Chapter 1

Software engineering

Introduction

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 is right.

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:

· 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?

Defining 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.

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, 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 SteadyState 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.

FIGURE 1.1
Failure curve for hardware

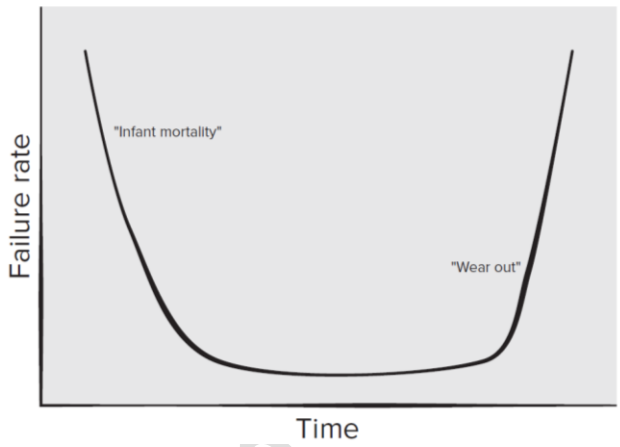
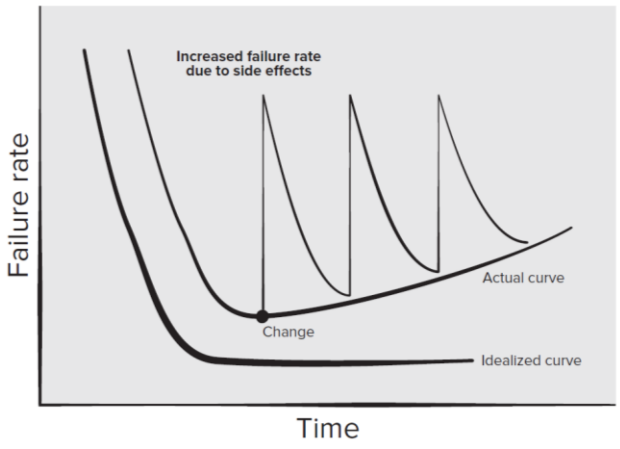


FIGURE 1.2
Failure curves for software



Software Application Domains

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

1. 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, information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

2. 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.

3. 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.

4. 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).

5. 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.

6. 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.

7. Artificial intelligence software. Makes use of heuristics 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.

Software Characteristics

1. Software is developed or engineered.
2. Most of software is custom build rather than assemble from existing component.
3. Computer program and associated documentation.
4. Easy to modified.
5. Easy to reproduce.
6. Software product may be developed for a particular customer or for the general market.

Programmer & Software Engineer

Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operate correctly.

A software system consists of:

-separate programs

-configuration files to setup programs

-system documentation to describe the structure of the system.

-User documentation to explain how to use the system .

- Web sites to down load recent product information.

The characteristic of software engineer

- Good programmer and fluent in one or more programming language .
- Well versed data structure and approaches .
- Familiar with several designs' approaches .
- Be able to translate vague (not clear) requirements and desires into precise specification .
- Be able to converse with the user of the system in terms of application not in "computer ."
- Able to a build a model. The model is used to answer questions about the system behavior and its performance .
- Communication skills and interpersonal skills.

The Attributes of Good Software

As well as the service which they provide software products have a number of other associated attributes which reflect the quality of that software.

These attributes are not directly concerned with what the software dose, rather they reflect its behavior which it is executing and the structure and organization of the source program and associated documentation. Examples of these attributes (some time called non-functional attributes) are the software's response time to use query and the understandability of the program code. The specific set of attributes which you might expect from a software system obviously depends on its application. Therefore a banking system must be secure, on interactive game must be responsive, a telephone switching system must be reliable, etc. these can be generated in the following attributes:

1- Maintainability: software should be written in such a way that it may evolve to meet the changing needs of customer. This is critical attribute because software change is an inevitable

2- Dependability: software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.

3- efficiency: software should not make wasteful use of system resources, such as memory and processor cycles. Therefore efficiency includes responsiveness, processing time, memory utilization etc…

4.Usability: software must be usable, without under effort by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

The Goals of Software Engineering

•Efficiency—The software is produced in the expected time and within the limits of the available resources. The software that is produced runs within the time expected for various computations to be completed.

• Reliability—The software performs as expected. In multiuser systems, the system performs its functions even with other loads on the system.

• Usability—The software can be used properly. This generally refers to the ease of use of the user interface but also concerns the applicability of the software to both the computer's operating system and the application environment.

• Modifiability—The software can be easily changed if the requirements of the system change.

• Portability—The software system can be ported to other computers or systems without major rewriting of the software.

• Testability—The software can be easily tested. This generally means that the software is written in a modular manner.

• Reusability—Some or all of the software can be used again in other projects. This means that the software is modular, that each individual software module has a well-defined interface, and that each individual module has a clearly defined outcome from its execution.

• Maintainability—The software can be easily understood and changed over time if problems occur. This term is often used to describe the lifetime of long-lived systems such as the air traffic control system that must operate for decades.

• Correctness—The program produces the correct output.

Legacy Software

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 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. 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."

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.

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 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.

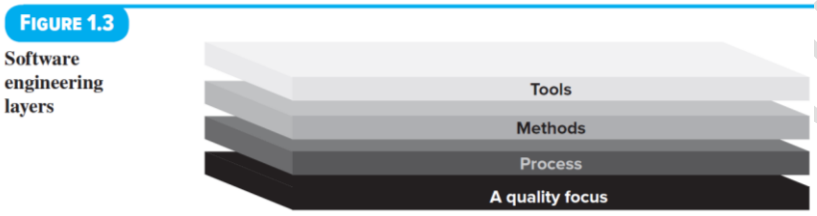


FIGURE 1.3 Software engineering layers

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 software engineering is to be applied. An action (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural model). A task focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

The Process Framework

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of

umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

1. Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders). The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

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

3. Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

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

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

These five generic framework activities can be used during the development of small, simple programs; the creation of Web applications; and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

Umbrella Activities

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

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

2.Risk management. Assesses risks that may affect the outcome of the project or the quality of the product.

3. Software quality assurance. Defines and conducts the activities required to ensure software quality.

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

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

6. Software configuration management. Manages the effects of change throughout the software process.

7. Reusability management. Defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

8. Work product preparation and production. Encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Process Adaptation

Previously in this section, we noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process

adopted for one project might be significantly different than a process adopted for another project. Among the differences are:

· Overall flow of activities, actions, and tasks and the interdependencies among them

· Degree to which actions and tasks are defined within each framework activity

· Degree to which work products are identified and required

· Manner in which quality assurance activities are applied

· Manner in which project tracking and control activities are applied

· Overall degree of detail and rigor with which the process is described

· Degree to which the customer and other stakeholders are involved with the project

· Level of autonomy given to the software team

· Degree to which team organization and roles are prescribed