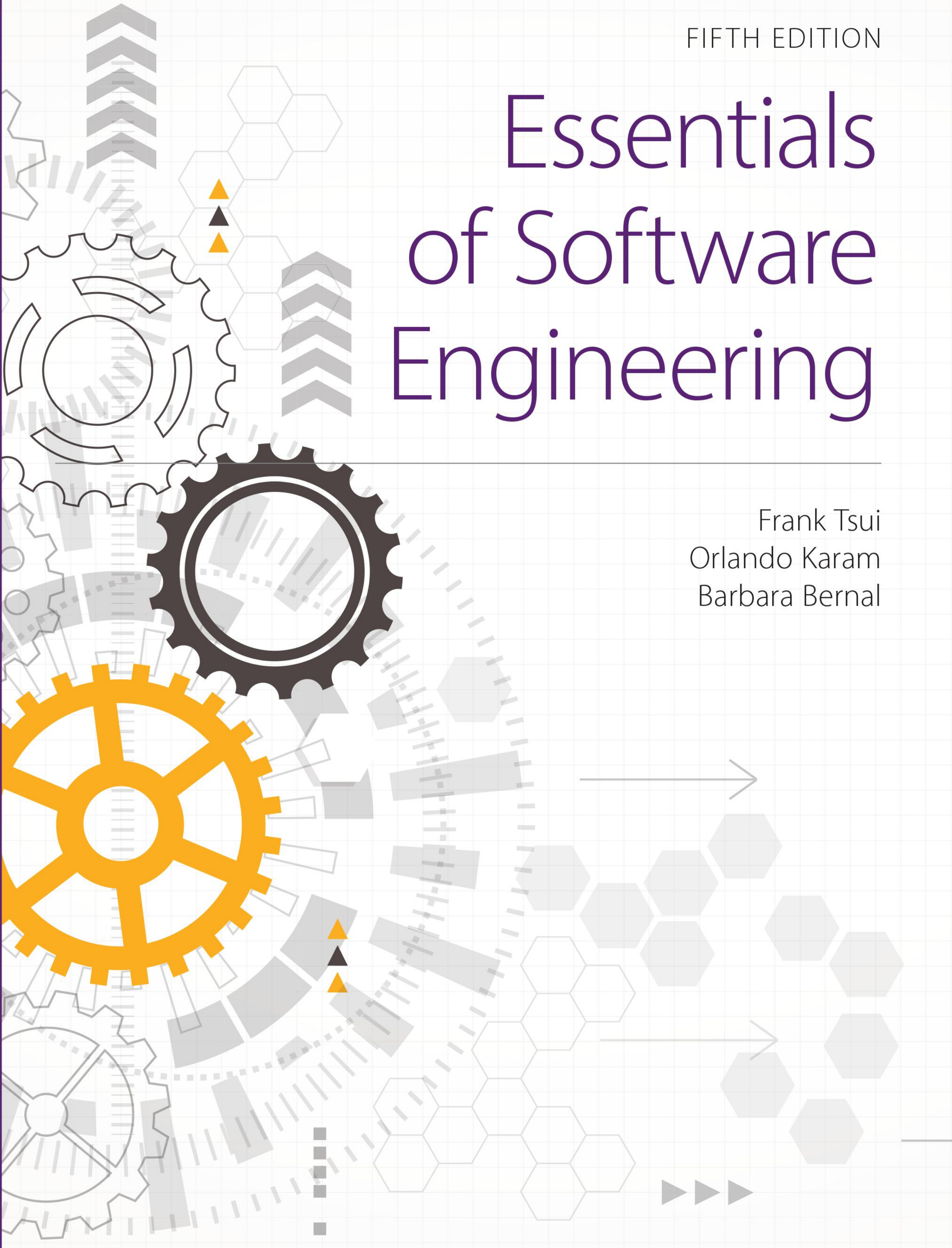
FIFTH EDITION

Essentials of Software Engineering

Frank Tsui
Orlando Karam
Barbara Bernal

JONES & BARTLETT LEARNING



FIFTH EDITION

Essentials of Software Engineering

Frank Tsui
Orlando Karam
Barbara Bernal

JONES & BARTLETT LEARNING

JONES & BARTLETT LEARNING

World Headquarters
Jones & Bartlett Learning
25 Mall Road
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.

Copyright © 2023 by Jones & Bartlett Learning, LLC, an Ascend Learning Company

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The content, statements, views, and opinions herein are the sole expression of the respective authors and not that of Jones & Bartlett Learning, LLC. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement or recommendation by Jones & Bartlett Learning, LLC and such reference shall not be used for advertising or product endorsement purposes. All trademarks displayed are the trademarks of the parties noted herein. *Essentials of Software Engineering, Fifth Edition* is an independent publication and has not been authorized, sponsored, or otherwise approved by the owners of the trademarks or service marks referenced in this product.

There may be images in this book that feature models; these models do not necessarily endorse, represent, or participate in the activities represented in the images. Any screenshots in this product are for educational and instructive purposes only. Any individuals and scenarios featured in the case studies throughout this product may be real or fictitious but are used for instructional purposes only.

37618-0

Production Credits
Vice President, Product Management: Marisa R. Urbano
Vice President, Product Operations: Christine Emerton
Content Coordinator: Mark Restuccia
Product Manager: Ned Hinman
Director, Content Management: Donna Gridley
Manager, Content Strategy: Carolyn Pershouse
Content Strategist: Melissa Duffy
Director, Project Management and Content Services: Karen Scott
Project Manager: Jessica deMartin
Project Specialist: Lori Weidert
Senior Digital Project Specialist: Angela Dooley
Marketing Manager: Suzy Balk
Composition: Straive
Project Management: Straive
Cover Design: Briana Yates
Media Development Editor: Faith Brosnan
Rights Specialist: James Fortney
Cover Image (Title Page, Part Opener, Chapter Opener):
© Zally23/Shutterstock
Printing and Binding: PA Hutchison Company

Library of Congress Cataloging-in-Publication Data
Names: Tsui, Frank F., author. | Bernal, Barbara, author. | Karam, Orlando, author.
Title: Essentials of software engineering / Frank Tsui, Kennesaw State University, Barbara Bernal, Southern Polytechnic State University (now retired), Orlando, Karam.
Description: Fifth edition. | Burlington : Jones & Bartlett Learning, 2022. | Includes bibliographical references and index. | Summary: “The basic concepts and theories of software engineering have stabilized considerably from the early days of thirty to forty years ago. Nevertheless, the technology and tools continue to evolve, expand and improve every four to five years. In this fifth edition, we will cover some of these newly established improvements in technology and tools but reduce some areas, such as process assessment models, that is becoming less relevant today. We will still maintain many of the historically important concepts that formed the foundation to this field, such as the traditional process models. Our goal is to continue to keep the content of this book to a concise amount that can be taught in a 16-week semester introductory course”– Provided by publisher.
Identifiers: LCCN 2021032440 | ISBN 9781284228991 (paperback)
Subjects: LCSH: Software engineering.
Classification: LCC QA76.758 .T78 2022 | DDC 005.1–dc23
LC record available at https://lccn.loc.gov/2021032440

6048

Printed in the United States of America
26 25 24 23 22 10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

Chapter opener image: © Zally23/Shutterstock

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 one-semester course that meets approximately three hours a week for sixteen 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 twenty-five 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 Fifth Edition

The basic concepts and theories of software engineering have stabilized considerably from the early days of thirty to forty years ago. Nevertheless, the technology and tools continue to evolve, expand, and improve every four to five years. In this fifth edition, we cover some of these newly established improvements in technology and tools but reduce some areas, such as process assessment models, that are becoming less relevant today. We will still maintain many of the historically important concepts that formed the foundation to this field, such as the traditional process models. Our goal is to continue to keep the content of this book to a concise amount that can be taught in a sixteen-week semester introductory course. The major modifications to this fifth edition are as follows"

- An existing and historical notion of "continuous integration" has expanded into a newer concept called Continuous Integration and Continuous Deployment (CI/CD) and picked

Chapter opener image: © Zally23/Shutterstock

up momentum with improved tools and maturing Agile methods. This is discussed in Chapter 2.

- To reflect more current thinking and terminologies, Chapter 4 is retitled as Traditional Software Process Models. Chapter 4's discussion on process assessment models, especially Capability Maturity Model Integrated (CMMI), is greatly reduced. Chapter 5 is retitled as Agile Software Process Models to more accurately reflect its contents. An extension to the popular Agile methodologies called Development and Operations or DevOps is added in Chapter 5 as the next level of new improvement in process.
- Many of the current design and development related ideas and tools such as Service Oriented Architecture (SOA), Enterprise Service Bus and microservices are added to Chapter 7.
- Some of the newer concepts and tools associated with virtualization and containerization are added in Chapter 9.
- To parallel the continuous integration and CI/CD discussions, the newer GitHub/Git tool is included in section 11.5 of Chapter 11.
- Although security is a very important topic, it has grown to be a separate, stand-alone discipline encompassing the software, hardware, and information infrastructure technology and services subjects. Instead of devoting a thorough treatise to this topic, a discussion of the more recent consideration of security that comes with approaches like Secure DevOps or DevSecOps is added to section 14.1 of Chapter 14.

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, third, and fourth editions of the book from different readers and universities and have corrected the grammatical and spelling errors. Any remaining error is totally ours.

The first through the fourth 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 fifth 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 user interface design discussion with an example of HTML-Script-structured query language (SQL) design and implementation. Chapter 8, following the chapter on software design, steps back and discusses design characteristics and metrics used in evaluating high-level and detailed designs. Chapter 11 discusses not only product release but also 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—Institute of Electrical and Electronics Engineers (IEEE) Standard
  - Example 4: Essential SRS—Narrative Approach
- Appendix C: Essential Software Design
  - Example 1: Essential Software Design—Unified Modeling Language (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 one 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, Source code, and sample Test Questions are available for free instructor download. To request access, please visit go.jblearning.com/Tsui5e 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 their work:

- Alan C. Verbit, Delaware County Community College
- Ayad Boudiab, Georgia Perimeter College
- Badari Eswar, San Jose State University
- Ben Geisler, University of Wisconsin, Green Bay
- Benjamin Sweet, Lawrence Technological University
- Brent Auernheimer, California State University, Fresno
- Bruce Logan, Lesley University
- Chip Anderson, Lake Washington Institute of Technology
- Dar-Biau Liu, California State University, Long Beach

- David Gustafson, Kansas State University
- Donna DeMarco, Kutztown University
- Dr. Alex Rudniy, University of Scranton
- Dr. Anthony Ruocco, Roger Williams University
- Dr. Andrew Scott, Western Carolina University
- Dr. Christopher Fox, James Madison University
- Dr. David A. Cook, Stephen F. Austin State University
- Dr. David Burris, Sam Houston State University
- Dr. Dimitris Papamichail, The College of New Jersey
- Dr. Edward G. Nava, University of New Mexico
- Dr. Emily Navarro, University of California, Irvine
- Dr. Jeff Roach, East Tennessee State University
- Dr. Jody Paul, Metro State Denver
- Dr. John Dalbey, California Polytechnic State University
- Dr. Jason Hibbeler, University of Vermont
- Dr. Jianchao Han, California State University Dominguez Hills
- Dr. Joe Hoffert, Indiana Wesleyan University
- Dr. Kenneth Magel, North Dakota State University
- Dr. Mazin Al-Hamando, Lawrence Technological University
- Dr. Michael Murphy, Concordia University Texas
- Dr. Reza Eftekari, George Washington University, University of Maryland at College Park
- Dr. Ronald Finkbine, Indiana University Southeast
- Dr. Sofya Poger, Felician University
- Dr. Sen Zhang, SUNY Oneonta
- Dr. Stephen Hughes, Coe College
- Dr. Steve Kreutzer, Bloomfield College
- Dr. Yenumula B. Reddy, Grambling State University
- Frank Ackerman, Montana Tech
- Ian Cottingham, Jeffrey S. Raikes School at The University of Nebraska, Lincoln
- Jeanna Matthews, Clarkson University
- John Sturman, Rensselaer Polytechnic Institute
- Kai Chang, Auburn University
- Katia Maxwell, Athens State University
- Lenis Hernandez, Florida International University
- Larry Stein, California State University, Northridge
- Mark Hall, Hastings College
- Michael Oudshoorn, Montana State University
- Paul G. Garland, Johns Hopkins University
- Salvador Almanza-Garcia, Vector CANtech, Inc.
- Theresa Jefferson, George Washington University
- William Saichek, Orange Coast College

We continue to appreciate the help from Melissa Duffy, Edward Hinman, Paula-Yuan Gregory, Baghyalakshmi Jagannathan, Padmapriya Soundararajan, Lori Weidert, and others at Jones & Bartlett Learning. Any remaining error is solely the mistake of the authors.

—Frank Tsui

—Orlando Karam

—Barbara Bernal

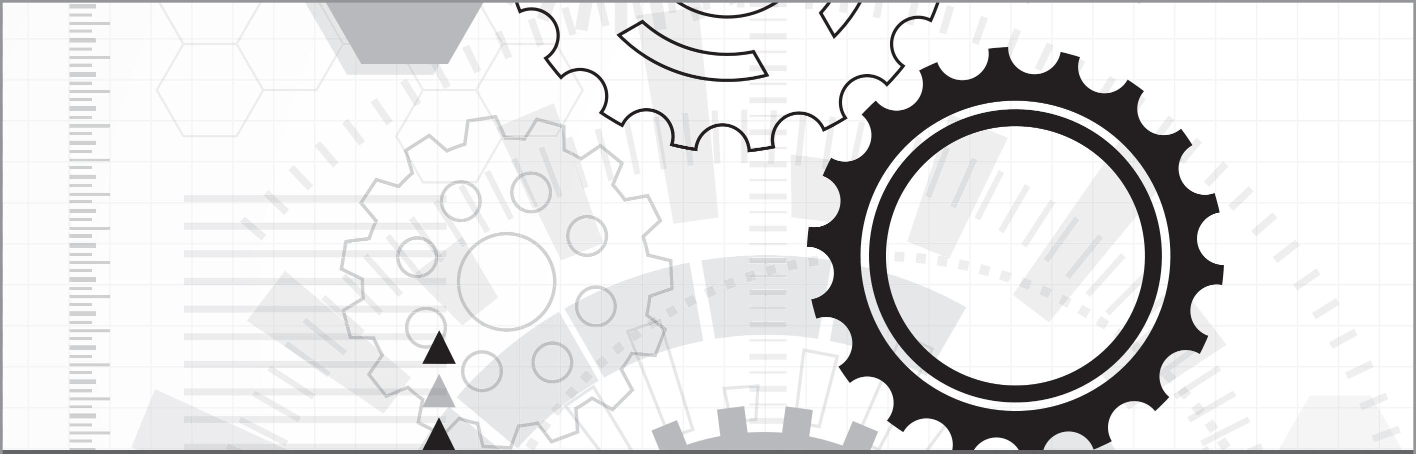
ABOUT THE AUTHORS

Frank Tsui Frank Tsui worked in the software industry since the early 1970s for more than twenty-five years before joining academia and teaching both undergraduate and graduate students. He has now retired and is advising some of his past students in their career choices. Frank's formal education includes a BS degree from Purdue University, MS degree from Indiana State University, and a PhD in computer science from Georgia Tech.

Orlando Karam Orlando Karam's experiences are in Agile development and open source environment. He has also developed software for the Yucatan State Government and several companies in Mexico. Orlando holds a PhD in computer science from Tulane University and is a faculty member of Kennesaw State University. Orlando is also actively involved in the studies of complexities of software. Orlando has spent the last 8 years working at places like Microsoft and Amazon.

Barbara Bernal Barbara is a professor emeritus of software engineering at Kennesaw State University. Her expertise is in the area of user interfaces and user-centered design. She has been active in the American Society for Engineering Education and the education of software engineers.

Chapter opener image: © Zally23/Shutterstock



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.

Chapter opener image: © Zally23/Shutterstock

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 preceding 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 to 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, whereas 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 before 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 one byte per character, which is enough for English and most Latin-derived languages. But some representations, such as Chinese or Arabic, require two 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 one 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 one 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 ten years or more, then we need to worry about making it easy to maintain and modify. Surely, the requirements will change during that ten-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 because 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 the user sees, feels, and hears from the system.

- *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.
- *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 dialog 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 one 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 and effort estimation is one of the toughest problems in software project management and software engineering. 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. This topic will be revisited in detail in Chapter 13.

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 global scope such as classes and public methods. Use short names for local references, which are used in a very limited scope such as local variables, private names, and so on.
- Test before using a function or method. Make sure that it works. That way if there are any errors, you know that they are in the module you are currently writing. Careful unit testing, with test cases written before or after the unit, will help you gain confidence in using that unit.
- Know thy standard library. In most modern programming languages, the standard library will implement many common functions, usually including sorting and collections of data, database access, utilities for web development, networking, and much more. Don't reinvent or reimplement the wheel. Using the standard libraries will save extra work, make the code more understandable, and usually run faster with fewer errors because the standard libraries are well debugged and optimized. Keep in mind that many exercises in introductory programming classes involve solving classic problems and implementing well-known data structures and algorithms. Although they are a valuable learning exercise, that does not mean you should use your own implementations in real life. For our sample programming problem, Java has a sorting routine that is robust and fast. Using it instead of writing your own would save time and effort and produce a better implementation. We will still implement our own for the sake of illustration but will also provide the implementation using the Java sorting routine.
- If possible, perform a review of your code. Software reviews are one of the most effective methods for reducing defects in software. Showing your code to other people will help detect not just functionality errors but also inconsistencies and bad naming. It will also help you learn from the other person's experience. This is another habit that does not blend well with school projects. In most such projects, getting help from another student might be considered cheating. Perhaps the code can instead be reviewed after it is handed in. Reviews are good for school assignments as well as for real-world programs.

1.4.2 Basic Design

Given that we will be implementing different user interfaces, our basic design separates the sorting functionality from the user interface, which is a good practice anyway because user interfaces tend to change much faster than functionality. We have a class, called `StringSorter`, that has four methods: (1) reading the strings from a file, (2) sorting the collection of strings, (3) writing the strings to a file, and (4) combining those three, taking the input and output file names. The different user interfaces will be implemented in separate classes. Given that `StringSorter` would not know what to do with exceptional conditions, such as errors when reading or writing streams, the exceptions pass through in the appropriate methods, with the user interface classes deciding what to do with them. We also have a class with all our unit tests, taking advantage of the `JUnit` framework.

1.4.3 Unit Testing with `JUnit`

`JUnit` is one of a family of unit testing frameworks, the `J` standing for Java. There are variations for many other languages—for example, `cppUnit` for C++; the original library was developed in `Smalltalk`. We just need to create a class that inherits from `junit.framework.TestCase`, which defines public methods whose names start with test. `JUnit` uses Java's reflection capabilities to execute all those methods. Within each test method, `assertEquals` can be used to verify whether two values that should be equal are truly equal. Here we discuss `JUnit` in a very basic way; `JUnit` is discussed further in Chapter 10.

1.4.4 Implementation of `StringSorter`

We will be presenting our implementation followed by the test cases. We are assuming a certain fundamental background with Java programming, although familiarity with another object-oriented programming language should be enough to understand this section. Although the methods could have been developed in a different order, we present them in the order we developed them, which is `Read`, then `Sort`, then `Write`. This is also the order in which the final program will execute, thus making it easier to test.

We import several namespaces, and declare the `StringSorter` class. The only instance variable is an `ArrayList` of lines. `ArrayList` is a container that can grow dynamically, and supports indexed access to its elements. It roughly corresponds to a vector in other programming languages. It is part of the standard Java collections library and another example of how using the standard library saves time. Notice we are not declaring the variable as private in **FIGURE 1.1** because the test class needs access to it. By leaving it with default protection, all classes in the same package can access it because Java has no concept like friend classes in C++. This provides a decent compromise. Further options will be discussed in Chapter 10. Our first method involves reading lines from a file or stream, as seen in **FIGURE 1.2**. To make the method more general, we take a `Reader`, which is a class for reading text-based streams. A stream is a generalization of a file. By using a `Reader` rather than a class explicitly based on `Files`, we could use this same method for reading from standard input or even from the network. Also, because we do not know how to deal with exceptions here, we will just let the `IOException` pass through.

```
import java.io.* ; // for Reader (and subclasses) , Writer (and subclasses) and IOException
import java.util.* ; // for List , ArrayList , Iterator

public class StringSorter {
    ArrayList<String> lines ;
```

FIGURE 1.1 Class declaration and `Import` statements.

```
    public void readFromStream( Reader r ) throws IOException
    {
        BufferedReader br=new BufferedReader ( r ) ;
        lines=new ArrayList<String> ( ) ;

        while ( true ) {
            String input=br . readLine ( ) ;
            if ( input==null )
                break ;
            lines . add ( input ) ;
        }
    }
```

FIGURE 1.2 The `readFromStream` method.

```
public class TestStringSorter extends TestCase {
    private ArrayList<String> make123 ( ) {
        ArrayList<String> l = new ArrayList<String> ( ) ;
        l . add ( "one" ) ;
        l . add ( "two" ) ;
        l . add ( "three" ) ;
        return l ;
    }
```

FIGURE 1.3 `TestStringSorter` declaration and `make123` method.

```
    public void testReadFromStream( ) throws IOException{
        Reader in=new FileReader ( "in.txt" ) ;
        StringSorter ss=new StringSorter();
        ArrayList<String> l= make123 ( ) ;
        ss . readFromStream ( in ) ;

        assertEquals ( l , ss . lines ) ;
    }
```

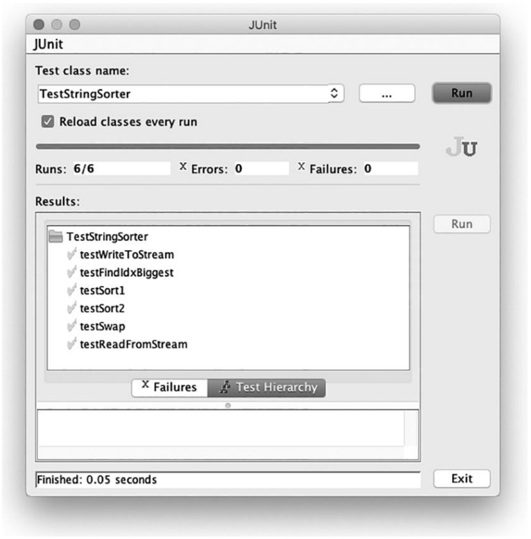
FIGURE 1.4 `testReadFromStream.`

For testing this method with `JUnit`, we create a class extending `TestCase`. We also define a utility method, called `make123`, that creates an `ArrayList` with three strings—`one`, `two`, and `three`—inserted in that order in **FIGURE 1.3**.

We then define our first method, `testReadFromStream`, in **FIGURE 1.4**. In this method we create an `ArrayList` and a `StringSorter`. We open a known file and make the `StringSorter` read from it. Given that we know what is in the file, we know what the internal `ArrayList` in our `StringSorter` should be. We just assert that it should be equal to that known value.

We can run `JUnit` after setting the classpath and compiling both classes, by typing `java junit.swingui.TestRunner.` This will present us with a list of classes to choose from. When choosing our `TestStringSorter` class, we find a user interface like the one shown in **FIGURE 1.5**, which indicates that all tests are implemented and successfully run. Pressing the run button will rerun all tests, showing you how many tests were successful. If any test is unsuccessful, the bar will be red rather than green. Classes are reloaded by default, so you can leave that window open, modify, recompile, and just press run again.

After we verify that our test is successful, we can begin the next method—building the sorting functionality. We decided on a simple algorithm: find the largest element in the array, then swap it


JUnit

Test class name: TestStringSorter

☑ Reload classes every run

Runs: 6/6 Errors: 0 Failures: 0

Results:

TestStringSorter
- testWriteToStream
- testFindIdxBiggest
- testSort1
- testSort2
- testSwap
- testReadFromStream

Finished: 0.05 seconds

FIGURE 1.5 `JUnit` **GUI.**

Courtesy of JUnit.

```
static void swap ( List<String> l, int i1, int i2 ) {
    String tmp=l . get ( i1 ) ;
    l . set ( i1, l . get ( i2 ) ) ;
    l . set ( i2 , tmp ) ;
}
```

FIGURE 1.6 The code for swapping two integers.

```
public void testSwap ( ) {
    ArrayList<String> l1= make123 ( ) ;

    ArrayList<String> l2=new ArrayList<String> ( ) ;
    l2 . add ( "one" ) ;
    l2 . add ( "three" ) ;
    l2 . add ( "two" ) ;

    StringSorter . swap ( l1 , 1,2 ) ;
    assertEquals ( l1, l2 ) ;
}
```

FIGURE 1.7 The `testSwap` **method.**

with the last element, placing the largest element at the end of the array, then repeat with the rest of the array. We need two supporting functions, one for swapping the two elements in the array and another for finding the index of the largest element. The code for a swap is shown in **FIGURE 1.6.**

Because swap is a generic function that could be reused in many situations, we decided to build it without any knowledge of the `StringSorter` class. Given that, it makes sense to have it as a static method. In C++ or other languages, it would be a function defined outside the class and not associated with any class. Static methods are the closest technique in Java. We get as parameters a `List`, where `List` is the generic interface that `ArrayList` implements, and the indexes of the two elements. The test for this method is shown in the `testSwap` method of `TestStringSorter` class in **FIGURE 1.7**.

```
static int findIdxBiggest ( List<String> l, int from, int to) {
    String biggest=l . get ( 0 ) ;
    int idxBiggest=from ;

    for ( int i=from+1; i<=to; ++i ) {
        if ( biggest . compareTo ( l . get ( i ) ) <0 ) {// it is bigger than biggest
            biggest= l . get ( i ) ;
            idxBiggest=i ;
        }
    }
    return idxBiggest ;
}
```

FIGURE 1.8 The `findIdxBiggest` method.

```
public void testFindIdxBiggest ( ) {
    StringSorter ss=new StringSorter ( ) ;
    ArrayList<String> l = make123 ( ) ;

    int i=StringSorter . findIdxBiggest( l , 0 , l . size ( ) -1 ) ;
    assertEquals ( i , 1) ;
}
```

FIGURE 1.9 The `testFindIdxBiggest` method.

```
public void sort() {
    for ( int i=lines . size ( ) - 1 ; i>0 ; --i ) {
        int big=findIdxBiggest ( lines , 0 , i ) ;
        swap ( lines , i , big ) ;
    }
}
```

FIGURE 1.10 The `sort` method.

The next method is the one that returns the index of the largest element on the list. Its name is `findIdxBiggest`, as shown in **FIGURE 1.8**. `Idx` as an abbreviation of index is ingrained in our minds. We debated whether to use `largest`, `biggest`, or `max/maximum` for the name (they are about equally appropriate in our minds). After settling on `biggest`, we just made sure that we did not use the other two for naming the variables.

We use the `compareTo` method of `Strings`, which returns –1 if the first element is less than the second, 0 if they are equal, and 1 if the first is largest. In this method we use the fact that the elements in the `ArrayList` are strings. Notice that Java (as of version 1.4) does not have support for generics (templates in C++), so the elements have to be explicitly casted to `Strings`. The test is shown in **FIGURE 1.9**.

With `swap` and `findIdxBiggest` in place, the `sort` method, shown in **FIGURE 1.10**, becomes relatively easy to implement. The test for it is shown in **FIGURE 1.11**. Note that if we knew our standard library, we could have used a much easier implementation, using the `sort` function in the standard Java library, as shown in **FIGURE 1.12**. We would have also avoided writing `swap` and `findIdxBiggest`! It definitely pays to know your standard library.

Now on to writing to the file; this is shown in **FIGURE 1.13**. We will test it by writing a known value to the file, then reading it again and performing the comparison in **FIGURE 1.14**. Now all that is needed

```
public void testSort1() {
      StringSorter ss= new StringSorter ( ) ;
      ss . lines=make123 ( ) ;

      ArrayList<String> l2=new ArrayList<String>( ) ;
      l2 . add ( "one" ) ;
      l2 . add ( "three" ) ;
      l2 . add ( "two" ) ;

      ss . sort ( ) ;

      assertEquals (l2 , ss . lines ) ;
}
```

FIGURE 1.11 The `testSort1` method.

```
void sort ( ) {
      java . util . Collections . sort ( lines ) ;
}
```

FIGURE 1.12 The `sort` method using Java's standard library.

```
public void writeToStream ( Writer w ) throws IOException {
      PrintWriter pw=new PrintWriter ( w ) ;
      Iterator i=lines . iterator ( ) ;
      while ( i . hasNext ( ) ) {
            pw . println ( ( String ) ( i . next ( ) ) ) ;
      }
}
```

FIGURE 1.13 The `writeToStream` method.

```
public void testWriteToStream ( ) throws IOException{
      StringSorter ss1=new StringSorter ( ) ;
      ss1 . lines=make123 ( ) ;
      Writer out=new FileWriter ( "test.out" ) ;
      ss1 . writeToStream ( out ) ;
      out . close ( ) ;

      // then read it and compare
      Reader in=new FileReader ( "in.txt" ) ;
      StringSorter ss2=new StringSorter ( ) ;
      ss2 . readFromStream ( in ) ;
      assertEquals (ss1 . lines , ss2 . lines ) ;
}
```

FIGURE 1.14 The `testWriteToStream` method.

is the `sort` method taking the file names as shown in **FIGURE 1.15**. Given that we have already seen how to do it for the test cases, it is very easy to do. The test for this method is shown in **FIGURE 1.16**.

1.4.5 User Interfaces

We now have an implementation of `StringSorter` and a reasonable belief that it works as intended. We realize that our tests were not that extensive; however, we can go on to build a user interface,

```
    public void sort ( String inputFileName , String outputFileName ) throws IOException{
            Reader in=new FileReader ( inputFileName ) ;
            Writer out=new FileWriter ( outputFileName ) ;

            StringSorter ss=new StringSorter ( ) ;
            ss . readFromStream ( in ) ;
            ss . sort ( ) ;
            ss . writeToStream ( out ) ;

            in . close ( ) ;
            out . close ( ) ;
    }
```

FIGURE 1.15 The `sort` method (taking file names).

```
    public void testSort2 ( ) throws IOException{
        StringSorter ss1=new StringSorter ( ) ;
        ss1 . sort ( "in.txt" , "test2.out" ) ;
        ArrayList<String> l=new ArrayList<String> ( ) ;
        l . add ( "one" ) ;
        l . add ( "three" ) ;
        l . add ( "two" ) ;
        Reader in=new FileReader ( "test2 . out" ) ;
        StringSorter ss2=new StringSorter ( ) ;
        ss2 . readFromStream ( in ) ;
        assertEquals ( l , ss2 . lines ) ;
    }
```

FIGURE 1.16 The `testSort2` method.

```
import java. io . IOException ;
public class StringSorterCommandLine {
    public static void main ( String args [ ] ) throws IOException {
        if ( args . length ! =2 ) {
            System . out . println ( "Usage: java Sort1 inputfile outputfile" ) ;
        } else {
            StringSorter ss=new StringSorter ( ) ;
            ss . sort ( args [ 0 ] , args [ 1 ] ) ;
        }
    }
}
```

FIGURE 1.17 The `StringSorter-CommandLine` class, which implements a command-line interface for `StringSorter` functionality.

which is an actual program that lets us access the functionality of `StringSorter`. Our first implementation is a command-line, not GUI, version, as shown in **FIGURE 1.17**. It takes the names of the input and output files as command parameters. Its implementation is as shown in the figure.

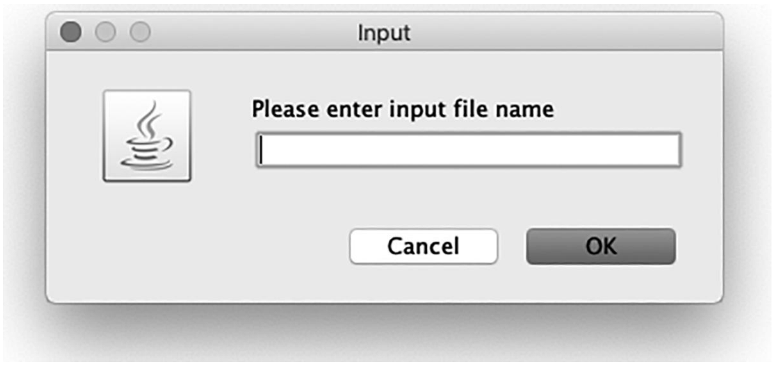
We would use it by typing the following command:

```
java StringSorterCommandLine abc.txt abc_sorted.txt
```

Do you believe this is a useful user interface? Actually, for many people it is. If you have a command-line window open all the time or if you are working without a GUI, then it is not that hard to type

```
public class StringSorterBadGUI {
    public static void main ( String args [ ] ) throws IOException {
        try {
            StringSorter ss=new StringSorter ( ) ;
            String inputFileName= JOptionPane . showInputDialog ( "Please enter input file name" ) ;
            String outputFileName= JOptionPane . showInputDialog ( "Please enter output file name" ) ;
            ss . sort ( inputFileName , outputFileName ) ;
        } finally {
            System . exit ( 1 ) ;
        }
    }
}
```

FIGURE 1.18 `StringSorterBadGUI` **class, which implements a hard-to-use GUI for** `StringSorter` **functionality.**



Please enter input file name

FIGURE 1.19 An input file name dialog box for a hard-to-use GUI.

Screenshot(s) reprinted with permission from Apple Inc.

the command. Also, it is very easy to use this command inside a script, to sort many files. In fact, you could use a script to more thoroughly test your implementation. Another important advantage, besides scriptability, is how easy it is to build the interface. This means less effort, lower costs, and fewer errors.

However, for some people this would not be a useful interface. If you are accustomed to only using GUIs or if you do not usually have a command window open and are not going to be sorting many files, then a GUI would be better. Nevertheless, GUI is not necessarily a better interface than a CLI. It depends on the use and the user. Also, it is extremely easy to design bad GUIs, such as the implementation shown in **FIGURE 1.18**. The code in this figure would display the dialog box shown in **FIGURE 1.19**. After the user presses OK, the dialog box in **FIGURE 1.20** would be shown. Notice the title “Input” in the top of the dialog box and the message “Please enter output file name” in Figure 1.20. This could be a communication contradiction for the user.

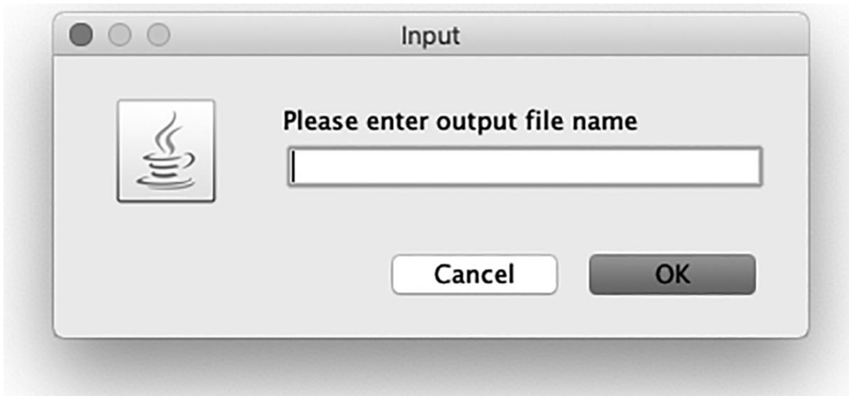


FIGURE 1.20 An output file name dialog for a hard-to-use GUI.

Screenshot(s) reprinted with permission from Apple Inc.

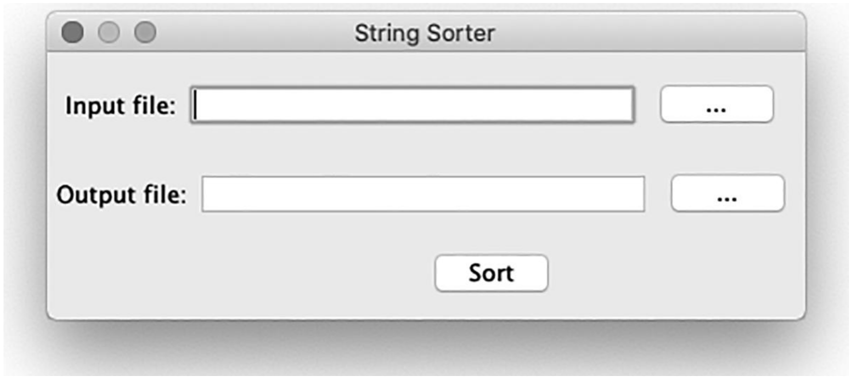
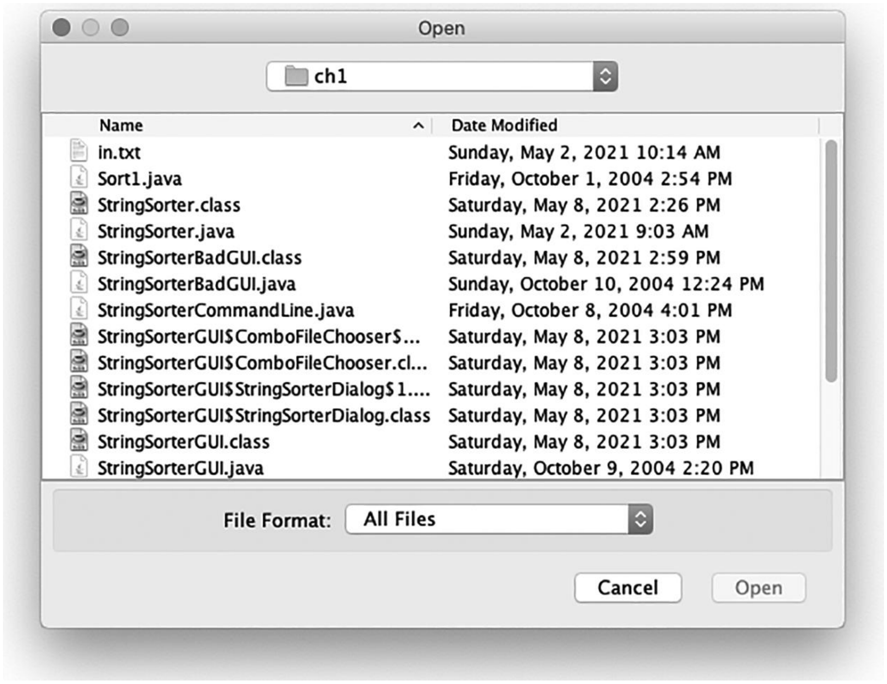


FIGURE 1.21 `Input` **and** `Output` **file name dialog for GUI.**

Screenshot(s) reprinted with permission from Apple Inc.

This does not involve much more effort than the command-line version, but it is very inefficient to use. Although it is a GUI, it is worse than the CLI for almost every user. A better interface is shown in **FIGURE 1.21**. Although it is not a lot better, at least both inputs are in the same place. What makes it more useful is that the buttons on the right open a dialog box as shown in **FIGURE 1.22** for choosing a file.

This would at least be a decent interface for most GUI users. Not terribly pretty, but simple and functional. The code for this GUI is available on the website for this book. We are not printing it because it requires knowledge of Java and Swing to be understood. We will note that the code is 75 lines long in Java, a language for which GUI building is one of its strengths, and it took us longer to produce than the `StringSorter` class! Sometimes GUIs come with a heavy cost. We will discuss user-interface design in Chapter 7.



ch1

Name	Date Modified
in.txt	Sunday, May 2, 2021 10:14 AM
Sort1.java	Friday, October 1, 2004 2:54 PM
StringSorter.class	Saturday, May 8, 2021 2:26 PM
StringSorter.java	Sunday, May 2, 2021 9:03 AM
StringSorterBadGUI.class	Saturday, May 8, 2021 2:59 PM
StringSorterBadGUI.java	Sunday, October 10, 2004 12:24 PM
StringSorterCommandLine.java	Friday, October 8, 2004 4:01 PM
StringSorterGUI$ComboFileChooser$...	Saturday, May 8, 2021 3:03 PM
StringSorterGUI$ComboFileChooser.cl...	Saturday, May 8, 2021 3:03 PM
StringSorterGUI$StringSorterDialog$1....	Saturday, May 8, 2021 3:03 PM
StringSorterGUI$StringSorterDialog.class	Saturday, May 8, 2021 3:03 PM
StringSorterGUI.class	Saturday, May 8, 2021 3:03 PM
StringSorterGUI.java	Saturday, October 9, 2004 2:20 PM

File Format: All Files

FIGURE 1.22 `File Open` **dialog for GUI.**

Screenshot(s) reprinted with permission from Apple Inc.

1.5 Summary

In this chapter we have discussed some of the many issues involved in writing a simple program. By now, you should have realized that even for simple programs there is much more than just writing the code. One has to consider many of the following items:

- Requirements
- Design
- Code implementation
- Unit testing
- Personal effort estimation
- User interface

Much of that material belongs to software engineering, and in this text we will provide an overview of it.

1.6 Review Questions

1.1 What are statements that define and qualify what the program needs to do?

1.2 What are statements that constrain the ways in which the software can be designed and implemented?

1.3 Which type of requirement statement defines what the program needs to do?

1.4 What requirements qualify as functional requirements? Specify in what manner they need to be achieved.

1.5 Which decisions are those taken by the software engineer about the best ways (processes, techniques, and technologies) to achieve the requirements?

1.6 What type of testing refers to testing done by the clients (or somebody on their behalf) to make sure the program runs as specified?

1.7 What is GUI? What is CLI?

1.8 List three of the typical kinds of nonfunctional requirements.

1.7 Exercises

1.1 For your next two software projects (assuming that you are getting programming assignments; otherwise consider a program to find the `max` and the `min` of a set of rational numbers) estimate how much effort they would take before doing them, then keep track of the actual time spent. How accurate were your estimates?

1.2 What sequence of activities did you observe in considering the programming effort discussed in this chapter?

1.3 Discuss whether you think a programming language constraint may be viewed as a requirement. Explain why you think so.

1.4 Download the programs for this chapter, and add at least one more test case for each method of the `StringSorter` class.

1.5 In the discussion of the simple program in this chapter, what were the items considered for "basic" design? Would you have written down these considerations and perhaps reviewed them with a trusted person before the actual coding?

1.6 Consider a CLI that, rather than taking the file names as parameters, asks for them from the keyboard (e.g., it displays "Input file name:" then reads it from the keyboard). Would this be a better user interface? Why or why not?

1.7 Consider a new user interface for our sorting program that combines the CLI and the GUI. If it receives parameters in the command line, it does the sort. If it does not, it displays the dialog. Would this be a better interface? What would be its advantages and disadvantages compared with other interfaces?

1.8 References and Suggested Readings

Beck, K., and C. Andres. 2004. *Extreme Programming Explained: Embrace Change*, 2nd ed. Reading, MA: Addison-Wesley.

Dale, N., C. Weems, and M. R. Headington. 2003. *Programming and Problem Solving with Java*. Sudbury, MA: Jones and Bartlett.

Humphrey, W. 1996. *Introduction to the Personal Software Process*. Reading, MA: Addison-Wesley.

Hunt, A., and D. Thomas. 2003. *Pragmatic Unit Testing in Java with Junit*. Sebastopol, CA: Pragmatic Bookshelf.

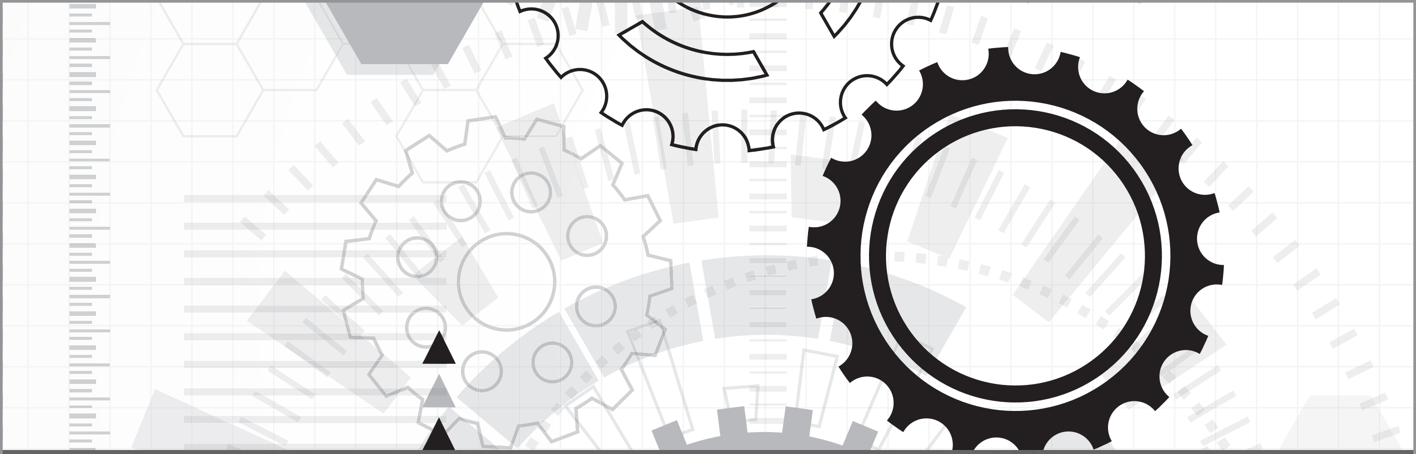
Kernighan, B. W., and R. Pike. 1999. *The Practice of Programming*. Reading, MA: Addison-Wesley.

Main, M., and W. Savitch. 2010. *Data Structures and Other Objects Using C++*, 4th ed. Reading, MA: Addison-Wesley.

McConnell, S. 2004. *Code Complete 2*. Redmond, WA: Microsoft Press.

Mead, N. R., and T. Stehney. 2005. "Security Quality Requirements Engineering (SQUARE) Methodology." In *SESS '05 Proceedings of the 2005 Workshop on Software Engineering for Secure Systems*, 1–7. New York: Association for Computing Machinery.

Wu, C. T. 2009. *Introduction to Objected Oriented Programming with Java*, 5th ed. New York: McGraw-Hill.



CHAPTER **2**

Building a System

Objectives

- Characterize the size and complexity issues of a system.
- Describe the technical issues in development and support of a system.
- Describe the nontechnical issues of developing and supporting a system.
- Demonstrate the concerns in the development and support activities of a large application software, using a payroll system example.
- Describe the coordination efforts needed for process, product, and people. These software engineering topics are expanded in later chapters.

Chapter opener image: © Zally23/Shutterstock

2.1 Characteristics of Building a System

The previous chapter focused on the environment and the conditions under which a single program may be developed by one person for, perhaps, just a few users. We have already seen multiple items that must be considered even when one person is writing a single program. In this chapter we will describe the problems and concerns associated with building a system that contains multiple components—anything from just a few components to maybe hundreds or thousands of components. The increase in number of components and complexity is what requires us to study and understand the various aspects, principles, and techniques of software engineering. This discussion introduces the rationale for software engineering as a discipline, especially for large and complex projects that require a team of people.

2.1.1 Size and Complexity

As software becomes ubiquitous, the development of systems involving software is also becoming more complex. Inherently, large projects involve more parts, more tasks, more people, and more sophisticated tools. Project size and complexity are closely intertwined. Software engineers are asked to solve both simple and complex problems and to deal with the distinct differences between them. The complex problems come in multiple levels of both: breadth and depth. The breadth issue addresses the sheer numbers involving the following:

- Number of major functions
- Number of features within each functional area
- Number of interfaces and linkages to other components or to other external systems
- Number of simultaneous users
- Number of types of data and data structures

The depth issue addresses the linkage and the relationships among items. The linkages may either be through the sharing of data or through the transfer of control or both. These relationships may be hierarchical, sequential, loop, recursive, or some other form. In the case of hierarchical relationships, the number of levels of the hierarchy is an example of the depth problem. Also, relationships such as nested loops tend to be more complex. Recursive relationships are a special kind of nested loop that requires extra attention to design and to test. In developing solutions to these complex problems, software engineers must design with possibly yet another set of relationships different from that of the problem. **FIGURE 2.1** shows the effect of introducing both (1) the size in terms of breadth and (2) the complexity in terms of depth and number of interactions. Although you can get a natural "feel" of the difference by just viewing the diagram, it would be worth taking the time to analyze the differences. The simple case in this figure has three major segments: (1) start process, (2) perform three normal tasks, and (3) stop process. In Figure 2.1(b), the number of normal tasks has increased from three to five with the addition of the "wait for signal" and "perform task A2." There is also a new decision task, represented by the diamond-shaped figure in the center. The decision task has greatly increased the number of paths or choices, and thus it causes the increase of complexity. In addition, the complexity is further exacerbated by introducing a loop relationship with the decision task. There are many more interactions involved in a loop

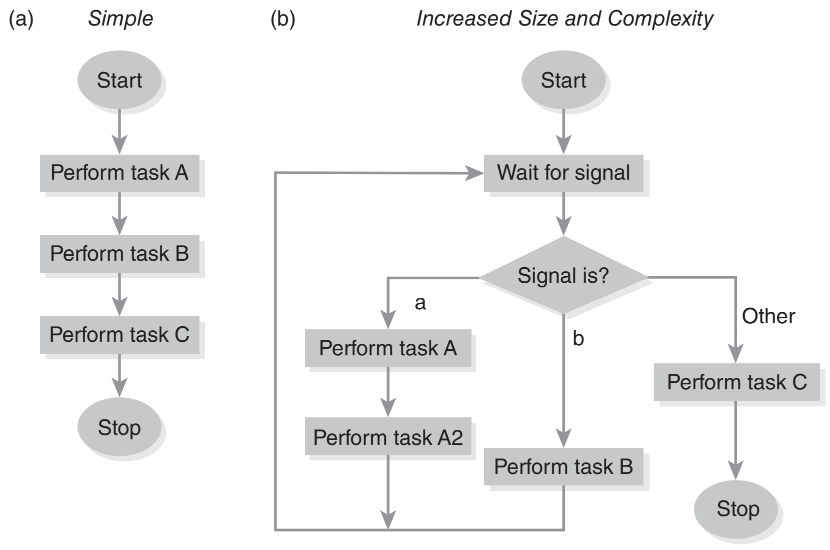


FIGURE 2.1 Size and complexity (a) Simple (b) Increased size and complexity.

or repeat relationship, which is more complex than the straight sequential relationship among the tasks portrayed in Figure 2.1(a).

As shown in Figure 2.1, a relatively minor increase in the number of tasks and decisions has greatly increased the complexity. As is the case for a single programming module, when both size and complexity are magnified several times in a software system problem, the solution to those problems also involves a comparable expansion in size and complexity. Software engineers are not only concerned with the detailed design but must take into account of the complexity impact to the overall architecture, coding, testing, customer deployment, subsequent customer support, and future extensions/modifications.

2.1.2 Technical Considerations of Development and Support

In the following three sections we will discuss a variety of technical issues related to developing and supporting a system.

Problem and Design Decomposition

When we move from a simple to a complex situation of building software systems, there are some technical issues that we must consider. The basic issue is how to handle all the pieces, parts, and relationships. One common solution is based on the concept of divide and conquer. This has its roots in the modularization concept first presented by Parnas (1972). Modularization will be further discussed in Chapter 7. The natural question—how we divide a large, complex problem and its solution into smaller parts—is more difficult than it sounds. We first need to simplify the large, complex problem by addressing the problem in smaller segments. After we have successfully completed that process, our next step is to decide whether we should design and decompose the software system solution along the dividing lines of the problem segments. Thus, if the problem description, or the requirement, is segmented by function and feature, should we design the solution along the same function and feature segments? Alternatively, should we choose another decomposition method for the solution, perhaps along the line of "objects"? (Further discussions on objects can be found in Chapter 7.) The key to attacking large and complex problems is to consider some form of simplification through the following types of activities:

- Decomposition
- Modularization
- Separation
- Incremental iterations

This notion is further expanded in Section 2.2.2, which discusses the design of a payroll system.

Technology and Tool Considerations

Aside from the important issue of decomposing a problem and its solution, there are problems related to technology and tool considerations that will also need to be addressed. If you are not writing a program alone for a limited set of users, the choice of the specific programming language may become an issue. A large, complex system requires more than one person to develop the software solution. Although all the developers involved may know several languages, each individual usually comes with different experience. This diversity in background and experience often results in personal biases for or against a certain programming language and the choice of development tool. A common development language and development environment needs to be picked. Beyond the programming language and the development tools, there are further considerations of other technical choices related to the following:

- Database
- Network
- Middleware
- Other technical components such as code version control

These must be agreed upon by all parties involved in building and supporting a complex software system.

Process and Methodology

We alluded to methodology and process earlier when we discussed the need for simplification and decomposition. When there is only a single person developing the solution, there is still a need to understand the problem or requirements. There is often a need to take the time to put together or design the solution and then implement it. The testing of the solution may be performed by the same person and, possibly, with a user. Under such conditions, there is very little communication among people. No material, such as a design document, is passed from the author to another person.

There still may be a need to document the work performed because even a single developer forgets some of the rationale behind the decisions made. There is usually no need to coordinate the work items because there may not be that many parts. The specific methodology used in performing any of the tasks does not need to be coordinated when only one or two people are involved.

In a large, complex development situation, the problem is decomposed and worked on by many different experts. A **software development process** is needed to guide and coordinate the group of people. Simple items, such as the syntax for the expression of a design, need agreement among all the developers so that they can all review, understand, author, and produce a consistent and cohesive design. Each method used for a specific task along with the entire development process must be agreed to by the group of people involved in the project. Software development and support processes were invented to coordinate and manage complex projects involving many people. The process is greatly facilitated when a group of people can be converted into a cooperating team of individuals. Although continuous improvements and new proposals are constantly being made, no one has yet proposed the complete elimination of process or methodology. Regardless of what is believed about software processes, it is commonly accepted that some process must exist to help coordinate a complex and successful software project. Traditional software process models and emerging process models, including the currently popular Agile methods, will be discussed in Chapters 4 and 5.

Software development process
The set of tasks, the sequence and flow of these tasks, the inputs to and the outputs from the tasks, and the preconditions and postconditions for each of the tasks involved in the production of a software.

Consider the simple scenario of depicting the six major tasks shown in **FIGURE 2.2**. These are the common tasks often performed in software development and support. Each task appears as an independent item, and each one begs the questions of what is expected and how we perform it. For example, is there a methodology to gathering requirements? If there is more than one person performing the requirements-gathering task, how that task should be broken down needs to be defined. Similarly, we might ask what constitutes user support and what problems must be fixed.

The tasks in Figure 2.2 are displayed independently. When several individuals are involved in software development and support, there has to be a clear understanding of the sequence, overlap,

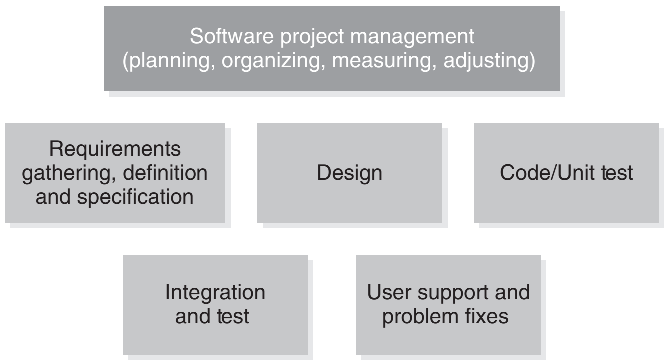

Software project management
(planning, organizing, measuring, adjusting)
Requirements gathering, definition and specification
Design
Code/Unit test
Integration and test
User support and problem fixes

FIGURE 2.2 Independent tasks.

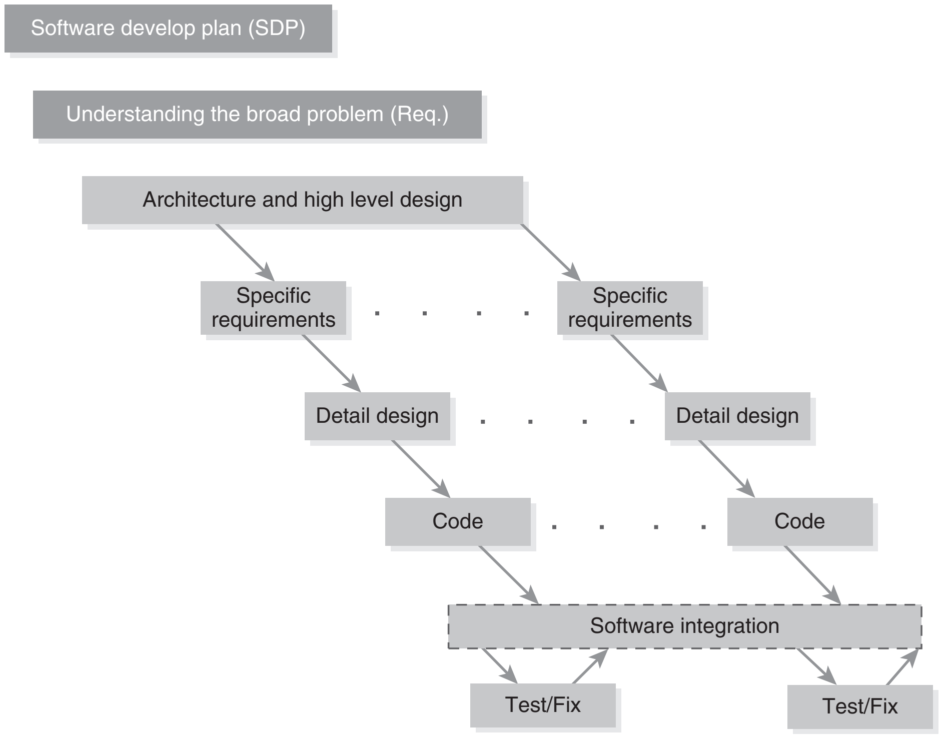


FIGURE 2.3 One possible process approach.

and starting conditions. For example, the designers and coders may be one group of people that is different from the requirements analysts who are working with the customers. At what point should the designers and coders start their tasks? How much can these tasks overlap? How should the completed code be integrated and tested? The process definition should answer these questions and help in coordinating the various tasks and in ensuring that they are carried out according to previously agreed on methodologies.

FIGURE 2.3 represents one approach that employs the concept of incremental development and continuous integration. Software integration is the process of linking together individually tested units into a coordinated whole system. Continuous integration has been practiced since the 1970s, when large systems were first being built (Tsui and Priven 1976). Recently, because of the widespread use of incremental development and Agile methodologies, continuous integration is gaining general popularity. The currently popular term CI/CD, **continuous integration/continuous deployment** (Pittet 2017), has expanded the process and now includes the tasks of continuously (1) integrating the completed functionality, (2) delivering that feature, and (3) having the users deploying that feature. The methodologies involved in incremental development must all be agreed to and practiced by the

Continuous integration/continuous deployment (CI/CD) The extension of incremental software development process to include the quick deployment of a completed functionality via (1) continuously integrating completed functionalities into the product, (2) delivering those functionalities to the users, and (3) having the users quickly deploy those functionalities.

entire development team. The seemingly simple boxes depicting the test–fix–integrate cycle in Figure 2.3 are extremely deceptive. That simple cycle requires a description of a methodology that answers the following questions:

- Is there a separate and independent test group?
- When a problem is found, how and to whom should it be reported?
- How much information must accompany a problem description?
- Who decides the severity level of a problem?
- How is a problem-fix returned to the tester?
- Should all problem-fixes be retested?
- How are the problem-fixes integrated back into the code?
- What should be done with the problems that are not fixed?

These are just some of the questions that must be determined and worked out for a portion of the process depicted in Figure 2.3. The process also assumes that incremental development is used and that both the problem and the design can be decomposed into increments. Figure 2.3 does not include the support and customer problem-fix activities. We must not forget that software products need usage support, problem-fixes, and enhancements. Process plays a vital part in defining and coordinating the activities for large, complex systems development and support. We will expand on the specifics relating to testing and integration methodologies and process in Chapters 10 and 11.

In Chapter 4, we will show how the two figures 4.3 and 4.4, which demonstrate the variation and growth of incremental development process, formulated the precursor to the current CI/CD. The notion of CI/CD will also be brought up in the discussion of Agile process and Kanban methodology in Chapter 5.

2.1.3 Nontechnical Considerations of Development and Support

In addition to technical implications, large and complex systems also require a cognizance of nontechnical issues. We will discuss two such issues here.

Effort Estimation and Schedule

For a small and fairly simple software project that involves a team of one to three people, the effort estimation and scheduling of the project is relatively easy. Both the functional and nonfunctional requirements of the project are fewer in number and complexity. Even then it is still not a trivial task. For complex and large systems, capturing and understanding the requirements alone can be overwhelming. Estimating the total effort and coming up with a reliable project schedule under this difficult condition is one of the main reasons behind so many software project failures; see Jorgensen (2004) for more details. The inaccurate effort estimates and schedules for large, complex systems are often extremely optimistic and aggressive; this places unrealistic expectations on both the customers and the suppliers of these systems.

As an example, consider a relatively simple software project that requires three major functions with a total of twelve features. The effort estimation of this project requires a good understanding of all the functional features and the productivity of the individuals in the small team who will be working on these twelve features. For a large, complex software system, the number of major functions is often in the tens or hundreds. The total number of features within these major functions may easily be in the hundreds and thousands. The number of people needed to develop such a system may easily be in the hundreds. Under such circumstances, the probability of understanding all the requirements well and of knowing the productivity of all the individuals accurately is very low. The sorting of the number of combinations of individuals assigned to the design and coding of such a large number of features alone can be a daunting task. The resulting effort estimation and the schedule is often a good "guess" and far from being accurate. The software industry has long recognized this problem and has been confronting this issue. In Chapter 13 we will address some of the techniques that have been developed and are now available.

Assignments and Communications

We touched on the problem of assigning people to designing and coding the different functional features when the number of features increases and the corresponding number of developers increases. Furthermore, there are other activities that require people resources. The assignment of different people to different tasks such as testing, integration, or tool support requires more understanding of the skills of the people involved and the specific tasks they have to perform. The assignment of the most effective and properly skilled people to the right tasks requires a deeper level of granularity and a finer level of scheduling.

Another related problem with the increase in personnel is the problem of communications. For a small project involving two or three people, the number of communications paths is one between two people and three among three people. **FIGURE 2.4** illustrates how maximum communication paths increase as the number of participants increases. The nodes in this figure represent the people,

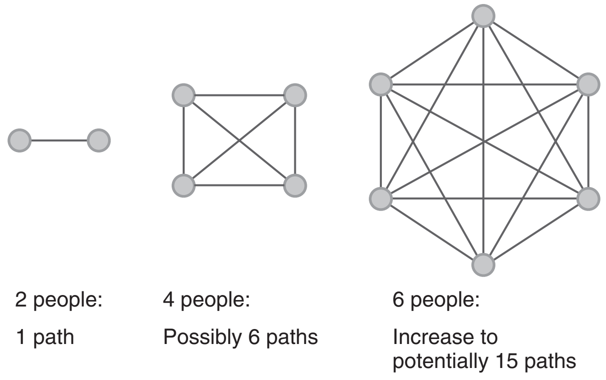


FIGURE 2.4 Maximizing communication paths.

and the lines represent the communication paths. The number of possible communications paths more than doubled when the number of team members increased from four to six.

In general, the number of communication paths for *n* people is SUM(n–1), where SUM is the arithmetic sum function of 1, 2, . . ., n (notice this is very close to $n^2/2$). Thus a modest increase from a four-person team to a twelve-person team would increase the potential number of communication paths from 6 to 66. A tripling of a small team would increase the potential communication paths by more than ten times!

Associated with this increase in the sheer number of communication paths is the chance of an error in communications. Consider, for example, that the chance of communicating correctly a particular message between any two people is 2/3. The probability that we will communicate properly from one person to another and then from that second person to a third person would be $(2/3 \times 2/3) = 4/9$. In general, for *n* people where *n* is 2 or more, the probability of correctly communicating this message would be $(2/3)^{n-1}$. Thus for this message, there is only a 16/81 chance of correctly passing it from the first person to the fifth person in the team. Suddenly, we have reduced a 2/3 chance of correctly communicating a message to less than 1/4. Such a low probability of correct communication among team members may be a serious problem, especially if the message is critical. Organizational structures of people need to be put in place to reduce the complexity and increase the chance of correct communications.

2.2 Building a Hypothetical System

In this section we will use a hypothetical payroll system to illustrate some of the problems introduced in Section 2.1. The discussion here will cover the major tasks of developing such a system and of supporting the system once it is released to users. The intent of this section is to provide only a glimpse of the different problems and concerns that arise in building our system but not to delve into all the details of constructing and supporting this system.

2.2.1 Requirements of the Payroll System

Everyone has some idea of what a payroll system is. Take a moment to think about what you would consider as the major functional and the nonfunctional requirements of a payroll system. The following functional capabilities represent only some of the tasks a payroll system should be able to perform. This list is far short of what a real payroll system would need.

- Add, modify, and delete the names and associated personal information of all employees.
- Add, modify, and delete all the benefits associated with all employees.
- Add, modify, and delete all the tax and other deductions associated with all employees.
- Add, modify, and delete all the gross income associated with all employees.
- Add, modify, and delete all the algorithms related to computing the net pay for each employee.
- Generate a paper check or an electronic direct bank deposit for each employee.

Each of these functional requirements may be expanded to several levels of more details. For example, just for the first item of names and associated personal information, one would need to understand what the associated information is. This is a simple question, but would require the software engineer to solicit the input for this. Where and who would the software engineer ask? Should the software engineer ask the users, some designated official requirements person, or the project leader? Once the question is answered, should the answer be documented? The next functional requirement in the preceding list speaks to all the benefits. What are all the benefits? What does having a benefit mean to an employee's payroll? Is there a list of all possible benefits? It does not take much to realize that the requirements solicitation, gathering, documentation, analysis, and validation of a payroll system will need a considerable effort. In order to properly handle the application side of the payroll system requirement, we may need to understand something about benefits, tax laws, and other domain-specific knowledge.

In addition, the payroll system must be able to generate the paychecks and direct deposits several times a month. What is the allowable payroll cycle? In other words, if the checks and deposits must be completed by the middle and end of the month, when must the inputs to the cycle, such as salary increase, be closed? Here we are interested in understanding the payroll-processing-cycle window that is allowable within the business environment and what performance capability the system must have to satisfy that processing window. This involves the nonfunctional requirement—performance. The answer to this question will require the software engineer to know the volume of payroll transactions and the speed of processing each payroll transaction. To analyze and handle this type of requirement, we may need to know the hardware and operating system environment capability on top of which the payroll system will be running. Some of the payroll system requirements will require, in addition to payroll domain knowledge, the knowledge of technical system and interface information.

There also needs to be an understanding of how the actual payroll run process works at the user/customer site. For example, if there is a bad record, how should that person's paycheck be reprocessed? Does this imply that there is a requirement to rerun the payroll system several times? The nonfunctional requirement of *security* should be addressed. What protection in the face of possible error, malice, or mischance is needed? There may also be some requirements that the users/customers may not even remember to provide initially. In Chapter 6 we discuss how we may handle these late requirements.

Once the requirements information of a payroll system is documented, the complexity of such a system will most likely necessitate a review with the users/customers before having the requirements specification passed forward to the design and coding phase. These reviews may be conducted gradually as the requirements are incrementally analyzed and documented or all at once when all the requirements are analyzed and specified together. Either situation will require a coordination of effort between the users/customers and the requirements analysts.

It is thus clear that the total number of activities needed to complete a payroll system requirement phase alone may be extremely high as well as time consuming. The requirement phase is critical for the system's success. Not just a single requirements analyst but a team of requirements analysts—individuals having diverse skills spanning everything from payroll domain-specific knowledge to IT and systems development expertise—may be needed. From a quality perspective, it has also been pointed out by Jones (1992) that approximately 15 percent of software defects are due to requirements errors. The activities related to completing a requirements specification for a system, such as that in our payroll example, are difficult and have significant impact on all downstream activities and on the final product. Complete books have been written on just this topic. (See the References and Suggested Readings section at the end of this chapter.)

2.2.2 Designing the Payroll System

Once the requirements of the payroll system are understood and agreed to, the system must still be designed. Put aside the fact that the payroll system requirements expressed in Section 2.2.1 are just an example and are incomplete. For example, we might naturally ask whether all the "add, update, and delete" functional requirements should be grouped together into a single component called "payroll administrative functions." We might then ask if all the processing functions such as the calculations of all the deductions and the net pay amount should be grouped together into a component called "payroll processing." Certainly, we must be prepared to handle errors and exceptions. So, those functions dealing with errors and exceptions processing may be aggregated into an exceptions-processing component. In addition, the payroll system must interface with external systems such as direct bank deposits or batch transmissions to remote sites for local check printing. We may decide to place all the interface functions into a component called "payroll interfaces." This grouping of related functions into components has several advantages:

- Provides some design cohesiveness within the component
- Matches the business flow and the payroll processing environment
- Provides a potential assignment of work by components
- Allows easier packaging of the software by components

There may be some drawbacks to this approach. It is conceivable that there are still heavy interactions or coupling among these components. The coupling, in this case, may arise from extensive usage of a common data file or a common table in a relational database. Even at this high level, designers need to look at both the characteristics of design cohesiveness and coupling. The concepts related to these topics are discussed extensively in Chapter 8.

There are also nonfunctional specific, but common-service, needs that must be designed. For example, the help service or the message service must be designed for all the functional components. These services may all be placed in one component called the *services component*. The combination of functional components and common services is shown in **FIGURE 2.5** as horizontal and vertical design entities. The horizontal entities are the common service functions such as the error handler that crosses all the individual application features. The vertical entities are the different application domain-specific functions such as the tax and benefits deduction function in a payroll system. The interaction, or coupling, of the various functional components with these common services is a key design concern.

It is during design that the screen interface layout is finalized. In the case of a payroll system, this is a heavily batch-oriented system rather than an interactive system. User interface in terms of screen architecture is thus not a prime design concern. Nevertheless, it needs to be addressed. The database tables and search keys, however, are important and would be a significant design concern for a large batch-processing application.

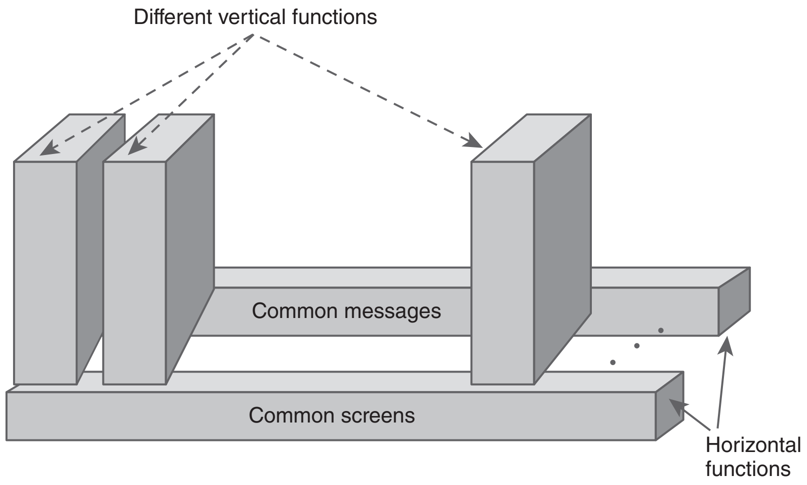


FIGURE 2.5 Vertical and horizontal design entities.

Although there are many ways to perform high-level and detailed design, the design of a payroll system requires a broad set of skills because of the breadth and depth of the system. From the breadth perspective, the design skills needed require complex knowledge: database, network, and transmission interfaces; printing interfaces; operating system interfaces; development tools environment; and the payroll application domain. From a depth perspective, the designer needs to understand and appreciate the specifics of a payroll system, such as performance and error processing. Although it is a batch-processing system, the sheer volume of payroll records for large enterprises often requires special design concerns that would make the seemingly simple process of error handling into a complex task. The design must not only catch erroneous information or conflicting information but must also consider what should happen to the people whose records cannot be processed. If these records are not dealt with immediately and allowed to accumulate until the end of the payroll cycle, there is no time to react. These records must be handled so that they can be converted to a paycheck within the payroll processing cycle. The designer must consider the payroll environment and the possibility of having to hand code the paychecks for a small number of unfortunate people. Thus the designer must design the system to include hand-processing exits from the system and the reconciliation of these hand-processed records back into the automated payroll system. The depth of error processing in a large system such as payroll can be a challenge for even the most experienced designers.

The payroll design mentioned here uses functional decomposition and synthesis techniques within each of the components. In addition to the intercomponent interactions, the various pieces within a component must be clearly divided and the intracomponent interaction between the pieces must also be designed. Clearly, designing a complex system is quite different from designing a single programming module and will require greater discipline and additional guiding principles as well as the possibility of several more team members.

2.2.3 Code and Unit Testing the Payroll System

The high-level design or architecture for the payroll system needs to be further refined and converted into running code. Within each design component, the individual, interacting, functional unit needs to be designed and converted into code. This activity is familiar to most people who enter the IT and computing field. The first course often taught to students entering software engineering or computer science involves a small problem that must be solved with a detailed functional design and code. At times, when the solution is small enough, the detail design is not even recorded and only the source code of the module is available.

For each of the functional units, the programmer must address and develop the following material:

- Precise layout of the screen interface in some language
- Precise functional processing logic in some programming language
- Precise data access and storage logic in some language
- Precise interface logic in some language

Furthermore, if there are many of these programming units, some common standards must be set. An example would be a naming convention for each of the modules that would uniquely identify each as the module of a specific component. Conventions may also need to be set for different database records such that all elements from a specific relational table have the same prefix. There may be conventions set to document some of the detail design such as providing comments on the conditions under which this module may be entered and exited. The comments may also describe the data that are vital to the processing and a short description of the intended function. A very important part is the design, code, and the documentation of how to handle the various error conditions. The error messages displayed from the different program modules need to be consistent; thus each program unit must follow the error message standard.

After the program module has been completed, the individual who performed the task should test the module to confirm that it performs the intended tasks. The first step in this unit-testing task is to set the conditions of the module and to choose the appropriate input data. The next step is to execute or run the module and observe the behavior of the module, mainly through checking the output from the module, to ensure that it is performing what it is intended to do. Finally, if there is any problem discovered through unit testing, it must be fixed and retested. When all the problems are fixed, the module is ready for integration into a larger unit such as a functional unit or into a component if the module itself is a functional unit.

The programming or coding and unit testing of a module is usually performed by one individual. For a large system, such as a payroll system, there may be hundreds of modules that need to be coded and unit tested. Thus, programming is a heavily human resource intensive activity. When the number of programmers increases to a large figure, then the coordination and integration of all the programming efforts become a management challenge. Once again, principles of software engineering management need to be brought in to alleviate the situation.

2.2.4 Integration and Functional Testing of the Payroll System

As the modules are completed and unit tested, they have to be formally collected from the individual programmers. The collection activity is known as *integration*, which is a part of a larger control mechanism known as *configuration management*. Configuration management will be mentioned throughout this text but will be formally discussed in Chapter 11. A simple reason for the integration step is that if the completed modules are left with the individual programmers, the programmers tend to make changes to an already unit-tested module and get confused about which is indeed the latest version. To ensure that the latest unit-tested modules do work together as a functional unit, these modules need to be compiled and linked together. A functional unit, in the case of the payroll system, may be a part of the previously mentioned administrative component that performs the add, modify, and delete functions of all the federal deduction laws, which almost always change annually. The integrated set of modules is then tested with functional test cases generated by a more objective group than the programmers who coded the modules.

Functional testing usually uncovers some problems that will require fixing by the programmers. The cycle of problem detection to problem-fix needs to be coordinated between the testers and the code fixers. The fix code must be integrated into the functional unit and be retested to ensure that all the fixes as a group have not impacted each other negatively. As a set of modules in a functional unit completes the functional test, it is electronically labeled as such and is locked from further changes. These functional units need to be managed by the configuration management mechanism as do the module units. In the case of a simple one or two module situation, there is not much need for an integration and configuration management mechanism. In a very large software system construction such as a payroll system, there is usually a tool, such as PVCS from Serena Software, used to help automate the configuration management mechanism. An additional reason of needing sophisticated tools today is the aforementioned CI/CD process with which we are releasing incremental functionalities to the users at a faster speed. The people and skills required to tackle the integration and functional testing of a payroll system are usually different from those needed for coding, designing, or requirements gathering. However, test scenarios and test scripts often require the knowledge of the requirements and the design. Various integration and configuration management concept and tools will be discussed in Chapter 11.

2.2.5 Release of the Payroll System

After the functional units are tested and integrated into components, these components must be tested together to ensure that the complete system works as a whole. This is important to ensure that all the interfaces across components actually do work. Also, the various fixes for the functional units and components may impact some other previously working functional units and components. Even after the entire payroll system is tested through all the user scenarios in the context of the user business environment, the system cannot be released unless no problem was found. At least all the major problems and showstoppers must be fixed before the system can be considered releasable to the users. Once again, the tested payroll system must be managed and protected from further changes.

Even if the payroll system is totally error free, the users must still be educated in the usage of the system—a process that for a large system cannot be an afterthought and must be planned and orchestrated. The development of the educational material alone for such a system is a nontrivial task. The effort may take several people and several months. The delivery of user education may require some different skills from technical design or coding. The emphasis would be presentation and communication skills. The people who develop the educational material content may be different from those who deliver the education.

Another area of preparation before releasing the payroll system would be the preparation of user support personnel. It would be rare that the users can master all the details of a payroll system just through education. Furthermore, it is also rare that a large and complex system will be totally error free. The support personnel themselves must be educated on the payroll system, user environments, and tools needed for supporting customers.

Once the system test is complete, the users have been trained, and the support group is trained and established, the payroll system is then ready for release to the users. Who should be the person who makes the final call of a product release? Should this be a group decision? And what criteria should that person use in making the determination for release? These topics fall under the umbrella of software project management, discussed further in Chapter 13.

2.2.6 Support and Maintenance

For a small, one- or two-module software product that is used by a few people, the support effort is not a major concern. For a large system such as payroll, the postrelease support of the users and customers may be a very complex set of tasks. Who does the user call for help, after consulting the user manual, when the payroll system stalls and pops up a message with several possible choices for the user before the payroll system can continue processing? Who does the user call when the direct deposit interface on the bank side has changed and the existing payroll system interface needs to be modified? Who does the user call when the payroll system shows a different behavioral problem after applying a previous problem-fix? These are just a few of the many questions that will arise after the payroll system is released. Several assumptions must be made and be included in the calculation of the expected payroll system support effort. Many of the following decision factors will play a role:

- Number of expected users and customers
- Number and type of known problems that existed at release time
- Projected number of problems that will be discovered by users
- Amount of user training
- Amount of support personnel training
- Number of development personnel committed to support the system
- Expected number of problem-fix releases and future functional releases