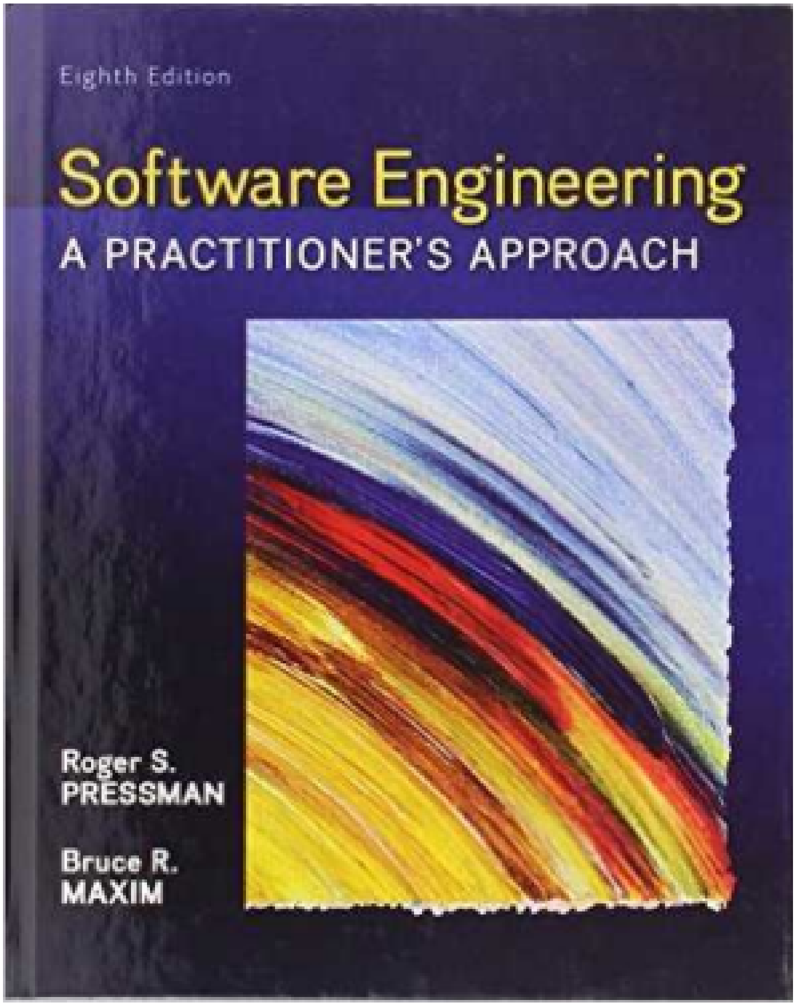


Software Engineering

This page intentionally left blank

Software Engineering

EIGHTH EDITION

Roger S. Pressman, Ph.D.
Bruce R. Maxim, Ph.D.

SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, EIGHTH EDITION

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 1 0 9 8 7 6 5 4

ISBN 978-0-07-802212-8
MHID 0-07-802212-6

Senior Vice President, Products & Markets: *Kurt L. Strand*
Vice President, General Manager: *Marty Lange* Vice President, Content Production & Technology Services: *Kimberly Meriwether David* Managing Director: *Thomas Timp*
Publisher: *Raghu Srinivasan*
Developmental Editor: *Vincent Bradshaw*
Marketing Manager: *Heather Wagner*
Director, Content Production: *Terri Schiesl*
Project Manager: *Heather Ervolino*
Buyer: *Sandy Ludovissy*
Cover Designer: *Studio Montage, St. Louis, MO.*
Cover Image: *Farinaz Taghavi/Getty images*
Compositor: *MPS Limited*
Typeface: *8.5/13.5 Impressum Std*
Printer: *R. R. Donnelley*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page. **Library of Congress Cataloging-in-Publication Data**

Pressman, Roger S.
Software engineering : a practitioner's approach / Roger S. Pressman, Ph.D. — Eighth edition.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-07-802212-8 (alk. paper)
ISBN-10: 0-07-802212-6 (alk. paper)
1. Software engineering. I. Title.
QA76.758.P75 2015
005.1—dc23
2013035493

www.mhhe.com

To my granddaughters o my granddaughters
Lily and Maya, who already ily and Maya, who already
understand the importance nderstand the importance
of software, even though they're f

software, even though they're still in preschool. still in preschool.

—Roger S. Pressman Roger S. Pressman

In loving memory of In loving memory of my parents, who taught y parents, who taught me from an early age that e from an early age that pursuing a good education ursuing a good education was far more important as far more important than pursuing money. han pursuing money.

—Bruce R. Maxim —Bruce R. Maxim

Roger S. Pressman is an internationally recognized consultant and author in soft ware engineering. For more than four decades, he has worked as a software engi neer, a manager, a professor, an author, a consultant, and an entrepreneur.

Dr. Pressman is president of R. S. Pressman & Associates, Inc., a consulting fi rm that specializes in helping companies establish effective software engineer ing practices. Over the years he has developed a set of techniques and tools that improve software engineering practice. He is also the founder of Teslaccessories, LLC, a start-up manufacturing company that specializes in custom products for the Tesla Model S electric vehicle.

Dr. Pressman is the author of nine books, including two novels, and many techni cal and management papers. He has been on the editorial boards of *IEEE Software* and *The Cutter IT Journal* and was editor of the "Manager" column in *IEEE Software.*

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He has presented tutorials at the International Conference on Soft ware Engineering and at many other industry meetings. He has been a member of the ACM, IEEE, and Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma. **Bruce R. Maxim** has worked as a software engineer, project manager, professor, author, and consultant for more than thirty years. His research interests include software engineering, human computer interaction, game design, social media, artifi cial intelligence, and computer science education.

Dr. Maxim is associate professor of computer and information science at the University of Michigan—Dearborn. He established the GAME Lab in the College of Engineering and Computer Science. He has published a number of papers on computer algorithm animation, game development, and engineering education. He is coauthor of a best-selling introductory computer science text. Dr. Maxim has supervised several hundred industry-based software development projects as part of his work at UM-Dearborn.

Dr. Maxim's professional experience includes managing research informa tion systems at a medical school, directing instructional computing for a medical campus, and working as a statistical programmer. Dr. Maxim served as the chief technology offi cer for a game development company.

Dr. Maxim was the recipient of several distinguished teaching awards and a distinguished community service award. He is a member of Sigma Xi, Upsilon Pi Epsilon, Pi Mu Epsilon, Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers, and International Game Developers Association.

CHAPTER 8 UNDERSTANDING REQUIREMENTS 131

CHAPTER 9 REQUIREMENTS MODELING: SCENARIO-BASED METHODS 166

CHAPTER 10 REQUIREMENTS MODELING: CLASS-BASED METHODS 184

CHAPTER 29 SOFTWARE CONFIGURATION MANAGEMENT 623

CHAPTER 30 PRODUCT METRICS 653

When computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

It has been almost three and a half decades since the first edition of this book was written. During that time, software engineering has evolved from an obscure idea practiced by a relatively small number of zealots to a legitimate engineering discipline. Today, it is recognized as a subject worthy of serious research, conscientious study, and tumultuous debate. Throughout the industry, software engineer has replaced programmer as the job title of preference. Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry segments.

Although managers and practitioners alike recognize the need for a more disciplined approach to software, they continue to debate the manner in which discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service today's most advanced technologies. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers, and bad things happen. In addition, debate and controversy about the true nature of the software engineering approach continue. The status of software engineering is a study in contrasts. Attitudes have changed, progress has been made, but much remains to be done before the discipline reaches full maturity.

The eighth edition of *Software Engineering: A Practitioner's Approach* is intended to serve as a guide to a maturing engineering discipline. The eighth edition, like the seven editions that preceded it, is intended for both students and practitioners, retaining its appeal as a guide to the industry professional and a comprehensive introduction to the student at the upper-level undergraduate or first-year graduate level.

The eighth edition is considerably more than a simple update. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. In addition, we have further enhanced the popular "support system" for the book, providing a comprehensive set of student, instructor, and professional resources to complement the content of the book. These resources are presented as part of a website (www.mhhe.com/pressman) specifically designed for *Software Engineering: A Practitioner's Approach.*

The Eighth Edition. The 39 chapters of the eighth edition are organized into five parts. This organization better compartmentalizes topics and assists instructors who may not have the time to complete the entire book in one term.

Part 1, *The Process,* presents a variety of different views of software process, considering all important process models and addressing the debate between prescriptive and agile process philosophies. Part 2, *Modeling,* presents analysis and design methods with an emphasis on object-oriented techniques and UML modeling. Pattern based design and design for Web and mobile applications are also considered. Part 3, *Quality Management,* presents the concepts, procedures, techniques, and methods that enable a software team to assess software quality, review software engineering work products, conduct SQA procedures, and apply an effective testing strategy and tactics. In addition, formal modeling and verifi cation methods are also considered. Part 4, *Managing Software Projects,* presents topics that are relevant to those who plan, manage, and control a software development project. Part 5, *Advanced Topics,* considers software process improvement and software engineering trends. Continu ing in the tradition of past editions, a series of sidebars is used throughout the book to present the trials and tribulations of a (fi ctional) software team and to provide supple mentary materials about methods and tools that are relevant to chapter topics.

The fi ve-part organization of the eighth edition enables an instructor to “cluster” topics based on available time and student need. An entire one-term course can be built around one or more of the fi ve parts. A software engineering survey course would select chapters from all fi ve parts. A software engineering course that empha

sizes analysis and design would select topics from Parts 1 and 2. A testing-oriented software engineering course would select topics from Parts 1 and 3, with a brief foray into Part 2. A “management course” would stress Parts 1 and 4. By organizing the eighth edition in this way, we have attempted to provide an instructor with a number of teaching options. In every case the content of the eighth edition is complemented by the following elements of the *SEPA, 8/e Support System.*

Student Resources. A wide variety of student resources includes an extensive on line learning center encompassing chapter-by-chapter study guides, practice quizzes, problem solutions, and a variety of Web-based resources including software engineer ing checklists, an evolving collection of “tiny tools,” a comprehensive case study, work product templates, and many other resources. In addition, over 1,000 categorized *Web References* allow a student to explore software engineering in greater detail and a *Reference Library* with links to more than 500 downloadable papers provides an in depth source of advanced software engineering information.

Instructor Resources. A broad array of instructor resources has been developed to supplement the eighth edition. These include a complete online *Instructor’s Guide* (also downloadable) and supplementary teaching materials including a complete set of more than 700 *PowerPoint Slides* that may be used for lectures, and a test bank. Of course, all resources available for students (e.g, tiny tools, the Web References, the downloadable Reference Library) and professionals are also available.

The *Instructor’s Guide for Software Engineering: A Practitioner’s Approach* pres ents suggestions for conducting various types of software engineering courses, rec ommendations for a variety of software projects to be conducted in conjunction with a course, solutions to selected problems, and a number of useful teaching aids.

Professional Resources. A collection of resources available to industry practitioners (as well as students and faculty) includes outlines and samples of software engineering documents and other work products, a useful set of software engineering checklists,

a catalog of software engineering tools, a comprehensive collection of Web-based re sources, and an “adaptable process model” that provides a detailed task breakdown of the software engineering process.

McGraw-Hill Connect® Computer Science provides online presentation, assignment, and assessment solu tions. It connects your students with the tools and resources they’ll need to achieve success. With Connect **Computer Science** you can deliver assignments, quizzes, and tests online. A robust set of questions and activi ties are presented and aligned with the textbook’s learning outcomes. As an instruc tor, you can edit existing questions and author entirely new problems. Integrate grade reports easily with Learning Management Systems (LMS), such as WebCT and Blackboard—and much more. ConnectPlus® **Computer Science** provides students with all the advantages of Connect **Computer Science**, plus 24/7 online access to a media-rich eBook, allowing seamless integration of text, media, and assessments. To learn more, visit **www.mcgrawhillconnect.com**

McGraw-Hill LearnSmart® is available as a standalone product or an integrated feature of McGraw-Hill Connect **Computer Science**. It is an adaptive learning system designed to help students learn faster, study more effi ciently, and retain more knowledge for greater success. LearnSmart assesses a student's knowl edge of course content through a series of adaptive questions. It pinpoints concepts the student does not understand and maps out a personalized study plan for success. This innovative study tool also has features that allow instructors to see exactly what students have accomplished and a built-in assessment tool for graded assignments. Visit the following site for a demonstration. **www.mhlearnsmart.com**

Powered by the intelligent and adaptive LearnSmart engine, **SmartBook™** is the fi rst and only continuously adaptive reading experience available today. Distin guishing what students know from what they don't, and honing in on concepts they are most likely to forget, SmartBook personalizes content for each student. Reading is no longer a passive and linear experience but an engaging and dynamic one, where students are more likely to master and retain important concepts, coming to class better prepared. SmartBook includes powerful reports that identify specifi c topics and learning objectives students need to study.

When coupled with its online support system, the eighth edition of *Software Engineering: A Practitioner's Approach,* provides fl exibility and depth of content that cannot be achieved by a textbook alone.

With this edition of *Software Engineering: A Practitioner's Approach,* Bruce Maxim joins me (Roger Pressman) as a coauthor of the book. Bruce brought copious software engineering knowledge to the project and has added new content and insight that will be invaluable to readers of this edition.

Acknowledgments. Special thanks go to Tim Lethbridge of the University of Ottawa who assisted us in the development of UML and OCL examples, and developed the case study that accompanies this book, and Dale Skrien of Colby College, who devel oped the UML tutorial in Appendix 1. Their assistance and comments were invaluable.

In addition, we'd like to thank Austin Krauss, Senior Software Engineer at Treyarch, for providing insight into software development in the video game industry. We also wish to thank the reviewers of the eighth edition: Manuel E. Bermudez, University of Florida; Scott DeLoach, Kansas State University; Alex Liu, Michigan State University; and Dean Mathias, Utah State University. Their in-depth comments and thoughtful criticism have helped us make this a much better book.

Special Thanks. BRM: I am grateful to have had the opportunity to work with Roger on the eighth edition of this book. During the time I have been working on this book my son Benjamin shipped his fi rst MobileApp and my daughter Katherine launched her interior design career. I am quite pleased to see the adults they have become. I am very grateful to my wife, Norma, for the enthusiastic support she has given me as I fi lled my free time with working on this book.

RSP: As the editions of this book have evolved, my sons, Mathew and Michael, have grown from boys to men. Their maturity, character, and success in the real world have been an inspiration to me. Nothing has fi lled me with more pride. They now have children of their own, Maya and Lily, who start still another generation. Both girls are already wizards on mobile computing devices. Finally, to my wife Barbara, my love and thanks for tolerating the many, many hours in the offi ce and encouraging still another edition of "the book."

Roger S. Pressman
Bruce R. Maxim

CHAPTER

1

OF SOFTWARE

THE NATURE

KEY CONCEPTS

As he fi nished showing me the latest build of one of the world's most popular fi rst-person shooter video games, the young developer laughed.

"You're not a gamer, are you?" he asked.

I smiled. "How'd you guess?"

The young man was dressed in shorts and a tee shirt. His leg bounced up and down like a piston, burning the nervous energy that seemed to be com monplace among his co-workers.

"Because if you were," he said, "you'd be a lot more excited. You've gotten a peek at our next generation product and that's something that our customers would kill for . . . no pun intended."

We sat in a development area at one of the most successful game develop ers on the planet. Over the years, earlier generations of the game he demoed sold over 50 million copies and generated billions of dollars in revenue. "So, when will this version be on the market?" I asked.

He shrugged. "In about fi ve months, and we've still got a lot of work to do." He had responsibility for game play and artifi cial intelligence functionality in an application that encompassed more than three million lines of code.

"Do you guys use any software engineering techniques?" I asked, half expecting that he'd laugh and shake his head.

What is it? Computer software is the product that software profession als build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

Why is it important? Software is important because it affects nearly every aspect of our lives and has become pervasive in our com merce, our culture, and our everyday activities.

What are the steps? Customers and other stakeholders express the need for computer software, engineers build the software prod uct, and end users apply the software to solve a specifi c problem or to address a specifi c need.

What is the work product? A computer pro gram that runs in one or more specifi c environ ments and services the needs of one or more end users.

How do I ensure that I've done it right? If you're a software engineer, apply the ideas contained in the remainder of this book. If you're an end user, be sure you un derstand your need and your environment and then select an application that best meets them both.

He paused and thought for a moment. Then he slowly nodded. “We adapt them to our needs, but sure, we use them.”

“Where?” I asked, probing.

“Our problem is often translating the requirements the creatives give us.” “The creatives?” I interrupted.

“You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game.”

“And after the requirements are established?”

He shrugged. “We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends.”

“You know my book?” I was honestly surprised.

“Sure, used it in school. There’s a lot there.”

“I’ve talked to some of your buddies here, and they’re more skeptical about the stuff in my book.”

He frowned. “Look, we’re not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques.” “And how will your subset change as the years pass?”

He paused as if to ponder the future. “Games will become bigger and more complex, that’s for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don’t, we’re dead.”

uote:

“Ideas and technological discoveries are the driving engines of economic growth.”

Wall Street Journal

Computer software continues to be the single most important technology on the world stage. And it’s also a prime example of the law of unintended conse quences. Sixty years ago no one could have predicted that software would be come an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineer ing and nanotechnology), the extension of existing technologies (e.g., telecom munications), and the radical change in older technologies (e.g., the media); that software would be the driving force behind the personal computer revolu tion; that software applications would be purchased by consumers using their smart phones; that software would slowly evolve from a product to a service as “on-demand” software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more infl uential than all industrial-era companies; that a vast software-driven network would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults. No one could foresee that software would become embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial,

entertainment, offi ce machines, . . . the list is almost endless. And if you believe the law of unintended consequences, there are many effects that we cannot yet predict. No one could predict that millions of computer programs would have to be corrected, adapted, and enhanced as time passed. The burden of performing these "maintenance" activities would absorb more people and more resources than all work applied to the creation of new software.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less ex pensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specifi c application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad-based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering.*

1.1 The Nature of Software

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers

Software is both a product and a vehicle that delivers a product.

that are accessible by local hardware. Whether it resides within a mobile phone, a hand-held tablet, on the desktop, or within a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, dis playing, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communica tion of information (networks), and the creation and control of other programs

(software tools and environments).

Software delivers the most important product of our time— *information.* It transforms personal data (e.g., an individual's fi nancial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information net

works (e.g., the Internet), and provides the means for acquiring information in all of its forms. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

The role of computer software has undergone signifi cant change over the

uote:

"Software is
a place where
dreams are planted and nightmares harvested, an abstract, mystical swamp where
terrible demons compete with
magical panaceas, a world of
werewolves and silver bullets."

Brad J. Cox

last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage ca pacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build and protect complex systems. Today, a huge software industry has become a dominant factor in the econ omies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone programmer are the same questions that are asked when mod ern computer-based systems are built: [1]

- Why does it take so long to get software fi nished?
- Why are development costs so high?
- Why can't we fi nd all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have diffi culty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has led to the adoption of software engineering practice.

1.1.1 Defi ning Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:

Software is: (1) instructions (computer programs) that when executed provide de sired features, function, and performance; (2) data structures that enable the pro

?

How should we defi ne *software*?

grams to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete defi nitions could be offered. But a more formal defi nition probably won't measurably improve your understanding.

"Infant mortality" "Wear out"

1 In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin ask ing 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

Failure curve for hardware

If you want to reduce software deterioration, you'll have to do better software design (Chapters 12 to 18).

Time

To accomplish that, it’s important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has one fundamen tal characteristic that makes it considerably different from hardware: *Software doesn’t “wear out.”*

Figure 1.1depicts failure rate as a function of time for hardware. The relation ship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental maladies that cause hard ware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 1.2 . Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve fl attens as shown. The idealized curve is a gross over simplifi cation of actual failure models for software. However, the implication is clear—software doesn’t wear out. But it does *deteriorate!*

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2 . During its life, [2] software will undergo change. As changes are

2 In fact, from the moment that development begins and long before the fi rst version is delivered, changes may be requested by a variety of different stakeholders.

Failure curves
for software

Change

Actual curve

Idealized curve

Time

Increased failure rate due to side

effects

Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve in Figure 1.2 .

made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve” (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and soft ware. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing chal lenges for software engineers:

System software— a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and fi le man agement utilities) processes complex, but determinate, [3] information struc tures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

3 Software is *determinate* if the order and timing of inputs, processing, and outputs is predict able. Software is *indeterminate* if the order and timing of inputs, processing, and outputs can not be predicted in advance.

One of the most comprehensive libraries of shareware/freeware can be found at shareware.cnet.com

uote:

“What a computer is to me is the most remarkable tool that we have ever come up with. It’s the equivalent of a bicycle for our minds.”

Steve Jobs

Application software —stand-alone programs that solve a specifi c business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.

Engineering/scientific software —a broad array of “number-crunching programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

Embedded software— resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric func tions (e.g., key pad control for a microwave oven) or provide signifi cant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software —designed to provide a specifi c capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer.

Web/Mobile applications —this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

Artificial intelligence software— makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artifi cial neural networks, theorem proving, and game playing.

Millions of software engineers worldwide are hard at work on software proj ects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.

1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modifi ed to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who fi nd them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispens able' to the business." Hence, legacy software is characterized by longevity and business criticality.

What do I do if I encounter a legacy system that exhibits poor quality?

Unfortunately, there is sometimes one additional characteristic that is pres ent in legacy software— *poor quality*. [4] Legacy systems sometimes have inextensi ble designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support "core business functions and are indispensable to the business." What to do?

The only reasonable answer may be: *Do nothing,* at least until the legacy sys tem must undergo some signifi cant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fi xed. However, as time passes, legacy systems often evolve for one or more of the fol lowing reasons:

What types of changes are made to legacy systems?

- The software must be adapted to meet the needs of new computing envi ronments or technology.

must be reengineered (Chapter 36) so that it remains viable into the future. The goal of modern soft ware engineering is to "devise methodologies that are founded on the notion of evolution;" that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other." [Day99]

Every software engi neer must recognize that change is natural. Don't try to fi ght it.

- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a evolving computing environment.

When these modes of evolution occur, a legacy system

4 In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

1.2 The Changing Nature of Software

Four broad categories of software are evolving to dominate the industry. And yet, these categories were in their infancy little more than a decade ago.

1.2.1 WebApps

In the early days of the World Wide Web (circa 1990 to 1995), *websites* consisted of little more than a set of linked hypertext fi les that presented information using text and limited graphics. As time passed, the augmentation of HTML by devel opment tools (e.g., XML, Java) enabled Web engineers to provide computing ca pability along with informational content. *Web-based systems and applications*[5] (we refer to these collectively as *WebApps*) were born.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

uote:

"By the time
we see any sort of stabilization, the Web will have turned
into something completely
different."

Louis Monier

A decade ago, WebApps "involve[d] a mixture between print publishing and software development, between marketing and computing, between inter nal communications and external relations, and between art and technology." [Pow98] But today, they provide full computing potential in many of the applica tion categories noted in Section 1.1.2.

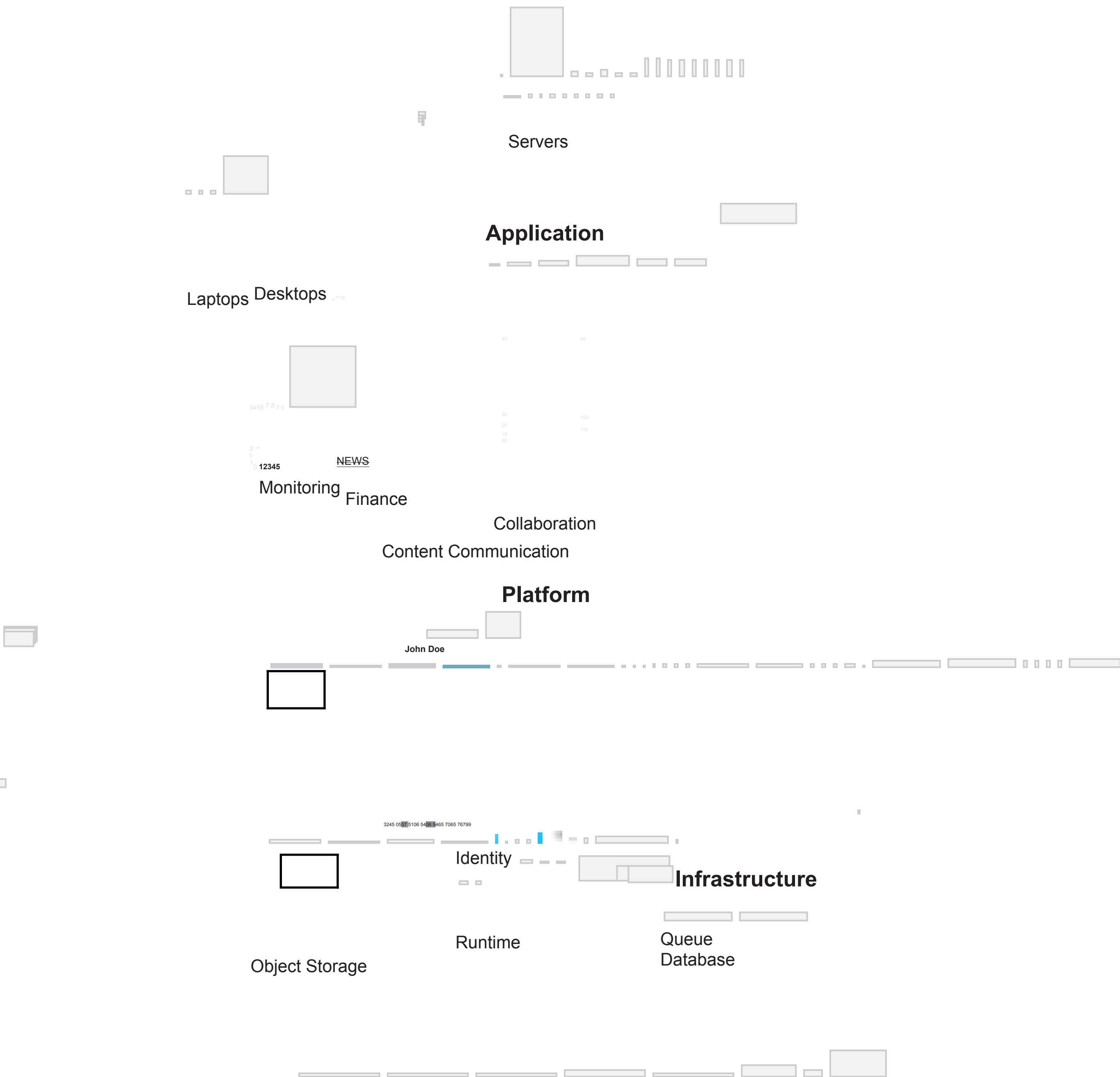
Over the past decade, Semantic Web technologies (often referred to as Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass "semantic databases [that] provide new functionality that requires Web linking, fl exible [data] representation, and external access APIs." [Hen10] Sophisticated relational data structures will lead to entirely new WebApps that allow access to disparate information in ways never before possible.

1.2.2 Mobile Applications

The term *app* has evolved to connote software that has been specifi cally de signed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile). In most instances, mobile applications encompass a user interface that takes ad vantage of the unique interaction mechanisms provided by the mobile platform, interoperability with Web-based resources that provide access to a wide array of information that is relevant to the app, and local processing capabilities that collect, analyze, and format information in a manner that is best suited to the mobile platform. In addition, a mobile app provides persistent storage capabili ties within the platform.

5 In the context of this book, the term *Web application* (WebApp) encompasses everything from a simple Web page that might help a consumer compute an automobile lease payment to a com prehensive website that provides complete travel services for businesspeople and vacationers. Included within this category are complete websites, specialized functionality within websites, and information processing applications that reside on the Internet or on an intranet or extranet.

Cloud computing logical architecture [Wik13]



Cloud Computing

It is important to recognize that there apps. A *mobile web application* is a subtle distinction between (WebApp) allows a mobile web applications and mobile

?
What is the difference between a WebApp and a mobile app?

mobile device to gain access to web-based content via a browser that has been specifi cally designed to accommodate the strengths and weaknesses of the mo bile platform. A *mobile app* can gain direct access to the hardware characteris tics of the device (e.g., accelerometer or GPS location) and then provide the local processing and storage capabilities noted earlier. As time passes, the distinction between mobile WebApps and mobile apps will blur as mobile browsers become more sophisticated and gain access to device level hardware and information.

1.2.3 Cloud Computing

Cloud computing encompasses an infrastructure or "ecosystem" that enables any user, anywhere, to use a computing device to share computing resources on a broad scale. The overall logical architecture of cloud computing is represented in Figure 1.3 .

Referring to the fi gure, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass ap plications, platforms, and infrastructure. In its simplest form, an external com puting device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data struc tures. In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device.

The implementation of cloud computing requires the development of an ar chitecture that encompasses front-end and back-end services. The *front-end* in cludes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed. The *back-end* includes servers and related computing resources, data storage systems (e.g., databases), server-resident ap plications, and administrative servers that use middleware to coordinate and monitor traffi c by establishing a set of protocols for access to the cloud and its resident resources. [Str08]

The cloud architecture can be segmented to provide access at a variety of different levels from full public access to private cloud architectures accessible only to those with authorization.

1.2.4 Product Line Software

The Software Engineering Institute defi nes a *software product line* as "a set of software-intensive systems that share a common, managed set of features sat isfying the specifi c needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [SEI13] The concept of a line of software products that are related in some way is not new. But the idea that a line of software products, all developed using the same underlying application and data architectures, and all implemented using a set of reusable software components that can be reused across the product line pro vides signifi cant engineering leverage.

A software product line shares a set of assets that include requirements (Chapter 8), architecture (Chapter 13), design patterns (Chapter 16), reusable components (Chapter 14), test cases (Chapters 22 and 23), and other software engineering work products. In essence, a software product line results in the development of many products that are engineered by capitalizing on the com

monality among all the products within the product line.

1.3 Summary

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble devel

oping high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

The nature of software is changing. Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these WebApps have unique features and requirements, they are software nonethe

less. Mobile applications present new challenges as apps migrate to a wide array of platforms. Cloud computing will transform the way in which software is deliv ered and the environment in which it exists. Product line software offers poten tial effi ciencies in the manner in which software is built.

Problems and Points to Ponder

1.1. Provide at least fi ve additional examples of how the law of unintended consequences applies to computer software.

1.2. Provide a number of examples (both positive and negative) that indicate the impact of software on our society.

1.3. Develop your own answers to the fi ve questions asked at the beginning of Section 1.1. Discuss them with your fellow students.

1.4. Many modern applications change frequently—before they are presented to the end user and then after the fi rst version has been put into use. Suggest a few ways to build soft ware to stop deterioration due to change.

1.5. Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.

Further Readings and Information Sources [6]

Literally thousands of books are written about computer software. The vast majority dis cuss programming languages or software applications, but a few discuss software itself. Pressman and Herron (*Software Shock,* Dorset House, 1991) presented an early discussion (directed at the layperson) of software and the way professionals build it. Negroponte's best-selling book (*Being Digital*, Alfred A. Knopf, 1995) provides a view of computing and its overall impact in the twenty-fi rst century. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software

6 The *Further Reading and Information Sources* section presented at the conclusion of each chapter presents a brief overview of print sources that can help to expand your understanding of the major topics presented in the chapter. We have created a comprehensive website to support *Software Engineering: A Practitioner's Approach* at **www.mhhe.com/pressman**. Among the many topics addressed within the website are chapter-by-chapter software engineering resources to Web-based information that can complement the material presented in each chapter. An Amazon.com link to every book noted in this section is contained within these resources.

and the process through which it is developed. Ray Kurzweil (*How to Create a Mind,* Viking, 2013) discusses how software will soon mimic human thought and lead to a "singularity" in the evolution of humans and machines.

Keeves (*Catching Digital,* Business Infomedia Online, 2012) discusses how business lead ers must adapt as software evolves at an ever-increasing pace. Minasi (*The Software Con spiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do,* McGraw-Hill, 2000) argues that the "modern scourge" of software bugs can be eliminated and suggests ways to accomplish this. Books by Eubanks (*Digital Dead End: Fighting for Social Justice in the Information Age,* MIT Press, 2011 *)* andCompaine (*Digital Divide: Facing a Crisis or Creating a Myth,* MIT Press, 2001) argue that the "divide" between those who have access to information resources (e.g., the Web) and those that do not is narrowing as we move into the fi rst decade of this century. Books by Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design,* Morgan Kaufman, 2010), Greenfi eld (*Everyware: The Dawning Age of Ubiquitous Computing,* New Riders Publishing, 2006), and Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications,* Auerbach, 2006) introduce the concept of "open-world" software and predict a wireless en vironment in which software must adapt to requirements that emerge in real time.

A wide variety of information sources that discuss the nature of software are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**

CHAPTER

SOFTWARE 2 ENGINEERING

KEY CONCEPTS

framework activities . . . . 17 general principles . . 21 principles . . . . 21 problem solving . . . 19 *SafeHome* . . . . 26 software engineering, defi nition . . . . 15 layers . . . . 15 practice . . . . 19 software myths . . . . 23 software process . . 16 umbrella activities. . 17

In order to build software that is ready to meet the challenges of the twenty-fi rst century, you must recognize a few simple realities:

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an inter est in the features and functions provided by a specifi c application [1] has grown dramatically. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*
- The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each pass ing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environ ment is now embedded inside everything from consumer electronics to medical devices to weapons systems. *It follows that design becomes a pivotal activity.*

What is it? Software engineering encompasses a process, a collec tion of methods (practice) and an array of tools that allow profession

by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

als to build high-quality computer software.

Who does it? Software engineers apply the software engineering process.

Why is it important? Software engineering is important because it enables us to build com plex systems in a timely manner and with high quality. It imposes discipline to work that can be come quite chaotic, but it also allows the people who build computer software to adapt their ap proach in a manner that best suits their needs.

What are the steps? You build computer soft ware like you build any successful product,

What is the work product? From the point of view of a software engineer, the work prod uct is the set of programs, content (data), and other work products that are computer soft ware. But from the user's viewpoint, the work product is the resultant information that some how makes the user's world better.

How do I ensure that I've done it right? Read the remainder of this book, se lect those ideas that are applicable to the soft ware that you build, and apply them to your work.

1 We will call these people "stakeholders" later in this book.

Understand the prob lem before you build a solution.

Both quality and maintainability are an outgrowth of good design.

- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*
- As the perceived value of a specifi c application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. *It fol lows that software should be maintainable.*

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book— *software engineering.*

2.1 Defining the Discipline

?

How do we defi ne software engineering?

The IEEE [IEE93a] has developed the following defi nition for software engineering:

> Software Engineering : (1) The application of a systematic, disciplined, quantifi able approach to the development, operation, and maintenance of software; that is, the

Software engineering encompasses a pro cess, methods for man aging and engineering software, and tools.

application of engineering to software. (2) The study of approaches as in (1).

And yet, a “systematic, disciplined, and quantifi able” approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software engineering is a layered technology. Referring to Figure 2.1 , any en gineering approach (including software engineering) must rest on an organiza tional commitment to quality. Total quality management, Six Sigma, and similar philosophies [2] foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engi neering is a quality focus.

The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defi nes a framework that must be established for effective delivery of software engineer ing technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are pro duced, milestones are established, quality is ensured, and change is properly managed.

2 Quality management and related approaches are discussed throughout Part 3 of this book.

Methods Process

Software engi neering layers

Tools

Software engineering *methods* provide the technical how-to’s for building software. Methods encompass a broad array of tasks that include communica tion, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

CrossTalk is a journal that provides pragmatic information on process, methods, and tools. It can be found at: www.stsc.hill.af.mil ,

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

2.2 The Software Process

? What are the elements of a software process?

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the ap plication domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural model). A *task* focuses on a small, but well-defi ned objective (e.g., conducting a unit test) that produces a tangible outcome.

uote:

"A process defi nes who is doing *what* *when* and *how* to reach a certain goal."

Ivar Jacobson, Grady Booch, and James Rumbaugh

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropri ate set of work actions and tasks. The intent is always to deliver software in a timely manner and with suffi cient quality to satisfy those who have sponsored its creation and those who will use it.

2.2.1 The Process Framework

? What are the fi ve generic process framework activities?

A *process framework* establishes the foundation for a complete software engi neering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In ad dition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses fi ve activities:

uote:

"Einstein argued that there must be a simplifi ed explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity."

Fred Brooks

Communication. Before any technical work can commence, it is critically im portant to communicate and collaborate with the customer (and other stake holders).[3] The intent is to understand stakeholders' objectives for the project and to gather requirements that help defi ne software features and functions.

Planning. Any complicated journey can be simplifi ed if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defi nes the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engi neer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fi t together, and many other char acteristics. If required, you refi ne the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. What you design must be built. This activity combines code gen eration (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed in crement) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These fi ve generic framework activitiescan be used during the development of small, simple programs, the creation of Web applications, and for the engineering

3 A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. Rob Thomsett jokes that, "a stakeholder is a person holding a large and sharp stake . . . If you don't look after your stake holders, you know where the stake will end up."

of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same. For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is pro duced, the software becomes more and more complete.

2.2.2 Umbrella Activities

Software engineering process framework activities are complemented by a num ber of *umbrella activities.* In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, qual ity, change, and risk. Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defi nes and conducts the activities required to ensure software quality.

Technical reviews—assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defi nes and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software confi guration management—manages the effects of change throughout the software process.

Reusability management—defi nes criteria for work product reuse (includ ing software components) and establishes mechanisms to achieve reusable components.

Software process adaptation is essential for project success.

Work product preparation and production—encompass the activities re quired to create work products such as models, documents, logs, forms, and lists.

Each of these umbrella activities is discussed in detail later in this book.

2.2.3 Process Adaptation

Previously in this section, we noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one proj ect might be signifi cantly different than a process adopted for another project. Among the differences are

- Overall fl ow of activities, actions, and tasks and the interdependencies among them.
- Degree to which actions and tasks are defi ned within each framework activity.
- Degree to which work products are identifi ed and required.
- Manner in which quality assurance activities are applied.
- Manner in which project tracking and control activities are applied.
- Overall degree of detail and rigor with which the process is described.
- Degree to which the customer and other stakeholders are involved with the project.
- Level of autonomy given to the software team.
- Degree to which team organization and roles are prescribed.

In Part 1 of this book, we examine software process in considerable detail.

uote:

"I feel a recipe is only a theme which an intelligent cook can play each time with a variation."

Madame Benoit

2.3 Software Engineering Practice

In Section 2.2, we introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Ge

A variety of thought provoking quotes on the practice of software engineering can be found at www.literate programming.com .

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

neric framework activities— **communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fi t in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities. [4]

2.3.1 The Essence of Practice

In the classic book, *How to Solve It,* written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving , and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

4 You should revisit relevant sections within this chapter as we discuss specifi c software engi neering methods and umbrella

activities later in this book.

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

Understand the problem. It's sometimes diffi cult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds

The most important element of problem understanding is listening*.*

uote:

"There is a grain of discovery in the solution of any problem."

George Polya

and then think, *Oh yeah, I understand, let's get on with solving this thing.* Unfor tunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the solution. Now you understand the problem (or so you think), and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recog nizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defi ned?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implemen tation?* Can a design model be created?

Carry out the plan. The design you've created serves as a road map for the sys tem you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the de sign model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

2.3.2 General Principles

The dictionary defi nes the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we'll dis cuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specifi c generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software en gineering practice as a whole. They are reproduced in the following paragraphs: [5]

The First Principle: *The Reason It All Exists he Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system re quirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is no, don't do it. All other principles support this one.

Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

uote:

"There is a certain majesty in simplicity which is far above all the quaintness of wit."

Alexander Pope (1688–1744)

The Second Principle: *KISS (Keep It Simple, Stupid!) ISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to con sider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be dis carded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a suffi cient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder

5 Reproduced with permission of the author [Hoo96]. Hooker defi nes patterns for these princi ples at **http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment**

often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision aintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incom patible designs, held together by the wrong kind of screws . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vi sion and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume hat You Produce, Others Will Consume*

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

The Fifth Principle: *Be Open to the Future e Open to the Future*

A system with a long lifetime has more value. In today's computing envi ronments, where specifi cations change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these sys tems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*. Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the spe cifi c one. [6] This could very possibly lead to the reuse of an entire system.

The Sixth Principle: *Plan Ahead for Reuse lan Ahead for Reuse*

Reuse saves time and effort. [7] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code

6 This advice can be dangerous if it is taken to extremes. Designing for the “general problem” sometimes requires performance compromises and can make specifi c solutions ineffi cient. 7 Although this is true for those who reuse the software on future projects, reuse can be expen sive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justifi ed.

and designs has been proclaimed as a major benefi t of using object-oriented technologies. However, the return on this investment is not automatic. To lever age the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to re alize reuse at every level of the system development process . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable compo nents and the systems into which they are incorporated*.

The Seventh Principle: *Think!*

This last Principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the fi rst six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the diffi culties we experience in building complex computer-based systems would be eliminated.

2.4 Software Development Myths

Software development myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced prac
titioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious prob lems for managers and practitioners alike. However, old attitudes and habits are diffi cult to modify, and remnants of software myths remain.

The Software Project Managers Network at www.spmn.com can help you dispel these and other myths.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep sched ules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are soft ware practitioners aware of its existence? Does it refl ect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these ques tions is no.

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like man ufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At fi rst, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive devel opment effort. People can be added but only in a planned and well-coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that fi rm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it out sources software projects.

Customer myths. A customer who requests computer software may be a per son at the next desk, a technical group down the hall, the marketing/sales de partment, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take.

Myth: *A general statement of objectives is suffi cient to begin writing programs—we can fi ll in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be eas ily accommodated because software is fl exible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When re quirements changes are requested early (before design or code

Whenever you think, we don't have time for software engineering, ask yourself, "Will we have time to do it over again?"

has been started), the cost impact is relatively small. [8] However, as time passes, the cost impact grows rapidly— resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modifi cation.

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 60 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the fi rst time.

Myth: *Until I get the program “running” I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project— *the technical review.* Software reviews (described in Chapter 20) are a “quality fi lter” that have been found to be more effective than testing for fi nding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software confi guration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful en gineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnec essary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Today, most software professionals recognize the fallacy of the myths just de scribed. Recognition of software realities is the fi rst step toward formulation of practical solutions for software engineering.

8 Many software engineers have adopted an “agile” approach that accommodates change incre mentally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 5.

2.5 How It All Starts

Every software project is precipitated by some business need—the need to cor rect a defect in an existing application; the need to adapt a “legacy system” to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

S H

How a Project Starts ow a Project Starts

The scene: Meeting room at CPI Corporation, a (fi ctional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, execu tive vice president, business development

The conversation:

Joe: Okay, Lee, what’s this I hear about your folks de veloping a what? A generic universal wireless box?

Lee: It’s pretty cool . . . about the size of a small match book . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11n wireless protocol. It allows us to access the device’s output without wires. We think it’ll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as fl at as they’ve been this year, we need something new. Lisa and I have been doing a little market research, and we think we’ve got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It’s a whole new generation of what we call “home management products.” We call ‘em *SafeHome.* They use the new wireless interface, provide homeown ers or small-businesspeople with a system that's con trolled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low man ufacturing cost. Most hardware is off the shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs and tablets have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer app. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as $30 to $40 million in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* prod uct line. The engineering effort will succeed only if *SafeHome* software succeeds.

9The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are fi ctitious, but the situations and problems are real.

The market will accept the product only if the software embedded within it prop erly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

2.6 S UMMARY

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates fi ve framework activities—communication, plan ning, modeling, construction, and deployment—that are applicable to all soft ware projects. Software engineering practice is a problem-solving activity that follows a set of core principles.

A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies re quired to build it grows. As you learn more about software engineering, you'll begin to understand why these myths should be debunked whenever they are encountered.

Problems and Points to Ponder

2.1. Figure 2.1places the three software engineering layers on top of a layer entitled "A quality focus." This implies an organizational quality program such as total quality man agement. Do a bit of research and develop an outline of the key tenets of a total quality management program.

2.2. Is software engineering applicable when WebApps are built? If so, how might it be mod ifi ed to accommodate the unique characteristics of WebApps?

2.3. As software becomes more pervasive, risks to the public (due to faulty programs) be come an increasingly signifi cant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm, either economic or human.

2.4. Describe a process framework in your own words. When we say that framework activ ities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.

2.5. Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities?

2.6. Add two additional myths to the list presented in Section 2.4. Also state the reality that accompanies the myth.

FURTHER READINGS AND INFORMATION SOURCES

The current state of the software engineering and the software process can best be determined from publications such as *IEEE Software, IEEE Computer, CrossTalk,* and *IEEE Transactions on Software Engineering.* Industry periodicals such as *Application Development Trends* and *Cutter IT Journal* often contain articles on software engineering topics. The discipline is "summarized" every year in the *Proceeding of the International Conference*

on Software Engineering, sponsored by the IEEE and ACM, and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes,* and *Annals of Software Engineering.* Tens of thousands of Web pages are dedicated to software engineering and the software process.

Many books addressing the software process and software engineering have been published in recent years. Some present an overview of the entire process, while others delve into a few important topics to the exclusion of others. Among the more popular offerings (in addition to this book!) are

SWEBOK: Guide to the Software Engineering Body of Knowledge,[10] IEEE, 2013, see: http://www.computer.org/portal/web/swebok

Andersson, E., et al., *Software Engineering for Internet Applications,* MIT Press, 2006.

Braude, E., and M. Bernstein, *Software Engineering: Modern Approaches,* 2nd ed., Wiley, 2010.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Hussain, S., *Software Engineering,* I K International Publishing House, 2013.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2nd ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* 3rd ed., Springer, 2010. Pfleeger,

S., *Software Engineering: Theory and Practice,* 4th ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering,* 8th ed., McGraw-Hill, 2010.

Sommerville, I., *Software Engineering,* 9th ed., Addison-Wesley, 2010.

Stober, T., and U. Hansmann, *Agile Software Development: Best Practices for Large Development Projects,* Springer, 2009.

Tsui, F., and O. Karam, *Essentials of Software Engineering,* 2nd ed.,Jones & Bartlett Publishers, 2009.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007), Richardson and Gwaltney (*Ship it! A Practical Guide to Successful Software Projects,* Pragmatic Bookshelf, 2005), and Humble and Farley (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010) present a broad collection of useful guidelines that are applicable to the deployment activity.

Many software engineering standards have been published by the IEEE, ISO, and their standards organizations over the past few decades. Moore (*The Road Map to Software Engineering: A Standards-Based Guide,* IEEE Computer Society Press [Wiley], 2006) provides a useful survey of relevant standards and how they apply to real projects.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**

10 Available free of charge at **<http://www.computer.org/portal/web/swebok/htmlformat>**

PART

THE SOFTWARE PROCESS

In this part of *Software Engineering: A Practitioner's Ap proach* you'll learn about the process that provides a frame work for software engineering practice. These questions are addressed in the chapters that follow:

- What is a software process?
- What are the generic framework activities that are present in every software process?
- How are processes modeled and what are process patterns?
- What are the prescriptive process models and what are their strengths and weaknesses?
- Why is *agility* a watchword in modern software engineering work?
- What is agile software development and how does it differ from more traditional process models?

Once these questions are answered you'll be better prepared to understand the context in which software engineering practice is applied.

CHAPTER 3 SOFTWARE PROCESS STRUCTURE

KEY CONCEPTS

In a fascinating book that provides an economist's view of software and soft ware engineering, Howard Baetjer Jr. [Bae98] comments on the software process:

> Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.

Indeed, building computer software is an iterative social learning process, and the outcome, something that Baetjer would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

What is it? When you work to build a product or system, it's im portant to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a "soft ware process."

Who does it? Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defi ning, building, and testing it.

Why is it important? Because it provides sta bility, control, and organization to an activity that can, if left uncontrolled, become quite cha otic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work prod ucts that are appropriate for the project team and the product that is to be produced.

What are the steps? At a detailed level, the process that you adopt depends on the soft ware that you're building. One process might be appropriate for creating software for an air craft avionics system, while an entirely differ ent process would be indicated for the creation of a website.

What is the work product? From the point of view of a software engineer, the work prod ucts are the programs, documents, and data that are produced as a consequence of the ac tivities and tasks defi ned by the process.

How do I ensure that I've done it right? There are a number of software process assess ment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the effi cacy of the pro cess that you use.

But what exactly is a software process from a technical point of view? Within the context of this book, we defi ne a *software process* as a framework for the activities, actions, and tasks that are required to build high-quality software. Is "process" synonymous with "software engineering"? The answer is yes and no. A software process defi nes the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

More important, software engineering is performed by creative, knowledge able people who should adapt a mature software process so that it is appropriate for the products that they build and the demands of their marketplace.

3.1 A Generic Process Model

In Chapter 2, a process was defi ned as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defi nes their relationship with the process and with one another.

The software process is represented schematically in Figure 3.1 . Referring to the fi gure, each framework activity is populated by a set of software engineering

The hierarchy of technical work within the software process is activities, encompass ing actions, populated by tasks.

actions. Each software engineering action is defi ned by a *task set* that identifi es the work tasks that are to be completed, the work products that will be pro duced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As we discussed in Chapter 2, a generic process framework for software engi neering defi nes fi ve framework activities— **communication, planning, modeling, construction,** and **deployment**. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, confi guration man agement, technical reviews, and others—are applied throughout the process.

?

What is process fl ow?

You should note that one important aspect of the software process has not yet been discussed. This aspect—called *process fl ow*—describes how the frame work activities and the actions and tasks that occur within each framework ac tivity are organized with respect to sequence and time and is illustrated in Figure 3.2 .

A *linear process fl ow* executes each of the fi ve framework activities in se quence, beginning with communication and culminating with deployment (Fig ure 3.2a) . An *iterative process fl ow* repeats one or more of the activities before proceeding to the next (Figure 3.2b) . An *evolutionary process fl ow* executes the activities in a "circular" manner. Each circuit through the fi ve activities leads to a more complete version of the software (Figure 3.2c) . A *parallel process fl ow* (Figure 3.2d)executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with con struction of another aspect of the software).

A software process

Software process

framework Process framework Umbrella

activities

framework activity # 1

software engineering action #1.1

work tasks

Task sets

work products

quality assurance

points project

milestones

software engineering action #1.k

work tasks

Task sets

work products

quality assurance

points project

milestones

framework activity # n

software engineering action

#n.1 work tasks

Task sets

work products

quality assurance

points project

milestones

software engineering action

#n.m work tasks

Task sets

work products

quality assurance

points project

milestones

3.2 Defining a Framework Activity

uote:

"If the process is right, the results will take care of themselves."

Takashi Osada

Although we have described fi ve framework activities and provided a basic defi - nition of each in Chapter 2, a software team would need signifi cantly more infor mation before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

Process fl ow

Communication Planning Modeling (a) Linear process flow

Construction Deployment

Communication Planning Modeling Construction Deployment

(b) Iterative process flow

Planning

Modeling

Communication

Deployment Construction

Increment released

(c) Evolutionary process flow

Communication Planning

Modeling Time

Construction Deployment

(d) Parallel process flow

? How does a framework activity change as the nature of the project changes?

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the **communication** activity might encompass little more than a phone call or email with the appropriate stake holder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and develop notes.

3. Organize notes into a brief written statement of requirements.
4. Email to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics.

with a different set of (sometime confl icting) requirements, the communication activity might have six distinct actions (described in Chapter 8): *inception, elici tation, elaboration, negotiation, specifi cation,* and *validation.* Each of these soft ware engineering actions would have many work tasks and a number of distinct work products.

3.3 Identifying a Task Set

Referring again to Figure 3.1 , each software engineering action (e.g., *elicitation,* an action associated with the **communication** activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

Task Set

A task set defi nes the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more com monly called "requirements gathering") is an important software engineering action that occurs during the **communication** activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project. 2. Invite all stakeholders to an informal meeting. 3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a fi nal list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a differ ent task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project. 2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specifi - cation meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refi ne user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be deliv ered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve "requirements gath ering," but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each action and still main tain quality and agility.

You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specifi c needs of the software project and the char acteristics of the project team.

3.4 Process Patterns

?

What is a process

Every software team encounters problems as it moves through the software pro cess. It would be useful if proven solutions to these problems were readily avail pattern?

uote:

"The repetition of patterns is quite a different thing than the repetition of parts. Indeed, the different parts will be unique because the patterns are the same."

Christopher Alexander

A pattern template provides a consistent means for describing a pattern.

able to the team so that the problems could be addressed and resolved quickly. A *process pattern*[1] describes a process-related problem that is encountered during software engineering work, identifi es the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template [Amb98]—a consistent method for describing problem solutions within the con

text of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project. Patterns can be defi ned at any level of abstraction. [2] In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating). Ambler [Amb98] has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specifi ed. Ambler [Amb98] suggests three types:

1. **Stage pattern**—defi nes a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates mul tiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication**. This pattern would incorporate the task pattern **RequirementsGathering** and others.

1 A detailed discussion of patterns is presented in Chapter 11

2 Patterns are applicable to many software engineering activities. Analysis, design, and testing patterns are discussed in Chapters 11, 13, 15, 16, and 20. Patterns and "antipatterns" for project management activities are discussed in Part 4 of this book.

2. **Task pattern**—defi nes a problem associated with a software engineer ing action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. **Phase pattern**—defi ne the sequence of framework activities that occurs within the process, even when the overall fl ow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping**. [3]

Initial Context. Describes the conditions under which the pattern applies.

uote:

"We think that software developers are missing a vital truth: most organizations don't know what they do. They think they know, but they don't know."

Tom DeMarco

Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists?

For example, the **Planning** pattern (a stage pattern) requires that (1) custom ers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns [specifi ed] for the **Communication** pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

Problem. The specifi c problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modifi ed as a consequence of the initiation of the pattern. It also describes how software engineering information or project informa tion that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encom

passes the task patterns: **ProjectTeam, CollaborativeGuidelines, ScopeIsolation**, **RequirementsGathering, ConstraintDescription,** and **ScenarioCreation.**

Known Uses and Examples. Indicate the specifi c instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the **Deployment** activity is under way.

3 These phase patterns are discussed in Chapter 4.

Comprehensive resources on process patterns can be found at www.ambysoft.com/ processPatternsPage .html .

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hier archical process description that begins at a high level of abstraction (a phase pattern). The description is then refi ned into a set of stage patterns that describe framework activities and are further refi ned in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the defi nition of process variants—that is, a customized process model can be defi ned by a software team using the patterns as building blocks for the process model.

An Example Process Pattern

The following abbreviated process pattern de scribes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specifi c software requirements.

Pattern Name. RequirementsUnclear

Intent. This pattern describes an approach for building a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

Type. Phase pattern.

Initial Context. The following conditions must be met prior to the initiation of this pattern: (1) stakeholders have been identifi ed; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identifi ed by stakeholders; (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem. Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem to be solved, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solution. A description of the prototyping process would be presented here and is described later in Section 4.1.3.

Resulting Context. A software prototype that identifi es basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

Related Patterns. The following patterns are related to this pattern: **CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.**

Known Uses and Examples. Prototyping is recommended when requirements are uncertain.

3.5 Process Assessment and Improvement

The existence of a software process is no guarantee that software will be de livered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapter 19). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself can be assessed to

ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. [4]

Assessment attempts to understand the current state of the software process with the intent of improving it.

uote:

"Software organizations have exhibited signifi cant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

A number of different approaches to software process assessment and im provement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement (SCAMPI)— provides a fi ve-step process assessment model that incorporates fi ve phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

SPICE (ISO/IEC15504)—a standard that defi nes a set of requirements for software process assessment. The intent of the standard is to assist organi zations in developing an objective evaluation of the effi cacy of any defi ned software process [ISO08].

ISO 9001:2000 for Software—a generic standard that applies to any organiza tion that wants to improve the overall quality of the products, systems, or ser vices that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

A more detailed discussion of software assessment and process improvement methods is presented in Chapter 37.

3.6 SUMMARY

A generic process model for software engineering encompasses a set of frame work and umbrella activities, actions, and work tasks. Each of a variety of pro cess models can be described by a different process fl ow—a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

PROBLEMS AND POINTS TO PONDER

3.1. In the introduction to this chapter Baetjer notes: "The process provides interaction be tween users and designers, between users and evolving tools, and between designers and evolving tools [technology]." List fi ve questions that (1) designers should ask users, (2) users should ask designers, (3) users should ask themselves about the software product that is to be built, (4) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

4 The SEI's CMMI [CMM07] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

3.2. Discuss the differences among the various process fl ows described in Section 3.1. Can you identify types of problems that might be applicable to each of the generic fl ows described?

3.3. Try to develop a set of actions for the communication activity. Select one action and defi ne a task set for it.

3.4. A common problem during **communication** occurs when you encounter two stakehold ers who have confl icting ideas about what the software should be. That is, you have mutually confl icting requirements .Develop a process pattern (this would be a stage pattern) using the template presented in Section 3.4 that addresses this problem and suggest an effective approach to it.

FURTHER READINGS AND INFORMATION SOURCES

Most software engineering textbooks consider process models in some detail. Books by Sommerville (*Software Engineering,* 9th ed., Addison-Wesley, 2010), Schach (*Object Oriented and Classical Software Engineering,* 8th ed., McGraw-Hill, 2010) and Pfl eeger and Atlee (*Software Engineering: Theory and Practice,* 4th ed., Prentice Hall, 2009) consider traditional paradigms and discuss their strengths and weaknesses. Munch and his col leagues (*Software Process Defi nition and Management,* Springer, 2012) present a software and systems engineering view of the process and the product. Glass (*Facts and Fallacies of Software Engineering,* Prentice Hall, 2002) provides an unvarnished, pragmatic view of the software engineering process. Although not specifi cally dedicated to process, Brooks (*The Mythical Man-Month,* 2nd ed., Addison-Wesley, 1995) presents age-old project wisdom that has everything to do with process.

Firesmith and Henderson-Sellers (*The OPEN Process Framework: An Introduction,* Addison-Wesley, 2001) present a general template for creating "fl exible, yet discipline software processes" and discuss process attributes and objectives. Madachy (*Software Process Dynamics,* Wiley-IEEE, 2008) discusses modeling techniques that allow the inter related technical and social elements of the software process to be analyzed. Sharpe and McDermott (*Workfl ow Modeling: Tools for Process Improvement and Application Develop ment,* 2nd ed., Artech House, 2008) present tools for modeling both software and business processes.

A wide variety of information sources on software engineering and the software pro cess are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/ pressman**

CHAPTER

PROCESS 4 MODELS

KEY CONCEPTS

Process models were originally proposed to bring order to the chaos of

software development. History has indicated that these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.

However, software engineering work and the products that are produced re main on "the edge of chaos."

In an intriguing paper on the strange relationship between order and chaos in the software world, Nogueira and his colleagues [Nog00] state

> The edge of chaos is defi ned as "a natural state between order and chaos, a grand compromise between structure and surprise." [Kau95] The edge of chaos can be visualized as an unstable, partially structured state . . . It is unstable because it is constantly attracted to chaos or to absolute order.

> We have the tendency to think that order is the ideal state of nature. This could be a mistake. Research . . . supports the theory that operation away from equilibrium generates creativity, self-organized processes, and increasing re turns [Roo96]. Absolute order means the absence of variability, which could be an

What is it? A process model pro vides a specifi c roadmap for soft ware engineering work. It defi nes the fl ow of all activities, actions and tasks, the degree of iteration, the work prod ucts, and the organization of the work that must be done.

Who does it? Software engineers and their managers adapt a process model to their needs and then follow it. In addition, the peo ple who have requested the software have a role to play in the process of defi ning, build ing, and testing it.

Why is it important? Because process pro vides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the proj ect team and the product that is to be produced.

What are the steps? The process model pro vides you with the "steps" you'll need to per form disciplined software engineering work.

What is the work product? From the point of view of a software engineer, the work prod uct is a customized description of the activities and tasks defi ned by the process.

How do I ensure that I've done it right? There are a number of software process assess ment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the effi cacy of the pro cess that you use.

The purpose of process models is to try to reduce the chaos pres ent in developing new software products.

advantage under unpredictable environments. Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossi ble. Lack of structure does not always mean disorder.

The philosophical implications of this argument are signifi cant for software engineering. Each process model described in this chapter tries to strike a bal ance between the need to impart order in a chaotic world and the need to be adaptable when things change constantly.

4.1 Prescriptive Process Models

An award-winning "pro cess simulation game" that includes most important prescriptive process models can be found at:

http://www.ics .uci.edu/~emilyo/ SimSE/ downloads.html .

Prescriptive process models defi ne a prescribed set of process elements and a predictable process work fl ow.

A *prescriptive process model*[1] strives for structure and order in software devel opment. Activities and tasks occur sequentially with defi ned guidelines for prog ress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we examine the pre scriptive process approach in which order and project consistency are domi nant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work prod ucts, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process fl ow (also called a *work fl ow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activi ties described in Chapters 2 and 3, but each applies a different emphasis to these activities and defi nes a process fl ow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

4.1.1 The Waterfall Model

There are times when the requirements for a problem are well understood— when work fl ows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defi ned adaptations or enhancements to an existing system must be made (e.g., an adaptation to ac counting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defi ned and reasonably stable.

1 Prescriptive process models are sometimes referred to as "traditional" process models.

The

waterfall model

Communication
project initiation
requirements
gathering

Planning estimating
scheduling
tracking

Modeling analysis
design

Construction
code test

Deployment
delivery support
feedback

The V-model illustrates how verifi cation and validation actions are associated with earlier engineering actions.

The *waterfall model,* sometimes called the *classic life cycle*, suggests a system atic, sequential approach [2] to software development that begins with customer specifi cation of requirements and progresses through planning, modeling, con struction, and deployment, culminating in ongoing support of the completed software (Figure 4.1).

A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 4.2 , the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refi ned into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models cre ated as the team moves down the left side. [3] In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verifi cation and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. How ever, over the past four decades, criticism of this process model has caused even ardent supporters to question its effi cacy [Han95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

?

Why does the waterfall model sometimes fail?

1. Real projects rarely follow the sequential fl ow that the model proposes. Although the linear model can accommodate iteration, it does so indi rectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often diffi cult for the customer to state all requirements explicitly. The waterfall model requires this and has diffi culty accommodating the natu ral uncertainty that exists at the beginning of many projects.

2 Although the original waterfall model proposed by Winston Royce [Roy70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

3 A detailed discussion of quality assurance actions is presented in Part 3 of this book.

The V-model

testing

Requirements
modeling

Integration
testing

Architectural
design

Unit
testing

Component
design

Code
generation
Acceptance
testing

Executable
software

System

uote:

“Too often,
software work
follows the fi rst law of bicycling: No matter where you’re going, it’s
uphill and against the wind.”

Author unknown

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to “blocking states” in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process. Today, software work is fast paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fi xed and work is to proceed to com pletion in a linear manner.

4.1.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well defi ned, but the overall scope of the development effort precludes a purely

The incremen
tal model

Deployment (delivery, feedback)

increment # 2

Communication

Planning

increment # 1

Modeling (analysis, design)

Construction (code, test)

increment # *n*

delivery of
2nd increment
delivery of *n*th increment

delivery of

The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

Your customer de mands delivery by a date that is impossible to meet. Suggest deliv ering one or more in crements by that date and the rest of the software (additional increments) later.

1st increment

Project Calendar Time

linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refi ne and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The incremental model combines the elements' linear and parallel process fl ows discussed in Chapter 3. Referring to Figure 4.3 , the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93].

For example, word-processing software developed using the incremental par adigm might deliver basic fi le management, editing, and document production functions in the fi rst increment; more sophisticated editing and document pro duction capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth incre ment. It should be noted that the process fl ow for any increment can incorporate the prototyping paradigm discussed in the next subsection.

When an incremental model is used, the fi rst increment is often a *core prod uct.* That is, basic requirements are addressed but many supplementary fea tures (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/ or evaluation, a plan is developed for the next increment. The plan addresses the modifi cation of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is re peated following the delivery of each increment, until the complete product is produced.

Evolutionary process models produce an increasingly more complete version of the software with each iteration.

uote:

"Plan to throw
one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers."

Frederick P. Brooks

When your customer has a legitimate need, but is clueless about the details, develop a prototype as a fi rst step.

4.1.3 Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defi ned. In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that grows and changes.

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, we present two common

evolutionary process models.

Prototyping . Often, a customer defi nes a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the effi ciency of an algorithm, the adapt ability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 4.4) begins with communication. You meet with other stakeholders to defi ne the overall objectives for the software, identify whatever requirements are known, and outline areas where further defi nition is mandatory. A prototyping iteration is planned quickly, and mod

eling (in the form of a "quick design") occurs. A quick design focuses on a rep resentation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refi ne require ments. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software require ments. If a working prototype is to be built, you can make use of existing pro gram fragments or apply tools that enable working programs to be generated quickly.

The prototyp
ing paradigm

Communication

Quick design

Deployment
Delivery
& Feedback
Quick plan

Construction
of
prototype

Modeling

Resist pressure to extend a rough proto type into a production product. Quality almost always suffers as a result.

But what do you do with the prototype when it has served the purpose de scribed earlier? Brooks [Bro95] provides one answer:

In most projects, the fi rst system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as “the fi rst system.” The one that Brooks recom mends you throw away. But this may be an idealized view. Although some pro totypes are built as “throwaways,” others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something im mediately. Yet, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be re built so that high levels of quality can be maintained, stakeholders cry foul and demand that “a few fi xes” be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operat ing system or programming language may be used simply because it is

S H

Selecting a Process Model, Part 1 electing a Process Model, Part 1

The scene: Meeting room for the software engineering group at CPI Corporation, a (fi ctional) company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

The conversation:

Lee: So let's recapitulate. I've spent some time dis cussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply defi ne the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

Doug: Seems like we've been pretty disorganized in our approach to software in the past.

Ed: I don't know, Doug, we always got product out the door.

Doug: True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

Jamie: Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particu larly if we have a very tight time line.

Doug (smiling): I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

Jamie (with a frown): My job is to build computer programs, not push paper around.

Doug: Give it a chance before you go negative on me. Here's what I mean. (Doug proceeds to describe the process framework described in Chapter 3 and the prescriptive process models presented to this point.)

Doug: So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

Vinod: Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

Doug: I agree.

Ed: That prototyping approach seems okay. A lot like what we do here anyway.

Vinod: That's a problem. I'm worried that it doesn't provide us with enough structure.

Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

available and known; an ineffi cient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to defi ne the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defi ning requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

The Spiral Model. Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative na ture of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more

complete versions of the software. Boehm [Boe01a] describes the model in the following manner:

The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally

The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

A typical spiral model

growing a system's degree of defi nition and implementation while decreasing its de gree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Planning
estimation
scheduling
risk analysis

Communication

Using the spiral model, software is developed in a series of evolutionary re leases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defi ned by the soft ware engineering team. For illustrative purposes, we use the generic framework activities discussed earlier. [4] Each of the framework activities represent one seg ment of the spiral path illustrated in Figure 4.5 . As this evolutionary process be gins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 35) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral— are noted for each evolutionary pass.

analysis
design

Start

Construction **Deployment**

Modeling

delivery feedback

code test

4 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98].

Useful information about the spiral model can be obtained at: **www.sei.cmu.edu/publications/ documents/00.reports/00sr008. html** .

If your management demands fi xed-budget development (gen erally a bad idea), the spiral can be a problem. As each circuit is completed, project cost is revisited and revised.

uote:

"I'm only this far and only tomorrow leads my way."

Dave Matthews Band

The fi rst circuit around the spiral might result in the development of a prod uct specifi cation; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the fi rst circuit around the spiral might represent a "concept develop ment project" that starts at the core of the spiral and continues for multiple iter ations [5] until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolution ary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach sug gested by the classic life cycle but incorporates it into an iterative framework that more realistically refl ects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly ap plied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be diffi cult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and man aged, problems will undoubtedly occur.

4.1.4 Concurrent Models

The *concurrent development model,* sometimes called *concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity

5 The arrows pointing inward along the axis separating the *deployment* region from the *commu nication* region indicate a potential for local iteration along the same spiral path.