



A T M E
College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

SOFTWARE ENGINEERING AND PROJECT MANAGEMENT- BCS501



MODULE-1

10 hours

Software and Software Engineering: The nature of Software, The unique nature of WebApps, Software Engineering, The software Process, Software Engineering Practice, Software Myths.

Process Models: A generic process model, Process assessment and improvement, Prescriptive process models: Waterfall model, Incremental process models, Evolutionary process models, Concurrent models, Specialized process models. Unified Process , Personal and Team process models

Textbook 1: Chapter 1: 1.1 to 1.6, Chapter 2: 2.1 to 2.5

Topics to be covered in

1. The Nature of Software

1.1 Defining Software

1.2 Software Application Domains

1.3 Legacy Software

1. The Nature of Software

- Software takes on a dual role.
 - It is a **product**, and at the same time,
 - the **vehicle** for delivering a product.
- ❖ As a **product**, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware
- ❖ As the **vehicle** used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

- ❖ Software delivers the most important product of our **time—information**
- Software **manages** business information to enhance competitiveness;
- Software provides a **gateway** to worldwide information networks (e.g., the Internet).
- Software provides the means for **acquiring information** in all of its forms.
- Software role has **undergone significant change** over the last half-century.
- Software industry has become **a dominant factor in the industrialized world**

1.1.1 Defining Software

A textbook description of software might take the following form:

Software is:

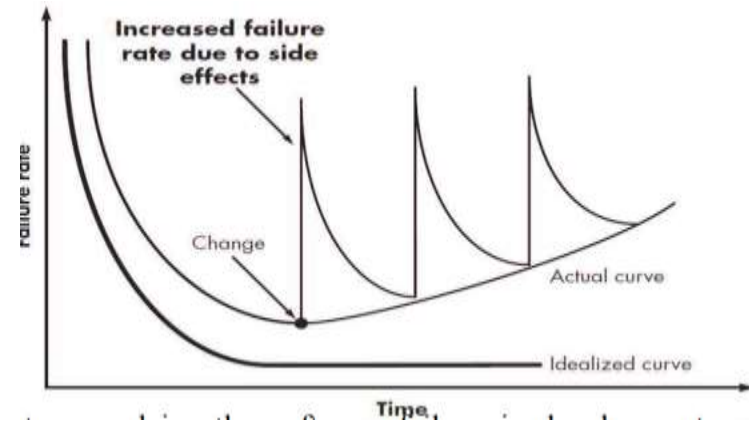
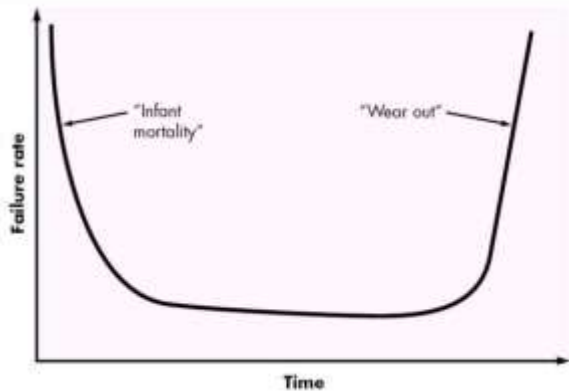
- (1) instructions (computer programs) that when executed provide desired features, function, and performance;
- (2) data structures that enable the programs to adequately manipulate information
- (3) descriptive information in both hard copy and virtual forms that describes the operation and use of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

Software characteristics

- 1. Software is developed or engineered; it is not manufactured in the classical sense.***
- 2. Software doesn't “wear out.”***
 1. Bath tub curve
 2. Idealized curve
- 3. Although the industry is moving toward component-based construction, most software continues to be custom built***
 1. reuse

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

1. “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life
2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software.



3. Another aspect of wear illustrates the difference between hardware and software.

1.2 Software Application Domains

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

1. System software

a collection of programs written to service other programs

2. Application software

stand-alone programs that solve a specific business need.

3. Engineering/scientific software

has been characterized by “number crunching” algorithms

4. 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.

5.Product-line software

designed to provide a specific capability for use by many different customers.

6.Web applications

called “WebApps,” this network-centric software category spans a wide array of applications.

7.Artificial intelligence software

makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis.

New Challenges

1. Open-world computing
2. Net Sourcing
3. Open source

1.1.3 Legacy Software

Legacy software systems :were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms.

As time passes, legacy systems often evolve for one or more of the following reasons:

1. The software must be adapted to meet the needs of new computing environments or technology.
2. The software must be enhanced to implement new business requirements.
3. The software must be extended to make it interoperable with other more modern systems or databases.
4. The software must be re-architected to make it viable within a network environment.

The Unique Nature of WebApps

The following attributes are encountered in the vast majority of WebApps

1. Network intensiveness
2. Concurrency
3. Unpredictable load
4. Performance
5. Availability
6. Data driven.
7. Content sensitive
8. Continuous evolution
9. Immediacy
10. Security
11. Aesthetics

1.3 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

1. Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application has grown dramatically.

➤ It follows that a concerted effort should be made to **understand the problem** before a software solution is developed.

2. The information technology requirements demanded by individuals, businesses, and governments grow increasingly complex with each passing year.

➤ It follows that **design** becomes a pivotal activity.

3. Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control.

➤ It follows that software should exhibit **high quality**.

4. As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow.

➤ It follows that software should be **maintainable**.

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by **Fritz Bauer** at the seminal conference on the subject still serves as a basis for discussion:

“[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines”

The IEEE has developed a more comprehensive definition when it states:

Software Engineering:

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

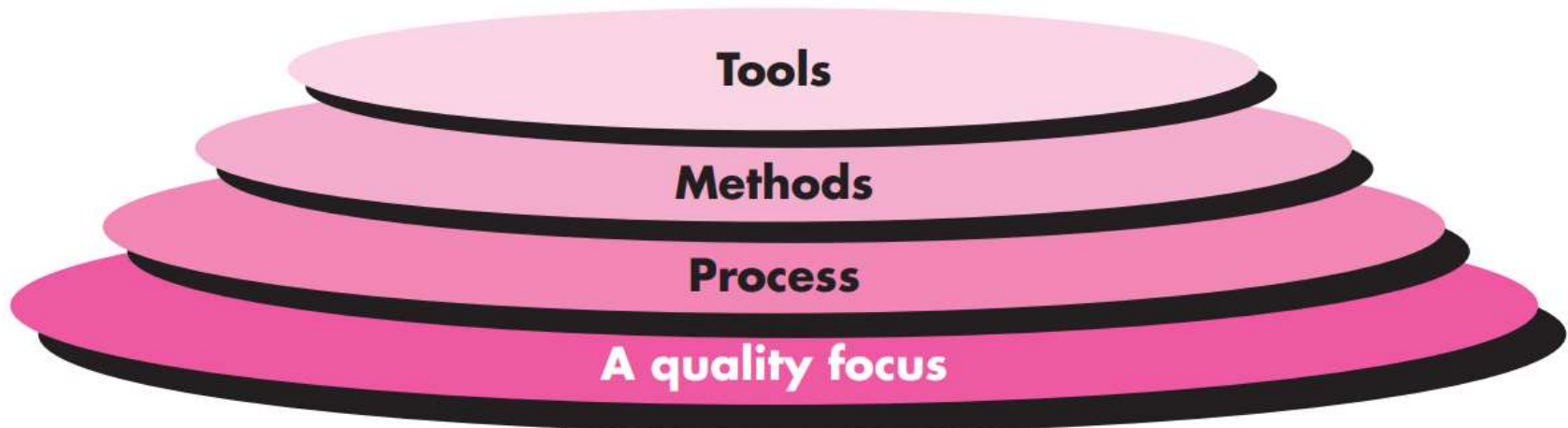
Software Engineering layers

❖ Software engineering is a layered technology.

1. Quality

❖ Any engineering approach must rest on an organizational commitment to *quality*.

❖ Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this



2.Process

- ❑ 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.

3.Methods

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

4.Tools

- ❑ 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** (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 generic process framework for software engineering encompasses five activities

1. Communication

Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders)

2.Planning

“map” that helps guide the team as it makes the journey

3.Modeling

sketch

4.Construction.

code generation and the testing

5.Deployment.

delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

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

Typical umbrella activities include:

1. **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.
2. **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.
3. **Software quality assurance**—defines and conducts the activities required to ensure software quality.
4. **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

- 5.**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.
- 6.**Software configuration management**—manages the effects of change throughout the software process.
- 7.**Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
- 8.**Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

A process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and 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 with which the process is described
- Degree to which the customer and other stakeholders are involved with the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

1.5.1 The Essence of Practice

George Polya outlined the essence of problem solving, and consequently, the essence of software engineering practice

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

1.Understand the problem

It's worth spending a little time answering a few simple questions:

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

2. Plan the solution.

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

- Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can sub problems be defined? If so, are solutions readily apparent for the sub problems?
- Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

3. Carry out the plan.

the “plan” will allow you to proceed without getting lost.

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

4.Examine the result.

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

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

1.5.2 General Principles

principles help you establish a mind-set for solid software engineering practice. David Hooker has proposed seven principles that focus on software engineering practice as a whole.

1. The First Principle: The Reason It All Exists
to provide value to its users.
2. The Second Principle: (Keep It Simple, Stupid!)
3. The Third Principle: Maintain the Vision
- 4.The Fourth Principle: What You Produce, Others Will
Consume
5. The Fifth Principle: Be Open to the Future
6. The Sixth Principle: Plan Ahead for Reuse
7. The Seventh principle: Think!

1. 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.

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

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

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

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks “adding people to a late software project makes it later.” As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2.Customer myths.

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

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

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

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

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

3.Practitioner's myths

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

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

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

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

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

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

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

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

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

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.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

A software process framework

Software process

Process framework

Umbrella activities

framework activity # 1

software engineering action #1.1

Task sets

⋮

software engineering action #1.k

Task sets

work tasks
work products
quality assurance points
project milestones

work tasks
work products
quality assurance points
project milestones

⋮

framework activity # n

software engineering action #n.1

Task sets

⋮

software engineering action #n.m

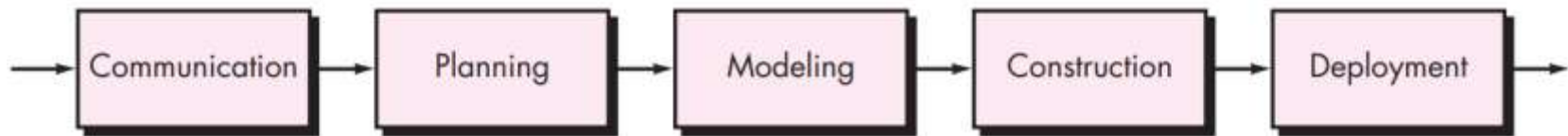
Task sets

work tasks
work products
quality assurance points
project milestones

work tasks
work products
quality assurance points
project milestones

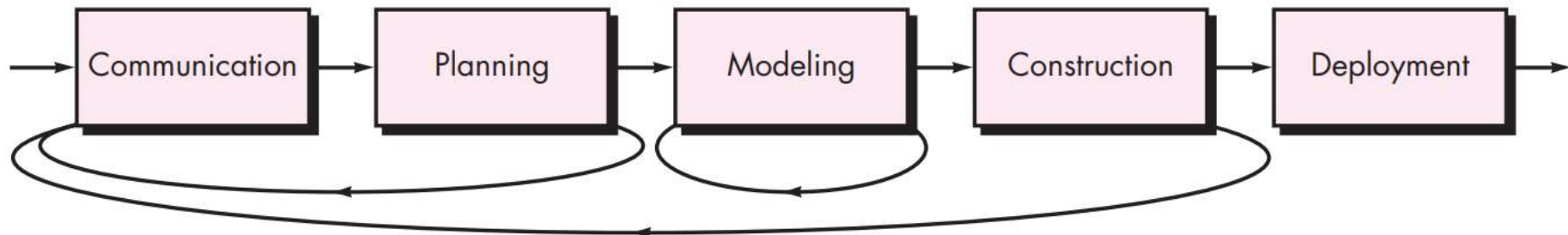
Process flow

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



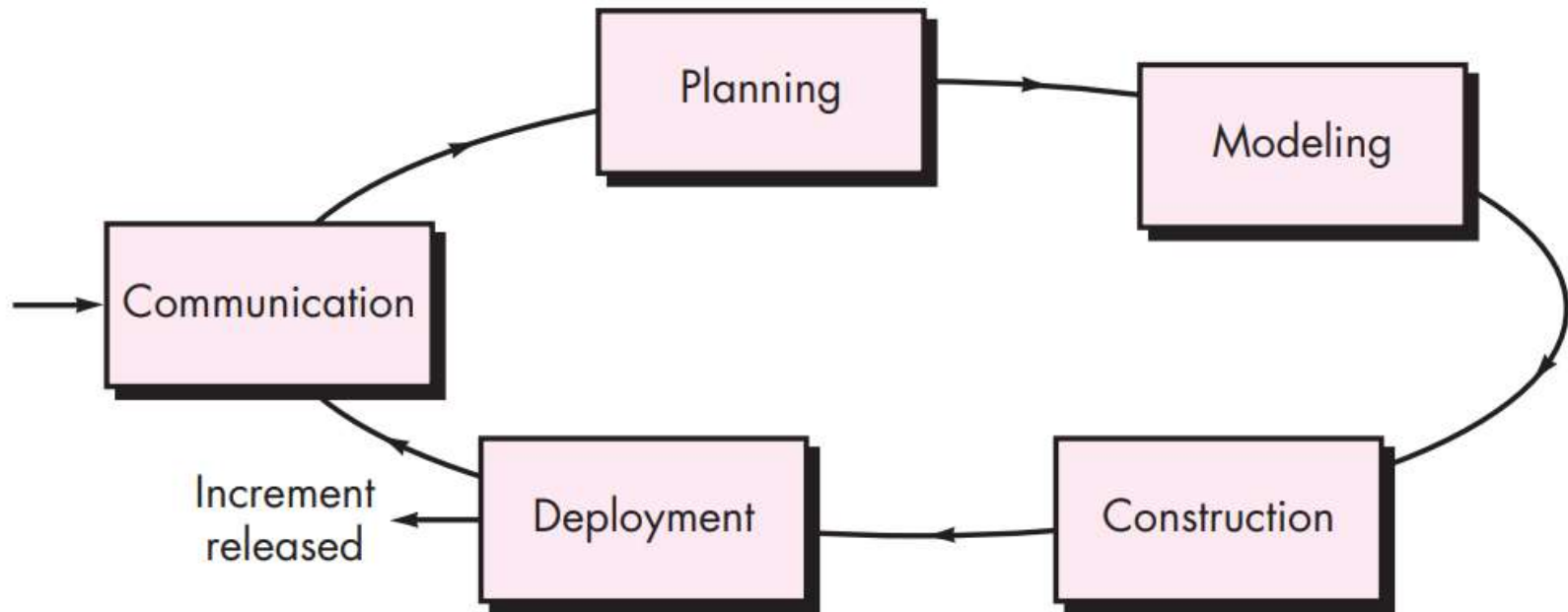
(a) Linear process flow

An iterative process flow repeats one or more of the activities before proceeding to the next



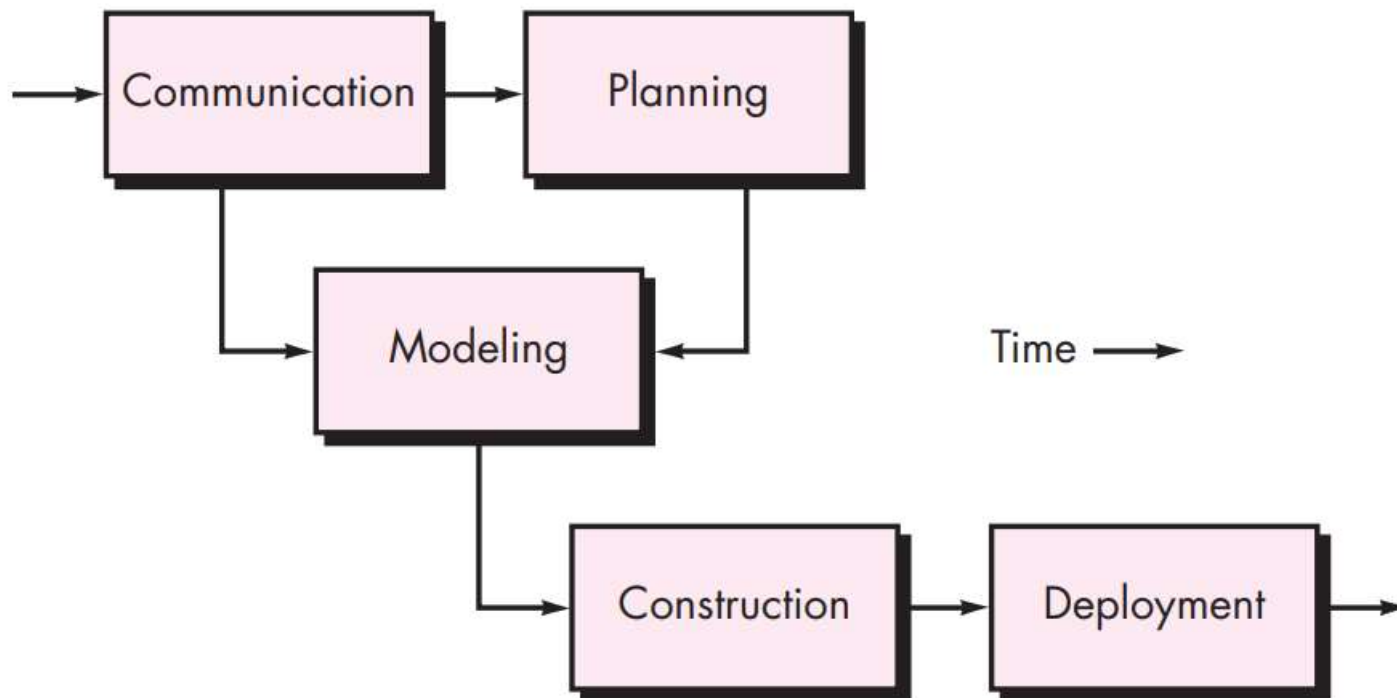
(b) Iterative process flow

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



(c) Evolutionary process flow

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)



(d) Parallel process flow

2.1.1 Defining a Framework Activity

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is phone conversation, and the work tasks (the task set) that this action encompasses are:

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

2.1.2 Identifying a Task Set

- Each software engineering action can be represented by a number of different **tasksets** —each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.
- We should choose a task set that best accommodates the needs of the project and the characteristics of our 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 pattern¹ 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.
- Patterns can be defined at any level of abstraction.² In some cases, 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 [Amb98] has proposed a template for describing a process pattern:

- Pattern Name
- Forces.
- Type
 - Stage pattern
 - Task pattern
 - Phase pattern
- Initial context.
- Problem.
- Solution.
- Resulting Context.
- Related Patterns.
- Known Uses and Examples

2.2 PROCESS ASSESSMENT AND IMPROVEMENT

- Process patterns must be coupled with solid software engineering practice
- The process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering

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:
 1. Initiating
 2. Diagnosing
 3. Establishing
 4. Acting
 5. learning.

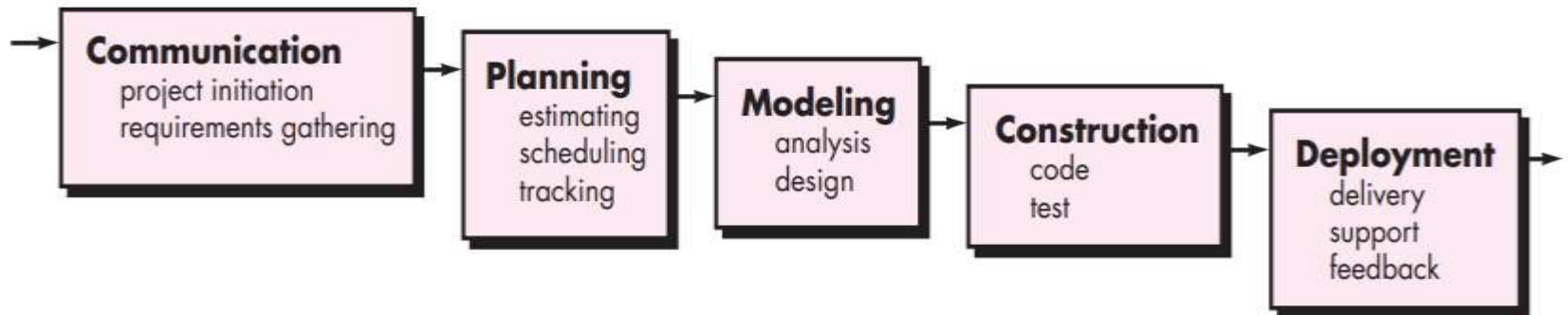
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment
- **SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software 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.

2.3 PRESCRIPTIVE PROCESS MODELS

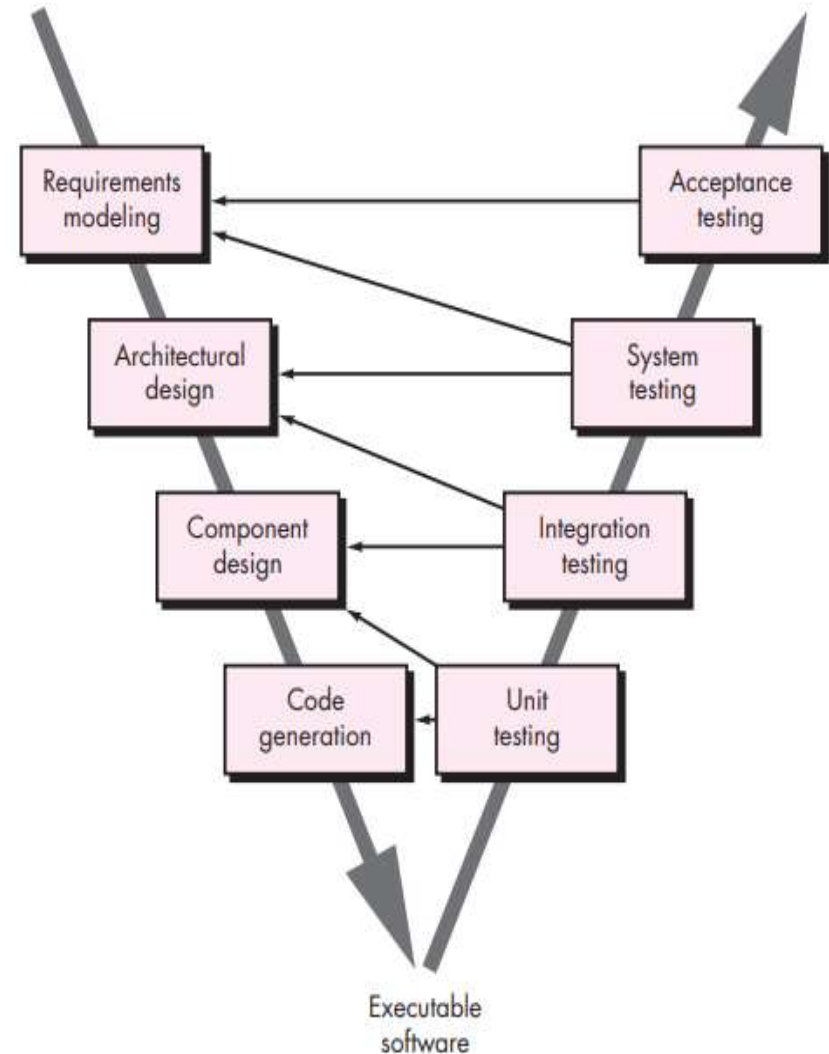
- Is called “prescriptive” because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project.
- Each process model also prescribes a process flow (work flow)—that is, the manner in which the process elements are interrelated to one another.

2.3.1 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,



A variation in the representation of the waterfall model is called the V-model. Represented in Figure



DISADVANTAGES OF WATERFALL MODEL

- 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.
 - 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.
 - 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
- “blocking states”

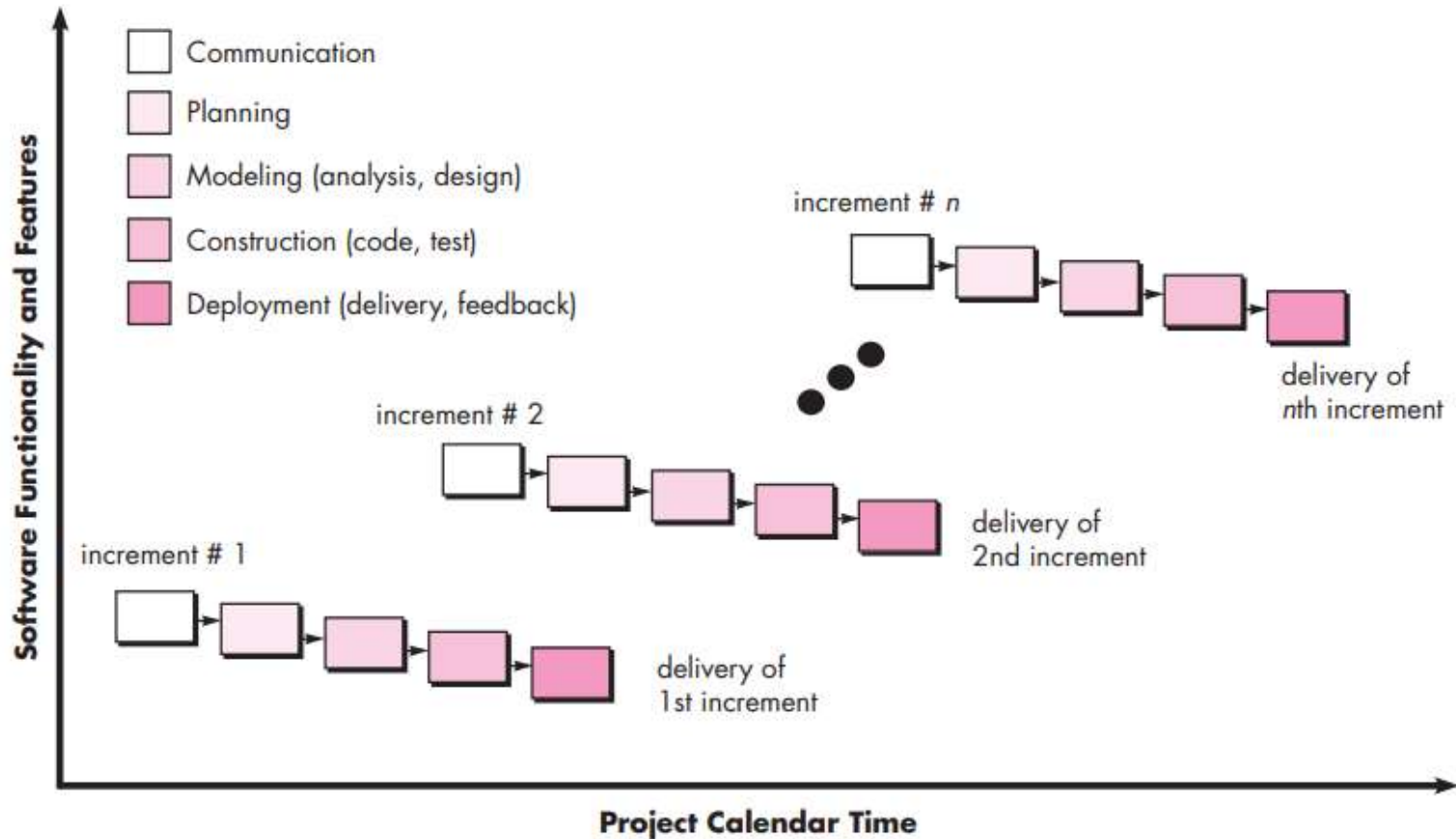
2.3.2 Incremental Process Models

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

- For example

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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



- 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 .
- 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.

ADVANTAGES

- 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 can be added to implement the next increment
- Increments can be planned to manage technical risks.
- For example,
 - a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

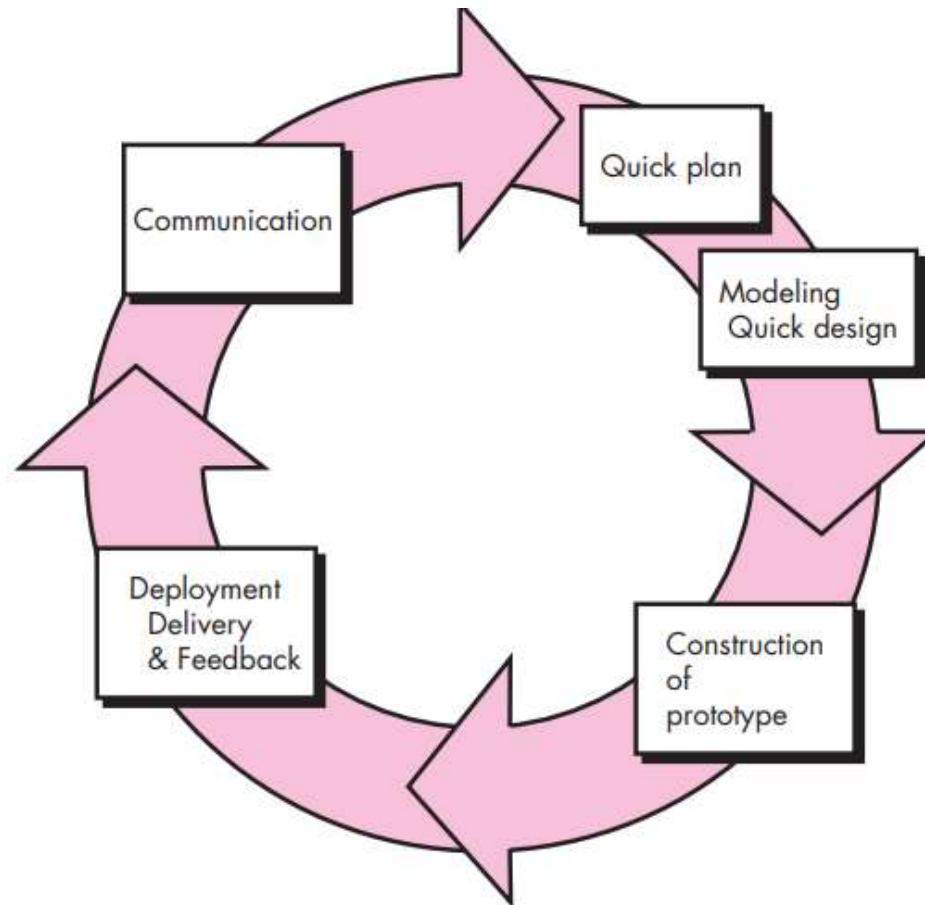
2.3.3 Evolutionary Process Model

- A process model that has been explicitly designed to accommodate a product that evolves over time.
- 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.
 - The Spiral Model.

Prototyping.

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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



- 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.
- Ideally, 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.

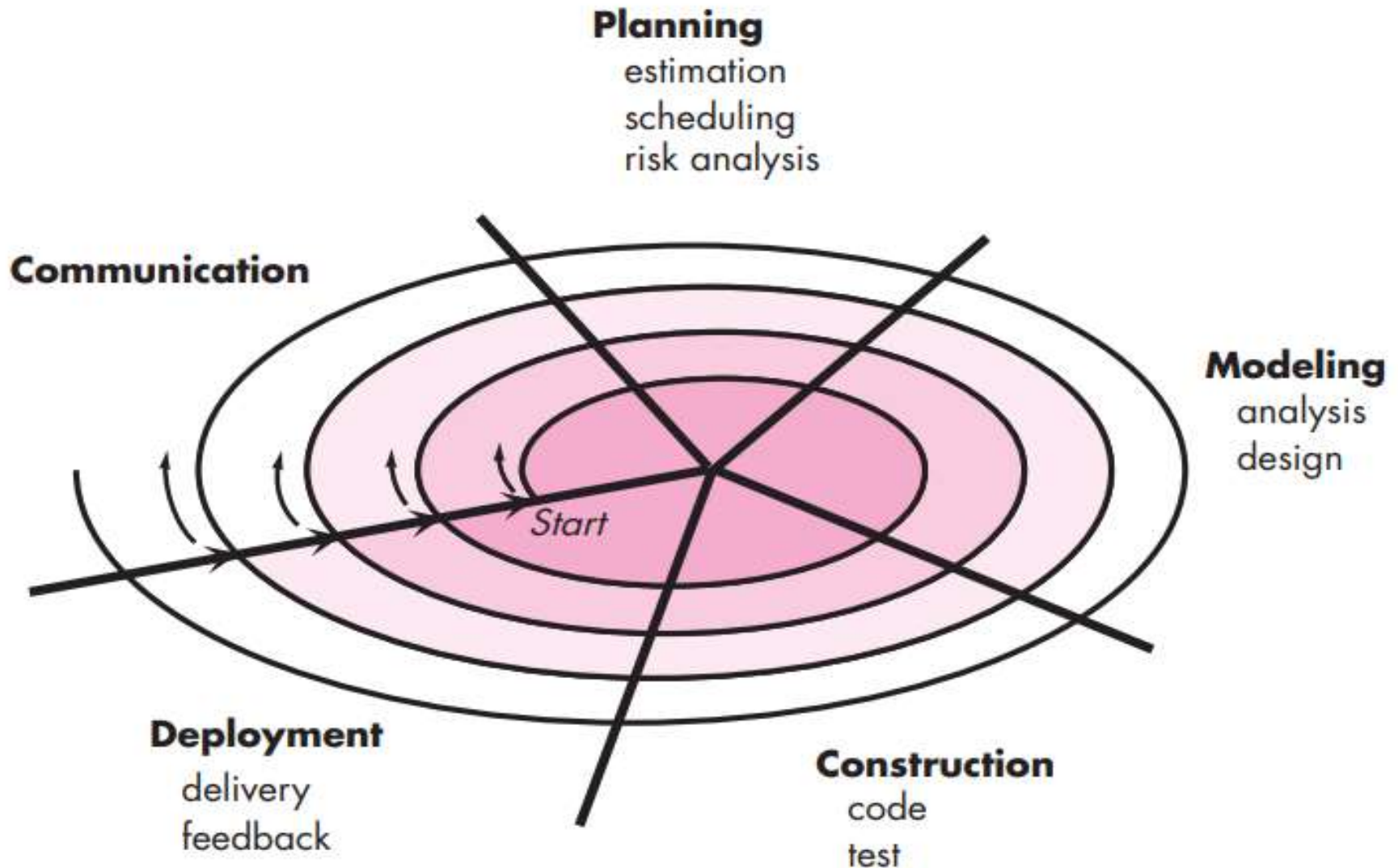
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

- 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. Too often, software development management relents.
- 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. 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.
- 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.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



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

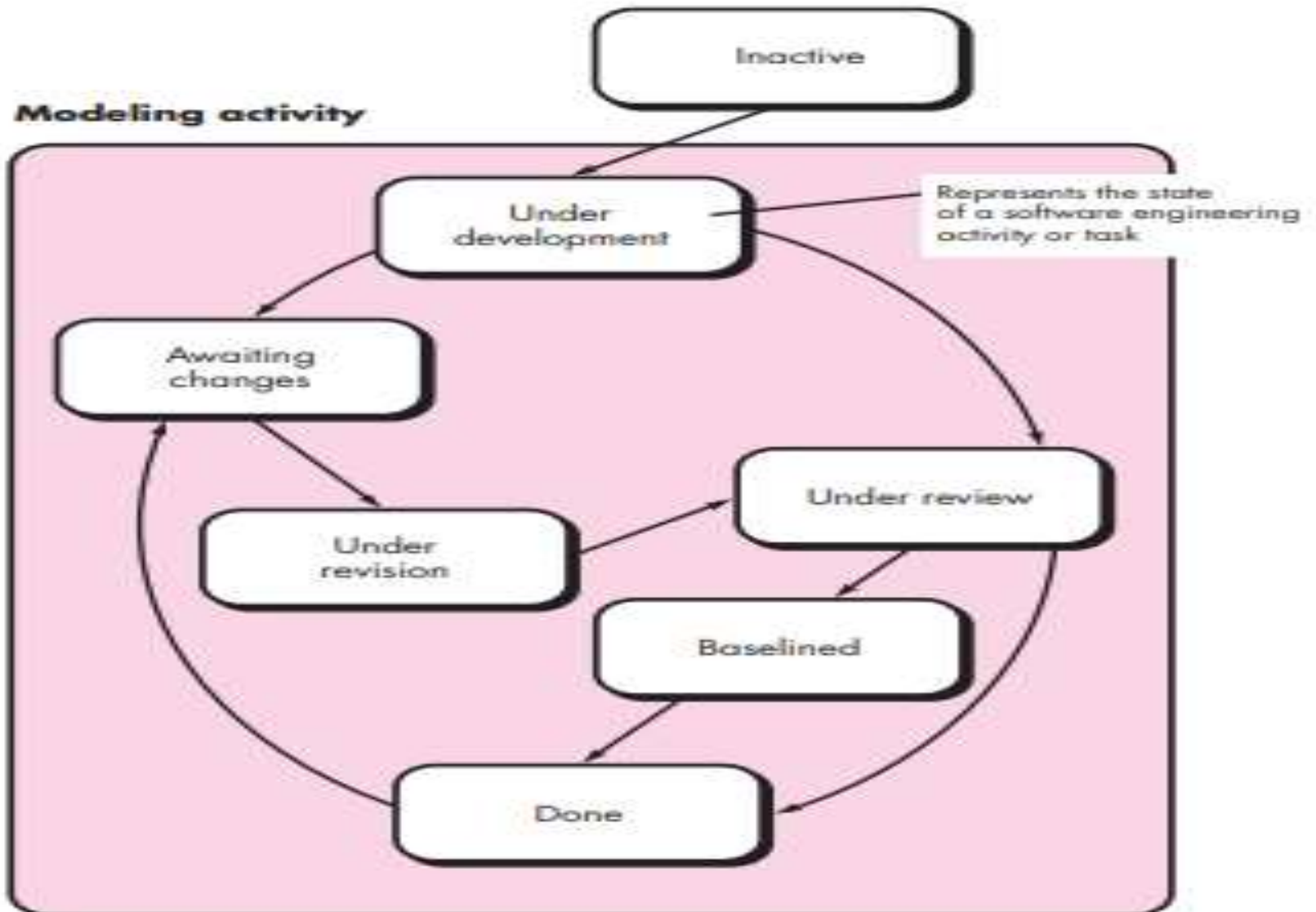
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery.
- In addition, the project manager adjusts the planned number of iterations required to complete the software
- The spiral model is a realistic approach to the development of large-scale systems and software.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

2.3.4 Concurrent Models

- The concurrent development model, sometimes called **concurrent engineering**, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



- 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.
- other activities, actions, or tasks can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

- For example,
- Early in a project the communication activity has completed its first iteration and exists in the **awaiting changes state**.
- The modeling activity which existed in the **inactive state** while initial communication was completed, now makes a transition into the **under development state**. However, the customer indicates that changes in requirements must be made, the modeling activity moves from the under development 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.

2.3.5 A Final Word on Evolutionary Processes

Advantages

- Modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction.
- In many cases, time-to-market is the most important management requirement. If a market window is missed, the software project itself may be meaningless. Evolutionary process models were conceived to address these issues.

Disadvantages

- Prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.
- If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected
- Third, software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

2.4 SPECIALIZED PROCESS MODELS

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

- 1 Component-Based Development
- 2 The Formal Methods Model
- 3 Aspect-Oriented Software Development

2.4.1 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

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

2.4.2 The Formal Methods Model

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

Disadvantages

- 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.

2.4.3 Aspect-Oriented Software Development

- 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—“mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern”

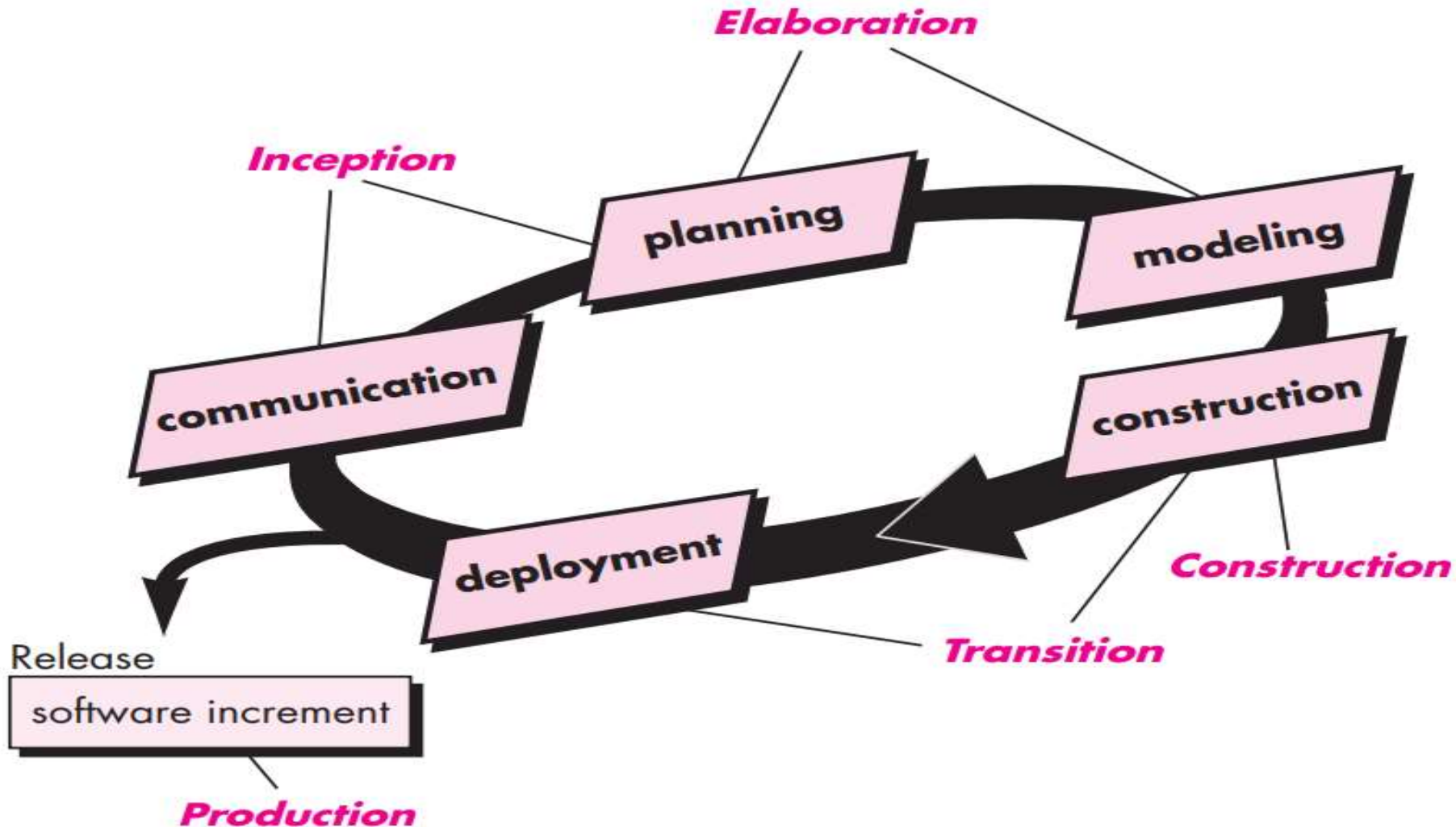
2.5 THE UNIFIED PROCESS

- ❖ The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system .
- ❖ It emphasizes the important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse” .
- ❖ It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development

2.5.1 A Brief History

- ❖ 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 in object-oriented modeling.
- ❖ By 1997, UML became a de facto industry standard for object-oriented software development.
- ❖ Over the next few years, Jacobson, Rumbaugh, and Booch developed the ***Unified Process***, a framework for object-oriented software engineering using UML.
- ❖ Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.

2.5.2 Phases of the Unified Process



1.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.
- Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires.
- Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them.
- Later, the architecture will be refined and expanded into a set of models that will represent different views of the system.
- Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed

2.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 part of the inception phase and expands the architectural representation to include five different views of the software—
 1. The use case model
 2. The requirements model
 3. The design model
 4. The implementation model
 5. The deployment model.
- In some cases, elaboration creates an “executable architectural baseline” that represents a “first cut” executable system.
- The plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable.
- Modifications to the plan are often made at this time.

3. 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 acquires the software components that will make each use case operational for end users.
- To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.
- All necessary and required features and functions for the software increment are then implemented in source code.
- As components are being implemented, unit tests are designed and executed for each. In addition, integration activities are conducted.
- Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

4. 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 activity.
- Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
- In addition, the software team creates the necessary support information that is required for the release.
- At the conclusion of the transition phase, the software increment becomes a usable software release.

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

2.6 Personal and Team process models

Software Process Customization

- **Best Process is People-Centric:**

The most effective software process is one that aligns closely with the individuals who are performing the work.

- **Corporate-Level Processes Need Flexibility:**
Organization-wide process models are only effective when they allow **adaptation to the specific needs** of project teams

- **Deal Scenario – Tailored Process:**

Ideally, a software process should be custom-built to meet the **specific requirements of a team**, while also fulfilling **organizational expectations**.

- **Team-Created Processes:**

Teams can develop their **own tailored processes** that serve:

- The **individual needs** of team members
- The **collective needs** of the organization

• **Watts Humphrey's Contribution:**

- Proposed the concept of:
 - **Personal Software Process (PSP)**
 - **Team Software Process (TSP)**
- Emphasized that both approaches are:
 - **Achievable**
 - Require **dedication, training, and coordination**

• **Balance is Crucial:**

A good software process balances:

- Individual developer productivity
- Team collaboration
- Organizational goals and standards