Software & Software Engineering

Computer Software engineers apply computer science engineering and maths to design, develop and test software.
Software Engineers first analyse user's needs, then they design, construct, test and maintain the needed software or system.

IMPORTANCE OF SOFTWARE ENGINEERING

Software Engineering is the discipline of designing, writing, testing, implementing and maintaining software.it forms the basis of operational design and development of virtually all computer system.
Software Engineering aims to deliver fault free software, on time and within budget, meeting the requirement and needs of the client.

THE PRIMARY GOALS OF SOFTWARE ENGINEERING ARE:

- To improve the quality of the software products.
- To increase the productivity &
- To give job satisfaction to the software engineers.

NEED FOR SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirement and environment on which the software is working.

The following factors contribute to the need of software engineering:

1. Large Software
2. Scalability
3. Cost
4. Dynamic Nature
5. Quality Management

THE NATURE 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.

1. It transforms personal data
2. It manages business information to enhance competitiveness
3. It provides a gateway to worldwide information networks

4. It provides the means for acquiring information.
5. Dramatic Improvements in hardware performance
6. Vast increases in memory and storage capacity
7. A wide variety of exotic input and output options

SOFTWARE

- Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product
- Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for general market.
- The product that software professionals build and then support over the long term.

Software encompasses:

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

(2) Data structures that enable the programs to adequately store and manipulate information.

(3) Documentation that describes the operation and use of the programs.

CHARACTERISTICS OF THE SOFTWARE

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

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

1. heavy usage by multiple users
2. concurrent operation that requires scheduling, resource sharing, and sophisticated process management
3. complex data structures
4. 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 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.
- Product-line software: designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace or address mass consumer markets.

  e.g., word processing, spread sheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications
- Web applications: called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.
- Artificial intelligence software: makes use of nonnumeric algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

LEGACY SOFTWARE

Hundreds of thousands of computer programs fall into one of the seven broad application domains. Some of these are stateof-the-art software—just released to individuals, industry, and government.

But other programs are older, in some cases much older. These 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 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 proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues 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.

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

THE UNIQUE NATURE OF WEBAPPS

In the early days of the World Wide Web, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. Web-based systems and applications were born.

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

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

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

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

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

Performance: 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.

Availability: Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.

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

Content sensitive: The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp

Continuous evolution: Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. 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.

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.

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

SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges, you must recognize a few simple realities:

1. effort should be made to understand the problem before a software solution is developed
2. Design is a pivotal software engineering activity
3. Both quality and maintainability are an outgrowth of good design

   These simple realities lead to one conclusion: software in all of its forms and across all of its application domains should be engineered.

It is all about developing products, using well-defined, scientific principles and methods.

So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Software engineers first analyze user's needs. Then they design, construct, test and maintain the needed software or system.

IEEE defines software engineering as:

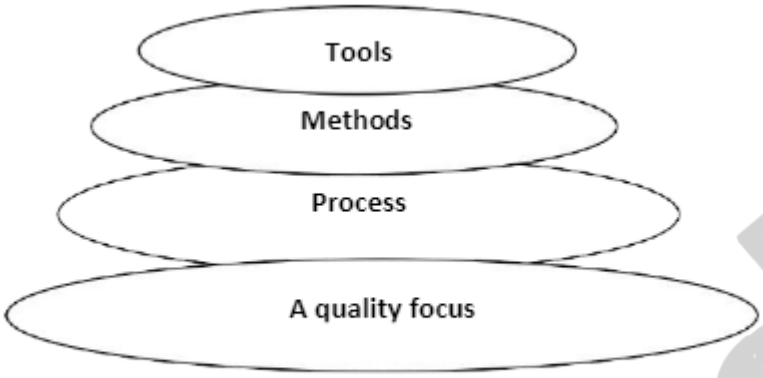
The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:

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

1. Technical methods are applied,
2. Work products are produced,
3. Milestones are established,
4. Quality is ensured, And change is properly managed.



Software Engineering Layers

THE SOFTWARE PROCESS:

- A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied
- An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
- A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

  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:

Communication: Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders'

objectives for the project and to gather requirements that help define software features and functions.

Planning: 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.

Modelling: 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.

Construction: This activity combines code generation (either manual or automated) and the testing that is required uncovering errors in the code.

Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

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

THE SOFTWARE ENGINEERING PRACTICE

THE ESSENCE OF PRACTICE

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 common-sense steps lead to a series of essential questions

1. **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, Oh yeah, I understand, let's get on with solving this thing. Unfortunately, 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?

2. Plan the solution. Now you understand the problem
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?

3.Carry out the plan. 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?

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

• Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?

• Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

GENERAL PRINCIPLE

The word principle as "an important underlying law or assumption required in a system of thought.

Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., communication), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs.

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 haphazard 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. This is not to say that features, even internal 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.

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.

SOFTWARE MYTHS

Software myths is 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 promulgated by experienced practitioners who "know the score."

1) **Management myths:** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. 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.

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

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

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

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "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

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

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

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (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, the cost impact is relatively small.However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification

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

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.

HOW IT ALL STARTS

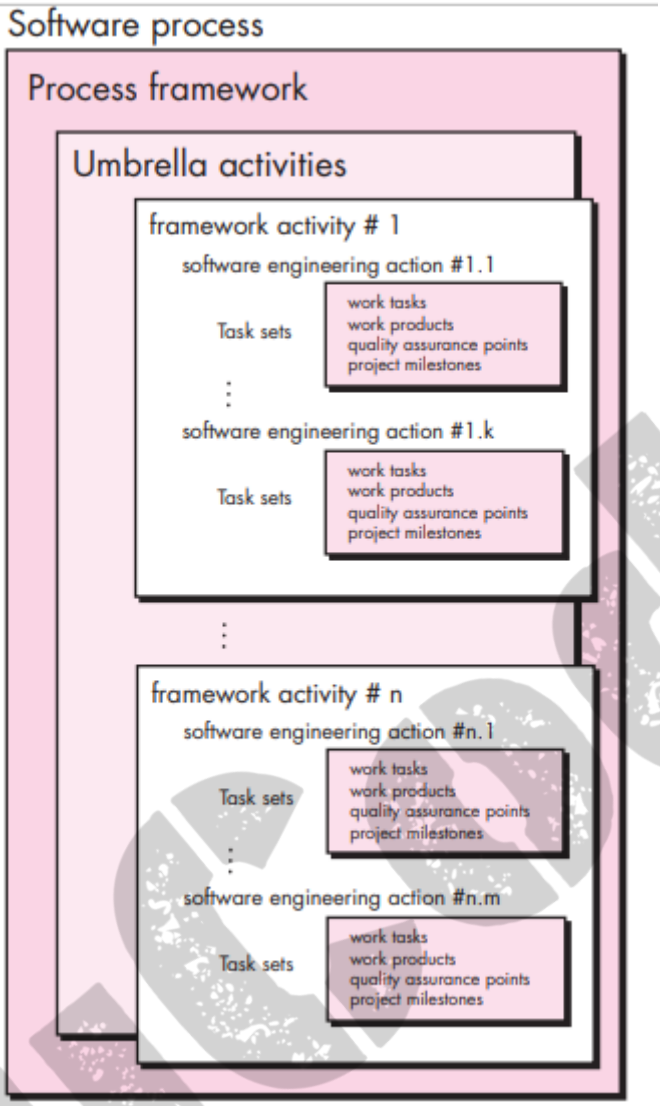
Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a "legacy system" to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system. At the beginning of a software project, the business need is often expressed informally as part of a simple conversation.

PROCESS MODELS

Process Models: A generic process model, Process assessment and improvement, Prescriptive process models, Waterfall model, Incremental process models, Evolutionary process models, Concurrent models, specialized process models.

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

The software process is represented schematically in Figure below; 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 MODEL

A generic process framework for software engineering defines five framework activities—communication, planning, modelling, 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.

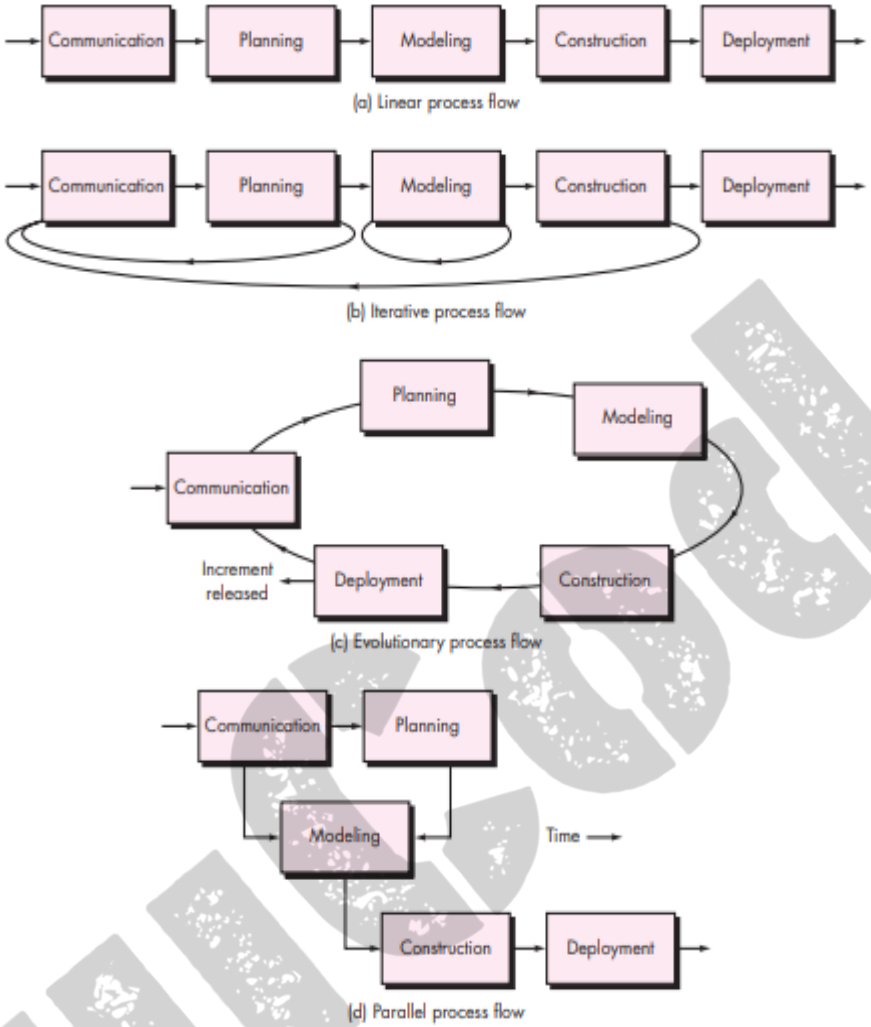
Process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure below

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

An iterative process flow repeats one or more of the activities before proceeding to the next (Figure b).

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 (Figure c).

A parallel process flow (Figure d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



Defining a Framework Activity

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

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass

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 (sometime conflicting) 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

Software engineering action (e.g., elicitation, an action associated with the communication activity) 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. You should choose a task set that best accommodates the needs of the project and the characteristics of your 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

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

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping).

A template for describing a process pattern:

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

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

Type. The pattern type is specified.

There are three types:

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

  An example of a stage pattern might be Establishing Communication. This pattern would incorporate the task pattern Requirements Gathering and others.
- **Task pattern**—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., Requirements Gathering is a task pattern).
- **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?

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 is modified as a consequence of 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: Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

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

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

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

Related Pattern:. 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: Project Team, Collaborative Guidelines, Scope Isolation, Requirements Gathering, Constraint Description, and Scenario Creation.

Known Uses and Example: 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

Process patterns provide an effective mechanism for addressing problems associated with any software process.

PROCESS ASSESSMENT AND IMPROVEMENT:

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

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:

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

- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—** provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment
- **SPICE (ISO/IEC15504)—**a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process
- **ISO 9001:2000** for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

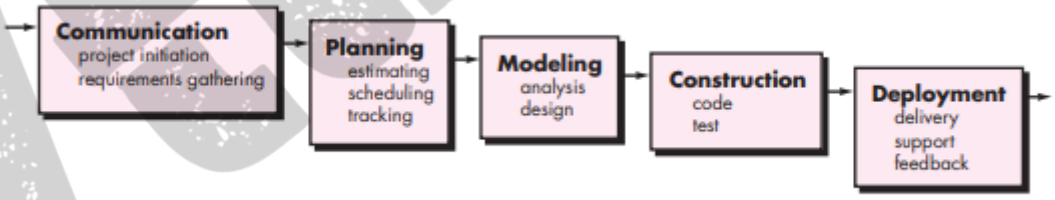
PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.

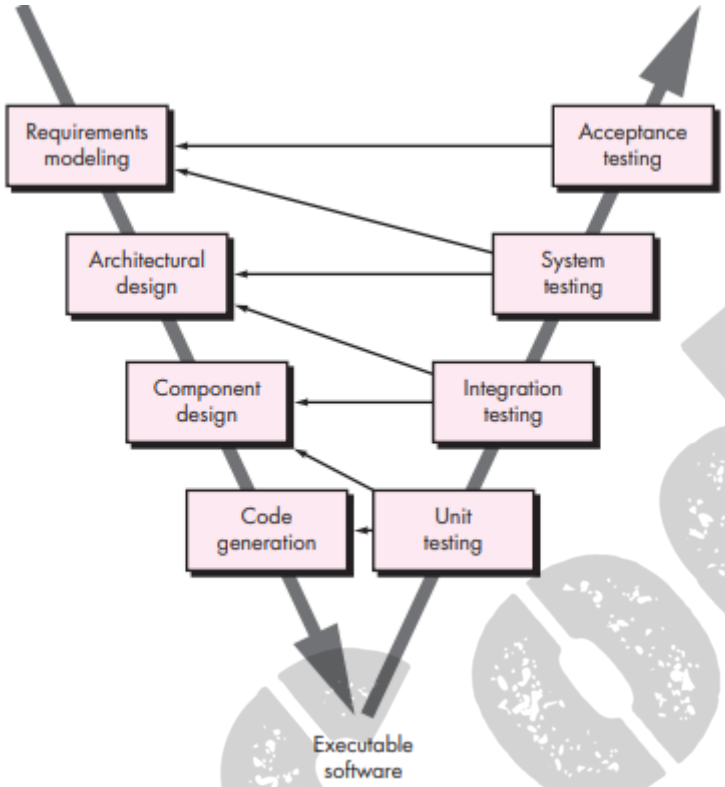
They prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements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

THE WATERFALL MODEL

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modelling, construction, and deployment, culminating in on-going support of the completed software.



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



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

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Incremental Process Models

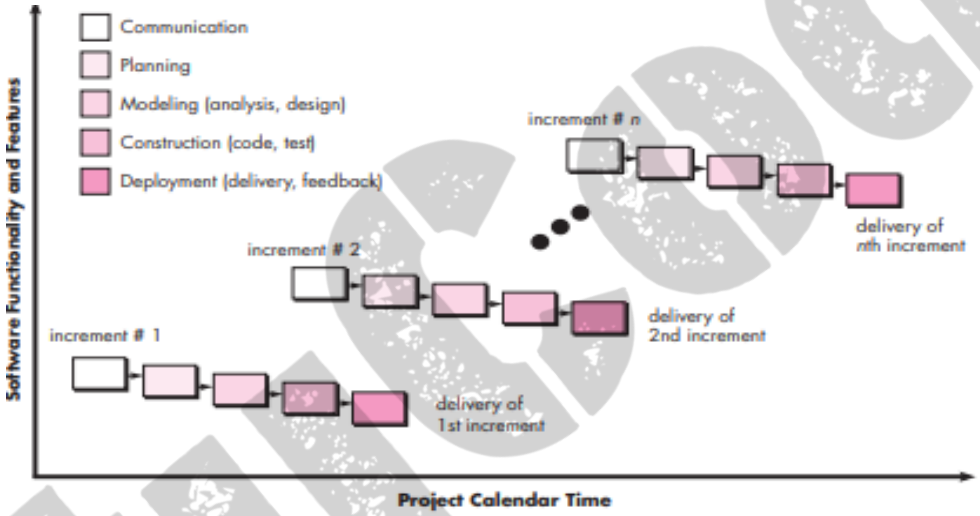
The incremental model combines elements of linear and parallel process flows discussed in. Referring to Figure below, the incremental model applies linear sequences in a

staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. 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

The incremental process model focuses on the delivery of an operational product with each increment

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



Evolutionary Process Models:

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow:

I present two common evolutionary process models.

Prototyping : Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. The prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping figure begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned

quickly, and modelling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users.
The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
prototyping can be problematic for the following reasons:

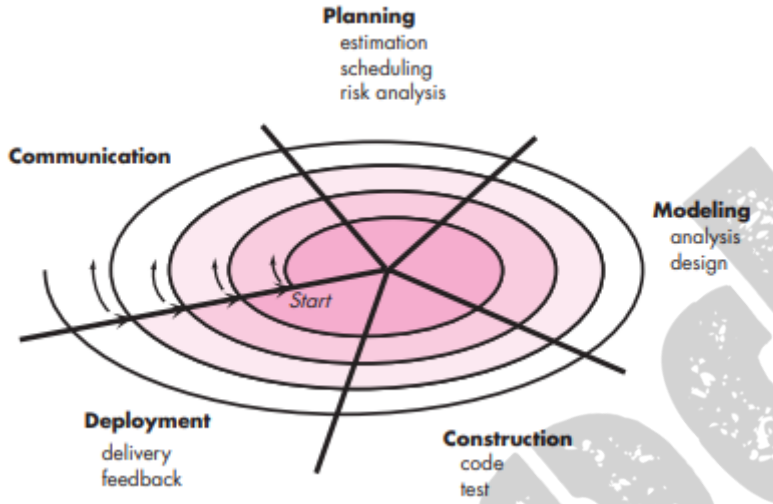
- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
- As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability

The prototyping paradigm

The Spiral Model:

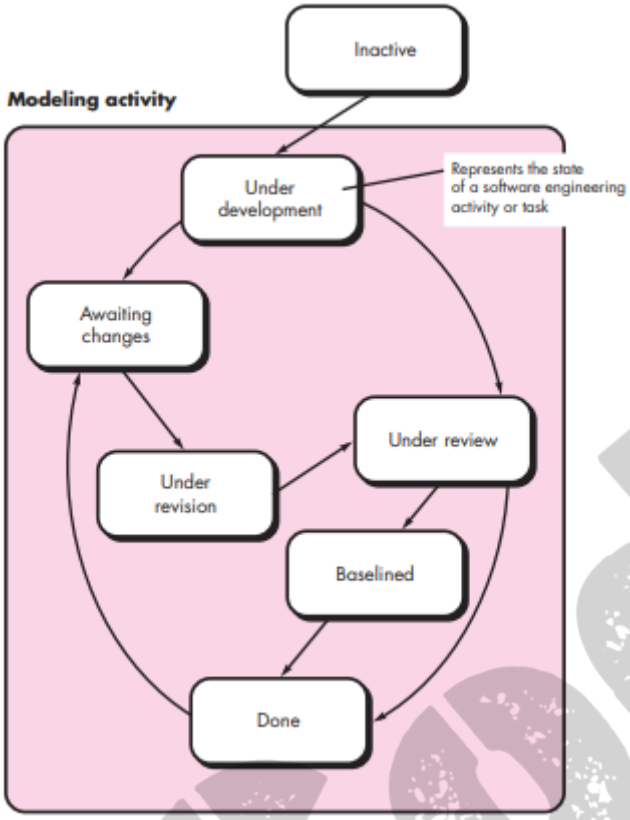
The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software
Each of the framework activities represent one segment of the spiral path illustrated in Figure
As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.
The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

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
The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.



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. Fig provides a schematic representation of one software engineering activity within the modelling activity using a concurrent modelling approach. The activity—modelling—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 has completed its first iteration and exists in the awaiting changes state. The modelling 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 modelling activity moves from the under development state into the awaiting changes state



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

A Final Word on Evolutionary Processes

I have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction
Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses.

- The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product
- Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affect
- Third, software processes should be focused on flexibility and extensibility rather than on high quality.

indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality

The intent of evolutionary models is to develop high-quality software14 in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters

Specialized process models

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen

- **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 [Nie92], demanding an iterative approach to the creation of software. 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, software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost

- **The Formal Methods model:**

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

The formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

• 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:**

The builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modelled as components (e.g., objectoriented classes) and then constructed within the context of a system 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

It is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. 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 of localized software components and yet, aspects have a direct impact on these components.