Module 1

Define Software Engineering? Explain briefly characteristics of WebApps.

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.

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

Explain the failure curve for software.

Explain various (5 to 7) software application domains

System software: a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,4 information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

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.

Engineering/scientific software: has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

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.

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

Web applications: so 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 non-numerical 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

Open-world computing: the rapid growth of wireless networking may soon lead to true pervasive, distributed computing..

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.

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.

With a neat diagram explain V Model.

Answer given below

Explain the activities a generic process framework for software engineering encompasses

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. 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 to uncover 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.

Brief on typical umbrella activities.

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.

Explain the principles that focus on software engineering practice.

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

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. . . .

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 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.

The Sixth Principle: Plan Ahead for 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.

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.

Describe the software process flow (any one may be asked)

The software process is represented schematically in below 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, 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.

This aspect—called 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.

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.

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder.

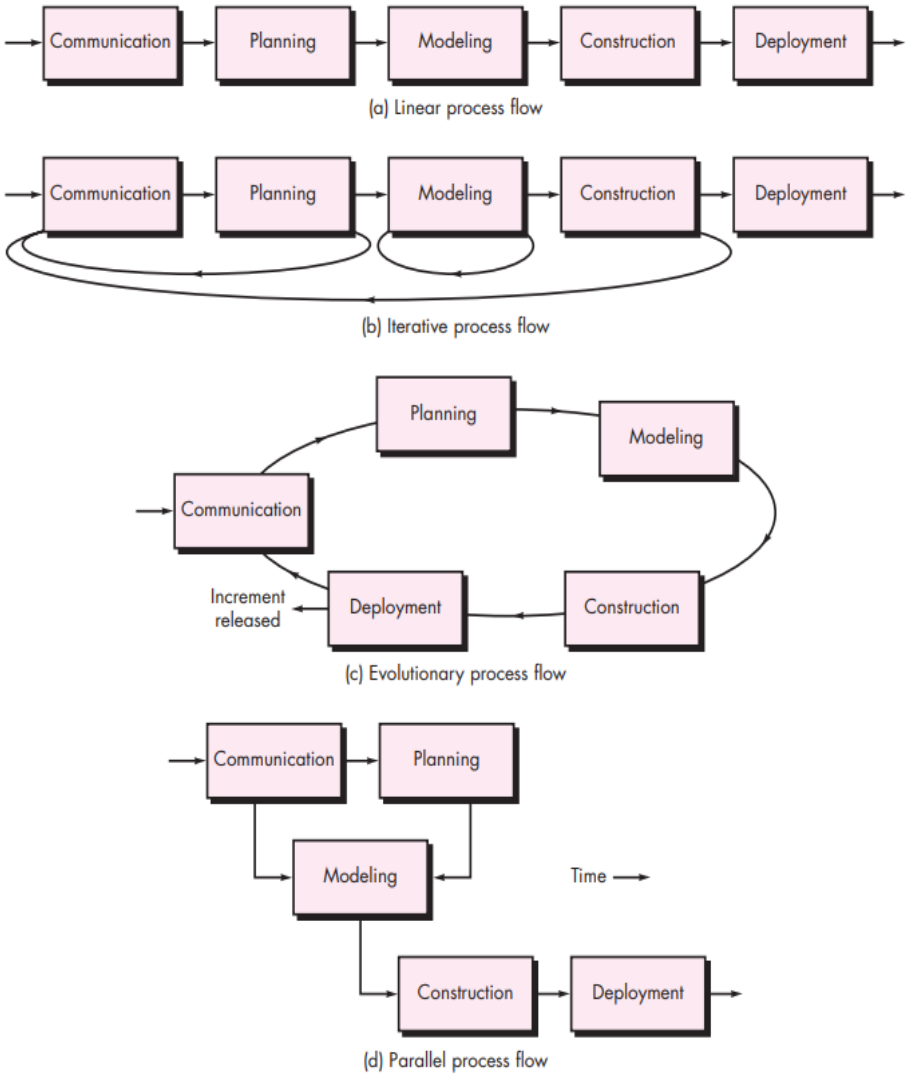
Therefore, the only necessary action is phone conversation, and the work tasks (the task set) that this action encompasses are:

1. Make contact with stakeholder via telephone.

2. Discuss requirements and take notes. How does a framework activity change as the nature of the project changes?

3. Organize notes into a brief written statement of requirements.

4. E-mail to stakeholder for review and approval.



Explain the waterfall/ Evolutionary/ Concurrent model with a neat diagram.

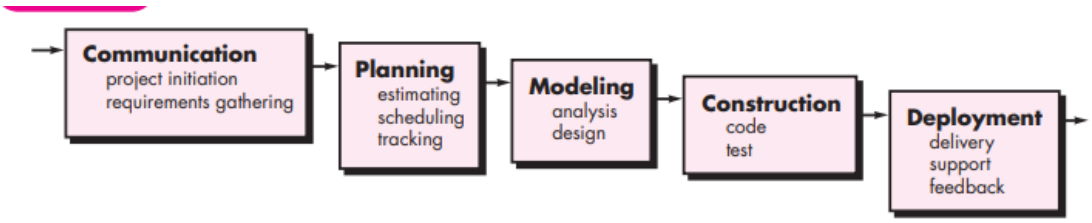
The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modelling, construction, and deployment, culminating in on-going support of the completed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

The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy. Among 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.



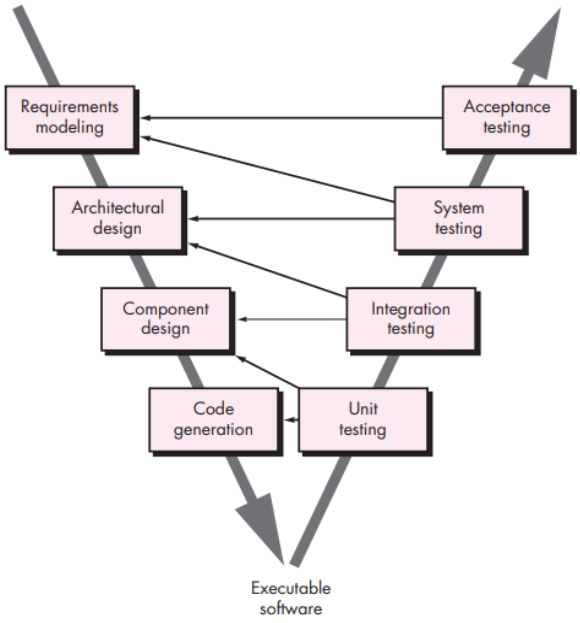
A variation in the representation of the waterfall model is called the V-model.

Represented in figure, the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modelling, 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.



There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process.

In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.

In such cases, we can choose a process model that is designed to produce the software in increments.

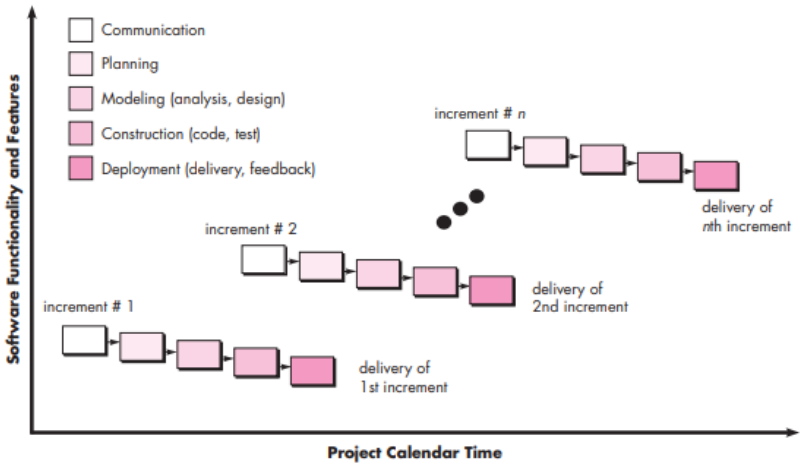
The incremental model combines elements of linear and parallel process flows.

Referring to below figure, the incremental model applies linear sequences in a staggered fashion as calendar time progresses.

Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.

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.

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.

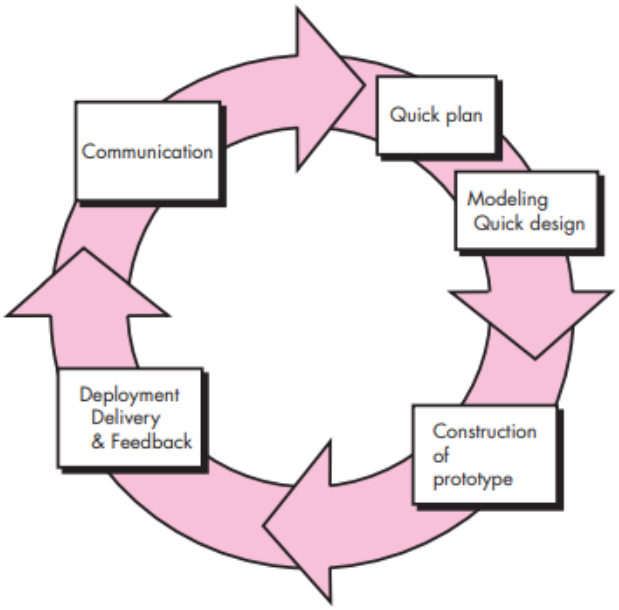


Evolutionary models are iterative.

They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

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 noted in this chapter.

Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.



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

A prototyping iteration is planned quickly, and modelling (in the form of a “quick design”) occurs.

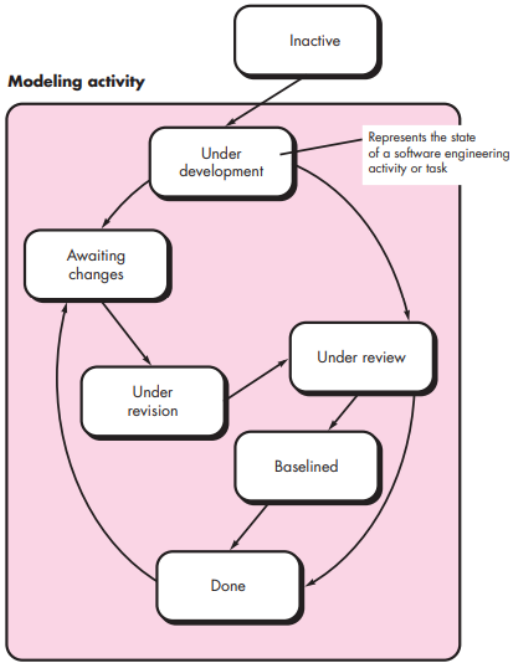
A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.

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

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built.



The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models.

Figure above 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.

The modelling activity 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 modelling is applicable to all types of software development and provides an accurate picture of the current state of a project.

Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network.

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.

Explain (any one may be asked) specialized process model.

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, 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.

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.

Elaborate process framework activities.

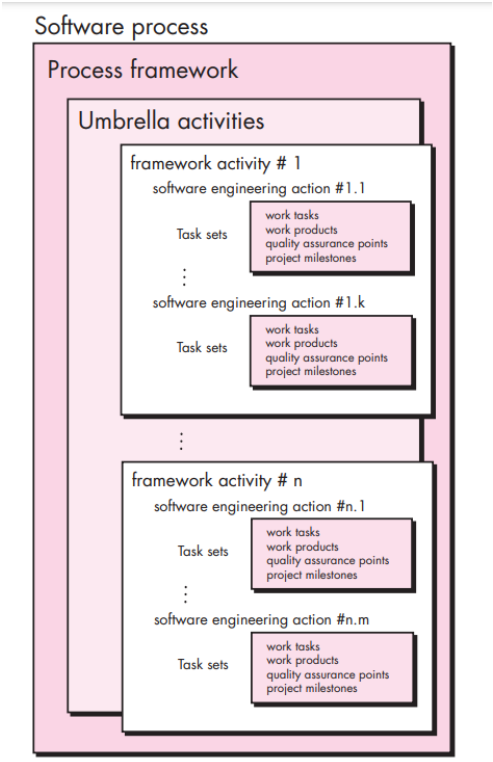
Process Models: A generic process model, Process assessment and improvement, Prescriptive process models, Waterfall model, Incremental process models, Evolutionary process models, Concurrent models, specialized process models.

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.

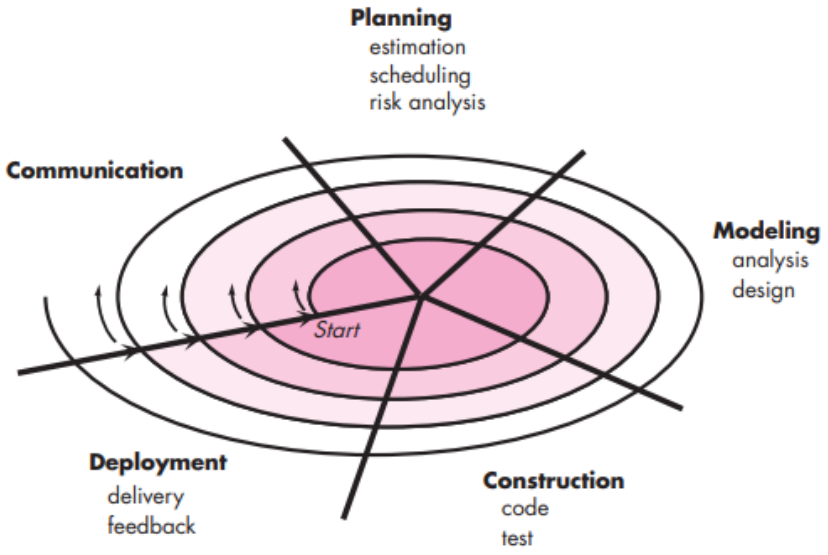
The software process is represented schematically in Figure below; each framework activity is populated by a set of software engineering actions.

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.



With a neat diagram explain the Spiral model.

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 model or prototype.

During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team.

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 a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. \

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 iterations10 until concept development is complete.

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.

The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product.

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 spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic. But like other paradigms, the spiral model is not a panacea.

It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success.

With a neat diagram explain Prototyping – an evolutionary model

Answer is in above section

What are the reasons for which the legacy systems evolve?

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.

Explain the factors considered for adopting a process model

• Overall flow of activities, actions, and tasks and the interdependencies among them

• Degree to which actions and tasks are defined within each framework activity

• Degree to which work products are identified and required

• Manner in which quality assurance activities are applied

• Manner in which project tracking and control activities are applied

• Overall degree of detail and rigor with which the process is described

• Degree to which the customer and other stakeholders are involved with the project

• Level of autonomy given to the software team • Degree to which team organization and roles are prescribed

---

Describe the software engineering practice with all its phases

The Essence of Practice

1. Understand the problem (communication and analysis).

2. Plan a solution (modelling and software design).

3. Carry out the plan (code generation).

4. Examine the result for accuracy (testing and quality assurance).

Understand the problem

• 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?

Plan the solution.

• 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?

Carry out the plan

• 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?

Examine the result

• 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?

Brief on various Software Myths

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).

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

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.

==========

Module 2

Briefly elaborate various phases in requirement engineering.

Inception, Elicitation, Elaboration and so on (description given below)

Discuss the questionnaires to be addressed during validating requirements.

• Is each requirement consistent with the overall objectives for the system/product?

• Have all requirements been specified at the proper level of abstraction?

• Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?

• Is each requirement bounded and unambiguous?

• Does each requirement have attribution?

• Do any requirements conflict with other requirements?

• Is each requirement achievable in the technical environment that will house the system or product?

Is each requirement testable, once implemented?

• Does the requirements model properly reflect the information, function, and behavior of the system to be built?

• Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

• Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated?

Explain the distinct tasks involved in Requirements Engineering.

Inception. How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort.

Elicitation. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis

A number of problems that are encountered as elicitation occurs.

- **Problems of scope**. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives
- **Problems of understanding**. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- **Problems of volatility**. The requirements change over time.

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

Specification. In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification5 to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds

---

Discuss about establishing the ground work regarding requirements gathering.

To establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution

Identifying Stakeholders: Stakeholders can be business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail

Recognizing Multiple Viewpoints: Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. Each of these stakeholders (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

Working toward Collaboration: If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

Asking the First Questions: Questions asked at the inception of the project should be "context free"

Who is behind the request for this work?

Who will use the solution?

What will be the economic benefit of a successful solution?

Is there another source for the solution that you need?

Comment on Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process"

QFD identifies three types of requirements

Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

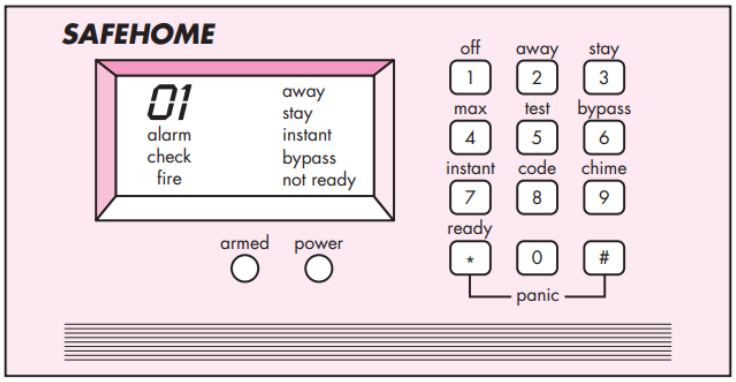
Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

Discuss the template for detailed descriptions of use cases considering SAFEHOME system as case study

Template for detailed descriptions of use cases: Use case: InitiateMonitoring

Primary actor: Homeowner.



Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.
Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions.
Scenario: 1. Homeowner: observes control panel

2. Homeowner: enters password

3. Homeowner: selects "stay" or "away"

4. Homeowner: observes read alarm light to indicate that SafeHome has been armed

Exceptions: 1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.

2. Password is incorrect (control panel beeps once): homeowner reenters correct password.

3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.

4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.

5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

Priority: Essential, must be implemented When available: First increment Frequency of use: Many times per day

Channel to actor: Via control panel interface

Secondary actors: Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues: 1. Should there be a way to activate the system without the use of a password or with an abbreviated password?

2. Should the control panel display additional text messages?

3. How much time does the homeowner have to enter the password from the time the first key is pressed?

4. Is there a way to deactivate the system before it actually activates?

Use cases for other homeowner interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.