."

Requirements are established by the client, with help from the software engineer, while the technical decisions are often made by the software engineer without much client input. Oftentimes, some of the technical decisions such as which programming languages or tools to use can be given as requirements because the program needs to interoperate with other programs or the client organization has expertise or strategic investments in particular technologies.

In the following pages we will illustrate the various issues that software engineers confront, even for simple programs. We will categorize these decisions into functional and nonfunctional requirements, design constraints, and design decisions. But do keep in mind that other software engineers may put some of these issues into a different category. We will use the simple sorting problem presented previously as an example.

1.1.2 Functional Requirements

We will have to consider several aspects of the problem and ask many questions prior to designing and programming the solution. The following is an informal summary of the thinking process involved with functional requirements:

- *Input formats:* What is the format for the input data? How should data be stored? What is a character? In our case, we need to define what separates the lines on the file. This is especially critical because several different platforms may use different separator characters. Usually some combination of new-line and carriage return may be considered. In order to know exactly where the boundaries are, we also need to know the input character set. The most common representation uses 1 byte per character, which is enough for English and most Latin-derived languages. But some representations, such as Chinese or Arabic, require 2 bytes per character because there are more than 256 characters involved. Others require a combination of the two types. With the combination of both single- and double-byte character representations, there is usually a need for an escape character to allow the change of mode from single byte to double byte or vice versa. For our sorting problem, we will assume the simple situation of 1 byte per character.
- *Sorting:* Although it seems to be a well-defined problem, there are many slightly and not so slightly different meanings for sorting. For starters—and of course, assuming that we have English characters only—do we sort in ascending or descending order? What do we do with nonalphabetic characters? Do numbers go before or after letters in the order? How about lowercase and uppercase characters? To simplify our problem, we define sorting among characters as being in numerical order, and the sorting of the file to be in ascending order.
- *Special cases, boundaries, and error conditions:* Are there any special cases? How should we handle boundary cases such as empty lines and empty files? How should different error conditions be handled? It is common, although not good practice, to not have all of these requirements completely specified until the detailed design or even the

implementation stages. For our program, we do not treat empty lines in any special manner except to specify that when the input file is empty the output file should be created but empty. We do not specify any special error-handling mechanism as long as all errors are signaled to the user and the input file is not corrupted in any way.

1.1.3 Nonfunctional Requirements

The thinking process involved in nonfunctional requirements can be informally summarized as follows:

- *Performance requirements:* Although it is not as important as most people may think, performance is always an issue. The program needs to finish most or all inputs within a certain amount of time. For our sorting problem, we define the performance requirements as taking less than 1 minute to sort a file of 100 lines of 100 characters each.
- *Real-time requirements:* When a program needs to perform in real-time, which means it must complete the processing within a given amount of time, performance is an issue. The variability of the running time is also a big issue. We may need to choose an algorithm with a less than average performance, if it has a better worst-case performance. For example, `Quick Sort` is regarded as one of the fastest sorting algorithms; however, for some inputs, it can have poor performance. In algorithmic terms, its expected running time is on the order of `n log(n)`, but its worst-case performance is on the order of `n squared`. If you have real-time requirements in which the average case is acceptable but the worst case is not, then you may want to choose an algorithm with less variability, such as `Heap Sort` or `Merge Sort`. Run-time performance analysis is discussed further in Main and Savitch (2010).
- *Modifiability requirements:* Before writing a program, it is important to know the life expectancy of the program and whether there is any plan to modify the program. If the program is to be used only once, then modifiability is not a big issue. On the other hand, if it is going to be used for 10 years or more, then we need to worry about making it easy to maintain and modify. Surely, the requirements will change during that 10-year period. If we know that there are plans to extend the program in certain ways, or that the requirements will change in specific ways, then we should prepare the program for those modifications as the program is designed and implemented. Notice that even if the modifiability requirements are low, this is not a license to write bad code, because we still need to be able to understand the program for debugging purposes. For our sorting example, consider how we might design and implement the solution if we know that down the road the requirement may change from descending to ascending order or may change to include both ascending and descending orders.
- *Security requirements:* The client organization and the developers of the software need to agree on security definitions derived from the client's business application goals, potential threats to project assets, and management controls to protect from loss, inaccuracy, alteration, unavailability, or misuse of the data and resources. Security might be functional or nonfunctional. For example, a software developer may argue that a system must protect against denial-of-service attacks in order to fulfill its mission. Security quality requirements engineering (SQUARE) is discussed in Mead and Stehney (2005).

- *Usability requirements:* The end users for the program have specific background, education, experience, needs, and interaction styles that are considered in the development of the software. The user, product, and environmental characteristics of the program are gathered and studied for the design of the user interface. This nonfunctional requirement is centered in the interaction between the program and the end user. This interaction is rated by the end user with regards to its effectiveness, efficiency, and success. Evaluation of usability requirements is not directly measurable since it is qualified by the usability attributes that are reported by the end users in specific usability testing.

1.1.4 Design Constraints

The thinking process related to design constraints can be summarized as follows:

- *User interface:* What kind of **user interface** should the program have? Should it be a command-line interface (CLI) or a graphical user interface (GUI)? Should we use a web-based interface? For the sorting problem, a web-based interface doesn't sound appropriate because users would need to upload the file and download the sorted one. Although GUIs have become the norm over the past decade or so, a CLI can be just as appropriate for our sorting problem, especially because it would make it easier to invoke inside a script, allowing for automation of manual processes and reuse of this program as a module for future ones. This is one of those design considerations that also involves user interface. In Section 1.4, we will create several implementations, some CLI based and some GUI based. Chapter 7 also discusses user-interface design in more detail.

  User interface What the user sees, feels and hears from the system.

- *Typical and maximum input sizes:* Depending on the typical input sizes, we may want to spend different amounts of time on algorithms and performance optimizations. Also, certain kinds of inputs are particularly good or bad for certain algorithms; for example, inputs that are almost sorted make the naive `Quick Sort` implementations take more time. Note that you will sometimes be given inaccurate estimates, but even ballpark figures can help anticipate problems or guide you toward an appropriate algorithm. In this example, if you have small input sizes, you can use almost any sorting algorithm. Thus you should choose the simplest one to implement. If you have larger inputs but they can still fit into the random access memory (RAM), you need to use an efficient algorithm. If the input does not fit on RAM, then you need to choose a specialized algorithm for on-disk sorting.
- *Platforms:* On which platforms does the program need to run? This is an important business decision that may include architecture, operating system, and available libraries and will almost always be expressed in the requirements. Keep in mind that, although cross-platform development has become easier and there are many languages designed to be portable across platforms, not all the libraries will be available in all platforms. There is always an extra cost on explicitly supporting a new platform. On the other hand, good programming practices help achieve portability, even when not needed. A little extra consideration when designing and implementing a program can minimize the potentially extensive work required to port to a new platform. It is good practice to

perform a quick cost-benefit analysis on whether to support additional platforms and to use technologies and programming practices that minimize portability pains, even when the need for supporting new platforms is not anticipated.

- *Schedule requirements:* The final deadline for completing a project comes from the client, with input from the technical side on feasibility and cost. For example, a dialogue on schedule might take the following form: Your client may make a request such as "I need it by next month." You respond by saying, "Well, that will cost you twice as much than if you wait two months" or "That just can't be done. It usually takes three months. We can push it to two, but no less." The client may agree to this, or could also say, "If it's not done by next month, then it is not useful," and cancel the project.

1.1.5 Design Decisions

The steps and thoughts related to design decisions for the sorting problem can be summarized as follows:

- *Programming language:* Typically this will be a technical design decision, although it is not uncommon to be given as a design constraint. The type of programming needed, the performance and portability requirements, and the technical expertise of the developers often heavily influence the choice of the programming language.
- *Algorithms:* When implementing systems, there are usually several pieces that can be influenced by the choice of algorithms. In our example, of course, there are a variety of algorithms we can choose from to sort a collection of objects. The language used and the libraries available will influence the choice of algorithms. For example, to sort, the easiest solution would be to use a standard facility provided by the programming language rather than to implement your own. Thus, use whatever algorithm that implementation chooses. Performance will usually be the most important influence in the choice of an algorithm, but it needs to be balanced with the effort required to implement it, and the familiarity of the developers with it. Algorithms are usually design decisions, but they can be given as design constraints or even considered functional requirements. In many business environments there are regulations that mandate specific algorithms or mathematical formulas to be used, and in many scientific applications the goal is to test several algorithms, which means that you must use certain algorithms.

1.2 Testing

It is always a good idea to test a program, while it is being defined, developed, and after it is completed. This may sound like obvious advice, but it is not always followed. There are several kinds of testing, including acceptance testing, which refers to testing done by clients, or somebody on their behalf, to make sure the program runs as specified. If this testing fails, the client can reject the program. A simple validation test at the beginning of the project can be done by showing hand-drawn screens of the "problem solution" to the client. This practice solidifies your perception of the problem and the client's solution expectations. The developers run their own internal tests to determine if the program works and is correct. These tests are called verification tests. Validation tests determine whether the developers are building the correct system for the client, and verification tests determine if the system build is correct.

Although there are many types of testing performed by the development organization, the most important kind of verification testing for the individual programmer is unit testing—a process followed by a programmer to test each piece or unit of software. When writing code, you must also write tests to check each module, function, or method you have written. Some methodologies, notably Extreme Programming, go as far as saying that programmers should write the test cases before writing the code; see the discussion on Extreme Programming in Beck and Andres (2004). Inexperienced programmers often do not realize the importance of testing. They write functions or methods that depend on other functions or methods that have not been properly tested. When a method fails, they do not know which function or method is actually failing.

Another useful distinction is between black-box and white-box testing. In black-box testing, the test cases are based only on the requirement specifications, not on the implementation code. In white-box testing, the test cases can be designed while looking at the design and code implementation. While doing unit testing, the programmer has access to the implementation but should still perform a mixture of black-box and white-box testing. When we discuss implementations for our simple program, we will perform unit testing on it. Testing will be discussed more extensively in Chapter 10.

1.3 Estimating Effort

One of the most important aspects of a software project is estimating how much effort it involves. The effort estimate is required to produce a cost estimate and a schedule. Before producing a complete effort estimate, the requirements must be understood. An interesting exercise illustrates this point.

Try the following exercise:

> Estimate how much time, in minutes, it will take you, using your favorite language and technology, to write a program that reads lines from one file and writes the sorted lines to another file. Assume that you will be writing the sort routine yourself and will implement a simple GUI like the one shown in **Figure 1.21**, with two text boxes for providing two file names, and two buttons next to each text box. Pressing one of the two buttons displays a `File Open` dialog, like the one shown in **Figure 1.22**, where the user can navigate the computer's file system and choose a file. Assume that you can work only on this one task, with no interruptions. Provide an estimate within 1 minute (in Step 1).

Step 1.
Estimated ideal total time: _____

Is the assumption that you will be able to work straight through on this task with no interruptions realistic? Won't you need to go to the restroom or drink some water? When can you spend the time on this task? If you were asked to do this task as soon as reasonably possible, starting right now, can you estimate when you would be finished? Given that you start now, estimate when you think you will have this program done to hand over to the client. Also give an estimate of the time you will not be on task (e.g., eating, sleeping, other courses, etc.) in Step 2.

Step 2.
Estimated calendar time started: _____ ended:_____breaks:_____

Now, let's create a new estimate where you divide the entire program into separate developmental tasks, which could be divided into several subtasks, where applicable. Your current task is a planning task, which includes a subtask: ESTIMATION. When thinking of the requirements for the project, assume you will create a class, called `StringSorter`, with three public methods: `Read`, `Write`, and `Sort`. For the sorting routine, assume that your algorithm involves finding the largest element, putting it at the end of the array, and then sorting the rest of the array using the same mechanism. Assume you will create a method called `IndexOfBiggest` that returns the index of the biggest element on the array. Using the following chart, estimate how much time it will take you to do each task (and the GUI) in Step 3.

Step 3.

Ideal Total Time	Calendar Time
`Planning`	
`IndexOfBiggest`	
`Sort`	
`Read`	
`Write`	
GUI	
Testing	
Total	

How close is this estimate to the previous one you did? What kind of formula did you use to convert from ideal time to calendar time? What date would you give the client as the delivery date?

Now, design and implement your solution while keeping track of the time in Step 4.

Step 4.

Keeping track of the time you actually spend on each task as well as the interruptions you experience is a worthwhile data collection activity. Compare these times with your estimates. How high or low did you go? Is there a pattern? How accurate is the total with respect to your original estimate?

If you performed the activities in this exercise, chances are that you found the estimate was more accurate after dividing it into subtasks. You will also find that estimates in general tend to be somewhat inaccurate, even for well-defined tasks. Project estimation and effort estimation is one of the toughest problems in software project management and software engineering. This topic will be revisited in detail in Chapter 13. For further reading on why individuals should keep track of their development time, see the Personal

Software Process (PSP) in Humphrey (1996). Accurate estimation is very hard to achieve. Dividing tasks into smaller ones and keeping data about previous tasks and estimates are usually helpful beginnings.

It is important that the estimation is done by the people who do the job, which is often the programmer. The client also needs to check the estimates for reasonableness. One big problem with estimating is that it is conceptually performed during the bid for the job, which is before the project is started. In reality a lot of the development tasks and information, possibly up to design, is needed in order to be able to provide a good estimate. We will talk more about estimating in Chapter 13.

1.4 Implementations

In this section we will discuss several implementations of our sorting program, including two ways to implement the sort functionality and several variations of the user interface. We will also discuss unit testing for our implementations. Sample code will be provided in Java, using JUnit to aid in unit testing.

1.4.1 A Few Pointers on Implementation

Although software engineering tends to focus more on requirements analysis, design, and processes rather than implementation, a bad implementation will definitely mean a bad program even if all the other pieces are perfect. Although for simple programs almost anything will do, following a few simple rules will generally make all your programming easier. Here we will discuss only a few language-independent rules, and point you to other books in the References and Suggested Readings section at the end of this chapter.

- The most important rule is to be consistent—especially in your choice of names, capitalization, and programming conventions. If you are programming alone, the particular choice of conventions is not important as long as you are consistent. You should also try to follow the established conventions of the programming language you are using, even if it would not otherwise be your choice. This will ensure that you do not introduce two conventions. For example, it is established practice in Java to start class names with uppercase letters and variable names with lowercase letters. If your name has more than one word, use capitalization to signal the word boundaries. This results in names such as `FileClass` and `fileVariable`. In C, the convention is to use lowercase almost exclusively and to separate with an underscore. Thus, when we program in C, we follow the C conventions. The choice of words for common operations is also dictated by convention. For example, printing, displaying, showing, or echoing a variable are some of the terminologies meaning similar actions. Language conventions also provide hints as to default names for variables, preference for shorter or longer names, and other issues. Try to be as consistent as possible in your choice, and follow the conventions for your language.
- Choose names carefully. In addition to being consistent in naming, try to make sure names for functions and variables are descriptive. If the names are too cumbersome or if a good name cannot be easily found, that is usually a sign that there may be a problem in the design. A good rule of thumb is to choose long, descriptive names for things that will have