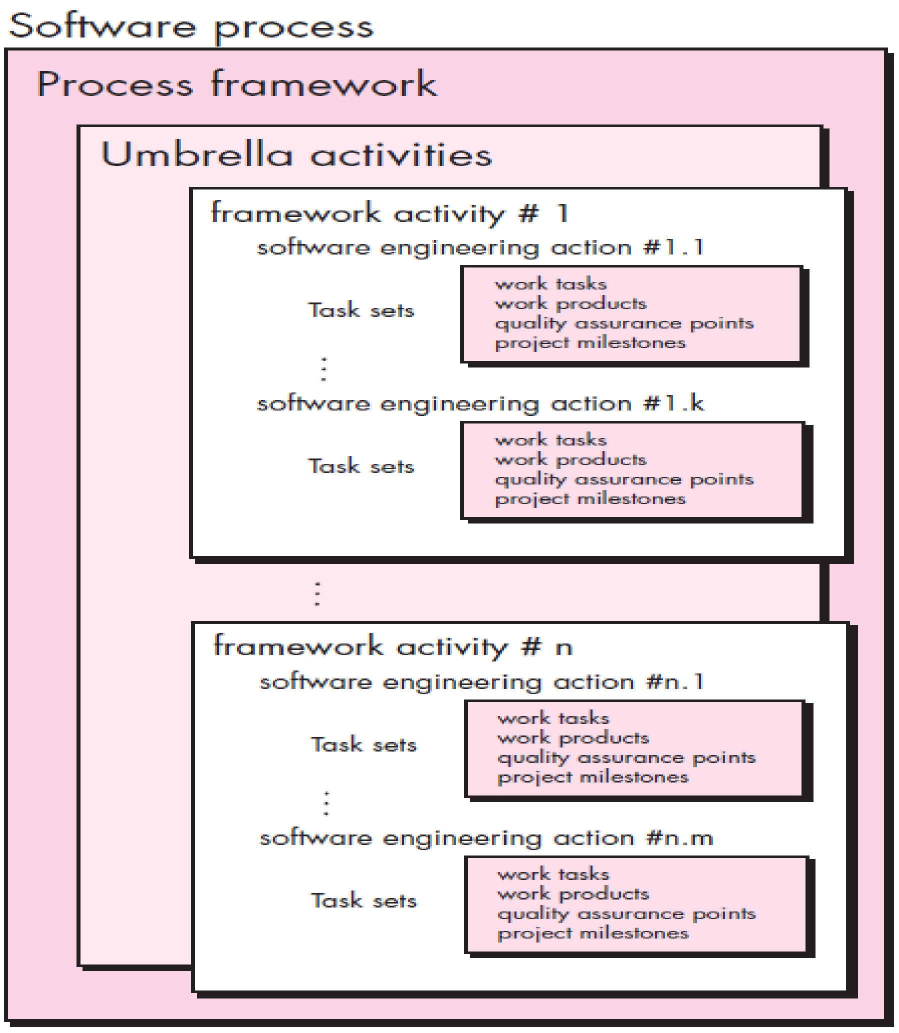
UNIT - III:

SYSTEM MODELS, PROJECT MANAGEMENT: System Models: Context models; Behavioural models; Data models; Object models; Structured methods. Project Management: Management activities; Project planning; Project scheduling; Risk management.

PROCESS MODELS

A GENERIC PROCESS MODEL

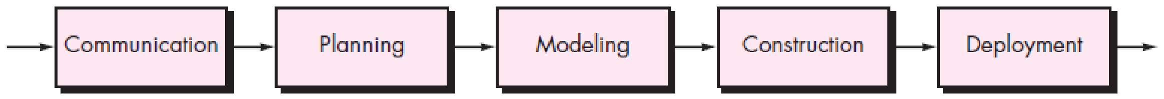
The software process is represented schematically in following figure. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.



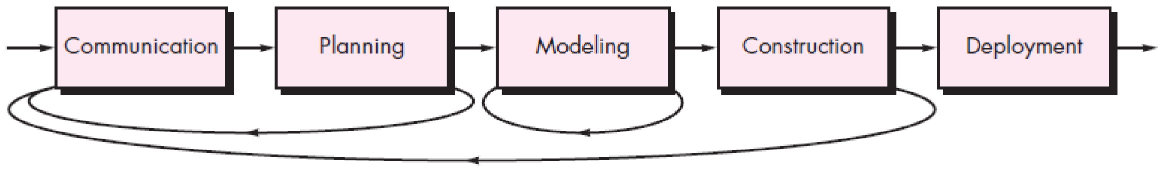
A generic process framework defines **five** framework activities—**communication, planning, modeling, construction,** and **deployment.**

In addition, a set of umbrella activities **project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others** are applied throughout the process.

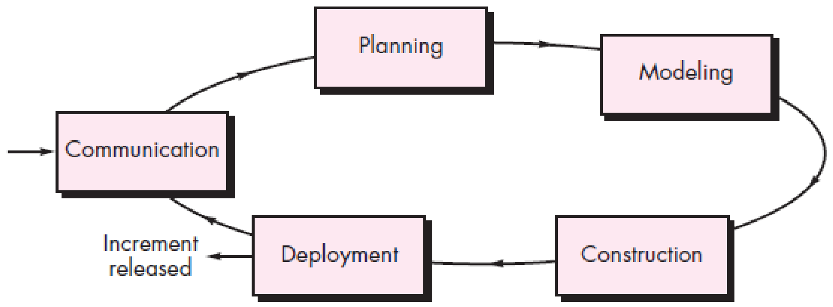
This aspect is called ***process flow.*** It describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in following figure



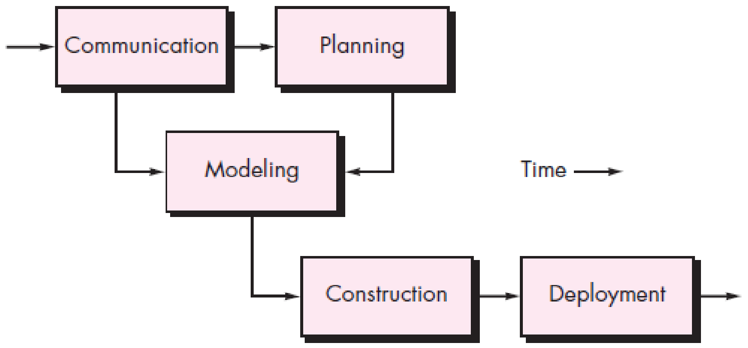
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

A generic process framework for software engineering A ***linear process flow*** executes each of the **five** framework activities in sequence, beginning with communication and culminating with deployment.

An ***iterative process flow*** repeats one or more of the activities before proceeding to the next. An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software. A *parallel process flow* executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

Defining a Framework Activity

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?

Identifying a Task Set

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics. A task set defines the actual work to be done to accomplish the objectives of a software engineering action.

Process Patterns

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

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

Ambler has proposed a template for describing a process pattern:

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

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

Type. The pattern type is specified. Ambler suggests **three** types:

1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **Establishing Communication.** This pattern would incorporate the task pattern **Requirements Gathering** and others.
2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., Requirements Gathering is a task pattern).
3. ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **Spira lModel** or **Prototyping.**

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

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

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

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

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

Solution. Describes how to implement the pattern successfully. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

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

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

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

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

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

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

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).

PROCESS ASSESSMENT AND IMPROVEMENT

Assessment attempts to understand the current state of the software process with the intent of improving it.

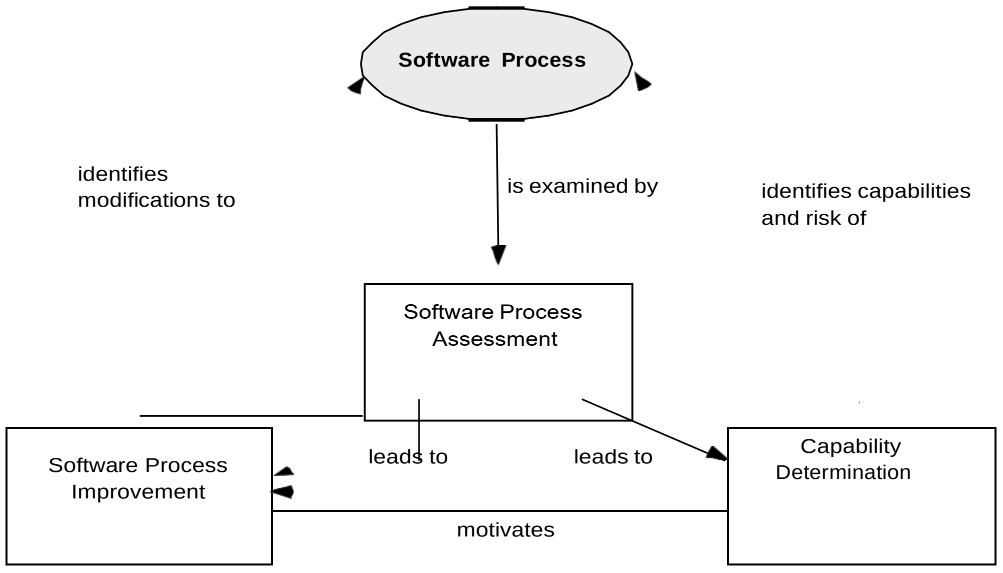
A number of different approaches to **software process assessment and improvement** have been proposed over the past few decades.

Standard CMMI Assessment Method for Process Improvement (SCAMPI)—provides a **five** step process assessment model that incorporates **five** phases: **initiating, diagnosing, establishing, acting, and learning.** The SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

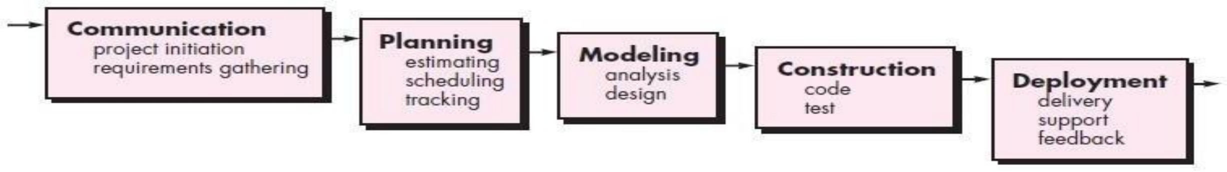

Software Process
identifies modifications to
is examined by
identifies capabilities and risk of
Software Process Assessment
Software Process Improvement
leads to
leads to
Capability Determination
motivates

PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. Prescriptive process models define a prescribed set of process elements and a predictable process work flow. “prescriptive” because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, qualityassurance, and change control mechanisms for each project.

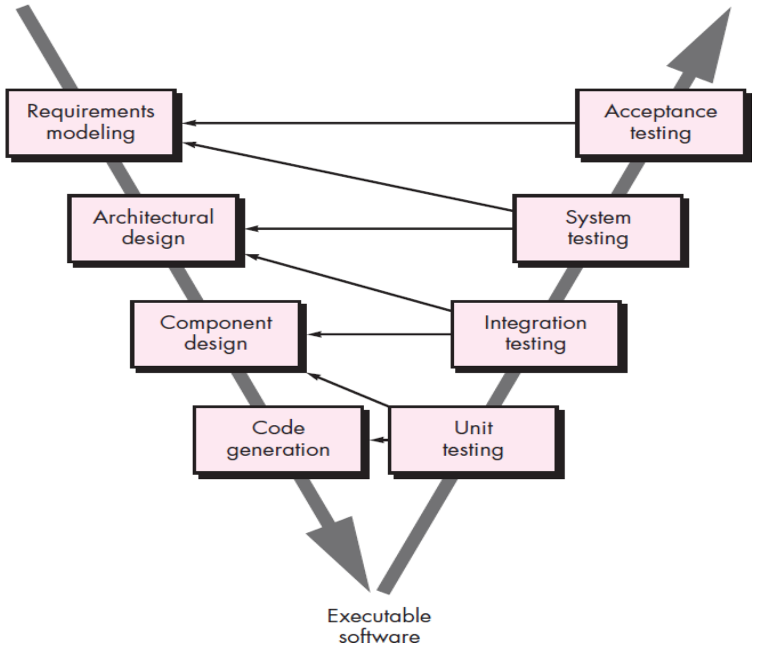
The Waterfall Model

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**.



The waterfall model

A variation in the representation of the waterfall model is called the ***V-model.*** Represented in following figure. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.



The V-model

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 that validate each of the models created as the team moved down the left side. 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. 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.

This model is suitable when ever limited number of new development efforts and when requirements are well defined and reasonably stable.

Incremental Process Models

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

The *incremental* model combines elements of linear and parallel process flows discussed in Section 1.7. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

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

Incremental development is particularly useful when **staffing is unavailable** for a complete implementation by the business deadline that has been established for the project. Earlyincrements can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

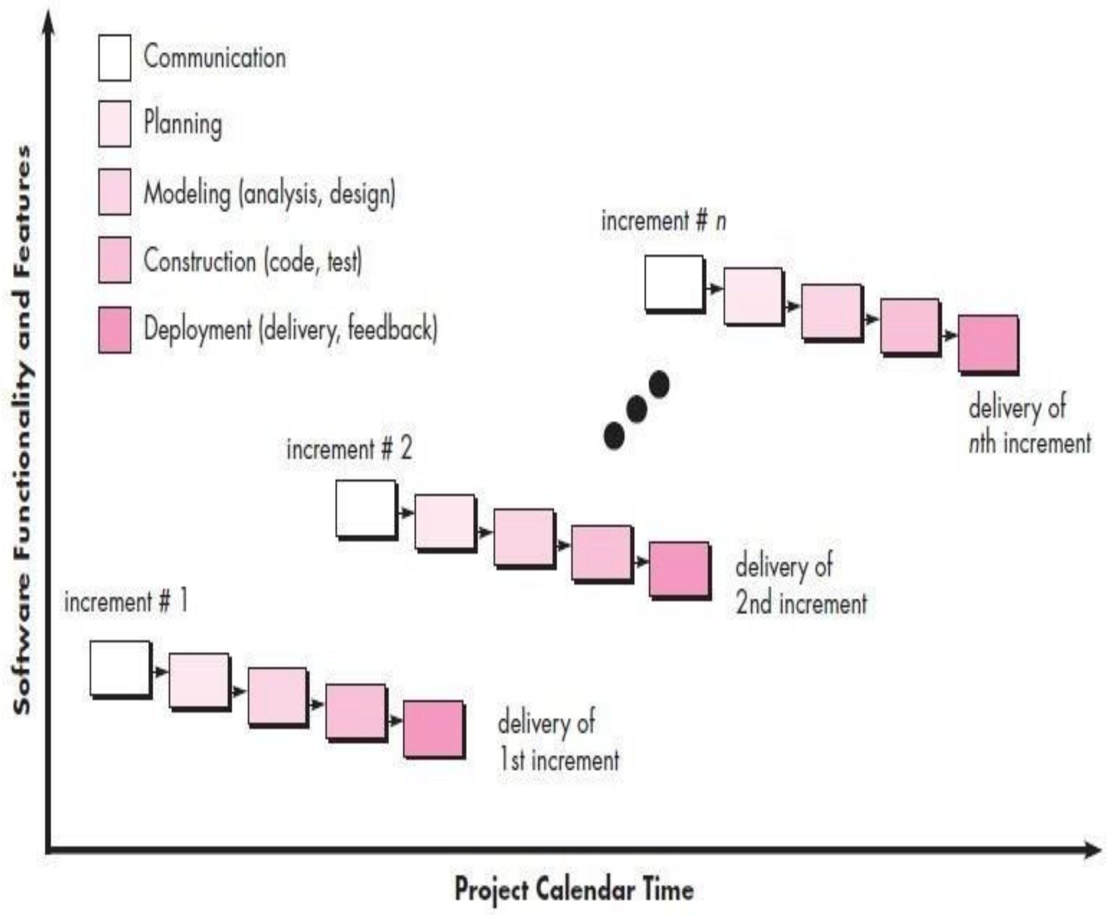

Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

Fig : Incremental Model

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 with each iteration. There are **two** common evolutionary process models.

Prototyping Model : Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a ***prototyping paradigm*** may offer the best approach.

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

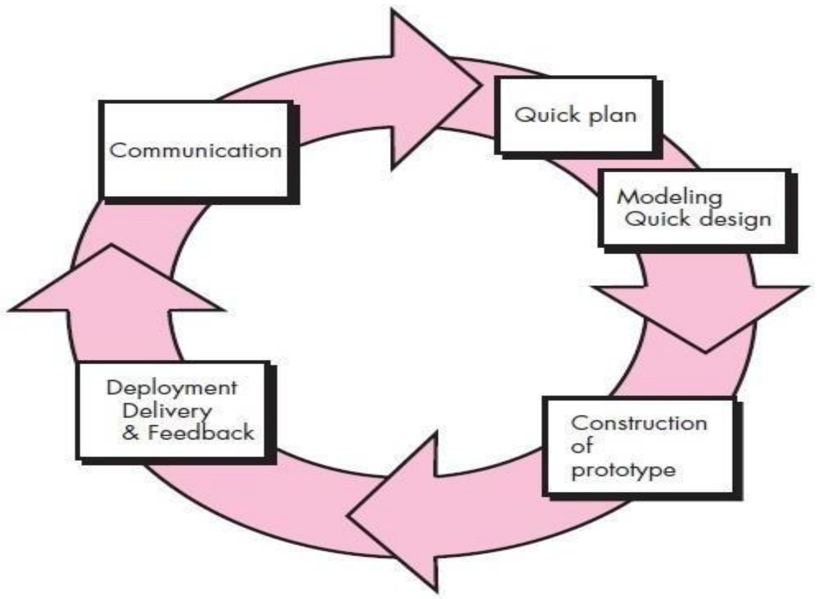


Fig : prototyping paradigm

The quick design leads to the **construction of a prototype**. The prototype is deployedand 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 prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly. The prototype can serve as **“the first system**.” 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.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability.

Although problems can occur, prototyping can be an **effective paradigm** for software engineering.

The Spiral Model : Originally proposed by **Barry Boehm**, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner

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

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

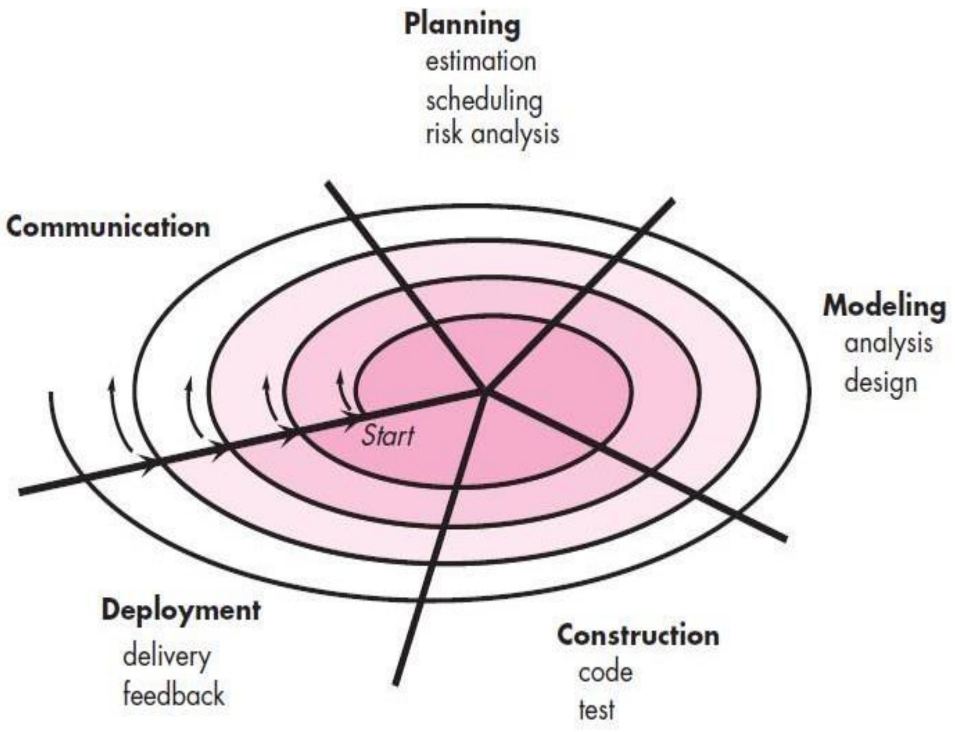


Fig : The Spiral Model

A spiral model is divided into a set of **framework activities** defined by the software engineering team. 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*** are a combination of work products and conditions that are attained along the path of the spiral are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a **product** specification; subsequent passes around the spiral might be used to develop a **prototype** and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan.

The spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "**concept development project**" that starts at the core of the spiral and continues for multiple iterations until concept development is complete. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "**product enhancement project.**"

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. 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.

Concurrent Models

The concurrent development model, sometimes called **concurrent engineering**, allows a software team to represent iterative and concurrent elements of any of the process models. The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

These models provides a schematic representation of one software engineering activity within the **modeling** activity using a concurrent modeling approach. The activity **modeling** may be in any one of the states noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner.

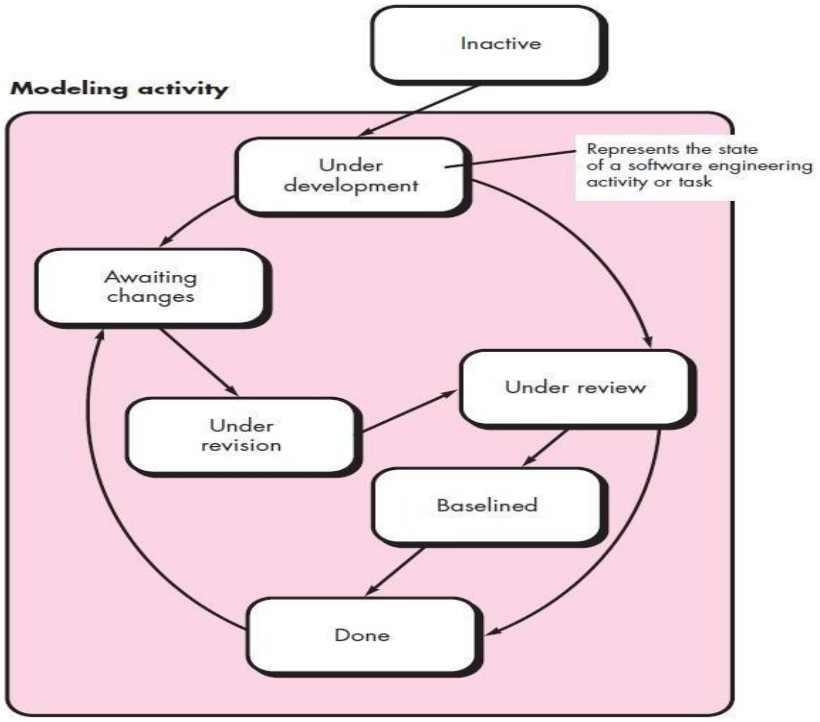


Fig : Concurrent development model

All software engineering activities exist concurrently but reside in different states. Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. This generates the event *analysis model correction,* which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

SPECIALIZED PROCESS MODELS

Component-Based Development

The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from **prepackaged** software components.

Modeling and construction activities begin with the identification of **candidate components.** These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called ***clean room software engineering***.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineeringparadigms. **Ambiguity, incompleteness, and inconsistency** can be discovered and corrected more easily, but through the application of mathematical analysis.

When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected. Although not a mainstream approach, the formal methods model offers the promiseof **defect-free software**.

Draw Backs:

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formalmethods, extensive training is required.
- It is difficult to use the models as a communication mechanism for Technica llyunsophisticated customers.

Aspect-Oriented Software Development

AOSD defines “aspects” that express customer concerns that cut across multiple system functions, features, and information. When concerns cut across multiple system functions, features, and information, they are often referred to as ***crosscutting concerns***. ***Aspectual requirements*** define those crosscutting concerns that have an impact across the software architecture.

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for **defining, specifying, designing, and constructing *aspects***.”

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

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called “**aspects**,” to characterize cross-cutting functional and non-functional properties of components.

Project management

I. Management Activities:

1. Definition:
    - Project management involves planning, organizing, and overseeing the execution of a project to achieve specific goals within a specified time frame.
2. Key Management Activities:
    - Planning:
        - Defining project goals, scope, and deliverables.
        - Identifying tasks and allocating resources.
    - Organizing:
        - Structuring the project team.
        - Defining roles and responsibilities.
    - Controlling:
        - Monitoring project progress.
        - Taking corrective actions as needed.
    - Leading:
        - Motivating and guiding the project team.
        - Resolving conflicts.
    - Closing:
        - Formalizing project completion.
        - Conducting post-project reviews.

II. Project Planning:

1. Definition:
    - Project planning is the process of defining the scope, objectives, tasks, and resources required to complete a project successfully.
2. Key Planning Components:
    - Scope Definition:
        - Clearly defining the project's boundaries and objectives.
        - Outlining what is and isn't included in the project.
    - Task Identification:
        - Breaking down the project into manageable tasks.
        - Creating a work breakdown structure (WBS).
    - Resource Allocation:

        - Assigning human, financial, and material resources to tasks.
        - Ensuring resources are available when needed.
    - Time Estimation:
        - Estimating the time required to complete each task.
        - Creating a project timeline.

III. Project Scheduling:

1. Definition:
    - Project scheduling involves creating a timeline that outlines when each task will be performed, ensuring that the project stays on track.
2. Key Scheduling Activities:
    - Task Sequencing:
        - Determining the order in which tasks will be performed.
        - Identifying dependencies between tasks.
    - Critical Path Analysis:
        - Identifying the critical path, which is the sequence of tasks that must be completed on time for the project to stay on schedule.
    - Gantt Charts:
        - Visual representation of project tasks over time.
        - Helps in understanding task durations and dependencies.
    - Resource Leveling:
        - Balancing resource usage to avoid overloading individuals or teams.

IV. Risk Management:

1. Definition:
    - Risk management involves identifying potential risks that could impact the project, assessing their likelihood and impact, and developing strategies to mitigate or respond to them.
2. Key Risk Management Activities:
    - Risk Identification:
        - Identifying potential risks to the project.
        - Classifying risks into categories (e.g., technical, organizational).
    - Risk Assessment:
        - Evaluating the likelihood and impact of each identified risk.
        - Prioritizing risks based on severity.

- Risk Mitigation:
    - Developing strategies to reduce the likelihood or impact of identified risks.
    - Creating contingency plans.
- Monitoring and Control:
    - Continuously monitoring for new risks.
    - Implementing risk responses as needed.

V. Best Practices:

1. Communication:
    - Open and transparent communication is crucial for effective project management.
2. Flexibility:
    - Projects often encounter unexpected challenges, and the ability to adapt and revise plans is essential.
3. Stakeholder Involvement:
    - Involving stakeholders in the planning and decision-making process increases project buy-in and success.
4. Documentation:
    - Maintaining detailed documentation of plans, schedules, and risk management strategies facilitates communication and accountability.