Software Engineering

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 [IEE93a] 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).

Software Engineering layers

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed. Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.
Software engineering *tools* provide automated or semiautomated 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.

The Software Process

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

In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

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

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

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

Construction. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

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

For many software projects, framework activities are applied iteratively as a project progresses. That is, **communication, planning, modeling, construction,** and **deployment** are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of *umbrella activities*.

In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

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

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.
Software quality assurance—defines and conducts the activities required to ensure software quality.
Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.
Software configuration management—manages the effects of change throughout the software process.
Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Prescriptive process model

The Waterfall Model

There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable. The *waterfall model,* sometimes called the *classic life cycle*, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).
A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 2.4, the V-model [Buc99] depicts the relationship of quality

FIGURE 2.3 The waterfall model

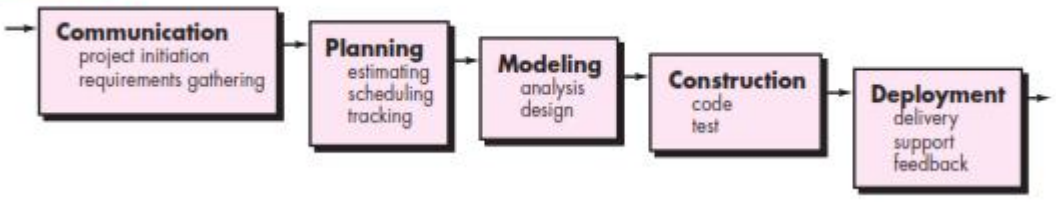
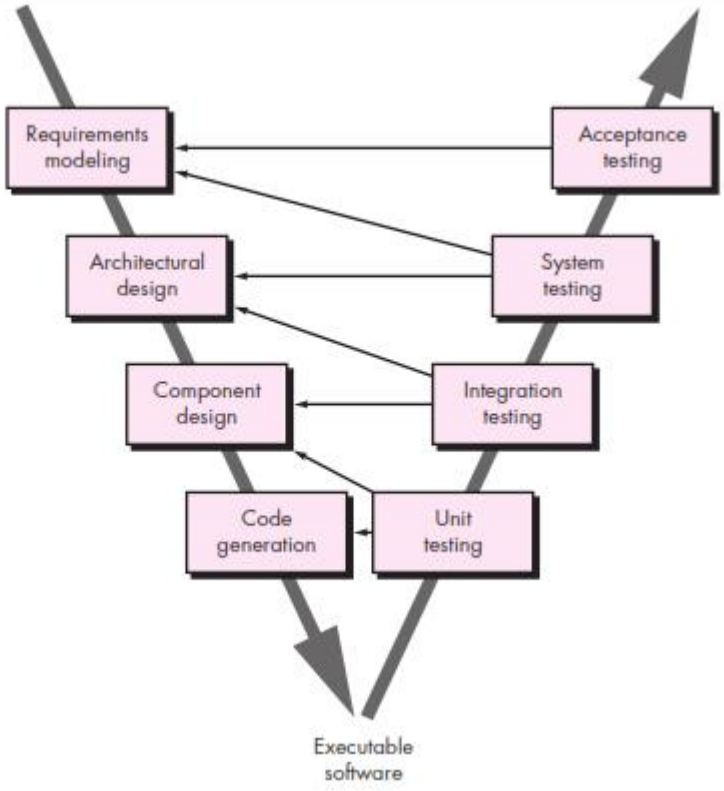


FIGURE 2.4
'he V-model



assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the Vbasic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.

In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. Howeverover the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy [Han95]. Among the problems that are sometimes encountered when the waterfall 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.

It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

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.

2.3.2 Incremental Process Models

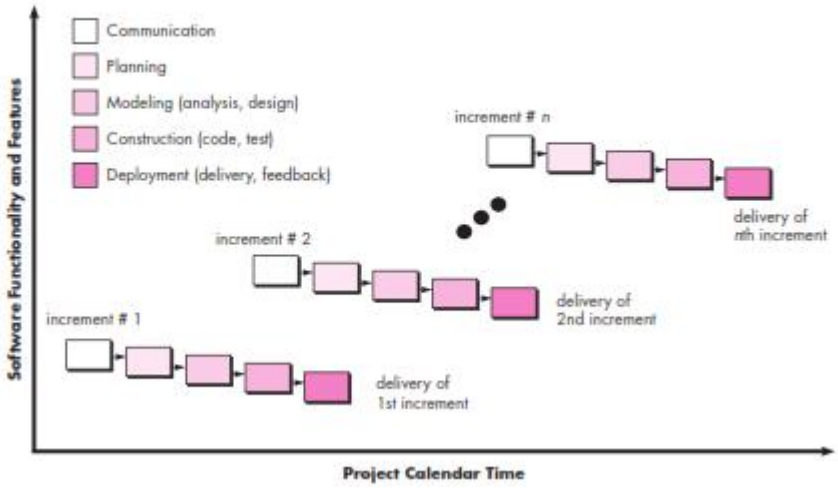
There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The *incremental* model combines elements of linear and parallel process flows discussed in Section 2.1. Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow (Section 2.3.3). 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 evaluation). As a result of use and/or evaluation, a

URE 2.5

mental



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. The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

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

2.3.3 Evolutionary Process Models

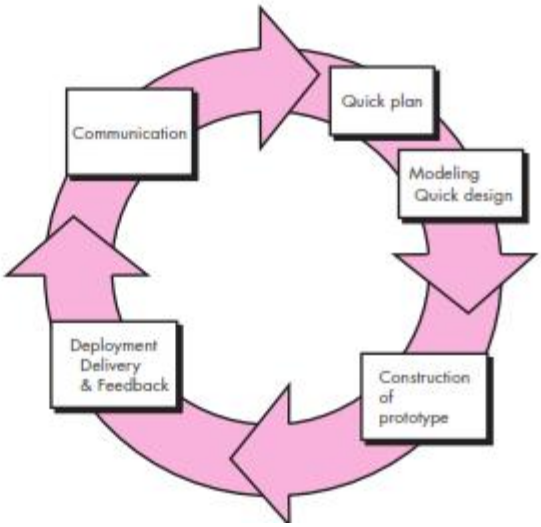
Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to
an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version 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, you need a process model that has been explicitly designed to accommodate a product that evolves over time.
Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, I present two common evolutionary process models.

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. 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. Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 2.6) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a “quick design”) occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display

FIGURE 2.6
The prototyping paradigm



Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. et, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't 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, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make 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, you may become comfortable 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.

The Spiral Model. Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm [Boe01a] describes the model in the following manner:

The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations,

increasingly more complete versions of the engineered system are produced.

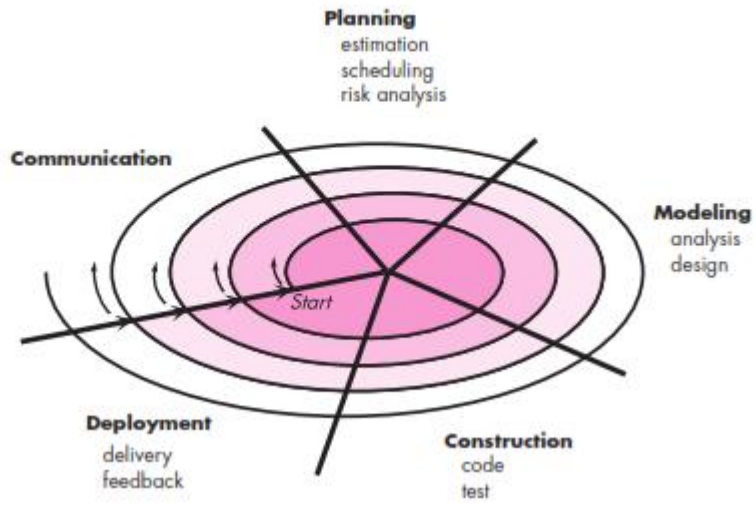


FIGURE 2.7 typical ...iral model

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction beginning at the center. Risk (Chapter 28) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

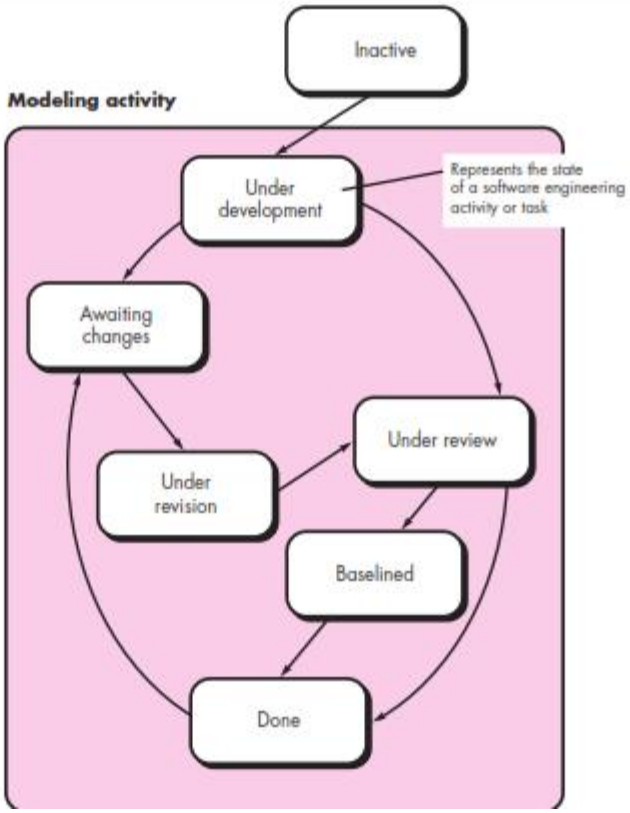
The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required To complete the software.

2.3.4 Concurrent Models

The *concurrent development model*, sometimes called *concurrent engineering*, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.

Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—**modeling**—may be in any one of the states noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

FIGURE 2.8 One element of the concurrent process model



For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **inactive** state while initial 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 modeling activity moves from the **under development** state into the **awaiting changes** state. Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event *analysis model correction,* which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state. Concurrent modeling 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, actions, and tasks to a sequence of events, it defines a process network. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

Software Porject Estimation

Software cost and effort estimation will never be an exact science. Tvariables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that promates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up-front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer

26.6.2 Problem-Based Estimation

In Chapter 25, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software Scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single-baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity,

and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for past productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.

An implicit indication of the degree of uncertainty is provided when a range of values is specified. A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) *S* can be computed as a weighted average of the optimistic

A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) *S* can be computed as a weighted average of the optimistic) estimates.

gives heaviest credence to the "most likely" estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only Reasonable answer to this question is: "You can't be sure." Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

26.6.3 An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, I consider a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mousedigitizer, high-resolution color display, and laser printersoftware scope can be developed:

FIGURE 26.2 Estimation table for the LOC methods

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

\

26.6.4 An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table presented in Figure 26.3, you would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. An FP value is computed using the technique discussed in Chapter 23. For the purposes of this estimate, the complexity weighting factor is assumed to be average. Figure 26.3 presents the results of this estimate.

FIGURE 26.3 Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
Count total						*320*

Each of the complexity weighting factors is estimated, and the value adjustment factor is computed as described in Chapter 23:

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Value adjustment factor	**1.17**

Earned Value Analysis

In Section 27.5, I discussed a number of qualitative approaches to project tracking. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA). Humphrey [Hum95] discusses earned value in the following manner:

Stated even more simply, earned value is a measure of progress. It enables you to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [Fle98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project." To determine the earned value, the following steps are performed:

1. The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWS the effort planned for work task *i*. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWS values for all work tasks that should have been completed by that point in time on the project schedule.

2. The BCWS values for all work tasks are summed to derive the *budget at completion* (BAC). Hence,

BAC _ _ (BCWS) for all tasks *k*

3. Next, the value for *budgeted cost of work performed* (BCWP) is computed. The *k* value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Wilkens [Wil99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed."

Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

Schedule performance index, SPI _BCWP BCWS is Schedule variance, SV _ BCWP _ BCWS

SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

Percent scheduled for completion _provides an indication of the percentage of work that should have been completed by time *t*.
Percent complete _
BCWP
BAC
BCWS
BAC
provides a quantitative indication of the percent of completeness of the project at a given point in time *t*.
It is also possible to compute the *actual cost of work performed* (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute
Cost performance index, CPI _
Cost variance, CV _ BCWP _ ACWP
A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.
Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.
BCWP
ACWP

Project scheduling

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled.
Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

27.2.1 Basic Principles

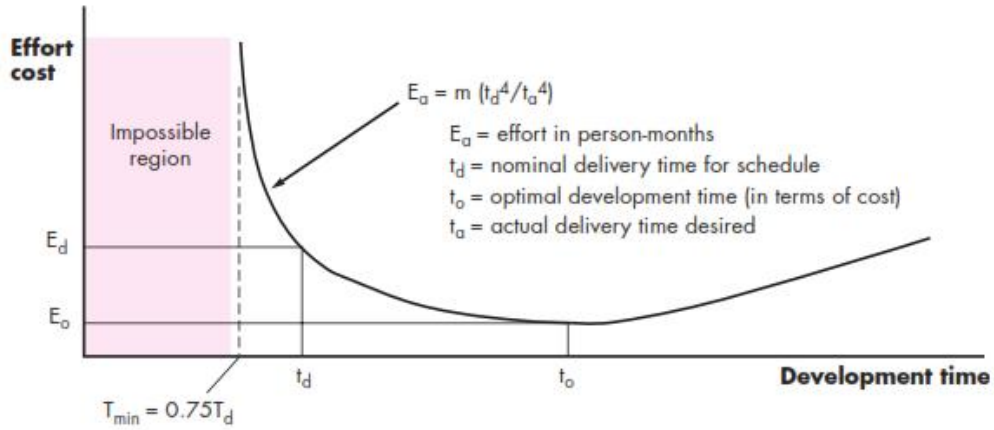
Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.
Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort. On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.
Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.
Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.
Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved.
Each of these principles is applied as the project schedule evolves.

27.2.2 The Relationship Between People and Effort

In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved. (We can rarely afford the luxury of approaching a 10 person-year effort with one person working for 10 years!)

FIGURE 27.1 The relationship between effort and delivery time



There is a common myth that is still believed by many managers who are responsible for software development projects: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind. In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources). The *Putnam-Norden-Rayleigh (PNR) Curve* provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 27.1. The curve indicates a minimum value *t* that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of *t*
(i.e., as we try to accelerate delivery), the curve rises nonlinearly.

As an example, we assume that a project team has estimated a level of effort *E* will be required to achieve a nominal delivery time *t*
d
that is optimal in terms of
d
schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of *t*
. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond 0.75*t*
d
. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes

very high. The PNR curve also indicates that the lowest cost delivery option,

. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.
The software equation [Put92] introduced in Chapter 26 is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), *L,* is related to effort and development time by the equation:
where *E* is development effort in person-months, *P* is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for *P* range between 2000 and 12,000), and *t* is the project duration in calendar months.
Rearranging this software equation, we can arrive at an expression for development effort *E*:
where *E* is the effort expended (in person-years) over the entire life cycle for software

where *E* is the effort expended (in person-years) over the entire life cycle for software development and maintenance and *t* is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).
This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described
in Equation (27.1) yields:

This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

27.2.3 Effort Distribution

Each of the software project estimation techniques discussed in Chapter 26 leads to estimates of work units (e.g., person-months) required to complete software development. A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule*. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.
This effort distribution should be used as a guideline only.

The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an

organization to large expenditures with high risk. Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort ex- pended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.
Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated

Risk Management

Reactive Risk versus Proactive Risk:

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [Tho92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would react in some heroic way.
Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire-fighting mode*. When this fails, "crisis management" [Cha92] takes over and the project is in real jeopardy.
A considerably more intelligent strategy for risk management is to be proactive. A *proactive* strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective
is to avoid risk, but because not all risks can be avoided, the team works to
develop a contingency plan that will enable it to respond in a controlled and effective
manner.
Throughout the remainder of this chapter, I discuss a proactive strategy
for risk management.

Software Risk:

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder,
and requirements problems and their impact on a software project.
In Chapter 26, project complexity,
size,
and the degree of structural uncertainty
were also defined as project (and estimation) risk factors.
Technical risks threaten the quality and timeliness of the software to be produced.
If a technical risk becomes a reality, implementation may become difficult or impossible.
Technical
risks identify potential design, implementation, interface,
verification,
and maintenance problems. In addition, specification ambiguity,
technical
uncertainty,
technical obsolescence,
and "leading-edge" technology are also risk
factors.
Technical
risks occur because the problem is harder to solve than you
thought
it would be.
Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building
a
product that no longer fits into the overall
business strategy for the company
(strategic
risk), (3) building a product that the sales force doesn't
understand how to
sell
(sales risk), (4) losing the support of senior management due to a change in focus
or
a change in people (management risk), and (5) losing budgetary or personnel commitment
(budget risks).
It is extremely important to note that simple risk categorization won't always work. Some risks are simply unpredictable in advance.
Another general categorization of risks has been proposed by Charette [Cha89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development

environment). *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk Identification:

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule,
resource loading, etc.). By identifying known and predictable risks,
the
project manager takes a first step toward avoiding them when possible and controlling them when necessary.
There are two distinct types of risks for each of the categories that have been presented in Section 28.2: generic risks and product-specific risks. *Generic*
risks
are a
potential threat to every software project. *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. To
identify product-specific
risks,
the project plan and the software statement of scope are examined, and an
answer
to the following question is developed: "What special characteristics of this
product
may threaten our project plan?"
One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

•
•

Product size—risks associated with the overall size of the software to be built or modified.
Integrate—a consideration of risk must be integrated
into the software process.
Emphasize a continuous process—the team
must be vigilant throughout the software process,
modifying identified risks as more information is
known and adding new ones as better insight is
achieved.
Develop a shared product vision—if all
stakeholders share the same vision of the software, it is likely that better risk identification and assessment will occur.

Encourage teamwork—the talents, skills, and knowledge of all stakeholders should be pooled when risk management activities are conducted.
Business impact—risks associated with constraints imposed by management or the marketplace.
Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.
Development environment—risks associated with the availability and quality of the tools to be used to build the product.
Technology to be built—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Risk Imapct:

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, you want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.
Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], you can apply the following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) using Figure 28.1, determine the impact for each component based on the criteria shown, and (3) complete the risk table and analyze the results as described in the preceding sections.
The overall *risk exposure* RE is determined using the following relationship [Hal98]:
RE _ *P* _ *C*
where *P* is the probability of occurrence for a risk, and *C* is the cost to the project should the risk occur.
For example, assume that the software team defines a project risk in the following manner:
Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.
Risk probability. 80 percent (likely).

Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 _ 100 _ 14 _ $25,200.
Risk exposure. RE _ 0.80 _ 25,200 _ $20,200.

Risk Refinement

This general condition can be refined in the following manner:
Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.