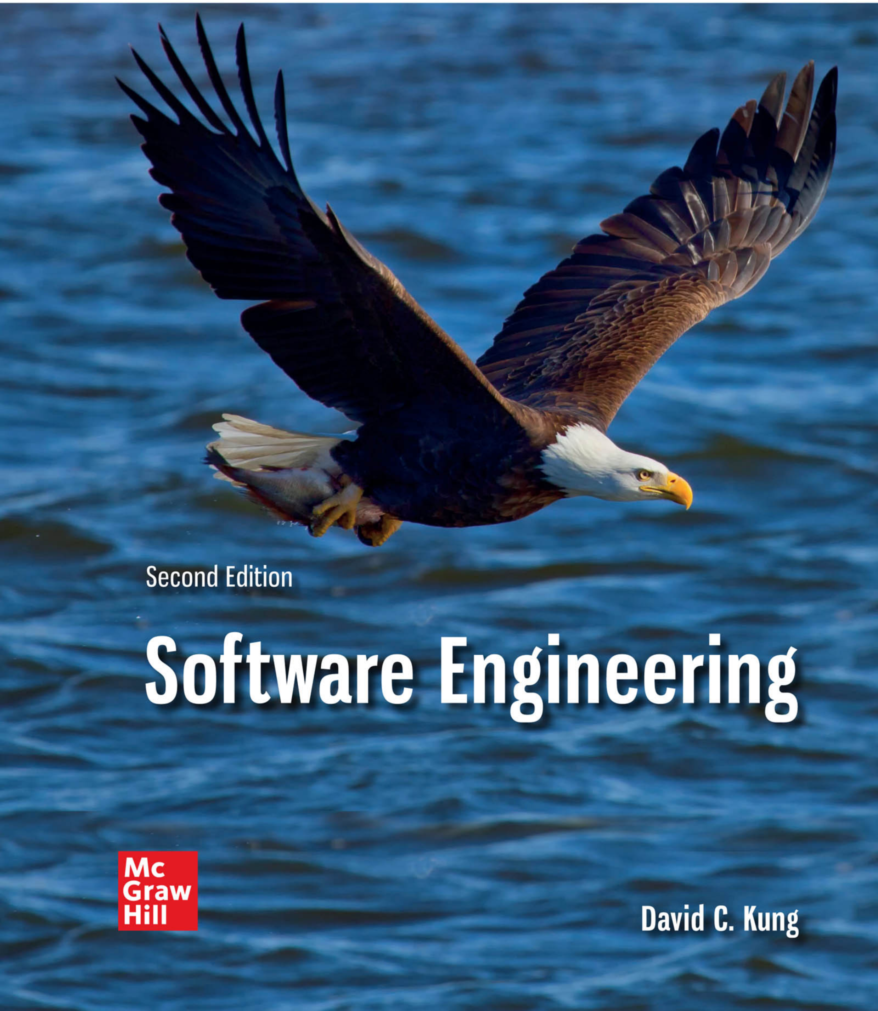

Second Edition

Software Engineering

McGraw Hill

David C. Kung

Software Engineering

Second Edition

David C. Kung

Mc
Graw
Hill

SOFTWARE ENGINEERING

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 28 27 26 25 24 23

ISBN 978-1-265-24243-5
MHID 1-265-24243-7

Cover Image: *Shutterstock Images, LLC*

mheducation.com/highered

Dedication

To My Father

Brief Contents

Contents

Preface

BACKGROUND

Computers are widely used in all sectors of our society, performing a variety of functions with the application software running on them. As a result, the market for software engineers is booming. There is a significant gap between the demand and supply, especially for graduates with software engineering education.

Many people do not know the scope and usefulness of software engineering, and the discipline is often misunderstood. Many media outlets deem software engineering as writing Java programs. Some students think that software engineering includes everything related to software. Others think that software engineering is drawing UML diagrams, as the following story illustrates. Years ago, after the first class of a software engineering course, a student told me, "professor, you know that this will be an easy course for me because we've drawn lots of UML diagrams before." At the end of the semester, the student came to me again and said, "professor, I want to tell you that we worked very hard, but we learned a lot about OO design. It is not just drawing UML diagrams." So what is software engineering? As a discipline, it encompasses research, education, and application of engineering processes, methodologies, quality assurance, and project management to significantly increase software productivity and software quality while reducing software cost and time to market. A software process describes the phases and *what* should be done in each phase. It does not specify (in detail) *how* to perform the activities in each phase. A modeling language, such as UML, defines the notations, syntax, and semantics for communicating and documenting analysis and design ideas. UML and the Unified Process (UP) are good and necessary but not sufficient. This is because *how* to produce the analysis and design ideas required to draw meaningful UML diagrams is missing.

MOTIVATION

To fill the gap mentioned above, we need a methodology or a "cook-book." Unlike a process, a methodology is a detailed description of the steps and procedures or *how* to carry out the activities to the extent that *a beginner can follow* to produce and deploy the desired software system. Without a methodology, a beginning software engineer would have to spend years of on-the-job training to learn design, implementation, and testing skills.

This book is also motivated by emerging interests in *agile processes, design patterns,* and *test-driven development* (TDD). Agile processes emphasize teamwork, design for change, rapid deployment of small increments of the software system, and joint development with the customer and users. Design patterns are effective design solutions to common design problems. They promote software reuse and improve team communication. Patterns also empower less-experienced software engineers to produce high-quality software because patterns encode software design principles. TDD advocates testable software, and requires test scripts to be produced before the implementation so that the latter can be tested immediately and frequently.

As an analogy, consider the development of an amusement park. The overall process includes the following phases: planning, public approval, analysis and design, financing, construction drawings, construction, procurement of equipment, installation of equipment, preopening, and grand opening. However, knowing the overall process is not enough. The development team must know how to perform the activities of the phases. For example, the planning activities include development of initial concept, feasibility study, and master plan generation. The theme park team must know how to perform these activities. The analysis and design activities include "requirements acquisition" from stakeholders, site investigation, design of park layout, design of theming for different areas of the park, creating models to study the layout design and theming, and producing the master design. Again, the theme park team must know how to perform these activities to produce the master design. Unlike a process that describes the phases of activities, a methodology details the steps and procedures or how to perform the activities.

The development of an amusement park is a multiyear project and costs billions of dollars. The investor wants the park to generate revenue as early as possible, but with the above process, the investor has to wait until the entire park is completed. Once the master design is finalized, it cannot be modified easily due to the restrictions imposed by the conventional process. If the park does not meet the expectations of the stakeholders, then changes are costly once the park is completed.

Agile processes are aimed to solve these problems. With an agile process, a list of preliminary theme park requirements is acquired quickly and allowed to evolve during the development process. The amusement and entertainment facilities are then derived from the requirements and carefully grouped into clusters of facilities. A plan to develop and deploy the clusters in relatively short periods of time is produced, that is, rapid deployment of small increments. Thus, instead of a finalized master design, the development process designs and deploys one cluster at a time. As the clusters of facilities are deployed and operational, feedback is sought and changes to the requirements, the development plan, budget, and schedule are worked out with the stakeholders—that is, joint development. In addition, the application of architectural design patterns improves quality and ability of the park to adapt to changing needs— that is, design for change. Teamwork is emphasized because effective collaboration and coordination between the teams and team members ensure that the facilities will be developed and deployed timely and seamlessly. The agile process has a number of merits. The investor can reap the benefits much earlier because the facilities are operational as early as desired and feasible. Since a small number of the facilities are developed and deployed at a time, errors can be corrected and changes can be made more easily.

In summary, this text is centered around an agile unified methodology that integrates UML, design patterns, and TDD, among others. The methodology presented in this book is called a "unified methodology" because it uses UML as the modeling language and it follows an agile unified process. It does not mean to unify any other

AUDIENCES

This book is for students majoring in computer science, software engineering or information systems, as well as software development professionals. In particular, it is intended to be used as the primary material for upper-division undergraduate and introductory graduate courses and professional training courses in the software industry. This book's material evolved over the last two decades from courses taught at universities and companies domestically and internationally, as well as from applications of the material to industry-sponsored projects and projects conducted by software engineers in various companies. These allowed the author to observe how students and software engineers applied UP, UML, design patterns, and TDD, and the difficulties they faced. Their feedback led to the development of the Agile Unified Methodology (AUM) presented in this book and the continual improvement of the material.

The book describes AUM in detail to facilitate students to learn and develop analysis and design abilities. In particular, each analysis or design activity is decomposed into a number of steps, and how to perform each step is described in detail. This treatment is intended to facilitate students learning how to perform analysis and design. Once acquired the abilities, one may skip some or most of the steps.

ORGANIZATION

The book has 24 chapters, divided into eight parts:

Part I. Introduction and System Engineering. This part consists of the first three chapters. It provides an overview of the software life-cycle activities. In particular, it covers software process models, the notion of a methodology, the difference between a process and a methodology, and system engineering.

Part II. Analysis and Architectural Design. This part presents the planning phase activities. It includes requirements elicitation, domain modeling, and architectural design.

Part III. Modeling and Design of Interactive Systems. This part deals with the modeling and design of interactive systems. It consists of six chapters. These chapters present how to identify use cases from the requirements, how to model and design actor–system interaction and object interaction behavior, how to apply responsibility assignment patterns, how to derive a design class diagram to serve as the design blueprint, and how to design the user interface.

Part IV. Modeling and Design of Other Types of Systems. This part consists of three chapters; each presents the modeling and design of one type of system. In particular, Chapter 13 presents the modeling and design of event-driven systems. Chapter 14 presents the modeling and design of transformational systems. Chapter 15 presents the modeling and design of business rule-based systems.

Part V. Applying Situation-Specific Patterns. This part consists of two chapters and presents how to apply situation-specific patterns. A case study, that is, the design of a state diagram editor, is used to help understand the process.

Part VI. Implementation and Quality Assurance. This part consists of three chapters. They present implementation considerations, software quality assurance concepts and activities, and software testing.

Part VII. Maintenance and Configuration Management. This part includes two chapters and covers software maintenance and software configuration management.

Part VIII. Project Management and Software Security. The last part of the book consists of the last two chapters. One of the chapters presents software project management. The other chapter covers software security, that is, life-cycle activities concerning the modeling and design of secure software systems.

The material can satisfy the needs of several software engineering courses. For example,

1. Part I through Part III and selected topics from Part VI to Part VIII are a good combination for an Object-Oriented Software Engineering (OOSE) course or an Introduction to Software Engineering course. This could be a junior- or senior-level undergraduate course as well as an introductory graduate-level course.
2. Part II, Part V, and selected sections from the other chapters could form a Software Design Patterns course. It is recommended that the OOSE course described above be a prerequisite for this course. However, many international students may not have taken the OOSE course. In this case, a review of the methodology presented in Part II and Part III is recommended. The review of the methodology provides the framework for applying patterns. The review may take two to four weeks.
3. Part VI and Part VII could be taught in various ways. They could form one course—Quality Assurance, Testing, and Maintenance. They could be taught as two courses—Software Quality Assurance, and Software Testing and Maintenance.
4. Chapters 13–15, 19, and 20 plus selected patterns from the other chapters may form a course on modeling, design, verification, and validation of complex systems.
5. Part I, Parts VI–VIII, and selected chapters from the other parts may form a Software Project Management course.

Various teaching supplements can be found at http://www.mhhe.com/kung. These include PowerPoint teaching slides, pop quiz and test generation software, databases of test questions, sample course descriptions, syllabi, and a solution manual. Instructors who have not taught the courses may find these helpful in reducing preparation time and effort.

ACKNOWLEDGMENTS

I would like to thank my numerous students who constantly stimulate me with their questions, feedback, enthusiasm, and effort to apply AUM to real-world projects. They provided valuable improvement suggestions. Many continue to practice AUM in industry after graduation and share with me their valuable experiences. I also want to thank the reviewers and numerous instructors for their comments and suggestions. These have significantly improved the organization, presentation, and many other aspects of the book. I am thankful for the opportunities to teach on-site training courses and for-credit courses for various companies. These allow me to interact with many software developers and learn from them. Some companies let the developers apply AUM and patterns to in-house or product-development projects. All of these projects generated very positive feedback including high-quality design documentation and drastic reduction in defect rates. I want to thank the managements of various companies for their constant support to industry-university collaboration and the opportunity for me to learn from practice.

This page intentionally left blank

part I

Introduction and System Engineering

1 Chapter

Introduction

Key Takeaway Points

- Software engineering aims to significantly improve software productivity and software quality while reducing software costs and time to market.
- Software engineering consists of three tracks of interweaving life-cycle activities: software development, software quality assurance, and software project management activities.

Computers are used everywhere in our society. It is difficult to find a hospital, school, retail shop, bank, factory, or any other organization that does not rely on computers. Our cell phones, cars, and televisions are also based on computer-powered platforms. The driving force behind the expanding use of computers is the market economy. However, it is the software that makes the computers work in the ways we want. Software or computer programs consist of thousands or millions of instructions that direct the computer to perform complex calculations and control the operations of hardware devices. The demand for computer software has been rapidly increasing during the last several decades. This trend is expected to continue for the foreseeable future.

The proliferation of computer applications creates a huge demand for application software developers. According to the Bureau of Labor Statistics (BLS), application software developer was one of the 30 fastest-growing occupations in America (bls.gov/emp/tables/fastest-growing-occupations.htm). The number of positions was projected to grow from 1,469,200 in 2019 to 1,789,200 in 2029, an increase of 316,000, or 21.50%. The median annual wage for an application software developer in May 2019 was $110,140, much higher than the median annual wage for all occupations ($41,950). Among the 10 computer and IT occupations surveyed by the BLS, only application software developer and information security analyst enter into the 30 fastest-growing list. Its median pay was also much higher than the median pay of $91,250 for the 10 computer and IT occupations surveyed by the BLS.

There are two popular misconceptions. One equates application software development with computer programming. The other equates an application software developer with a computer programmer. However, according to the BLS, software developers create the applications or systems that run on a computer or another device.

Computer programmers write and test code that allows computer applications and software programs to function properly. The BLS survey also showed that the median pay for a computer programmer in May 2019 was $89,190, which was lower than the median pay for computer and IT occupations and much lower than the median pay for an application software developer.

Unlike a computer programmer, an application software developer is required to identify and formulate feasible and cost-effective solutions to solve large, complex real-world problems and design software to implement such solutions. The solutions and the software must take into account potential impact to public health, safety, security, and welfare as well as cultural, social, and environmental aspects (abet.org). To be able to perform the work required of an application software developer, an education in software engineering is highly desired.

1.1 WHAT IS SOFTWARE ENGINEERING?

Software systems are complex intellectual products. Software development must ensure that the software system meets the needs of the intended application, the budget is not overrun, and the system is delivered on time. To accomplish these goals, the term "software engineering" was proposed at a NATO conference in 1968 to advocate the need for an engineering approach to software production. Since then, software engineering has become a discipline and made remarkable progress. The efforts that take place in the field lead to the following:

Definition 1.1 *Software engineering* as a discipline is focused on the research, education, and practice of engineering processes, methods, and techniques to significantly increase software productivity and software quality while reducing software costs and time to market.

This definition includes several important points. First, the overall objective of software engineering is significantly increasing software productivity (P) and quality (Q) while reducing software production and operating costs (C) as well as time to market (T). These are abbreviated as PQCT in this book. In other words, significantly improving PQCT means producing higher-quality software more quickly, efficiently, and cost-effectively. These will eventually contribute to the improvement of our lives. Second, research, education, and practice of software engineering processes, methods, and techniques are the means to significantly improve PQCT.

Software development involves three tracks of interweaving activities, as Figure 1.1 exhibits. These activities take place simultaneously throughout the software life cycle:

1. Software development activities.
2. Software quality assurance activities.
3. Software project management activities.

Software development activities are a set of activities performed to transform an initial system concept into a software system running in the target environment. Like many engineering projects, software development activities include software

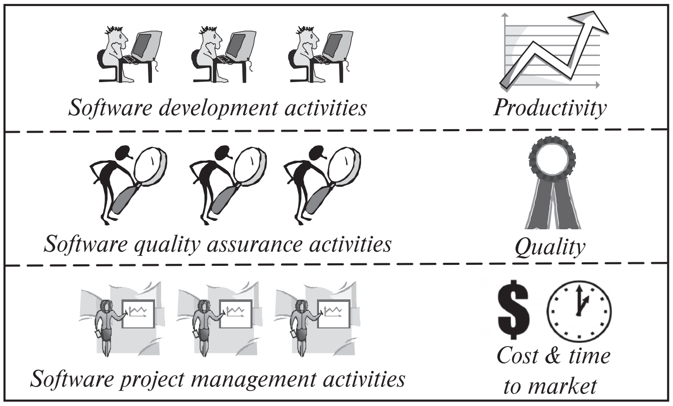
Software development activities
Productivity
Software quality assurance activities
Quality
Software project management activities
Cost & time to market

FIGURE 1.1 Three tracks of life-cycle activities

specification, software design, implementation, testing, deployment, and maintenance. Software specification determines what the customer and users want. These are specified as requirements or capabilities that the software system must deliver. Software design produces a software solution to realize the software requirements. In particular, it determines the overall software structure, called the software architecture, of the software system. The architecture depicts the major system components and how they relate, interface, and interact with each other. Software design also defines the user interfaces as well as high-level algorithms for the system components. During implementation and testing, the design is converted into computer programs, which are tested to ensure that they work as the customer and users expect. The software system is then installed in the target environment, tested and modified to ensure that it works properly. During the maintenance phase, the software system is continually modified to correct errors and enhance functionality until it is abandoned or replaced.

Software quality assurance (QA) activities are carried out alongside the development activities. QA activities ensure that the development activities are carried out correctly; the required artifacts, such as software requirements document (SRD) and software design document (SDD), are produced and conform to quality standards; and the software system will fulfill the requirements. These are accomplished through requirements review, design review, code review and inspection, as well as testing.

Software project management activities ensure that the software system under development will be delivered on time and within budget constraint. One important activity of project management is project planning. It takes place at the beginning of a project, immediately after the requirements for the software system are determined. In particular, effort and time required to perform the three tracks of activities for the project are estimated. A schedule of activities is produced to guide the project. During the development and deployment process, project management is responsible for continuous monitoring of project progress and costs, and executing necessary actions to adapt the project to emerging situations.

1.2 WHY SOFTWARE ENGINEERING?

First, software is used in all sectors of our society. Companies rely on software to run and expand their businesses. Airplanes, vehicles, medical equipment, and numerous other machines and devices rely on software to operate. Internet of Things (IoT), cloud computing, and AI applications also heavily rely on software. Software systems are getting much larger, extremely complex, and highly distributed. Today, it is common to develop systems that contain millions of lines of code. For example, the F35 fighter runs on 8 million lines of code, Microsoft's Windows operating system has about 50 million lines of code, and Google Search plus Gmail plus Google Maps consist of 2 billion lines of code. For many embedded systems, which consist of hardware and software, the software cost has increased to 90%–95% of the total system cost from 5%–10% three decades ago. Some embedded systems use application-specific integrated circuits (ASIC), system on chip (SoC), and/or firmware. These are integrated circuits with the software burned into the hardware. They are costly to replace; hence, the quality of the software is critical. These call for a software engineering approach to system development.

Second, software engineering supports teamwork, which is needed for large system development. Large software systems require considerable effort to design, implement, test, and maintain. A typical software engineer can produce on an average 50–100 lines of source code per day. This includes the time required to perform analysis, design, implementation, integration, and testing. Thus, a small system of 10,000 lines of code would require one software engineer to work between 100 and 200 days or 5–10 months. A medium-sized system of 500,000 lines of source code would require a software engineer to work 5,000–10,000 days, or 20–40 years. It is not acceptable for any business to wait this long. Therefore, real-world software systems must be designed and implemented by a team or teams of software engineers. For example, a medium-sized software system requires 20–40 software engineers to work for one year. When two or more software engineers work together to develop a software system, they face serious conceptualization, communication, and coordination challenges.

Conceptualization is the process of observing and classifying real-world phenomena to form a mental model to help understand the application for which the system is built. Conceptualization is a challenge for teamwork because the software engineers may perceive the world differently due to differences in their education, cultural backgrounds, career experiences, assumptions, and other factors. The parable of the blind men and an elephant explains this. We as software developers are like the four blind men trying to perceive or understand an application. If the team members perceive the application incorrectly, then how can they produce software that will correctly automate the application? If the team members have different perceptions, then how can they design and implement software components that will work with each other? Software engineering provides modeling languages such as the Unified Modeling Language (UML), methods and techniques to help developers establish a common understanding about an application for which the software is built.

When a team of software engineers works together, they need to communicate their analysis and design ideas. However, the natural language is too informal and sometimes ambiguous. Again, UML improves the communication among the developers. Finally, when teams of software engineers work together, how can they collaborate and coordinate their efforts? For example, how do they divide the work and assign the pieces to the teams and team members? How do they integrate the components designed and implemented by different teams and team members? Software engineering provides a solution. That is, software development processes and methodologies, software project management, and QA solve these problems.

1.3 SOFTWARE ENGINEERING ETHICS

Software is present everywhere in our society, and controls and affects every aspect of our lives. Software can do good or cause harm to our society or others. Therefore, software engineers must consider social and ethical responsibilities when designing, implementing, and testing software. In this regard, the ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices recommended the "Software Engineering Code of Ethics and Professional Practice" (Figure 1.2) as the standards for teaching and practicing software engineering.

Software engineers should adhere to these ethical standards in their professional practice as well as daily lives. For example, a software engineer must respect the confidentiality of the client or employer. A software engineer must also respect and protect the intellectual property of the client or employer. Sometimes, a software engineer must choose one act or another. For example, a software engineer happens

Software Engineering Code of Ethics and Professional Practice (Short Version)

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC—Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.
8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Source: https://ethics.acm.org/code-of-ethics/software-engineering-code/

FIGURE 1.2 The ACM/IEEE code of ethics

to know that a component may behave abruptly in rare circumstances, which might cause property damage or loss of lives. He also knows that his company wants to release the product quickly to gain back market share. If he reports the problem, then the release has to push back considerably, and he would become the "trouble maker." If he does not report, then devastating tragedy might happen. Such a hypothetic scenario has actually occurred again and again in our industry. Management persons also have to choose between right and wrong. If the software engineer reports the problem, would the management take it seriously? As a matter of fact, wrong doings pay big prices—companies were ordered to pay hefty fines, and individuals were jailed for their wrong acts.

Ethical dilemmas can occur in our daily lives. A college student had a job interview soon, but unfortunately his laptop broke. He wanted to borrow his girlfriend's laptop for the weekend to prepare for the interview, but the laptop belonged to her company. In this case, should she lend the laptop to her boyfriend, or not help when he needs the help? What would you think his girlfriend should do?

1.4 SOFTWARE ENGINEERING AND COMPUTER SCIENCE

What is the difference between software engineering and computer science? This question is often asked by students and working professionals. First of all, computer science emphasizes computational efficiency, resource sharing, accuracy, optimization, and performance. These can be measured accurately and relatively quickly. In the last several decades (i.e., from 1950 to the present), all efforts and resources spent in computer science research are aimed to improve these aspects. Most chapters of a computer science textbook are written about methods, algorithms, and techniques to improve or optimize these aspects.

Unlike computer science, software engineering emphasizes software PQCT. For example, obtaining an optimal solution is often the goal of computer science. Software engineering would use a good-enough solution to reduce development or maintenance time and costs. Efforts and resources spent in software engineering R&D are aimed at significantly improving software PQCT. Most chapters of a software engineering textbook are written about methods and techniques to improve these four aspects. Unfortunately, the impact of a software engineering process or methodology cannot be measured easily and immediately. To be meaningful, the impact must be assessed during a long period of time and consume tremendous resources. For example, researchers took more than one decade to realize that the uncontrolled goto statement is harmful. That is, the uncontrolled use of the goto statement results in poorly structured programs, which are difficult to understand, test, and maintain.

Computer science focuses only on technical aspects. Software engineering has to deal with nontechnical issues. For example, the early stages of the development process focus on identifying business needs and formulating requirements and constraints. These activities require domain knowledge, analysis and design experience, communication skill, and customer relations. Software engineering also requires knowledge and experience in project management. User interface design has to consider human factors such as user preference and how users would use the system.

In addition, software development must consider political issues because the system may affect many people in one way or another.

Recognizing the differences between software engineering and computer science could help in understanding and appreciating software engineering processes, methodologies, and principles. Consider, for example, the design of a software system that needs to access a database. Computer science might emphasize efficient data storage and retrieval, and favor a design in which the program accesses the database directly. Such a design would make the program sensitive to changes to the database design and database management system (DBMS). If the database schema or the DBMS is changed or replaced, then considerable changes have to be made to the program. This could be difficult and costly. Therefore, software engineering would not consider this a good design decision unless efficient database access is highly desired. Instead, software engineering would prefer a design that will minimize the impact of database change to reduce maintenance effort, costs, and time.

Despite the differences, software engineering and computer science are closely related. Computer science to software engineering is like physics to electrical and electronics engineering, or chemistry to chemical engineering. That is, computer science is a theoretical and technological foundation for software engineering. Software engineering is application of computer science. However, software engineering has its own research topics. These include research in software processes and methodologies, software verification, validation and testing techniques, among others.

Software engineering is a broad area. A software engineer should know areas of computer science including programming languages, algorithms and data structures, operating systems, database systems, artificial intelligence, and computer networks, to mention a few. Embedded systems development requires the software engineer to have a basic understanding of electronic circuits and how to interface with hardware devices. Finally, it takes time for a software engineer to gain domain knowledge and design experience to become a good software architect. These challenges and the ability to design and implement large complex systems to meet practical needs make software engineering an exciting area. The ever-expanding computer application creates great opportunities for the software engineer and software engineering researcher.

1.5 SUMMARY

Software engineering is defined as a discipline that investigates and applies engineering processes and methodologies to improve software PQCT. The need for software engineering is discussed, and software life-cycle activities are described. The chapter ends with a discussion of software engineering ethics and relationship between computer science and software engineering. That is, computer science is a foundation for software engineering. While computer science is mainly concerned with optimization and efficiency, software engineering is concerned with software PQCT. Knowing these should help understand software engineering and the rationale behind the processes, methodologies, modeling languages, design patterns, and many others. All these are designed to improve software PQCT.

1.6 CHAPTER REVIEW QUESTIONS

1. What is software engineering? Why is it needed?
2. What is a software development process?
3. What is software quality assurance?
4. What is software project management?
5. What are the differences and relationship between software engineering and computer science? Can we have one without the other?

1.7 EXERCISES

1.1 Search the literature and find four other definitions of software engineering in addition to the one given in this chapter. Discuss the pros and cons of these definitions.

1.2 A number of methods have been proposed for measuring software productivity. These include counting the lines of source code, number of classes, and number of methods delivered. Each of these has drawbacks. For example, each line of a program could be split into two to "double" the productivity although the functionality of the program has not changed. Discuss the pros and cons of each of these methods.

1.3 Describe in a brief article the functions of the three tracks of life-cycle activities. Discuss how the three tracks of activities work together during the software development life cycle. Discuss how they improve software PQCT.

1.4 Should optimization be a focus of software engineering? Briefly explain, and justify your answer with a practical example.

1.5 Identify three computer science courses of your choice. Show the usefulness of these courses in the software life-cycle activities.

1.6 There are interdependencies between software productivity, quality, cost, and time to market. For example, more time and effort spent in coding could increase productivity. This may result in less time and effort in quality assurance because the total time and effort of a project are fixed. Poor quality could reduce productivity due to rework. Identify three pairs of such interdependencies of your choice. Discuss their short-term and long-term impacts on the software development organization. How should software engineering solve this "dilemma" induced by these interdependencies?

1.7 What would you do if you were the software engineer described in Section 1.4?

1.8 What would you do if your boy/girlfriend desperately needs to use your laptop, but it belongs to your company?