SOFTWARE ENGINEERING

UNIT-1

1. The Nature of Software

Defining Software
Software Application Domains
Legacy Software

2. The Unique Nature of WebApps
3. Software Engineering
4. The Software Process
5. Software Engineering Practice
The Essence of Practice
General Principles
6. Software Myths

7. A Generic Process Model

Defining a Framework Activity
Identifying a Task Set
Process Patterns

8. Process Assessment and Improvement
9. Prescriptive Process Models
9.1The Waterfall Model
9.2Incremental Process Models
9.3Evolutionary Process Models
9.4Concurrent Models
9.5A Final Word on Evolutionary Processes
10. Specialized Process Models
Component-Based Development
The Formal Methods Model
Aspect-Oriented Software Development
11. The Unified Process
A Brief History
Phases of the Unified Process
12. Personal and Team Process Models
Personal Software Process (PSP)
Team Software Process (TSP)
13. Process Technology
14. Product and Process

www.jntufastupdates.com

1. The Nature of Software

Today, software takes on a dual role. *It is a* ***product****, and at the same time, the* ***vehicle*** *for delivering a product.*

As a product, it delivers the computing potential embodied by computer hardware or more broadly, *by a network of computers* that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, ***software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources***.

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.* It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context.

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 industrialized world.

Engineering Discipline:

- Engineering is a disciplined approach with some organized steps in a managed way to construction, operation, and maintenance of software.
- Engineering of a product goes through a series of stages, i.e., planning, analysis and specification, design, construction, testing, documentation, and deployment.
- The disciplined approach may lead to better results.
- The general stages for engineering the software include feasibility study and preliminary investigation, requirement analysis and specification, design, coding, testing, deployment, operation, and maintenance.

www.jntufastupdates.com

Software Crisis:

- Software crisis, the symptoms of the problem of engineering the software, began to enforce the practitioners to look into more disciplined software engineering approaches for software development.

- The software industry has progressed from the desktop PC to network-based computing to service-oriented computing nowadays.
- The development of programs and software has become complex with increasing requirements of users, technological advancements, and computer awareness among people.

- *Software crisis symptoms*

  - complexity,

  - hardware versus software cost,

  - Lateness and costliness,

  - poor quality,

  - unmanageable nature,

  - immaturity,

  - lack of planning and management practices,

  - Change, maintenance and migration,

  - etc.

What is Software Engineering?

- *The solution to these software crises is to introduce systematic software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.*

- The systematic means the methodological and pragmatic way of development, operation and maintenance of software.

- Systematic development of software helps to understand problems and satisfy the client needs.

www.jntufastupdates.com

- Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

- Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

- Operational software must be correct, efficient, understandable, and usable for work at the client site.

- IEEE defines

- *The systematic approach to the development, operation, maintenance, and retirement of software.*
-
- Defining Software

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

Software characteristics

- Software has logical properties rather than physical.
- Software is mobile to change.
- Software is produced in an engineering manner rather than in classical sense.
- Software becomes obsolete but does not wear out or die.
- Software has a certain operating environment, end user, and customer.
- Software development is a labor-intensive task

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

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

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.

In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

www.jntufastupdates.com

2. Software doesn't "wear out."

The above picture depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); ***defects are corrected*** and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.

As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

Stated simply, the hardware begins to wear out.
Software doesn't "wear out."
But Software does deteriorate!

The above picture explains the software failure in development process. During development process, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—***the software is deteriorating due to change.***

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

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.

The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.

In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software Application Domains

(Types of Software or Categories of Software)

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

i. **System software**—a collection of programs written to service other programs.

System software processes complex, but determinate information structures.
e.g., compilers, editors, and file management utilities

Systems applications process largely indeterminate data.
(e.g., operating system components, drivers, networking software, telecommunications processors)

ii. **Application software**—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.
e.g., point-of-sale transaction processing, real-time manufacturing process control.

iii. **Engineering/scientific software**—is a special software to implement Engineering and Scientific applications. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

www.jntufastupdates.com

iv. **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.

e.g., key pad control for a microwave oven.

Provide significant function and control capability

e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

v. **Product-line software**—designed to provide a specific capability for use by many different customers.

e.g., inventory control products, word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications.

vi. **Web applications**—called “WebApps,” this network-centric software category spans a wide array of applications. WebApps are linked with hypertext files. WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

vii. **Artificial intelligence software**— makes use of nonnumerical algorithms to solve complex problems of straightforward analysis.

Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

viii. **Open-world computing**—Software related to wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

ix. **Netsourcing**—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

x. **Open source**—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development.

The challenge for software engineers is to build source code 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.

www.jntufastupdates.com

Legacy Software

Older programs —often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The maintenance of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.
Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

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

When these modes of evolution occur, a legacy system must be reengineered, so that it remains useful in the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other".

2. The Unique Nature of WebApps

Web Apps means Web Applications. WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

Web engineers to provide computing capability along with informational content. *Web-based systems and applications* were born.

WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different.

The following attributes are encountered in the vast majority of WebApps.

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

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

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

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

v. **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often ***demand access on a 24/7/365 basis***. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

vi. **Data driven.** The primary function of many WebApps is to use hypermedia to present ***text, graphics, audio, and video*** content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

vii. **Content sensitive.** The quality and aesthetic (beauty) ***nature of content*** remains an important determinant of the quality of a WebApp.

viii. **Continuous evolution. (Updated version) -**Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

ix. **Immediacy.** Although *immediacy*—the compelling need to get ***software to market quickly***—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.7

x. **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to ***protect sensitive content and provide secure modes of data transmission***, ***strong security measures must be implemented*** throughout the infrastructure that supports a WebApp and within the application itself.

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

www.jntufastupdates.com

3. Software Engineering

What is Software Engineering?
software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.

• The systematic means the methodological and pragmatic way of development, operation and maintenance of software.

• Systematic development of software helps to understand problems and satisfy the client needs.

• Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

• Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

• Operational software must be correct, efficient, understandable, and usable for work at the client site.

• IEEE defines
The systematic approach to the development, operation, maintenance, and retirement of software.

Software Engineering -

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

. *It follows that design becomes a unique activity.*

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

• *It follows that software should be maintainable, software in all of its forms and across all of its application domains should be engineered.*

4. <u>The Software Process</u>

- 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.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to ***deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation*** and those who will use it.

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

A generic process framework for software engineering encompasses five activities:

i. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders11 The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
ii. **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
iii. **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
v. **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and

www.jntufastupdates.com

provides feedback based on the evaluation. These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

For many software projects, framework activities are applied iteratively as a project progresses. That is,

communication,
planning,
modeling,
construction, and
deployment are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

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

Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists. Each of these umbrella activities is discussed in detail later in this book.

The software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, 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

www.jntufastupdates.com

- 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**

5. Software Engineering Practice

Generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work.

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).
In the context of software engineering, these commonsense steps lead to a series of essential questions

i. Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think. Understanding isn't always that easy. 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?

ii. 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 subproblems be defined?*** If so, are solutions readily apparent for the subproblems?
- ***Can you represent a solution in a manner that leads to effective implementation?*** Can a design model be created?

www.jntufastupdates.com

iii. **<u>Carry out the plan.</u>** The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but 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?

iv. **<u>Examine the result.</u>** 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? It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

General Principles

David Hooker has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following.

The First Principle: ***The Reason It All Exists***

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: ***KISS (Keep It Simple, Stupid!)***

Software design is not a random process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system.

Features should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: ***Maintain the Vision***

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: ***What You Produce, Others Will Consume***

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone*

else will have to understand what you are doing. The audience for any product of software development is potentially large.

Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: ***Be Open to the Future***

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*.

Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.14 This could very possibly lead to the reuse of an entire system. This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

The Sixth Principle: ***Plan Ahead for Reuse***

Reuse saves time and effort.15Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh principle: ***Think!***

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer.

When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer based systems would be eliminated.

www.jntufastupdates.com

6. Software Myths

- Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.
- Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often announced by experienced practitioners who "know the score."
- Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.
- However, old attitudes and habits are difficult to modify, and remnants of software myths remain.
- Software Myths are three types
  1. Managers Myths
  2. Customers Myths (and other non-technical stakeholders)
  3. Practitioners Myths

i. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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 Mr. Brooks "adding people to a late software project makes it later."

At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.

People can be added but only in a planned and well coordinated manner.

Myth: *If 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.

ii. **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 (usually derived iteratively) 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 requires additional resources and major design modification.

iii. **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. Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost.

www.jntufastupdates.com

SOFTWARE ENGINEERING
UNIT-1B

7. A Generic Process Model

Defining a Framework Activity

Identifying a Task Set

Process Patterns

8. Process Assessment and Improvement

9. Prescriptive Process Models

The Waterfall Model

Incremental Process Models

Evolutionary Process Models

Concurrent Models

10. Specialized Process Models

Component-Based Development

The Formal Methods Model

Aspect-Oriented Software Development

11. The Unified Process

A Brief History

Phases of the Unified Process

12. Personal and Team Process Models

Personal Software Process (PSP)

Team Software Process (TSP)

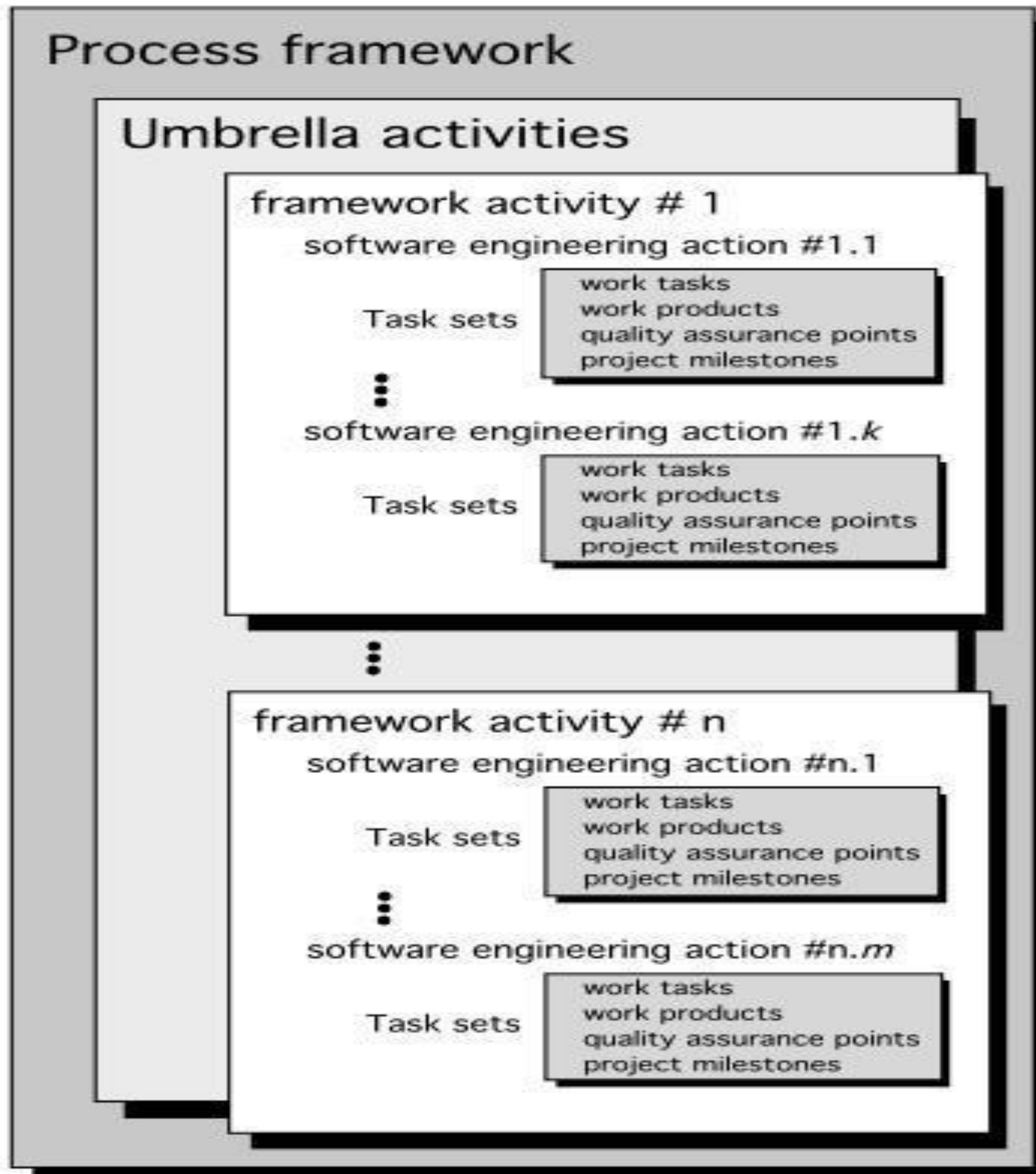
13. Process Technology

14. Product and Process

www.jntufastupdates.com

7. A GENERIC PROCESS MODEL

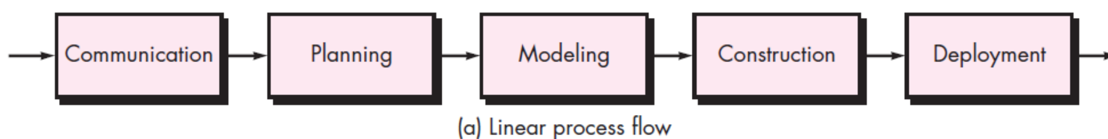
A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.



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

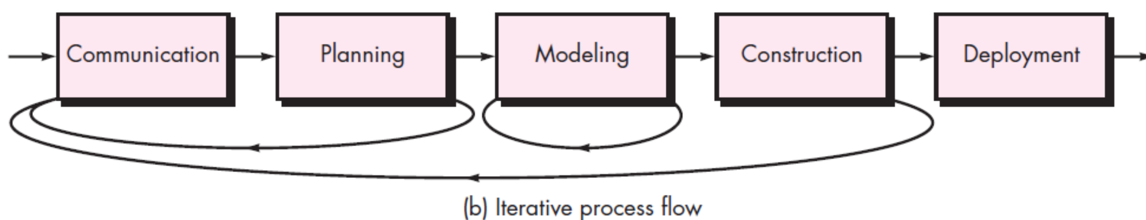
- A generic process framework for software engineering defines five framework activities
    - **communication,**
    - **planning,**
    - **modeling,**
    - **construction,** and
    - **deployment.**
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

- The *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in the above Figure.
- A linear *process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment which is shown in the following Figure.



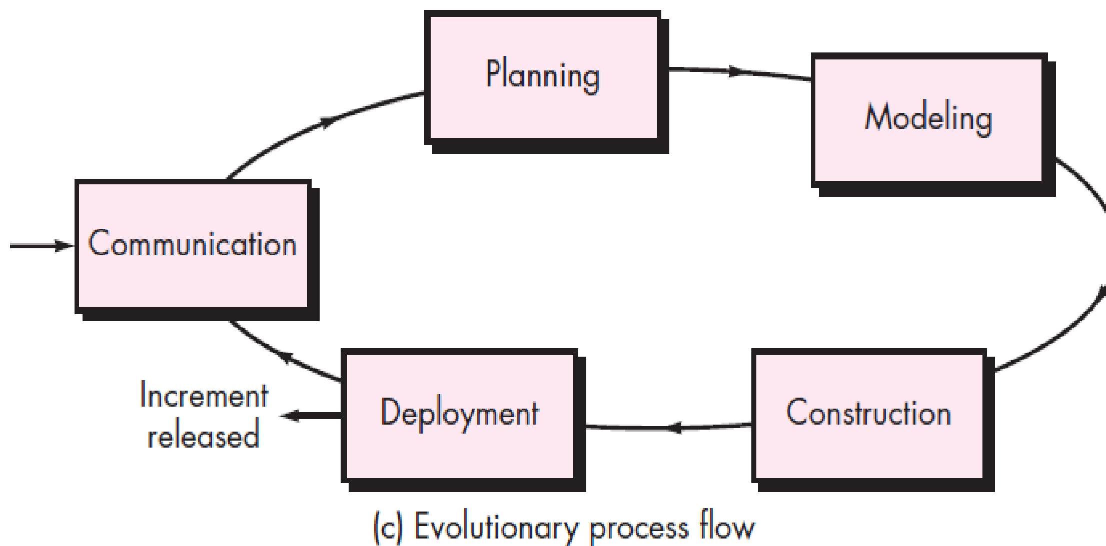
(a) Linear process flow

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (shown in the following Figure).



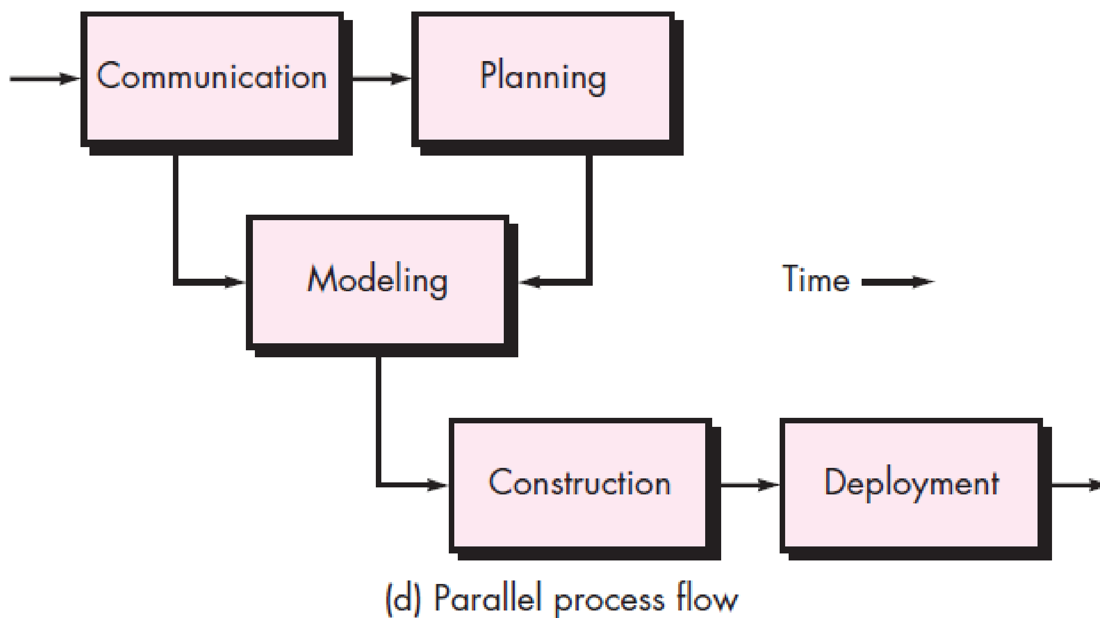
(b) Iterative process flow

- 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 (shown in the following Figure).



(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. (shown in the following figure)



(d) Parallel process flow

Defining a Framework Activity

There are five framework activities, they are

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

These five framework activities provide a basic definition of Software Process. These Framework activities provides basic information like *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

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

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

Identifying a Task Set

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

Process Patterns

- Every software team encounters problems as it moves through the software process.
- It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.
- A *process pattern*1 describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a template a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction.
- In some cases, a pattern might be used to describe a problem and solution associated with a complete process model (e.g., prototyping).

www.jntufastupdates.com

- In other situations, patterns can be used to describe a problem and solution associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).
- Ambler has proposed a template for describing a process pattern:

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

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

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

1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity).

An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

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

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

For example, the **Planning** pattern (a stage pattern) requires that

(1) customers and software engineers have established a collaborative communication;
(2) successful completion of a number of task patterns [specified] for the **Communication** pattern has occurred; and
(3) the project scope, basic business requirements, and project constraints are known.

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

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern.

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

(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process?

www.jntufastupdates.com

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

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns:
ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and **ScenarioCreation.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Conclusion on Process Patterns

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

8. PROCESS ASSESSMENT AND IMPROVEMENT

- The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs.
- Process patterns must be coupled with solid software engineering practice
- In addition, 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:

1. **Standard CMMI Assessment Method for Process Improvement (SCAMPI)**— provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.
2. **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.
3. **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.
4. **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.

www.jntufastupdates.com

9. PRESCRIPTIVE PROCESS MODELS

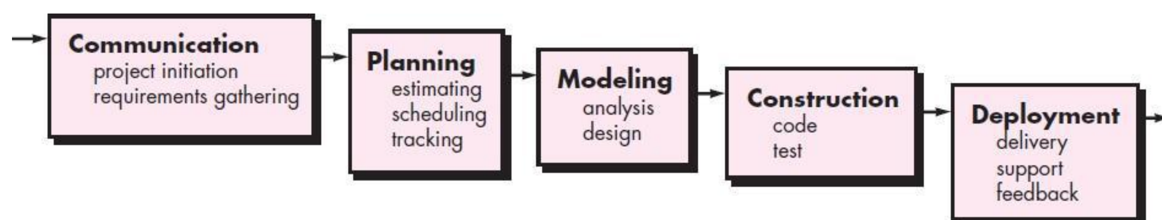
- Prescriptive process models were originally proposed to bring order to the chaos (disorder) of software development. History
- these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.
- The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise".
- The edge of chaos can be visualized as an unstable, partially structured state.
- It is unstable because it is constantly attracted to chaos or to absolute order.
- The prescriptive process approach in which order and project consistency are dominant issues.
- "prescriptive" means prescribe a set of process elements
    - framework activities,
    - software engineering actions,
    - tasks,
    - work products,
    - quality assurance, and
    - change control mechanisms for each project.
- Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.
- All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity in a different manner.

The Waterfall Model

The waterfall model, 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.



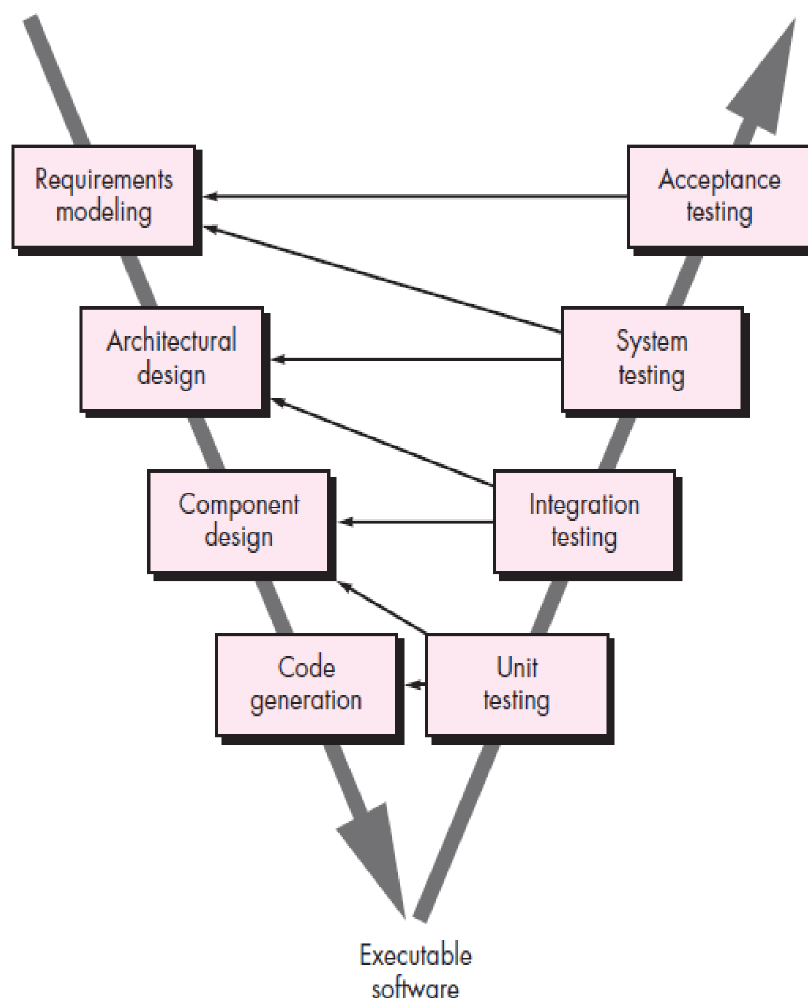
The **problems** that are sometimes encountered when the ***waterfall model*** is applied are:

i. 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.

ii. 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.

iii. 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.

9.1.2V-model



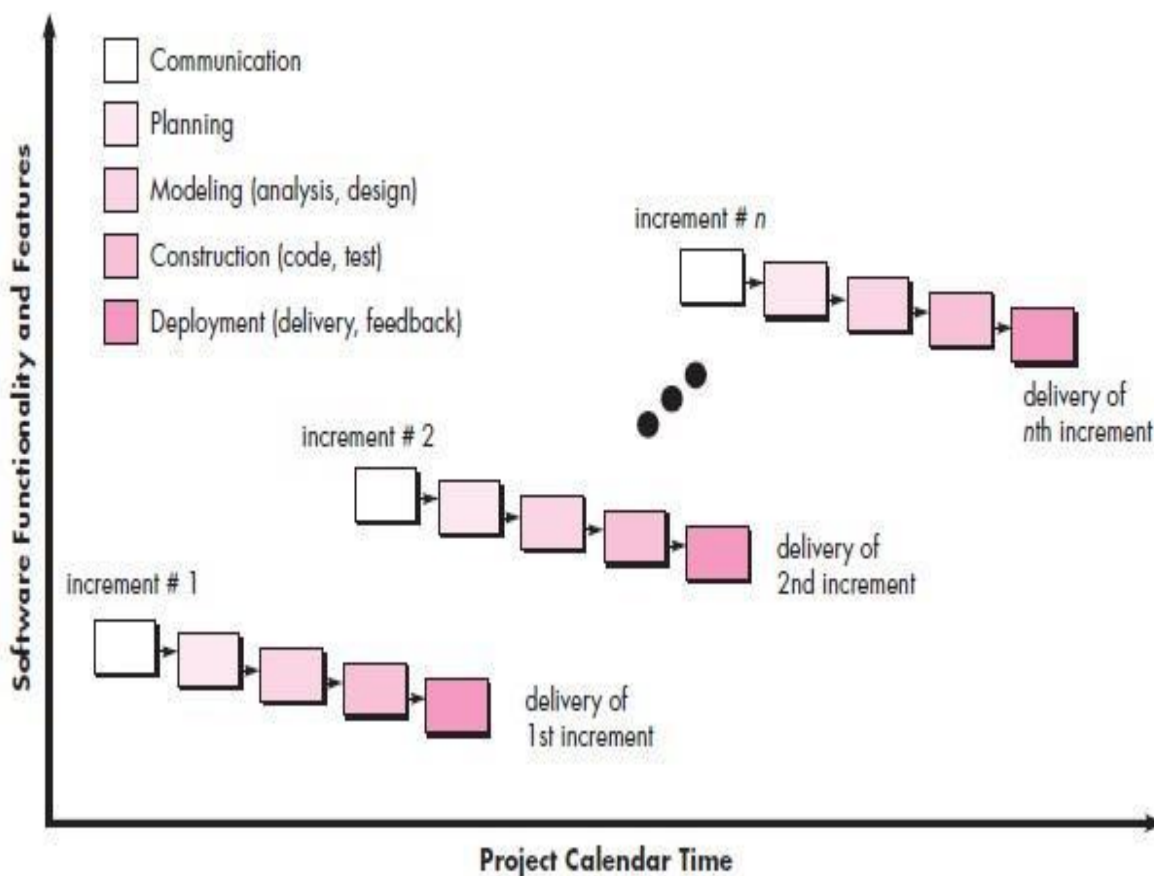
- A variation in the representation of the waterfall model is called the *V-model*.

Represented in the above Figure.

- The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.
- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- In reality, there is no fundamental difference between the classic life cycle and the V-model.
- The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

Incremental Process Models



- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort difficulty to implement linear process.
- 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.
- The *incremental* model combines elements of linear and parallel process flows. The above Figure shows the incremental model which applies linear sequences
- Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.
- For example, MS-Word software developed using the incremental paradigm might deliver
  - basic file management, editing, and document production functions in the first increment;

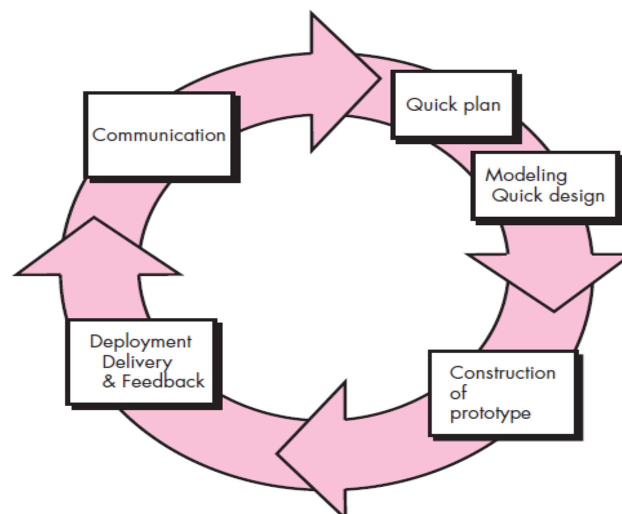
  - more sophisticated editing and document production capabilities in the second increment;
  - spelling and grammar checking in the third increment; and
  - advanced page layout capability in the fourth increment.
  - It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment.
- Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

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.

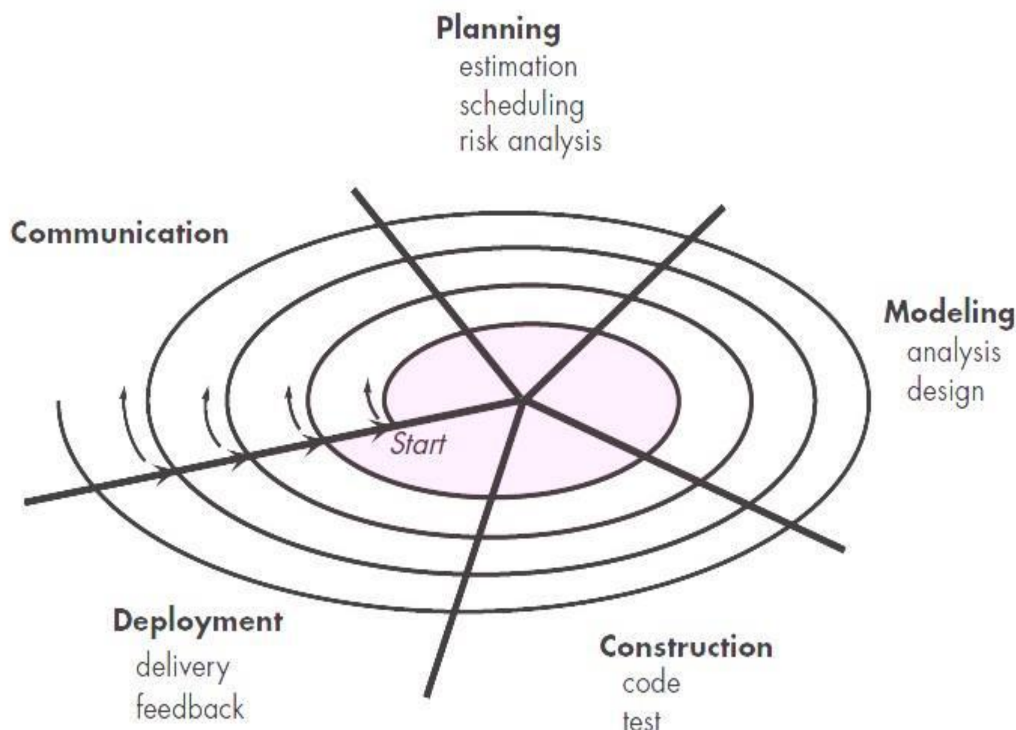
Example:

- 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:
- 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.
- 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.

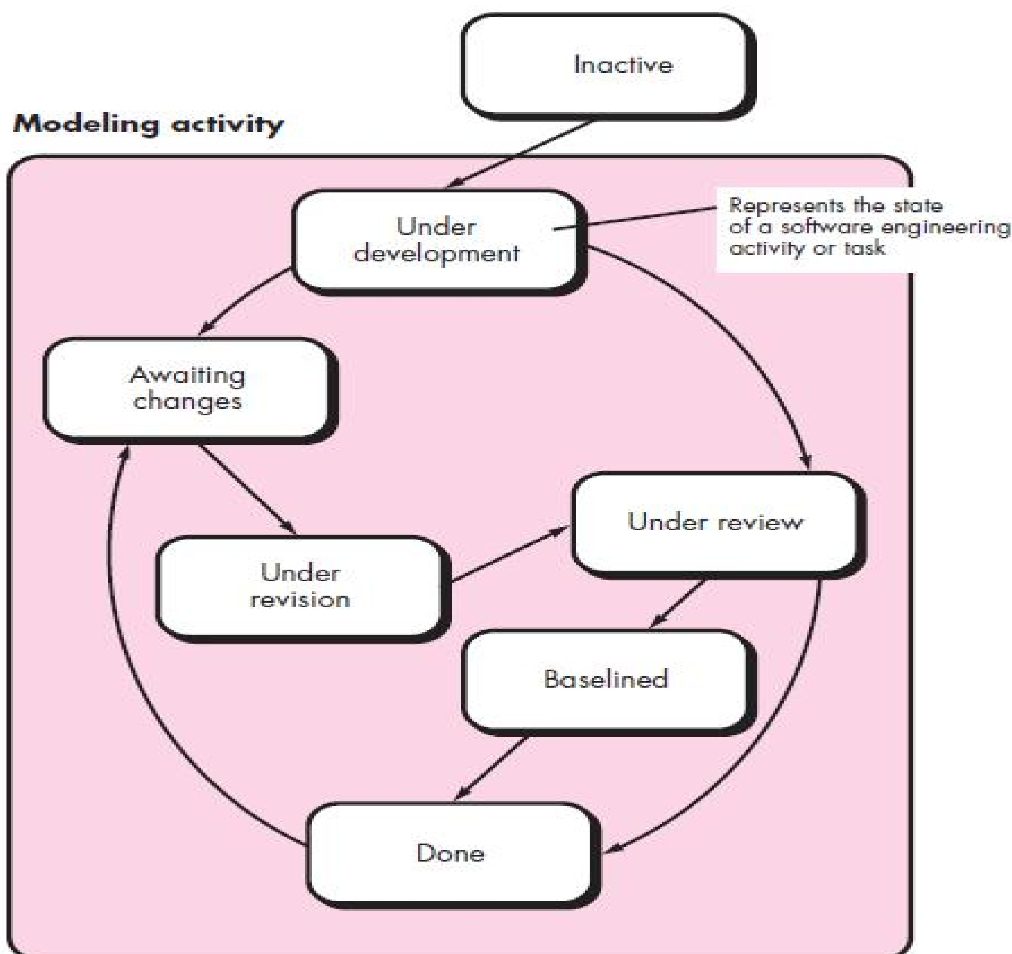
The Spiral Model.



- The Spiral model is proposed by Barry Boehm.
- The *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are
- **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 pecification; 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.

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.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.
- The above Figure provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach.
- The activity—**modeling**—may be in any one of the states - noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

- For example, early in a project the communication activity (not shown in the figure) 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.
- 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.
- Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.
- For example, during early stages of design an inconsistency in the requirements model is uncovered. This generates the event *analysis model correction,* which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.
- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project.
- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

10. SPECIALIZED PROCESS MODELS

- Specialized process models take on many of the characteristics of one or more of the traditional models.

Component-Based Development

- Commercial off-the-shelf (COTS) **software components**, developed by vendors who offer them as **products**, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The *component-based development model* incorporates many of the characteristics of the **spiral model**.
- It is **evolutionary in nature**, demanding an iterative approach to the creation of software.
- However, the component-based development model **constructs applications from prepackaged software components.**
- Modeling and construction activities begin with the identification of **candidate components**.
- These **candidate components** can be designed as either conventional software modules or object-oriented classes or packages of classes.
- Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps:
  1. Available component-based products are researched and evaluated for the application domain in question.
  2. Component integration issues are considered.
  3. A software architecture is designed to accommodate the components.
  4. Components are integrated into the architecture.
  5. Comprehensive testing is conducted to ensure proper functionality.

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

The Formal Methods Model

- The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.
- A variation on this approach, called *clean room software engineering*, is currently applied by some software development organizations.
- When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through the review, but through the application of mathematical analysis.
- When formal methods are used during design, they serve as a basis for program verification and "discover and correct errors" that might otherwise go undetected.
- 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.

Aspect-Oriented Software Development

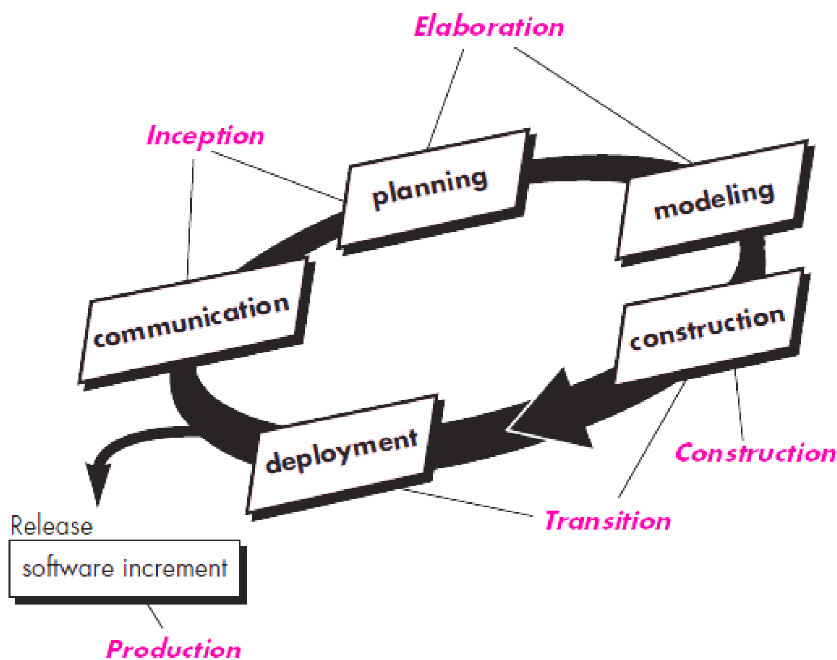
- Localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.
- As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture.
- Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).
- When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns.*
- ***Aspectual requirements*** define those crosscutting concerns that have an impact across the software architecture.
- ***Aspect-oriented software development*** **(AOSD)**, often referred to as ***aspect-oriented programming*** **(AOP),** is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".
- ***aspect-oriented component engineering*** **(AOCE):** AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects".
- Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects);
  - event generation,
  - transport and receiving (distribution aspects);
  - data store/retrieve and indexing (persistency aspects);
  - authentication,
  - encoding and access rights (security aspects);

  - transaction atomicity,
  - concurrency control and logging strategy (transaction aspects); and so on.
- Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.
- The evolutionary model is appropriate as aspects are identified and then constructed.
- The parallel nature of concurrent development is essential because aspects are engineered independently.

11. THE UNIFIED PROCESS

- The unified process related to "use case driven, architecture-centric, iterative and incremental" software process.
- The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models.
- 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.
- It suggests a process flow that is iterative and incremental.
- During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method".
- The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.
- By 1997, UML became a de facto industry standard for object-oriented software development.
- UML is used to represent both requirements and design models.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework.
- 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.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

Phases of the Unified Process



- The above figure depicts the different phases in Unified Process.
- 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.
    - The architecture will be refined.
    - Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases.
- 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—the use case model, the requirements model, the design model, the implementation model, and the deployment model.
    - In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.
    - The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system.
    - In addition, the plan is carefully reviewed.
    - Modifications to the plan are often made at this time.
- 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.
    - The elaboration phase 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.

- The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.
  - Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
  - In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.
  - At the conclusion of the transition phase, the software increment becomes a usable software release.
- The ***production phase*** of the UP coincides with the deployment activity of the generic process.
  - During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
  - 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.

- A software engineering workflow is distributed across all UP phases.
- In the context of UP, a *workflow* is 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.
- It should be noted that not every task identified for a UP workflow is conducted for every software project.
- The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

12. PERSONAL AND TEAM PROCESS MODELS

- Software process model has been developed at a corporate or organizational level.
- It can be effective only if it is helpful to significant adaptation to meet the needs of the project team that is actually doing software engineering work.
- In an ideal setting, it create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization.
- Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.
- It is possible to create a "personal software process" and/or a "team software process."
- Both require hard work, training, and coordination, but both are achievable.

www.jntufastupdates.com

Personal Software Process (PSP)

- Every developer uses some process to build computer software.
- The process may be temporary; may change on a daily basis; may not be efficient, effective, or even successful; but a “process” does exist.
- Personal process, an individual must move through four phases, each requiring training and careful instrumentation.
- The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.
- In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.
- The PSP model defines five framework activities:

  1. Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

  2. 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.

  3. High-level design review. Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

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

  5. Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), 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 to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.
- PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97].
- However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach.
- PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high.
- The required level of measurement is culturally difficult for many software people. Can PSP be used as an effective software process at a personal level? The answer is an unequivocal “yes.” But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

www.jntufastupdates.com

Team Software Process (TSP)

- *Team Software Process* (TSP) build a "self-directed" project team that organizes itself to produce high-quality software.
- 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 (IPTs) 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 has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality);
- TSP identifies a team process that is appropriate for the project and a strategy for implementing the process;
- TSP defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.
- TSP defines the following framework activities:
  - **project launch,**
  - **high-level design,**
  - **implementation,**
  - **integration and test,** and
  - **postmortem.**
- These activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product.
- The postmortem sets the stage for process improvements.
- TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
- TSP recognizes that the best software teams are self-directed.
- Team members set
  - project objectives,
  - adapt the process to meet their needs,
  - control the project schedule, and
  - analysis of the metrics collected,
  - work continually to improve the team's approach to software engineering.
- Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality.
- The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

www.jntufastupdates.com

13. PROCESS TECHNOLOGY

- ***Process technology tools*** have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.
- ***Process technology tools*** allow a software organization to build an automated model of the process framework, task sets, and umbrella activities.
- The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.
- Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model.
- Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted.
- The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

PRODUCT AND PROCESS

- If the process is weak, the end product will undoubtedly suffer.
- But an obsessive overreliance on process is also dangerous.
- Product is final developed software
- Process is set of activities, actions and tasks to develop product.
- structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process).
- These swings are harmful in and of themselves because they confuse the average software practitioner.
- So software analyst must concentrate on Product by streamline the Process.
- All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result (Product) in a representation.
- Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity.
- The duality of product and process is one important element in keeping creative Speople engaged as software engineering continues to evolve.