MODULE –1

CHAPTER 1 - INTRODUCTION

Software and Software Engineering

Software Engineering is the product of two words, software, and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called a **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software Engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Definitions

IEEE defines software engineering as:

(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 the above statement.

Fritz Bauer, a German computer scientist, defines software engineering as: Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

1.1 The Nature of software

Software takes on a dual role. It is a **Product** and at the same time a **Vehicle** (Process) for delivering a product.

As a Product, it produces, manages, modifies, acquires and displays the information–It is an **information transformer** that can be a single bit or a complex multimedia presentation delivered from data acquired from dozens of independent sources.

As a Vehicle (Process), delivers the product. Software acts as the basis for:

- Control of other computer (Operating Systems)

- Communication of Information (Networks)
- Creation and control of other programs (Software Tools and Environment)

Software delivers the most important product of our time called information. It transforms personal data (Individual financial transactions), It manages business information, It provides gateway to worldwide information networks (Internet) and provides the means for acquiring information in all of its forms.

1.1.1 Software

Defining Software : Software is defined as

1. **Instructions** : Programs that when executed provide desired function, features, and performance.
2. **Data structures:** Enable the programs to adequately manipulate information.
3. **Documents:** Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics of software

Software has characteristics that are considerably different than those of hardware:

1) Software is developed or engineered; it is not manufactured in the Classical Sense.

- Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce **quality problems** that are nonexistent or easily corrected for software.
- Both the activities are dependent on people, but the relationship between people is totally varying. These two activities require the construction of a "product" but the approaches are different.
- Software costs are concentrated in engineering which means that software projects cannot be managed as if they were manufacturing.

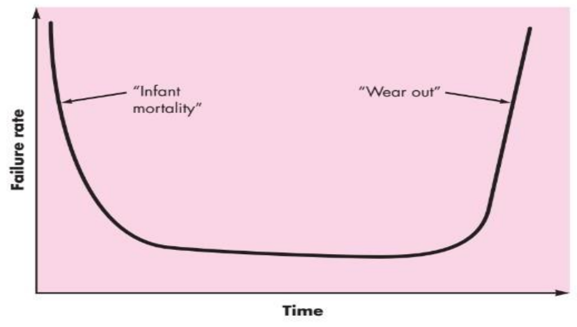
2) Software doesn't "Wear Out"

- In early stage of hardware development process the failure rate is very high due to manufacturing defects, but after correcting defects failure rate gets reduced.
- Hardware components suffer from the growing effects of many other environmental factors. Stated simply, the hardware begins to **wear out.**
- Software is not susceptible to the environmental maladies (extreme temperature, dusts and vibrations) that cause hardware to wear out [Fig:1.1]

  The following figure shows the relationship between failure rate and time.

FIGURE 1.1

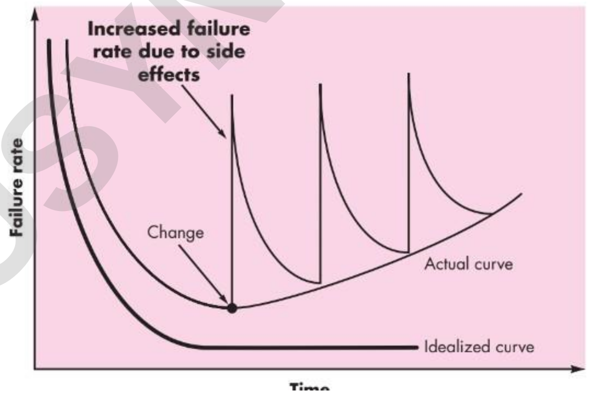
Failure curve for hardware



- When a hardware component wears out, it is replaced by a spare part. There are no software spare parts.
- Every software failure indicates an error in design or in the process through which the design was translated into machine-executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance. However, the implication is clear—the software doesn't wear out. But it does **deteriorate** (frequent changes in requirement) **[Fig:1.2].**

FIGURE 1.2

Failure curves for software



3) Most Software is custom-built rather than being assembled from components:

- A software part should be planned and carried out with the goal that it tends to be reused in various projects (algorithms and data structures).
- Today software industry is trying to make library of reusable components E.g. Software GUI is built using the reusable components such as message windows, pull down menu and many more such components.
- In the hardware world, component reuse is a natural part of the engineering process.

1.1.2 Software Application Domains

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

1. **System Software:**
   - A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities). Other system applications (e.g. Operating system components, drivers, networking software, telecommunication processors) process largely indeterminate data.
   - In both cases there is heavy interaction with computer hardware, heavy usage by multiple users, scheduling and resource sharing.

2. **Application Software:**
   - Stand-alone programs that solve a specific business need. [Help users to perform specific tasks].
   - Application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

3. **Engineering/Scientific Software:**
   - It has been characterized by "number crunching" algorithms. (Complex numeric computations).
   - Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. Computer-aided design, system simulation, and other interactive applications have begun to take a real-time and even system software characteristic.

4. **Embedded Software:**
   - It 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., keypad 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).

5. **Product-line Software:**
   - Designed to provide a specific capability for use by many different customers.
   - Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

6. **Web Applications**:
   - It is a client-server computer program that the client runs on the web browser.

- These applications are called “WebApps,” this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

7. **Artificial Intelligence Software:**
   - These makes use of non-numerical algorithms to solve complex problems. Knowledge based expert systems.
   - Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

New Software Challenges:

- **Open-world computing:** Creating software to allow machines of all sizes to communicate with each other across vast networks (Distributed computing—wireless networks).

- **Net sourcing**: Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide (the Web as a computing engine).

- **Open Source:** Distributing source code for computing applications so customers can make local modifications easily and reliably (“free” source code open to the computing community).

1.1.3 Legacy Software

- Legacy software is **older programs** that are developed decades ago.

- The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.

As time passes legacy systems evolve due to following reasons:

- The software must be adapted to meet the needs of new computing environment or technology.

- The software must be enhanced to implement new business requirements.

- The software must be extended to make it interoperable with more modern systems or database.

- The software must be re-architected to make it viable within a network environment.

1.2 Unique Nature of Web Apps

In the early days of the World Wide Web (1990-95), websites consisted of little more than a set of linked **hypertext files** (HTML) 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 applications (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.

WebApps are one of a number of distinct software categories. 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 attributes are encountered in the vast majority of WebApps.

- **Network intensiveness**. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

- **Concurrency**. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

- **Performance**. WebApp should work effectively in terms of processing speed. If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

- **Availability**. Although expectation of 100 percent availability is un reasonable, 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. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

- **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**. Although immediacy—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.

- **Security**. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

- **Aesthetics**. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

1.3 Software Engineering -A Layered Technology

In order to build software that is ready to meet the challenges of the 21st century, you must recognize a few simple realities.

- **Problems should be understood before a software solution is developed.**

- **Design is a pivotal Software Engineering activity.**

- **Software should exhibit high quality.**

- **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**.

Software Engineering:
Fritz Bauer defined as:

Software engineering is the establishment and use of sound engineering principles in order to obtain software that is reliable and works efficiently on real machines in economical manner.

IEEE has developed a more comprehensive definition as:

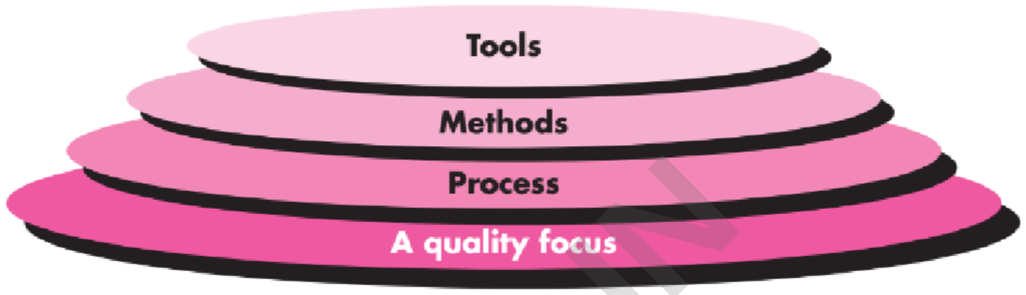
1) Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

2) The study approaches as in (1)

*Software Engineering is a **layered technology.** Software Engineering encompasses a **Process**, **Methods** for managing and engineering software and **tools**.*

Software engineering is a fully layered technology, to develop software we need to go from one layer to another. All the layers are connected and each layer demands the fulfillment of the previous layer. **[Fig:1.3]**

FIGURE 1.3
Software engineering layers



Software engineering is a layered technology. Referring to above Figure, any engineering approach 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 that must be established for effective delivery of software engineering technology.
- 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. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering is established.

1.4 THE SOFTWARE PROCESS

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

A generic process framework for software engineering encompasses five activities:

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

- **Planning**. 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**. Creation of models to help developers and customers understand the requirements and software design.

- **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 process framework activities are complemented by a number of ***Umbrella Activities***. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical umbrella activities include:

- **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- **Risk management—**assesses risks that may affect the outcome of the project or the quality of the product.
- **Software quality assurance—**defines and conducts the activities required to ensure software quality.
- **Technical reviews—**assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders needs; can be used in conjunction with all other framework and umbrella activities.
- **Software configuration management**—manages the effects of change throughout the software process.
- **Reusability management**—defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
- **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

The Software Engineering process is not rigid---It should be agile and adaptable. Therefore, a process adopted for one project might be significantly different than a process adopted for another project.
Among the differences are:

- Overall flow and level of interdependencies among tasks
- Degree to which work tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor of process description

- Degree to which stakeholders are involved in the project
- Level of autonomy given to project team
- Degree to which team organization and roles are prescribed

1.5 THE SOFTWARE ENGINEERING PRACTICE

Generic concepts and principles that apply to framework activities:

1.5.1 The Essence of Practice

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

These steps lead to series of questions:

Understand the Problem

- Who are the stakeholders?
- What are the **Unknowns**? What functions and features are required to solve the problem?
- Is it possible to create smaller problems that are easier to understand?
- Can a graphic analysis model be created?

Plan the Solution

- Have you seen similar problems before? Is there existing software that implements these features.
- Has a similar problem been solved? ‘
- Can readily solvable sub problems be defined?
- Can a design model be created? –Effective implementation.

Carry Out the Plan

- Does solution conform to the plan? Is source code traceable to the design model?
- Is each solution component provably, correct? Have design and code be reviewed?

Examine the Result

- Is it possible to test each component part of the solution?
- Does the solution produce results that conform to the data, functions, and features required?

1.5.2 Software General Principles

The dictionary defines the word **principle** as "**an important underlying law or assumption** required in a system of thought." David Hooker has Proposed seven principles that focus on software Engineering practice.

The First Principle: The Reason It All Exists
A software system exists for one reason: to provide value to its users.

The Second Principle: KISS (Keep It Simple, Stupid!)
Software design is not a haphazard process. There are many factors to consider in any design effort. All design should be as simple as possible, but no simpler.

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.

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 Fifth Principle: Be Open to the Future A system with a long lifetime has more value. Never design yourself into a corner. Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

The Sixth Principle: Plan Ahead for Reuse Reuse saves time and effort. Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

The Seventh principle: Think! Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right.

1.6 SOFTWARE MYTHS

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

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.

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 this entire question is NO.

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

Reality: Software development is not a mechanistic process like manufacturing. "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 we 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 project internally, it will invariably struggle when it out sources' software project.

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 led to false expectations and ultimately, dissatisfaction with the developers.

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

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

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

Reality: It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, 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 50 years of programming culture. During the early days of software, 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 consumed on software will be consumed 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 formal 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. Documentation provides 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 quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

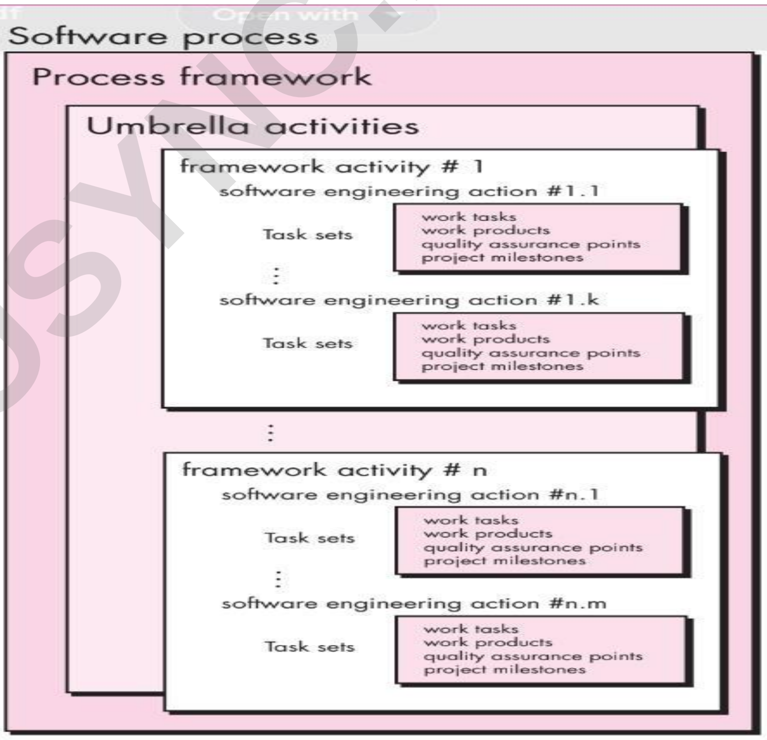
CHAPTER 2 - PROCESS MODELS

2.1 A GENERIC PROCESS MODEL

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in **Figure 2.1.** 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.***

FIGURE 2.1 A software process framework



As I discussed in Chapter 1, a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction, and deployment**. In addition, a set of umbrella activities—**project tracking and control, risk management, quality**

assurance, configuration management, technical reviews, and others—are applied throughout the process.

The important aspect of software process is **"Process Flow"** which describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2.

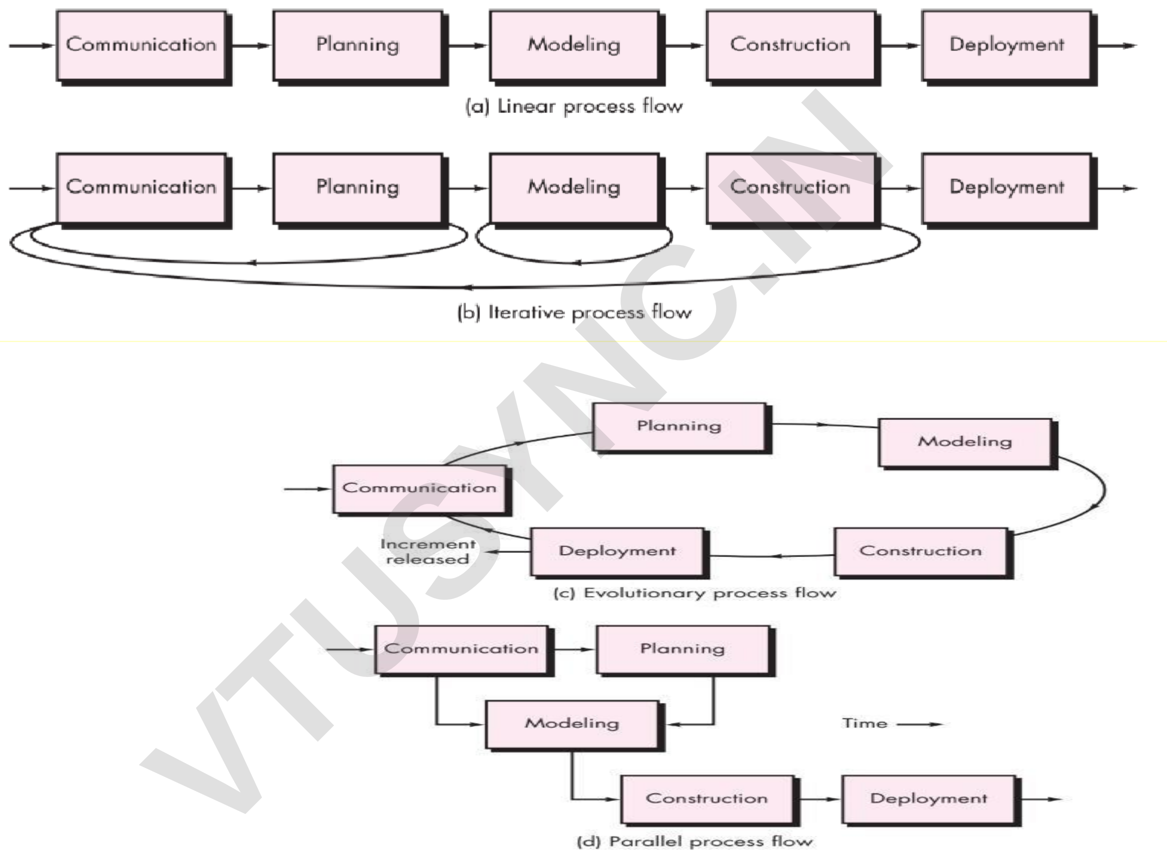


FIGURE 2.2 Process flow

A linear process [Fig:2.2(a)] flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

An iterative process flow [Fig:2.2(b)] repeats one or more of the activities before proceeding to the next.

An evolutionary process flow [Fig:2.2(c)] 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 [Fig:2.2(d)] 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).

2.1.1 Defining a Framework Activity

There are five framework activities, they are

- communication,
- planning,
- modeling,
- construction, and
- deployment.

These five framework activities provide a basic definition of Software Process. These Framework activities provides basic information like ***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?***

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

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

2.1.2 Identifying a Task Set

- Each software engineering action can be represented by a number of different task sets
- Each a collection of software engineering o work tasks,
  - related work products,
  - quality assurance points, and
  - project milestones.
- Choose a task set that best accommodates the needs of the project and the characteristics of software team.
- This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

2.1.3 Process Patterns

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

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

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

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

   I. **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 that are relevant to the stage (framework activity). E.g.: **Establishing Communicatio**n.
   This pattern would incorporate the task pattern **Requirements Gathering** and others.

   II. **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**).

   III. **Phase pattern**—Define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. eg: **Spiral Model** or **Prototyping**.

4. **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?

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

6. **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.

7. **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?

8. **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.

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

Conclusion on Process patterns

- Provide an effective mechanism for addressing problems associated with any software process.
- The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).
- The description is then refined into a set of stage patterns that describe framework activities
- Once process patterns have been developed, they can be reused for the definition of process variants.

2.2 PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs.

Process patterns must be coupled with solid software engineering practice.

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

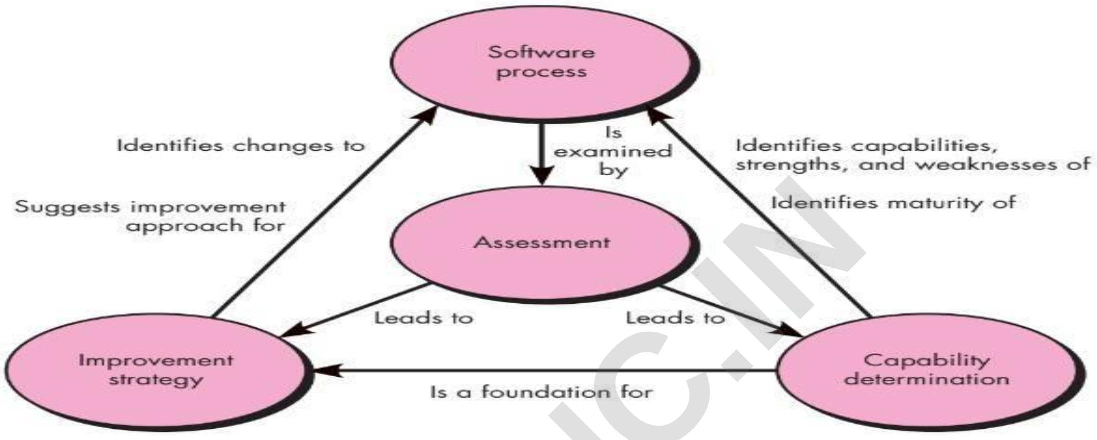
A number of different approaches to software process assessment and improvement have been proposed over the past few decades.

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.



Elements of SPI (Software Process Improvement) framework.

2.3 PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos (disorder) of software development.

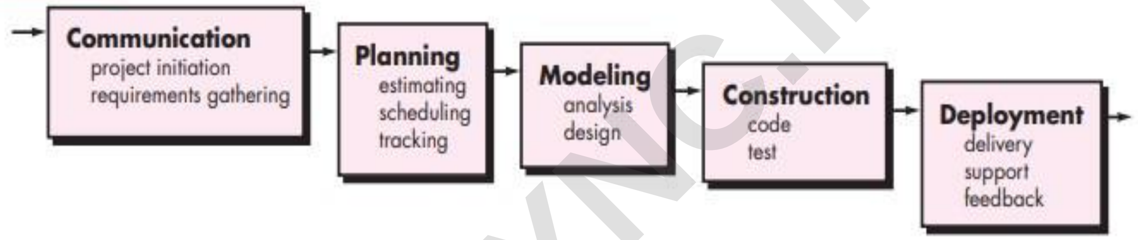
- These models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.
- The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise".
- The edge of chaos can be visualized as an unstable, partially structured state.
- It is unstable because it is constantly attracted to chaos or to absolute order.
- The prescriptive process approach in which order and project consistency are dominant issues.
- "prescriptive" means 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, 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

The waterfall model, **Fig [2.3]** 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.**

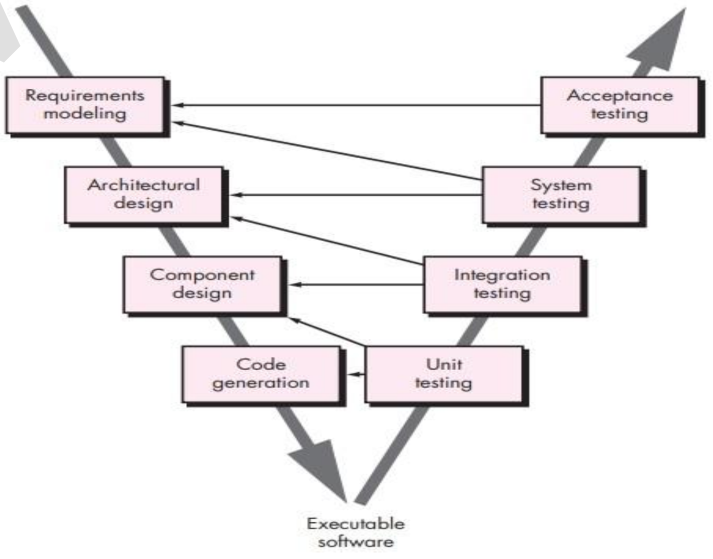
FIGURE 2.3 The waterfall model



V-model

A variation in the representation of the waterfall model is called the **V-model.** represented in the following **Fig:2.4**. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.

FIGURE 2.4
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 a 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 whenever limited number of new development efforts and when requirements are well defined and reasonably stable.

2.4 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. Referring to **Fig 2.5** The *incremental* model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

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

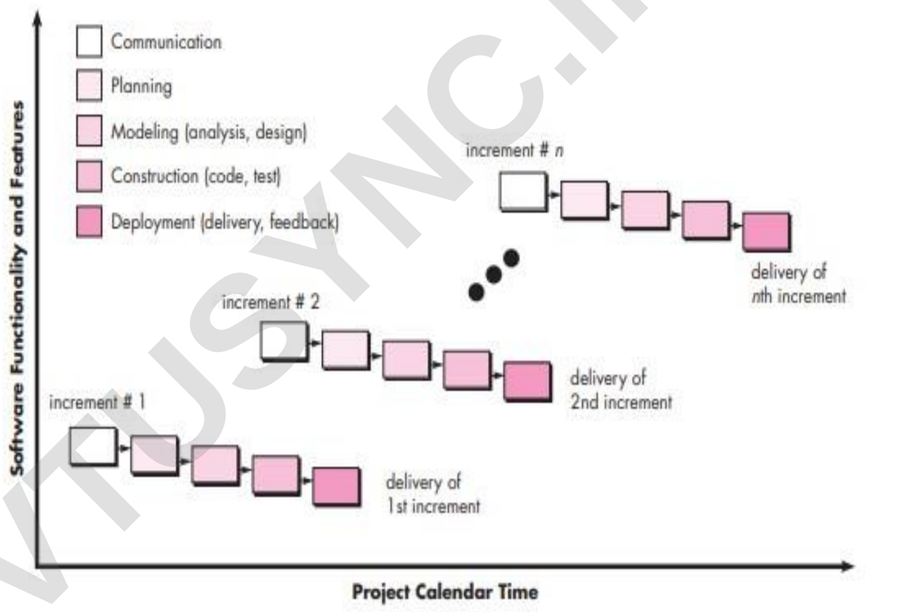
- When an incremental model is used, the first increment is often a **core product**. That is, basic requirements are addressed but many extra features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment.

- 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. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage **technical risks.**

FIGURE 2.5

The incremental model



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

2.5 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.

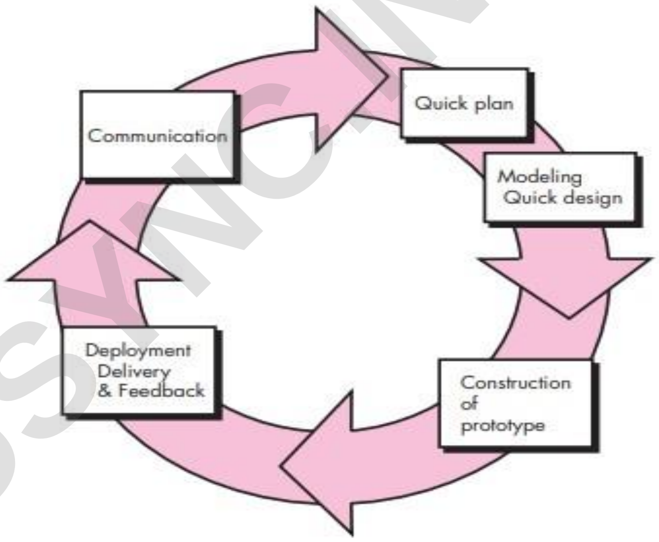
2.5.1 Prototyping Model

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

FIGURE 2.6
The prototyping paradigm



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 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 randomly, 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.

2.5.2 The Spiral Model:

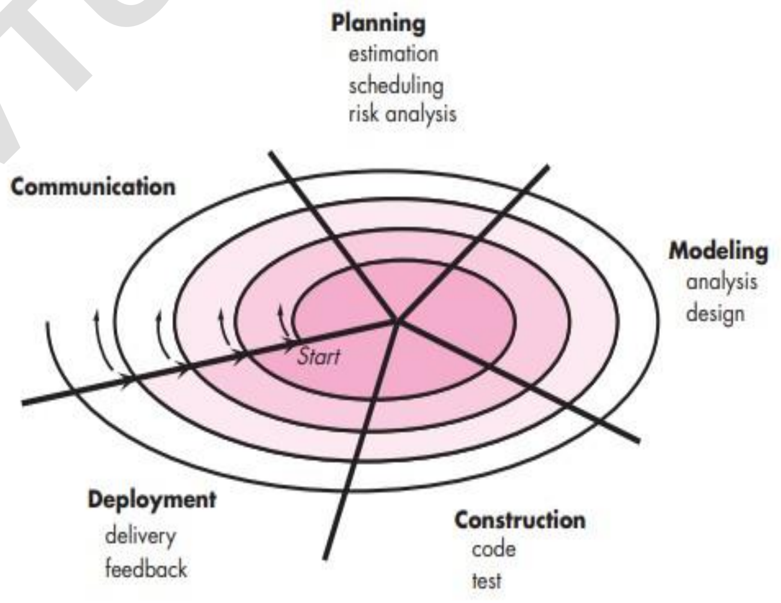
Originally proposed by Barry Boehm, the spiral model is an **evolutionary software process model** that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions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

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

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

FIGURE 2.7

A typical spiral model



A spiral model is divided into a set of framework activities defined by the software engineering team.**[Fig:2.7]** As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.

Risk is considered as each revolution is made. Anchor point milestones 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."**

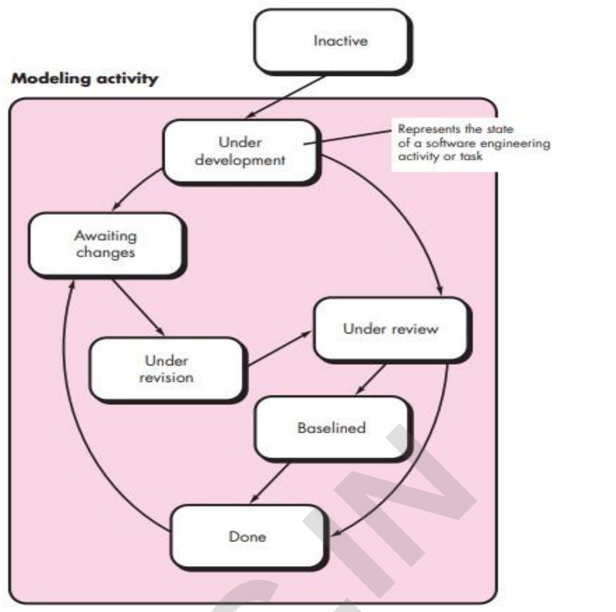
The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. It maintains the systematic stepwise approach suggested by the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

2.6 CONCURRENT MODELS

The concurrent development model sometimes called as concurrent engineering can be represented schematically as a series of framework activities, actions, tasks and their associated rules.

The Fig 2.8 represents on Software Engineering activity within the modelling activity using a concurrent model approach.

FIGURE 2.8
One element of the concurrent process model



The activity modelling may be in any one of the states noted at any given time, similarly other activities, actions or tasks (Communication, Construction) can be represented in analogous manner.

All Software Engineering activities exist concurrently but reside in different states. E.g. Early in a project the communication activity has completed its 1st iteration and exists in the **awaiting changes** state.

The **modelling activity** (which existed in **inactive state**) while initial communication was completed now make a transition into the **under-development state.**

If however, the customer indicates that changes in requirement must be made, the modelling activity moves from under-development state to awaiting changes state.

Concurrent modelling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks.

Concurrent modelling is applicable for all types of software development and provides an accurate picture of the current state of the project.

2.7 SPECIALIZED PROCESS MODELS

2.7.1 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.

2.7.2 The Formal Methods Model

- The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software.

- Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called ***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 engineering paradigms. 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 which discover and correct errors that might otherwise go undetected. The formal methods model offers the promise of 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 formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for Technically unsophisticated customers.

2.7.3 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.