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. The March 2006 issue of *Money* magazine ranked software engineer as number 1 of the 50 best jobs in the United States. According to the Bureau of Labor Statistics (BLS) 2010–2020 projections, the total number of jobs in application development software engineer and systems analyst positions is expected to increase from 520,800 to 664,500 (27.6%) and from 544,400 to 664,800 (22.10%), respectively. To be able to perform the work required of an application development software engineer or systems analyst, an education in software engineering is highly desired. However, according to the data released by BLS ("Earned Awards and Degrees, by Field of Study, 2005–2006"), only 160 bachelor and 600 master's degrees in software engineering, and 10,289 bachelor and 4,512 master's degrees in computer science were awarded in 2006. Thus, there is a significant gap between the demand and supply, especially for graduates with a software engineering degree.

Many people do not know the scope and usefulness of software engineering as a practice, and the discipline is often misunderstood. Many media outlets seem to define 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. Several years ago, after the first class of an object-oriented software engineering (OOSE) course, a student said to 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 as I thought." 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. OOSE is a branch of software engineering that is characterized by its view of the world as consisting of objects relating to and interacting with each other. The advent of the C++ programming language in the 1980s marked the beginning of the OOSE era. Since then, software production began its unprecedented worldwide growth and was further accelerated by the creation and worldwide adoption of the unified modeling language (UML) and the unified process (UP). Strictly speaking, a software process describes the phases and *what* should be done in each phase. It does not define (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 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 gaps discussed in the last paragraph, we need a methodology or a "cookbook." 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 a few years of on-the-job training to learn OO 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. Design patterns promote software reuse and improve team communication. 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, OO software testing, 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 methods or to be used as a "unified" methodology for all projects.

AUDIENCES

This book is for students majoring in computer science or software engineering as well as for 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 IT industry. This book's material evolved over the last decade from courses taught at several universities and companies, both 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 projects allowed me to observe closely how students and software engineers applied UP, UML, design patterns, and TDD, and the difficulties they faced. Their feedback led to continual improvement of the material.

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, lifecycle 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. Alternatively, these chapters could be taught as three courses—Software Quality Assurance, Software Testing, and Software 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 to VIII, and selected chapters from the other parts may form a Software Project Management course.
6. Finally, Part I, Parts II, and Chapter 24 plus selected patterns and topics from the other chapters may comprise a course on Introduction to Software Security. The instructor may supply additional materials to make this course more comprehensive.

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 and syllabi, lab work manuals, and software tools. Instructors who have not taught the courses may find these teaching tools 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 the methodology to real-world projects. My students also read and used the textbook and teaching materials. They have provided me with valuable feedback and improvement suggestions. I want to thank my daughter, Jacquelyn Kung, for editing a few chapters of the draft manuscript. Thanks are due to Raghu Srinivasan, the Global Publisher of McGraw-Hill Engineering & Computer Science Department, for his valuable improvement suggestions and guidance throughout the publication process. I also want to thank the reviewers for their comments and suggestions. These comments and suggestions have significantly improved the organization, presentation, and many other aspects of the book. During the lengthy writing process, my wife, Cindy Kung, and many of my colleagues in the academic institutions and industry in which I have collaborated, provided me constant encouragement and invaluable support. Thanks are due to them as well.