MODULE- 1

- **Computer software** is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.
- **Engineering**- Application of science, tools and methods to find cost effective solution to problems.
- **Software engineering** encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high quality computer software.
- Who does it? **Software engineers** build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.
- Why is it important? **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.
- Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.
- What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.
- What is the work product? From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software. But from the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

THE NATURE 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, by a network of computers that are accessible by local hardware. Whether it resides within a mobile 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 derived from data acquired from dozens of independent sources.

- 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. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.
- Sophistication and complexity can produce brilliant results when a system succeeds, but they can also pose huge problems for those who must build complex systems.
- The questions that are asked to programmers when modern computer-based systems are built are:
  - ✓ Why does it take so long to get software finished?
  - ✓ Why are development costs so high?
  - ✓ Why can't we find all errors before we give the software to our customers?
  - ✓ Why do we spend so much time and effort maintaining existing programs?
  - ✓ Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

Definition of Software:

Software is:

(1) instructions (computer programs) that when executed provide desired features, function, and performance;

(2) data structures that enable the programs to adequately manipulate information, and

(3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics Of Software:

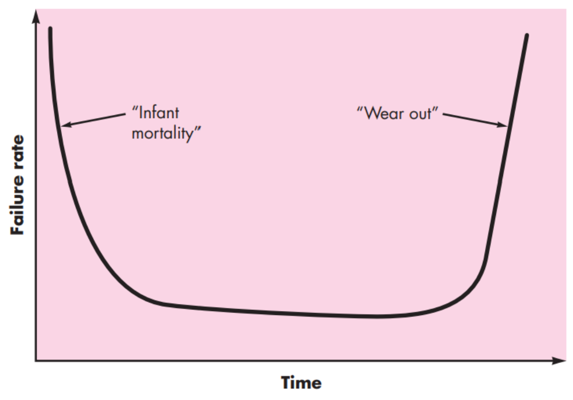
1. **Software is developed or engineered; it is not manufactured in the classical sense.**
   Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are 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. 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 (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out. Software is not susceptible to the environmental condition that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!

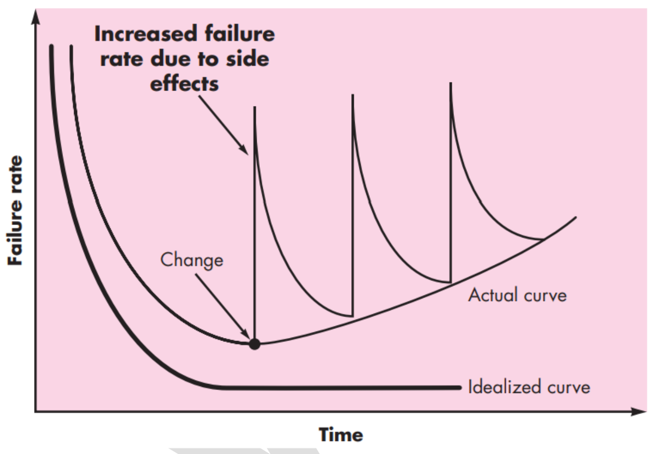
FIGURE 1.1
Failure curve for hardware



This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change. Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

FIGURE 1.2

Failure curves for software



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

As an engineering discipline evolves, a collection of standard design components is created. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design. 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. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software Application Domains

There are seven broad categories of computer software:

1) **System software:** A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.
2) **Application software:** Stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).
3) **Engineering/scientific software**: It has 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.
4) **Embedded software**: Resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).
5) **Product-line software**: Designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets,

computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

6) **Web applications:** called “WebApps,” this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

7) **Artificial intelligence software**: Makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

New challenges for Software Engineers:

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

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

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Legacy software

- Legacy software is an older version of a computer program or application that is still in use, even after newer versions are available.
- Legacy systems are often outdated, but still meet the needs they were originally designed for.
- Legacy systems are often critical to day-to-day operations, but can be challenging to maintain and support.
- They may not be compatible with newer operating systems or hardware, and they don't receive technical support or updates from their developers.
- This can make them vulnerable to data breaches, which can cost organizations millions of dollars.
- A common legacy system example is Microsoft's Windows 7, which was released in 2009 and was no longer supported after 2020.

However, as time passes, legacy systems often evolve for one or more of the following reasons:

• The software must be adapted to meet the needs of new computing environments or technology.

• The software must be enhanced to implement new business requirements.

• The software must be extended to make it interoperable with other more modern systems or databases.

• The software must be re-architected to make it viable within a network environment.

THE UNIQUE NATURE OF WEBAPPS

The following attributes are encountered in majority of WebApps.

1. **Network intensiveness:** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
2. **Concurrency:** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
3. **Unpredictable load**: The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday. Performance. If a WebApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.

4. **Availability:** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.
5. **Data driven:** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
6. **Content sensitive:** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
7. **Continuous evolution:** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.
8. **Immediacy:** Although immediacy—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.
9. **Security:** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.
10. **Aesthetics:** An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

SOFTWARE MYTHS

Management myths:

Myth: We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is “no.”

Myth: If we get behind schedule, we can add more programmers and catch up .

Reality: Software development is not a mechanistic process like manufacturing. 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

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

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous “statement of objectives” is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: Software requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner’s myths

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 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 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.

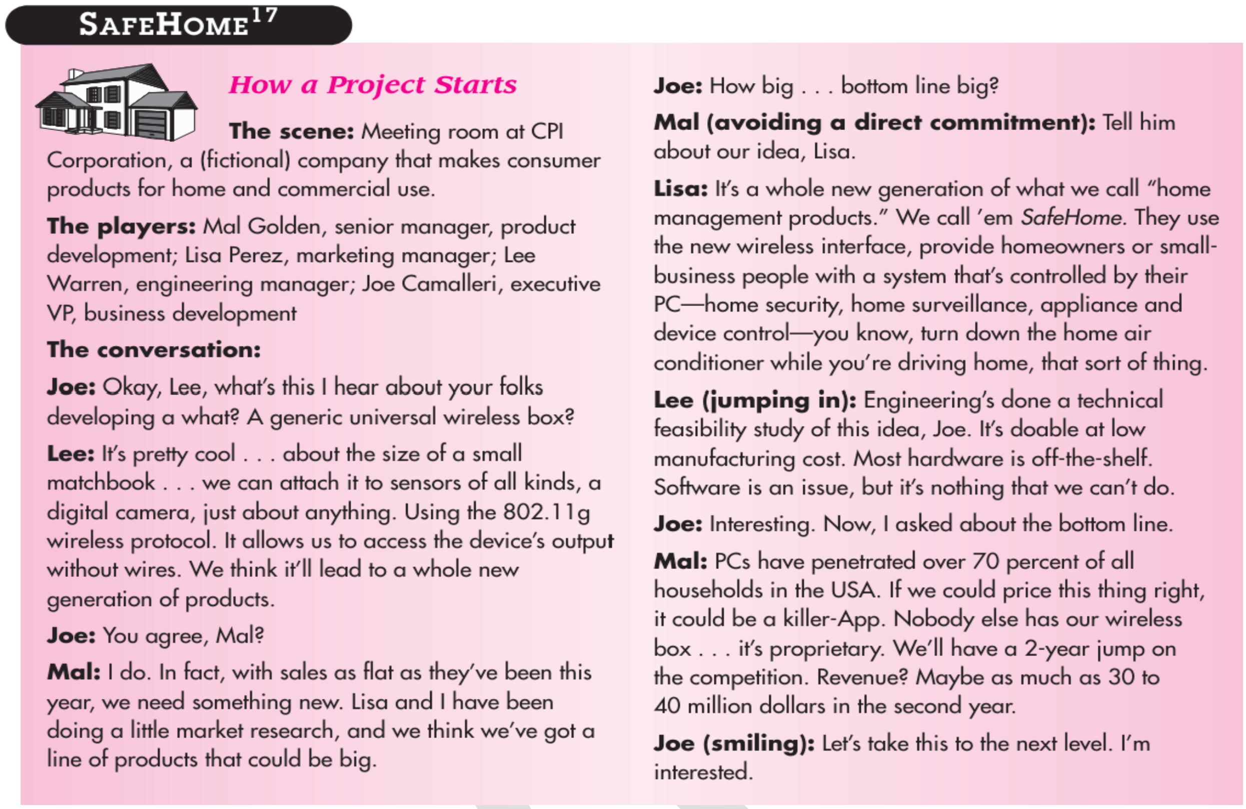
Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide 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 a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

HOW IT ALL STARTS



SAFEHOME[17]

How a Project Starts

The scene: Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

The conversation:

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

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

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

SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges of the twenty-first century, we must consider:

- Software's have impacted our life so much that, the number of people who have interest in the features and functions provided by a specific application has grown dramatically. When a new application or embedded system is to be built, all suggestions must be considered. Efforts should be made to understand the problem before a software solution is developed.
- The information technology requirements demanded by individuals, businesses, and governments is increasing with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. It follows that design becomes a pivotal activity.
- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and

major enterprises can experience anything from minor inconvenience to catastrophic failures. Software should exhibit high quality.

- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. It follows that software should be maintainable.

➢ Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

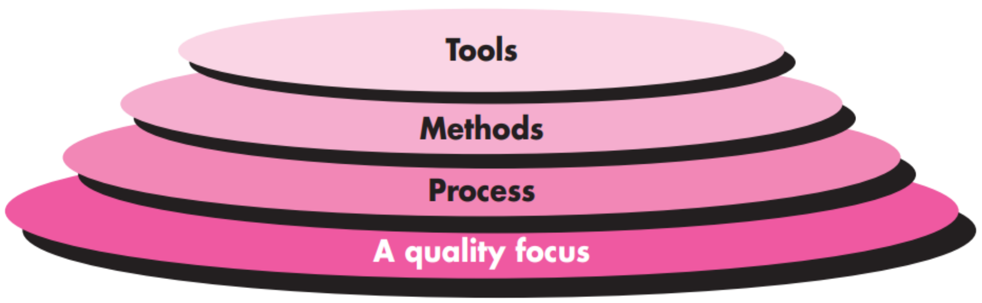
➢ Software engineering is a layered technology.

- 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 promotes 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 basis 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. T
- 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.

FIGURE 1.3

Software engineering layers



PROCESS MODEL

1. Defining a Framework Activity
2. Identifying a Task Set
3. Process Patterns

SOFTWARE PROCESS FRAMEWORK

- A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
- An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
- A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.
- 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.
- 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:** 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:** A software engineer creates 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.

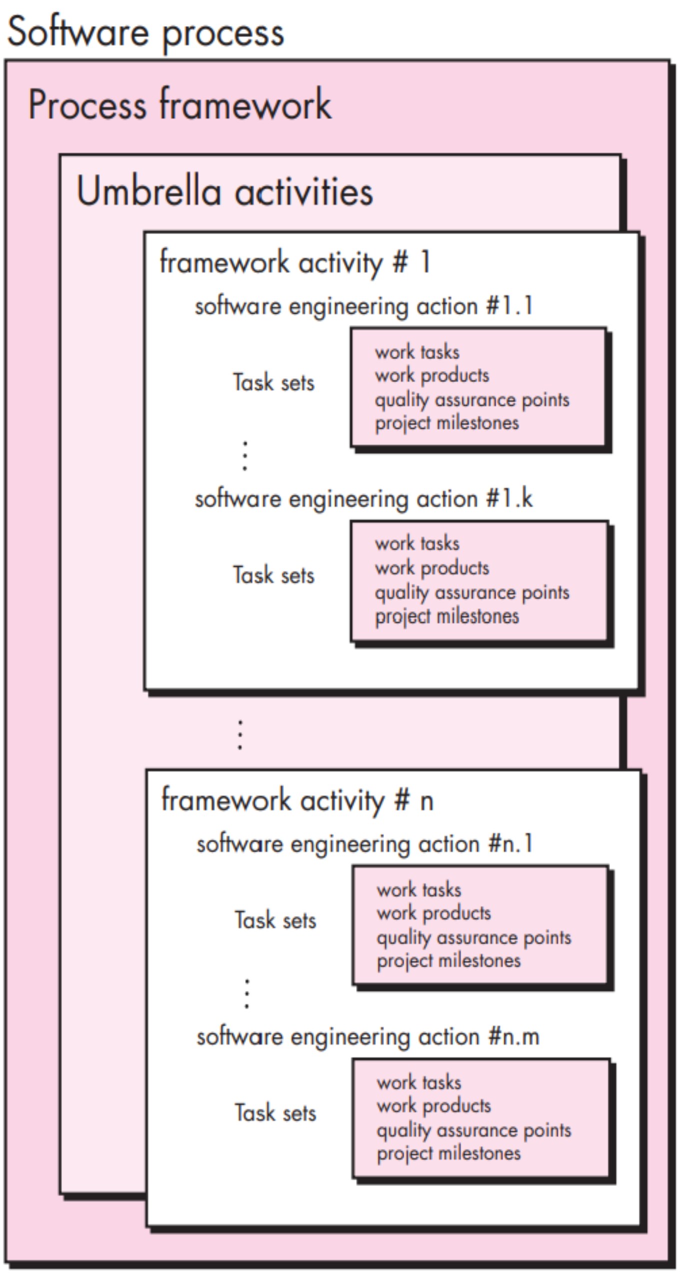
- For many software projects, framework activities are applied iteratively as a project progresses.
- Software engineering process framework activities are accompanied by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team 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.
- **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.

FIGURE 2.1

A software process framework



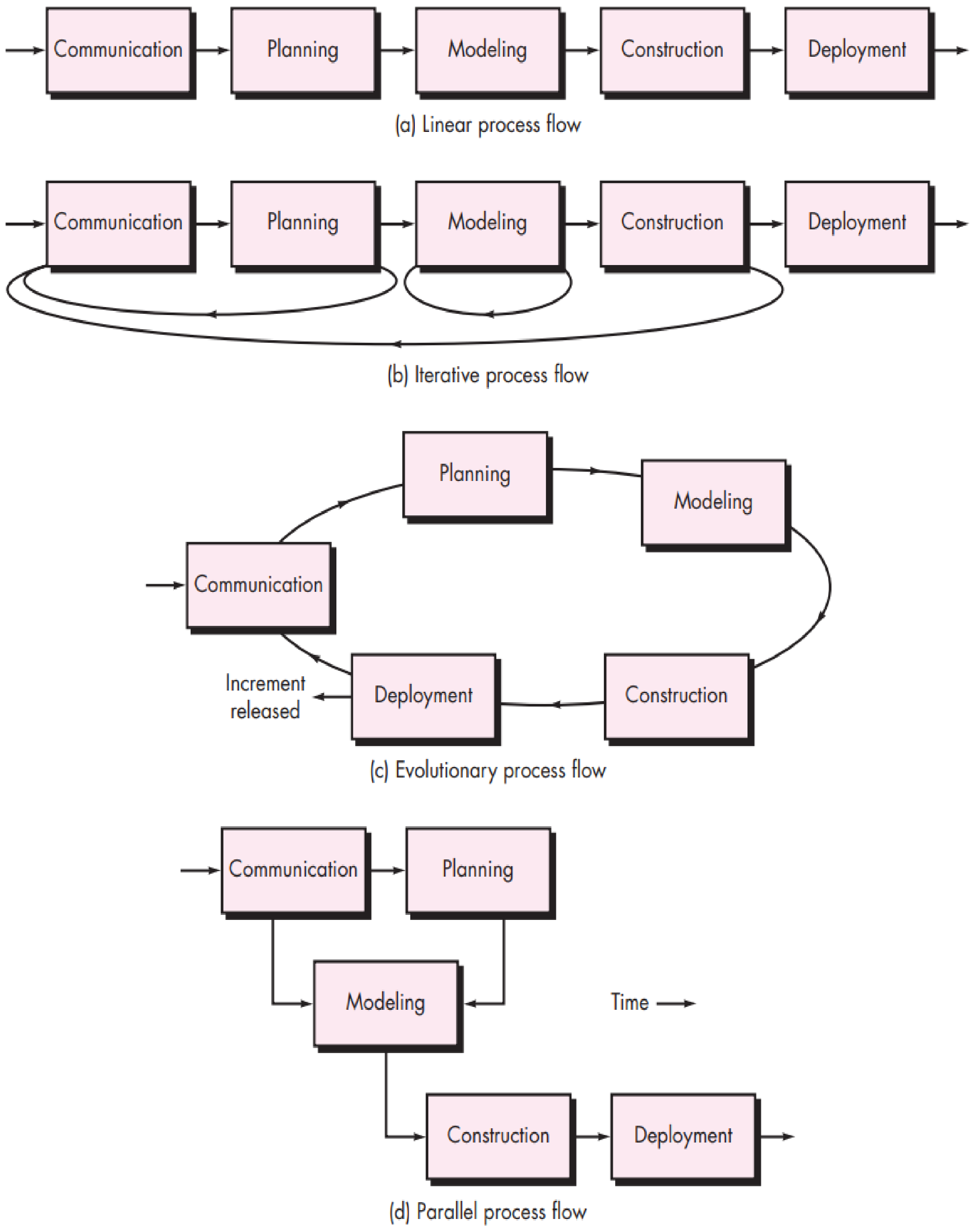
- One important aspect of the software process is **process flow**—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2.

- **A linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a).
- An **iterative process flow** repeats one or more of the activities before proceeding to the next (Figure 2.2b).
- An **evolutionary process flow** executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c).
- A **parallel process flow** (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

IDENTIFYING A TASK SET

- A **task set** defines the actual work to be done to accomplish the objectives of a software engineering action.
- For example, elicitation (more commonly called "requirements gathering") is an important software engineering action that occurs during the communication activity.
- The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.
- For a small, relatively simple project, the task set for requirements gathering might look like this:
  1. Make a list of stakeholders for the project.
  2. Invite all stakeholders to an informal meeting.
  3. Ask each stakeholder to make a list of features and functions required.
  4. Discuss requirements and build a final list.
  5. Prioritize requirements.
  6. Note areas of uncertainty.

FIGURE 2.2 Process flow



PROCESS PATTERNS

- A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- A process pattern provides you with a template - a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Ambler [Amb98] has proposed a template for describing a process pattern:

  Pattern Name: The pattern is given a meaningful name describing it within the context of the software process (e.g., Technical Reviews).

  Forces: The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

  Type: The pattern type is specified.

  There are three types:

  1. Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be Establishing Communication. This pattern would incorporate the task pattern Requirements Gathering and others.

  2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., Requirements Gathering is a task pattern).

  3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be Spiral Model or Prototyping.

  Initial context: Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:

  (1) What organizational or team-related activities have already occurred?

  (2) What is the entry state for the process?

  (3) What software engineering information or project information already exists?

For example, the Planning pattern (a stage pattern) requires that

(1) customers and software engineers have established a collaborative communication;

(2) successful completion of a number of task patterns [specified] for the Communication pattern has occurred; and

(3) the project scope, basic business requirements, and project constraints are known. Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context: Describes the conditions that will result once the pattern has been successfully implemented.

Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?

(2) What is the exit state for the process?

(3) What software engineering information or project information has been developed?

Related Patterns: Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form.

For example, the stage pattern Communication encompasses the task patterns: ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and ScenarioCreation.

Known Uses and Examples: Indicate the specific instances in which the pattern is applicable. For example, Communication is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

An Example Process Pattern

Pattern name: RequirementsUnclear

Intent: This pattern describes an approach for building a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

Type: Phase pattern.

Initial context: The following conditions must be met prior to the initiation of this pattern:

(1) stakeholders have been identified;

(2) a mode of communication between stakeholders and the software team has been established;
(3) the overriding software problem to be solved has been identified by stakeholders;
(4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem. Requirements are hazy or non-existent, yet there is clear recognition that there is a problem to be solved, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Resulting context: A software prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders.

Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern. Related patterns. The following patterns are related to this pattern: CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.

Known uses and examples: Prototyping is recommended when requirements are uncertain.

PROCESS ASSESSMENT AND IMPROVEMENT

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)-**provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI) -** provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].
- **SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].
- **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

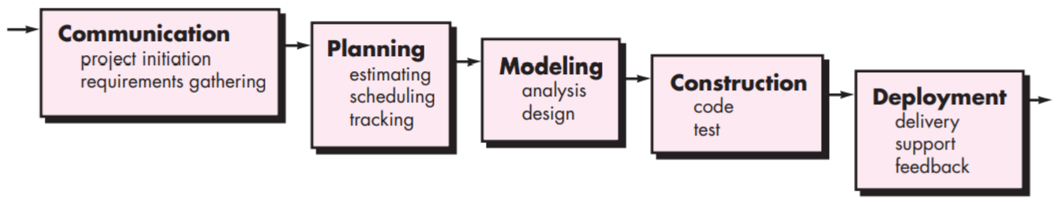
PERSONAL AND TEAM PROCESS MODELS

- In software engineering, personal software process (PSP) models focus on improving the performance of individuals, while team software process models focus on improving the performance of a group.
- PSP is a bottom-up approach that includes tools and techniques to help software engineers improve the quality of their work.
- PSP provides a framework that helps engineers analyze their performance and achieve goals.
- Team software process models depend on a group of people.

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).

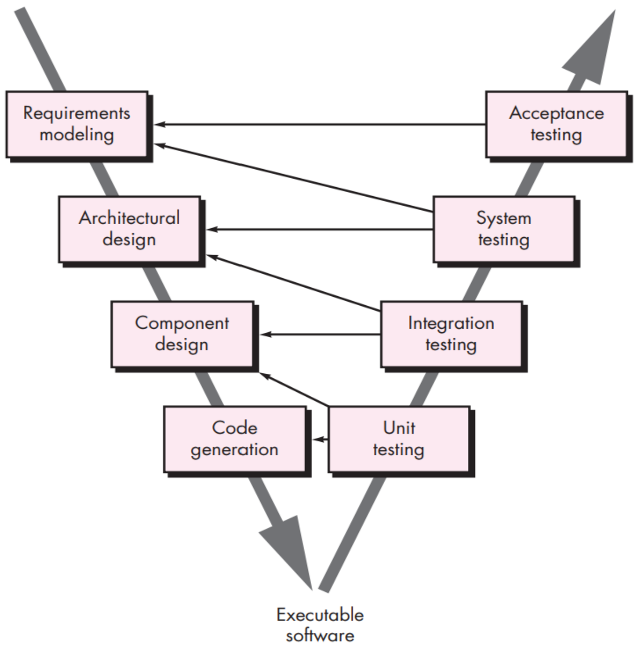
FIGURE 2.3 The waterfall model



- 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 assurance actions to the actions associated with communication, modeling, and early construction activities.

- As a software team moves down the left side of the V, basic 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.
- However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy.
- Problems that are 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.
    2. 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.
    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 diastrous.

FIGURE 2.4
The V-model



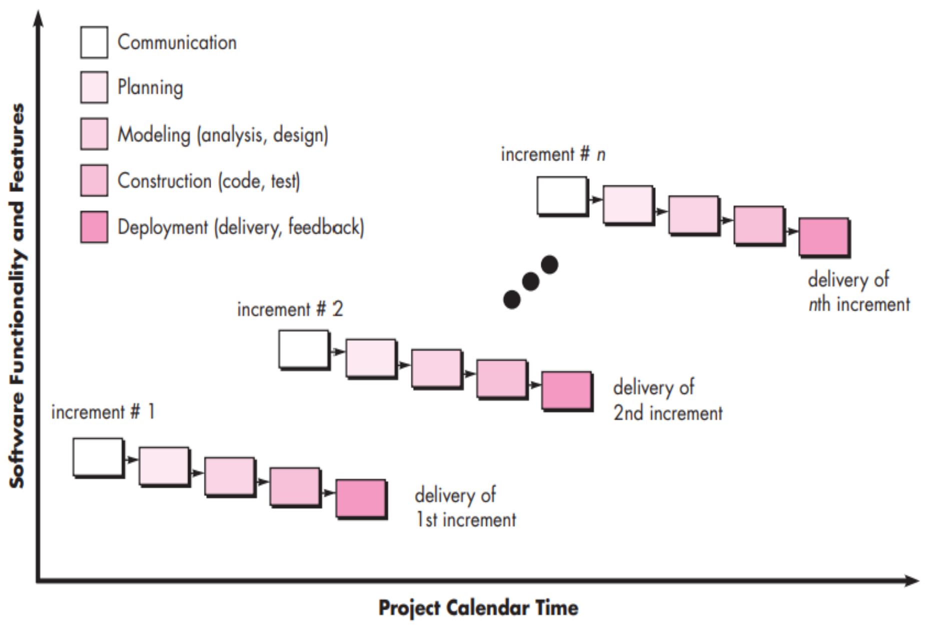
INCREMENTAL PROCESS MODEL

- 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 .
- 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 in a manner that is similar to the increments produced by an evolutionary process flow.
- 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 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.8 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.

FIGURE 2.5

The incremental model


Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
Software Functionality and Features
increment # 1
delivery of 1st increment
increment # 2
delivery of 2nd increment
increment # n
delivery of nth increment
Project Calendar Time

EVOLUTIONARY PROCESS MODELS

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. There are two common evolutionary process models:

1) Prototyping

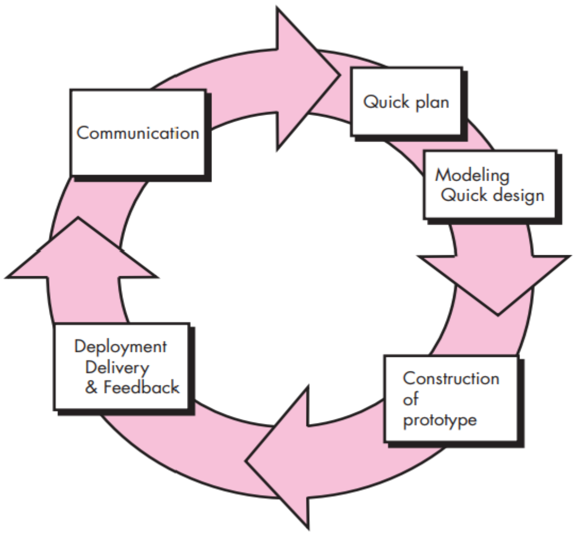
2) The Spiral Model

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.
- 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 formats).
- The quick design leads to the construction of a prototype.
- The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
- Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.
- Ideally, the prototype serves as a mechanism for identifying software requirements.
- If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.
- The prototype can serve as “the first system.” Although some prototypes are built as “throwaways,” others are evolutionary in the sense that the prototype slowly evolves into the actual system
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately.
- Yet, prototyping can be problematic for the following reasons:
  - ➢ Stakeholders see the working version of the software, unaware that the prototype is held together randomly, 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 demand that “a few fixes” be applied to make the prototype a working product. Too often, software development management relents.
  - ➢ 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.

FIGURE 2.6
The prototyping paradigm



THE SPIRAL MODEL

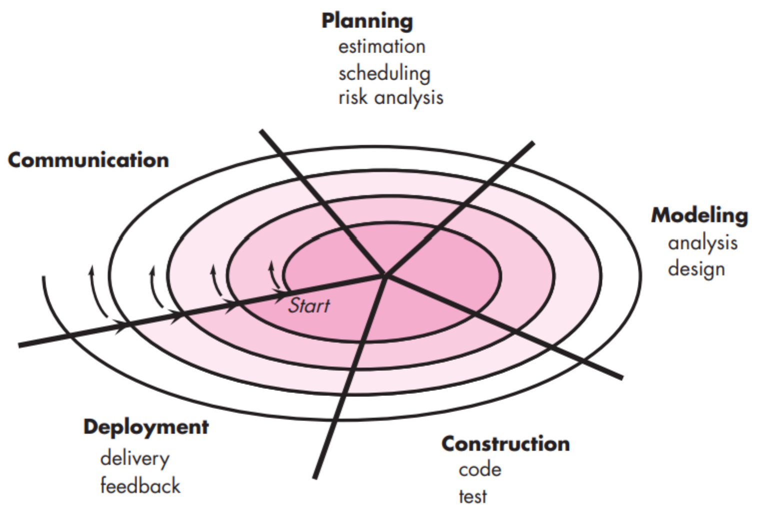
- Originally proposed by Barry Boehm the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software. Boehm 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.
- A spiral model is divided into a set of framework activities defined by the software engineering team. 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. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.
- Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences.
- The new product will evolve through a number of iterations around the spiral.
- 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 you 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.

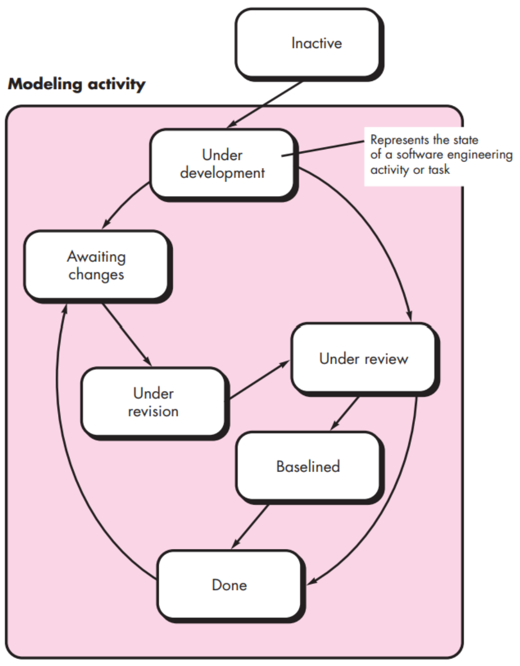
FIGURE 2.7
A typical spiral model



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 modelling 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
One element of the concurrent process model



SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models.

Component-Based Development

- Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.
- Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes.
- Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps
  1. Available component-based products are researched and evaluated for the application domain in question.
  2. Component integration issues are considered.
  3. A software architecture is designed to accommodate the components.
  4. Components are integrated into the architecture.
  5. Comprehensive testing is conducted to ensure proper functionality.
- The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

The Formal Methods Model

- The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable you 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 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 you to discover and correct errors that might otherwise go undetected.
- Although not a mainstream approach, the formal methods model offers the promise of defect-free software.
- Yet, concern about its applicability in a business environment has been voiced:
  - ✓ The development of formal models is currently quite time consuming and expensive
  - ✓ Because few software developers have the necessary background to apply formal methods, extensive training is required.
  - ✓ It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Aspect-Oriented Software Development

- ✓ Aspectual requirements define those crosscutting concerns that have an impact across the software architecture.
- ✓ Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".