

INTRODUCTION TO SOFTWARE ENGINEERING

1

KEY CONCEPTS

application categories
 challenges
 deterioration
 evolution
 failure curves
 history
 legacy software
 myths
 software characteristics
 software definition

Have you ever noticed how the invention of one technology can have profound and unexpected effects on other seemingly unrelated technologies, on commercial enterprises, on people, and even on culture as a whole? This phenomenon is often called “the law of unintended consequences.”

Today, computer software is the single most important technology on the world stage. And it is also a prime example of the law of unintended consequences. No one in the 1950s 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), the extension of existing technologies (e.g., telecommunications), and the demise of older technologies (e.g., the printing industry); that software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that a software company would become larger and more influential than the vast majority of industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to the dating habits of young (and not-so-young) adults.

No one could have foreseen 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 we are to believe the law of unintended consequences, there are many effects that we cannot yet predict.

QUICK LOOK

What is it? Software engineering is the application of engineering principles to the development of computer software. It involves the systematic and disciplined application of engineering principles to the development of computer software. It is a process that leads to the creation of software products that meet the needs of users and are reliable, efficient, and easy to use. You apply engineering principles to the development of software products. From the point of view of the user, the work product is a software product that meets the user's view of the problem and is easy to use. Informalized world view: It often involves the use of formalized methods and tools to create a better world.

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

Why is it important? Because it affects every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

And, finally, no one could have foreseen that millions of computer programs would have to be corrected, adapted, and enhanced as time passed and that the burden of performing these “maintenance” activities would absorb more people and more resources than all work applied to the creation of new software.

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

The Wall Street Journal

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., Web-site 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 security, and their very lives on computer software. It better be right.

This book presents a framework for 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*.

“The primary activity of engineering is to provide systems and products that enhance the material aspects of human life, thus making life easier, safer, more secure, and more enjoyable.”

Richard Fairley and Mary Williams

1.1 THE EVOLVING ROLE OF SOFTWARE

KEY POINT

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

Today, software takes on a dual role. It is both a product and a 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 software resides within a cellular phone or operates inside a mainframe computer, it 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. As the vehicle for delivering 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.

The role of computer software has undergone significant change over a span of little more than 50 years. Dramatic improvements in hardware performance, pro-

WebRef

Take a look back at the software industry of www.softwarehistory.org.

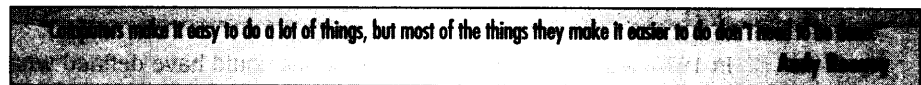
found changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

Popular books published during the 1970s and 1980s provide useful historical insight into the changing perception of computers and software and their impact on our culture. Osborne [OSB79] characterized a “new industrial revolution.” Toffler [TOF80] called the advent of microelectronics part of “the third wave of change” in human history, and Naisbitt [NAI82] predicted the transformation from an industrial society to an “information society.” Feigenbaum and McCorduck [FEI83] suggested that information and knowledge (controlled by computers) would be the focal point for power in the twenty-first century, and Stoll [STO89] argued that the “electronic community” created by networks and software was the key to knowledge interchange throughout the world. All of these writers were correct.

As the 1990s began, Toffler [TOF90] described a “power shift” in which old power structures (governmental, educational, industrial, economic, and military) disintegrate as computers and software lead to a “democratization of knowledge.” Yourdon [YOU92] worried that U.S. companies might lose their competitive edge in software-related businesses and predicted “the decline and fall of the American programmer.” Hammer and Champy [HAM93] argued that information technologies were to play a pivotal role in the “reengineering of the corporation.” During the mid-1990s, the pervasiveness of computers and software spawned a rash of books by “neo-Luddites” (e.g., *Resisting the Virtual Life*, edited by James Brook and Iain Boal, and *The Future Does Not Compute* by Stephen Talbot). These authors demonized the computer, emphasizing legitimate concerns but ignoring the profound benefits that have already been realized [LEV95].



If you have some time, take a look at one or more of these classic books. Pay attention to what these experts got wrong as they predicted future events and technologies. Stay humble: none of us can really know the future of the systems we build.



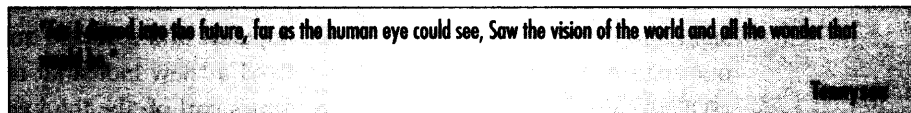
During the later 1990s, Yourdon [YOU96] reevaluated the prospects of the software professional and suggested the “the rise and resurrection” of the American programmer. As the Internet grew in importance, Yourdon’s change of heart proved to be correct. As the twentieth century closed, the focus shifted once more, this time to the impact of the Y2K “time bomb.” (e.g., [YOU98a], [KAR99]). Although the dire predictions of the Y2K doomsayers were overreactions, their popular writings drove home the pervasiveness of software in our lives.

As the 2000s progressed, Johnson [JOH01] discussed the power of “emergence”—a phenomenon that explains what happens when interconnections among relatively simple entities result in a system that “self-organizes to form more intelligent, more adaptive behavior.” Yourdon [YOU02] revisited the tragic events of 9/11 to discuss

WebRef

For commentary on a wide array of software-related topics, visit www.yourdon.com.

the continuing impact of global terrorism on the IT community. Wolfram [WOL02] presented a treatise on “a new kind of science” that posits a unifying theory based primarily on sophisticated software simulations. Daconta and his colleagues [DAC03] discussed the evolution of “the semantic Web” and ways in which it will change the way people interact across global networks.



Today, a huge software industry has become a dominant factor in the economies of the industrialized world. The lone programmer of an earlier era has been replaced by teams of software specialists, each focusing on one part of the technology required to deliver a complex application. 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 questions and many others demonstrate the industry's concern about software and the manner in which it is developed—a concern that has led to the adoption of software engineering practice.



In 1970, less than 1 percent of the public could have defined what “computer software” meant. Today, most professionals and many members of the public at large feel that they understand software. But do they?

How should we define software?

A textbook definition 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) documents that describe the operation and use of the programs.*

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.”

There is no question that more complete definitions could be offered. But we need more than a formal definition.

To gain an understanding of software (and ultimately an understanding of software engineering), it is 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 characteristics that are considerably different than those of hardware:

1. *Software is developed or engineered; it is not manufactured in the classical sense.*

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different (see Chapter 24). Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. *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 then corrected, and failure rate

KEY POINT

Software is engineered, not manufactured.

KEY POINT

Software doesn't wear out, but it does deteriorate.

FIGURE 1.1

Failure curve for hardware

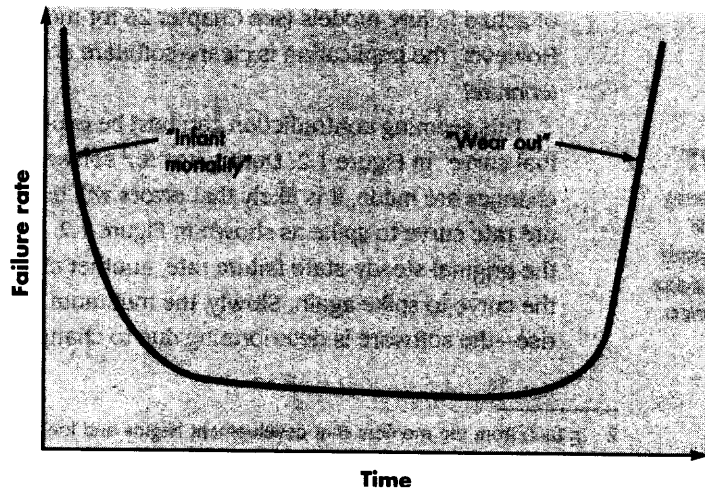
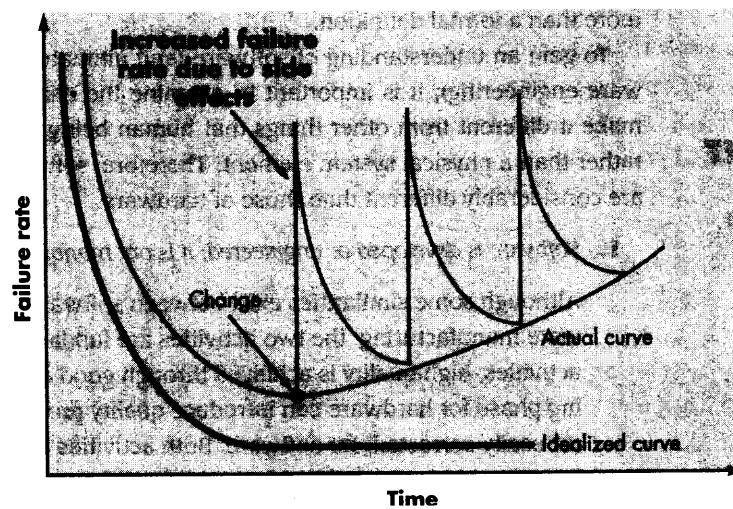


FIGURE 1.2

Failure curves for software



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 affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out*.



ADVICE
If you want to reduce software deterioration, you'll have to do better software design (Chapters 9–12).



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

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 (hopefully, without introducing other errors), and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models (see Chapter 26 for more information) 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 made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in 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.

² In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by the customer.

**KEY
POINT**

Most software continues to be custom built.

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, software maintenance involves considerably more complexity than hardware maintenance.

3. *Although the industry is moving toward component-based construction, most software continues to be custom built.*

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to ensure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, i.e., the parts that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it has only begun to be achieved on a broad scale.

"Screws are the building blocks of ideas."

Jason Zeberry

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.³ For example, today's user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

³ Component-based software engineering is presented in Chapter 30.

1.3 THE CHANGING NATURE OF SOFTWARE

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

System software. System software is 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. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

WebRef

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

Application software. Application software consists of standalone 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. In addition to conventional data processing applications, application software is used to control business functions in real-time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software. Formerly characterized by “number crunching” algorithms, engineering and scientific software applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software. 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., keypad 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, braking systems, etc.).

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 mar-

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.

ketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications).

Web-applications. “WebApps,” span a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as e-commerce and B2B applications grow in importance, WebApps are evolving into sophisticated computing environments that not only provide standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software. AI 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.

“There is no computer that has common sense.”

Marvin Minsky

Millions of software engineers worldwide are hard at work on projects in one or more of these categories. In some cases, new systems are being built, but in others, existing applications are being corrected, adapted, and enhanced. It is common 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 left behind by this generation will ease the burden of future software engineers. And yet, new challenges have appeared on the horizon:

Ubiquitous computing. The rapid growth of wireless networking may soon lead to true distributed computing. The challenge for software engineers will be to develop systems and application software that will allow small devices, personal computers, and enterprise system to communicate across vast networks.

Netsourcing. The World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide benefit to targeted end-user markets worldwide.

“You can’t always predict, but you can always prepare.”

Anonymous

Open source. A growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that customers can make local modifications. The challenge for software

engineers is to build source code that is self-descriptive, but, more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

The “new economy.” The dot-com insanity that gripped financial markets during the late 1990s and the bust that followed in the early 2000s have lead many business people to believe that the new economy is dead. The new economy is alive and well, but it will evolve slowly. It will be characterized by mass communication and distribution. Andy Lippman [LIP02] notes this when he writes:

We are entering an era characterized by communications among distributed machines and dispersed people, rather than being mostly about a connection between two individuals or between an individual and a machine. The old approach to telephony was about “connections to”; the next wave is about “connections among.” Napster, instant messaging, short message systems, and BlackBerries are examples.

The challenge for software engineers is to build applications that will facilitate mass communication and mass product distribution using concepts that are only now forming.

Each of these “new challenges” will undoubtedly obey the law of unintended consequences and have effects (for business people, software engineers, and end-users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and business rules that are sure to come in the next decade.

“The computer itself will make a historic transition from something that is used for analytic tasks . . . to something that is used for creative tasks.”
David Yachnick

1.4 LEGACY SOFTWARE


Hundreds of thousands of computer programs fall into one of the seven broad application domains—system software, application software, engineering/scientific software, embedded software, product software, WebApps, and AI applications—discussed in Section 1.3. 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.

1.4.1 The Quality of Legacy Software

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

Unfortunately, there is one additional characteristic that can be present in legacy software—*poor quality*.⁵ 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” [LIU98]. What can one do?

The only reasonable answer may be to 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 more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.

 **ADVICE**

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 31) 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].

1.4.2 Software Evolution

Regardless of its application domain, size, or complexity, computer software will evolve over time. Change (often referred to as *software maintenance*) drives this process and occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when

⁵ 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.

the application is reengineered to provide benefit in a modern context. Sam Williams [WIL02] describes this when he writes:

As large-scale programs such as Windows and Solaris expand well into the range of 30 to 50 million lines of code, successful project managers have learned to devote as much time to combing the tangles out of legacy code as to adding new code. Simply put, in a decade that saw the average PC microchip performance increase a hundredfold, software's inability to scale at even linear rates has gone from dirty little secret to an industry-wide embarrassment.

Over the past 30 years, Manny Lehman [e.g., LEH97a] and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution*. The details of this work are beyond the scope of this book,⁶ but the underlying laws that have been derived are worthy of note [LEH97b]:

The Law of Continuing Change (1974). E-type systems⁷ must be continually adapted, or else they become progressively less satisfactory.

The Law of Increasing Complexity (1974). As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

The Law of Self-Regulation (1974). The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.


The Law of Conservation of Organizational Stability (1980). The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

The Law of Conservation of Familiarity (1980). As an E-type system evolves all associated with it, developers, sales personnel, and users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The Law of Continuing Growth (1980). The functional content of E-type systems must be continually increased to maintain user satisfaction over the system's lifetime.

The Law of Declining Quality (1996). The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The Feedback System Law (1996). E-type evolution processes constitute multilevel, multiloop, multiagent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

 Why do legacy systems evolve as time passes?

⁶ The interested reader should see [LEH97a] for a comprehensive discussion of software evolution.

⁷ *E-types systems* are software that has been implemented in a real-world computing context and will therefore evolve over time.

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality. For the remainder of this book, we discuss software process models, software engineering methods, and management techniques that strive to maintain quality as software evolves.

1.5 SOFTWARE MYTHS

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

“In the absence of meaningful standards, a new industry like software comes to depend instead on folklore.”

Tom DeMarco

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules 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).

WebRef

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

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 software practitioners aware of its existence? Does it reflect 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 questions 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 manufacturing. In the words of Brooks [BRO75]: “Adding people to a late software project makes it later.” At first, 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 development

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 firm build it.*

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

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, 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 sufficient to begin writing programs—we can fill 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: *Project requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early (before design or code has been started), cost impact is relatively small.⁸ However, as time passes, 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 modification.



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

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days of software, 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

⁸ Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 4.

60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first 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 *formal technical review*. Software reviews (described in Chapter 26) are a “quality filter” that have been found to be more effective than testing for finding certain classes of software errors.

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 configuration that includes many elements. Documentation provides a foundation for successful engineering and, more importantly, guidance for software support.

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

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

Many software professionals recognize the fallacy of software myths. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.0 HOW IT ALL STARTS

Every software project is precipitated by some business need—the need to correct 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 engineering project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar (next page) is typical.

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* product line. The engineering effort will succeed only if *SafeHome* software succeeds. The market will accept the product only if the software embedded within it properly meets the customer’s (as yet unstated) needs. We’ll follow the progression of *SafeHome* software engineering in subsequent chapters.

SAFEHOME⁹**How a Project Starts**

The scene: Meeting room at CPI Corporation, a (fictional) 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, executive VP, business development.

The conversation:

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

Lee: It's pretty cool, about the size of a small matchbook. We can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11b 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 flat 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 homeowners or small business people with a system that's controlled 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 manufacturing 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 have penetrated 60 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 two-year jump on the competition. Revenue? Maybe as much as \$30–40 million in the second year.

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

1.7 SUMMARY

Software has become 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 documents—addresses a wide array of technology and application areas, yet all software evolves according to a set of laws that have remained the same for over 30 years. The intent of software engineering is to provide a framework for building higher quality software.

⁹ The *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 purely fictitious, but the situations and problems are real.

REFERENCES

- [BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.
- [DAC03] Daconta, M., L. Obrst, and K. Smith, *The Semantic Web*, Wiley, 2003.
- [DAY99] Dayani-Fard, H., et al., "Legacy Software Systems: Issues, Progress, and Challenges," IBM Technical Report: TR-74.165-k, April 1999, available at <http://www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html>.
- [DEM95] DeMarco, T., *Why Does Software Cost So Much?*, Dorset House, 1995.
- [FEI83] Feigenbaum, E. A., and P. McCorduck, *The Fifth Generation*, Addison-Wesley, 1983.
- [HAM93] Hammer, M., and J. Champy, *Reengineering the Corporation*, HarperCollins Publishers, 1993.
- [JOH01] Johnson, S., *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*, Scribner, 2001.
- [KAR99] Karlson, E., and J. Kolber, *A Basic Introduction to Y2K: How the Year 2000 Computer Crisis Affects YOU*, Next Era Publications, Inc., 1999.
- [LEH97a] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.
- [LEH97b] Lehman, M., et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, can be downloaded from <http://www.ece.utexas.edu/~perry/work/papers/feast1.pdf>.
- [LEV95] Levy, S., "The Luddites Are Back," *Newsweek*, July 12, 1995, p. 55.
- [LIP02] Lippman, A., "Round 2.0," *Context Magazine*, August 2002, <http://www.contextmag.com/>.
- [LIU98] Liu, K., et al., "Report on the First SEBPC Workshop on Legacy Systems," Durham University, February, 1998, available at <http://www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html>.
- [OSB79] Osborne, A., *Running Wild—The Next Industrial Revolution*, Osborne/McGraw-Hill, 1979.
- [NAI82] Naisbitt, J., *Megatrends*, Warner Books, 1982.
- [STO89] Stoll, C., *The Cuckoo's Egg*, Doubleday, 1989.
- [TOF80] Toffler, A., *The Third Wave*, Morrow Publishers, 1980.
- [TOF90] Toffler, A., *Powershift*, Bantam Publishers, 1990.
- [WIL02] Williams, S., "A Unified Theory of Software Evolution," *salon.com*, 2002, <http://www.salon.com/tech/feature/2002/04/08/lehman/index.html>.
- [WOL02] Wolfram, S., *A New Kind of Science*, Wolfram Media, Inc, 2002.
- [YOU92] Yourdon, E., *The Decline and Fall of the American Programmer*, Yourdon Press, 1992.
- [YOU96] Yourdon, E., *The Rise and Resurrection of the American Programmer*, Yourdon Press, 1996.
- [YOU98a] Yourdon, E., and J. Yourdon, *Time Bomb 2000*, Prentice-Hall, 1998.
- [YOU98b] Yourdon, E., *Death March Projects*, Prentice-Hall, 1999.
- [YOU02] Yourdon, E., *Byte Wars*, Prentice-Hall, 2002.

PROBLEMS AND POINTS TO PONDER

- 1.1. Does the definition for software presented in Section 1.2 apply to Web sites? If you answered yes, indicate the subtle difference between a Web site and conventional software, if any.
- 1.2. Develop your own answers to the questions asked in Section 1.1. Discuss them with your fellow students.
- 1.3. Provide a number of examples (both positive and negative) that indicate the impact of software on our society. Review one of the pre-1990 references in Section 1.1 and indicate where the author's predictions were right and where they were wrong.
- 1.4. Provide at least five additional examples of how the law of unintended consequences applied to computer software.

- 1.5.** Select one of the new challenges noted in Section 1.3 (or an even newer challenge that has arisen since this book was printed) and write a one-page paper that describes the technology and the challenges it poses for software engineers.
- 1.6.** Describe *The Law of Conservation of Familiarity* (Section 1.4.2) in your own words.
- 1.7.** 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.
- 1.8.** Peruse the Internet newsgroup comp.risks and prepare a summary of risks to the public that have recently been discussed. Alternate source: *Software Engineering Notes* published by the ACM.
- 1.9.** Consider the seven software categories presented in Section 1.3. Can the same approach to software engineering be applied for each? Explain your answer.
- 1.10.** As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a realistic doomsday scenario where the failure of a computer program could do great harm (either economic or human).
- 1.11.** Describe *The Law of Declining Quality* (Section 1.4.2) in your own words.
- 1.12.** Describe *The Law of Conservation of Organizational Stability* (Section 1.4.2) in your own words.

FURTHER READINGS AND INFORMATION SOURCES¹⁰

There are literally thousands of books 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) present an early discussion (directed at the layman) of software and the way professionals build it. Negroponte's best-selling book (*Being Digital*, Alfred A. Knopf, Inc., 1995) provides a view of computing and its overall impact in the twenty-first century. DeMarco [DEM95] has produced a collection of amusing and insightful essays on software and the process through which it is developed. Books by Norman (*The Invisible Computer*, MIT Press, 1998) and Bergman (*Information Appliances and Beyond*, Academic Press/Morgan Kaufmann, 2000) suggest that the widespread impact of the PC will decline as information appliances and pervasive computing connect everyone in the industrialized world and almost every "appliance" that they own to a new Internet infrastructure.

Minasi (*The Software Conspiracy: 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. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) argues that the "divide" between those who have access to information resources (e.g., the Web) and those who do not is narrowing as we move into the first decade of this century.

A wide variety of information sources on software related topics and management are available on the Internet. An up-to-date list of World Wide Web resources that are relevant to software can be found at our Web site:

<http://www.mhhe.com/pressman>.

¹⁰ The *Further Readings 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 Web site to support *Software Engineering: A Practitioner's Approach* at <http://www.mhhe.com/pressman>. Among the many topics addressed within the Web site 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.