CHAPTER

1 THE PRODUCT

KEY CONCEPTS

The warnings began more than a decade before the event, but no one paid much attention. With less than two years to the deadline, the media picked up the story. Then government officials voiced their concern, business and industry leaders committed vast sums of money, and finally, dire warnings of pending catastrophe penetrated the public's consciousness. Software, in the guise of the now-infamous Y2K bug, would fail and, as a result, stop the world as we then knew it.

As we watched and wondered during the waning months of 1999, I couldn't help thinking of an unintentionally prophetic paragraph contained on the first page of the fourth edition of this book. It stated:

> Computer software has become a driving force. It is the engine that drives business decision making. It serves as the basis for modern scientific investigation and engineering problem solving. It is a key factor that differentiates modern products and services. It is embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial processes, entertainment, office products, . . . the list is almost endless. Software is virtually inescapable in a modern world. And as we move into the twenty-first century, it will become the driver for new advances in everything from elementary education to genetic engineering.

QUICK LOOK

What is it? Computer software is the product that software engineers design and build. It encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text but also includes representations of pictorial, video, and audio information.

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

Why is it 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? You build computer software like you build any successful product, by applying a process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

What is the work product? From the point of view of a software engineer, the work product is the programs, documents, and data that are computer software. But from the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

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

In the five years since the fourth edition of this book was written, the role of software as the "driving force" has become even more obvious. A software-driven Internet has spawned its own $500 billion economy. In the euphoria created by the promise of a new economic paradigm, Wall Street investors gave tiny "dot-com" companies billion dollar valuations before these start-ups produced a dollar in sales. New software-driven industries have arisen and old ones that have not adapted to the new driving force are now threatened with extinction. The United States government has litigated against the software's industry's largest company, just as it did in earlier eras when it moved to stop monopolistic practices in the oil and steel industries.

Quote:
"Ideas and technological discoveries are the driving engines of economic growth."
The Wall Street Journal

Software's impact on our society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build 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); 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 comfort, 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 technology encompasses a process, a set of methods, and an array of tools that we call *software engineering*.

1.1 THE EVOLVING ROLE OF SOFTWARE

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside 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. 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).

KEY POINT
Software is both a product and a vehicle for delivering a product.

Software delivers the most important product of our time—information. Software 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., Internet) and provides the means for acquiring information in all of its forms.

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

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.

Quote:
"For I dipped into the future, far as the human eye could see, Saw the vision of the world, and all the wonder that would be."
Tennyson

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 a 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.

Quote:
"Computers make it easy to do a lot of things, but most of the things that they make it easier to do don't need to be done."
Andy Rooney

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 loose 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]

During the later 1990s, Yourdon [YOU96] re-evaluated the prospects for the software professional and suggested the "the rise and resurrection" of the American programmer. As the Internet grew in importance, his 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., [YOU98b], [DEJ98], [KAR99]). Although the predictions of the Y2K doomsayers were incorrect, their popular writings drove home the pervasiveness of software in our lives. Today, "ubiquitous computing" [NOR98] has spawned a generation of information appliances that have broadband connectivity to the Web to provide "a blanket of connectedness over our homes, offices and motorways" [LEV99]. Software's role continues to expand.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application. And yet, the same questions asked of the lone programmer are being 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 the errors before we give the software to customers?
- Why do we continue to have difficulty in measuring progress as software is being developed?

These, and many other questions,[1] are a manifestation of the concern about software and the manner in which it is developed—a concern that has lead to the adoption of software engineering practice.

1.2 SOFTWARE

In 1970, less than 1 percent of the public could have intelligently described 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 description of software might take the following form: *Software is (1) instructions (computer programs) that when executed provide desired 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.* There is no question that other, more complete definitions could be offered. But we need more than a formal definition.

1.2.1 Software Characteristics

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. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and breadboarded prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

KEY POINT Software is engineered, not manufactured.

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

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high qual-

1 In an excellent book of essays on the software business, Tom DeMarco [DEM95] argues the counterpoint. He states: "Instead of asking 'why does software cost 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."

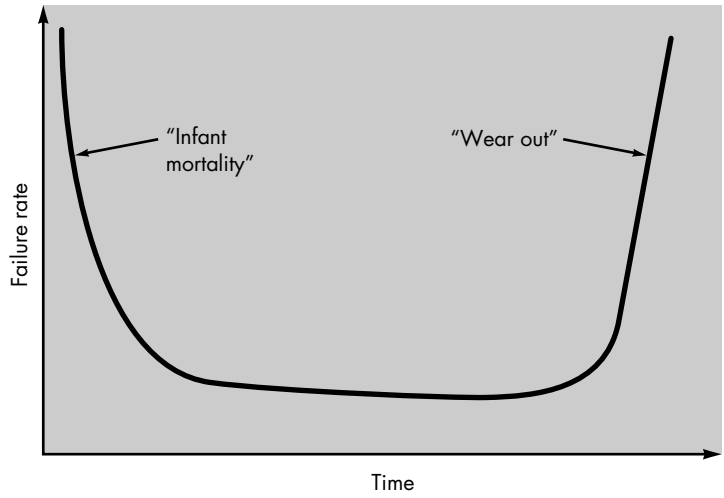


FIGURE 1.1
Failure curve for hardware

ity 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 7). 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.

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

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 corrected and the failure rate drops to a steady-state level (ideally, 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.

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 (ideally, without introducing other errors) and the curve flattens as shown.The idealized curve is a gross oversimplification of actual failure models (see Chapter 8 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" shown in Figure 1.2. During its life, software will undergo change (maintenance). As

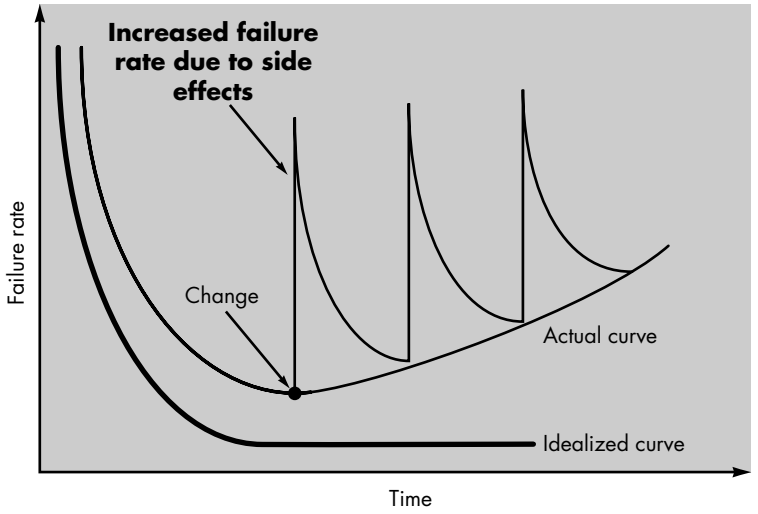


FIGURE 1.2 Idealized and actual failure curves for software

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

changes are made, it is likely that some new defects 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.

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.

KEY POINT
Most software continues to be custom built.

3. Although the industry is moving toward component-based assembly, 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 assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an *IC* or a *chip*) 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, that is, the

parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

XRef

Software reuse is discussed in Chapter 13. Component-based software engineering is presented in Chapter 27.

1.2.2 Software Applications

Software may be applied in any situation for which a prespecified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

Information determinacy refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system, on the other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate.

It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

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) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications

processors) process largely indeterminate data. In either case, the system 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.

Real-time software. Software that monitors/analyzes/controls real-world events as they occur is called *real time.* Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

Business software. Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point-of-sale transaction processing).

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

Engineering and scientific software. Engineering and scientific software have been characterized by "number crunching" algorithms. 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. Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very 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, and braking systems).

Personal computer software. The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software. The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g.,

hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

Artificial intelligence software. Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

1.3 SOFTWARE: A CRISIS ON THE HORIZON?

Quote:

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents."

Nathaniel Borenstein

Many industry observers (including this author) have characterized the problems associated with software development as a "crisis." More than a few books (e.g., [GLA97], [FLO97], [YOU98a]) have recounted the impact of some of the more spectacular software failures that have occurred over the past decade. Yet, the great successes achieved by the software industry have led many to question whether the term *software crisis* is still appropriate. Robert Glass, the author of a number of books on software failures, is representative of those who have had a change of heart. He states [GLA98]: "I look at my failure stories and see exception reporting, spectacular failures in the midst of many successes, a cup that is [now] nearly full."

It is true that software people succeed more often than they fail. It also true that the software crisis predicted 30 years ago never seemed to materialize. What we really have may be something rather different.

The word *crisis* is defined in *Webster's Dictionary* as "a turning point in the course of anything; decisive or crucial time, stage or event." Yet, in terms of overall software quality and the speed with which computer-based systems and products are developed, there has been no "turning point," no "decisive time," only slow, evolutionary change, punctuated by explosive technological changes in disciplines associated with software.

The word *crisis* has another definition: "the turning point in the course of a disease, when it becomes clear whether the patient will live or die." This definition may give us a clue about the real nature of the problems that have plagued software development.

What we really have might be better characterized as a chronic affliction.[2] The word *affliction* is defined as "anything causing pain or distress." But the definition of the adjective *chronic* is the key to our argument: "lasting a long time or recurring often; continuing indefinitely." It is far more accurate to describe the problems we have endured in the software business as a chronic affliction than a crisis.

Regardless of what we call it, the set of problems that are encountered in the development of computer software is not limited to software that "doesn't function

2 This terminology was suggested by Professor Daniel Tiechrow of the University of Michigan in a talk presented in Geneva, Switzerland, April 1989.

properly." Rather, the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

We live with this affliction to this day—in fact, the industry prospers in spite of it. And yet, things would be much better if we could find and broadly apply a cure.

1.4 SOFTWARE MYTHS

Quote:

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

Tom DeMarco

Many causes of a software affliction can be traced to a mythology that arose during the early history of software development. Unlike ancient myths that often provide human lessons well worth heeding, software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact (sometimes containing elements of truth), they had an intuitive feel, and they were often promulgated by experienced practitioners who "knew the score."

Today, most knowledgeable 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).

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 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: My people have state-of-the-art software development tools, after all, we buy them the newest computers.
Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

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.

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

Myth: If I decide to outsource[3] 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: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough 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. Figure 1.3 illustrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

The management and control of change is considered in detail in Chapter 9.

3 The term "outsourcing" refers to the widespread practice of contracting software development work to a third party—usually a consulting firm that specializes in building custom software for its clients.

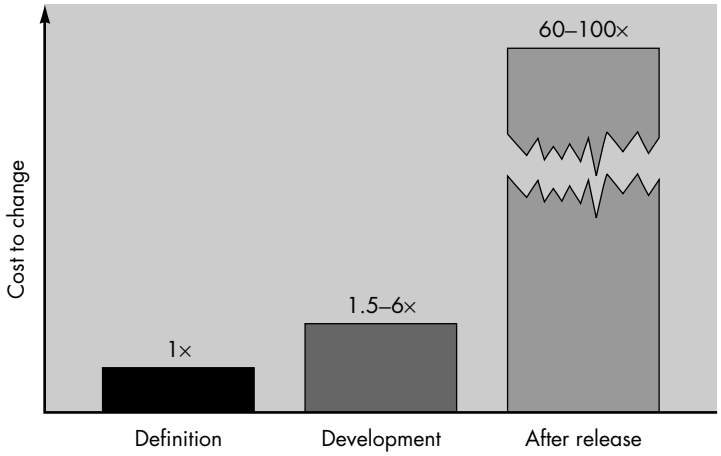


FIGURE 1.3
The impact of change

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by 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 ([LIE80], [JON91], [PUT97]) indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

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

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 8) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.
Reality: A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, 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 the myths just described. 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.

1.5 SUMMARY

Software has become the key element in the evolution of computer-based systems and products. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. But early "programming" culture and history have created a set of problems that persist today. Software has become the limiting factor in the continuing evolution of computer-based systems. Software is composed of programs, data, and documents. Each of these items comprises a configuration that is created as part of the software engineering process. The intent of software engineering is to provide a framework for building software with higher quality.

REFERENCES

[BRO75] Brooks, F., *The Mythical Man-Month,* Addison-Wesley, 1975.

[DEJ98] De Jager, P. et al., *Countdown Y2K: Business Survival Planning for the Year 2000,* Wiley, 1998.

[DEM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995, p. 9.

[FEI83] Feigenbaum, E.A. and P. McCorduck, *The Fifth Generation,* Addison-Wesley, 1983.

[FLO97] Flowers, S., *Software Failure, Management Failure—Amazing Stories and Cautionary Tales,* Wiley, 1997.

[GLA97] Glass, R.L., *Software Runaways,* Prentice-Hall, 1997.

[GLA98] Glass, R.L., "Is There Really a Software Crisis?" *IEEE Software,* vol. 15, no. 1, January 1998, pp. 104–105.

[HAM93] Hammer, M., and J. Champy, *Reengineering the Corporation,* HarperCollins Publishers, 1993.

[JON91] Jones, C., *Applied Software Measurement,* McGraw-Hill, 1991.

[KAR99] Karlson, E. and J. Kolber, *A Basic Introduction to Y2K: How the Year 2000 Computer Crisis Affects YOU,* Next Era Publications, 1999.

[LEV95] Levy, S., "The Luddites Are Back," *Newsweek,* July 12, 1995, p. 55.

[LEV99] Levy, S., "The New Digital Galaxy," *Newsweek,* May 31, 1999, p. 57.

[LIE80] Lientz, B. and E. Swanson, *Software Maintenance Management,* Addison-Wesley, 1980.

[NAI82] Naisbitt, J., *Megatrends,* Warner Books, 1982.

[NOR98] Norman, D., *The Invisible Computer,* MIT Press, 1998.

[OSB79] Osborne, A., *Running Wild—The Next Industrial Revolution,* Osborne/McGraw-Hill, 1979.

[PUT97] Putnam, L. and W. Myers, *Industrial Strength Software,* IEEE Computer Society Press, 1997.

[STO89] Stoll, C., *The Cuckoo's Egg,* Doubleday, 1989.

[TOF80] Toffler, A., *The Third Wave,* Morrow, 1980.

[TOF90] Toffler, A., *Powershift,* Bantam Publishers, 1990.
[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., *Death March Projects,* Prentice-Hall, 1998.
[YOU98b] Yourdon, E. and J. Yourdon, *Time Bomb 2000,* Prentice-Hall, 1998.

PROBLEMS AND POINTS TO PONDER

1.1. Software is the differentiating characteristic in many computer-based products and systems. Provide examples of two or three products and at least one system in which software, not hardware, is the differentiating element.

1.2. In the 1950s and 1960s, computer programming was an art form learned in an apprenticelike environment. How have the early days affected software development practices today?

1.3. Many authors have discussed the impact of the "information era." 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. Choose a specific application and indicate: (a) the software application category (Section 1.2.2) into which it fits; (b) the data content associated with the application; and (c) the information determinacy of the application.

1.5. As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a realistic doomsday scenario (other than Y2K) where the failure of a computer program could do great harm (either economic or human).

1.6. Peruse the Internet newsgroup comp.risks and prepare a summary of risks to the public that have recently been discussed. An alternate source is *Software Engineering Notes* published by the ACM.

1.7. Write a paper summarizing recent advances in one of the leading edge software application areas. Potential choices include: advanced Web-based applications, virtual reality, artificial neural networks, advanced human interfaces, intelligent agents.

1.8. The "myths" noted in Section 1.4 are slowly fading as the years pass, but others are taking their place. Attempt to add one or two "new" myths to each category.

FURTHER READINGS AND INFORMATION SOURCES

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 (*Being Digital,* Alfred A. Knopf, 1995) best-selling book provides a view of computing and its overall impact in the twenty-first century. Books by Norman [NOR98] 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. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software and the process through which it is developed.

A wide variety of information sources on software-related topics and management is available on the Internet. An up-to-date list of World Wide Web references that are relevant to software can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/product.mhtml

CHAPTER

2 THE PROCESS

KEY CONCEPTS

In a fascinating book that provides an economist's view of software and software engineering, Howard Baetjer, Jr. [BAE98], comments on the software process:

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

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

QUICK LOOK

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

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

Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic.

What are the steps? At a detailed level, the process that you adopt depends on the software you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a Web site.

What is the work product? From the point of view of a software engineer, the work products are the programs, documents, and data produced as a consequence of the software engineering activities defined by the process.

How do I ensure that I've done it right? A number of software process assessment mechanisms enable organizations to determine the "maturity" of a software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.

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

More important, software engineering is performed by creative, knowledgeable people who should work within a defined and mature software process that is appropriate for the products they build and the demands of their marketplace. The intent of this chapter is to provide a survey of the current state of the software process and pointers to more detailed discussion of management and technical topics presented later in this book.

2.1 SOFTWARE ENGINEERING: A LAYERED TECHNOLOGY

Quote:

"More than a discipline or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem."

Scott Whitmire

Although hundreds of authors have developed personal definitions of *software engineering,* a definition proposed by Fritz Bauer [NAU69] at the seminal conference on the subject still serves as a basis for discussion:

> [Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of a mature process. And yet, Bauer's definition provides us with a baseline. What "sound engineering principles" can be applied to computer software development? How do we "economically" build software so that it is "reliable"? What is required to create computer programs that work "efficiently" on not one but many different "real machines"? These are the questions that continue to challenge software engineers.

? How do we define *software engineering*?

The IEEE [IEE93] has developed a more comprehensive definition when it states:

> Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

2.1.1 Process, Methods, and Tools

Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the

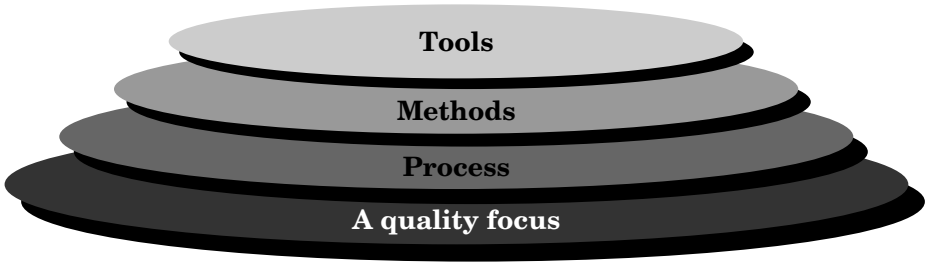


FIGURE 2.1 Software engineering layers

development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the *process* layer. 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 for a set of *key process areas* (KPAs) [PAU93] that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish 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.

KEY POINT

Software engineering encompasses a process, management and technical methods, and tools.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, 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. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

2.1.2 A Generic View of Software Engineering

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:

- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?

WebRef

Crosstalk is a journal that provides pragmatic software engineering advice and comment. On-line issues are available at **www.stsc.hill.af.mil**

- How will the entity (and the solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity.

Throughout this book, we focus on a single entity—computer software. To engineer software adequately, a software engineering process must be defined. In this section, the generic characteristics of the software process are considered. Later in this chapter, specific process models are addressed.

KEY POINT

Software is engineered by applying three distinct phases that focus on definition, development, and support.

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted previously.

The *definition phase* focuses on *what.* That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form: system or information engineering (Chapter 10), software project planning (Chapters 3, 5, 6, and 7), and requirements analysis (Chapters 11, 12, and 21).

Quote:

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

Fred Brooks

The *development phase* focuses on *how*. That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur: software design (Chapters 13–16, and 22), code generation, and software testing (Chapters 17, 18, and 23).

The *support phase* focuses on *change* associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software. Four types of change are encountered during the support phase:

Correction. Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. *Corrective maintenance* changes the software to correct defects.

Adaptation. Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was

developed is likely to change. *Adaptive maintenance* results in modification to the software to accommodate changes to its external environment.

Enhancement. As software is used, the customer/user will recognize additional functions that will provide benefit. *Perfective maintenance* extends the software beyond its original functional requirements.

Prevention. Computer software deteriorates due to change, and because of this, *preventive maintenance,* often called *software reengineering,* must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

When you use the term maintenance, *recognize that it's much more than simply fixing bugs.*

In addition to these support activities, the users of software require continuing support. In-house technical assistants, telephone-help desks, and application-specific Web sites are often implemented as part of the support phase.

Today, a growing population of legacy programs[1] is forcing many companies to pursue software reengineering strategies (Chapter 30). In a global sense, software reengineering is often considered as part of business process reengineering.

The phases and related steps described in our generic view of software engineering are complemented by a number of *umbrella activities.* Typical activities in this category include:

- Software project tracking and control
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Document preparation and production
- Reusability management
- Measurement
- Risk management

Umbrella activities

Umbrella activities are applied throughout the software process and are discussed in Parts Two and Five of this book.

2.2 THE SOFTWARE PROCESS

A software process can be characterized as shown in Figure 2.2. A *common process framework* is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of *task sets*—each a collection of software engineering work tasks, project milestones,

1 The term *legacy programs* is a euphemism for older, often poorly designed and documented software that is business critical and must be supported over many years.

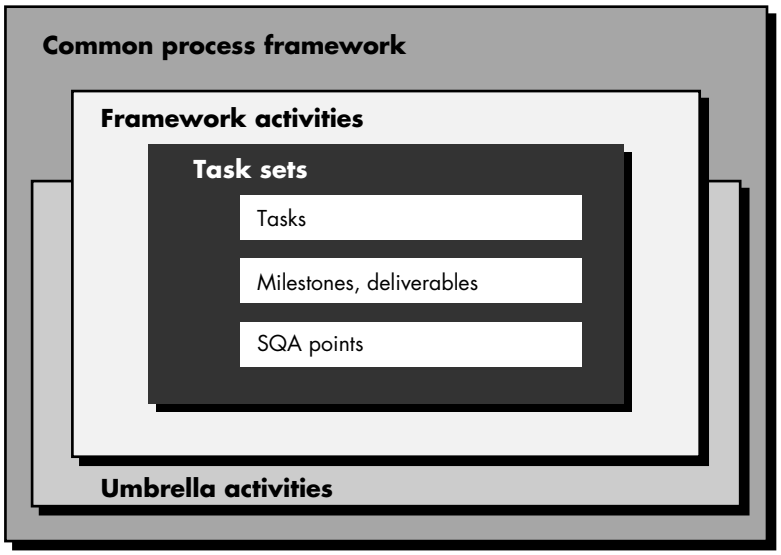


FIGURE 2.2 The software process

work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement[2]—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

Select a common process framework that is tuned to the product, the people, and the project.

In recent years, there has been a significant emphasis on "process maturity." The Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization's current state of process maturity, the SEI uses an assessment that results in a five point grading scheme. The grading scheme determines compliance with a *capability maturity model* (CMM) [PAU93] that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial. The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

2 These topics are discussed in detail in later chapters.

The SEI offers a wide array of process-related information at **www.sei.cmu.edu**

Level 3: Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organizationwide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

Level 4: Managed. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing. Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

Every organization should strive to achieve the intent of the SEI CMM. However, implementing every aspect of the model may be overkill in your situation.

The SEI has associated key process areas (KPAs) with each of the maturity levels. The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

- *Goals*—the overall objectives that the KPA must achieve.
- *Commitments*—requirements (imposed on the organization) that must be met to achieve the goals or provide proof of intent to comply with the goals.
- *Abilities*—those things that must be in place (organizationally and technically) to enable the organization to meet the commitments.
- *Activities*—the specific tasks required to achieve the KPA function.
- *Methods for monitoring implementation*—the manner in which the activities are monitored as they are put into place.
- *Methods for verifying implementation*—the manner in which proper practice for the KPA can be verified.

Eighteen KPAs (each described using these characteristics) are defined across the maturity model and mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level:[3]

Process maturity level 2

- Software configuration management
- Software quality assurance

3 Note that the KPAs are additive. For example, process maturity level 4 contains all level 3 KPAs plus those noted for level 2.

- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

Process maturity level 3

- Peer reviews
- Intergroup coordination
- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

Process maturity level 4

- Software quality management
- Quantitative process management

Process maturity level 5

- Process change management
- Technology change management
- Defect prevention

A tabular version of the complete SEI-CMM, including all goals, commitments, abilities, and activities, is available at **sepo.nosc.mil/CMMmatrices.html**

Each of the KPAs is defined by a set of *key practices* that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines *key indicators* as "those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved." Assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.

2.3 SOFTWARE PROCESS MODELS

"Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."
author unknown

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in Section 2.1.1 and the generic phases discussed in Section 2.1.2. This strategy is often referred to as a *process model* or a *software engineering paradigm.* A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. In an intriguing paper on the nature of the software process, L. B. S. Raccoon [RAC95] uses fractals as the basis for a discussion of the true nature of the software process.

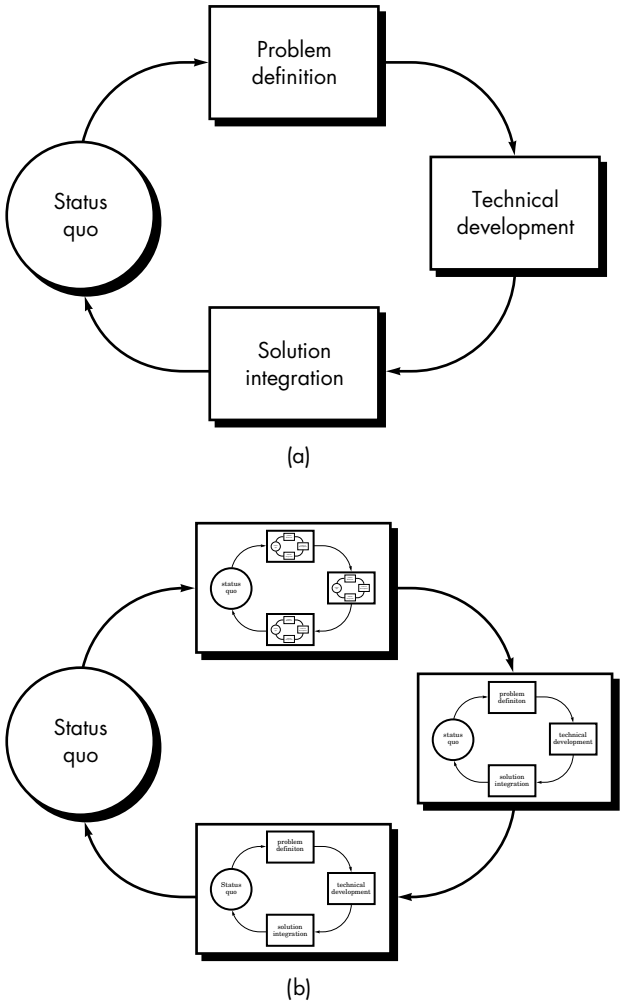


FIGURE 2.3

(a) The phases of a problem solving loop [RAC95]

(b) The phases within phases of the problem solving loop [RAC95]

All software development can be characterized as a problem solving loop (Figure 2.3a) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration. Status quo "represents the current state of affairs" [RAC95]; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results (e.g., documents, programs, data, new business function, new product) to those who requested the solution in the first place. The generic software engineering phases and steps defined in Section 2.1.2 easily map into these stages.

This problem solving loop applies to software engineering work at many different levels of resolution. It can be used at the macro level when the entire application is considered, at a mid-level when program components are being engineered, and

even at the line of code level. Therefore, a fractal[4] representation can be used to provide an idealized view of process. In Figure 2.3b, each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop (this continues to some rational boundary; for software, a line of code).

Realistically, it is difficult to compartmentalize activities as neatly as Figure 2.3b implies because cross talk occurs within and across stages. Yet, this simplified view leads to a very important idea: regardless of the process model that is chosen for a software project, all of the stages—status quo, problem definition, technical development, and solution integration—coexist simultaneously at some level of detail. Given the recursive nature of Figure 2.3b, the four stages discussed apply equally to the analysis of a complete application and to the generation of a small segment of code.

KEY POINT

All stages of a software process—status quo, problem definition, technical development, and solution integration—coexist simultaneously at some level of detail.

Raccoon [RAC95] suggests a "Chaos model" that describes "software development [as] a continuum from the user to the developer to the technology." As work progresses toward a complete system, the stages are applied recursively to user needs and the developer's technical specification of the software.

In the sections that follow, a variety of different process models for software engineering are discussed. Each represents an attempt to bring order to an inherently chaotic activity. It is important to remember that each of the models has been characterized in a way that (ideally) assists in the control and coordination of a real software project. And yet, at their core, all of the models exhibit characteristics of the Chaos model.

2.4 THE LINEAR SEQUENTIAL MODEL

Sometimes called the *classic life cycle* or the *waterfall model,* the *linear sequential model* suggests a systematic, sequential approach[5] to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure 2.4 illustrates the linear sequential model for software engineering. Modeled after a conventional engineering cycle, the linear sequential model encompasses the following activities:

System/information engineering and modeling. Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interact with other elements such as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level

4 Fractals were originally proposed for geometric representations. A pattern is defined and then applied recursively at successively smaller scales; patterns fall inside patterns.

5 Although the original waterfall model proposed by Winston Royce [ROY70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

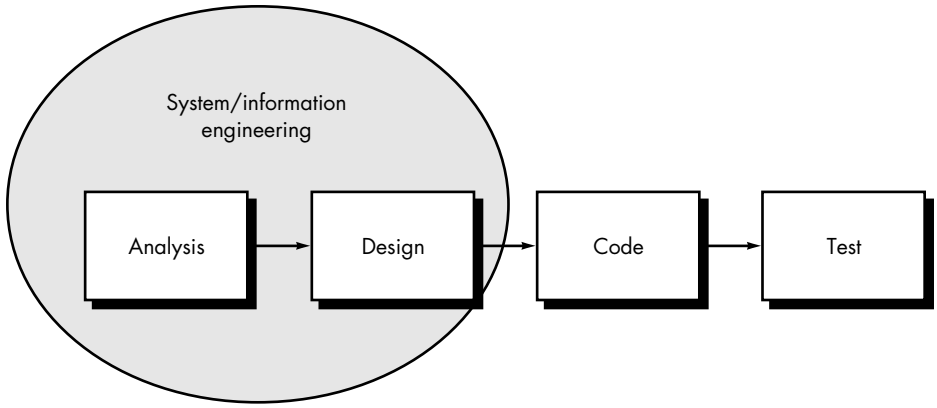


FIGURE 2.4
The linear sequential model

design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

Software requirements analysis. The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain (described in Chapter 11) for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

Design. Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

Although the linear model is often derided as "old fashioned," it remains a reasonable approach when requirements are well understood.

Code generation. The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing. Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Support. Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy [HAN95]. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

? Why does the linear model sometimes fail?

In an interesting analysis of actual projects Bradac [BRA94], found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

2.5 THE PROTOTYPING MODEL

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

The prototyping paradigm (Figure 2.5) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of

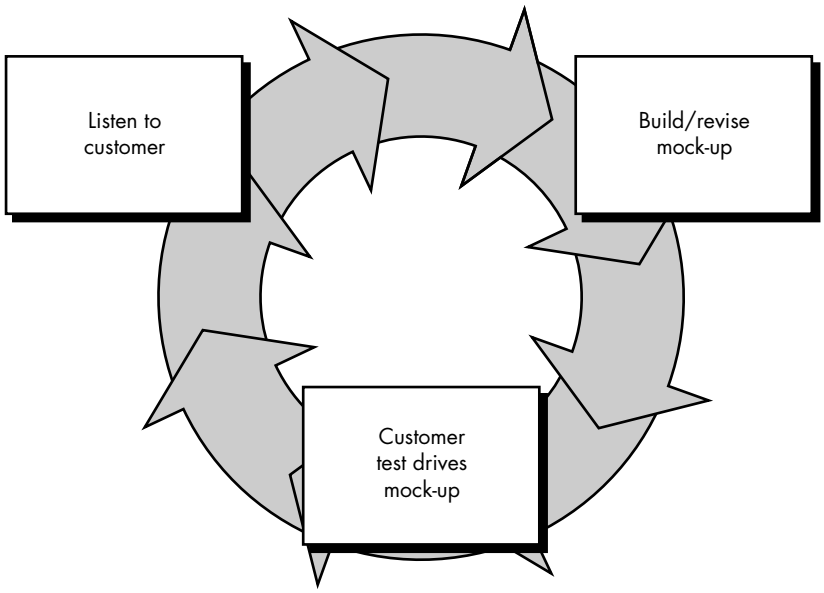


FIGURE 2.5
The prototyping paradigm

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

a prototype. The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.

But what do we do with the prototype when it has served the purpose just described? Brooks [BRO75] provides an answer:

> In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved . . . When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers . . .

The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system and

developers get to build something immediately. Yet, prototyping can also be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire," unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

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

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability.

2.6 THE RAD MODEL

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days) [MAR91]. Used primarily for information systems applications, the RAD approach encompasses the following phases [KER94]:

Business modeling. The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it? Business modeling is described in more detail in Chapter 10.

Data modeling. The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The char-

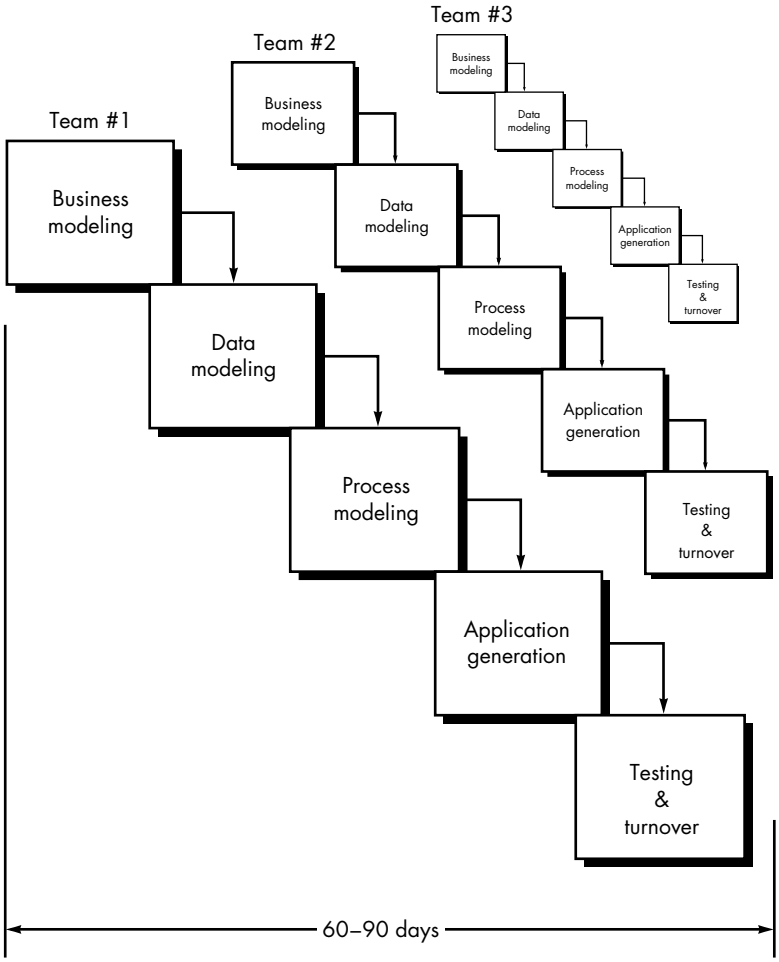


FIGURE 2.6
The RAD model

acteristics (called *attributes*) of each object are identified and the relationships between these objects defined. Data modeling is considered in Chapter 12.

Process modeling. The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation. RAD assumes the use of fourth generation techniques (Section 2.10). Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover. Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

The RAD process model is illustrated in Figure 2.6. Obviously, the time constraints imposed on a RAD project demand "scalable scope" [KER94]. If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

XRef

RAD makes heavy use of reusable components. For further information on component-based development, see Chapter 27.

Like all process models, the RAD approach has drawbacks [BUT94]:

- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.
- Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

2.7 EVOLUTIONARY SOFTWARE PROCESS MODELS

There is growing recognition that software, like all complex systems, evolves over a period of time [GIL88]. Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

The linear sequential model (Section 2.4) is designed for straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model (Section 2.5) is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

2.7.1 The Incremental Model

The *incremental model* combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. Referring to Figure 2.7, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software [MDE93]. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

The incremental model delivers software in small but usable pieces, called "increments." In general, each increment builds on those that have already been delivered.

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

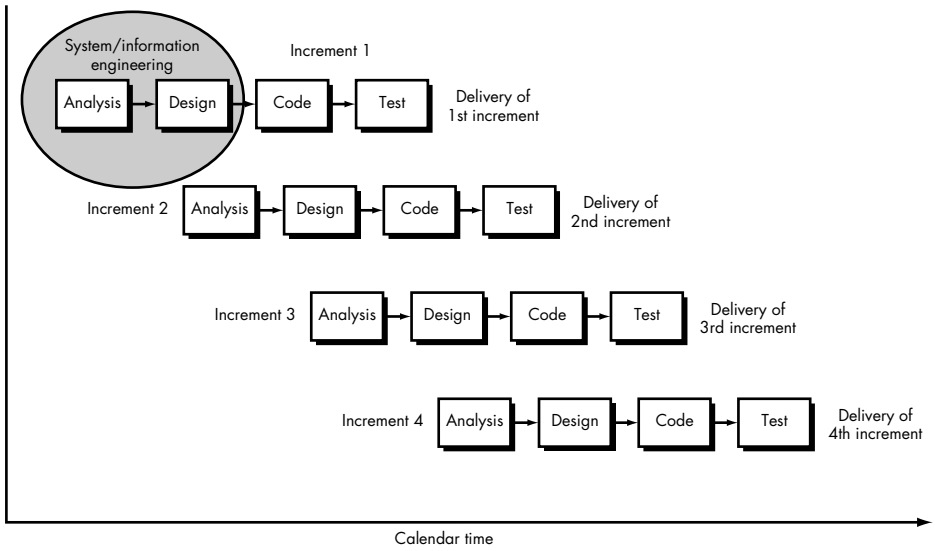


FIGURE 2.7 The incremental model

When you encounter a difficult deadline that cannot be changed, the incremental model is a good paradigm to consider.

The incremental process model, like prototyping (Section 2.5) and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

2.7.2 The Spiral Model

The *spiral model,* originally proposed by Boehm [BOE88], is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called *task regions.*[6] Typically, there are between three and six task regions. Figure 2.8 depicts a spiral model that contains six task regions:

Framework activities apply to every software project you undertake, regardless of size or complexity.

- **Customer communication**—tasks required to establish effective communication between developer and customer.
- **Planning**—tasks required to define resources, timelines, and other project-related information.
- **Risk analysis**—tasks required to assess both technical and management risks.
- **Engineering**—tasks required to build one or more representations of the application.
- **Construction and release**—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

6 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [BOE88]. More recent discussion of Boehm's spiral model can be found in [BOE98].

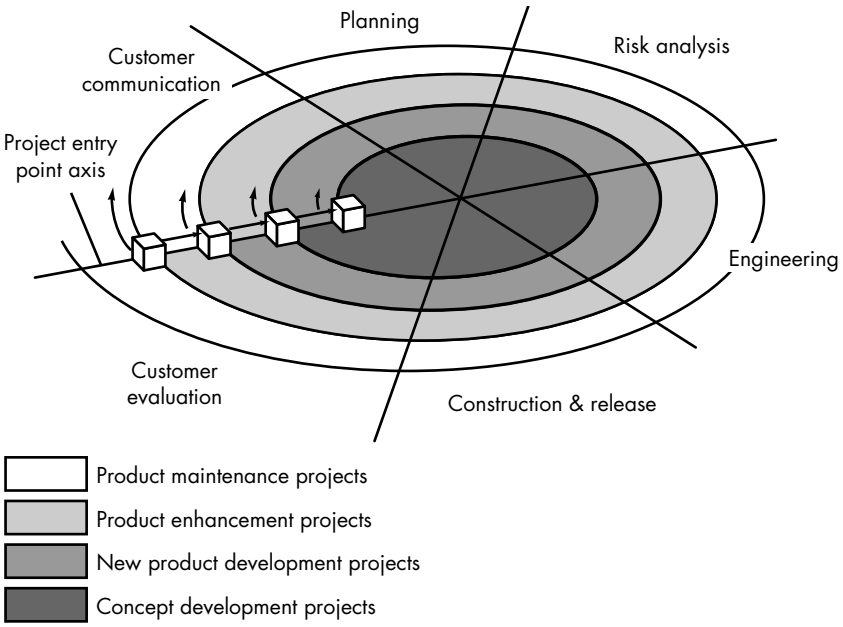


FIGURE 2.8
A typical spiral model

- **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each of the regions is populated by a set of work tasks, called a *task set,* that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality. In all cases, the umbrella activities (e.g., software configuration management and software quality assurance) noted in Section 2.2 are applied.

What is a "task set"?

As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Adaptable process model

Unlike classical process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. An alternative view of the spiral model can be considered by examining the *project entry point axis,* also shown in Figure 2.8. Each cube placed along the axis can be used to represent the starting point for different types of projects. A "concept development

project" starts at the core of the spiral and will continue (multiple iterations occur along the spiral path that bounds the central shaded region) until concept development is complete. If the concept is to be developed into an actual product, the process proceeds through the next cube (new product development project entry point) and a "new development project" is initiated. The new product will evolve through a number of iterations around the spiral, following the path that bounds the region that has somewhat lighter shading than the core. In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

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

XRef

Evolutionary models, such as the spiral model, are particularly well-suited to the development of object-oriented systems. See Part Four for details.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

2.7.3 The WINWIN Spiral Model

The spiral model discussed in Section 2.7.2 suggests a framework activity that addresses customer communication. The objective of this activity is to elicit project requirements from the customer. In an ideal context, the developer simply asks the customer what is required and the customer provides sufficient detail to proceed. Unfortunately, this rarely happens. In reality, the customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market.

KEY POINT

Eliciting software requirements demands negotiation. Successful negotiation occurs when both sides "win".

The best negotiations strive for a "win-win" result.[7] That is, the customer wins by getting the system or product that satisfies the majority of the customer's needs and the developer wins by working to realistic and achievable budgets and deadlines.

---

7 Dozens of books have been written on negotiating skills (e.g., [FIS91], [DON96], [FAR97]). It is one of the more important things that a young (or old) engineer or manager can learn. Read one.

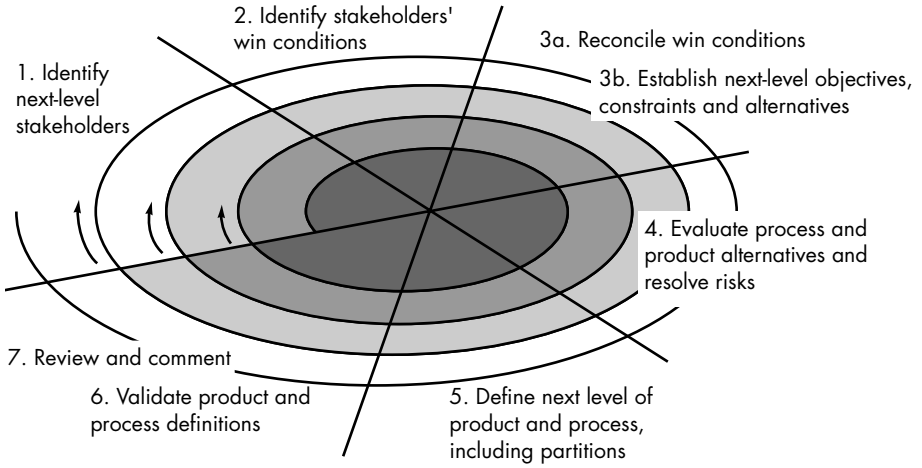


FIGURE 2.9 The WINWIN spiral model [BOE98].

Boehm's WINWIN spiral model [BOE98] defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following activities are defined:

Negotiating skills

1. Identification of the system or subsystem's key "stakeholders."[8]
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition. The WINWIN spiral model is illustrated in Figure 2.9.

In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestones, called *anchor points* [BOE96], that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.

In essence, the anchor points represent three different views of progress as the project traverses the spiral. The first anchor point, *life cycle objectives* (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements. The second anchor point, *life cycle architecture* (LCA), establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions. *Initial operational capability* (IOC) is the third

8 A stakeholder is anyone in the organization that has a direct business interest in the system or product to be built and will be rewarded for a successful outcome or criticized if the effort fails.

anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.

2.7.4 The Concurrent Development Model

The concurrent development model, sometimes called *concurrent engineering,* has been described in the following manner by Davis and Sitaram [DAV94]:

> Project managers who track project status in terms of the major phases [of the classic life cycle] have no idea of the status of their projects. These are examples of trying to track extremely complex sets of activities using overly simple models. Note that although . . . [a large] project is in the coding phase, there are personnel on the project involved in activities typically associated with many phases of development simultaneously. For example, . . . personnel are writing requirements, designing, coding, testing, and integration testing [all at the same time]. Software engineering process models by Humphrey and Kellner [[HUM89], [KEL89]] have shown the concurrency that exists for activities occurring during any one phase. Kellner's more recent work [KEL91] uses statecharts [a notation that represents the states of a process] to represent the concurrent relationship existent among activities associated with a specific event (e.g., a requirements change during late development), but fails to capture the richness of concurrency that exists across all software development and management activities in the project. . . . Most software development process models are driven by time; the later it is, the later in the development process you are. [A concurrent process model] is driven by user needs, management decisions, and review results.

The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For example, the engineering activity defined for the spiral model (Section 2.7.2) is accomplished by invoking the following tasks: prototyping and/or analysis modeling, requirements specification, and design.[9]

Figure 2.10 provides a schematic representation of one activity with the concurrent process model. The activity—analysis—may be in any one of the states[10] noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the *customer communication* activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The *analysis* activity (which existed in the **none** state while initial customer communication was completed) now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the *analysis* activity moves from the **under development** state into the **awaiting changes** state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example,

---

9 It should be noted that analysis and design are complex tasks that require substantial discussion. Parts Three and Four of this book consider these topics in detail.

10 A state is some externally observable mode of behavior.

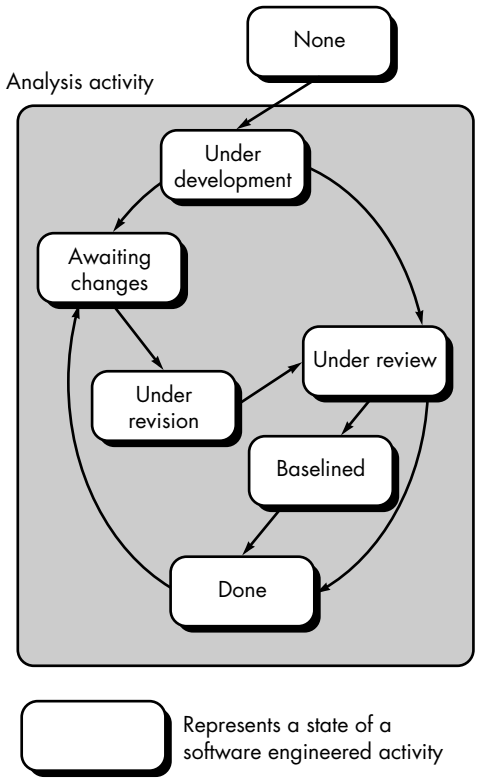


FIGURE 2.10
One element of the concurrent process model

during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event *analysis model correction* which will trigger the *analysis* activity from the **done** state into the **awaiting changes** state.

The concurrent process model is often used as the paradigm for the development of client/server[11] applications (Chapter 28). A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions [SHE94]: a system dimension and a component dimension. System level issues are addressed using three activities: design, assembly, and use. The component dimension is addressed with two activities: design and realization. Concurrency is achieved in two ways: (1) system and component activities occur simultaneously and can be modeled using the state-oriented approach described previously; (2) a typical client/server application is implemented with many components, each of which can be designed and realized concurrently.

In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than

11 In a client/server applications, software functionality is divided between clients (normally PCs) and a server (a more powerful computer) that typically maintains a centralized database.

confining software engineering activities to a sequence of events, it defines a network of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the states of an activity.

2.8 COMPONENT-BASED DEVELOPMENT

Object-oriented technologies (Part Four of this book) provide the technical framework for a component-based process model for software engineering. The object-oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithms used to manipulate the data. If properly designed and implemented, object-oriented classes are reusable across different applications and computer-based system architectures.

XRef

The underlying technology for CBD is discussed in Part Four of this book. A more detailed discussion of the CBD process is presented in Chapter 27.

The component-based development (CBD) model (Figure 2.11) incorporates many of the characteristics of the spiral model. It is evolutionary in nature [NIE92], demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components (called *classes*).

The engineering activity begins with the identification of candidate classes. This is accomplished by examining the data to be manipulated by the application and the algorithms that will be applied to accomplish the manipulation.[12] Corresponding data and algorithms are packaged into a class.

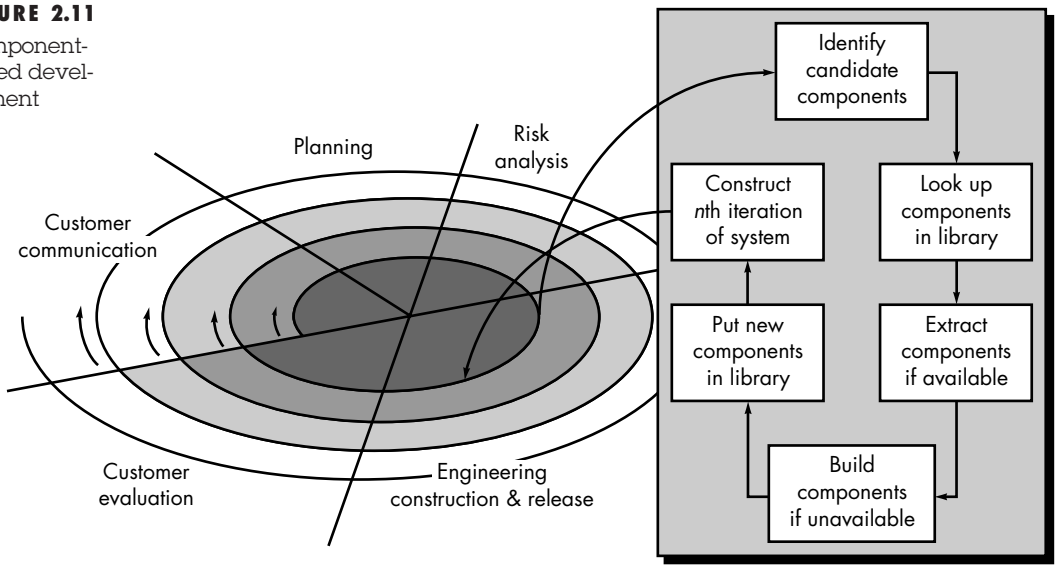


FIGURE 2.11 Component-based development

12 This is a simplified description of class definition. For a more detailed discussion, see Chapter 20.

Classes created in past software engineering projects are stored in a class library or repository (Chapter 31). Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods (Chapters 21–23). The first iteration of the application to be built is then composed, using classes extracted from the library and any new classes built to meet the unique needs of the application. Process flow then returns to the spiral and will ultimately re-enter the component assembly iteration during subsequent passes through the engineering activity.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability, QSM Associates, Inc., reports component assembly leads to a 70 percent reduction in development cycle time; an 84 percent reduction in project cost, and a productivity index of 26.2, compared to an industry norm of 16.9. [YOU94] Although these results are a function of the robustness of the component library, there is little question that the component-based development model provides significant advantages for software engineers.

XRef

UML is discussed in some detail in Chapters 21 and 22.

The *unified software development process* [JAC99] is representative of a number of component-based development models that have been proposed in the industry. Using the *Unified Modeling Language* (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components. Using a combination of iterative and incremental development, the unified process defines the function of the system by applying a scenario-based approach (from the user point of view). It then couples function with an architectural framework that identifies the form the the software will take.

2.9 THE FORMAL METHODS MODEL

XRef

Formal methods are discussed in Chapters 25 and 26.

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [MIL87, DYE92], is currently applied by some software development organizations.

When formal methods (Chapters 25 and 26) are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily, not through ad hoc review but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and

therefore enable the software engineer to discover and correct errors that might go undetected.

Although it is not destined to become a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, the following concerns about its applicability in a business environment have been voiced:

1. The development of formal models is currently quite time consuming and expensive.
2. Because few software developers have the necessary background to apply formal methods, extensive training is required.
3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

2.10 FOURTH GENERATION TECHNIQUES

The term *fourth generation techniques* (4GT) encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level. The tool then automatically generates source code based on the developer's specification. There is little debate that the higher the level at which software can be specified to a machine, the faster a program can be built. The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand.

Currently, a software development environment that supports the 4GT paradigm includes some or all of the following tools: nonprocedural languages for database query, report generation, data manipulation, screen interaction and definition, code generation; high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for Web-site creation using advanced software tools. Initially, many of the tools noted previously were available only for very specific application domains, but today 4GT environments have been extended to address most software application categories.

Like other paradigms, 4GT begins with a requirements gathering step. Ideally, the customer would describe requirements and these would be directly translated into an operational prototype. But this is unworkable. The customer may be unsure of what is required, may be ambiguous in specifying facts that are known, and may be unable or unwilling to specify information in a manner that a 4GT tool can consume.

For this reason, the customer/developer dialog described for other process models remains an essential part of the 4GT approach.

Even though you use a 4GT, you still have to emphasize solid software engineering by doing analysis, design, and testing.

For small applications, it may be possible to move directly from the requirements gathering step to implementation using a nonprocedural fourth generation language (4GL) or a model composed of a network of graphical icons. However, for larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used. The use of 4GT without design (for large projects) will cause the same difficulties (poor quality, poor maintainability, poor customer acceptance) that have been encountered when developing software using conventional approaches.

Implementation using a 4GL enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. Obviously, a data structure with relevant information must exist and be readily accessible by the 4GL.

To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

Like all software engineering paradigms, the 4GT model has advantages and disadvantages. Proponents claim dramatic reduction in software development time and greatly improved productivity for people who build software. Opponents claim that current 4GT tools are not all that much easier to use than programming languages, that the resultant source code produced by such tools is "inefficient," and that the maintainability of large software systems developed using 4GT is open to question.

There is some merit in the claims of both sides and it is possible to summarize the current state of 4GT approaches:

1. The use of 4GT is a viable approach for many different application areas. Coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problems.
2. Data collected from companies that use 4GT indicate that the time required to produce software is greatly reduced for small and intermediate applications and that the amount of design and analysis for small applications is also reduced.
3. However, the use of 4GT for large software development efforts demands as much or more analysis, design, and testing (software engineering activities) to achieve substantial time savings that result from the elimination of coding.

To summarize, fourth generation techniques have already become an important part of software engineering. When coupled with component-based development approaches (Section 2.8), the 4GT paradigm may become the dominant approach to software development.

2.11 PROCESS TECHNOLOGY

The process models discussed in the preceding sections must be adapted for use by a software project team. To accomplish this, process technology tools have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality [BAN95].

Process technology tools allow a software organization to build an automated model of the common process framework, task sets, and umbrella activities discussed in Section 2.3. The model, normally represented as a network, can then be analyzed to determine typical work flow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering tasks defined as part of the process model. Each member of a software project team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other computer-aided software engineering tools (Chapter 31) that are appropriate for a particular work task.

2.12 PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer, but an obsessive overreliance on process is also dangerous. In a brief essay, Margaret Davis [DAV95] comments on the duality of product and process:

> About every ten years, give or take five, the software community redefines "the problem" by shifting its focus from product issues to process issues. Thus, we have embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process).
>
> While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve "the problem" for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.
>
> There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920's when Louis de Broglie proposed it. I believe that the observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product . . .

Quote:

"[If it is developed thoughtlessly and applied mindlessly, process can become] the death of common sense."

Philip K. Howard

All of human activity may be a process, but each of us derives a sense of self worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.

Quote:

"Any activity becomes creative when the doer cares about doing it right, or doing it better."

John Updike

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. A creative software professional should also derive as much satisfaction from the process as the end-product.

The work of software people will change in the years ahead. The duality of product and process is one important element in keeping creative people engaged as the transition from programming to software engineering is finalized.

2.13 SUMMARY

Software engineering is a discipline that integrates process, methods, and tools for the development of computer software. A number of different process models for software engineering have been proposed, each exhibiting strengths and weaknesses, but all having a series of generic phases in common. The principles, concepts, and methods that enable us to perform the process that we call *software engineering* are considered throughout the remainder of this book.

REFERENCES

[BAE98] Baetjer, Jr., H., *Software as Capital,* IEEE Computer Society Press, 1998, p. 85.

[BAN95] Bandinelli, S. et al., "Modeling and Improving an Industrial Software Process," *IEEE Trans. Software Engineering,* vol. SE-21, no. 5, May 1995, pp. 440–454.

[BOE88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer,* vol. 21, no. 5, May 1988, pp. 61–72.

[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[BOE98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer,* vol. 31, no. 7, July 1998, pp. 33–44.

[BRA94] Bradac, M., D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Trans. Software Engineering,* vol. SE-20, no. 10, October 1994, pp. 774–784.

[BRO75] Brooks, F., *The Mythical Man-Month,* Addison-Wesley, 1975.

[BUT94] Butler, J., "Rapid Application Development in Action," *Managing System Development,* Applied Computer Research, vol. 14, no. 5, May 1994, pp. 6–8.

[DAV94] Davis, A. and P. Sitaram, "A Concurrent Process Model for Software Development," *Software Engineering Notes,* ACM Press, vol. 19, no. 2, April 1994, pp. 38–51.

[DAV95] Davis, M.J., "Process and Product: Dichotomy or Duality," *Software Engineering Notes,* ACM Press, vol. 20, no. 2, April 1995, pp. 17–18.

[DON96] Donaldson, M.C. and M. Donaldson, *Negotiating for Dummies,* IDG Books Worldwide, 1996.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development,* Wiley, 1992.

[FAR97] Farber, D.C., *Common Sense Negotiation: The Art of Winning Gracefully,* Bay Press, 1997.

[FIS91] Fisher, R., W. Ury, and B. Patton, *Getting to Yes: Negotiating Agreement Without Giving In,* 2nd edition, Penguin USA, 1991.

[GIL88] Gilb, T., *Principles of Software Engineering Management,* Addison-Wesley, 1988.

[HAN95] Hanna, M., "Farewell to Waterfalls," *Software Magazine,* May 1995, pp. 38–46.

[HUM89] Humphrey, W. and M. Kellner, "Software Process Modeling: Principles of Entity Process Models," *Proc. 11th Intl. Conference on Software Engineering,* IEEE Computer Society Press, pp. 331–342.

[IEE93] *IEEE Standards Collection: Software Engineering,* IEEE Standard 610.12—1990, IEEE, 1993.

[JAC99] Jacobson, I, Booch, G., and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.

[KEL89] Kellner, M., *Software Process Modeling: Value and Experience,* SEI Technical Review—1989, SEI, 1989.

[KEL91] Kellner, M., "Software Process Modeling Support for Management Planning and Control," *Proc. 1st Intl. Conf. on the Software Process,* IEEE Computer Society Press, 1991, pp. 8–28.

[KER94] Kerr, J. and R. Hunter, *Inside RAD,* McGraw-Hill, 1994.

[MAR91] Martin, J., *Rapid Application Development,* Prentice-Hall, 1991.

[MDE93] McDermid, J. and P. Rook, "Software Development Process Models," in *Software Engineer's Reference Book,* CRC Press, 1993, pp. 15/26–15/28.

[MIL87] Mills, H.D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software,* September 1987, pp. 19–25.

[NAU69] Naur, P. and B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee,* NATO, 1969.

[NIE92] Nierstrasz, O., S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *CACM,* vol. 35, no. 9, September 1992, pp. 160–165.

[PAU93] Paulk, M. et al., "Capability Maturity Model for Software," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[RAC95] Raccoon, L.B.S., "The Chaos Model and the Chaos Life Cycle," *ACM Software Engineering Notes,* vol. 20., no. 1, January, 1995, pp. 55–66.

[ROY70] Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. WESCON,* August 1970.

[SHE94] Sheleg, W., "Concurrent Engineering: A New Paradigm for C/S Development," *Application Development Trends,* vol. 1, no. 6, June 1994, pp. 28–33.

[YOU94] Yourdon, E., "Software Reuse," *Application Development Strategies,* vol. 6, no. 12, December 1994, pp. 1–16.

PROBLEMS AND POINTS TO PONDER

2.1. Figure 2.1 places the three software engineering layers on top of a layer entitled *a quality focus.* This implies an organization quality program such as Total Quality Management. Do a bit of research and develop an outline of the key tenets of a Total Quality Management program.

2.2. Is there ever a case when the generic phases of the software engineering process don't apply? If so, describe it.

2.3. The SEI's capability maturity model is an evolving document. Do some research and determine if any new KPAs have been added since the publication of this book.

2.4. The Chaos model suggests that a problem solving loop can be applied at any degree of resolution. Discuss the way in which you would apply the loop to (1) understand requirements for a word-processing product; (2) develop an advanced spelling/grammar checking component for the word processor; (3) generate code for a program module that determines the subject, predicate, and object in an English language sentence.

2.5. Which of the software engineering paradigms presented in this chapter do you think would be most effective? Why?

2.6. Provide five examples of software development projects that would be amenable to prototyping. Name two or three applications that would be more difficult to prototype.

2.7. The RAD model is often tied to CASE tools. Research the literature and provide a summary of a typical CASE tool that supports RAD.

2.8. Propose a specific software project that would be amenable to the incremental model. Present a scenario for applying the model to the software.

2.9. As you move outward along the process flow path of the spiral model, what can you say about the software that is being developed or maintained?

2.10. Many people believe that the only way in which order of magnitude improvements in software quality and productivity will be achieved is through component-based development. Find three or four recent papers on the subject and summarize them for the class.

2.11. Describe the concurrent development model in your own words.

2.12. Provide three examples of fourth generation techniques.

2.13. Which is more important—the product or the process?

FURTHER READINGS AND INFORMATION SOURCES

The current state of the art in software engineering can best be determined from monthly publications such as *IEEE Software, Computer,* and the *IEEE Transactions on Software Engineering.* Industry periodicals such as *Application Development Trends, Cutter IT Journal* and *Software Development* often contain articles on software engineering topics. The discipline is 'summarized' every year in the *Proceedings of the International Conference on Software Engineering,* sponsored by the IEEE and ACM and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes,* and *Annals of Software Engineering.*

Many software engineering books have been published in recent years. Some present an overview of the entire process while others delve into a few important topics to the exclusion of others. Three anthologies that cover a wide range of software engineering topics are

Keyes, J., (ed.), *Software Engineering Productivity Handbook,* McGraw-Hill, 1993.

McDermid, J., (ed.), *Software Engineer's Reference Book,* CRC Press, 1993.

Marchiniak, J.J. (ed.), *Encyclopedia of Software Engineering,* Wiley, 1994.

Gautier (*Distributed Engineering of Software,* Prentice-Hall, 1996) provides suggestions and guidelines for organizations that must develop software across geographically dispersed locations.

On the lighter side, a book by Robert Glass (*Software Conflict,* Yourdon Press, 1991) presents amusing and controversial essays on software and the software engineering process. Pressman and Herron (*Software Shock,* Dorset House, 1991) consider software and its impact on individuals, businesses, and government.

The Software Engineering Institute (SEI is located at Carnegie-Mellon University) has been chartered with the responsibility of sponsoring a software engineering monograph series. Practitioners from industry, government, and academia are contribut-

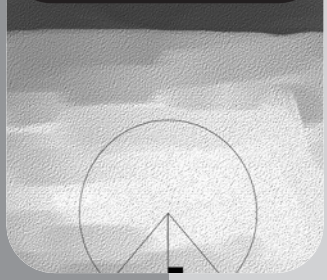
ing important new work. Additional software engineering research is conducted by the Software Productivity Consortium.

A wide variety of software engineering standards and procedures have been published over the past decade. The *IEEE Software Engineering Standards* contains standards that cover almost every important aspect of the technology. ISO 9000-3 guidelines provide guidance for software organizations that require registration to the ISO 9001 quality standard. Other software engineering standards can be obtained from the Department of Defense, the FAA, and other government and nonprofit agencies. Fairclough (*Software Engineering Guides,* Prentice-Hall, 1996) provides a detailed reference to software engineering standards produced by the European Space Agency (ESA).

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

PART

Two



MANAGING SOFTWARE PROJECTS

In this part of *Software Engineering: A Practitioner's Approach,* we consider the management techniques required to plan, organize, monitor, and control software projects. In the chapters that follow, you'll get answers to the following questions:

- How must the people, process, and problem be managed during a software project?
- What are software metrics and how can they be used to manage a software project and the software process?
- How does a software team generate reliable estimates of effort, cost, and project duration?
- What techniques can be used to formally assess the risks that can have an impact on project success?
- How does a software project manager select the set of software engineering work tasks?
- How is a project schedule created?
- How is quality defined so that it can be controlled?
- What is software quality assurance?
- Why are formal technical reviews so important?
- How is change managed during the development of computer software and after delivery to the customer?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product.

CHAPTER

3 PROJECT MANAGEMENT CONCEPTS

KEY CONCEPTS

In the preface to his book on software project management, Meiler Page-Jones [PAG85] makes a statement that can be echoed by many software engineering consultants:

> I've visited dozens of commercial shops, both good and bad, and I've observed scores of data processing managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be conducted for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and the six that follow, we consider the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 4 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost and resource requirements and establish an effective project plan are discussed in Chapter 5. The man-

QUICK LOOK

What is it? Although many of us (in our darker moments) take Dilbert's view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to an operational implementation.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies with the person doing it. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and the software professionals.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four P's—people, product, process, and project. People must be organized to perform software work effectively. Communication with the customer must occur so that product scope and requirements are understood. A process must be selected that is appropriate for the people and the product. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and

QUICK LOOK

establishing mechanisms to monitor and control work defined by the plan.

What is the work product? A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until you've delivered a high-quality product on time and within budget. However, a project manager does it right when he encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

agement activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 6. Chapter 7 discusses the activities that are required to define project tasks and establish a workable project schedule. Finally, Chapters 8 and 9 consider techniques for ensuring quality as a project is conducted and controlling changes throughout the life of an application.

3.1 THE MANAGEMENT SPECTRUM

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.

3.1.1 The People

Quote:

"There exists enormous variability in the ability of different people to perform programming tasks."

Bill Curtis

The cultivation of motivated, highly skilled software people has been discussed since the 1960s (e.g., [COU80], [WIT94], [DEM98]). In fact, the "people factor" is so important that the Software Engineering Institute has developed a *people management capability maturity model* (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability" [CUR94].

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The PM-CMM is a companion to the software capability maturity model (Chapter 2) that guides organizations in the creation of a mature software process. Issues

associated with people management and structure for software projects are considered later in this chapter.

3.1.2 The Product

XRef

A taxonomy of application areas that spawn software "products" is discussed in Chapter 1.

Before a project can be planned, product[1] objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

The software developer and customer must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering (Chapter 10) and continues as the first step in software requirements analysis (Chapter 11). Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved. Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to *bound* these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

3.1.3 The Process

Framework activities are populated with tasks, milestones, work products, and quality assurance points.

A software process (Chapter 2) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

3.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, we still struggle. In 1998, industry data indicated that 26 percent of software projects failed outright and 46 percent experienced cost and schedule overruns [REE99]. Although the success rate for

---

1 In this context, the term *product* is used to encompass any software that is to be built at the request of others. It includes not only software products but also computer-based systems, embedded software, and problem-solving software (e.g., programs for engineering/scientific problem solving).

software projects has improved somewhat, our project failure rate remains higher than it should be.[2]

In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project. Each of these issues is discussed in Section 3.5 and in the chapters that follow.

3.2 PEOPLE

In a study published by the IEEE [CUR88], the engineering vice presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way:

Quote:

"Companies that sensibly manage their investment in people will prosper in the long run."

Tom DeMarco & Tim Lister

VP 1: I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.

VP 2: The most important ingredient that was successful on this project was having smart people . . . very little else matters in my opinion. . . . The most important thing you do for a project is selecting the staff . . . The success of the software development organization is very, very much associated with the ability to recruit good people.

VP 3: The only rule I have in management is to ensure I have good people—real good people—and that I grow good people—and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section we examine the players who participate in the software process and the manner in which they are organized to perform effective software engineering.

3.2.1 The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies:

1. **Senior managers** who define the business issues that often have significant influence on the project.

2 Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially and the software industry continues to post double digit sales growth. Part of the answer, I think, is that a substantial number of these "failed" projects are ill-conceived in the first place. Customers lose interest quickly (because what they requested wasn't really as important as they first thought), and the projects are cancelled.

2. **Project (technical) managers** who must plan, motivate, organize, and control the practitioners who do software work.
3. **Practitioners** who deliver the technical skills that are necessary to engineer a product or application.
4. **Customers** who specify the requirements for the software to be engineered and other *stakeholders* who have a peripheral interest in the outcome.
5. **End-users** who interact with the software once it is released for production use.

Every software project is populated by people who fall within this taxonomy. To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

3.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers." [EDG95]

? What do we look for when we select someone to lead a software project?

In an excellent book of technical leadership, Jerry Weinberg [WEI86] suggests a MOI model of leadership:

> **Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.
>
> **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
>
> **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Quote:

"In simplest terms, a leader is one who knows where he wants to go, and gets up, and goes."
John Erskine

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Another view [EDG95] of the characteristics that define an effective project manager emphasizes four key traits:

> **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain

flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

A software wizard may not have the temperament or desire to be a team leader. Don't force the wizard to become one.

3.2.3 The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

"Not every group is a team, and not every team is effective."
Glenn Parker

The following options are available for applying human resources to a project that will require n people working for k years:

1. n individuals are assigned to m different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.
2. n individuals are assigned to m different functional tasks ($m < n$) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager.
3. n individuals are organized into t teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

How should a software team be organized?

Although it is possible to voice arguments for and against each of these approaches, a growing body of evidence indicates that a formal team organization (option 3) is most productive.

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [MAN81] suggests three generic team organizations:

Democratic decentralized (DD). This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

Controlled decentralized (CD). This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC). Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

Mantei [MAN81] describes seven project factors that should be considered when planning the structure of software engineering teams:

? What factors should we consider when structuring a software team?

- The difficulty of the problem to be solved.
- The size of the resultant program(s) in lines of code or function points (Chapter 4).
- The time that the team will stay together (team lifetime).
- The degree to which the problem can be modularized.
- The required quality and reliability of the system to be built.
- The rigidity of the delivery date.
- The degree of sociability (communication) required for the project.

Because a centralized structure completes tasks faster, it is the most adept at handling simple problems. Decentralized teams generate more and better solutions than individuals. Therefore such teams have a greater probability of success when working on difficult problems. Since the CD team is centralized for problem solving, either a CD or CC team structure can be successfully applied to simple problems. A DD structure is best for difficult problems.

It's often better to have a few small, well-focused teams than a single large team.

Because the performance of a team is inversely proportional to the amount of communication that must be conducted, very large projects are best addressed by teams with a CC or CD structures when subgrouping can be easily accommodated.

The length of time that the team will "live together" affects team morale. It has been found that DD team structures result in high morale and job satisfaction and are therefore good for teams that will be together for a long time.

The DD team structure is best applied to problems with relatively low modularity, because of the higher volume of communication needed. When high modularity is possible (and people can do their own thing), the CC or CD structure will work well.

CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurance activities that are applied by the team. Decentralized teams generally require more time to complete a project than a centralized structure and at the same time are best when high sociability is required.

Constantine [CON93] suggests four "organizational paradigms" for software engineering teams:

Quote:

"Working with people is difficult, but not impossible."

Peter Drucker

1. A *closed paradigm* structures a team along a traditional hierarchy of authority (similar to a CC team). Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. The *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. The *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. The *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

As an historical footnote, the earliest software team organization was a controlled centralized (CD) structure originally called the *chief programmer team.* This structure was first proposed by Harlan Mills and described by Baker [BAK72]. The nucleus of the team was composed of a *senior engineer* (the chief programmer), who plans, coordinates and reviews all technical activities of the team; *technical staff* (normally two to five people), who conduct analysis and development activities; and a *backup engineer,* who supports the senior engineer in his or her activities and can replace the senior engineer with minimum loss in project continuity.

XRef

The role of the librarian exists regardless of team structure. See Chapter 9 for details.

The chief programmer may be served by one or more specialists (e.g., telecommunications expert, database designer), support staff (e.g., technical writers, clerical personnel), and a *software librarian.* The librarian serves many teams and performs the following functions: maintains and controls all elements of the software configuration (i.e., documentation, source listings, data, storage media); helps collect and format software productivity data; catalogs and indexes reusable software compo-

nents; and assists the teams in research, evaluation, and document preparation. The importance of a librarian cannot be overemphasized. The librarian acts as a controller, coordinator, and potentially, an evaluator of the software configuration.

A variation on the democratic decentralized team has been proposed by Constantine [CON93], who advocates teams with creative independence whose approach to work might best be termed *innovative anarchy.* Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

Quote:

"No matter what the problem is, it's always a people problem."

Jerry Weinberg

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, *Peopleware,* DeMarco and Lister [DEM98] discuss this issue:

> We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

ADVICE

Jelled teams are the ideal, but they're not easy to achieve. At a minimum, be certain to avoid a "toxic environment."

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

But not all teams jell. In fact, many teams suffer from what Jackman calls "team toxicity" [JAC98]. She defines five factors that "foster a potentially toxic team environment":

1. A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
2. High frustration caused by personal, business, or technological factors that causes friction among team members.
3. "Fragmented or poorly coordinated procedures" or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.

4. Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
5. "Continuous and repeated exposure to failure" that leads to a loss of confidence and a lowering of morale.

Jackman suggests a number of antitoxins that address these all-too-common problems.

? How do we avoid "toxins" that often infect a software team?

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. In addition, bad news should not be kept secret but rather, delivered to the team as early as possible (while there is still time to react in a rational and controlled manner).

Although frustration has many causes, software people often feel it when they lack the authority to control their situation. A software team can avoid frustration if it is given as much responsibility for decision making as possible. The more control over process and technical decisions given to the team, the less frustration the team members will feel.

An inappropriately chosen software process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided in two ways: (1) being certain that the characteristics of the software to be built conform to the rigor of the process that is chosen and (2) allowing the team to select the process (with full recognition that, once chosen, the team has the responsibility to deliver a high-quality product).

Quote:

"Do or do not; there is no try."

Yoda
(Star Wars)

The software project manager, working together with the team, should clearly refine roles and responsibilities before the project begins. The team itself should establish its own mechanisms for accountability (formal technical reviews[3] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform.

Every software team experiences small failures. The key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving. In addition, failure by any member of the team must be viewed as a failure by the team itself. This leads to a team-oriented approach to corrective action, rather than the finger-pointing and mistrust that grows rapidly on toxic teams.

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts, others are introverted. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want

3 Formal technical reviews are discussed in detail in Chapter 8.

a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.[4] However, it is important to note that recognition of human differences is the first step toward creating teams that jell.

3.2.4 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The *scale* of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. *Uncertainty* is common, resulting in a continuing stream of changes that ratchets the project team. *Interoperability* has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively noninteractive and impersonal communication channels" [KRA95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

Kraul and Streeter [KRA95] examine a collection of project coordination techniques that are categorized in the following manner:

How do we coordinate the actions of team members?

Formal, impersonal approaches include software engineering documents and deliverables (including source code), technical memos, project milestones, schedules, and project control tools (Chapter 7), change requests and related documentation (Chapter 9), error tracking reports, and repository data (see Chapter 31).

Formal, interpersonal procedures focus on quality assurance activities (Chapter 8) applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures include group meetings for information dissemination and problem solving and "collocation of requirements and development staff."

4 An excellent introduction to these issues as they relate to software project teams can be found in [FER98].

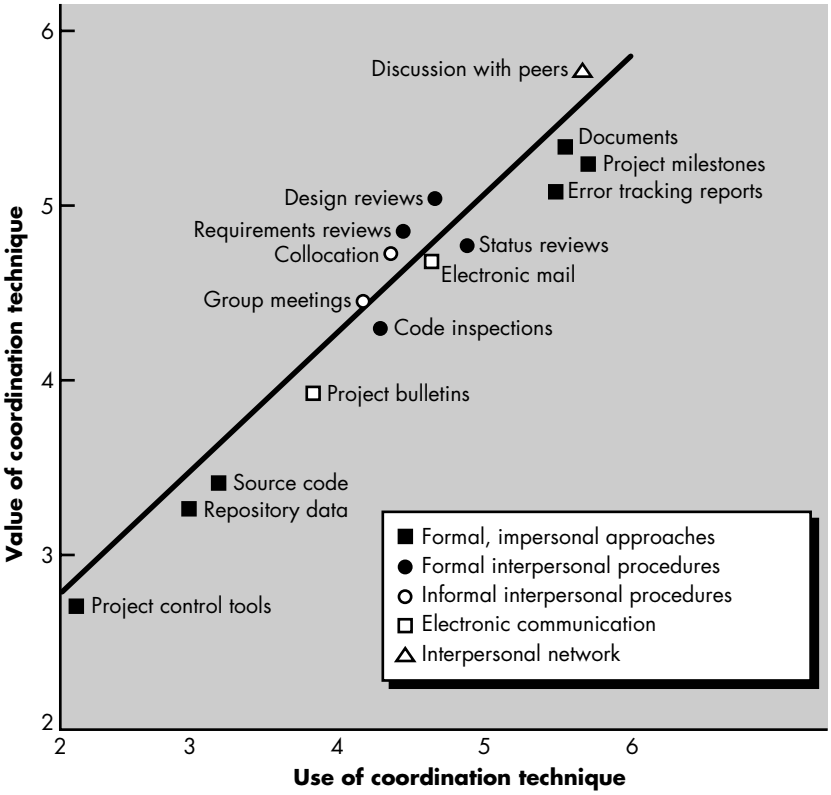


FIGURE 3.1 Value and Use of Coordination and Communication Techniques

Electronic communication encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.

Interpersonal networking includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

To assess the efficacy of these techniques for project coordination, Kraul and Streeter studied 65 software projects involving hundreds of technical staff. Figure 3.1 (adapted from [KRA95]) expresses the value and use of the coordination techniques just noted. Referring to figure, the perceived value (rated on a seven point scale) of various coordination and communication techniques is plotted against their frequency of use on a project. Techniques that fall above the regression line were "judged to be relatively valuable, given the amount that they were used" [KRA95]. Techniques that fell below the line were perceived to have less value. It is interesting to note that interpersonal networking was rated the technique with highest coordination and communication value. It is also important to note that early software quality assurance mechanisms (requirements and design reviews) were perceived to have more value than later evaluations of source code (code inspections).

3.3 THE PRODUCT

A software project manager is confronted with a dilemma at the very beginning of a software engineering project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"

Therefore, we must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

3.3.1 Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

If you can't bound a characteristic of the software you intend to build, list the characteristic as a major project risk.

Context. How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
Information objectives. What customer-visible data objects (Chapter 11) are produced as output from the software? What data objects are required for input?
Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly; constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in C++) are described.

3.3.2 Problem Decomposition

In order to develop a reasonable project plan, you have to functionally decompose the problem to be solved.

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapter 11). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality that must be delivered and (2) the process that will be used to deliver it.

Human beings tend to apply a divide and conquer strategy when they are confronted with a complex problems. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation

(Chapter 5). Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.

XRef

A useful technique for problem decomposition, called a *grammatical parse,* is presented in Chapter 12.

As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as keyboard input, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, document production, and the like). For example, will continuous voice input require that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be?

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), and (4) section and chapter reference validation for large documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier.

3.4 THE PROCESS

The generic phases that characterize the software process—definition, development, and support—are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team. In Chapter 2, a wide array of software engineering paradigms were discussed:

- the linear sequential model
- the prototyping model
- the RAD model
- the incremental model
- the spiral model
- the WINWIN spiral model
- the component-based development model
- the concurrent development model
- the formal methods model
- the fourth generation techniques model

Once the process model is chosen, populate it with the minimum set of work tasks and work products that will result in a high-quality product—avoid process overkill!

The project manager must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work,

(2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of common process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 7.

3.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities (Chapter 2):

Remember : framework activities are applied on every project—no exceptions.

- *Customer communication*—tasks required to establish effective requirements elicitation between developer and customer.
- *Planning*—tasks required to define resources, timelines, and other project-related information.
- *Risk analysis*—tasks required to assess both technical and management risks.
- *Engineering*—tasks required to build one or more representations of the application.
- *Construction and release*—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- *Customer evaluation*—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

Product and process decomposition occur simultaneously as the project plan evolves.

The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 3.2 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.[5] The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapters 5 and 7.

---

5 It should be noted that work tasks must be adapted to the specific needs of a project. Framework activities always remain the same, but work tasks will be selected based on a number of adaptation criteria. This topic is discussed further in Chapter 7 and at the SEPA Web site.

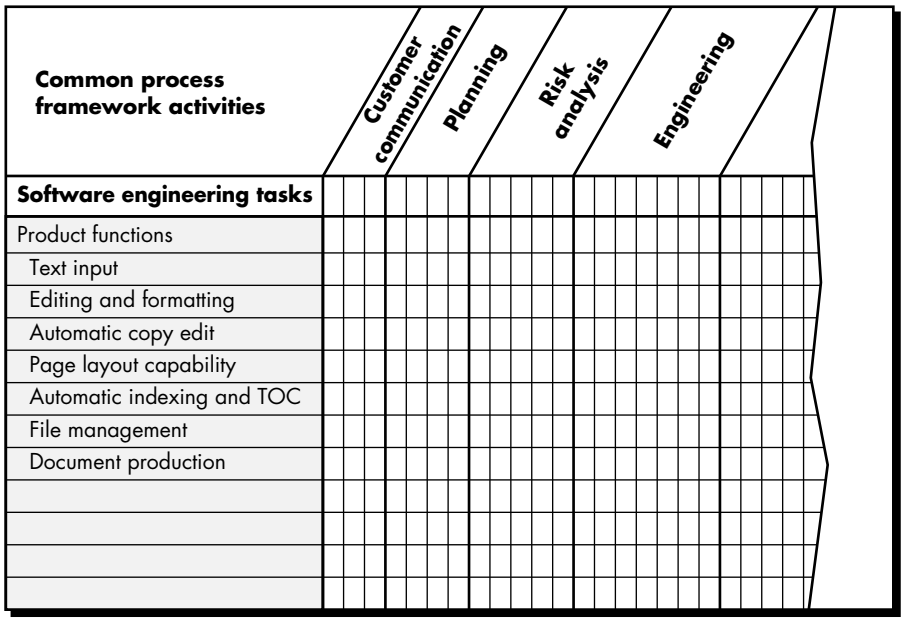


FIGURE 3.2 Melding the Problem and the Process

3.4.2 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software engineering paradigm that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.[6]

Always apply the CPF, regardless of project size, criticality, or type. Work tasks may vary, but the CPF does not.

Once the process model has been chosen, the common process framework (CPF) is adapted to it. In every case, the CPF discussed earlier in this chapter—customer communication, planning, risk analysis, engineering, construction and release, customer evaluation—can be fitted to the paradigm. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The CPF is invariant and serves as the basis for all software work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this CPF activity?" For example, a small,

6 Recall that project characteristics also have a strong bearing on the structure of the team that is to do the work. See Section 3.2.3.

relatively simple project might require the following work tasks for the *customer communication* activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Adaptable process model

Now, we consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document with all concerned.
10. Modify the scoping document as required.

Both projects perform the framework activity that we call "customer communication," but the first project team performed half as many software engineering work tasks as the second.

3.5 THE PROJECT

Quote:

"At least 7 of 10 signs of IS project failures are determined before a design is developed or a line of code is written . . ."

John Reel

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right. In an excellent paper on software projects, John Reel [REE99] defines ten signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.

4. The chosen technology changes.
5. Business needs change [or are ill-defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time [ZAH94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceeding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [REE99] suggests a five-part commonsense approach to software projects:

1. **Start on the right foot.** This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objects and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 3.2.3) and giving the team the autonomy, authority, and technology needed to do the job.
2. **Maintain momentum.** Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.[7]
3. **Track progress.** For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 4) can be collected and used to assess progress against averages developed for the software development organization.
4. **Make smart decisions.** In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are

A broad array of resources that can help both neophyte and experienced project managers can be found at **www.pmi.org, www.4pm.com,** and **www.projectmanagement.com**

7 The implication of this statement is that bureacracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is eliminated. The team should be allowed to do its thing.

available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).

5. **Conduct a postmortem analysis.** Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

3.6 THE W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [BOE96] states: "you need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the WWWWWHH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

What questions need to be answered in order to develop a project plan?

Why is the system being developed? The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done, by when? The answers to these questions help the team to establish a project schedule by identifying key project tasks and the milestones that are required by the customer.

Who is responsible for a function? Earlier in this chapter, we noted that the role and responsibility of each member of the software team must be defined. The answer to this question helps accomplish this.

Where are they organizationally located? Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

Software Project Plan

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 5) based on answers to earlier questions.

Boehm's W^{5}HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide an excellent planning outline for the project manager and the software team.

3.7 CRITICAL PRACTICES

The Airlie Council[8] has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [AIR99]. In an effort to enable a software organization to determine whether a specific project has implemented critical practices, the Airlie Council has developed a set of "QuickLook" questions [AIR99] for a project:[9]

Airlie Project Quicklook

Formal risk management. What are the top ten risks for this project? For each of the risks, what is the chance that the risk will become a problem and what is the impact if it does?

Empirical cost and schedule estimation. What is the current estimated size of the application software (excluding system software) that will be delivered into operation? How was it derived?

Metric-based project management. Do you have in place a metrics program to give an early indication of evolving problems? If so, what is the current requirements volatility?

Earned value tracking. Do you report monthly earned value metrics? If so, are these metrics computed from an activity network of tasks for the entire effort to the next delivery?

Defect tracking against quality targets. Do you track and periodically report the number of defects found by each inspection (formal technical review) and execution test from program inception and the number of defects currently closed and open?

People-aware program management. What is the average staff turnover for the past three months for each of the suppliers/developers involved in the development of software for this system?

If a software project team cannot answer these questions or answers them inadequately, a thorough review of project practices is indicated. Each of the critical practices just noted is addressed in detail throughout Part Two of this book.

3.8 SUMMARY

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development, and support of computer software.

---

8 The Airlie Council is a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering.

9 Only those critical practices associated with "project integrity" are noted here. Other best practices will be discussed in later chapters.

Four P's have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. The product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, formal reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation, risk analysis, schedules, tracking, and control. Each of these topics is considered in the chapters that follow.

REFERENCES

[AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.

[BAK72] Baker, F.T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal,* vol. 11, no. 1, 1972, pp. 56–73.

[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization, *CACM,* vol. 36, no. 10, October 1993, pp. 34–43.

[COU80] Cougar, J. and R. Zawacki, *Managing and Motivating Computer Personnel,* Wiley, 1980.

[CUR88] Curtis, B. et al., "A Field Study of the Software Design Process for Large Systems," *IEEE Trans. Software Engineering,* vol. SE-31, no. 11, November 1988, pp. 1268–1287.

[CUR94] Curtis, B., et al., *People Management Capability Maturity Model,* Software Engineering Institute, 1994.

[DEM98] DeMarco, T. and T. Lister, *Peopleware,* 2nd ed., Dorset House, 1998.

[EDG95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends,* vol. 2, no. 5, May 1995, pp. 37–42.

[FER98] Ferdinandi, P.L., "Facilitating Communication," *IEEE Software,* September 1998, pp. 92–96.

[JAC98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software,* July 1998, pp. 43–45.
[KRA95] Kraul, R. and L. Streeter, "Coordination in Software Development," *CACM,* vol. 38, no. 3, March 1995, pp. 69–81.
[MAN81] Mantei, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM,* vol. 24, no. 3, March 1981, pp. 106–113.
[PAG85] Page-Jones, M., *Practical Project Management,* Dorset House, 1985, p. vii.
[REE99] Reel, J.S., "Critical Success Factors in Software Projects, *IEEE Software,* May, 1999, pp. 18–23.
[WEI86] Weinberg, G., *On Becoming a Technical Leader,* Dorset House, 1986.
[WIT94] Whitaker, K., *Managing Software Maniacs,* Wiley, 1994.
[ZAH94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development,* March 1994, pp. 35–38.

PROBLEMS AND POINTS TO PONDER

3.1. Based on information contained in this chapter and your own experience, develop "ten commandments" for empowering software engineers. That is, make a list of ten guidelines that will lead to software people who work to their full potential.

3.2. The Software Engineering Institute's people management capability maturity model (PM-CMM) takes an organized look at "key practice areas" that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

3.3. Describe three real-life situations in which the customer and the end-user are the same. Describe three situations in which they are different.

3.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

3.5. Review a copy of Weinberg's book [WEI86] and write a two- or three-page summary of the issues that should be considered in applying the MOI model.

3.6. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

3.7. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team struc-

ture would you choose and why? What software process model(s) would you choose and why?

3.8. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next generation version of its widely used word-processing software. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

3.9. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal to to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

3.10. Referring to Figure 3.1, based on the results of the referenced study, documents are perceived to have more use than value. Why do you think this occurred and what can be done to move the documents data point above the regression line in the graph? That is, what can be done to improve the perceived value of documents?

3.11. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

3.12. Do a first level functional decomposition of the page layout function discussed briefly in Section 3.3.2.

FURTHER READINGS AND INFORMATION SOURCES

An excellent four volume series written by Weinberg (*Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts, explains how to use measurements effectively, and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Brooks (*The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. Purba and Shah (*How to Manage a Successful Software Project,* Wiley, 1995) present a number of case studies that indicate why some projects succeed and others fail. Bennatan (*Software Project Management in a Client/Server Environment,* Wiley, 1995) discusses special management issues associated with the development of client/server systems.

It can be argued that the most important aspect of software project management is people management. The definitive book on this subject has been written by

DeMarco and Lister [DEM98], but the following books on this subject have been published in recent years and are worth examining:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work,* Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Humphrey, W.S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W.S., *Introduction to the Team Software Process,* Addison-Wesley, 1999.

Jones, P.H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D.S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Mayer, M., *The Virtual Edge: Embracing Technology for Distributed Project Team Success,* Project Management Institute Publications, 1999.

Another excellent book by Weinberg [WEI86] is must reading for every project manager and every team leader. It will give you insight and guidance in ways to do your job more effectively. House (*The Human Side of Project Management,* Addison-Wesley, 1988) and Crosby *(Running Things: The Art of Making Things Happen,* McGraw-Hill, 1989) provide practical advice for managers who must deal with human as well as technical problems.

Even though they do not relate specifically to the software world and sometimes suffer from over-simplification and broad generalization, best-selling "management" books by Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999) and Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy, Older titles such as *The One-Minute Manager* and *In Search of Excellence* continue to provide valuable insights that can help you to manage people issues more effectively.

A wide variety of information sources on software project issues are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software projects can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/project-mgmt.mhtml

CHAPTER

4 SOFTWARE PROCESS AND PROJECT METRICS

KEY CONCEPTS

Measurement is fundamental to any engineering discipline, and software engineering is no exception. Measurement enables us to gain insight by providing a mechanism for objective evaluation. Lord Kelvin once said:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has finally begun to take Lord Kelvin's words to heart. But not without frustration and more than a little controversy!

Software metrics refers to a broad range of measurements for computer software. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds.

QUICK LOOK

What is it? Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Who does it? Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers.

Why is it important? If you don't measure, judgement can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

What are the steps? Begin by defining a limited set of process, project, and product measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analyzed and compared to past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

QUICK LOOK

What is the work product? A set of software metrics that provide insight into the process and understanding of the project.

How do I ensure that I've done it right? By applying a consistent, yet simple measurement scheme that is never to be used to assess, reward, or punish individual performance.

XRef

Technical metrics for software engineering are presented in Chapters 19 and 24.

Within the context of software project management, we are concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us plan and estimate more accurately?

In their guidebook on software measurement, Park, Goethert, and Florac [PAR96] discuss the reasons that we measure:

There are four reasons for measuring software processes, products, and resources: to characterize, to evaluate, to predict, or to improve.

We characterize to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments.

We evaluate to determine status with respect to plans. Measures are the sensors that let us know when our projects and processes are drifting off track, so that we can bring them back under control. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes.

We predict so that we can plan. Measuring for prediction involves gaining understandings of relationships among processes and products and building models of these relationships, so that the values we observe for some attributes can be used to predict others. We do this because we want to establish achievable goals for cost, schedule, and quality—so that appropriate resources can be applied. Predictive measures are also the basis for extrapolating trends, so estimates for cost, time, and quality can be updated based on current evidence. Projections and estimates based on historical data also help us analyze risks and make design/cost trade-offs.

We measure to improve when we gather quantitative information to help us identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.

Quote:

"Software metrics let you know when to laugh and when to cry."

Tom Gilb

4.1 MEASURES, METRICS, AND INDICATORS

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. Because *measure*

can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terms* [IEE93] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

Quote:

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each). A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person-hour expended on reviews.[1]

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself [RAG95]. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better.

For example, four software teams are working on a large software project. Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another, less formal review approach. She may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

4.2 METRICS IN THE PROCESS AND PROJECT DOMAINS

Measurement is commonplace in the engineering world. We measure power consumption, weight, physical dimensions, temperature, voltage, signal-to-noise ratio . . . the list is almost endless. Unfortunately, measurement is far less common in the software engineering world. We have trouble agreeing on what to measure and trouble evaluating measures that are collected.

---

1 This assumes that another measure, person-hours expended, is collected for each review.

Metrics should be collected so that process and product indicators can be ascertained. *Process indicators* enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement.

WebRef

A comprehensive software metrics guidebook can be downloaded from **www.ivv.nasa.gov/SWG/resources/NASA-GB-001-94.pdf**

Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

4.2.1 Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance [PAU94]."

KEY POINT

The skill and motivation of the people doing the work are the most important factors that influence software quality.

Referring to Figure 4.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods) that populate the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication).

? How do I measure the effectiveness of a software process?

We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent

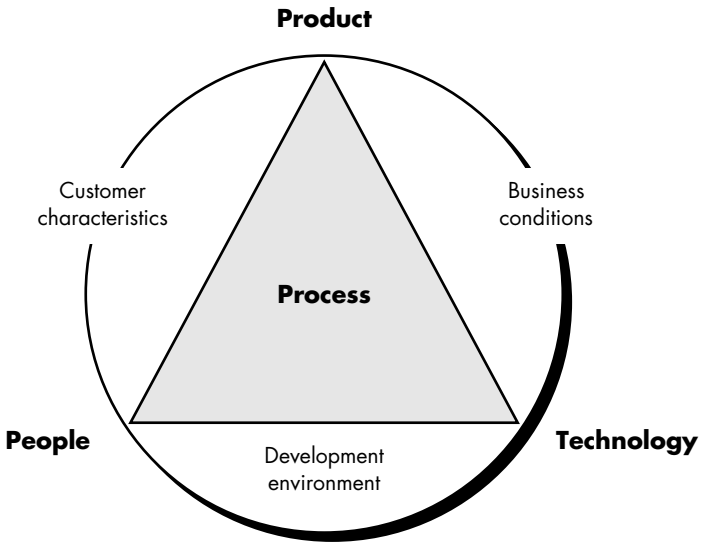


FIGURE 4.1 Determinants for software quality and organizational effectiveness (adapted from [PAU94])

performing the umbrella activities and the generic software engineering activities described in Chapter 2.

Grady [GRA92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of *private metrics* include defect rates (by individual), defect rates (by module), and errors found during development.

The "private process data" philosophy conforms well with the *personal software process* approach proposed by Humphrey [HUM95]. Humphrey describes the approach in the following manner:

> The personal software process (PSP) is a structured set of process descriptions, measurements, and methods that can help engineers to improve their personal performance. It provides the forms, scripts, and standards that help them estimate and plan their work. It shows them how to define processes and how to measure their quality and productivity. A fundamental PSP principle is that everyone is different and that a method that is effective for one engineer may not be suitable for another. The PSP thus helps engineers to measure and track their own work so they can find the methods that are best for them.

Humphrey recognizes that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as the individual software engineer works to improve.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that

have been developed by a number of practitioners), errors found during formal technical reviews, and lines of code or function points per module and function.[2] These data are reviewed by the team to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

KEY POINT

Public metrics enable an organization to make strategic changes that improve the software process and tactical changes during a software project.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [GRA92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

? What guidelines should be applied when we collect software metrics?

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects[3] encountered as an application, system, or product is developed and used. Failure analysis works in the following manner:

WebRef

SSPI and other quality related information is available through the American Society for Quality at **www.asq.org**

1. All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).
2. The cost to correct each error and defect is recorded.

---

2 See Sections 4.3.1 and 4.3.2 for detailed discussions of LOC and function point metrics.

3 As we discuss in Chapter 8, an *error* is some flaw in a software engineering work product or deliverable that is uncovered by software engineers before the software is delivered to the end-user. A *defect* is a flaw that is uncovered after delivery to the end-user.

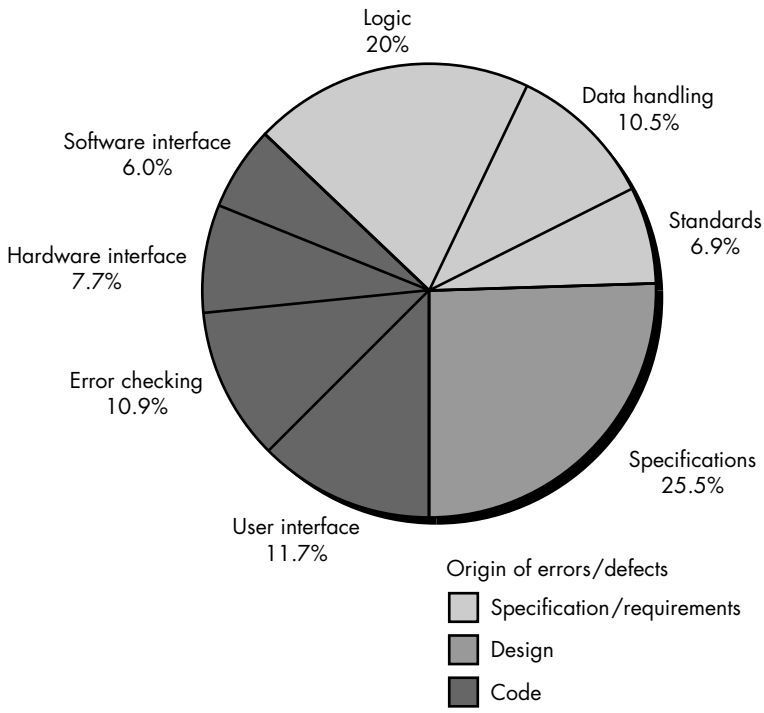


FIGURE 4.2
Causes of defects and their origin for four software projects [GRA94]

You can't improve your approach to software engineering unless you understand where you're strong and where you're weak. Use SSPI techniques to gain that understanding.

3. The number of errors and defects in each category is counted and ranked in descending order.
4. The overall cost of errors and defects in each category is computed.
5. Resultant data are analyzed to uncover the categories that result in highest cost to the organization.
6. Plans are developed to modify the process with the intent of eliminating (or reducing the frequency of) the class of errors and defects that is most costly.

Following steps 1 and 2, a simple defect distribution can be developed (Figure 4.2) [GRA94]. For the pie-chart noted in the figure, eight causes of defects and their origin (indicated by shading) are shown. Grady suggests the development of a *fishbone diagram* [GRA92] to help in diagnosing the data represented in the frequency diagram. Referring to Figure 4.3, the spine of the diagram (the central line) represents the quality factor under consideration (in this case specification defects that account for 25 percent of the total). Each of the ribs (diagonal lines) connecting to the spine indicate potential causes for the quality problem (e.g., missing requirements, ambiguous specification, incorrect requirements, changed requirements). The spine and ribs notation is then added to each of the major ribs of the diagram to expand upon the cause noted. Expansion is shown only for the *incorrect* cause in Figure 4.3.

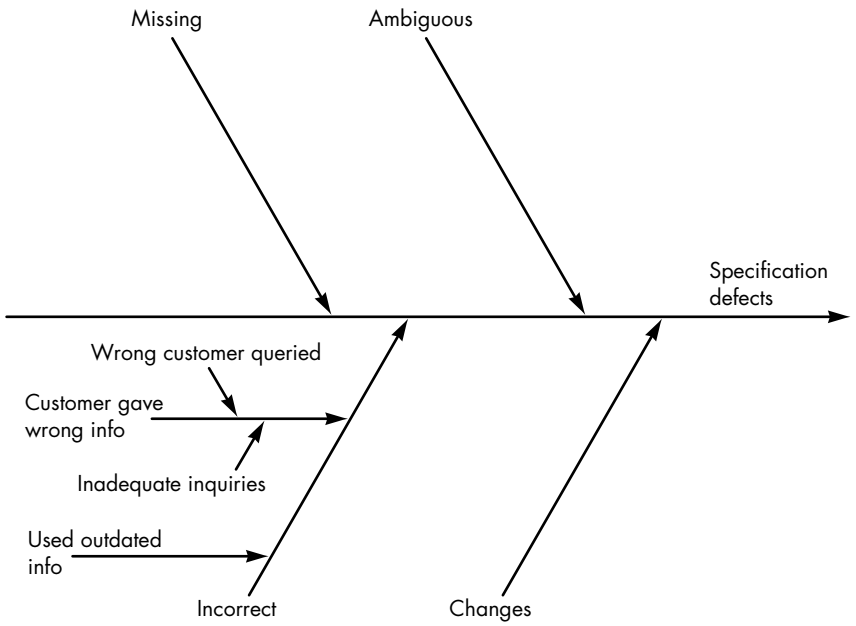


FIGURE 4.3 A fishbone diagram (adapted from [GRA92])

The collection of process metrics is the driver for the creation of the fishbone diagram. A completed fishbone diagram can be analyzed to derive indicators that will enable a software organization to modify its process to reduce the frequency of errors and defects.

4.2.2 Project Metrics

Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

XRef

Project estimation techniques are discussed in Chapter 5.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics (Chapters 19 and 24) are collected to assess

How should we use metrics during the project itself?

design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

Another model of software project metrics [HET93] suggests that every project should measure:

- *Inputs*—measures of the resources (e.g., people, environment) required to do the work.
- *Outputs*—measures of the deliverables or work products created during the software engineering process.
- *Results*—measures that indicate the effectiveness of the deliverables.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity (or task) to the next.

4.3 SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

What is the difference between direct and indirect measures?

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. *Indirect measures* of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" that are discussed in Chapter 19.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have already partitioned the software metrics domain into process, project, and product metrics. We have also noted that product metrics that are private to an

individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

Because many factors influence software work, don't use metrics to compare individuals or teams.

To illustrate, we consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because we do not know the size or complexity of the projects, we cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

4.3.1 Size-Oriented Metrics

What data should we collect to derive size-oriented metrics?

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

FIGURE 4.4 Size-oriented metrics

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

Metrics collection template

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects[4] per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

Size-oriented metrics are widely used, but debate about their validity and applicability continues.

4.3.2 Function-Oriented Metrics

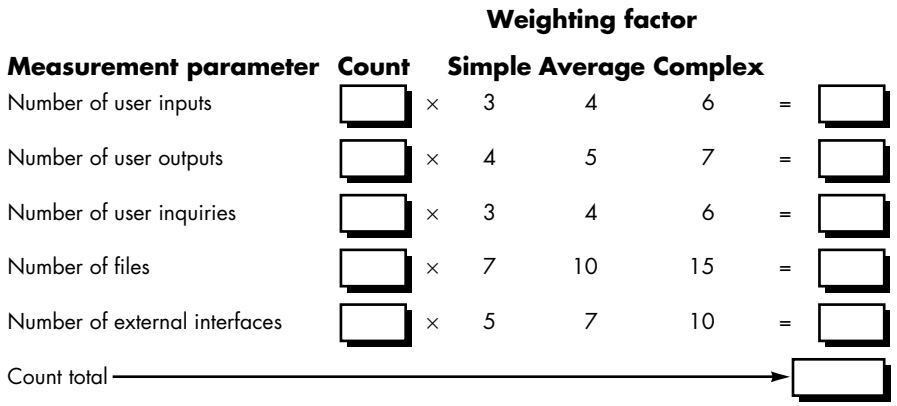
Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht [ALB79], who suggested a measure called the *function point.* Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Comprehensive information on function points can be obtained at **www.ifpug.org**

Function points are computed [IFP94] by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in

4 A defect occurs when quality assurance activities (e.g., formal technical reviews) fail to uncover an error in a work product produced during the software process.

FIGURE 4.5 Computing function points



the appropriate table location. Information domain values are defined in the following manner:[5]

Number of user inputs. Each user input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

KEY POINT

Function points are derived from direct measures of the information domain.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] \qquad (4\text{-}1)$$

where count total is the sum of all FP entries obtained from Figure 4.5.

5 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP94] for details.

The F_i (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions [ART85]:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

4.3.3 Extended Function Point Metrics

KEY POINT

Extending function points are used for engineering, real-time, and control-oriented applications.

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the exclusion of the functional and behavioral (control) dimensions. For this reason, the function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic function point measure have been proposed to remedy this situation.

A function point extension called *feature points* [JON91], is a superset of the function point measure that can be applied to systems and engineering software applications.

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point.

To compute the feature point, information domain values are again counted and weighted as described in Section 4.3.2. In addition, the feature point metric counts a new software characteristic—*algorithms*. An *algorithm* is defined as "a bounded computational problem that is included within a specific computer program" [JON91]. Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms.

A useful FAQ on function points (and extended function points) can be obtained at **http://ourworld.compuserve.com/homepages/softcomp/**

Another function point extension for real-time systems and engineered products has been developed by Boeing. The Boeing approach integrates the data dimension of software with the functional and control dimensions to provide a function-oriented measure amenable to applications that emphasize function and control capabilities. Called the *3D function point* [WHI95], characteristics of all three software dimensions are "counted, quantified, and transformed" into a measure that provides an indication of the functionality delivered by the software.[6]

The *data dimension* is evaluated in much the same way as described in Section 4.3.2. Counts of retained data (the internal program data structure; e.g., files) and external data (inputs, outputs, inquiries, and external references) are used along with measures of complexity to derive a data dimension count. The *functional dimension* is measured by considering "the number of internal operations required to transform input to output data" [WHI95]. For the purposes of 3D function point computation, a "transformation" is viewed as a series of processing steps that are constrained by a set of semantic statements. The *control dimension* is measured by counting the number of transitions between states.[7]

A state represents some externally observable mode of behavior, and a transition occurs as a result of some event that causes the software or system to change its mode of behavior (i.e., to change state). For example, a wireless phone contains software that supports auto dial functions. To enter the *auto-dial* state from a *resting state,* the user presses an **Auto** key on the keypad. This event causes an LCD display to prompt for a code that will indicate the party to be called. Upon entry of the code and hitting the **Dial** key (another event), the wireless phone software makes a transition to the *dialing* state. When computing 3D function points, transitions are not assigned a complexity value.

To compute 3D function points, the following relationship is used:

$$\text{index} = I + O + Q + F + E + T + R \qquad (4\text{-}2)$$

---

6 It should be noted that other extensions to function points for application in real-time software work (e.g., [ALA97]) have also been proposed. However, none of these appears to be widely used in the industry.

7 A detailed discussion of the behavioral dimension, including states and state transitions, is presented in Chapter 12.

FIGURE 4.6
Determining the complexity of a transformation for 3D function points [WHI95].

Semantic statements / Processing steps	1–5	6–10	11+
1–10	Low	Low	Average
11–20	Low	Average	High
21+	Average	High	High

where *I, O, Q, F, E, T*, and *R* represent complexity weighted values for the elements discussed already: inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions, respectively. Each complexity weighted value is computed using the following relationship:

$$\text{complexity weighted value} = N_{il}W_{il} + N_{ia}W_{ia} + N_{ih}W_{ih} \quad (4\text{-}3)$$

where N_{il}, N_{ia}, and N_{ih} represent the number of occurrences of element i (e.g., outputs) for each level of complexity (low, medium, high); and W_{il}, W_{ia}, and W_{ih} are the corresponding weights. The overall complexity of a transformation for 3D function points is shown in Figure 4.6.

It should be noted that function points, feature points, and 3D function points represent the same thing—"functionality" or "utility" delivered by software. In fact, each of these measures results in the same value if only the data dimension of an application is considered. For more complex real-time systems, the feature point count is often between 20 and 35 percent higher than the count determined using function points alone.

The function point (and its extensions), like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages; that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data; that counts of the information domain (and other dimensions) can be difficult to collect after the fact; and that FP has no direct physical meaning—it's just a number.

4.4 RECONCILING DIFFERENT METRICS APPROACHES

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. To quote Albrecht and Gaffney [ALB83]:

> The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components[8] of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed.

The following table [JON98] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

If I know the number of LOC, is it possible to estimate the number of function points?

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (code generator)	16
SQL	12

Use backfiring data judiciously. It is far better to compute FP using the methods discussed earlier.

A review of these data indicates that one LOC of C++ provides approximately 1.6 times the "functionality" (on average) as one LOC of FORTRAN. Furthermore, one LOC of a Visual Basic provides more than three times the functionality of a LOC for a conventional programming language. More detailed data on the relationship between FP and LOC are presented in [JON98] and can be used to "backfire" (i.e., to compute the number of function points when the number of delivered LOC are known) existing programs to determine the FP measure for each.

LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answers

8 It is important to note that "the itemization of major components" can be interpreted in a variety of ways. Some software engineers who work in an object-oriented development environment (Part Four) use the number of classes or objects as the dominant size metric. A maintenance organization might view project size in terms of the number of engineering change orders (Chapter 9). An information systems organization might view the number of business processes affected by an application.

to these questions is an emphatic "No!" The reason for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 5), a historical baseline of information must be established.

4.5 METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

An excellent source of information on software quality and related topics (including metrics) can be found at **www.qualityworld.com**

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer must use technical measures (Chapters 19 and 24) to evaluate quality in objective, rather than subjective ways.

XRef

A detailed discussion of software quality assurance activities is presented in Chapter 8.

The project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are assimilated to provide project-level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed in Section 4.5.3.

4.5.1 An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano [MCC78] defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assess software from three distinct points of view: (1) product operation (using it), (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment; i.e., "porting" it). In their work, the authors describe the

relationship between these quality factors (what they call a *framework*) and other aspects of the software engineering process:

> First, the framework provides a mechanism for the project manager to identify what qualities are important. These qualities are attributes of the software in addition to its functional correctness and performance which have life cycle implications. Such factors as maintainability and portability have been shown in recent years to have significant life cycle cost impact . . .
>
> Secondly, the framework provides a means for quantitatively assessing how well the development is progressing relative to the quality goals established . . .
>
> Thirdly, the framework provides for more interaction of QA personnel throughout the development effort . . .
>
> Lastly, . . . quality assurance personal can use indications of poor quality to help identify [better] standards to be enforced in the future.

Surprisingly, the factors that defined software quality in the 1970s are the same factors that continue to define software quality in the first decade of this century.

A detailed discussion of McCall and Cavano's framework, as well as other quality factors, is presented in Chapter 19. It is interesting to note that nearly every aspect of computing has undergone radical change as the years have passed since McCall and Cavano did their seminal work in 1978. But the attributes that provide an indication of software quality remain the same.

What does this mean? If a software organization adopts a set of quality factors as a "checklist" for assessing software quality, it is likely that software built today will still exhibit quality well into the first few decades of this century. Even as computing architectures undergo radical change (as they surely will), software that exhibits high quality in operation, transition, and revision will continue to serve its users well.

4.5.2 Measuring Quality

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and measures for each.

Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

Maintainability. Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in require-

ments. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

Hitachi [TAJ81] has used a cost-oriented metric for maintainability called *spoilage*—the cost to correct defects encountered after the software has been released to its end-users. When the ratio of spoilage to overall project cost (for many projects) is plotted as a function of time, a manager can determine whether the overall maintainability of software produced by a software development organization is improving. Actions can then be taken in response to the insight gained from this information.

Integrity. Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents.

To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

$$\text{integrity} = \text{summation } [(1 - \text{threat}) \times (1 - \text{security})]$$

where threat and security are summed over each type of attack.

Usability. The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system. Detailed discussion of this topic is contained in Chapter 15.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 19 considers this topic in additional detail.

4.5.3 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

What is defect removal efficiency?

$$\text{DRE} = E/(E + D) \tag{4-4}$$

where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

ADVICE

Use DRE as a measure of the efficacy of your early SQA activities. If DRE is low during analysis and design, spend some time improving the way you conduct formal technical reviews.

$$\text{DRE}_i = E_i/(E_i + E_{i+1}) \tag{4-5}$$

where E_i is the number of errors found during software engineering activity i and E_{i+1} is the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i.

A quality objective for a software team (or an individual software engineer) is to achieve DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

4.6 INTEGRATING METRICS WITHIN THE SOFTWARE PROCESS

The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted earlier in this chapter, the problem is cultural. Attempting to collect measures where none had been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner.

In this section, we consider some arguments for software metrics and present an approach for instituting a metrics collection program within a software engineering

organization. But before we begin, some words of wisdom are suggested by Grady and Caswell [GRA87]:

> Some of the things we describe here will sound quite easy. Realistically, though, establishing a successful company-wide software metrics program is hard work. When we say that you must wait at least three years before broad organizational trends are available, you get some idea of the scope of such an effort.

The caveat suggested by the authors is well worth heeding, but the benefits of measurement are so compelling that the hard work is worth it.

4.6.1 Arguments for Software Metrics

Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If we do not measure, there no real way of determining whether we are improving. And if we are not improving, we are lost.

By requesting and evaluating productivity and quality measures, senior management can establish meaningful goals for improvement of the software engineering process. In Chapter 1 we noted that software is a strategic business issue for many companies. If the process through which it is developed can be improved, a direct impact on the bottom line can result. But to establish goals for improvement, the current status of software development must be understood. Hence, measurement is used to establish a process baseline from which improvements can be assessed.

Quote:

"We manage things 'by the numbers' in many aspects of our lives. . . . These numbers give us insight and help steer our actions."

Michael Mah
Larry Putnam

The day-to-day rigors of software project work leave little time for strategic thinking. Software project managers are concerned with more mundane (but equally important) issues: developing meaningful project estimates, producing higher-quality systems, getting product out the door on time. By using measurement to establish a project baseline, each of these issues becomes more manageable. We have already noted that the baseline serves as a basis for estimation. Additionally, the collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.[9]

At the project and technical levels (in the trenches), software metrics provide immediate benefit. As the software design is completed, most developers would be anxious to obtain answers to the questions such as

- Which user requirements are most likely to change?
- Which components in this system are most error prone?
- How much testing should be planned for each component?
- How many errors (of specific types) can I expect when testing commences?

---

9 These ideas have been formalized into an approach called *statistical software quality assurance* and are discussed in detail in Chapter 8.

Answers to these questions can be determined if metrics have been collected and used as a technical guide. In later chapters, we examine how this is done.

4.6.2 Establishing a Baseline

By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 4.4 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

What critical information can metrics provide for a developer?

To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes: (1) data must be reasonably accurate—"guestimates" about past projects are to be avoided; (2) data should be collected for as many projects as possible; (3) measures must be consistent, for example, a line of code must be interpreted consistently across all projects for which data are collected; (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

4.6.3 Metrics Collection, Computation, and Evaluation

KEY POINT

Baseline metrics data should be collected from a large, representative sampling of past software projects.

The process for establishing a baseline is illustrated in Figure 4.7. Ideally, data needed to establish a baseline has been collected in an ongoing manner. Sadly, this is rarely the case. Therefore, data collection requires a historical investigation of past projects to reconstruct required data. Once measures have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of measures collected, metrics can span a broad range of LOC or FP metrics as well as other quality- and project-oriented metrics. Finally, metrics must be evaluated and applied during estimation, technical work, project control, and process improvement. Metrics evaluation focuses on the underlying reasons for the results obtained and produces a set of indicators that guide the project or process.

4.7 MANAGING VARIATION: STATISTICAL PROCESS CONTROL

Because the software process and the product it produces both are influenced by many parameters (e.g., the skill level of practitioners, the structure of the software team, the knowledge of the customer, the technology that is to be implemented, the tools to be used in the development activity), metrics collected for one project or product will not be the same as similar metrics collected for another project. In fact, there is often significant variability in the metrics we collect as part of the software process.

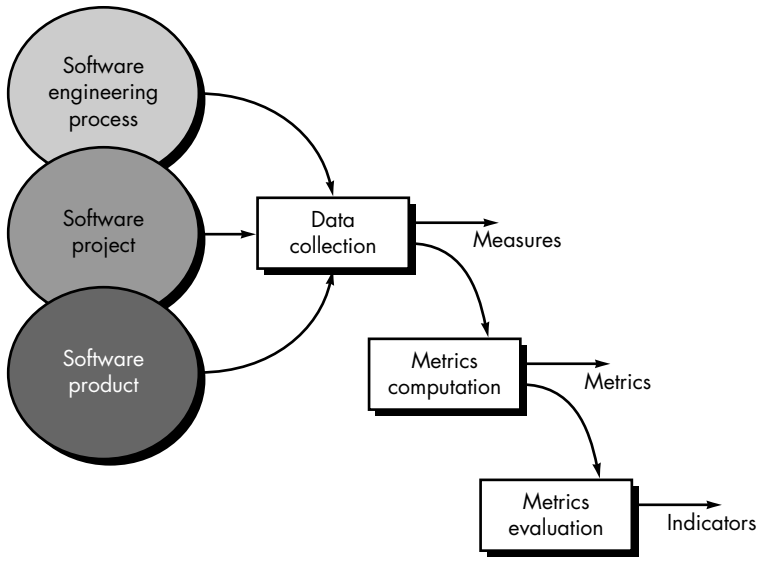


FIGURE 4.7 Software metrics collection process

Quote:

"If I had to reduce my message for management to just a few words, I'd say it all had to do with reducing variation."

W. Edwards Deming

Since the same process metrics will vary from project to project, how can we tell if improved (or degraded) metrics values that occur as consequence of improvement activities are having a quantitative impact? How do we know whether we're looking at a statistically valid trend or whether the "trend" is simply a result of statistical noise? When are changes (either positive or negative) to a particular software metric meaningful?

A graphical technique is available for determining whether changes and variation in metrics data are meaningful. Called the *control chart* and developed by Walter Shewart in the 1920s,[10] this technique enables individuals interested in software process improvement to determine whether the dispersion (variability) and "location" (moving average) of process metrics are *stable* (i.e., the process exhibits only natural or controlled changes) or *unstable* (i.e., the process exhibits out-of-control changes and metrics cannot be used to predict performance). Two different types of control charts are used in the assessment of metrics data [ZUL99]: (1) the moving range control chart and (2) the individual control chart.

? How can we be sure that the metrics we collect are statistically valid?

To illustrate the control chart approach, consider a software organization that collects the process metric, errors uncovered per review hour, E_r. Over the past 15 months, the organization has collected E_r for 20 small projects in the same general software development domain. The resultant values for E_r are represented in Figure 4.8. In the figure, E_r varies from a low of 1.2 for project 3 to a high of 5.9 for project 17. In an effort to improve the effectiveness of reviews, the software organization provided training and mentoring to all project team members beginning with project 11.

10 It should be noted that, although the control chart was originally developed for manufacturing processes, it is equally applicable for software processes.

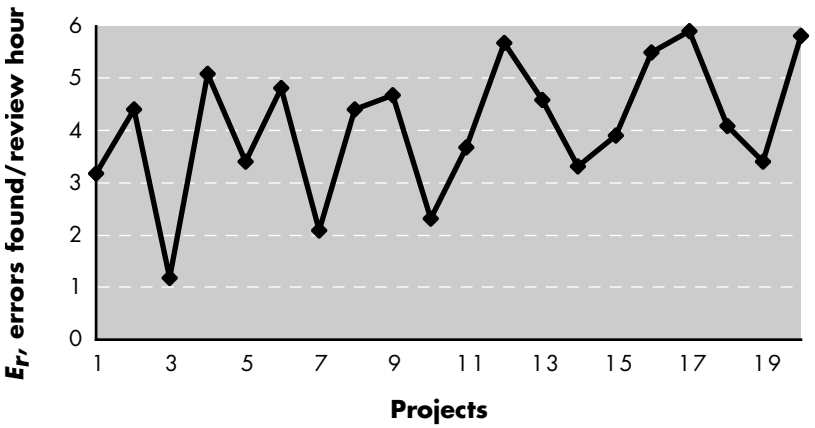


FIGURE 4.8 Metrics data for errors uncovered per review hour

Richard Zultner provides an overview of the procedure required to develop a *moving range* (mR) *control chart* for determining the stability of the process [ZUL99]:

1. Calculate the moving ranges: the absolute value of the successive differences between each pair of data points . . . Plot these moving ranges on your chart.
2. Calculate the mean of the moving ranges . . . plot this ("mR bar") as the center line on your chart.
3. Multiply the mean by 3.268. Plot this line as the *upper control limit* [UCL]. This line is three standard deviations above the mean.

Quote:

"If we can't tell signals from noise, how will we ever know if changes to the process are improvement—or illusions?"

Richard Zultner

Using the data represented in Figure 4.8 and the steps suggested by Zultner, we develop an mR control chart shown in Figure 4.9. The mR bar (mean) value for the moving range data is 1.71. The upper control limit is 5.58.

To determine whether the process metrics dispersion is stable, a simple question is asked: Are all the moving range values inside the UCL? For the example noted, the answer is "yes." Hence, the metrics dispersion is stable.

The individual control chart is developed in the following manner:[11]

1. Plot individual metrics values as shown in Figure 4.8.
2. Compute the average value, A_m, for the metrics values.
3. Multiply the mean of the mR values (the mR bar) by 2.660 and add A_m computed in step 2. This results in the *upper natural process limit* (UNPL). Plot the UNPL.
4. Multiply the mean of the mR values (the mR bar) by 2.660 and subtract this amount from A_m computed in step 2. This results in the *lower natural process limit* (LNPL). Plot the LNPL. If the LNPL is less than 0.0, it need not be plotted unless the metric being evaluated takes on values that are less than 0.0.
5. Compute a standard deviation as (UNPL $- A_m$)/3. Plot lines one and two standard deviations above and below A_m. If any of the standard deviation

11 The discussion that follows is a summary of steps suggested by Zultner [ZUL99].

FIGURE 4.9
Moving range control chart

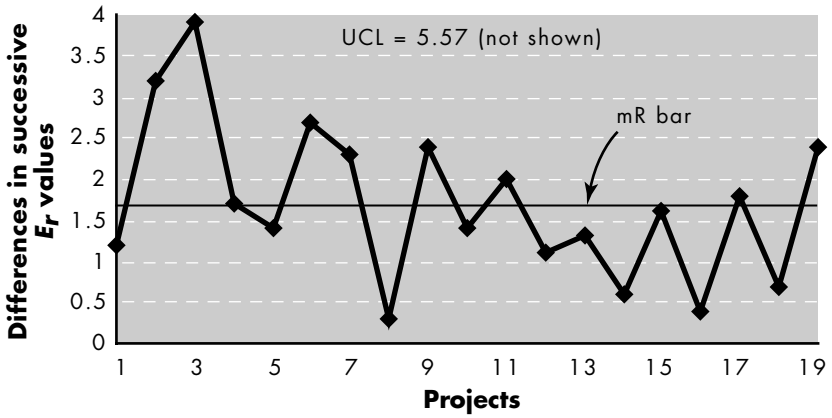
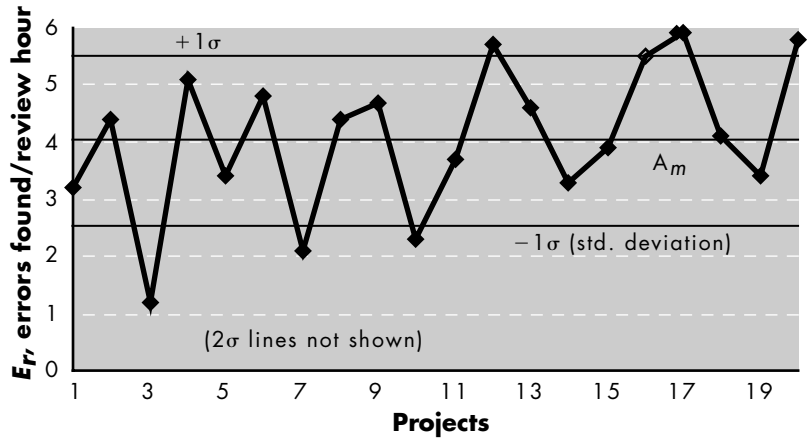


FIGURE 4.10
Individual control chart



The *Common Control Chart Cookbook* covers the topic at some length and can be found at **www.sytsma.com/tqmtools/ctlchtprinciples.html**

lines is less than 0.0, it need not be plotted unless the metric being evaluated takes on values that are less than 0.0.

Applying these steps to the data represented in Figure 4.8, we derive an *individual control chart* as shown in Figure 4.10.

Zultner [ZUL99] reviews four criteria, called *zone rules,* that may be used to evaluate whether the changes represented by the metrics indicate a process that is in control or out of control. If any of the following conditions is true, the metrics data indicate a process that is out of control:

1. A single metrics value lies outside the UNPL.
2. Two out of three successive metrics values lie more than two standard deviations away from A_m.
3. Four out of five successive metrics values lie more than one standard deviation away from A_m.
4. Eight consecutive metrics values lie on one side of A_m.

Since all of these conditions fail for the values shown in Figure 4.10, the metrics data are derived from a stable process and trend information can be legitimately inferred from the metrics collected. Referring to Figure 4.10, it can be seen that the variability of E_r decreases after project 10 (i.e., after an effort to improve the effectiveness of reviews). By computing the mean value for the first 10 and last 10 projects, it can be shown that the mean value of E_r for projects 11–20 shows a 29 percent improvement over E_r for projects 1–10. Since the control chart indicates that the process is stable, it appears that efforts to improve review effectiveness are working.

4.8 METRICS FOR SMALL ORGANIZATIONS

If you're just starting to collect software metrics, remember to keep it simple. If you bury yourself with data, your metrics effort will fail.

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it is reasonable to suggest that software organizations of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce. Kautz [KAU99] describes a typical scenario that occurs when metrics programs are suggested for small software organizations:

> Originally, the software developers greeted our activities with a great deal of skepticism, but they eventually accepted them because we kept our measurements simple, tailored them to each organization, and ensured that they produced valuable information. In the end, the programs provided a foundation for taking care of customers and for planning and carrying out future work.

What Kautz suggests is a commonsense approach to the implementation of any software process related activity: keep it simple, customize to meet local needs, and be sure it adds value. In the paragraphs that follow, we examine how these guidelines relate to metrics for small shops.

How do I derive a set of "simple" software metrics?

"Keep it simple" is a guideline that works reasonably well in many activities. But how do we derive a "simple" set of software metrics that still provides value, and how can we be sure that these simple metrics will meet the needs of a particular software organization? We begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires improvement. For example, "reduce the time to evaluate and implement change requests." A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, t_{queue}.
- Effort (person-hours) to perform the evaluation, W_{eval}.
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, t_{eval}.

- Effort (person-hours) required to make the change, W_{change}.
- Time required (hours or days) to make the change, t_{change}.
- Errors uncovered during work to make change, E_{change}.
- Defects uncovered after change is released to the customer base, D_{change}.

Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data, E_{change} and D_{change}. The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce t_{queue}, W_{eval}, t_{eval}, W_{change}, and/or E_{change}. In addition, the defect removal efficiency can be computed as

$$\text{DRE} = E_{change} / (E_{change} + D_{change})$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

For small groups, the cost of collecting measures and computing metrics ranges from 3 to 8 percent of project budget during the learning phase and then drops to less than 1 percent of project budget after software engineers and project managers have become familiar with the metrics program [GRA99]. These costs can show a substantial return on investment if the insights derived from metrics data lead to meaningful process improvement for the software organization.

4.9 ESTABLISHING A SOFTWARE METRICS PROGRAM

The Software Engineering Institute has developed a comprehensive guidebook [PAR96] for establishing a "goal-driven" software metrics program. The guidebook suggests the following steps:

A Guidebook for Goal-Driven Software Measurement can be downloaded from **www.sei.cmu.edu**

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used, and make these definitions operational.

9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

A detailed discussion of these steps is best left to the SEI's guidebook. However, a brief overview of key points is worthwhile.

The software metrics you choose are driven by the business or technical goals you wish to accomplish.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider a company that makes advanced home security systems which have substantial software content. Working as a team, software engineering and business managers can develop a list of prioritized business goals:

1. Improve our customers' satisfaction with our products.
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

The software organization examines each business goal and asks: "What activities do we manage or execute and what do we want to improve within these activities?" To answer these questions the SEI recommends the creation of an "entity-question list" in which all things (entities) within the software process that are managed or influenced by the software organization are noted. Examples of entities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process.

Consider the fourth goal: "Make support for our products easier." The following list of questions might be derived for this goal [PAR96]:

- Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?
- How large is the change request backlog?
- Is our response time for fixing bugs acceptable based on customer need?
- Is our change control process (Chapter 9) followed?
- Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following subgoal: Improve the performance of the change management process. The software

process entities and attributes that are relevant to the subgoal are identified and measurement goals associated with them are delineated.

The SEI [PAR96] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, a process of stepwise refinement is applied in which goals are refined into questions that are further refined into entities and attributes that are then refined into metrics.

4.10 SUMMARY

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of a software project; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project work flow and technical approach in a real-time manner.

Both size- and function-oriented metrics are used throughout the industry. Size-oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. The function point is derived from measures of the information domain and a subjective assessment of problem complexity.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

Metrics are meaningful only if they have been examined for statistical validity. The control chart is a simple method for accomplishing this and at the same time examining the variation and location of metrics results.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business. By creating a metrics baseline—a database containing process and product measurements—software engineers and their managers can gain better insight into the work that they do and the product that they produce.

REFERENCES

[ALA97] Alain, A., M. Maya, J.M. Desharnais, and S. St. Pierre, "Adapting Function Points to Real-Time Software," *American Programmer,* vol. 10, no. 11, November 1997, pp. 32–43.

[ALB79] Albrecht, A.J., "Measuring Application Development Productivity," *Proc. IBM Application Development Symposium,* Monterey, CA, October 1979, pp. 83–92.

[ALB83] Albrecht, A.J. and J.E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Engineering,* November 1983, pp. 639–648.

[ART85] Arthur, L.J., *Measuring Programmer Productivity and Software Quality,* Wiley-Interscience, 1985.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[GRA87] Grady, R.B. and D.L. Caswell, *Software Metrics: Establishing a Company-wide Program,* Prentice-Hall, 1987.

[GRA92] Grady, R.G., *Practical Software Metrics for Project Management and Process Improvement,* Prentice-Hall, 1992.

[GRA94] Grady, R., "Successfully Applying Software Metrics," *Computer,* vol. 27, no. 9, September 1994, pp. 18–25.

[GRA99] Grable, R., et al., "Metrics for Small Projects: Experiences at SED," *IEEE Software,* March 1999, pp. 21–29.

[GIL88] Gilb, T., *Principles of Software Project Management,* Addison-Wesley, 1988.

[HET93] Hetzel, W., *Making Software Measurement Work,* QED Publishing Group, 1993.

[HUM95] Humphrey, W., *A Discipline for Software Engineering,* Addison-Wesley, 1995.

[IEE93] *IEEE Software Engineering Standards,* Standard 610.12-1990, pp. 47–48.

[IFP94] *Function Point Counting Practices Manual,* Release 4.0, International Function Point Users Group, 1994.

[JON86] Jones, C., *Programming Productivity,* McGraw-Hill, 1986.

[JON91] Jones, C., *Applied Software Measurement,* McGraw-Hill, 1991.

[JON98] Jones, C., *Estimating Software Costs,* McGraw-Hill, 1998.

[KAU99] Kautz, K., "Making Sense of Measurement for Small Organizations," *IEEE Software,* March 1999, pp. 14–20.

[MCC78] McCall, J.A. and J.P. Cavano, "A Framework for the Measurement of Software Quality," ACM Software Quality Assurance Workshop, November 1978.

[PAR96] Park, R.E., W.B. Goethert, and W.A. Florac, *Goal Driven Software Measurement—A Guidebook,* CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.

[PAU94] Paulish, D. and A. Carleton, "Case Studies of Software Process Improvement Measurement," *Computer,* vol. 27, no. 9, September 1994, pp. 50–57.

[RAG95] Ragland, B., "Measure, Metric or Indicator: What's the Difference?" *Crosstalk,* vol. 8, no. 3, March 1995, p. 29–30.

[TAJ81] Tajima, D. and T. Matsubara, "The Computer Software Industry in Japan," *Computer,* May 1981, p. 96.

[WHI95] Whitmire, S.A., "An Introduction to 3D Function Points", *Software Development,* April 1995, pp. 43–53.

[ZUL99] Zultner, R.E., "What Do Our Metrics Mean?" *Cutter IT Journal,* vol. 12, no. 4, April 1999, pp. 11–19.

PROBLEMS AND POINTS TO PONDER

4.1. Suggest three measures, three metrics, and corresponding indicators that might be used to assess an automobile.

4.2. Suggest three measures, three metrics, and corresponding indicators that might be used to assess the service department of an automobile dealership.

4.3. Describe the difference between process and project metrics in your own words.

4.4. Why should some software metrics be kept "private"? Provide examples of three metrics that should be private. Provide examples of three metrics that should be public.

4.5. Obtain a copy of Humphrey (*Introduction to the Personal Software Process,* Addison-Wesley, 1997) and write a one- or two-page summary that outlines the PSP approach.

4.6. Grady suggests an etiquette for software metrics. Can you add three more rules to those noted in Section 4.2.1?

4.7. Attempt to complete the fishbone diagram shown in Figure 4.3. That is, following the approach used for "incorrect" specifications, provide analogous information for "missing, ambiguous, and changed" specifications.

4.8. What is an indirect measure and why are such measures common in software metrics work?

4.9. Team A found 342 errors during the software engineering process prior to release. Team B found 184 errors. What additional measures would have to be made for projects A and B to determine which of the teams eliminated errors more efficiently? What metrics would you propose to help in making the determination? What historical data might be useful?

4.10. Present an argument against lines of code as a measure for software productivity. Will your case hold up when dozens or hundreds of projects are considered?

4.11. Compute the function point value for a project with the following information domain characteristics:

Number of user inputs: 32
Number of user outputs: 60
Number of user inquiries: 24
Number of files: 8
Number of external interfaces: 2

Assume that all complexity adjustment values are average.

4.12. Compute the 3D function point value for an embedded system with the following characteristics:

Internal data structures: 6
External data structure: 3

Number of user inputs: 12
Number of user outputs: 60
Number of user inquiries: 9
Number of external interfaces: 3
Transformations: 36
Transitions: 24

Assume that the complexity of these counts is evenly divided between low, average, and high.

4.13. The software used to control a photocopier requires 32,000 of C and 4,200 lines of Smalltalk. Estimate the number of function points for the software inside the photocopier.

4.14. McCall and Cavano (Section 4.5.1) define a "framework" for software quality. Using information contained in this and other books, expand each of the three major "points of view" into a set of quality factors and metrics.

4.15. Develop your own metrics (do not use those presented in this chapter) for correctness, maintainability, integrity, and usability. Be sure that they can be translated into quantitative values.

4.16. Is it possible for spoilage to increase while at the same time defects/KLOC decrease? Explain.

4.17. Does the LOC measure make any sense when fourth generation techniques are used? Explain.

4.18. A software organization has DRE data for 15 projects over the past two years. The values collected are 0.81, 0.71, 0.87, 0.54, 0.63, 0.71, 0.90, 0.82, 0.61, 0.84, 0.73, 0.88, 0.74, 0.86, 0.83. Create mR and individual control charts to determine whether these data can be used to assess trends.

FURTHER READINGS AND INFORMATION SOURCES

Software process improvement (SPI) has received a significant amount of attention over the past decade. Since measurement and software metrics are key to successfully improving the software process, many books on SPI also discuss metrics. Worthwhile additions to the literature include:

Burr, A. and M. Owen, *Statistical Methods for Software Quality,* International Thomson Publishing, 1996.

El Emam, K. and N. Madhavji (eds.), *Elements of Software Process Assessment and Improvement,* IEEE Computer Society, 1999.

Florac, W.A. and A.D. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement,* Addison-Wesley, 1999.

Garmus, D. and D. Herron, *Measuring the Software Process: A Practical Guide to Functional Measurements,* Prentice-Hall, 1996.

Humphrey, W., *Introduction to the Team Software Process,* Addison-Wesley Longman, 2000.

Kan, S.H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995.

Humphrey [HUM95], Yeh (*Software Process Control,* McGraw-Hill, 1993), Hetzel [HET93], and Grady [GRA92] discuss how software metrics can be used to provide the indicators necessary to improve the software process. Putnam and Myers (*Executive Briefing: Controlling Software Development,* IEEE Computer Society, 1996) and Pulford and his colleagues (*A Quantitative Approach to Software Management,* Addison-Wesley, 1996) discuss process metrics and their use from a management point of view.

Weinberg (*Quality Software Management,* Volume 2: *First Order Measurement,* Dorset House, 1993) presents a useful model for observing software projects, ascertaining the meaning of the observation, and determining its significance for tactical and strategic decisions. Garmus and Herron (*Measuring the Software Process,* Prentice-Hall, 1996) discuss process metrics with an emphasis on function point analysis. The Software Productivity Consortium (*The Software Measurement Guidebook,* Thomson Computer Press, 1995) provides useful suggestions for instituting an effective metrics approach. Oman and Pfleeger (*Applying Software Metrics,* IEEE Computer Society Press, 1997) have edited an excellent anthology of important papers on software metrics. Park, et al. [PAR96] have developed a detailed guidebook that provides step-by-step suggestions for instituting a software metrics program for software process improvement.

The newsletter *IT Metrics* (edited by Howard Rubin and published by Cutter Information Services) presents useful commentary on the state of software metrics in the industry. The magazines *Cutter IT Journal* and *Software Development* have regular articles and entire features dedicated to software metrics.

A wide variety of information sources on software process and project metrics are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process and project metrics can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/process-metrics.mhtml