UNIT – 1

Unit I:
Software and Software Engineering: The Nature of Software, The Unique Nature of WebApps, Software Engineering, The Software Process, Software Engineering Practice, Software Myths
Process Models: A Generic Process Model, Process Assessment and Improvement, Prescriptive Process Models, Specialized Process Models, The Unified Process, Personal and Team Process Models, Process Technology, Product and Process.

Agile Development: Agility, Agility and the Cost of Change, Agile Process, Extreme Programming, Other Agile Process Models

1.0 SOFTWARE AND SOFTWARE ENGINEERING

What is it? Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media. Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high quality computer software.

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

Why is it important? Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

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

1.1 THE NATURE OF SOFTWARE

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

As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—information. It transforms personal data so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks, and provides the means for acquiring information in all of its forms.

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.

1.1.1 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.

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. Software projects cannot be managed as if they were manufacturing projects.

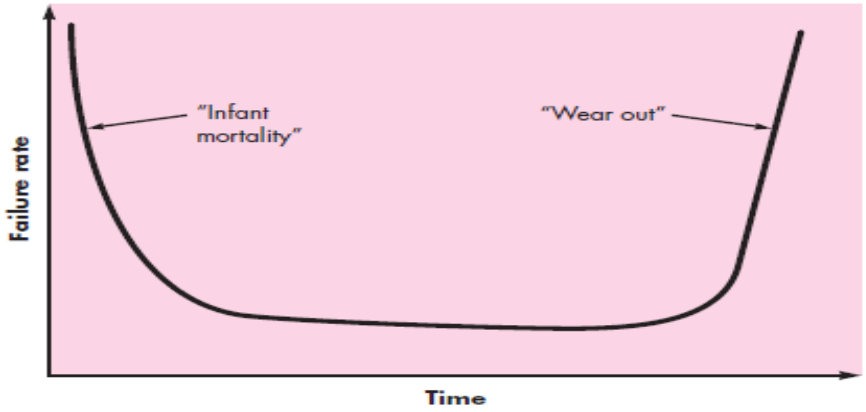

Fig 1.1: Failure Curve for hardware

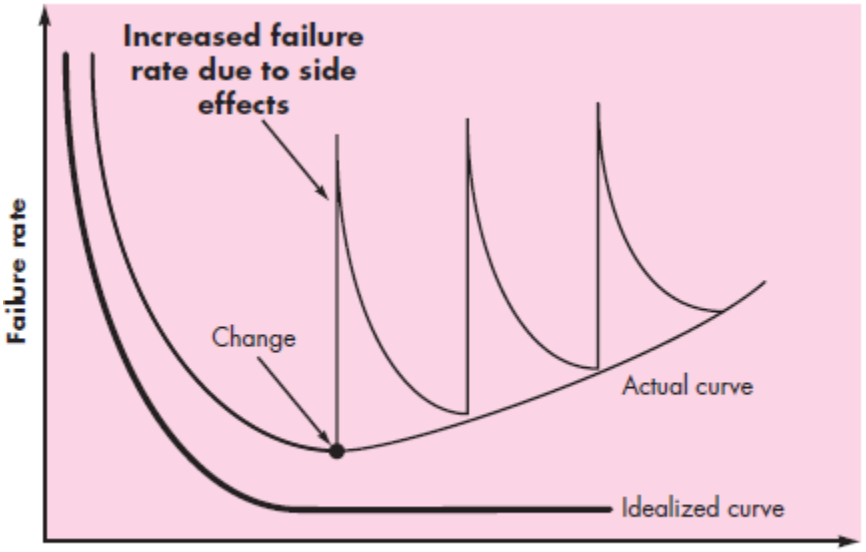

Fig 1.2: Failure curves for software

2. Software doesn't "wear out.": Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "*bathtub curve*," indicates that hardware exhibits relatively high failure rates early in its life; 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 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized

curve is a gross oversimplification of actual failure models for software. However, the implication is clear—*software doesn't wear out. But it does deteriorate*.

3. Although the industry is moving toward component-based construction, most software continues to be custom built: As an engineering discipline evolves, a collection of standard design components is created. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.

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.

1.1.2 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.

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.

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

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.

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 or address mass consumer markets.

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.

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

New Challenges

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

Net sourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications so that many people can contribute to its development.

1.1.3 Legacy Software: These *older programs*—often referred to as legacy software. Legacy software systems . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. Legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—poor quality. Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results.

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.

1.2 THE UNIQUE NATURE OF WEBAPPS

The following attributes are encountered in the vast majority of WebApps.

Network intensiveness. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
Concurrency. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
Performance. If a WebApp user must wait too long, he or she may decide to go elsewhere.
Availability. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.
Data driven. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment
Content sensitive. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
Continuous evolution. Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.
Immediacy. Although immediacy—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.
Security. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.
Aesthetics. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

1.3 SOFTWARE ENGINEERING

Software engineering encompasses a process, methods for managing and engineering software, and tools.

In order to build software that is ready to meet the challenges of the twenty-first century, few simple realities are:

• Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application has grown dramatically.
• The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. It follows that design becomes a pivotal activity.
• Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures.
• As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. It follows that software should be maintainable.

The IEEE has developed a more comprehensive definition when it states:
Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

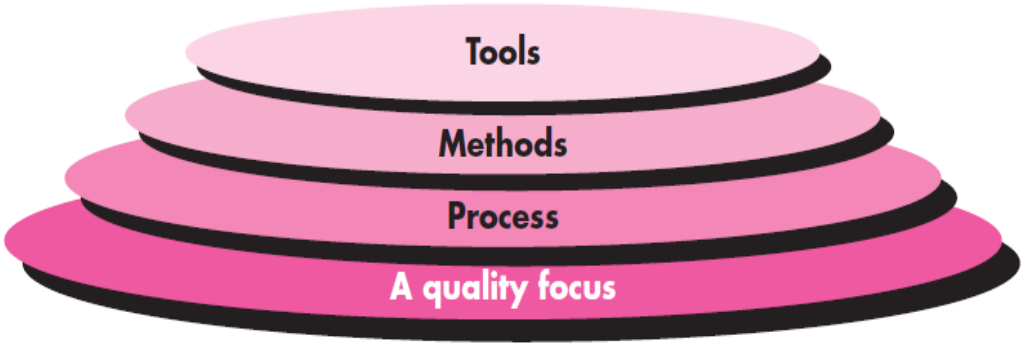

Fig 1.3: Software Engineering layers

Layered Technology: Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an

organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock of software engineering is a ***quality focus***.

The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology.

The *software* ***process*** forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering ***methods*** provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering ***tools*** provide automated or semi automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

1.4 The software Process

A process defines who is doing what when and how to reach a certain goal.
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 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** encompasses a set of tasks that produce a major work product. A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

A process is not a rigid rather it is an adaptable approach to 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. The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. The planning activity creates a "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. You create a "sketch" of the thing so that you'll understand the big picture. 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 and the testing that is required to uncover errors in the code.

Deployment. The software 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.

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.

Software engineering process framework activities are complemented by a number of ***umbrella activities***. In general, umbrella activities are applied throughout a software project.

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.

1.5 Software Engineering Practice

Generic framework activities—communication, planning, modeling, construction, and deployment—and umbrella activities establish a skeleton architecture for software engineering work. Practices are as follows
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 worth spending a little time to understand, 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 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 sub problems be defined? If so, are solutions readily apparent for the sub problems?

• Can you represent a solution in a manner that leads to effective implementation? Can a design 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 sub problems be defined? If so, are solutions readily apparent for the sub problems?
• Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

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?

1.5.2 General Principles: David Hooker has proposed seven principles that focus on software engineering practice as a whole. They are as following.

The First Principle: The Reason It All Exists
A software system exists for one reason: to provide value to its users. 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!)
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. Indeed, the more elegant designs are usually the more simple ones. 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. 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

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. 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

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.

1.6 Software myths

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.

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

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

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

Reality: Software development is not a mechanistic process like manufacturing. 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 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 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 the cost impact is relatively small.16 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 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 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.

PROCESS MODELS

What is it? When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a "software process."
Who does it? Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.
Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.
What are the steps? At a detailed level, the process that you adopt depends on the software that you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a website.
What is the work product? From the point of view of a software engineer, the work products are the programs, documents, and data that are produced as a consequence of the activities and tasks defined by the process.

1.7 A GENERIC PROCESS MODEL

The software process is represented schematically. 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.

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.

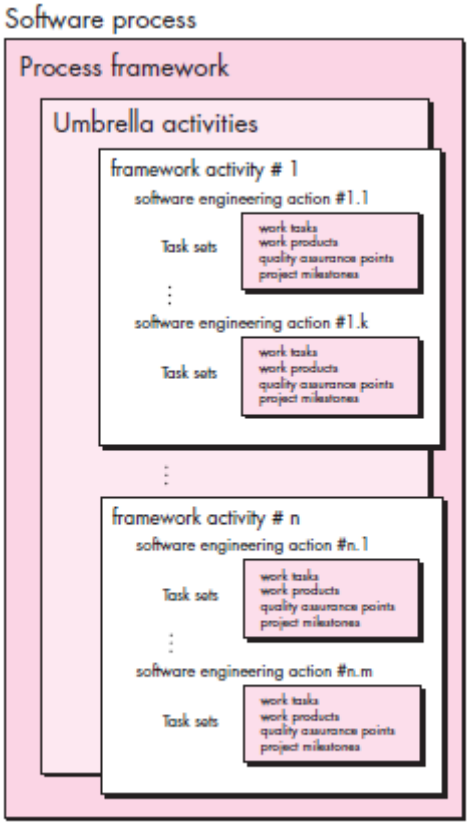

Fig: 2.1 A software process Framework

Process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2.

A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An iterative process flow repeats one or more of the activities before proceeding to the next (Figure 2.2b). An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c). A parallel process flow (Figure 2.2d) executes one or more activities in parallel with other activities.

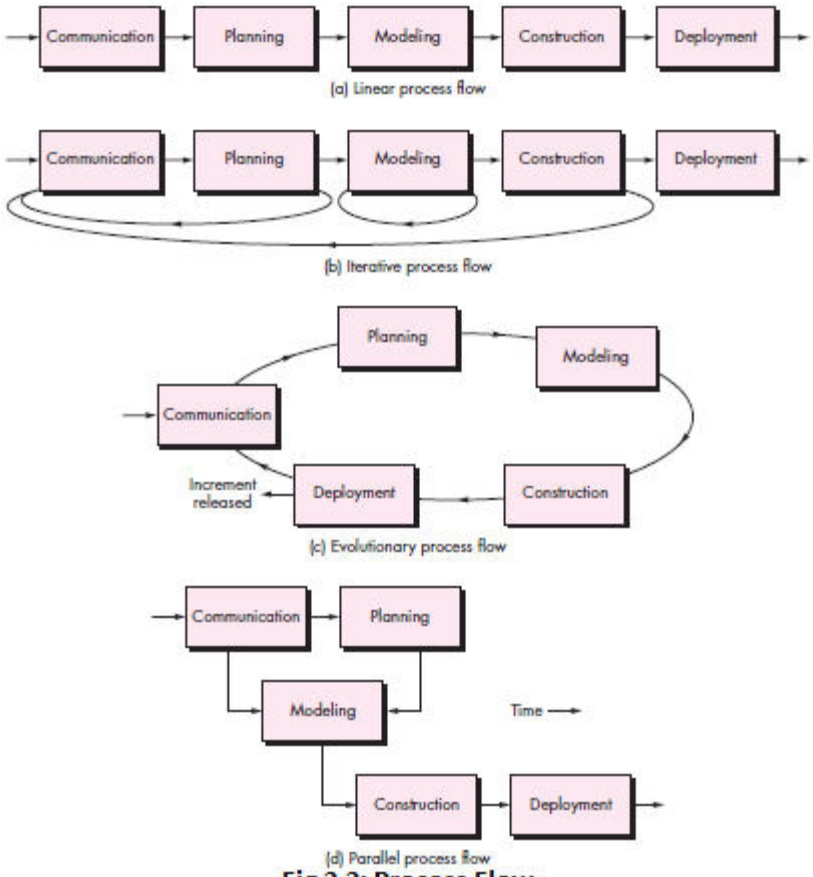

Fig 2.2: Process Flow

1.7.1 Defining a Framework Activity: A software team would need significantly more information before it could properly execute any one of these five activities (Communication, Planning, Modeling, Construction, Deployment) as part of the software process. 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 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.

1.7.2 Identifying a Task Set: Each software engineering action 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.

1.7.3 Process Patterns

A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template. 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

Type. The pattern type is specified. They are three types:

1. Stage pattern—defines a problem associated with a framework activity for the process.
2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
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.

Initial context. Describes the conditions under which the pattern applies.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern.

Related Patterns. Provide a list of all process patterns that are directly related to this one.

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable.

1.8 PROCESS ASSESSMENT AND IMPROVEMENT

A number of different approaches to software process assessment and improvement have been proposed.

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.

1.9 PRESCRIPTIVE PROCESS MODELS

Prescriptive process models define a prescribed set of process elements and a predictable process work flow. Prescriptive process models were originally proposed to bring order to the chaos of software development.

All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity in a different manner.

1.9.1 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. 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).

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

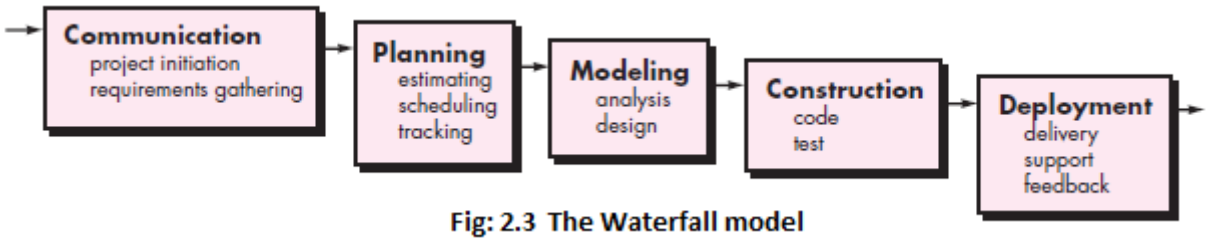

Fig: 2.3 The Waterfall model

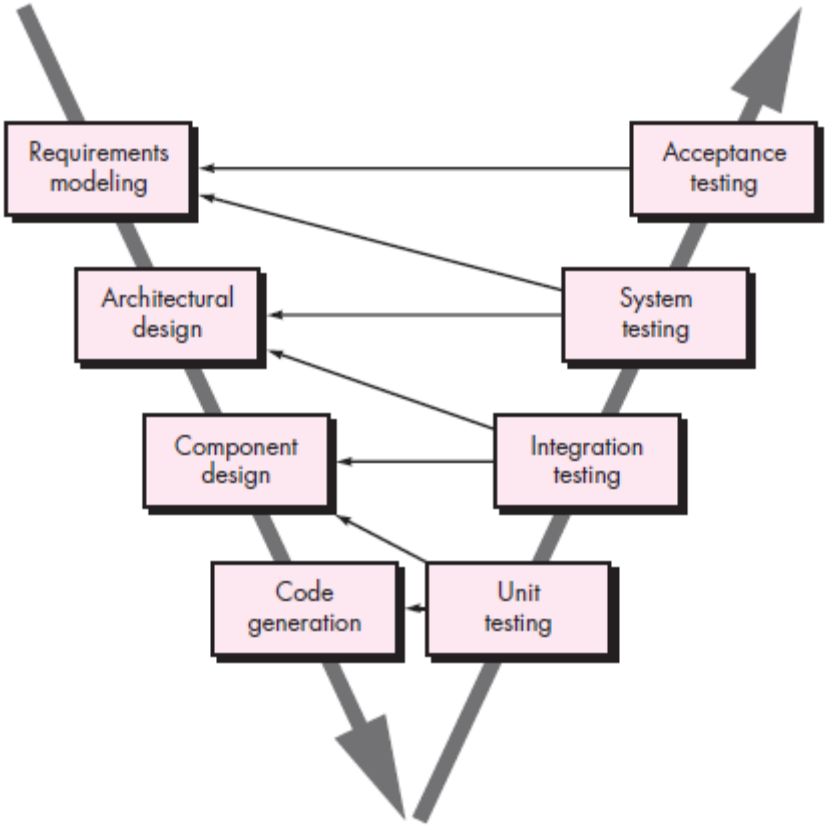

Fig 2.4 V - Model

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.7 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, 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.
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.

1.9.2 Incremental Process Models: *The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.*

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.

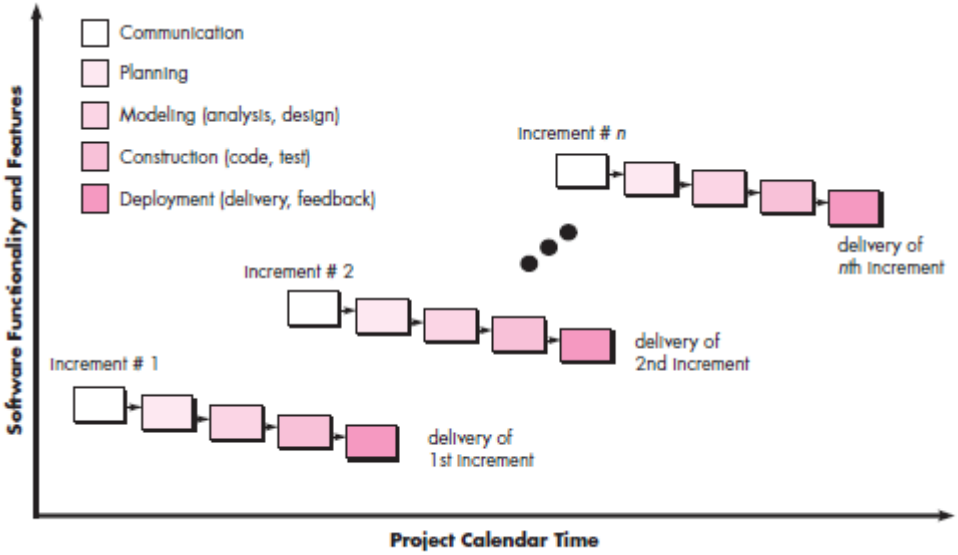

Fig 2.5: The incremental Model

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.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the

delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.
The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user. Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

1.9.3 Evolutionary Process Models: Evolutionary process models produce an increasingly more complete version of the software with each iteration.

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure.
.
Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. The two common evolutionary process models are presented here.

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.

Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

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

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.

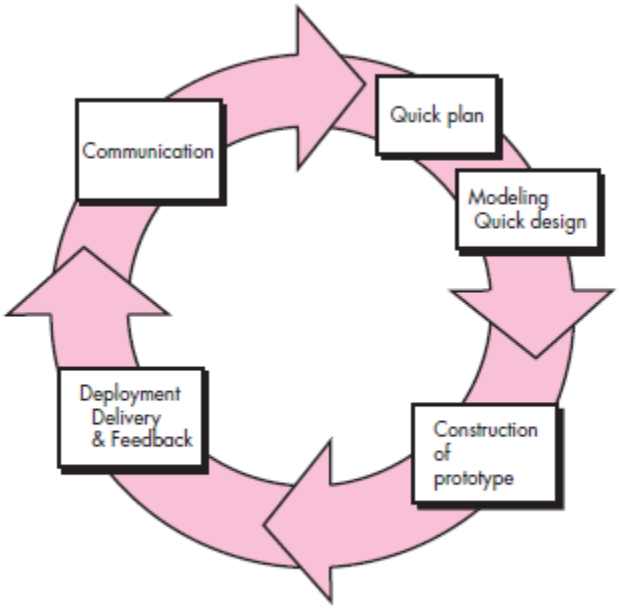

Fig 2.6: The Prototyping Paradigm

Prototyping can be problematic for the following reasons:
1. Stakeholders see what appears to be a working version of the software, unaware that there is pending work.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly.

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. Boehm describes the model in the following manner:

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

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7.

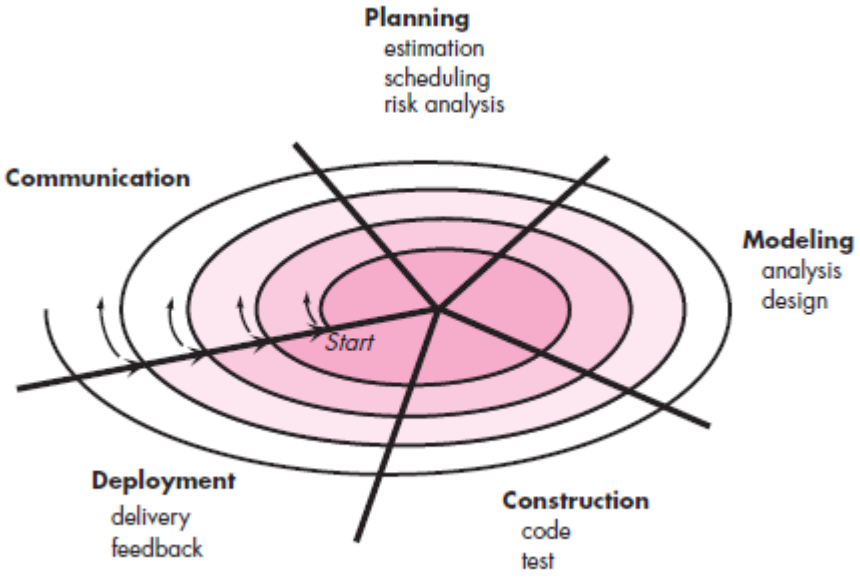

Fig 2.7: A typical spiral model

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. 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.

1.9.4 Concurrent Models: The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models. 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.

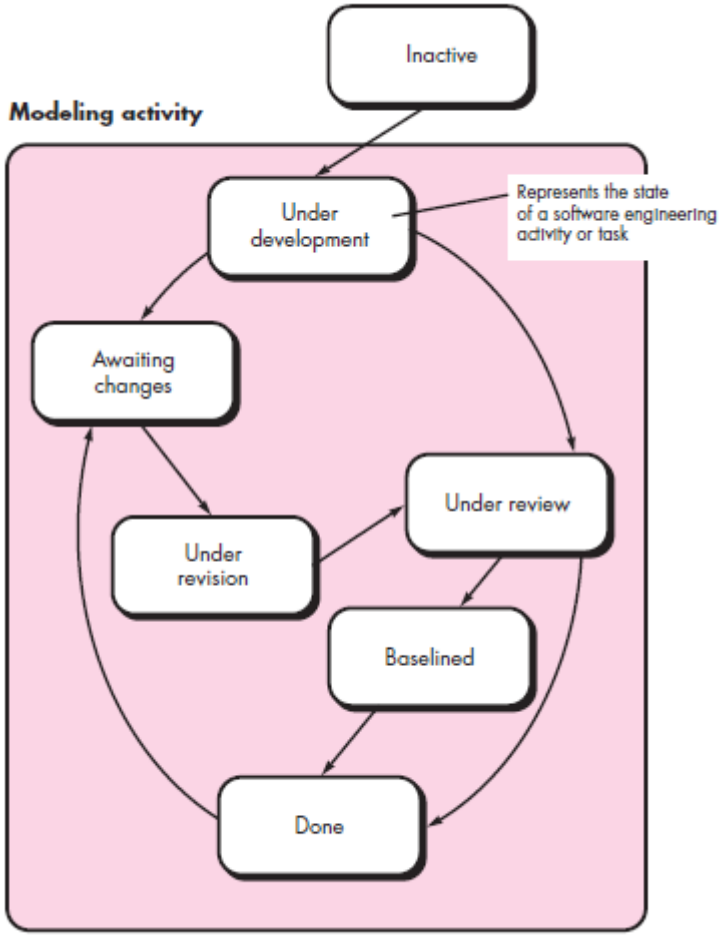

Fig 2.8: One element of the concurrent process model

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, 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 odeling is applicable to all types of software development and provides an accurate picture of the current state of a project.

1.10 SPECIALIZED PROCESS MODELS

These models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

1.10.1 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

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.

1.10.2 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 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.

1.10.3 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), 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.

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.

1.11 The UNIFIED PROCESS

1.11.1 Introduction: The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but haracterize them in a way that implements many of the best principles of agile software development.

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the

important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse”.

1.11.2 Phases of the Unified Process: The Unified Process is with five basic framework activities depicted in figure 2.9. It depicts the “phases” of the UP and relates them to the generic activities.

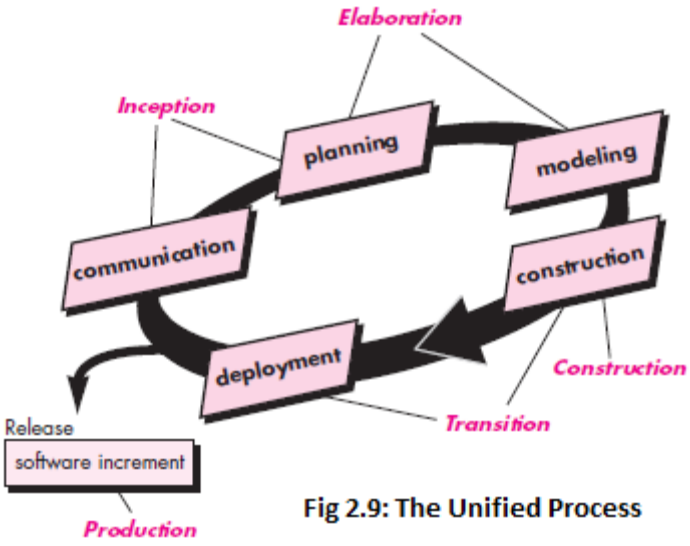

Fig 2.9: The Unified Process

The inception phase of the UP encompasses both customer communication and planning activities. The elaboration phase encompasses the communication and modeling activities of the generic process model. The construction phase of the UP is identical to the construction activity defined for the generic software process. The transition phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated. A software engineering workflow is distributed across all UP phases.

1.12 PERSONAL AND TEAM PROCESS MODELS

1.12.1 Introduction: If a software process model has been developed at a corporate or organizational level, it can be effective only if it is agreeable 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.

1.12.2 Personal Software Process (PSP): Every developer uses some process to build computer software. 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 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, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP emphasizes the need to record and analyze the types of errors you make, so that you can develop strategies to eliminate them.

1.12.3 Team Software Process (TSP): Because many industry-grade software projects are addressed by a team of practitioners, The goal of TSP is to build a “self-directed” 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 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.*

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. 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.

1.13 PROCESS TECHNOLOGY

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

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

1.14 PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer. All of human activity may be a **Process**. And the outcome of the process is **Product.**

Software is more than just a program code. Software engineering is an engineering branch associated with development of **software product** using well-defined scientific principles, methods and procedures which is **process**. The outcome of software engineering is an efficient and reliable software product.

About every ten years give or take five, the software community redefines “the problem” by shifting its focus from product issues to process issues. Thus, we have embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute’s Software Development Capability Maturity Model (process)

Fig 2.10: : Software Product with Process

People: The primary element of any project is the people. People gather requirements, people interview users (people), people design software, and people write software for people. No people -- no software.

Process: Process is how we go from the beginning to the end of a project. All projects use a process. Many project managers, however, do not choose a process based on the people and product at hand. They simply use the same process they've always used or misused. Let's focus on two points regarding process: (1) process improvement and (2) using the right process for the people and product at hand.

Product: The product is the result of a project. The desired product satisfies the customers and keeps them coming back for more. Sometimes, however, the actual product is something less.

The product pays the bills and ultimately allows people to work together in a process and build software. Always keep the product in focus. Our current emphasis on process sometimes causes us to forget the product. This results in a poor product, no money, no more business, and no more need for people and process.

AGILITY

1.15 WHAT IS AGILITY?

An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products.

Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

1.16 AGILITY AND THE COST OF CHANGE

The conventional wisdom in software development is that the cost of change increases nonlinearly as a project progresses (Figure: solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a

project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project.

The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.

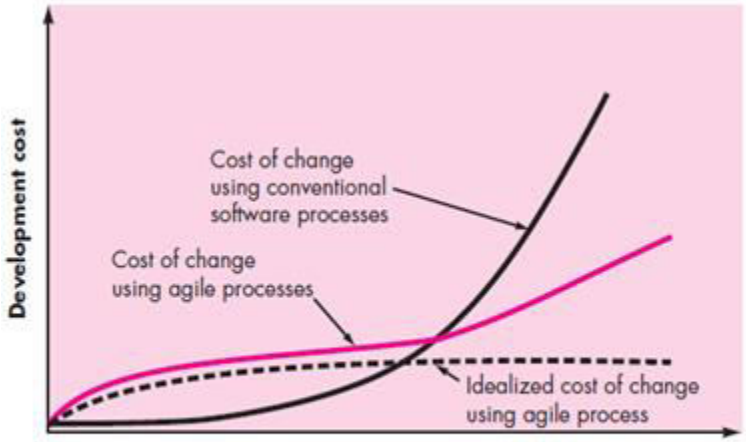

Fig 2.11: Change Costs as a function of time in development

1.17 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 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. An agile process, therefore, must be adaptable. But continual adaptation without forward progress accomplishes little. Therefore, an agile software

process must adapt incrementally. To accomplish incremental adaptation, an agile team requires customer feedback. An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an incremental development strategy should be instituted. Software increments must be delivered in short time periods so that adaptation keeps pace with change. This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

1.17.1 Agility Principles: The Agile Alliance defines 12 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.

1.17.2 The Politics of Agile Development: There is considerable debate about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. 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" 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.

1.17.3 Human Factors: Proponents of agile software development take great pains to emphasize the importance of "people factors." 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 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 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 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.

1.18 EXTREME PROGRAMMING (XP)

Extreme Programming (XP), the most widely used approach to agile software development.

1.18.1 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.

Communication: 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.

Simplicity: 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: It 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 provides the agile team with feedback 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.

Courage: Beck argues that strict adherence to certain XP practices demands courage. A better word might be discipline. 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.

Respect: By following each of these values, the agile team inculcates respect among its 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.

1.18.2 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 .

Planning: The planning activity 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.

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. If a difficult design problem is encountered as part of the design of a story, XP

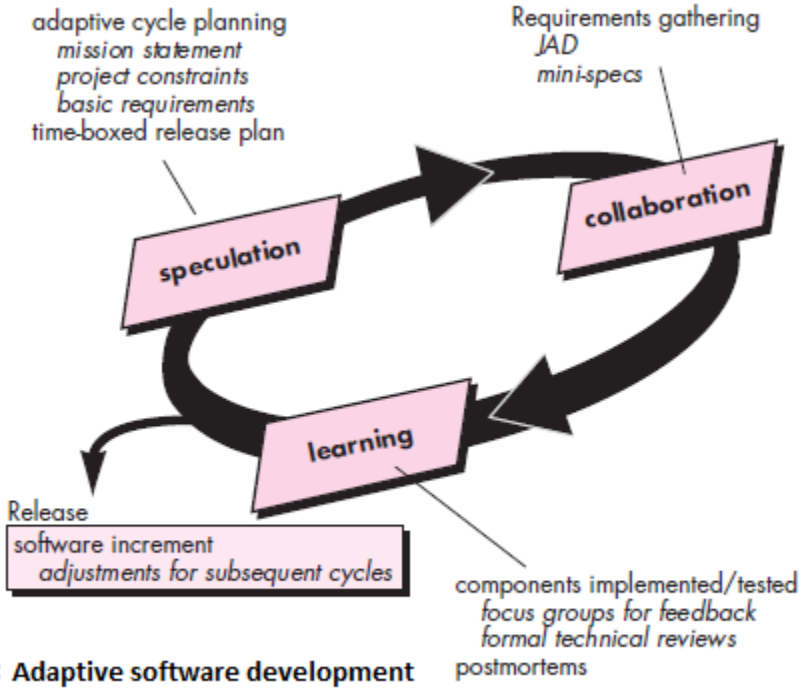

Fig 2.12: **Adaptive software development**

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. 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. Once the unit test 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 is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. Problem solving (two heads are often better than one) and real-time quality assurance.

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. This encourages a regression testing strategy whenever code is modified. 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. 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.

1.18.3 Industrial XP: IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. 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 to learn new methods and techniques that can lead to a higher quality product.

1.18.4 The XP Debate: 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.
• ***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.

1.19 OTHER AGILE PROCESS MODELS

The most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. Among the most common are:
• Adaptive Software Development (ASD)
• Scrum
• Dynamic Systems Development Method (DSDM)
• Crystal
• Feature Drive Development (FDD)
• Lean Software Development (LSD)
• Agile Modeling (AM)
• Agile Unified Process (AUP)

1.19.1 Adaptive Software Development (ASD): Adaptive Software Development (ASD) has been proposed by *Jim Highsmith* as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

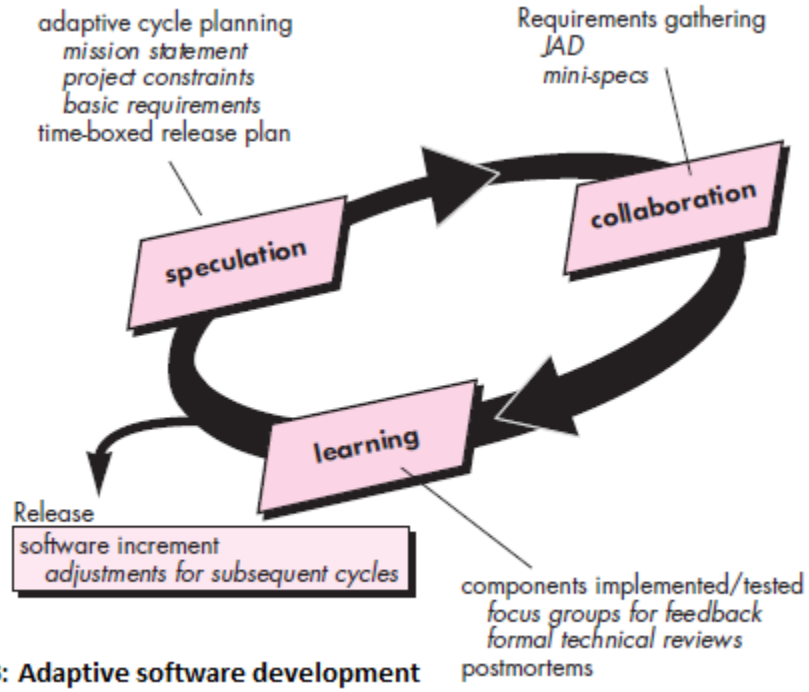

Fig 2.13: Adaptive software development

1.19.2 Scrum: Scrum is an agile software development method that was conceived by Jeff Sutherland in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: ***requirements, analysis, design, evolution, and delivery***. Within each framework activity, work tasks occur within a process pattern called a sprint. The work conducted within a sprint is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure

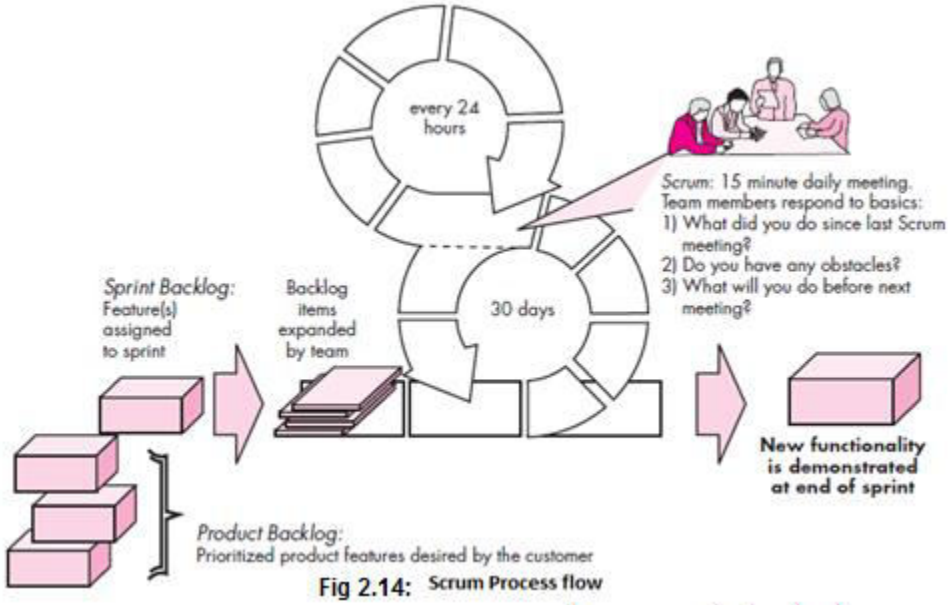

Fig 2.14: Scrum Process flow

Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box.

Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members:

• What did you do since the last team meeting?
• What obstacles are you encountering?
• What do you plan to accomplish by the next team meeting?

A team leader, called a ***Scrum master***, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

1.19.3 Dynamic Systems Development Method (DSDM): The Dynamic Systems Development Method (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment". The DSDM philosophy is borrowed from a modified version of the ***Pareto principle***—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

DSDM consortium has defined an agile process model, called the DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.
Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.
Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.
Implementation—places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

1.19.4 Crystal: To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for project and environment.

1.19.5 Feature Driven Development (FDD): Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means. FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns

In the context of FDD, a feature “is a client-valued function that can be implemented in two weeks or less”. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.

- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

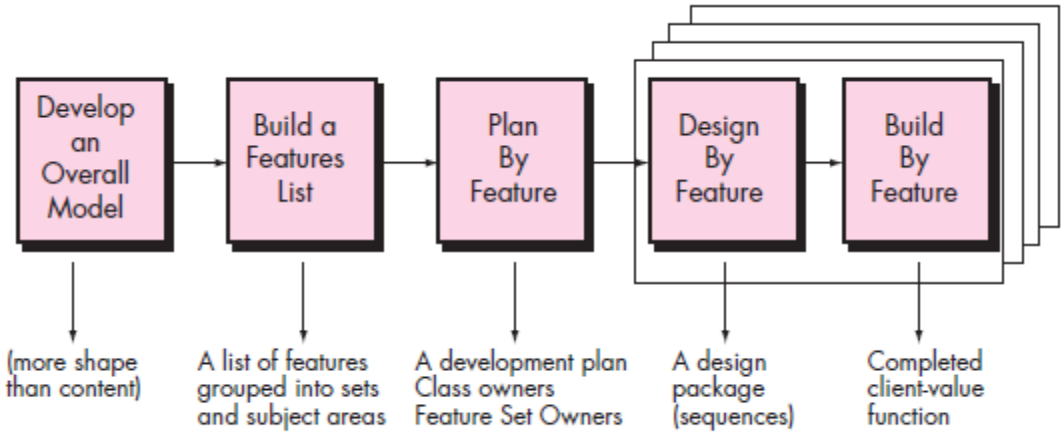

Fig 2.14: **Feature Driven Development**

1.19.6 Lean Software Development (LSD): The lean principles that inspire the LSD process can be summarized as eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole. Each of these principles can be adapted to the software process. For example, eliminate waste within the context of an agile software project can be interpreted to mean (1) adding no extraneous features or functions, (2) assessing the cost and schedule impact of any newly requested requirement, (3) removing any superfluous process steps, (4) establishing mechanisms to improve the way team members find information, (5) ensuring the testing finds as many errors as possible, (6) reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and (7) streamlining the manner in which information is transmitted to all stakeholders involved in the process.

1.19.7 Agile Modeling (AM): There are many situations in which software engineers must build large, business critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and re factor. The team must also

have the humility to recognize that technologists do not have all the answers and that business expert and other stakeholders should be respected and embraced. Although AM suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are:

- Model with a purpose
- Use multiple models
- Travel light
- Content is more important than representation
- Know the models and the tools you use to create them
- Adapt locally

1.19.8 Agile Unified Process (AUP): The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities

• **Modeling**. UML representations of the business and problem domains are created. However, to stay agile, these models should be “just barely good enough” to allow the team to proceed.

• **Implementation**. Models are translated into source code.

• **Testing**. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.

• **Deployment**. Like the generic process activity. Deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.

• **Configuration and project management.** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

• **Environment management.** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.