UNIT 1

Contents:

Software and Software Engineering: The Nature of Software, The Unique Nature of Web Apps, Software Engineering, Software Process, Software Engineering Practice, Software Myths.

Process Models: A Generic Process Model: Defining a framework activity, Prescriptive Process Models: The Waterfall Model, Incremental Process Model, Evolutionary Process Model, The Unified Process, What is an Agile Process? XP Process.

The Nature of Software:

Computer software continues to be the single most important technology on the world stage. 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 delivers the most important product of our time—information. The role of computer software has undergone significant change over the last half-century.

Defining Software:

A textbook description of software might take the following form:

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

Characteristics of Software that makes it different from Hardware:

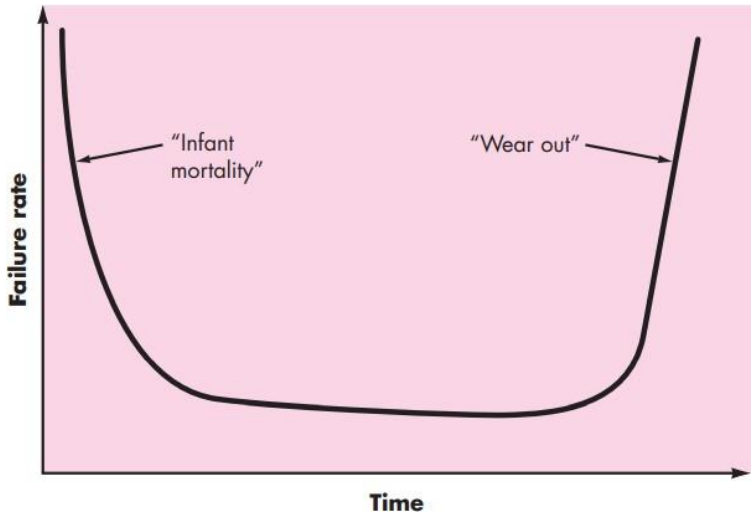
1. *Software is developed or engineered; it is not manufactured in the classical sense.*

   -- In both our focus is high quality. Both activities are dependent on people, but the relationship between people and work accomplished are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. *Software doesn't "wear out."*

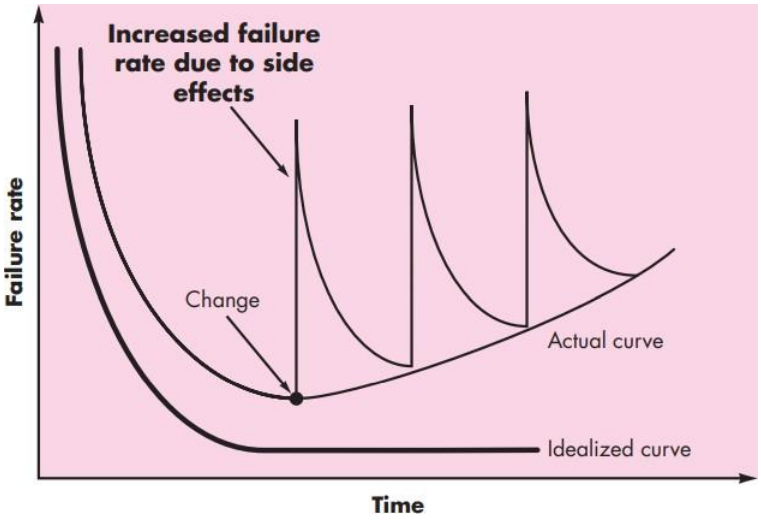
   -- Hardware exhibits relatively high failure rates early in its life (design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware

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.



Failure Curve (bathtub curve) for hardware

Software is not susceptible to environmental maladies like hardware. Undiscovered defects will cause high failure rates early in the life of a program. 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!



Failure curves for software

During its life, software will undergo change. As changes are made, errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". Slowly,

the minimum failure rate level begins to rise—the software is deteriorating due to change.

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

-- In the hardware world, component reuse is a natural part of the engineering process. Inthe software world, it is something that has only begun to be achieved on a broad scale. A software component should be designed and implemented so that it can be reused inmany 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.

Software Application Domains:

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,4 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** -- 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 (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics etc)
- **Web Applications** -- his network-centric software category spans a wide array of applications. WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications
- **Artificial Intelligence software** -- makes use of 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.

The unique nature of WebApps:

In the early days of the World Wide Web, websites were just a set of linked hypertext files which presented information using text and limited graphics. The augmentation of HTML by development tools like Java, XML enabled web engineers to provide computing capability along with informational content.

Web-based systems and applications (WebApps) are sophisticated tools that not only present stand-alone information but also integrate databases and business applications.

Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

The following are the common attributes for WebApps:

- **Network intensiveness:** A WebApp resides on a network (Internet or Intranet) and must serve the needs of a diverse community of clients.
- **Concurrency:** A large number of users may access the WebApp at one time.
- Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day.
- **Performance:** If a WebApp user must wait too long, he or she may decide to go elsewhere.
- **Availability:** Although the expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.
- **Data driven:** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user
- **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.
- **Immediacy:** 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 sensitive content must be protected and secure modes of data transmission must be provided.
- **Aesthetics:** An undeniable part of the appeal of a WebApp is its look and feel. W

Software Engineering:

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

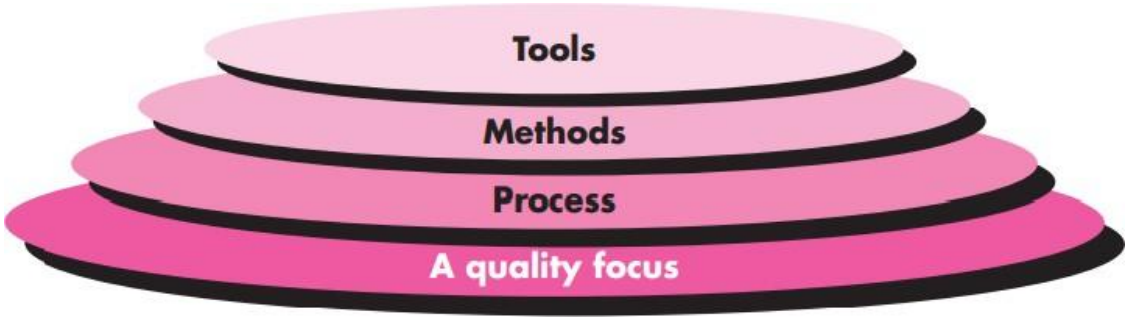
- ➔ It follows that a concerted effort should be made to understand the problem before a software solution is developed.
- ➔ It follows that design becomes a pivotal activity.
- ➔ It follows that software should exhibit high quality.
- ➔ It follows that software should be maintainable.

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered*.
The IEEE has developed a comprehensive definition for Software Engineering, where it states:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

Software engineering is a layered technology

- Any engineering approach (including software engineering) must rest on an organizational commitment to *quality*. 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
- 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 *tools* provide automated or semi automated support for the process and the methods.



The Software Process:

A process is a collection of activities, actions, and tasks that are performed when some work product is to be created.

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

A process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and

choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

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

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, to understand objectives of the project and to gather requirements.
- ***Planning:*** 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 tobe conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- ***Modeling:*** 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.

Software Engineering Practice:

The Essence of Practice:

The essence of problem solving is essentially the essence of software engineering practice. Here we try to answer some of the questions under different phases of problem solving.

- Understand the problem (communication and analysis).
  - Who has a stake in the solution to the problem?
  - What are the unknowns?
  - Can the problem be compartmentalized?
  - Can the problem be represented graphically?
- Plan a solution (modeling and software design).
  - Have you seen similar problems before?
  - Has a similar problem been solved?
  - Can subproblems be defined?
  - Can subproblems be defined?
- Carry out the plan (code generation).

- Does the solution conform to the plan?

- Is each component part of the solution provably correct?
- Examine the result for accuracy (testing and quality assurance).
  - Is each component part of the solution provably correct?
  - Does the solution produce results that conform to the data, functions, andfeatures that are required?

General Principles of Software Engineering:

The word principle is "an important underlying law or assumption required in a system of thought." David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole.

1. ***The First Principle: The Reason It All Exists***
   A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind.
2. ***The Second Principle: KISS (Keep It Simple, Stupid!)***
   All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system.
3. ***The Third Principle: Maintain the Vision***
   A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself.
4. ***The Fourth Principle: What You Produce, Others Will Consume***
   always specify, design, and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large.
5. ***The Fifth Principle: Be Open to the Future***
   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.
6. ***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.
7. ***The Seventh principle: Think!***
   Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again.

Software Myths:

Software myths are erroneous beliefs about software and the process that is used to build it. We categorize myths from three different perspectives.

Management myths:

***Myth**: We already have a book that's full of standards and procedures for building software. Won't*

that provide my people with everything they need to know?

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

Myth*: If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*
Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who are 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 firmbuild 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.
Myth*: A general statement of objectives is suficient 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.

Myth*: Software requirements continually change, but change can be easily accommodated because software is flexible.*
Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly

Practitioner's myths:
Myth*: Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time

Myth*: Until I get the program "running" I have no way of assessing its quality.*
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review.

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.

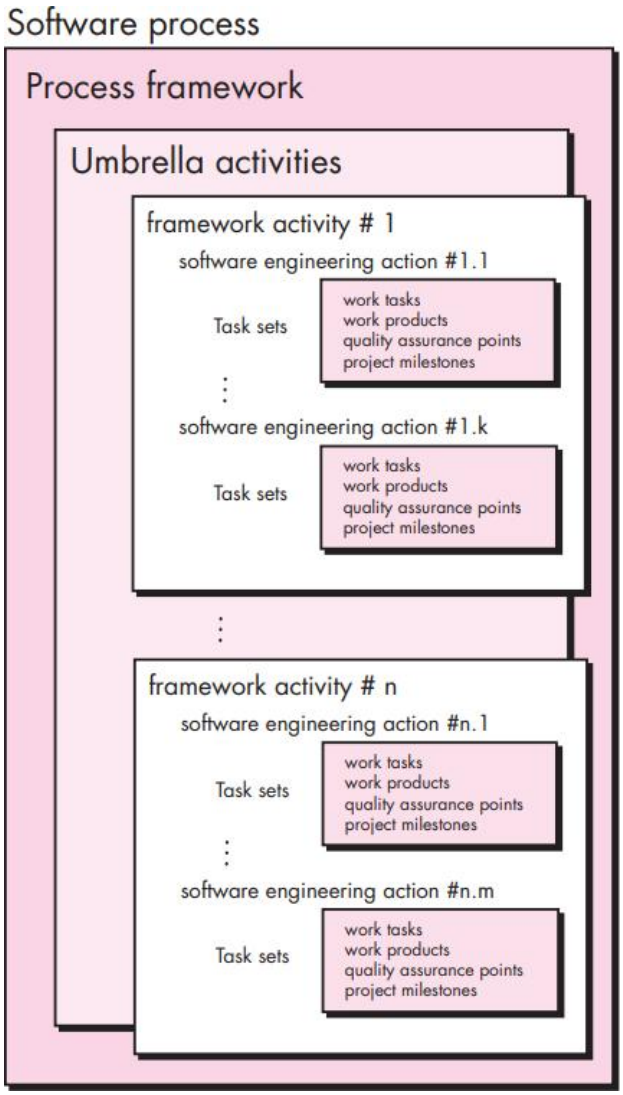
***Myth**: Software engineering will make us create voluminous and unnecessary documentation andwill invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a qualityproduct. Better quality leads to reduced rework.

Process Models

Generic Process Model:

- A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.
- Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.
- A generic process framework for software engineering defines five framework activities — ***communication, planning, modeling, construction, and deployment.***
- 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.
- One important aspect of the software process called process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.



Defining a Framework Activity:

To properly execute any one of these activities as part of the software process, a key question that the software team needs to ask is:

> *What actions are appropriate for a framework activity, given the nature of the problem tobe solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

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. The work tasks 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*.

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.

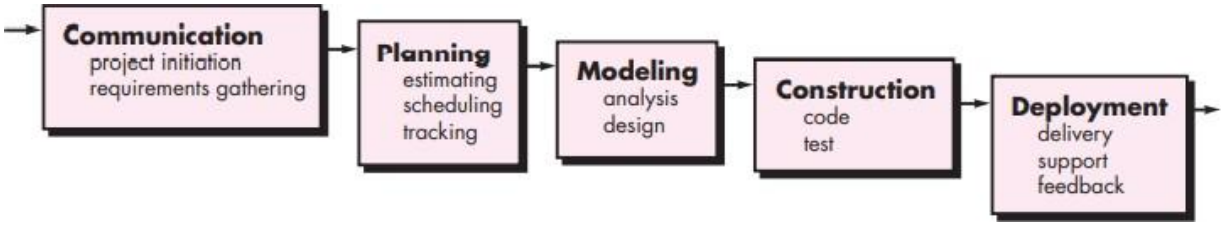
Prescriptive Process Models

Prescriptive process models were originally proposed to bring order to the chaos of software development. Software engineering work and the product that it produces remain on "the edge of chaos."

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.

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 approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

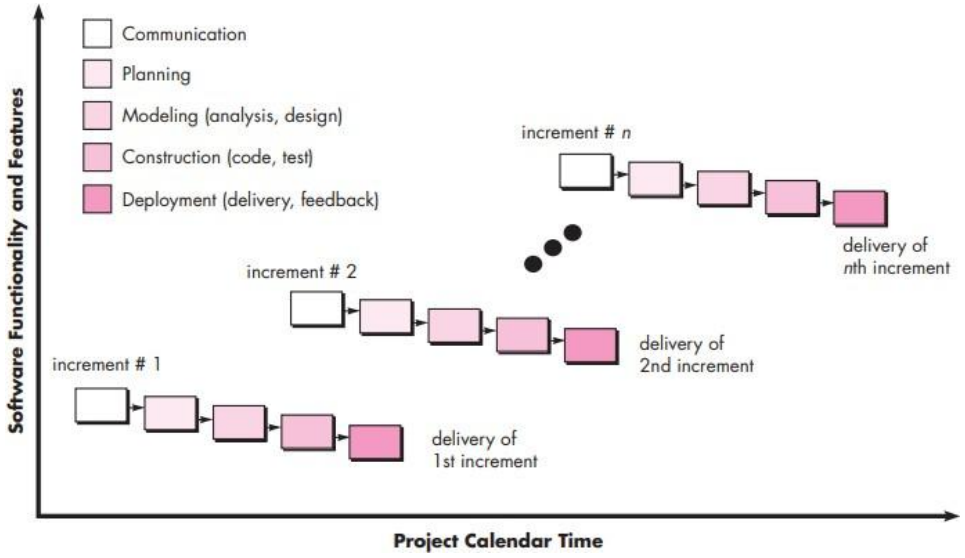


Waterfall Model or Classical lifecycle model

- The original waterfall model proposed by Winston Royce made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.
- The waterfall model is the oldest paradigm for software engineering. 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.
  4. 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 waterfall model is often inappropriate today, as software work is fast-paced and subject to a never-ending stream of changes.

Incremental Process Models

- The incremental model combines elements of 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 in a manner that is similar to the increments produced by an evolutionary process flow.



The Incremental Model

- 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 (or undergoes detailed evaluation).
- As a result, 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.
    - *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.*
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Evolutionary Process Models

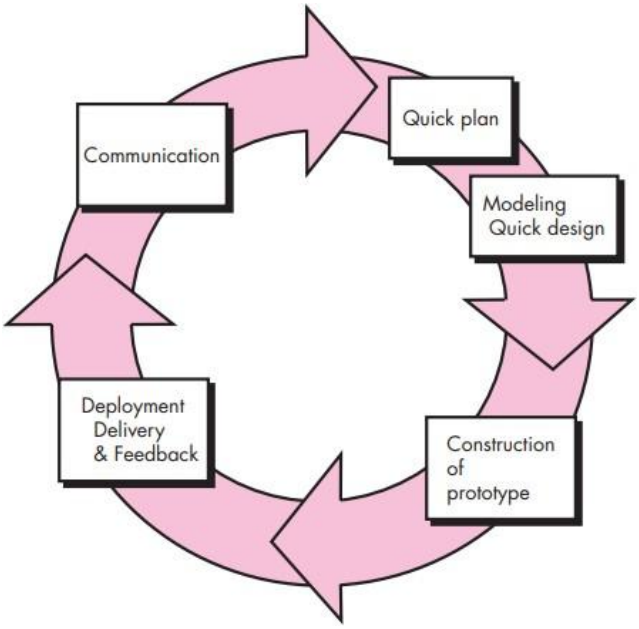
Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. There are two common evolutionary process models.

Prototyping

- A customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features.
- In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take.
- In these, and many other situations, a prototyping paradigm may offer the best approach. Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models.
- The prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

➔ The prototyping paradigm begins with communication.
    ◆ You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

- ➔ A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs.
  - ◆ A quick design focuses on a representation of those aspects of the software that willbe 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.



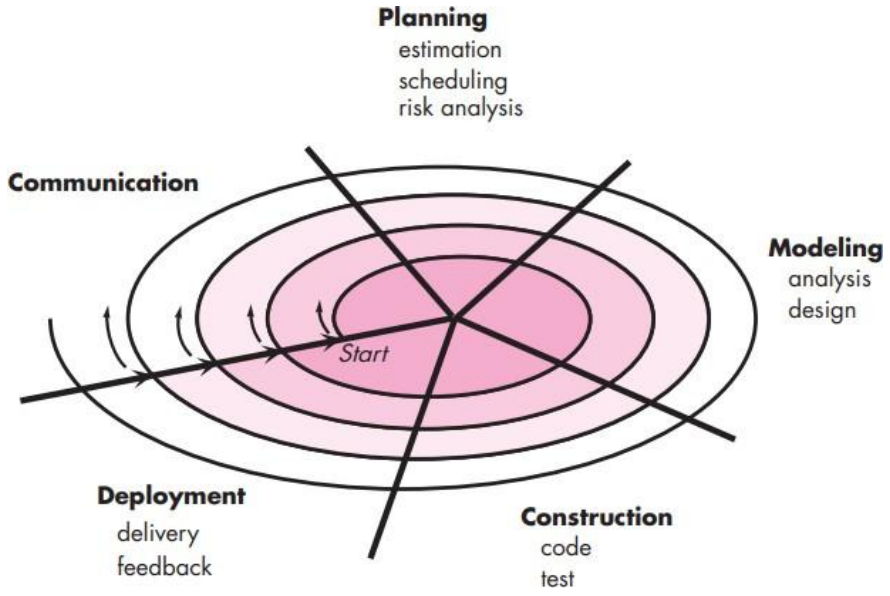
The prototyping paradigm

- The prototype serves as a mechanism for identifying software requirements.
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately.
- Yet, prototyping can be problematic for the following reasons:
  1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product.
  2. As a software engineer, you often make implementation compromises in order to geta prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were

inappropriate. The less-than-ideal choice has now become an integral part of the system.

The Spiral Model

- Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- 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.
  - 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.



- 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.

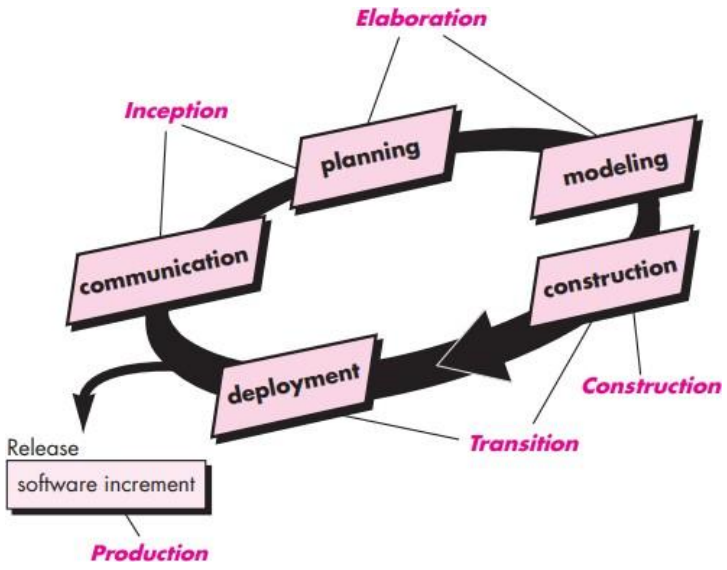
- The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.
- The spiral model uses prototyping as a risk reduction mechanism but, more importantly, enables you to apply the prototyping approach at any stage in the evolution of the product.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

The Unified Process Model

- During the early 1990s James Rumbaugh, Grady Booch, and Ivar Jacobson began working on a "unified method" that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling.
- The result was UML—a unified modeling language that contains a robust notation for the modeling and development of object-oriented systems.
- By 1997, UML became a de facto industry standard for object-oriented software development.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework to guide project teams in their application of the technology.
- Over the next few years, Jacobson, Rumbaugh, and Booch developed the Unified Process, a framework for object-oriented software engineering using UML.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

Phases of the Unified Process:

The following figure depicts the "phases" of the UP and relates them to the generic activities.



Unified Process Model

1. ***The Inception phase:***
   - 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 proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.
2. ***The elaboration phase:***
   - 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 partof the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.
   - In some cases, elaboration creates an "executable architecture baseline" that represents a "first cut" executable system.
3. ***The construction phase:***
   - The construction phase of the UP is identical to the construction activity defined for the generic software process.
   - Using the architectural model as input, the construction phase develops or acquiresthe software components that will make each use case operational for end users.
   - All necessary and required features and functions for the software increment are then implemented in source code.
   - As components are being implemented, unit tests are designed and executed for each.
   - In addition, integration activities (component assembly and integration testing) are conducted.
4. ***The transition 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 feedback reports both defects and necessary changes.
   - At the conclusion of the transition phase, the software increment becomes a usable software release.
5. ***The production phase:***
   - 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.

- 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.

Agile Development

- Agile software engineering combines a philosophy and a set of development guidelines.
- The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.
- The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

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 to 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.

- Agility can be applied to any software process.
- The modern business environment that spawns computer-based systems and software products is fast-paced and ever changing.
- Agile software engineering represents a reasonable alternative to conventional software engineering for certain classes of software and certain types of software projects.
- It has been demonstrated to deliver successful systems quickly.

Self study:

☐ ***Agility and the cost of change***

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:

- 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.
- 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.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

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 harnesschange 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.

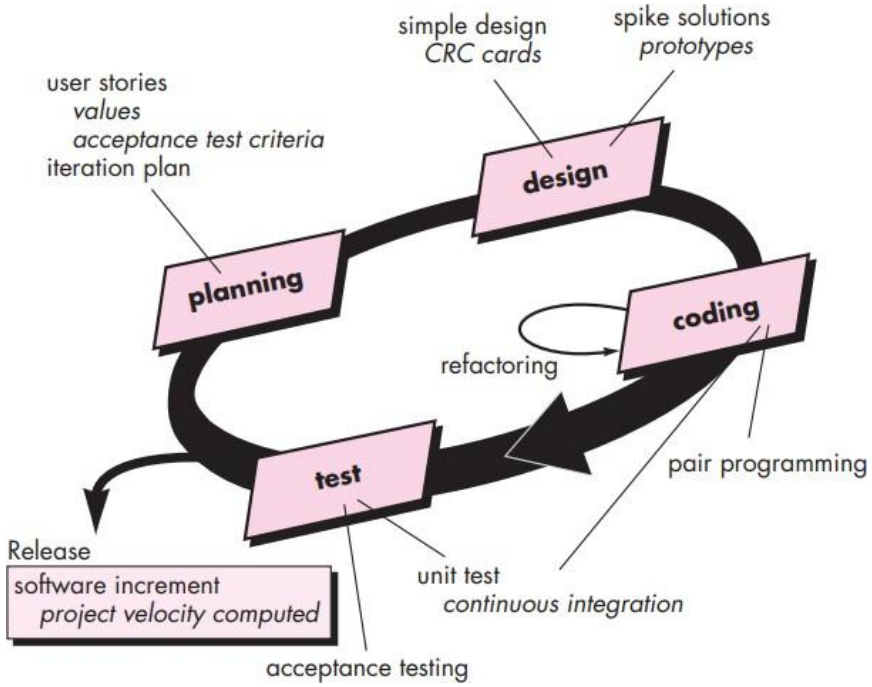
Extreme Programming (XP):

- Extreme Programming (XP), the most widely used approach to agile software development.
- Beck defines a set of five values that establish a foundation for all work performed as part of XP—***communication, simplicity, feedback, courage, and respect***.
- Each of these values is used as a driver for specific XP activities, actions, and tasks.
- In order to achieve effective ***communication*** between software engineers and other stakeholders,, 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.
- ***Feedback*** is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy, the software provides the agile team with feedback. XP makes use of the unit test as its primary testing tactic
- Beck argues that strict adherence to certain XP practices demands ***courage***. A better word might be discipline.
- 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.

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***.
- The following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.



XP Process

1. **Planning:**
   - The planning activity (also called the planning game) begins with listening—a requirements gathering activity.
   - Listening leads to the creation of a set of "stories" (also called user stories) that describe required output, features, and functionality for software to be built.
   - Each story is written by the customer and is placed on an index card.
   - The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function.
   - Members of the XP team then assess each story and assign a cost—measured in development weeks—to it.
   - If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again.
   - It is important to note that new stories can be written at any time.
   - After the first project release (also called a software increment) has been delivered, the XP team computes project velocity.

- Project velocity is the number of customer stories implemented during the first release.

2. **Design:**
   - XP design rigorously follows the KIS (keep it simple) principle.
   - A simple design is always preferred over a more complex representation.
   - XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context.
   - CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment.
   - If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design, called a ***spike solution***.
   - 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.
3. **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
   - Once the unit test has been created, the developer is better able to focus on what mustbe implemented to pass the test.
   - 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.
   - This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created).
   - It also keeps the developers focused on the problem at hand.
4. **Testing:**
   - We 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.
   - 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.