CHAPTER

1 The Nature of Software

Key Concepts

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.

"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.

Quick Look

What is it? Computer software is the product that software professionals 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 commerce, our culture, and our everyday activities.

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

What is the work product? A computer program that runs in one or more specific environments 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 understand 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."

Quote:

"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 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 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 influential 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, office 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 expensive to build and maintain 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*.

1.1 The Nature of Software

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

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 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, displaying, 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 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 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.

Quote:

"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

The role of computer software has undergone significant change over the last half-century. 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:[1]

- 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?

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 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:

? How should we define *software*?

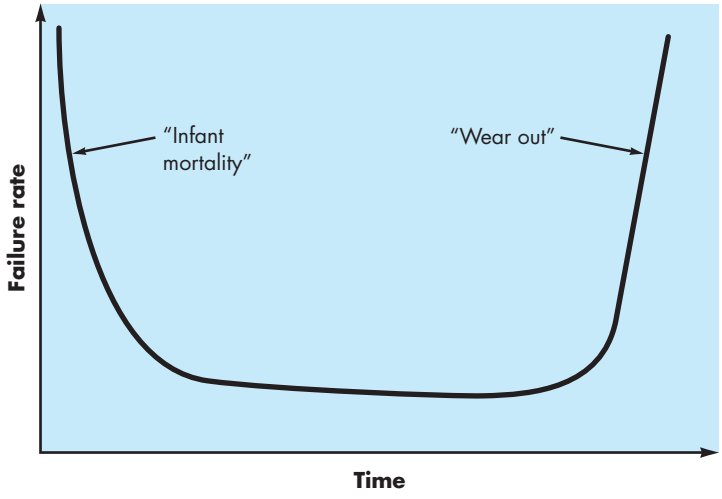
> Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation 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.

---

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 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."

FIGURE 1.1
Failure curve for hardware



ADVICE

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

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."*

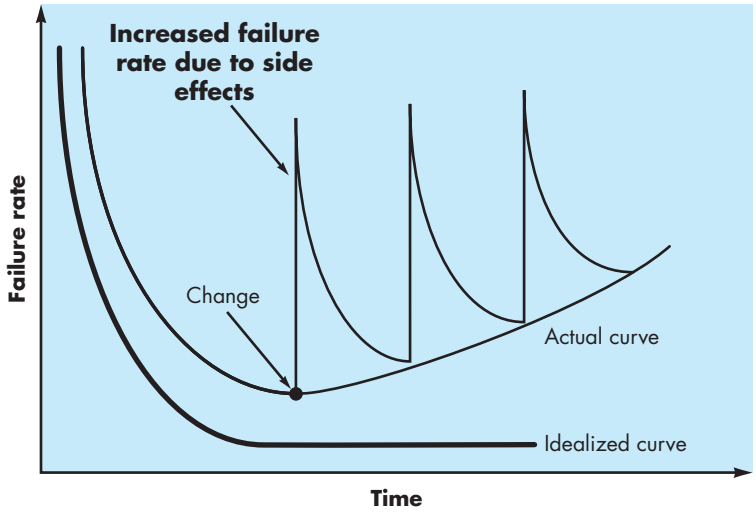
Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and 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, 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,[2] software will undergo change. As changes are

2 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.

FIGURE 1.2
Failure curves for software



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 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.

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,[3] information structures. 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 predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

WebRef

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

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 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 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—designed to provide a specific 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.

Quote:

"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

Artificial intelligence software—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, 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.

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 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.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' 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 present in legacy software—*poor quality*.[4] Legacy systems sometimes have inextensible 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 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:

? What types of changes are made to legacy systems?

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

Every software engineer must recognize that change is natural. Don't try to fight it.

When these modes of evolution occur, a legacy system must be reengineered (Chapter 36) so that it remains viable into 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 continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other." [Day99]

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 files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability 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.

Quote:

"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 internal communications and external relations, and between art and technology." [Pow98] But today, they provide full computing potential in many of the application categories noted in Section 1.1.2.

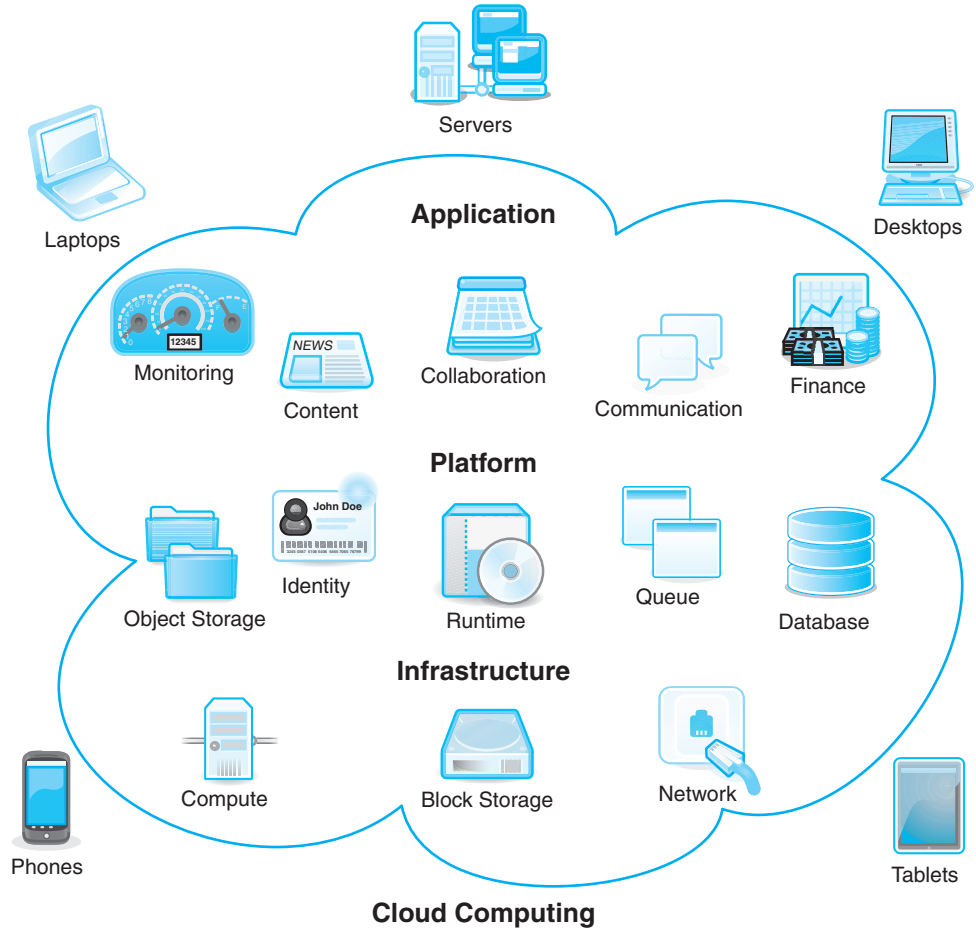
Over the past decade, Semantic Web technologies (often referred to as Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass "semantic databases [that] provide new functionality that requires Web linking, flexible [data] representation, and external access APIs." [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 specifically designed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile). In most instances, mobile applications encompass a user interface that takes advantage 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 capabilities 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 comprehensive 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.

FIGURE 1.3 Cloud computing logical architecture [Wik13]



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

It is important to recognize that there is a subtle distinction between mobile web applications and mobile apps. A *mobile web application* (WebApp) allows a mobile device to gain access to web-based content via a browser that has been specifically designed to accommodate the strengths and weaknesses of the mobile platform. A *mobile app* can gain direct access to the hardware characteristics 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 figure, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure. In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data structures. 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 architecture that encompasses front-end and back-end services. The *front-end* includes 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 applications, and administrative servers that use middleware to coordinate and monitor traffic 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 defines a *software product line* as "a set of software-intensive systems that share a common, managed set of features satisfying the specific 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 provides significant 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 commonality 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 developing 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 nonetheless. Mobile applications present new challenges as apps migrate to a wide array of platforms. Cloud computing will transform the way in which software is delivered and the environment in which it exists. Product line software offers potential efficiencies in the manner in which software is built.

Problems and Points to Ponder

1.1. Provide at least five 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 five 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 first version has been put into use. Suggest a few ways to build software 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 discuss 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-first 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.