LECTURE NOTES

ON

SOFTWARE ENGINEERING

III B. Tech I semester (JNTUH-R18)

UNIT-I
INTRODUCTION TO SOFTWARE ENGINEERING

Software: Software is

(1) Instructions (computer programs) that provide desired features, function, and performance, when executed

(2) Data structures that enable the programs to adequately manipulate information,

(3) Documents that describe the operation and use of the programs.

Characteristics of Software:

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

(2) Software does not "wear out"

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

Software Engineering:

(1) The 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)

EVOLVING ROLE OF SOFTWARE:

Software takes dual role. It is both a **product** and a **vehicle** for delivering a product.

As a **product**: It delivers the computing potential embodied by computer Hardware or by a network of computers.

As a **vehicle**: It is information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as single bit or as complex as a multimedia presentation. Software delivers the most important product of our time-information.

- It transforms personal data
- It manages business information to enhance competitiveness
- It provides a gateway to worldwide information networks
- It provides the means for acquiring information

The role of computer software has undergone significant change over a span of little more than 50 years

- Dramatic Improvements in hardware performance
- Vast increases in memory and storage capacity
- A wide variety of exotic input and output options

1970s and 1980s:

- *Osborne* characterized a "new industrial revolution"
- *Toffler* called the advent of microelectronics part of "the third wave of change" in human history
- *Naisbitt* predicted the transformation from an industrial society to an "information society"
- *Feigenbaum and McCorduck* suggested that information and knowledge would be the focal point for power in the twenty-first century
- *Stoll* argued that the "electronic community" created by networks and software was the key to knowledge interchange throughout the world

1990s began:

- *Toffier* described a "power shift" in which old power structures disintegrate as computers and software lead to a "democratization of knowledge".
- *Yourdon* worried that U.S companies might lose their competitive edge in software related business and predicted "the decline and fall of the American programmer".
- *Hammer and Champy* argued that information technologies were to play a pivotal role in the "reengineering of the corporation".

Mid-1990s:

- The pervasiveness of computers and software spawned a rash of books by neo-luddites.

Later 1990s:

- *Yourdon* reevaluated the prospects of the software professional and suggested "the rise and resurrection" of the American programmer.
- The impact of the Y2K "time bomb" was at the end of 20th century

2000s progressed:

- *Johnson* discussed the power of "emergence" a phenomenon that explains what happens when interconnections among relatively simple entities result in a system that "self-organizes to form more intelligent, more adaptive behavior".
- *Yourdon* revisited the tragic events of 9/11 to discuss the continuing impact of global terrorism on the IT community
- *Wolfram* presented a treatise on a "new kind of science" that posits a unifying theory based primarily on sophisticated software simulations
- *Daconta* and his colleagues discussed the evolution of "the semantic web".

Today a huge software industry has become a dominant factor in the economies of the industrialized world.

THE CHANGING NATURE OF SOFTWARE:

The 7 broad categories of computer software present continuing challenges for software engineers:

1) System software
2) Application software
3) Engineering/scientific software
4) Embedded software
5) Product-line software
6) Web-applications
7) Artificial intelligence software.

- **System software:** System software is a collection of programs written to service other programs. The systems software is characterized by
  - heavy interaction with computer hardware
  - heavy usage by multiple users
  - concurrent operation that requires scheduling, resource sharing, and sophisticated process management
  - complex data structures
  - multiple external interfaces

  E.g. compilers, editors and file management utilities.

- **Application software:**
  - Application software consists of standalone programs that solve a specific business need.
  - It facilitates business operations or management/technical decision making.
  - It is used to control business functions in real-time

  E.g. point-of-sale transaction processing, real-time manufacturing process control.

- **Engineering/Scientific software:** Engineering and scientific applications range
  - from astronomy to volcanology
  - from automotive stress analysis to space shuttle orbital dynamics
  - from molecular biology to automated manufacturing

  E.g. computer aided design, system simulation and other interactive applications.

- **Embedded software:**
  - 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.
  - It can perform limited and esoteric functions or provide significant function and control capability.

E.g. Digital functions in automobile, dashboard displays, braking systems etc.

- **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 market place or address mass consumer markets

  E.g. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications

- **Web-applications:** WebApps are evolving into sophisticated computing environments that not only provide standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.
- **Artificial intelligence software:** AI software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Application within this area includes robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

The following are the **new challenges** on the horizon:

- **Ubiquitous computing**
- **Netsourcing**
- **Open source**
- **The "new economy"**

Ubiquitous computing: The **challenge** for software engineers will be to develop systems and application software that will allow small devices, personal computers and enterprise system to communicate across vast networks.

Net sourcing: The **challenge** for software engineers is to architect simple and sophisticated applications that provide benefit to targeted end-user market worldwide.

Open Source: The **challenge** for software engineers is to build source that is self descriptive but more importantly to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

The "new economy": The **challenge** for software engineers is to build applications that will facilitate mass communication and mass product distribution.

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.

Management myths: Manages 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 - Wont 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?

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

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spend 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 built 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.

Customer myths: The customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin with 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 recipe for disaster.

Myth: Project 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 and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths: Myths that are still believed by software practitioners: during the early days of software, programming was viewed as an art from old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our jobs are 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: The only deliverable work product for a successful project is the working program.

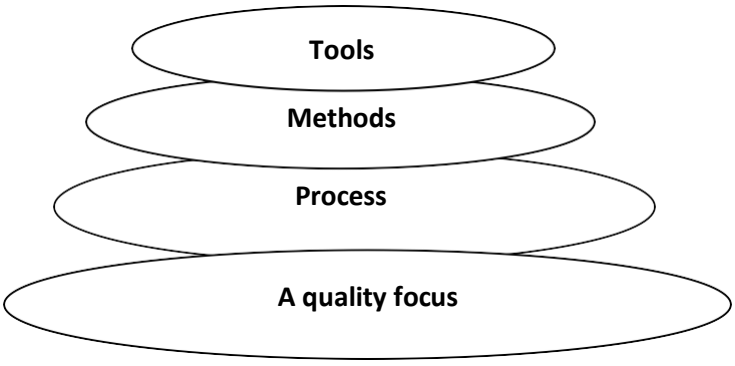
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides guidance for software support.

Myth: software engineering will make us create voluminous and unnecessary documentation and will invariably slows 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.

A GENERIC VIEW OF PROCESS

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:



Software Engineering Layers

Software engineering is a layered technology. 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. Software engineering process is the glue that holds the technology layers. **Process defines a framework that must be established for effective delivery of software engineering technology**.

The software forms the basis for management control of software projects and establishes the context in which

- technical methods are applied,
- work products are produced,
- milestones are established,
- quality is ensured,
- And change is properly managed.

Software engineering methods rely on a set of basic principles that govern area of the technology and include modeling activities.

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.

A PROCESS FRAMEWORK:

- **Software process** must be established for effective delivery of software engineering technology.
- A **process framework** establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- The process framework encompasses a **set of umbrella activities** that are applicable across the entire software process.
- Each **framework activity** is populated by a set of software engineering actions
- Each **software engineering action** is represented by a number of different task sets- each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

In brief
"A **process** defines who is doing what, when, and how to reach a certain goal."

A **Process Framework**

- establishes the foundation for a complete software process
- identifies a small number of **framework activities**
    - applies to all s/w projects, regardless of size/complexity.
- also, set of **umbrella activities**
    - applicable across entire s/w process.

- Each **framework activity** has
    - set of **s/w engineering actions**.
- Each **s/w engineering action** (e.g., design) has

SOFTWARE ENGINEERING

- collection of related **tasks (**called **task sets):**

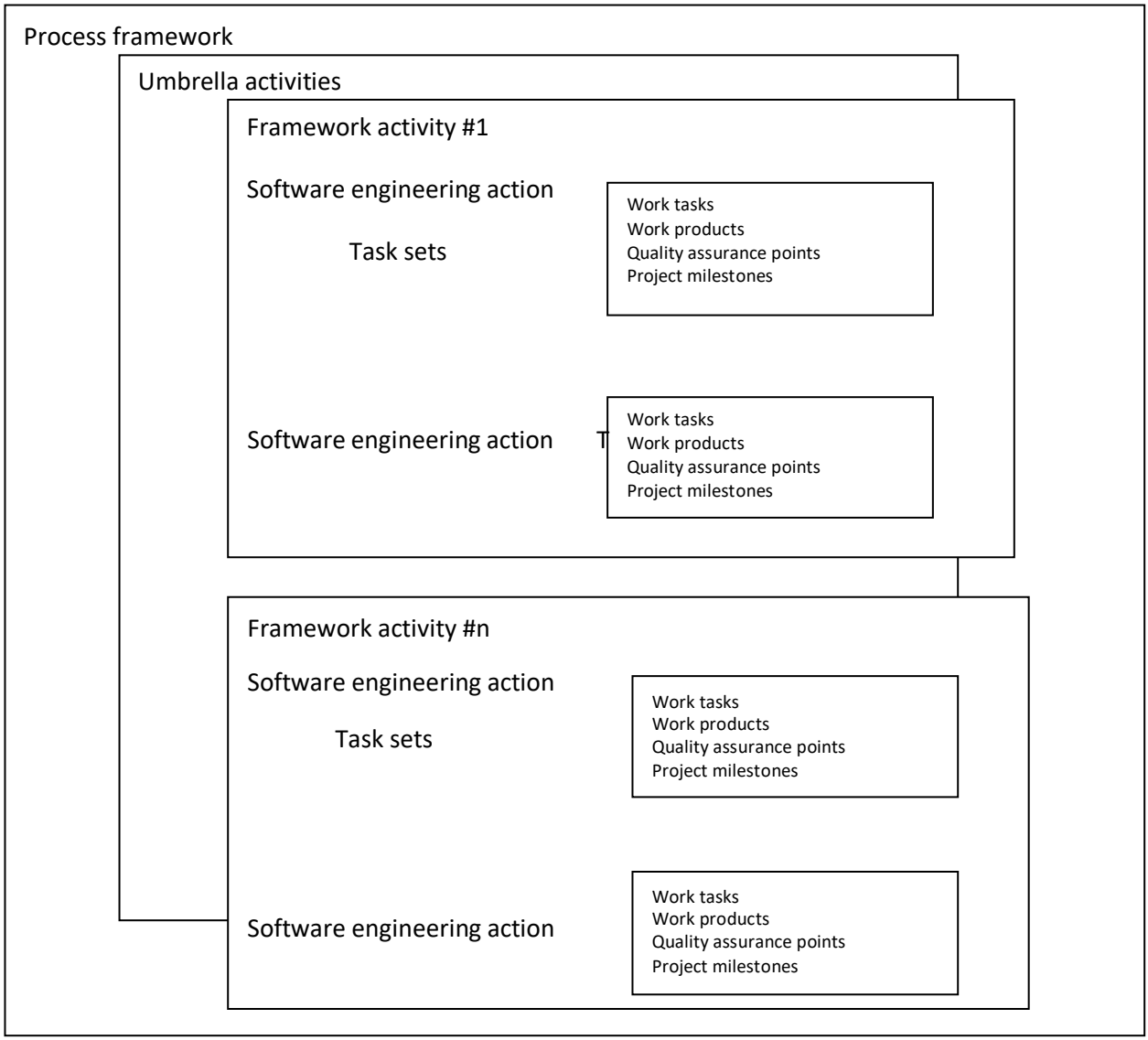
work tasks

work products (deliverables)

quality assurance points

project milestones.

Software process



\

Generic Process Framework: It is applicable to the vast majority of software projects

- Communication activity
- Planning activity
- Modeling activity
    - analysis action
        - requirements gathering work task
        - elaboration work task
        - negotiation work task
        - specification work task
        - validation work task
    - design action
        - data design work task
        - architectural design work task
        - interface design work task
        - component-level design work task
- Construction activity
- Deployment activity

1) **Communication**: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.
2) **Planning**: This activity establishes a plan for the software engineering work that follows. It describes 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.
3) **Modeling:** This activity encompasses the creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements. The modeling activity is composed of 2 software engineering actions- analysis and design.
    - ✓ Analysis encompasses a set of work tasks.
    - ✓ Design encompasses work tasks that create a design model.
4) **Construction:** This activity combines core generation and the testing that is required to uncover the errors in the code.
5) **Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evolution.

These 5 generic framework activities can be used during the development of small programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.

The following are the set of **Umbrella Activities.**

1) **Software project tracking and control –** allows the software team to assess progress against the project plan and take necessary action to maintain schedule.
2) **Risk Management -** assesses risks that may effect 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) **Formal Technical Reviews** - assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.

5) **Measurement -** define and collects process, project and product measures that assist the team in delivering software that needs customer's 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 and establishes mechanisms to achieve reusable components.

8) **Work Product preparation and production -** encompasses the activities required to create work products such as models, document, logs, forms and lists.

Intelligent application of any software process model must recognize that adaption is essential for success but process models do differ fundamentally in:

- ✓ The overall flow of activities and tasks and the interdependencies among activities and tasks.
- ✓ The degree through which work tasks are defined within each frame work activity.
- ✓ The degree through which work products are identified and required.
- ✓ The manner which quality assurance activities are applied.
- ✓ The manner in which project tracking and control activities are applied.
- ✓ The overall degree of the detailed and rigor with which the process is described.
- ✓ The degree through which the customer and other stakeholders are involved with the project.
- ✓ The level of autonomy given to the software project team.
- ✓ The degree to which team organization and roles are prescribed.

THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):

The CMMI represents a process meta-model in two different ways:

- As a continuous model
- As a staged model.

Each process area is formally assessed against specific goals and practices and is rated according to the following capability levels.

Level 0: Incomplete. The process area is either not performed or does not achieve all goals and objectives defined by CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed;

Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributes and work products, measures and other process-improvement information to the organizational process assets".

Level 4: Quantitatively managed. All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment."Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

Level 5: Optimized. All level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration"

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. Specific practices refine a goal into a set of process-related activities.

The specific goals (SG) and the associated specific practices(SP) defined for project planning are

SG 1 Establish estimates

SP 1.1 Estimate the scope of the project

SP 1.2 Establish estimates of work product and task attributes

SP 1.3 Define project life cycle

SP 1.4 Determine estimates of effort and cost

SG 2 Develop a Project Plan

SP 2.1 Establish the budget and schedule

SP 2.2 Identify project risks

SP 2.3 Plan for data management

SP 2.4 Plan for needed knowledge and skills

SP 2.5 Plan stakeholder involvement

SP 2.6 Establish the project plan

SG 3 Obtain commitment to the plan

SP 3.1 Review plans that affect the project

SP 3.2 Reconcile work and resource levels

SP 3.3 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, **the generic goals (GG) and practices (GP)** for the project planning process area are

GG 1 Achieve specific goals

GP 1.1 Perform base practices

GG 2 Institutionalize a managed process

GP 2.1 Establish and organizational policy

GP 2.2 Plan the process

GP 2.3 Provide resources

GP 2.4 Assign responsibility

GP 2.5 Train people

GP 2.6 Manage configurations

GP 2.7 Identify and involve relevant stakeholders

GP 2.8 Monitor and control the process

GP 2.9 Objectively evaluate adherence

GP 2.10 Review status with higher level management

GG 3 Institutionalize a defined process

GP 3.1 Establish a defined process

GP 3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

GP 4.1 Establish quantitative objectives for the process

GP 4.2 Stabilize sub process performance

GG 5 Institutionalize and optimizing process

GP 5.1 Ensure continuous process improvement

GP 5.2 Correct root causes of problems

PROCESS PATTERNS

The software process can be defined as a collection patterns that define a set of activities, actions, work tasks, work products and/or related behaviors required to develop computer software.

A process pattern provides us with a template- a consistent method for describing an important characteristic of the software process. A pattern might be used to describe a complete process and a task within a framework activity.

Pattern Name: The pattern is given a meaningful name that describes its function within the software process.

Intent: The objective of the pattern is described briefly.

Type: The pattern type is specified. There are three types

1. **Task patterns** define a software engineering action or work task that is part of the process and relevant to successful software engineering practice. *Example:* Requirement Gathering
2. **Stage Patterns** define a framework activity for the process. This pattern incorporates multiple task patterns that are relevant to the stage.

   Example: Communication
3. **Phase patterns** define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.

   Example: Spiral model or prototyping.

Initial Context: The conditions under which the pattern applies are described prior to the initiation of the pattern, we ask

(1) What organizational or team related activities have already occurred.
(2) What is the entry state for the process
(3) What software engineering information or project information already exists

Problem: The problem to be solved by the pattern is described.

Solution: The implementation of the pattern is described.

This section describes how the initial state of the process is modified as a consequence the initiation of the pattern.
It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern

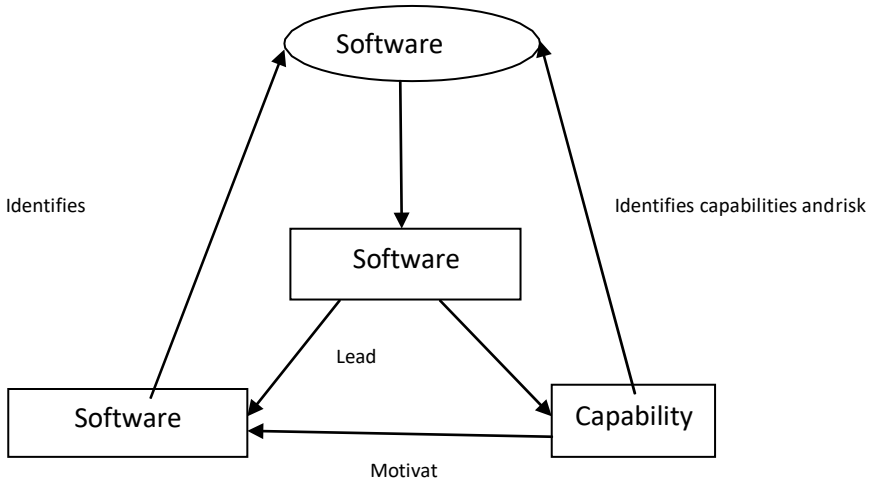
Resulting Context: The conditions that will result once the pattern has been successfully implemented are described. Upon completion of the pattern we ask

(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?

Known Uses: The specific instances in which the pattern is applicable are indicated
Process patterns provide and effective mechanism for describing any software process.
The patterns enable a software engineering organization to develop a hierarchical process description that begins at a high-level of abstraction.
Once process pattern have been developed, they can be reused for the definition of process variants-that is, a customized process model can be defined by a software team using the pattern as building blocks for the process models.

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. In addition, the process itself should be assessed to be essential 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 have been proposed over the past few decades.

Standards CMMI Assessment Method for Process Improvement (SCAMPI) provides a five step process assessment model that incorporates initiating, diagnosing, establishing, acting & 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, using the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504) standard defines a set of requirements for software process assessments. 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 is a generic standard that applies to any organization that wants to improve the overall quality of the products, system, or services that it provides. Therefore, the standard is directly applicable to software organizations &companies.

PERSONAL AND TEAM PROCESS MODELS:

The best software process is one that is close to the people who will be doing the work.Each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization. Alternatively, the team itself would create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.

Personal software process (PSP)

The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.

The PSP process model defines five framework activities: planning, high-level design, high level design review, development, and postmortem.

Planning: This activity isolates requirements and, base on these develops both size and resource estimates. In addition, a defect estimate is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High level design: External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High level design review: Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development: The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important task and work results.

Postmortem: Using the measures and metrics collected the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP stresses the need for each software engineer to identify errors early and, as important, to understand the types of errors that he is likely to make.

PSP represents a disciplined, metrics-based approach to software engineering.

Team software process (TSP): The goal of TSP is to build a "self-directed project team that organizes itself to produce high-quality software. The following are the objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams(IPT) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

A self-directed team defines

- roles and responsibilities for each team member
- tracks quantitative project data
- identifies a team process that is appropriate for the project
- a strategy for implementing the process
- defines local standards that are applicable to the teams software engineering work;
- continually assesses risk and reacts to it
- Tracks, manages, and reports project status.
-

TSP defines the following framework activities: launch, high-level design, implementation, integration and test, and postmortem.
TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
Scripts define specific process activities and other more detailed work functions that are part of the team process.
Each project is "launched" using a sequence of tasks.

The following launch script is recommended

- Review project objectives with management and agree on and document team goals
- Establish team roles
- Define the teams development process
- Make a quality plan and set quality targets
- Plan for the needed support facilities

PROCESS MODELS

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

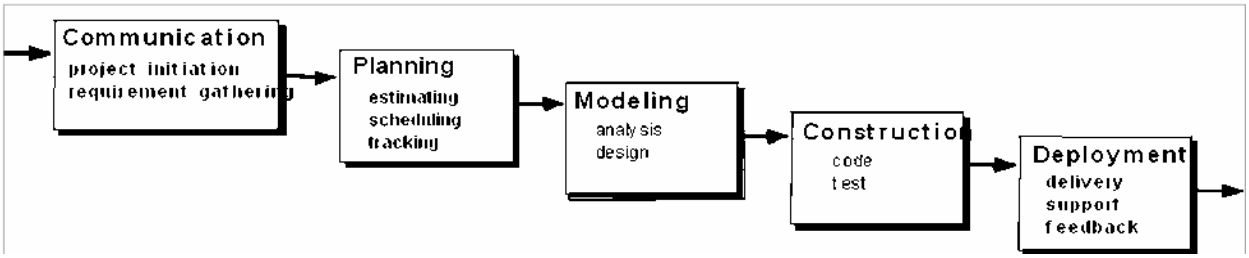
THE WATERFALL MODEL:

The waterfall model, sometimes called the *classic life cycle*, suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

Context: Used when requirements are reasonably well understood.

Advantage:

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



The **problems** that are sometimes encountered when the waterfall model is applied are:

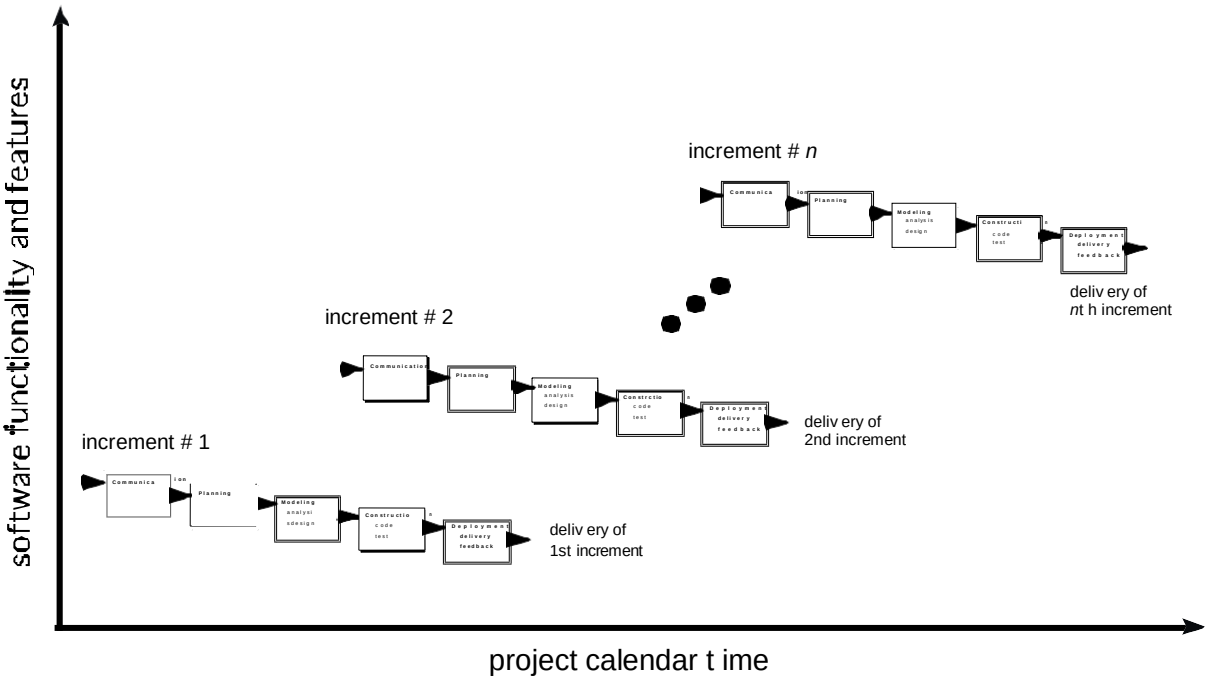
1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.
3. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

INCREMENTAL PROCESS MODELS:

1) The incremental model
2) The RAD model

THE INCREMENTAL MODEL:

Context: 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, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



- The incremental model combines elements of the waterfall model applied in an iterative fashion.
- The incremental model delivers a series of releases called increments that provide progressively more functionality for the customer as each increment is delivered.
- When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed. The core product is used by the customer. As a result, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.

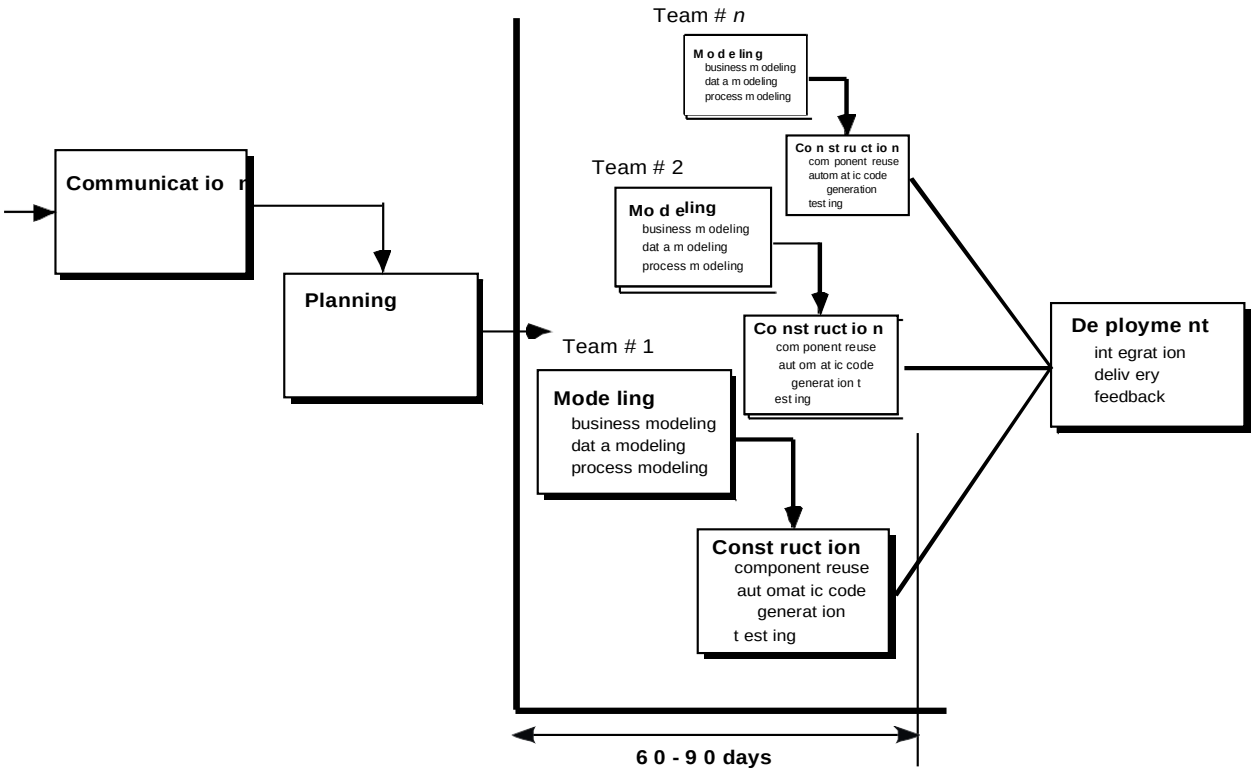
For *example*, word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking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

Difference: The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on delivery of an operational product with each increment

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within a very short time period.



The RAD approach maps into the generic framework activities.
Communication works to understand the business problem and the information characteristics that the software must accommodate.

Planning is essential because multiple software teams works in parallel on different system functions.

Modeling encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.

Deployment establishes a basis for subsequent iterations.

The RAD approach has **drawbacks**:

For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail

If a system cannot be properly modularized, building the components necessary for RAD will be problematic

If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and

RAD may not be appropriate when technical risks are high.

EVOLUTIONARY PROCESS MODELS:

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

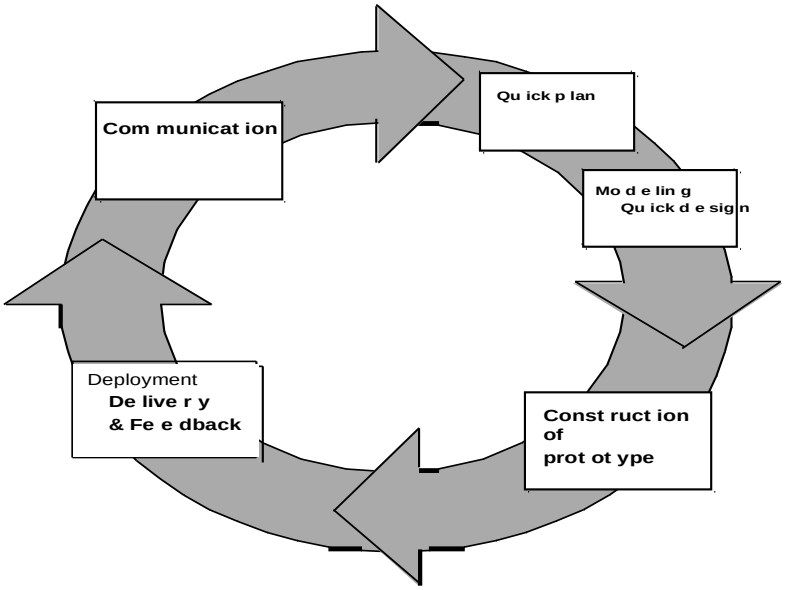
PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.



Context:

If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.

If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

Prototyping can be **problematic** for the following reasons:

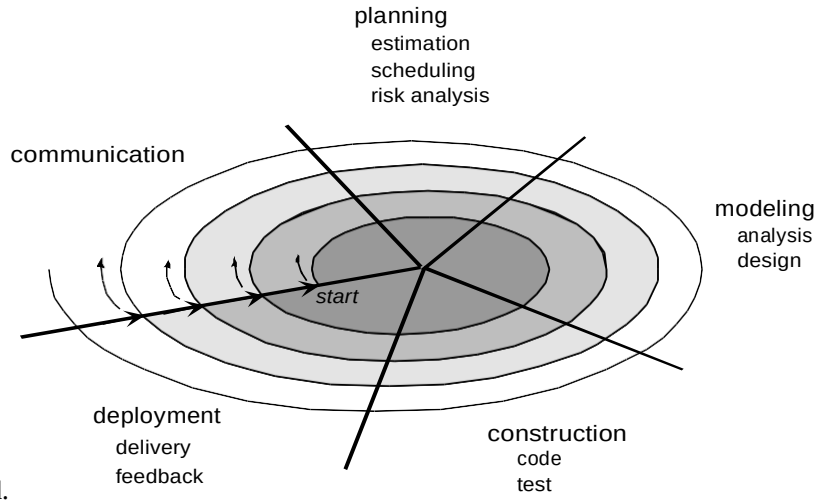
1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.
2. The developer often makes 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, the developer 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

- The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

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



produced.

- **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.
- The first circuit around the spiral might result in the development of 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.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
  - The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.
  - If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.
  - Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

Context: The spiral model can be adopted to apply throughout the entire life cycle of an application, from concept development to maintenance.

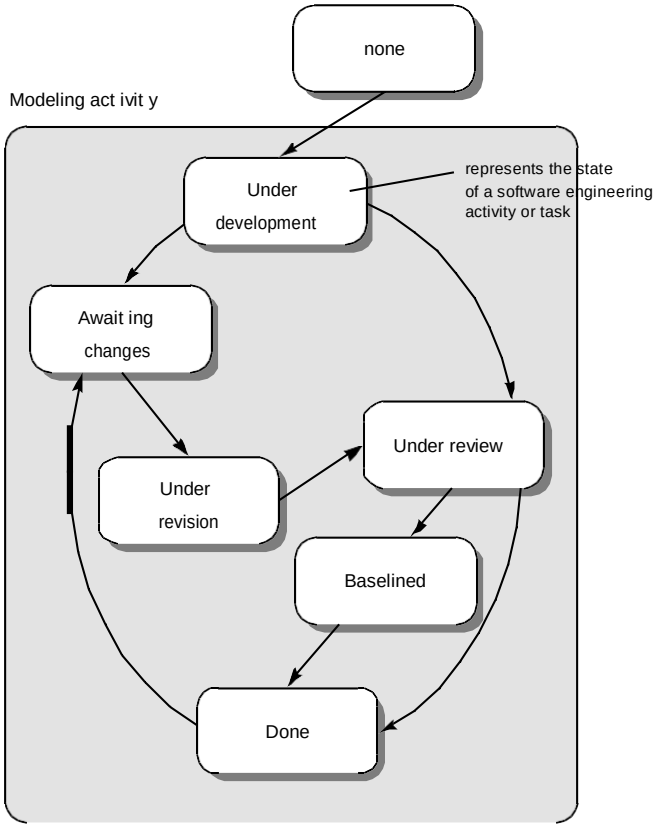
Advantages:
It provides the potential for rapid development of increasingly more complete versions of the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product.

Draw Backs:
The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

THE CONCURRENT DEVELOPMENT MODEL:

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



The activity *modeling* may be in anyone of the states noted at any given time. Similarly, other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states.

Any of the activities of a project may be in a particular state at any one time:

- under development
- awaiting changes
- under revision
- under review

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 **none** state while initial communication was

completed, now makes a transition into the **under development** state. If, 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.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.

The event analysis model correction which will trigger the analysis action from the **done** state into the **awaiting changes** state.

Context: The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved.

Advantages:

- The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.
- It defines a network of activities rather than each activity, action, or task on the network exists simultaneously with other activities, action and tasks.

A FINAL COMMENT ON EVOLUTIONARY PROCESSES:

- The concerns of evolutionary software processes are:
- The first concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, evolutionary software process do not establish the maximum speed of the evolution. If the evolution occurs too fast, without a period of relaxation, it is certain that the process will fall into chaos.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality.

THE UNIFIED PROCESS:

The unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development.

The Unified process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. 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". If suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

A BRIEF HISTORY:

During the 1980s and into early 1990s, object-oriented (OO) methods and programming languages gained a widespread audience throughout the software engineering community. A wide variety of object-oriented analysis (OOA) and design (OOD) methods were proposed during the same time period.

During the early 1990s James Rumbaugh, Grady Booch, and Ival Jacobsom began working on a "Unified method" that would combine the best features of each of OOD & OOA. The result was UML- a unified modeling language that contains a robust notation fot the modeling and development of OO systems.

By 1997, UML became an industry standard for object-oriented software development. At the same time, the Rational Corporation and other vendors developed automated tools to support UML methods.

Over the next few years, Jacobson, Rumbugh, and Booch developed the Unified process, a framework for object-oriented software engineering using UML. Today, the Unified process and UML are widely used on OO projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

PHASES OF THE UNIFIED PROCESS:

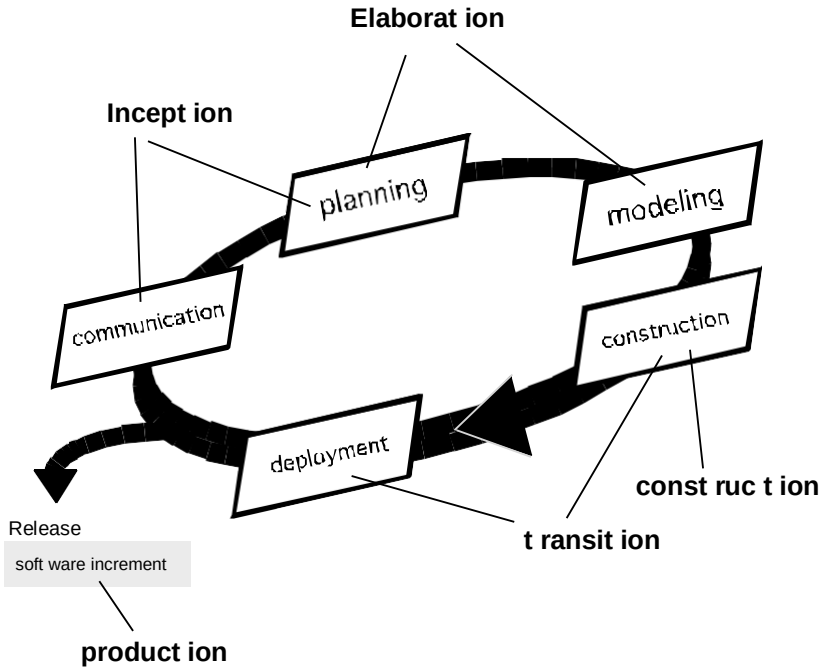
The ***inception*** phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, business requirements for the software are identified, a rough architecture for the system is proposed and a plan for the iterative, incremental nature of the ensuing project is developed.

The ***elaboration*** phase encompasses the customer communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use-cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software- the use-case model, the analysis model, the design model, the implementation model, and the deployment model.

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, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.

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 given to end-users for beta testing, and user feedback reports both defects and necessary changes.

The ***production*** phase of the UP coincides with the deployment activity of the generic process. During this phase, the on-going use of the software is monitored, support for the operating environment is provided, and defect reports and requests for changes are submitted and evaluated.



A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set. That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.

UNIFIED PROCESS WORK PRODUCTS:

During the ***inception phase***, the intent is to establish an overall "vision" for the project,

- identify a set of business requirements,
- make a business case for the software, and
- define project and business risks that may represent a threat to success.

The most important work product produced during the inception is the use-case modell-a collection of use-cases that describe how outside actors interact with the system and gain value from it. The use-case model is a collection of software features and functions by describing a set of preconditions, a flow of events and a set of post-conditions for the interaction that is depicted.

The use-case model is refined and elaborated as each UP phase is conducted and serves as an important input for the creation of subsequent work products. During the inception phase only 10 to 20 percent of the use-case model is completed. After elaboration, between 80 to 90 percent of the model has been created.

The ***elaboration phase*** produces a set of work products that elaborate requirements and produce and architectural description and a preliminary design. The UP analysis model is the work product that is developed as a consequence of this activity. The classes and analysis packages defined as part of the analysis model are refined further into a design model which identifies design classes, subsystems, and the interfaces between subsystems. Both the analysis and design models expand and refine an evolving representation of software architecture. In addition the elaboration phase revisits risks and the project plan to ensure that each remains valid.

The ***construction phase*** produces an implementation model that translates design classes into software components into the physical computing environment. Finally, a test model describes tests that are used to ensure that use cases are properly reflected in the software that has been constructed.

The ***transition phase*** delivers the software increment and assesses work products that are produced as end-users work with the software. Feedback from beta testing and qualitative requests for change is produced at this time.