Unit I

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.

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.

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

Computer Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

Computer Software can be built 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. Software Engineering approach is applied to build a computer software.

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).

- **The Evolving Role of Software**

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). 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.

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.

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

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.

- **Software**

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

- **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:

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

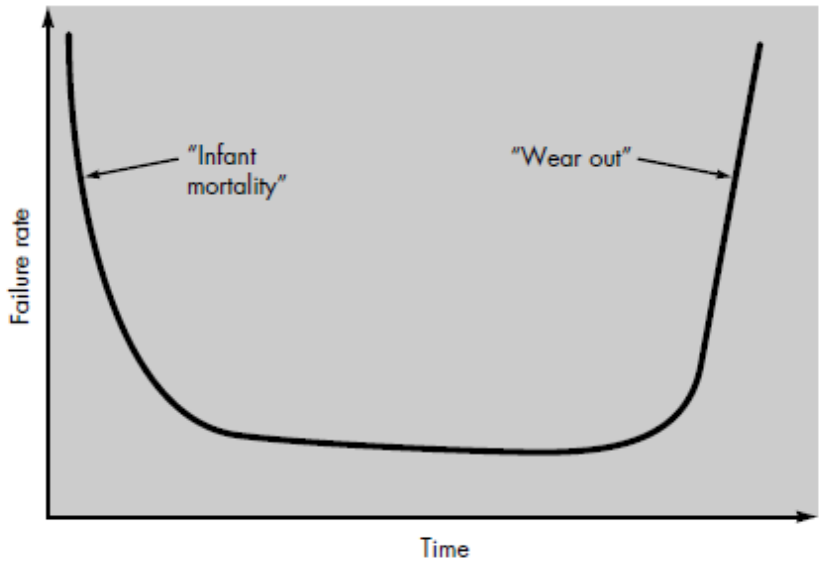
Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are non-existent (or easily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

2. Software doesn't "wear out."

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life

(these failures are often attributable to design or manufacturing defects); defects are 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.

FIGURE 1.1
Failure curve for hardware



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 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 changes are made, it is likely that some new defects will be introduced, causing then 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.

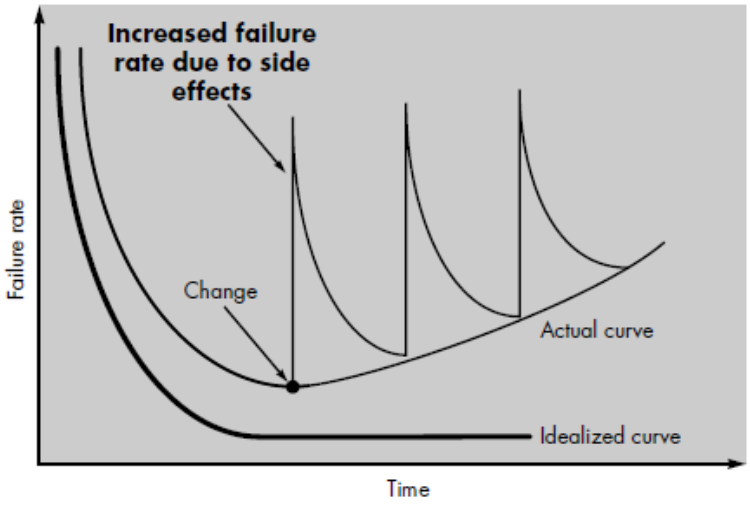


FIGURE 1.2 Idealized and actual failure curves for software

KEY POINT Software engineering methods strive to

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Although the industry is moving toward component-based 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.

- **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:

1. 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.

2. 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
3. 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., pointed-sale transaction processing).

4. 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.

5. 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).

6. 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.

7. 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.

SOFTWARE: A CRISIS ON THE HORIZON?

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.

SOFTWARE MYTHS

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.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

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

- **Customer myths.** A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation.

  Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

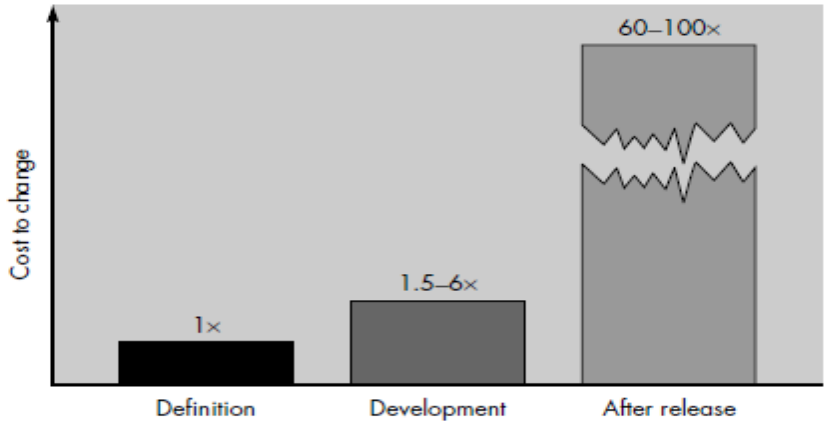
  Myth: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

  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.

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.

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 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.

SOFTWARE ENGINEERING: A LAYERED TECHNOLOGY

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.

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).

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 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.

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 modelling 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.

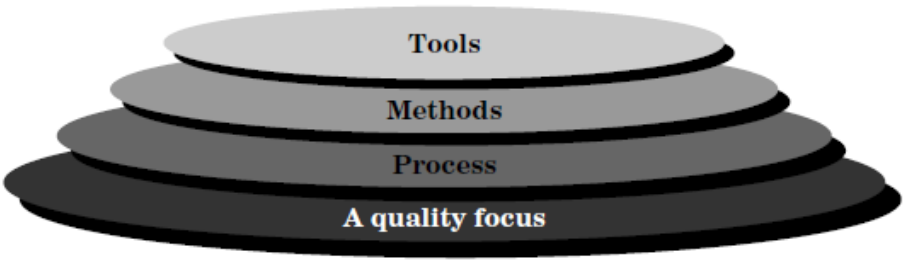


FIGURE 2.1 Software engineering layers

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?
- 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.

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, software project planning, and requirements analysis.

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, code generation, and software testing.

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.

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 programs1 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 are applied throughout the software process

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, 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 measurement2—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

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:

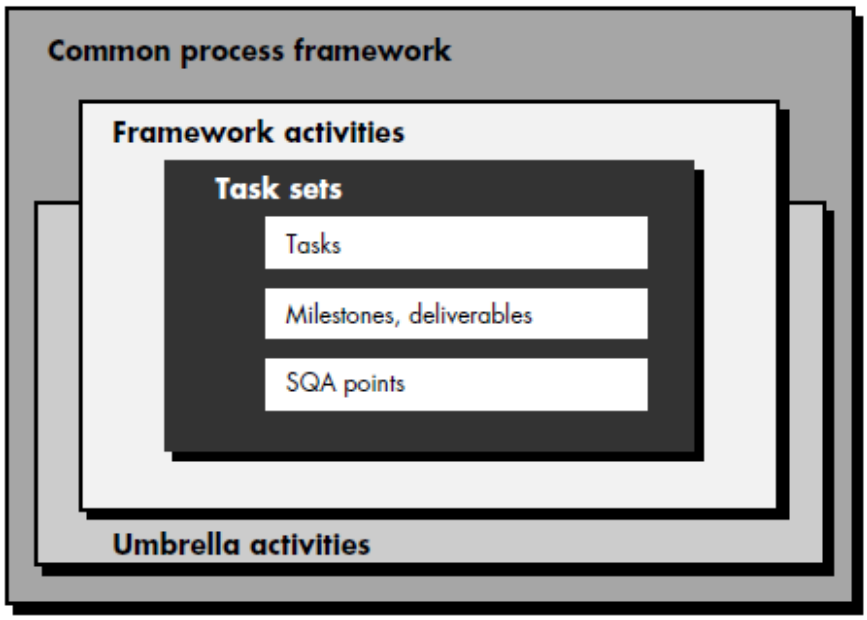


FIGURE 2.2 The software process

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.
Level 3: Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide 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.

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
• 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

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."

SOFTWARE PROCESS MODELS

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 and the generic phases

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.

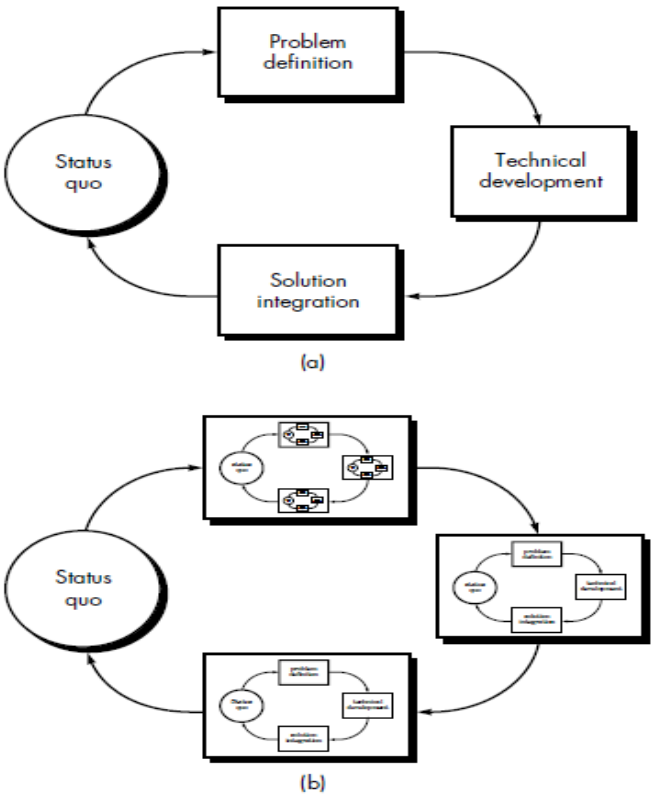
All software development can be characterized as a problem-solving loop (Figure 2.3a) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration. Status quo “represents the current state of affairs” [RAC95]; problem definition identifies 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 fractal4 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).

FIGURE 2.3

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

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



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. 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.

THE LINEAR SEQUENTIAL MODEL

Sometimes called the classic life cycle or the waterfall model, the linear sequential model suggests a systematic, sequential approach5 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 design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

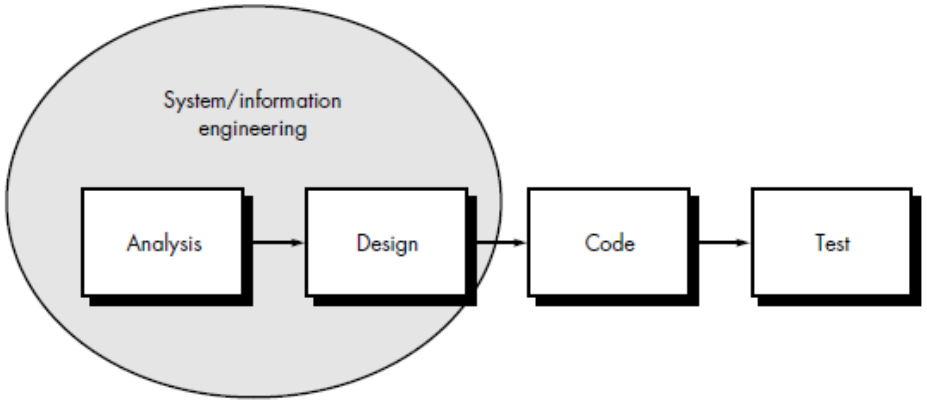


FIGURE 2.4
The linear sequential model

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 for the software, as well as required function, behaviour, 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.

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.

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.

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 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.

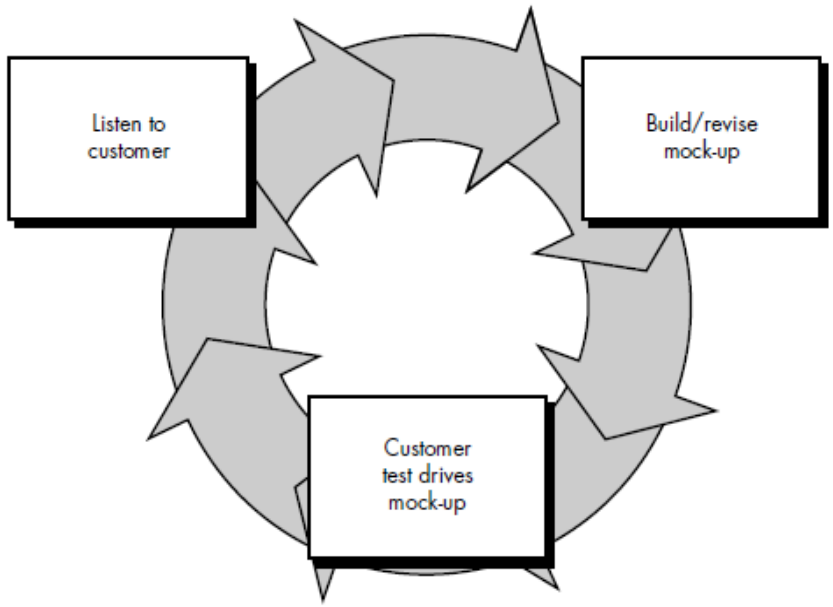
Yet, prototyping can also be problematic for the following reasons:

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.

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.

FIGURE 2.5
The prototyping paradigm



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 modelling. 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?

Data modelling. 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 characteristics (called attributes) of each object are identified and the relationships between these objects defined.

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.

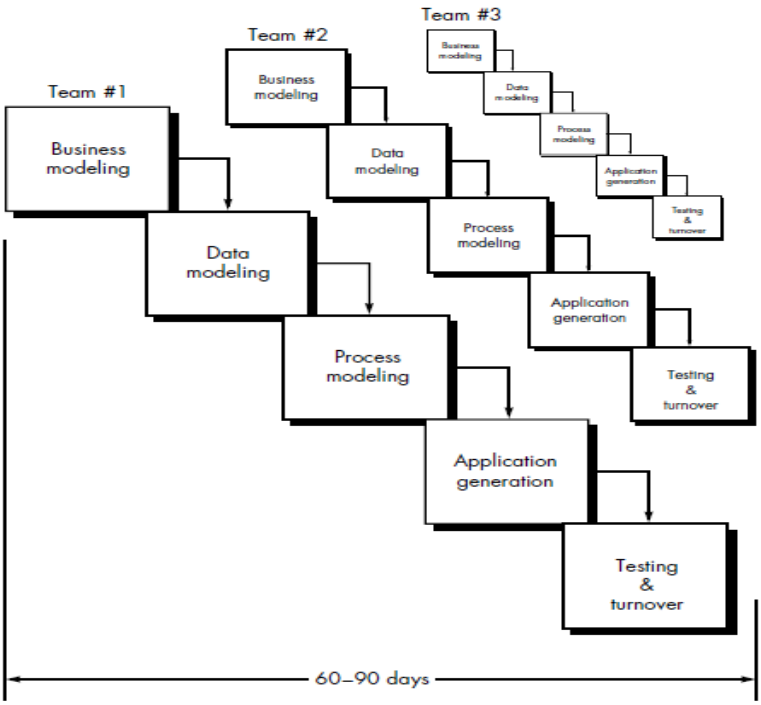
Application generation. RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases,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

Like all process models, the RAD approach has drawbacks:
• 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.

FIGURE 2.6
The RAD model



EVOLUTIONARY SOFTWARE PROCESS MODELS

There is growing recognition that software, like all complex systems, evolves over a period of time. 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 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 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.

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 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.

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.

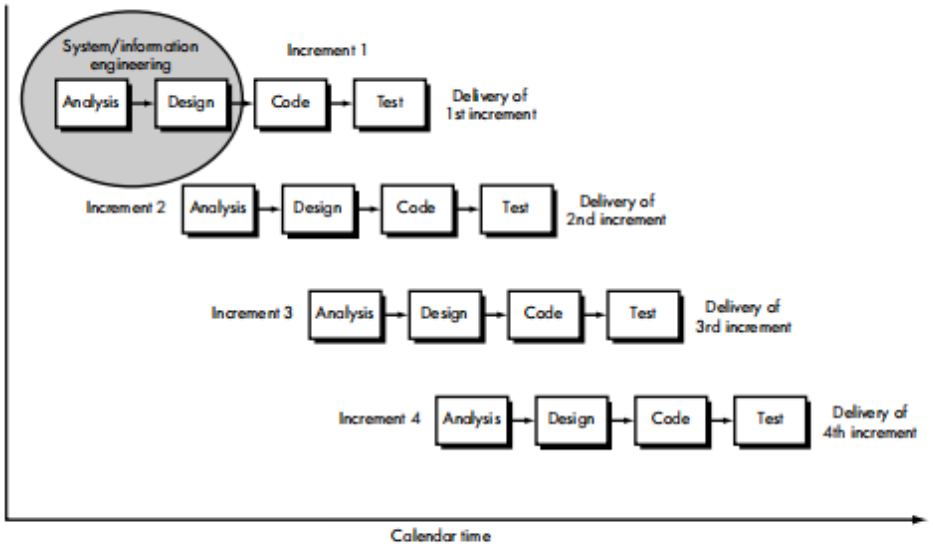


FIGURE 2.7 The incremental model

The incremental process model, like prototyping 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

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. Typically, there are between three and six task regions. Figure 2.8 depicts a spiral model that contains six task regions:

• Customer communication—tasks required to establish effective communication between developer and customer.
• Planning—tasks required to define resources, timelines, and other projectrelated 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 stage and implemented during the installation stage.

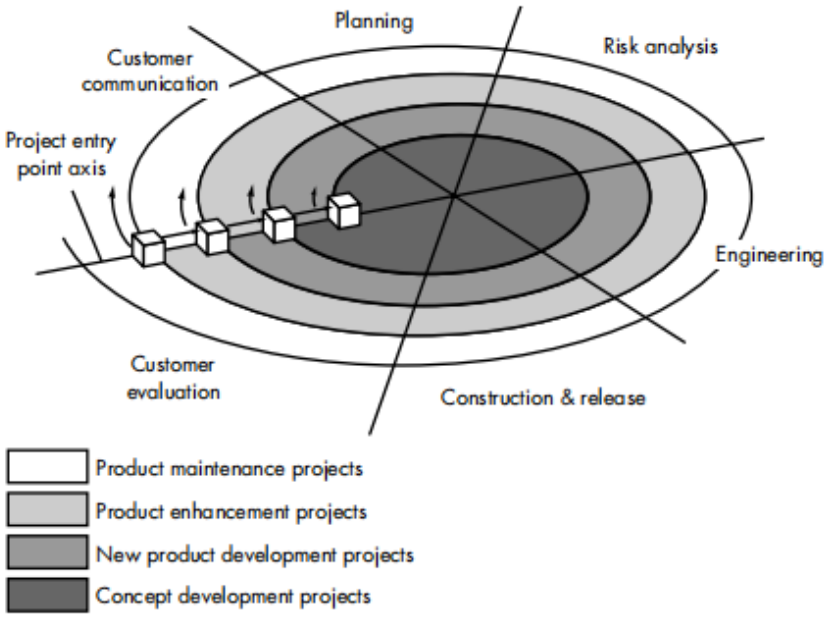


FIGURE 2.8
A typical spiral model

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) are applied.

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

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. 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.

The WINWIN Spiral Model

The spiral 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.

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.

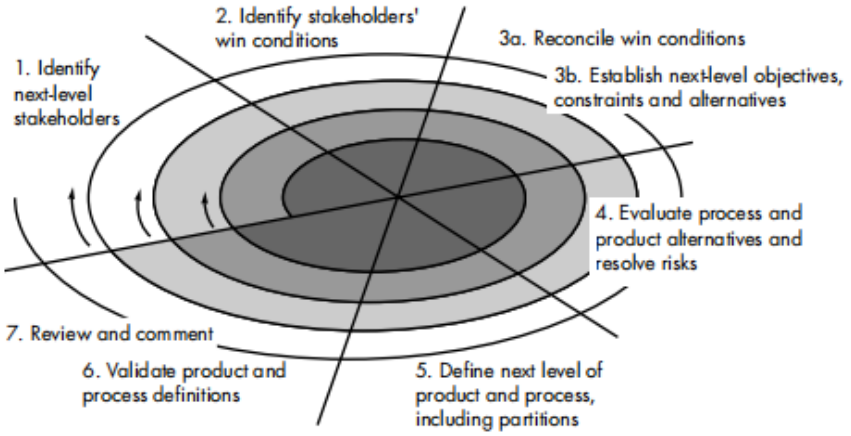


FIGURE 2.9 The WINWIN spiral model [BOE98].

Boehm's WINWIN spiral model 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:

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, 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 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.

The Concurrent Development Model

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

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 is accomplished by invoking the following tasks: prototyping and/or analysis modeling, requirements specification, and design

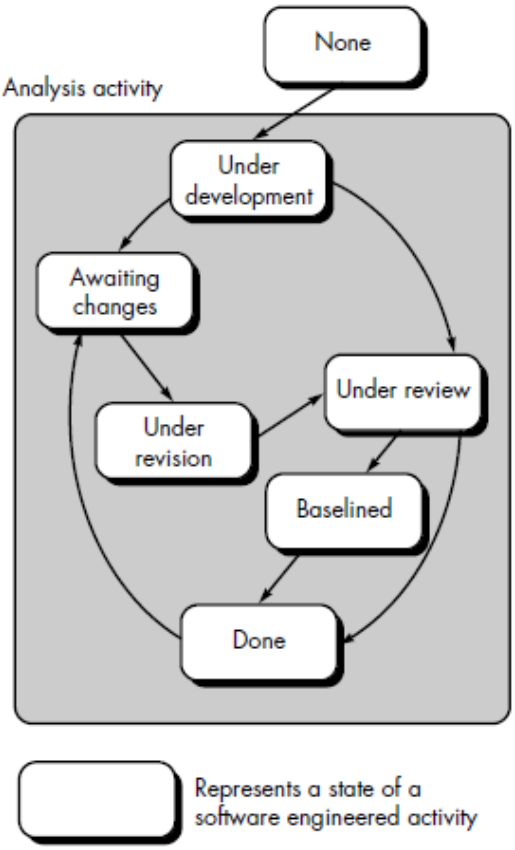
Figure 2.10 provides a schematic representation of one activity with the concurrent process model. The activity—analysis—may be in any one of the states10 noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities 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, 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/server11 applications. 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: 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 modelled 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 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.

FIGURE 2.10
One element of the concurrent process model



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 objectoriented 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.

The component-based development (CBD) model (Figure 2.11) incorporates many of the characteristics of the spiral model. It is evolutionary in nature, 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.

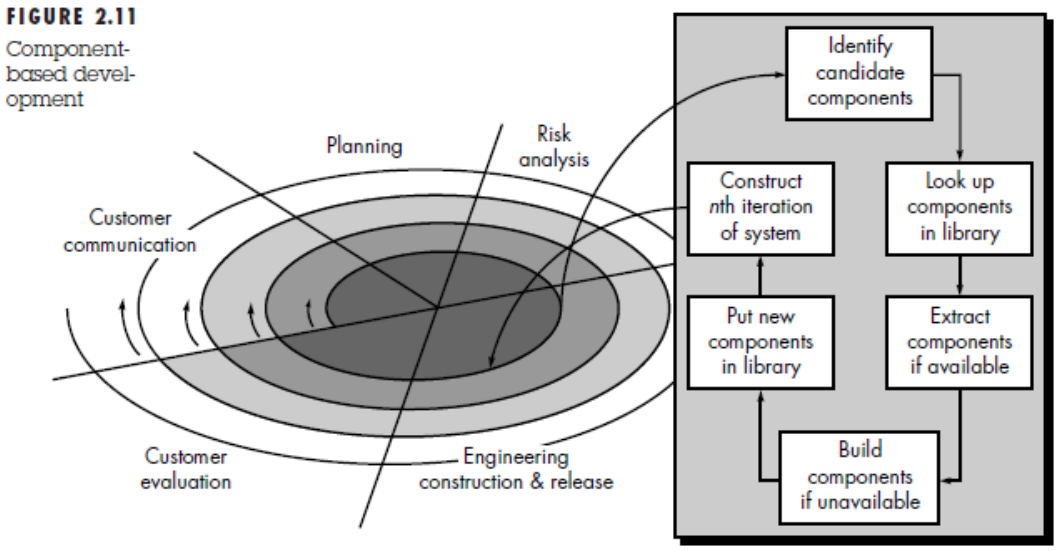


FIGURE 2.11
Component-based development

Classes created in past software engineering projects are stored in a class library or repository 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. 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.

The unified software development process 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 software will take

THE FORMAL METHODS MODEL

The formal methods model encompasses a set of 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 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.

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. 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.

PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer, but an obsessive over- reliance on process is also dangerous. In a brief essay, Margaret Davis [DAV95] com- ments 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 struc- tured programming languages (product) followed by structured analysis methods (process) followed

by data encapsulation (product) followed by the current emphasis on the Soft- ware 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 per- form 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 . . .

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 our- selves 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.

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 prod- uct and process is one important element in keeping creative people engaged as the transition from programming to software engineering is finalized.

PROJECT MANAGEMENT CONCEPTS

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 soft- ware 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.

1. The People:
The cultivation of motivated, highly skilled software people has been discussed since the 1960. 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".

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 that guides organizations in the creation of a mature software process.

2. The Product

Before a project can be planned, product1 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 rea- sonable (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 and continues as the first step in software requirements analysis 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. The Process

A software process 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.

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. Although the success rate for software projects has improved somewhat, our project failure rate remains higher than it should be.
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 common sense approach for planning, monitoring and controlling the project.

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 suc- cessful software project. They answered in the following way:
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 pre- ceding 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.

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. 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.

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."

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 soft- ware product or application.
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 of the characteristics that define an effective project man- ager 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 proj- ect. 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.

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 man- ager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

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 man- ager 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.

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 suggests three generic team organizations:

What factors should we consider when structuring a software team?

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

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

- 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.

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 suggests four “organizational paradigms” for software engineering teams:
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 soft- ware 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 break- through 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 per- formed 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. 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.
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- Jelled teams are the ideal, but they're not easy to achieve. At a minimum, be certain to avoid a "toxic environment and assists the teams in research, evaluation, and document preparation. The importance of a librarian cannot be overemphasized. The librarian acts as a con- troller, coordinator, and potentially, an evaluator of the software configuration.
A variation on the democratic decentralized team has been proposed by Constantine 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, channelling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:
•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.

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 non- interactive 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:

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

Formal, interpersonal procedures focus on quality assurance activities 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."

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.

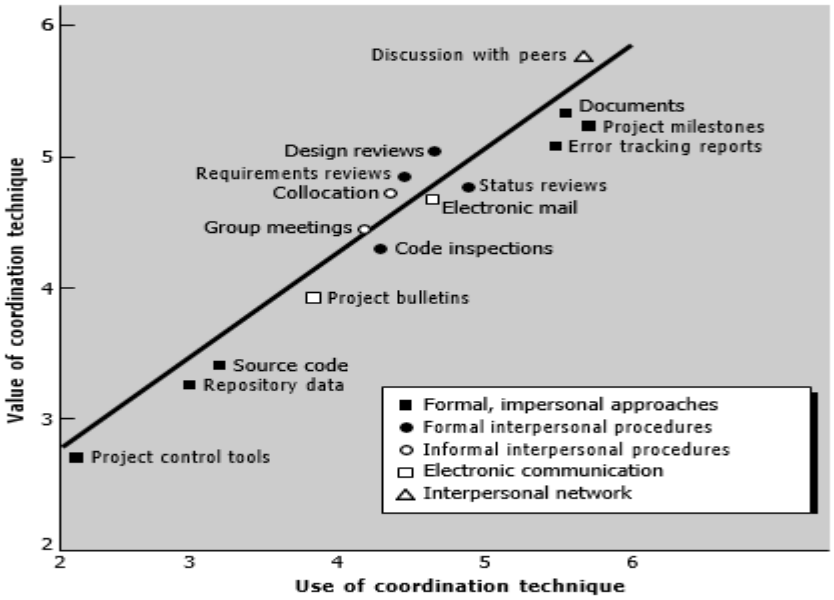


FIGURE 3.1 Value and Use of Coordination and Communication Techniques

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 require- ments 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.

1. Software Scope

The first software project management activity is the determination of software scope.
Scope is defined by answering the following questions:
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 manage- ment 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, maxi- mum 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.

2. Problem Decomposition

Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis. 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 con- fronted 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 Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.

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 soft- ware. Each can be further refined if the decomposition will make planning easier.

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 Chap- ter 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

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 frame- work activities must be created

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:

•Customer communication—tasks required to establish effective requirements elicitation between developer and customer.

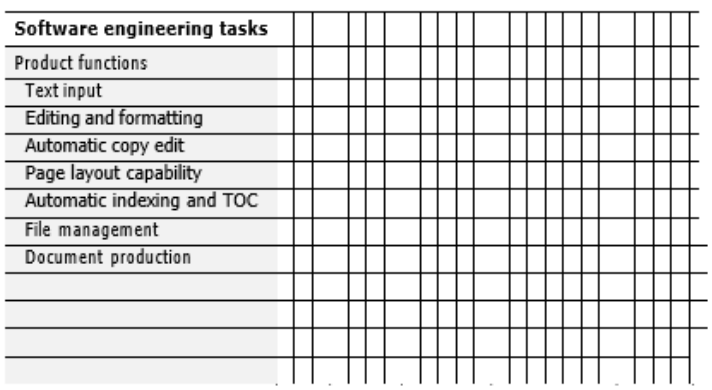
•Planning—tasks required to define resources, timelines, and other project- related information.

•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 pro- vide 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.

The team members who work on a product function will apply each of the frame- work 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-pro- cessing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each frame- work activity) would be entered in the following row.5 The job of the project man- ager (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.

FIGURE 3.2 Melding the Problem and the Process



2. Process Decomposition

A software team should have a significant degree of flexibility in choosing the soft- ware 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

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, 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.
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.

4.THE PROJECT

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 pro- vide 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 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 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.

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, mile- stones 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:

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 stake- holders also have responsibilities.

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 based on answers to earlier questions.

CRITICAL PRACTICES

The Airlie Council8 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

- 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.