UNIT-I: Introduction to Software Engineering

Syllabus

a. NATURE OF SOFTWARE
- ➢ Defining Software
- ➢ Software Application Domains
- ➢ Legacy Software

b. SOFTWARE PROCESS

c. SOFTWARE ENGINEERING PRACTICE
- ➢ The Essence of Practice
- ➢ General Principles

d. SOFTWARE MYTHS

e. GENERIC PROCESS MODEL
- ➢ Defining a Framework Activity
- ➢ Identifying a Task Set
- ➢ Process Patterns

f. PROCESS ASSESSMENT AND IMPROVEMENT

g. PERSPECTIVE PROCESS MODELS
- ➢ The Waterfall Model
- ➢ Incremental Process Models
- ➢ Evolutionary Process Models
- ➢ Concurrent Models

h. SPECIALIZED PROCESS MODELS
- ➢ Component-Based Development
- ➢ The Formal Methods Model
- ➢ Aspect-Oriented Software Development

i. PERSONAL AND TEAM PROCESS MODELS
- ➢ Personal Software Process (PSP)
- ➢ Team Software Process (TSP)

AGILE PROCESS MODELS:

j. AGILE PROCESS
- ➢ Agility Principles
- ➢ The Politics of Agile Development
- ➢ Human Factors

k. EXTREME PROGRAMMING
- ➢ XP Values
- ➢ The XP Process
- ➢ Industrial XP
- ➢ The XP Debate

a. THE NATURE OF SOFTWARE

The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options, have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

➢ **Defining Software**

Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered.
But a more formal definition probably won't measurably improve your understanding.To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

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

    Although some similarities exist between software development and hardware 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 nonexistent (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.
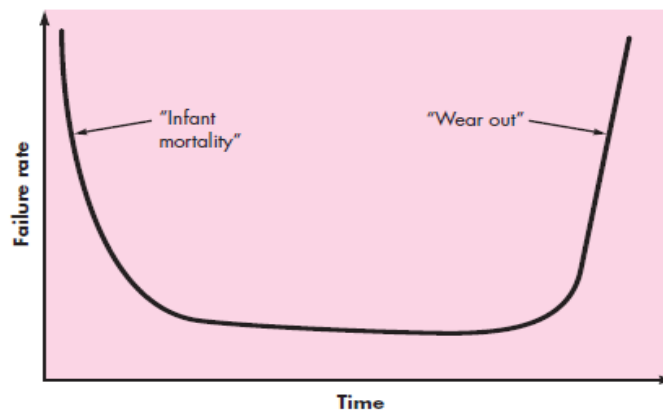
2. Software doesn't "wear out."

Figure 1: Failure curve for hardware

Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative

effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear— software doesn't wear out. But it does deteriorate!

This seeming contradiction can best be explained by considering the actual curve in Figure 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 .2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.
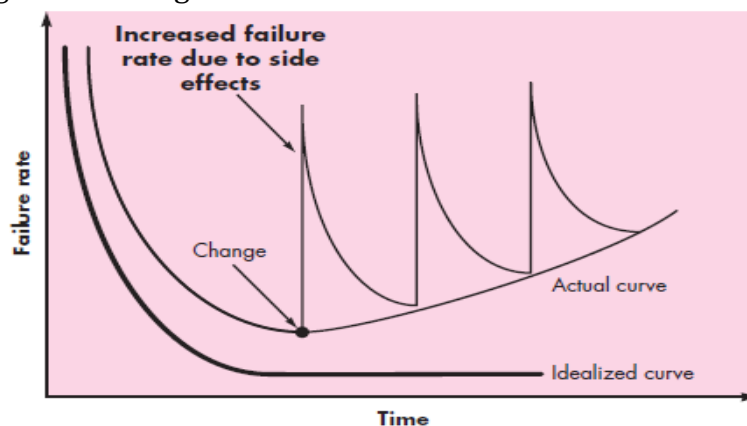

Figure 1.2: Failure curves for software

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. 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. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. 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.3 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**

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

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.

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).

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—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).

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).

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.

Artificial intelligence software—makes use of non numerical 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.

➢ Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases much older.

These older programs—often referred to as legacy software—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues describe legacy software in the following way:

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

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.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future.

b. THE SOFTWARE PROCESS

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

In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it. 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 stakeholders11 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.

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

c. SOFTWARE ENGINEERING PRACTICE

A generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—communication, planning, modeling, construction, and deployment— and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the following sections , you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.

➢ **The Essence of Practice**

In a classic book, *How to Solve It,* written before modern computers existed, George Polya outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing.* Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:
- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the solution. Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:
- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry out the plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.
- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.
- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements? It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

➢ **General Principles**

David Hooker has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:

The First Principle: *The Reason It All Exists* A software system exists for one reason: *to provide value to its users.* All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)* Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant

designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself.

Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume* Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future* A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start.

Never design yourself into a corner. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one. This could very possibly lead to the reuse of an entire system.

The Sixth Principle: *Plan Ahead for Reuse* Reuse saves time and effort. Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies.

However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer based systems would be eliminated.

d. SOFTWARE MYTHS

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it 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 (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

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

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

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

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

Reality: 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. Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.
Myth: *Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data 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. Many software professionals recognize the fallacy of the myths just described.
Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

e. A GENERIC PROCESS MODEL

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.
The software process is represented schematically in Figure 3. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.
A generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process. You should note that one important aspect of the software process has not yet been discussed. This aspect—called *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 4.

A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 4a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 4b). 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 4c). A *parallel process flow* (Figure 4d) 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).
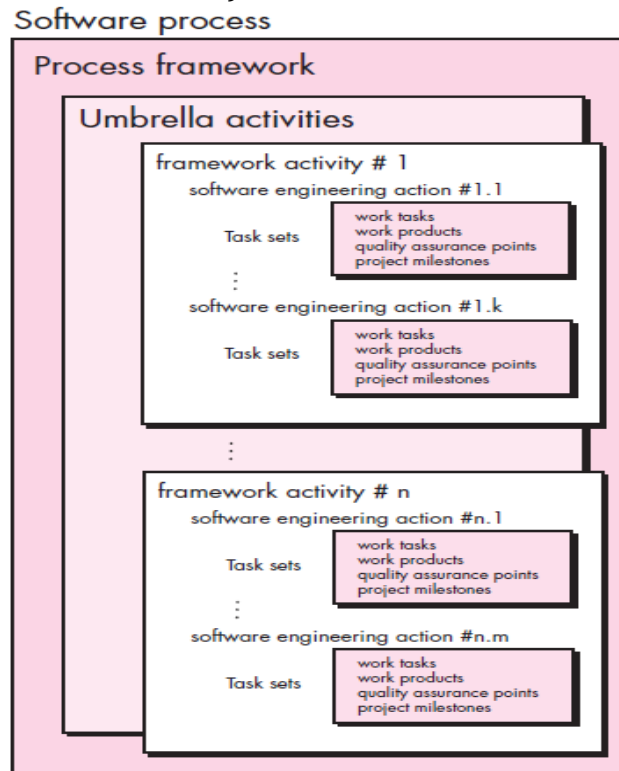

Figure 3: A software process framework

➢ Defining a Framework Activity

Although I have described five framework activities and provided a basic definition of each, a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*
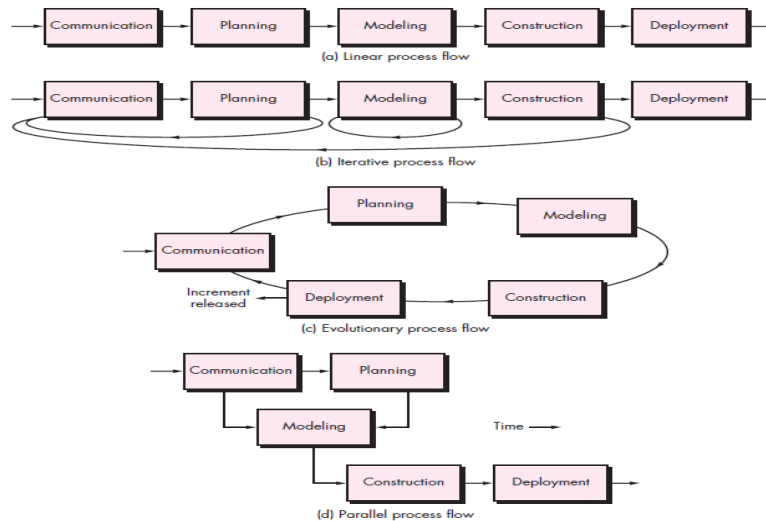
Figure 4: Process flow

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and take notes.
3. Organize notes into a brief written statement of requirements.
4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

➢ **Identifying a Task Set**

Referring again to Figure 3, each software engineering action (e.g., *elicitation,* an action associated with the communication activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering can be adapted to the specific needs of the software project and the characteristics of the project team.

➢ **Process Patterns**

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. 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. Stated in more general terms, 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.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler 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., **TechnicalReviews**).

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. Ambler suggests 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 **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** 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., **RequirementsGathering** 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 **SpiralModel** 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.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern). The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

f. PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice. In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

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.

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.

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.

g. PRESCRIPTIVE PROCESS MODELS

➢ 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 approach6 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
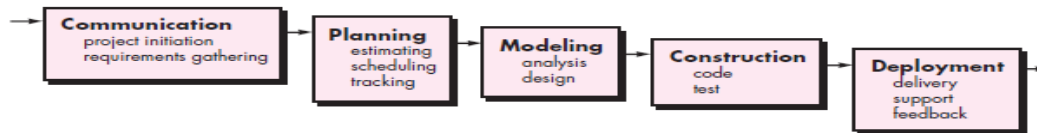
A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 2.4, the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left 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. 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.

FIGURE 2.4
The V-model
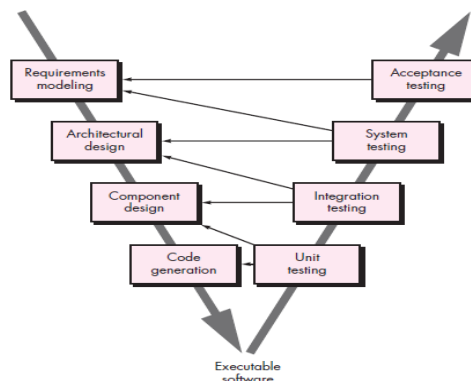
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

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

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.
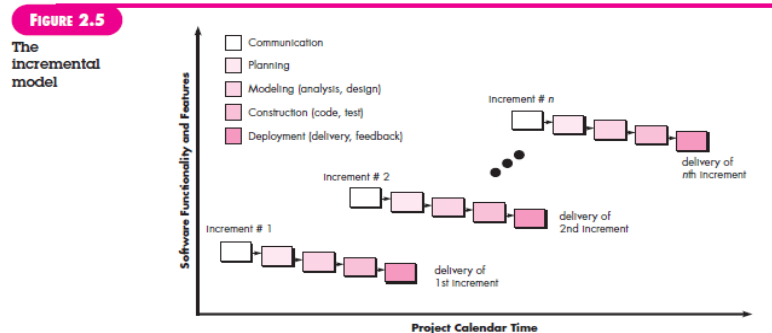
➢ Incremental Process Models

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. 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.

Figure 2.5
The incremental model

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.

➢ 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.

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.

FIGURE 2.6
The prototyping paradigm
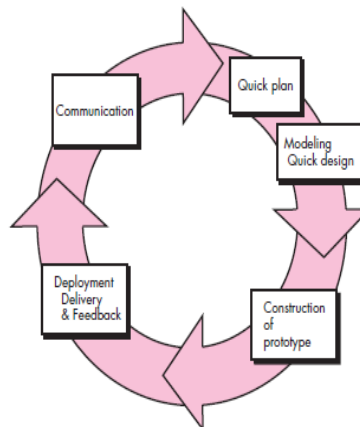
Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

But what do you do with the prototype when it has served the purpose described earlier? Brooks provides one answer:

In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. 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:

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.

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

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.

FIGURE 2.7
A typical spiral model

RCPIT SHIRPUR
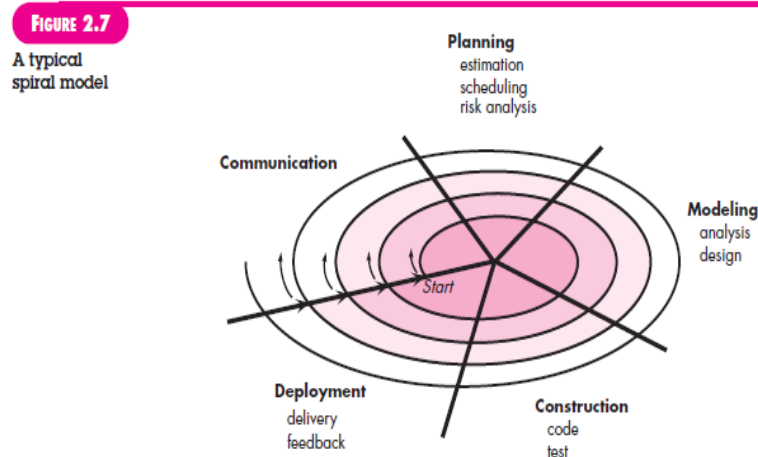
A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities. Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk 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. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables 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.

➢ 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. 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
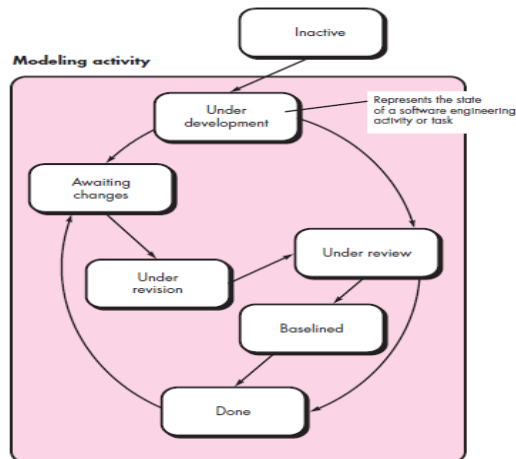
For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **inactive** state while initial communication was completed, now makes a transition into the **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.

h. SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

➤ 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, 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 (implemented using an evolutionary approach):

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. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

➤ 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.

These concerns not withstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

➤ Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object oriented classes) and then constructed within the context of a system architecture.

As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect

functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns. 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".

Grundy provides further discussion of aspects in the context of what he calls *aspect-oriented component engineering* (AOCE):

> AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

i. PERSONAL AND TEAM PROCESS MODELS

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey and argues that it is possible to create a "personal software process" and/or a "team software process." Both require hard work, training, and coordination, but both are achievable.

➢ Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey suggests that in order to change an ineffective personal process, an individual

must move through four phases, each requiring training and careful instrumentation. The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce. PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers, the resulting improvement in software engineering productivity and software quality are significant. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people. Can PSP be used as an effective software process at a personal level? The answer is an unequivocal "yes." But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

➢ Team Software Process (TSP)

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a *Team Software Process* (TSP). The goal of TSP is to build a "selfdirected" project team that organizes itself to produce high-quality software. Humphrey defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: **project launch, high-level design, implementation, integration and test,** and **postmortem.** Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

TSP recognizes that the best software teams are self-directed. Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering. Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

AGILE PROCESS MODELS:

j. AGILE PROCESS

➢ What Is An Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability?* The answer, as I have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable.*

➢ Agility Principles

The Agile Alliance defines agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained.

➤ The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim High smith (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making. No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models, each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

➤ Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team,* not the other way around. If members of the software

team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

Self-organization. In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."

k. EXTREME PROGRAMMING (XP)

In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed. IXP refines XP and targets the agile process specifically for use within large organizations.

➢ XP Values

Beck defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for

specific XP activities, actions, and tasks. In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium. To achieve *simplicity,* XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored* at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy, the software (via test results) provides the agile team with feedback. XP makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback.

Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact. Beck argues that strict adherence to certain XP practices demands *courage.* A better word might be *discipline.* For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates *respect* among it members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

➢ The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.2 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

FIGURE 3.2
The Extreme Programming process
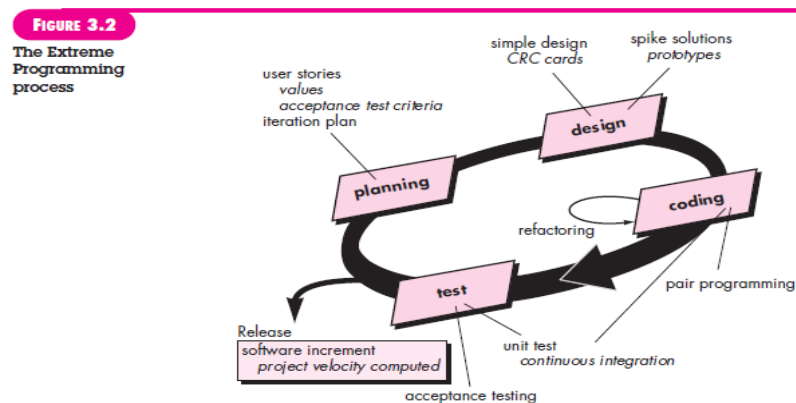
Planning. The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* is written by the customer and is placed on an index card.

The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an over commitment has been made for all stories across the entire development project. If an over commitment occurs, the content of releases is modified or end delivery dates are changed.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.

XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes7 that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one. The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution*, the design prototype is implemented and evaluated.

The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem. In the preceding section, we noted that XP encourages *refactoring*—a construction technique that is also a method for design optimization. Fowler describes refactoring in the following manner: Because XP design uses virtually no notation and produces few, if any, work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design". It should

be noted, however, that the effort required for refactoring can grow dramatically as the size of an application grows.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

Coding. After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment). Once the unit test9 has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility.

This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

Testing. I have already noted that the creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy whenever code is modified (which is often, given the XP refactoring philosophy). As the individual unit tests are organized into a "universal testing suite", integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

XP *acceptance tests*, also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

➢ Industrial XP

Joshua Kerievsky describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

Readiness assessment. Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the

organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.

Project community. Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project". In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

Test-driven management. An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.

Continuous learning. Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

In addition to the six new practices discussed, IXP modifies a number of existing XP practices. *Story-driven development* (SDD) insists that stories for acceptance tests be written before a single line of code is generated. *Domain-driven design* (DDD) is an improvement on the "system metaphor" concept used in XP. DDD suggests the evolutionary creation of a domain model that "accurately represents how domain experts think about their subject". *Pairing* extends the XP pair programming concept to include managers and other stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development. *Iterative usability* discourages front-loaded interface design in favor of usability design that evolves as software increments are delivered and users' interaction with the software is studied.

IXP makes smaller modifications to other XP practices and redefines certain roles and responsibilities to make them more amenable to significant projects for large organizations. For further discussion of IXP, visit **http://industrialxp.org**.

➢ **The XP Debate**

All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP, Stephens and Rosenberg argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic. The authors suggest that the codependent nature of XP practices are both its strength and its weakness. Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process. Proponents counter that XP is continuously evolving and that many of the

issues raised by critics have been addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are:

• *Requirements volatility*. Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.

- *Conflicting customer needs*. Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.

- *Requirements are expressed informally*. User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.

- *Lack of formal design*. XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

You should note that every software process has flaws and that many software organizations have used XP successfully. The key is to recognize where a process may have weaknesses and to adapt it to the specific needs of your organization.

Question for Credit Test I

Note: Each Question carry 10 marks.

1. What is Software engineering? Explain Characteristics of software.
2. Explain Spiral model with its advantages and disadvantages.
3. Given in details generic process model.
4. Explain myths and realities in content with manager and practitioners.
5. Differentiate between prototyping model and spiral model.
6. Differentiate between waterfall model and incremental model.
7. What is mean by agile development? Explain agile principles.
8. Explain XP values.
9. Explain in details XP process.
10. Explain in brief.
    a. Component-Based Development
    b. The Formal Methods Model
    c. Aspect-Oriented Software Development
11. Write a short notes on,
    a. PSP
    b. TSP
12. Enlist various software application domains. What are the general principles of Software engineering?