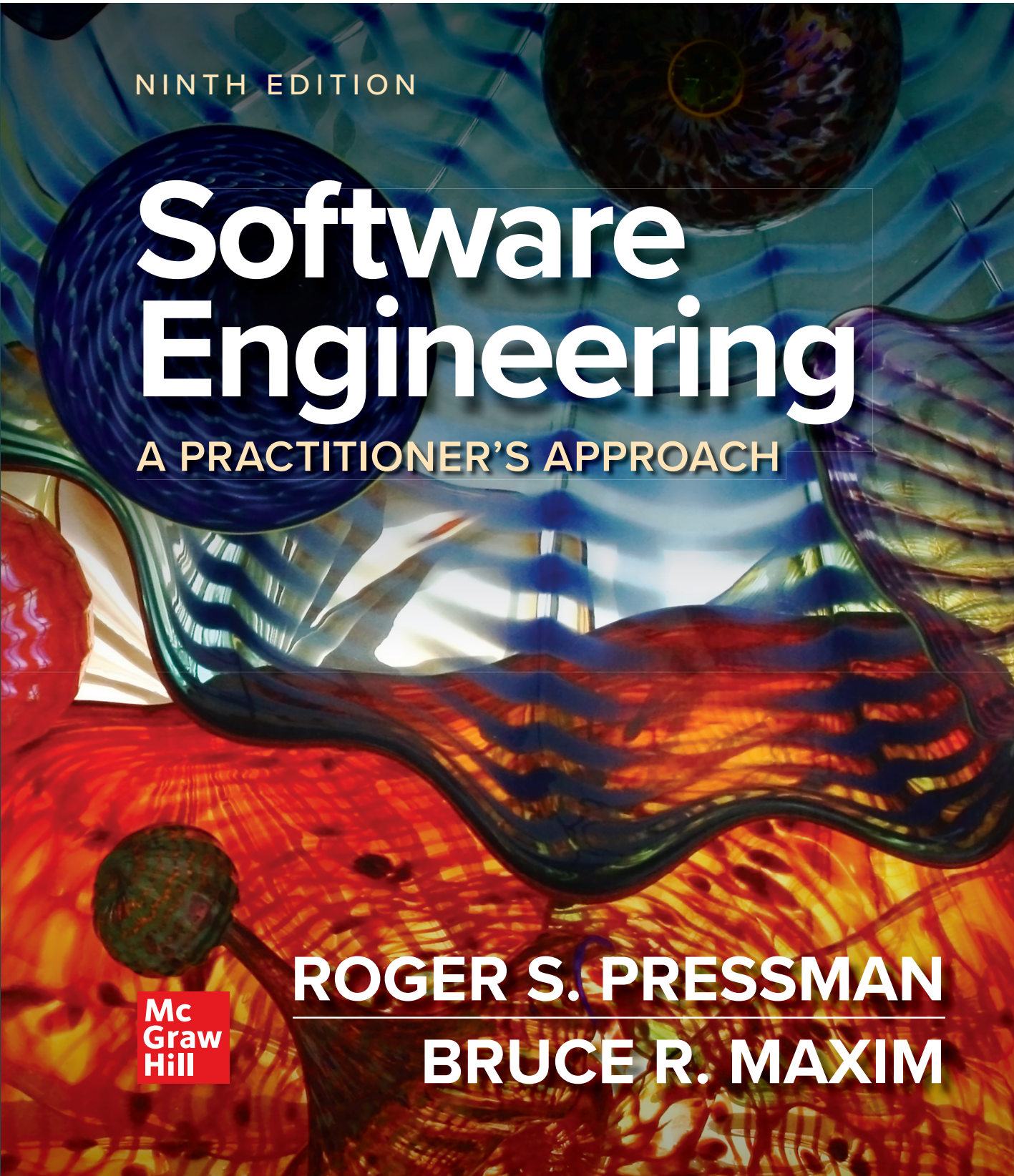
This International Student Edition is for use outside of the U.S.



Software Engineering

A PRACTITIONER'S APPROACH

Software Engineering

A PRACTITIONER'S APPROACH

NINTH EDITION

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

Mc Graw Hill

SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 24 23 22 21 20 19

ISBN 978-1-260-54800-6
MHID 1-260-54800-7

Cover Image: ©*R.L. Hausdorf/Shutterstock*

mheducation.com/highered

To Barbara, Matt, Mike,
Shiri, Adam, Lily,
and Maya.

—Roger S. Pressman

To my family who
support me in all
that I do.

—Bruce R. Maxim

About the Authors

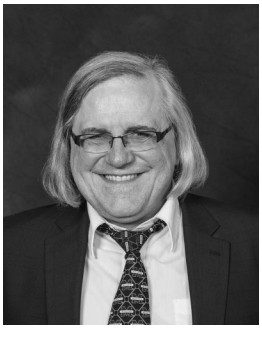
Courtesy of Roger Pressman

Roger S. Pressman is an internationally recognized consultant and author in software engineering. For almost five decades, he has worked as a software engineer, a manager, a professor, an author, a consultant, and an entrepreneur.

Dr. Pressman was president of R. S. Pressman & Associates, Inc., a consulting firm that specialized in helping companies establish effective software engineering strategies. Over the years he developed a set of techniques and tools that improved software engineering practice. He is also the founder and chief technology officer of EVANNEX®, an automotive aftermarket company that specializes in the design and manufacture of accessories for the Tesla line of electric vehicles.

Dr. Pressman is the author of ten books, including two novels, and many technical 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 Software Engineering and at many other industry meetings. He has been a member of the ACM, IEEE, Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.



Michigan Creative/
UM-Dearborn

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, user experience design, serious game development, artificial intelligence, and engineering education.

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

Dr. Maxim's professional experience includes managing research information 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 officer for a game development company.

Dr. Maxim was the recipient of several distinguished teaching awards, a distinguished community service award, and a distinguished faculty governance 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.

Contents at a Glance

Table of Contents

PREFACE

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 nearly four 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 or coder 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.

NEW TO THE NINTH EDITION

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

The ninth 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.

Readers of the past few editions of *Software Engineering: A Practitioner's Approach* will note that the ninth edition has actually been reduced in page length. Our goal was concision, making the book stronger from a pedagogical viewpoint and less daunting for the reader who desires to journey through the entire book. An anecdote, attributed to Blaise Pascal, the famous mathematician and physicist, goes like this: In writing a overly long letter to a friend, Pascal ended with this sentence. "I wanted to write you a shorter letter, but I didn't have the time." As we worked on concision for the ninth edition, we came to appreciate Pascal's words.

The 30 chapters of the ninth 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 Software Process,* presents a variety of different views of software process, considering several important process models and frameworks that allow us to address 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 mobility computing applications are also considered. The coverage of user experience design has been expanded in this section. Part 3, *Quality and Security,* 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, we present software security practices that can be inserted into incremental software development models. 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. Boxed features are included throughout the book to present the trials and tribulations of a (fictional) software team and to provide supplementary materials about methods and tools that are relevant to chapter topics.

The five-part organization of the ninth 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 five parts. A software engineering survey course would select chapters from all five parts. A software engineering course that emphasizes 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 ninth edition in this way,

we have attempted to provide an instructor with a number of teaching options. In every case the content of the ninth edition is complemented by the following elements of the *SEPA, 9/e Support System.*

Additional Resources

A wide variety of resources can be accessed through the instructor website, including an extensive online learning center encompassing problem solutions, a variety of Web-based resources with software engineering checklists, an evolving collection of "tiny tools," and comprehensive case studies. *Professional Resources* provide several hundred categorized web references which allow students to explore software engineering in greater detail, along with a reference library with links to several hundred downloadable references providing an in-depth source of advanced software engineering information. Additionally, a complete online *Instructor's Guide* and supplementary teaching materials along with several hundred PowerPoint slides that may be used for lectures are included.

The *Instructor's Guide for Software Engineering: A Practitioner's Approach* presents suggestions for conducting various types of software engineering courses, recommendations 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.

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

Bruce Maxim has taken the lead in developing new content for the ninth edition of *Software Engineering: A Practitioner's Approach,* while Roger Pressman has served as editor-in-chief as well as providing contributions in select circumstances.

Acknowledgments Special thanks go to Nancy Mead from Software Engineering Institute at Carnegie Mellon University who wrote the chapter on software security engineering; 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; Dale Skrien of Colby College who developed the UML tutorial in Appendix 1; William Grosky of the University of Michigan–Dearborn who developed the overview of data science in Appendix 2 with his student Terry Ruas; and our Australian colleague Margaret Kellow for updating the pedagogical Web materials that accompany this book. In addition, we would like to thank Austin Krauss for providing insight into software development in the video game industry from his perspective as a senior software engineer.

Special Thanks BRM: I am grateful to have had the opportunity to work with Roger on the ninth edition of this book. During the time I have been working on this book, my son, Benjamin, has become a software engineering manager and my daughter, Katherine, used her art background to create the figures that appear in the book chapters. I am quite pleased to see the adults they have become and enjoy my time with their children

(Isla, Emma, and Thelma). I am very grateful to my wife, Norma, for her enthusiastic support as I filled my free time 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. After many years of following our own professional paths, the three of us now work together in a business that we founded in 2012. Nothing has filled me with more pride. Both of my sons now have children of their own, Maya and Lily, who start still another generation. Finally, to my wife, Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of "the book."

Bruce R. Maxim
Roger S. Pressman

Affordability & Outcomes = Academic Freedom!

You deserve choice, flexibility and control. You know what's best for your students and selecting the course materials that will help them succeed should be in your hands.

That's why providing you with a wide range of options that lower costs and drive better outcomes is our highest priority.

Mc Graw Hill connect®

Students—study more efficiently, retain more and achieve better outcomes. Instructors—focus on what you love—teaching.

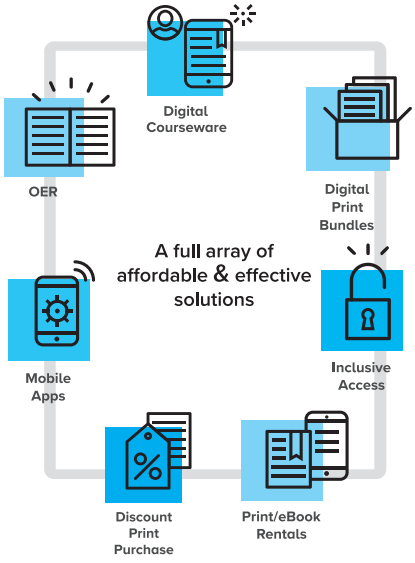
Laptop: McGraw-Hill Education

They'll thank you for it.

Study resources in Connect help your students be better prepared in less time. You can transform your class time from dull definitions to dynamic discussion. Hear from your peers about the benefits of Connect at **www.mheducation.com/highered/connect/smartbook**

Make it simple, make it affordable.

Connect makes it easy with seamless integration using any of the major Learning Management Systems—Blackboard®, Canvas, and D2L, among others—to let you organize your course in one convenient location. Give your students access to digital materials at a discount with our inclusive access program. Ask your McGraw-Hill representative for more information.



Learning for everyone.

McGraw-Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services office and ask them to email accessibility@mheducation.com, or visit **www.mheducation.com/about/accessibility.html** for more information.

Learn more at: www.mheducation.com/realvalue

Rent It

Affordable print and digital rental options through our partnerships with leading textbook distributors including Amazon, Barnes & Noble, Chegg, Follett, and more.

Go Digital

A full and flexible range of affordable digital solutions ranging from Connect, ALEKS, inclusive access, mobile apps, OER and more.

Get Print

Students who purchase digital materials can get a loose-leaf print version at a significantly reduced rate to meet their individual preferences and budget.

CHAPTER

1 SOFTWARE AND SOFTWARE ENGINEERING

As he finished showing me the latest build of one of the world's most popular first-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 commonplace among his co-workers.

KEY CONCEPTS

QUICK LOOK

What is it? Computer software is a work product that software professionals build and then support over many years. These work products include programs that execute within computers of any size and architecture. Software engineering encompasses a process, a collection of methods (practice), and an array of tools that allow professionals to build high-quality computer software.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it. Software engineers apply the software engineering process.

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

What are the steps? You build computer software like you build any successful product, 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.

What is the work product? From the software engineer's point of view, the work product is the set of programs, content (data), and other work products that support computer software. But from the user's point of view, the work product is a tool or product that somehow makes the user's world better.

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

"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 developers 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 five months, and we've still got a lot of work to do."

He had responsibility for game play and artificial 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.

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."

*****

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 consequences. Sixty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the media); that software would be the driving force behind the personal computer revolution; that software applications would be purchased by

consumers using their mobile devices; 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 influential than all industrial-era companies; or 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.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and support high-quality computer programs. Some of these technologies are targeted at a specific 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.*

To build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- Software has become deeply embedded in virtually every aspect of our lives. The number of people who have an interest in the features and functions provided by a specific application[1] has grown dramatically. *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 increasingly complex with each passing year. Large teams of people now create computer programs. Sophisticated software that was once implemented in a predictable, self-contained computing environment is now embedded inside everything from consumer electronics to medical devices to autonomous vehicles. *Design has become a pivotal activity.*
- 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 consequences. *Software should exhibit high quality.*
- As the perceived value of a specific 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. *Software should be maintainable.*

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

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

1.1 The Nature of Software

Today, software takes on a dual role. It is a product, and 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 that are accessible by local hardware. Whether it resides within a mobile device, on the desktop, in the cloud, or within a mainframe computer or autonomous machine, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as an augmented-reality representation derived from data acquired from dozens of independent sources and then overlaid on the real world. As the vehicle used to deliver a product, software acts as the basis for the control of the computer (operating systems), the communication 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 financial 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 networks (e.g., the Internet); and provides the means for acquiring information in all 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 significant change over the last 60 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, 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 economies 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 modern computer-based systems are built:[2]

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find 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 difficulty in measuring progress as software is being developed and maintained?

2 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 asking '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."

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

1.1.1 Defining 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 desired features, function, and performance; (2) data structures that enable the programs 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 definitions could be offered. But a more formal definition probably won't measurably improve your understanding. 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 fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, 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,

FIGURE 1.1

Failure curve for hardware

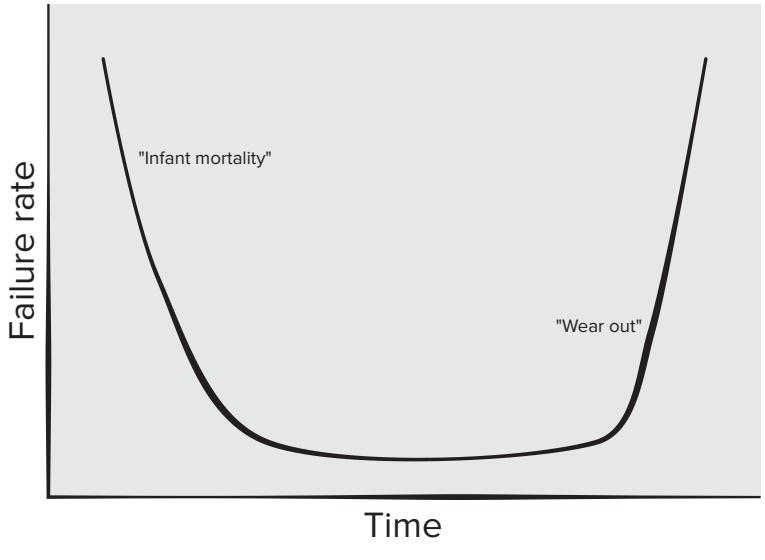
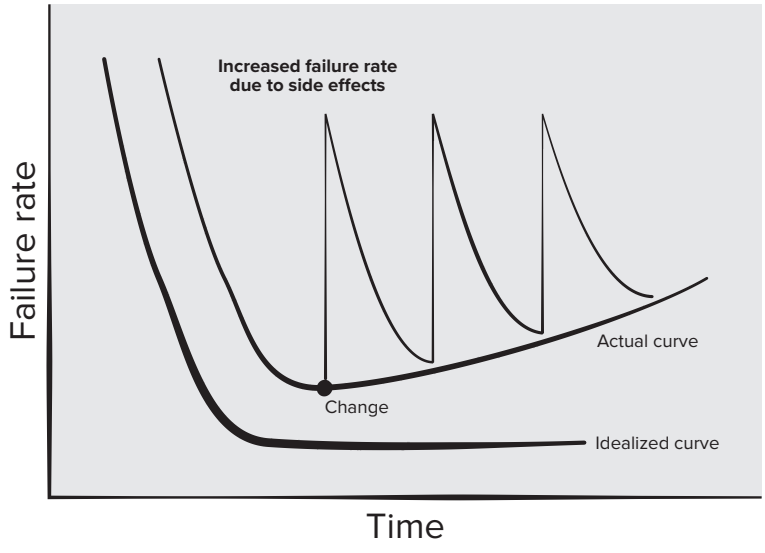


FIGURE 1.2

Failure curves for software



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 hardware 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 flattens as shown. The idealized curve is a gross oversimplification 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,[3] software will undergo change. As changes are 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 software. 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.

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

1.1.2 Software Application Domains

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

System software. A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,[4] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

Application software. Stand-alone programs that solve a specific 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" or data science programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, from computer-aided design to consumer spending habits, and 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 functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software. Composed of reusable components and designed to provide specific capabilities for use by many different customers. It may focus on a limited and esoteric marketplace (e.g., inventory control products) or attempt to address the mass consumer market.

Web/mobile applications. This network-centric software category spans a wide array of applications and encompasses browser-based apps, cloud computing, service-based computing, and software that resides on mobile devices.

Artificial intelligence software. Makes use of heuristics[5] to solve complex problems that are not amenable to regular computation or straightforward analysis. Applications within this area include robotics, decision-making systems, pattern recognition (image and voice), machine learning, theorem proving, and game playing.

Millions of software engineers worldwide are hard at work on software projects 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.

---

4 Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

5 The use of heuristics is an approach to problem solving that employs a practical method or "rule of thumb" not guaranteed to be perfect, but sufficient for the task at hand.

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. 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 modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

These changes may create an additional side effect that is often present in legacy software—*poor quality*.[6] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, and a poorly managed change history. The list can be quite long. And yet, these systems often support "core functions and are indispensable to the business." What to do?

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

- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it work with other more modern systems or databases.
- The software must be re-architected to make it viable within an evolving computing environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable in the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution; that is, the notion that software systems change continually, new software systems can be built from the old ones, and . . . all must interact and cooperate with each other" [Day99].

1.2 Defining the Discipline

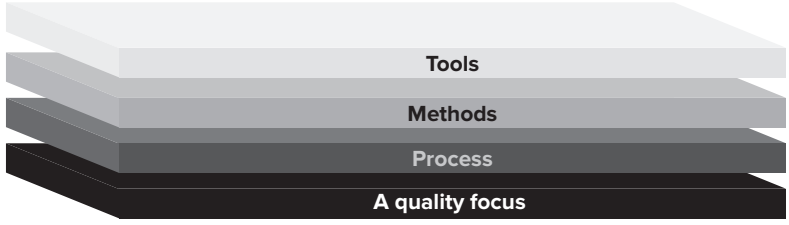
The IEEE [IEE17] has developed the following definition for software engineering:

> Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

6 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.

FIGURE 1.3

Software engineering layers



And yet, a "systematic, disciplined, and quantifiable" 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 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. You may have heard of total quality management (TQM) or Six Sigma, and similar philosophies[7] that foster a culture of continuous process improvement. It is this culture that ultimately leads to more effective approaches to software engineering. The bedrock that supports software engineering 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 defines a framework that must be established for effective delivery of software engineering 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 produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, 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.

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.

1.3 THE 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 application domain, size of the project, complexity of the effort, or degree of rigor with which

7 Quality management and related approaches are discussed throughout Part Three of this book.