UNIT - I

Introduction to Software Engineering:

Software- Definition: The IEEE (Institute of Electrical and Electronics Engineers) defines software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. Essentially, it's the engineering discipline focused on all aspects of software production, applying engineering principles to the creation of reliable and efficient software.

Here's a breakdown of the key aspects:

Systematic and Disciplined:

This emphasizes the structured and organized approach to software development, moving away from ad-hoc methods.

Quantifiable:

This highlights the need for measurable and objective assessments of the software's quality and performance.

Development, Operation, and Maintenance:

This covers the entire lifecycle of software, from initial creation to ongoing support and updates.

Application of Engineering to Software:

This stresses the use of engineering principles, methods, and tools to build software systems.

Software Engineering- Fundamentals to Software Engineering and the importance of Software Engineering.

Fundamentals of software engineering

- Software engineering is a layered technology: It starts with a quality focus, moves through process models, methods, and ultimately to tools used for development.

- A structured software process: This involves a systematic approach to development, operation, and maintenance of software, encompassing activities like communication, planning, modeling, construction, and deployment.
- Emphasis on key activities: Pressman's approach focuses on major activities in the generic software process like process modeling, requirements engineering, design engineering, quality management, and software project management.
- Design principles and concepts: These guide the development of high-quality software systems, including principles like modularity, separation of concerns, and information hiding.
- Quality assurance throughout the lifecycle: This involves activities like testing, verification, and validation at different stages to ensure the software meets requirements and is free from defects.
- Continuous improvement: Pressman emphasizes continuous process improvement based on quantitative feedback and the use of innovative techniques.

Importance of software engineering

- Producing high-quality software: Software engineering aims to create robust, reliable, and user-friendly software that meets the specified requirements and adheres to quality standards.
- Managing complexity: By providing a systematic and structured approach, software engineering helps manage the inherent complexity of software development, especially for large and intricate projects.
- Cost-effectiveness: By adopting a methodical approach, software engineering helps reduce development costs and minimize rework through effective planning, design, and testing.
- Meeting deadlines and budget constraints: A structured approach and proper management help keep software projects on schedule and within budget.
- Ensuring maintainability and adaptability: Well-engineered software is easier to maintain, adapt to new environments or technologies, and enhance with new features.

- Facilitating communication and collaboration: Pressman's approach promotes clear communication and collaboration among stakeholders throughout the development lifecycle.
- Adapting to new technologies and trends: While Pressman's foundational work provides a strong base, it's recognized that modern software engineering requires adapting to new technologies like cloud computing, micro services, and Agile methodologies.

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. Whether it resides within a mobile phone, a hand-held tablet, on the desktop, or within a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, dis- playing, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information net- works (e.g., the Internet), and provides the means for acquiring information in all of its forms. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone

programmer are the same questions that are asked when mod- ern computer-based systems are built:

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has led to the adoption of software engineering practice.

Software Application Domains:

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: a broad array of "number-crunching programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

Embedded software: resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software: designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer.

Web/Mobile applications: this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

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

The changing nature of software

Four broad categories of software are evolving to dominate the industry. And yet, these categories were in their infancy little more than a decade ago:

WebApps

In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. Web-based systems and applications5 (we refer to these collectively as WebApps) were born.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

A decade ago, WebApps "involve[d] a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

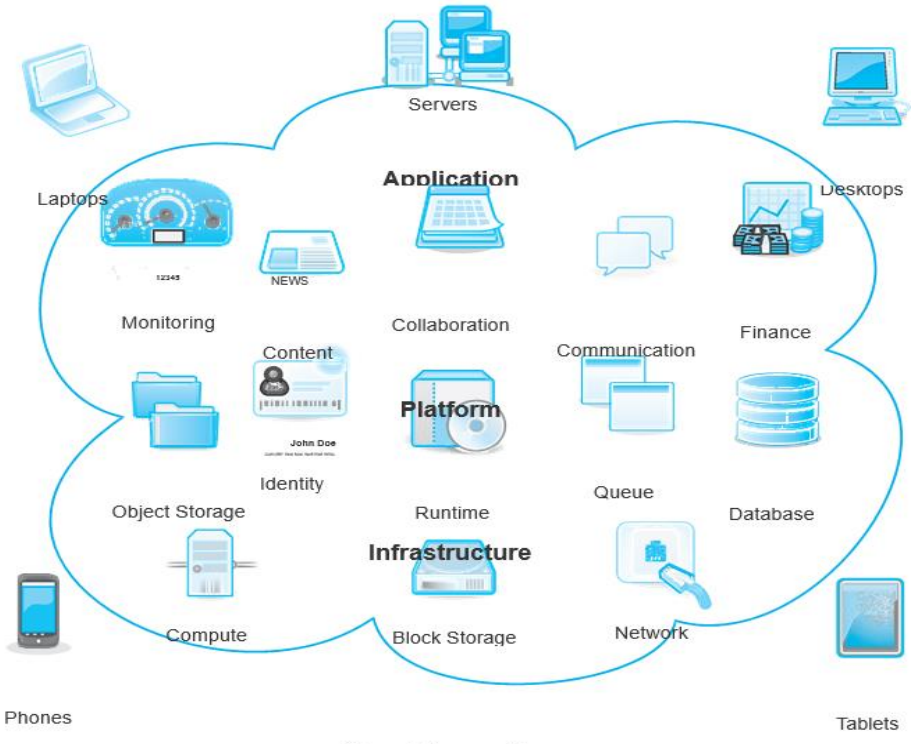
Mobile Applications

The term app has evolved to connote software that has been specifically de- signed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile). In most instances, mobile applications encompass a user interface that takes advantage of the unique interaction mechanisms provided by the mobile platform, interoperability with Web-based resources that provide access to a wide array of information that is relevant to the app, and local processing capabilities that collect, analyze, and format information in a manner that is best suited to the mobile platform. In addition, a mobile app provides persistent storage capabilities within the platform.

Cloud Computing

Cloud computing encompasses an infrastructure or "ecosystem" that enables any user, anywhere, to use a computing device to share computing resources.

Referring to the **figure**, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure. In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data structures. In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device.



Cloud Computing

The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services. The front-end includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed. The back-end includes servers and related computing resources, data storage systems (e.g., databases), server-resident applications, and administrative servers that use middleware to coordinate and monitor traffic by establishing a set of protocols for access to the cloud and its resident resources.

The cloud architecture can be segmented to provide access at a variety of different levels from full public access to private cloud architectures accessible only to those with authorization.

Product line Software:

The Software Engineering Institute defines a software product line as "a set of software-intensive systems that share a common, managed set of features sat- isfying the specific needs of a particular market segment or mission and that are developed from a common

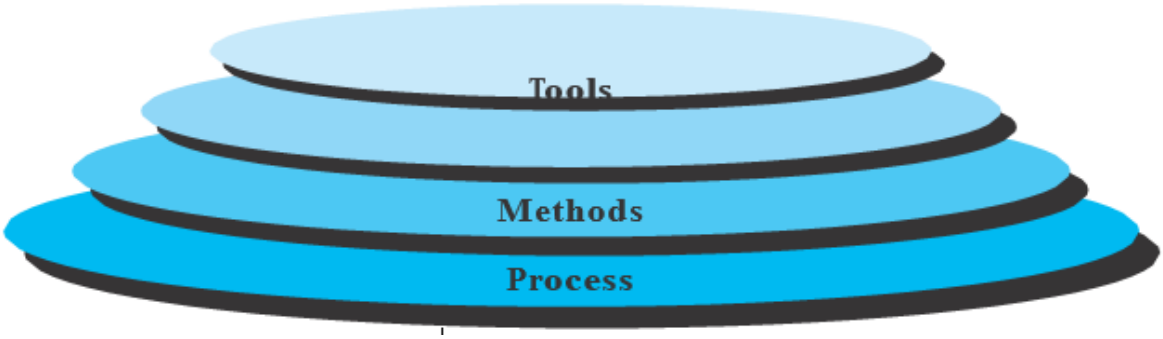
set of core assets in a prescribed way." [SEI13] The concept of a line of software products that are related in some way is not new. But the idea that a line of software products, all developed using the same underlying application and data architectures, and all implemented using a set of reusable software components that can be reused across the product line provides significant engineering leverage.

A software product line shares a set of assets that include requirements, architecture, design patterns, reusable components, test cases, and other software engineering work products. In essence, a software product line results in the development of many products that are engineered by capitalizing on the commonality among all the products within the product line.

Software engineering-A layered technology.

Software engineering is a layered technology. Referring to Figure, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies2 foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are



Applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modelling, 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.

Software Process:

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

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

A process framework:

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

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

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

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

creating models to better understand software requirements and the design that will achieve those requirements.

Construction. What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

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

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

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

There are some other activities we need to look into:

Umbrella Activities:

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

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

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assess 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—encompass the activities required to create work products such as models, documents, logs, forms, and lists.

Software engineering practice:

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

The Essence of Practice:

In the classic book, how to Solve It, written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

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

In the context of software engineering, these common sense steps lead to a series of essential questions.

Understand the problem: It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, Oh yeah, I understand, let's get on with solving this thing. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- Who has a stake in the solution to the problem? That is, who are the stakeholders?
- What are the unknowns? What data, functions, and features are required to properly solve the problem?
- Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?

Plan the solution: Now you understand the problem (or so you think), and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can 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?

Carry out the plan: The design you've created serves as a road map for the sys- tem you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- Does the solution conform to the plan? Is source code traceable to the de- sign model?
- Is each component part of the solution provably correct? Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm

Examine the result: You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

  It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

Software development myths:

Software development myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable

statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

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

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

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

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

Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "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 out- sources software projects.

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

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

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

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

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.8 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 60 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

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

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

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

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

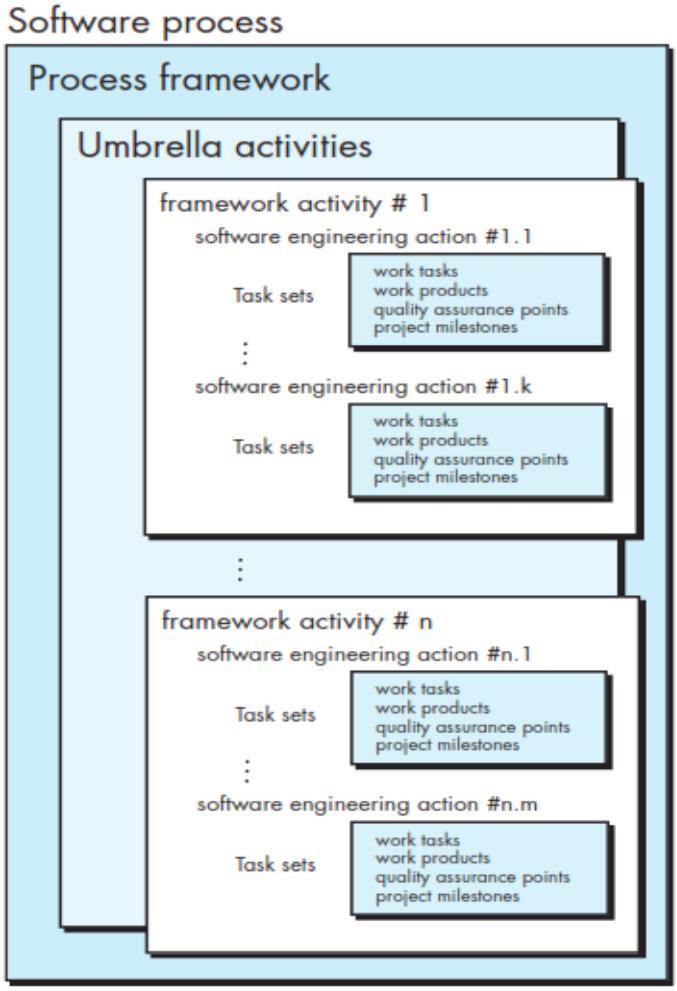
Today, most software professionals recognize the fallacy of the myths just de- scribed. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

Process models:

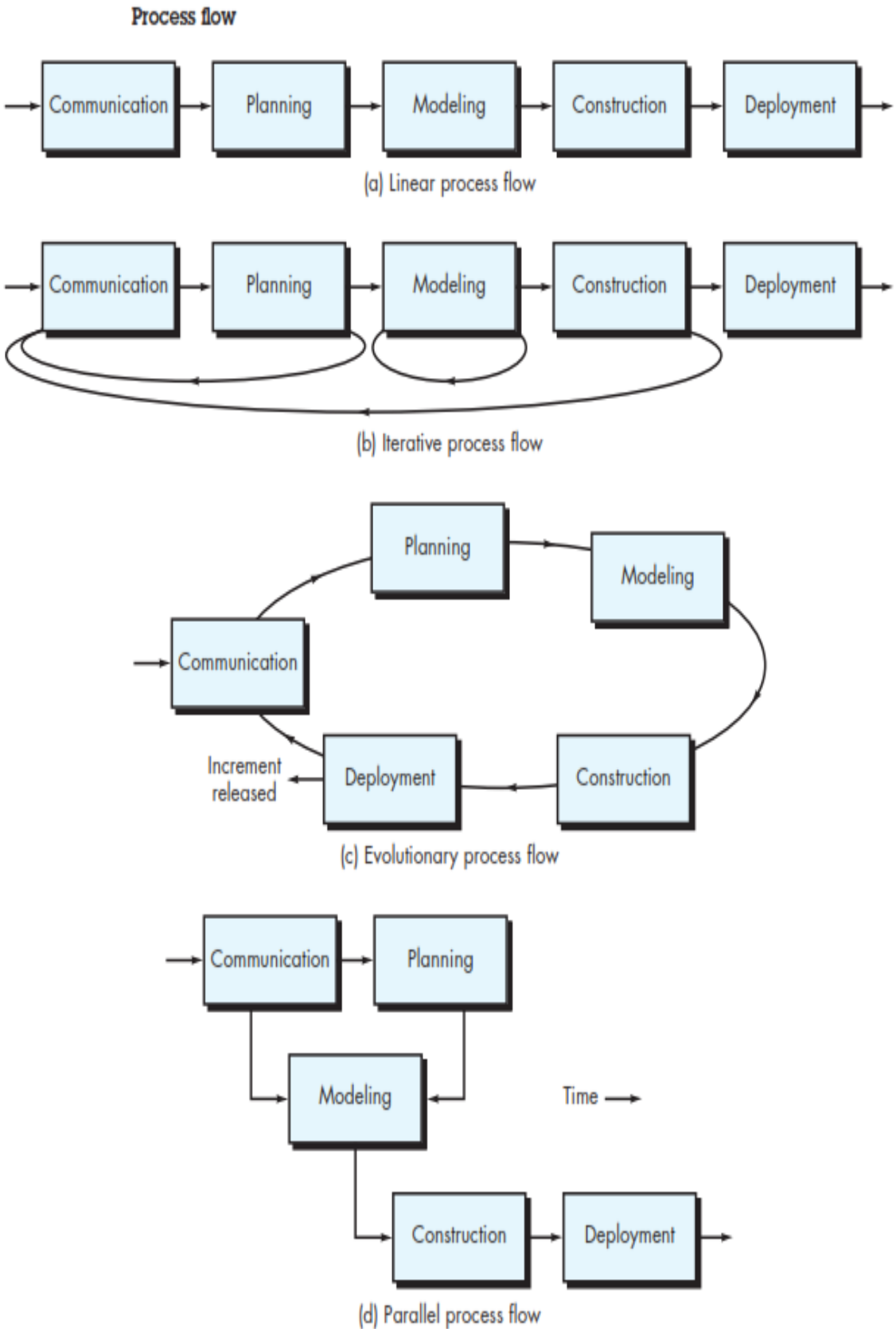
A GENERIC PROCESS MODEL:

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another. The software process is represented schematically in Figure 3.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A generic process framework for software engineering defines five framework activities—communication, planning, modeling, construction, and deployment.



There are different types of process flows are available:



Prescriptive Process models:

A prescriptive process model strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we examine the prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

The Waterfall Model:

There are times when the requirements for a problem are well understood— when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to ac- counting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

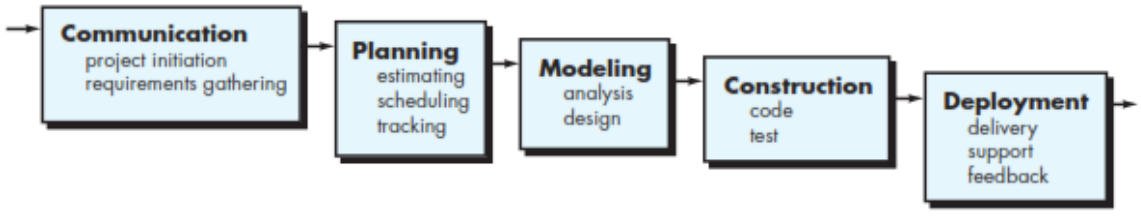
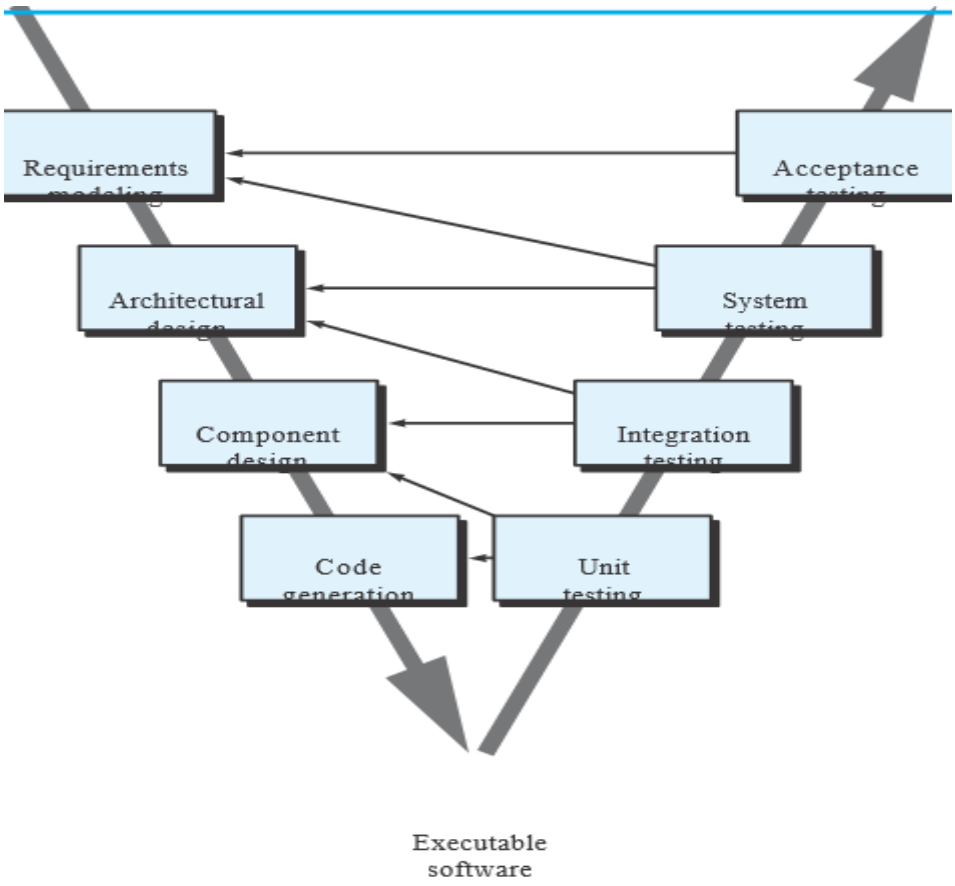


Fig: **The Waterfall Model**

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach2 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.

A variation in the representation of the waterfall model is called the V-model. Represented in below Figure, the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side. in reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.



The waterfall model is the oldest paradigm for software engineering. How- ever, over the past four decades, criticism of this process model has caused even ardent supporters to

question its efficacy [Han95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes.

   Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

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

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

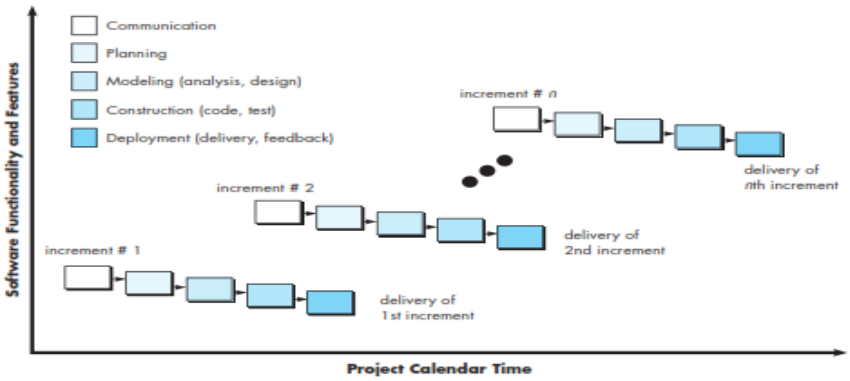
In an interesting analysis of actual projects, found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive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

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

Incremental process model:

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

The incremental model



The incremental model combines the elements' linear and parallel process flows. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93].

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm discussed in the next subsection.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/ or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

Unified Process Model.

The need for a "use case driven, architecture centric, iterative and incremental" software process when they state:

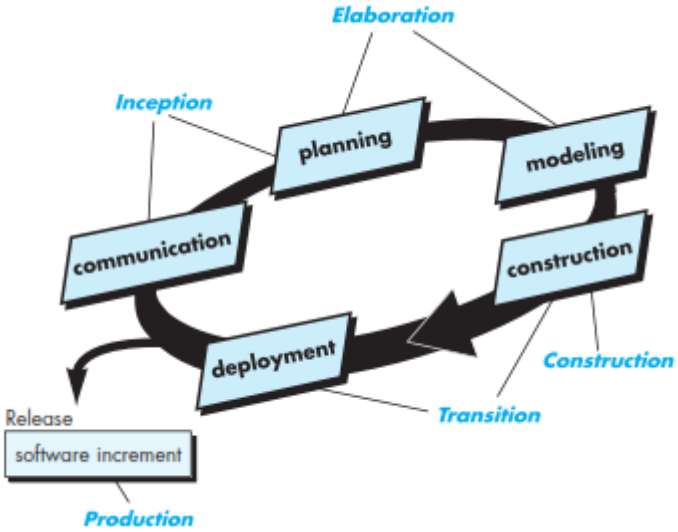
In some ways the Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many

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 (the use case).

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". It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of the Unified Process:

Earlier, we discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process is no exception. Below Figure depicts the "phases" of the UP and relates them to the generic activities.



The inception phase of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is pro- posed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases (Chapter 8) that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the functions and features that populate them. Later, the architecture will be refined and expanded into a set of models that will

represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The elaboration phase encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [Arl02] that represents a "first cut" executable system. The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

The construction phase of the UP is identical to the construction activity de- fined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests16 are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to de- rive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

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 feed- back 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. At the conclusion of the transition phase, the software increment becomes a usable software 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

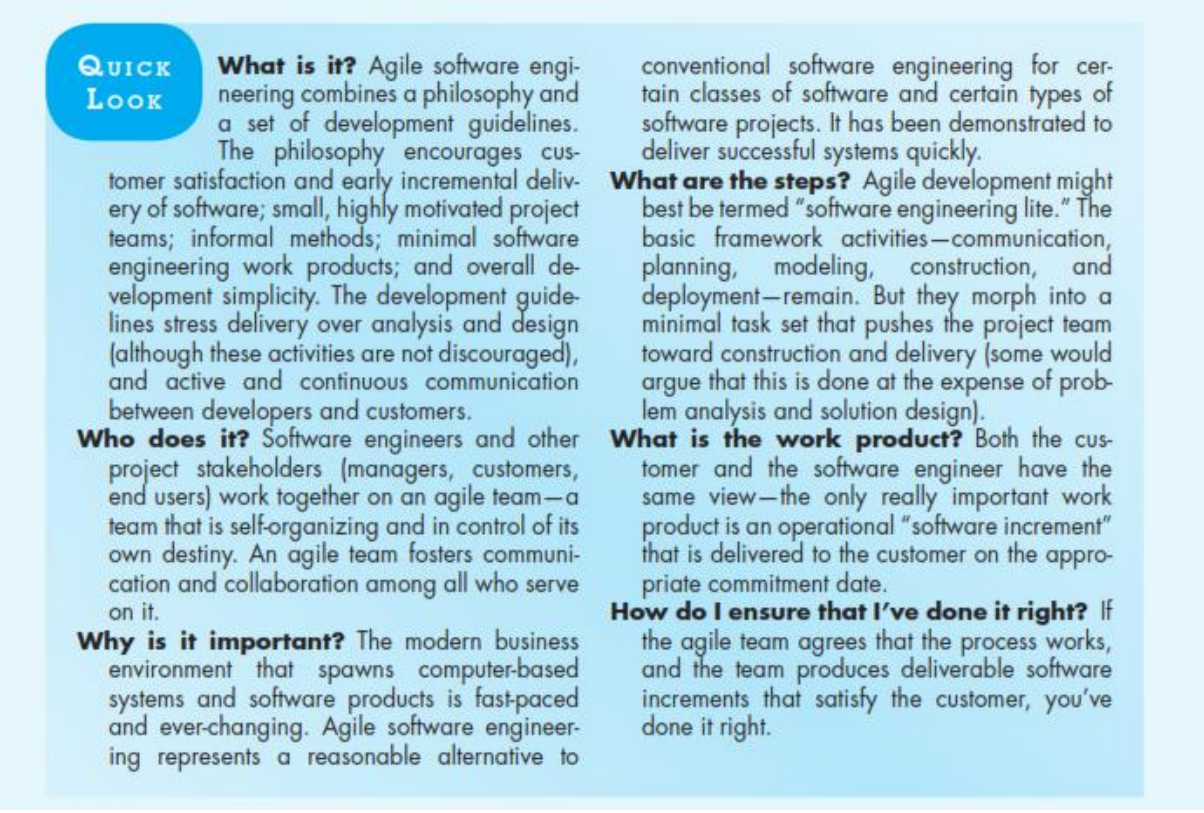
operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a workflow is analogous to a task set. That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet the needs.

Agile process model:

We can have quick look into, what is agile software engineering? And related questions.



WHAT IS AGILITY?

Agility has become today's buzzword when describing a modern software process.

Everyone is agile. 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.

The Agile Process Model in software engineering is a flexible, iterative, and customer-focused approach to software development. It emphasizes rapid delivery, continuous improvement, collaboration, and the ability to respond quickly to changing requirements.

Key Features of Agile Process Model:

Iterative and Incremental Development

- Software is developed in small, manageable units called **iterations or sprints** (typically 1 to 4 weeks).
- Each iteration results in a **working product increment**.

2. **Customer Collaboration**
   - Frequent interaction with the **customer or end-user** to gather feedback and refine requirements.
3. **Responding to Change**
   - Agile welcomes **changing requirements**, even in later stages of development.
4. **Cross-functional Teams**
   - Teams include **developers, testers, designers**, and **product owners** working together.
5. **Working Software is the Primary Measure of Progress**
   - Regular delivery of **functional software** rather than documentation.
6. **Continuous Feedback and Improvement**
   - After every sprint, the team conducts a **retrospective** to improve future processes.

Agile Process Flow:

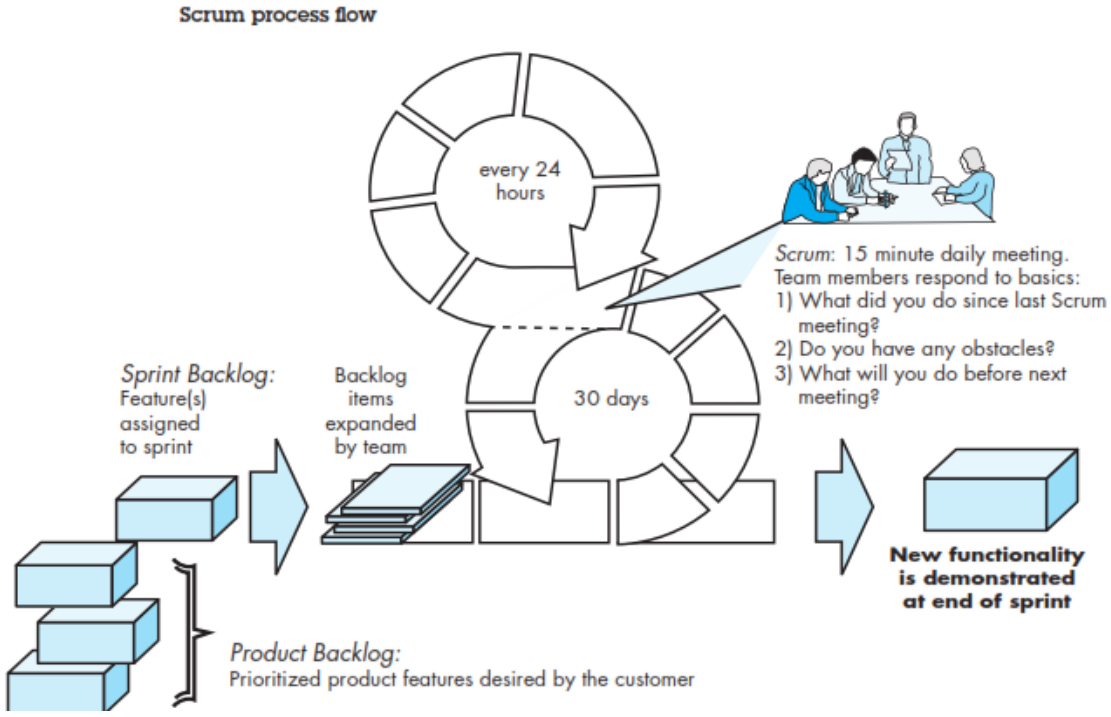
1. **Requirement Gathering**
   - Gather initial high-level user stories.
2. **Planning**
   - Prioritize tasks and create a sprint backlog.
3. **Design and Development**
   - Design and develop features within the sprint duration.

4. **Testing**
    - Continuous testing during and after each sprint.
5. **Delivery**
    - Deliver the working product increment.
6. **Review and Feedback**
    - Conduct sprint reviews with stakeholders.
7. **Retrospective**
    - Analyze what went well and what can be improved.

Popular Agile Frameworks:

- **Scrum** – Most widely used Agile framework with fixed-length sprints.
- **Kanban** – Focuses on continuous delivery with visual task boards.
- **Extreme Programming (XP)** – Emphasizes technical practices like pair programming and test-driven development (TDD).
- **Lean Software Development** – Focuses on eliminating waste and maximizing value.

Example framework:



Scrum:

Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwalbe and Beedle.

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following frame- work activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern (discussed in the following paragraph) called a sprint. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.

Scrum emphasizes the use of a set of software process patterns [Noy02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development activities:

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 (this is how changes are introduced). 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-box10 (typically 30 days). 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-minute) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [Noy02]:

- 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" [Bee99] 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.

Beedle and his colleagues [Bee99] present a comprehensive discussion of these patterns in which they state: “Scrum assumes up-front the existence of chaos . . .” The Scrum process patterns enable a software team to work success- fully in a world where the elimination of uncertainty is impossible.

Advantages of Agile:

- Faster delivery of usable software
- High customer satisfaction
- Flexible and adaptive to change
- Continuous improvement
- Better risk management

Disadvantages of Agile:

- Less documentation
- Requires active user involvement
- Not suitable for very large teams without proper coordination
- Can lead to scope creep if not properly managed