Unit –1: Software Process Models

Define Software Engineering.

Software engineering is a disciplined approach to designing, developing, and maintaining software systems. It involves applying engineering principles and practices to the entire software development lifecycle, including requirements analysis, design, coding, testing, deployment, and maintenance.

Defining software

Software refers to a collection of programs, instructions, and data that enable computers to perform specific tasks or functions. It encompasses both applications, such as word processors, web browsers, and video games, as well as system software like operating systems, device drivers, and utilities.

Software Application Domain

i. System Software:

System software is responsible for managing and controlling the operation of computer hardware and providing a platform for running application software. It includes operating systems (such as Windows, macOS, and Linux), device drivers, firmware, and utility programs. Operating systems manage resources such as memory, processors, input/output devices, and provide services like file management, networking, and security. Device drivers allow the operating system to communicate with hardware devices like printers, graphics cards, and storage drives. Firmware is low-level software embedded in hardware components, providing essential functionality like booting up the system. Utility programs perform tasks like disk defragmentation, antivirus scanning, and system maintenance.

ii. Application Software:

Application software refers to programs designed to perform specific tasks or provide services for end-users. Unlike system software, application software is

targeted towards fulfilling user needs rather than managing computer hardware. Examples of application software include word processors (Microsoft Word, Google Docs), spreadsheets (Microsoft Excel, Google Sheets), web browsers (Google Chrome, Mozilla Firefox), multimedia players (VLC, Windows Media Player), graphics editors (Adobe Photoshop, GIMP), and database management systems (Oracle, MySQL). Application software serves various purposes, ranging from productivity tools and entertainment to communication and creativity.

iii. Embedded Software:

Embedded software refers to software that is embedded within hardware devices and is dedicated to controlling the functions of that device. It is often found in embedded systems such as consumer electronics, automotive systems, medical devices, industrial equipment, and household appliances. Embedded software is typically tailored to the specific requirements and constraints of the hardware platform it runs on, focusing on real-time operation, efficiency, and reliability. Examples of devices with embedded software include smartphones, digital cameras, microwave ovens, automotive engine control units (ECUs), and industrial robots.

iv. Web Application:

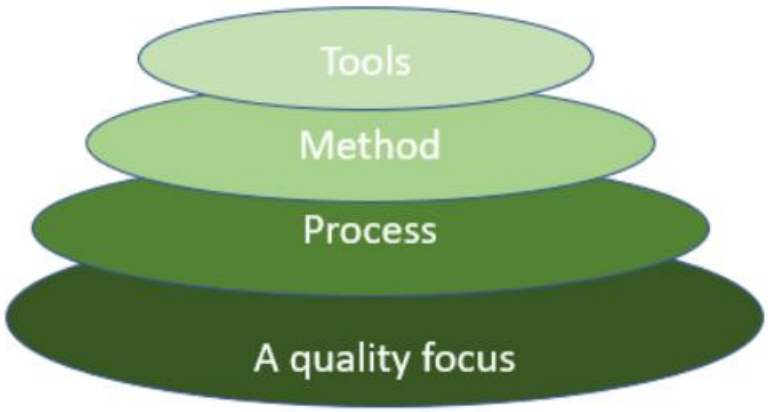
A web application is software accessed and used via a web browser over a network, typically the Internet. Unlike traditional desktop applications that are installed locally on a user's device, web applications are hosted on remote servers and accessed through a web browser interface. Users interact with web applications through graphical user interfaces (GUIs) presented in the browser, which communicate with the server-side components to process requests and deliver responses. Web applications encompass a wide range of functionalities, including online banking systems, e-commerce platforms, social media networks, email services, and collaboration tools like Google Docs and Trello.

v. Artificial Intelligence Software:

Artificial intelligence (AI) software refers to programs that utilize machine learning, natural language processing, computer vision, and other AI techniques to

perform tasks that typically require human intelligence. AI software can analyze large datasets, recognize patterns, make predictions, understand and generate natural language, and even mimic human-like behavior in certain domains. Examples of AI software include virtual assistants (such as Siri, Alexa, and Google Assistant), recommendation systems (like those used by Netflix and Amazon), image recognition software, language translation tools, and autonomous vehicles. AI software is increasingly being integrated into various applications to automate tasks, enhance decision-making, and improve user experiences.

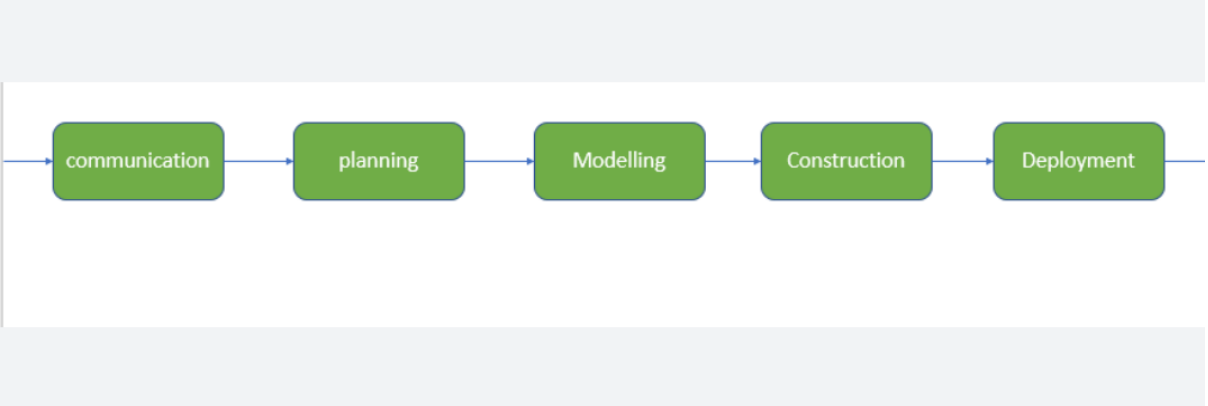
Software Engineering – A layered Approach:



Layered technology is divided into four parts:

1. A quality focus: It defines the continuous process improvement principles of software. It provides integrity that means providing security to the software so that data can be accessed by only an authorized person, no outsider can access the data. It also focuses on maintainability and usability.

2. Process: It is the foundation or base layer of software engineering. It is key that binds all the layers together which enables the development of software before the deadline or on time. Process defines a framework that must be established for the effective delivery of software engineering technology. The software process covers all the activities, actions, and tasks required to be carried out for software development.



Process activities are listed below:-

- Communication: It is the first and foremost thing for the development of software. Communication is necessary to know the actual demand of the client.
- Planning: It basically means drawing a map for reduced the complication of development.
- Modeling: In this process, a model is created according to the client for better understanding.
- Construction: It includes the coding and testing of the problem.
- Deployment:- It includes the delivery of software to the client for evaluation and feedback.

3. Method: During the process of software development the answers to all "how-to-do" questions are given by method. It has the information of all the tasks which includes communication, requirement analysis, design modeling, program construction, testing, and support.

4. Tools: Software engineering tools provide a self-operating system for processes and methods. Tools are integrated which means information created by one tool can be used by another.

Describe Generic Framework Activity:

A generic framework activity refers to a fundamental process or task that is common across various software development projects, regardless of the specific technology, domain, or methodology being employed. These activities provide a structured approach to software development and help guide the overall project lifecycle. Here's a description of some generic framework activities commonly found in software engineering:

Requirements Elicitation and Analysis: This activity involves gathering, understanding, and documenting the requirements of the software system from stakeholders. It includes

techniques such as interviews, surveys, workshops, and analysis of existing documentation to identify and prioritize system features, constraints, and user needs.

System Design: System design encompasses the process of transforming requirements into a blueprint or architecture for the software system. It involves defining the overall structure, components, interfaces, and behavior of the system, as well as selecting appropriate technologies, patterns, and frameworks to support its implementation.

Implementation: Implementation involves writing code to realize the design of the software system. It includes tasks such as coding, unit testing, integration testing, debugging, and code reviews. Developers translate design specifications into executable code, following coding standards, best practices, and quality guidelines.

Testing: Testing activities focus on verifying and validating the software system to ensure that it meets its requirements and performs as expected. It includes various types of testing, such as unit testing, integration testing, system testing, acceptance testing, and regression testing. Testing aims to identify defects, errors, and deviations from expected behavior, allowing for their correction and improvement.

Deployment: Deployment involves releasing the software system into production or operational environments, making it available for end-users to use. It includes tasks such as installation, configuration, setup, and rollout of the software, as well as data migration, user training, and documentation. Deployment ensures that the software is installed and configured correctly and that users can access and use it effectively.

Maintenance: Maintenance activities involve making modifications, enhancements, and updates to the software system after it has been deployed. It includes tasks such as bug fixing, feature additions, performance tuning, and security patches. Maintenance aims to keep the software operational, reliable, and aligned with changing user needs and technological advancements.

These generic framework activities provide a high-level structure for organizing and managing software development projects. They serve as a guide for project planning, execution, and control, helping teams to systematically address key aspects of the software development lifecycle and deliver high-quality software products on time and within budget.

Software Development Models:

1.Waterfall Model:

Waterfall model is a famous and good version of SDLC(System Development Life Cycle) for software engineering. The waterfall model is a linear and sequential model, which means that a development phase cannot begin until the previous phase is completed. We cannot overlap phases in waterfall model.

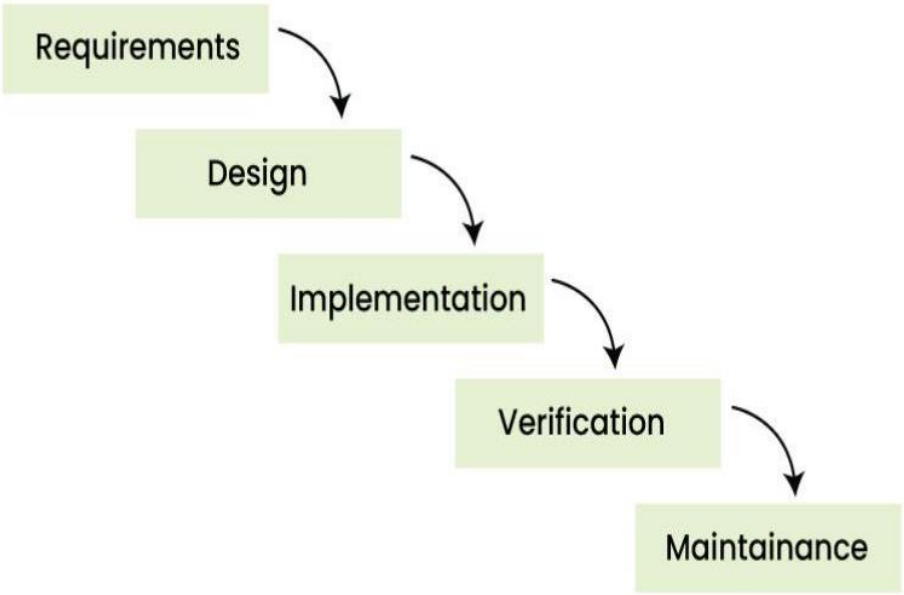
We can imagine waterfall in the following way:-

“Once the water starts flowing over the edge of the cliff, it starts falling down the mountain and the water cannot go back up.”

Similarly waterfall model also works, once one phase of development is completed then we move to the next phase but cannot go back to the previous phase.

In the waterfall model, the output of one phase serves as the input for the other phase.

Phases of Waterfall model:



1. **Requirement phase:-** Requirement phase is the first phase of the waterfall model. In this phase the requirements of the system are collected and documented. This phase is very crucial because the next phases are based on this phase.
2. **Design phase:-** Design phase is based on the fact how the software will be built. The main objective of the design phase is to prepare the blueprint of the software system so that no problems are faced in the coming phases and solutions to all the requirements in the requirement phase are found.
3. **Implementation phase:-** In this phase, hardware, software and application programs are installed and the database design is implemented. Before the database design can be implemented, the software has to go through a testing, coding, and debugging process. This is the longest lasting phase in waterfall.
4. **Verification phase:-** In this phase the software is verified and it is evaluated that we have created the right product. In this phase, various types of testing are done and every area of the software is checked. It is believed that if we do not verify the software properly and there is any defect in it then no one will use it, hence verification is very important. One advantage of verification is that it reduces the risk of software failure.
5. **Maintenance phase:-** This is the last phase of waterfall. When the system is ready and users start using it, then the problems that arise have to be solved time-to-time.

Taking care of the finished software and maintaining it as per time is called maintenance.

Advantages of Waterfall Model

- This model is simple and easy to understand.
- This is very useful for small projects.
- This model is easy to manage.
- The end goal is determined early.
- Each phase of this model is well explained.
- It provides a structured way to do things.
- This is a base model, all the SDLC models that came after this were created keeping this in mind, although they worked to remove its shortcomings.
- In this model, we can move to the next phase only after the first phase is successfully completed so that there is no overlapping between the phases.

Disadvantages of Waterfall Model

- In this model, complete and accurate requirements are expected at the beginning of the development process.
- Working software is not available for very long during the development life cycle.
- We cannot go back to the previous phase due to which it is very difficult to change the requirements.
- Risk is not assessed in this, hence there is high risk and uncertainty in this model.
- In this the testing period comes very late.
- Due to its sequential nature this model is not realistic in today's world.
- This is not a good model for large and complex projects.

2. Incremental Model:

In Incremental Model, the software development process is divided into several increments and the same phases are followed in each increment. In simple language, under this model a complex project is developed in many modules or builds.

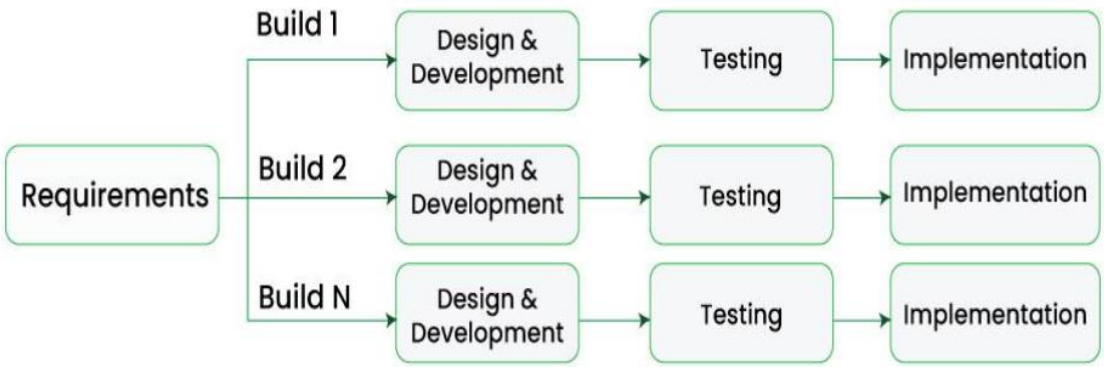
For example, we collect the customer's requirements, now instead of making the entire software at once, we first take some requirements and based on them create a module or function of the software and deliver it to the customer. Then we take some more requirements and based on them add another module to that software.

Similarly, modules are added to the software in each increment until the complete system is created. However, the requirements for making a complex project in multiple iterations/parts should be clear.

If we understand the entire principle of Incremental methodology, then it starts by developing an initial implementation, then user feedback is taken on it, and it is developed through

several versions until an accepted system is developed. Important functionalities of the software are developed in the initial iterations.

Each subsequent release of a software module adds functions to the previous release. This process continues until the final software is obtained.



Following are the different phases of Incremental Model:-

Communication: In the first phase, we talk face to face with the customer and collect his mandatory requirements. Like what functionalities does the customer want in his software, etc.

Planning: In this phase the requirements are divided into multiple modules and planning is done on their basis.

Modeling: In this phase the design of each module is prepared. After the design is ready, we take a particular module among many modules and save it in DDS (Design Document Specification). Diagrams like ERDs and DFDs are included in this document.

Construction: Here we start construction based on the design of that particular module. That is, the design of the module is implemented in coding. Once the code is written, it is tested.

Deployment: After the testing of the code is completed, if the module is working properly then it is given to the customer for use. After this, the next module is developed through the same phases and is combined with the previous module. This makes new functionality available to the customer. This will continue until complete modules are developed.

Advantages of Incremental Model

- Important modules/functions are developed first and then the rest are added in chunks.

- Working software is prepared quickly and early during the software development life cycle (SDLC).
- This model is flexible and less expensive to change requirements and scope.
- The customer can respond to each module and provide feedback if any changes are needed.
- Project progress can be measured.
- It is easier to test and debug during a short iteration.
- Errors are easy to identify.

Disadvantages of Incremental Model

- Management is a continuous activity that must be handled.
- Before the project can be dismantled and built incrementally,
- The complete requirements of the software should be clear.
- This requires good planning and designing.
- The total cost of this model is higher.

3. Prototype model:

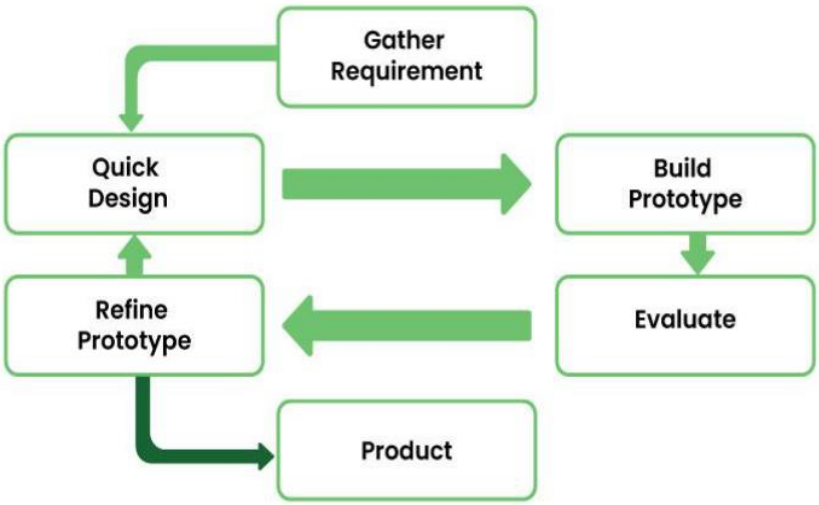
Prototype model is an activity in which prototypes of software applications are created. First a prototype is created and then the final product is manufactured based on that prototype.

The prototype model was developed to overcome the shortcomings of the waterfall model.

This model is created when we do not know the requirements well.

The specialty of this model is that this model can be used with other models as well as alone.

One problem in this model is that if the end users are not satisfied with the prototype model, then a new prototype model is created again, due to which this model consumes a lot of money and time.



The prototype model has the following phases:-

Requirement gathering: The first step of prototype model is to collect the requirements, although the customer does not know much about the requirements but the major requirements are defined in detail.

Build the initial prototype: In this phase the initial prototype is built. In this some basic requirements are displayed and user interface is made available.

Review the prototype: When the construction of the prototype is completed, it is presented to the end users or customer and feedback is taken from them about this prototype. This feedback is used to further improve the system and possible changes are made to the prototype.

Revise and improve the prototype: When feedback is taken from end users and customers, the prototype is improved on the basis of feedback. If the customer is not satisfied with the prototype, a new prototype is created and this process continues until the customer gets the prototype as per his desire.

Advantages of Prototype model :-

- Prototype Model is suggested to create applications whose prototype is very easy and which always includes human machine interaction within it.
- When we know only the general objective of creating software, but we do not know anything in detail about input, processing and output. Then in such a situation we make it a Prototype Model.
- When a software developer is not very sure about the capability of an algorithm or its adaptability to an operating system, then in this situation, using a prototype model can be a better option.

Disadvantages of Prototype model :-

- When the first version of the prototype model is ready, the customer himself often wants small fixes and changes in it rather than rebuilding the system. Whereas if the system is redesigned then more quality will be maintained in it.
- Many compromises can be seen in the first version of the Prototype Model.
- Sometimes a software developer may make compromises in his implementation, just to get the prototype model up and running quickly, and after some time he may become comfortable with making such compromises and may forget that it is completely inappropriate to do so.

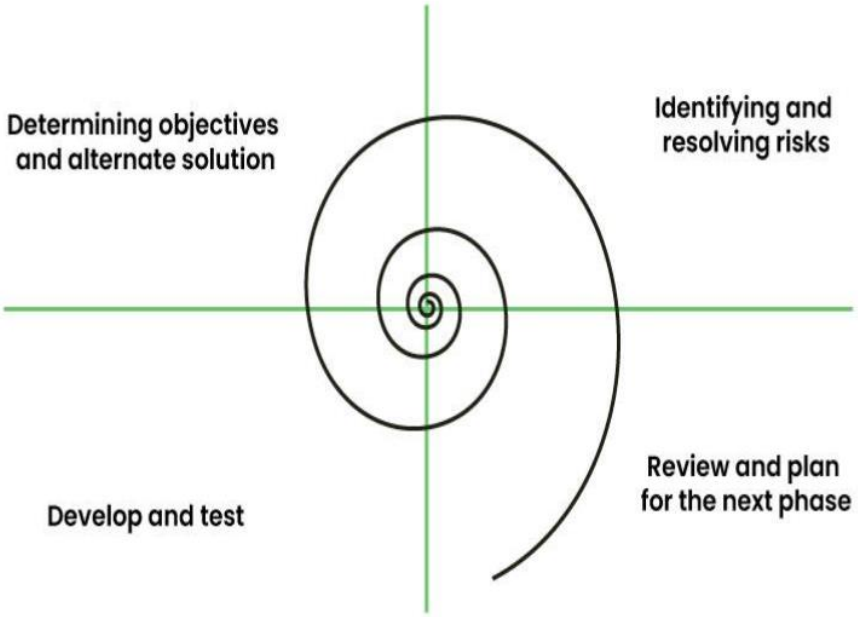
4.Spiral Model:

Spiral model is a software development process model. This model has characteristics of both iterative and waterfall models. This model is used in projects which are large and complex. This model was named spiral because if we look at its figure, it looks like a spiral, in which a

long curved line starts from the center point and makes many loops around it. The number of loops in the spiral is not decided in advance but it depends on the size of the project and the changing requirements of the user. We also call each loop of the spiral a phase of the software development process.

A software project goes through these loops again and again in iterations. After each iteration a more and more complete version of the software is developed. The most special thing about this model is that risks are identified in each phase and they are resolved through prototyping. This feature is also called Risk Handling.

Since it also includes the approaches of other SDLC models, it is also called Meta Model. It was first developed by Barry Boehm in 1986.



In Spiral Model the entire process of software development is described in four phases which are repeated until the project is completed. Those phases are as follows:-

Determining objectives and alternate solutions: In the first phase, whatever requirements the customer has related to the software are collected. On the basis of which objectives are identified and analyzed and various alternative solutions are proposed.

Identifying and resolving risks: In this phase, all the proposed solutions are assessed and the best solution is selected. Now that solution is analyzed and the risks related to it are identified. Now the identified risks are resolved through some best strategy.

Develop and test: Now the development of software is started. In this phase various features are implemented, that is, their coding is done. Then those features are verified through testing.

Review and plan for the next phase: In this phase the developed version of the software is given to the customer and he evaluates it. Gives his feedback and tells new requirements. Finally planning for the next phase (next spiral) is started.

Advantages of Spiral Model:-

- If we have to add additional functionality or make any changes to the software, then through this model we can do so in the later stages also.
- Spiral model is suitable for large and complex projects.
- It is easy to estimate how much the project will cost.
- Risk analysis is done in each phase of this model.
- The customer can see the look of his software only in the early stages of the development process.
- Since continuous feedback is taken from the customer during the development process, the chances of customer satisfaction increases.

Disadvantage of Spiral Model:-

- This is the most complex model of SDLC, due to which it is quite difficult to manage.
- This model is not suitable for small projects.
- The cost of this model is quite high.
- It requires more documentation than other models.
- Experienced experts are required to evaluate and review the project from time to time.
- Using this model, the success of the project depends greatly on the risk analysis phase.

5.Agile Model:

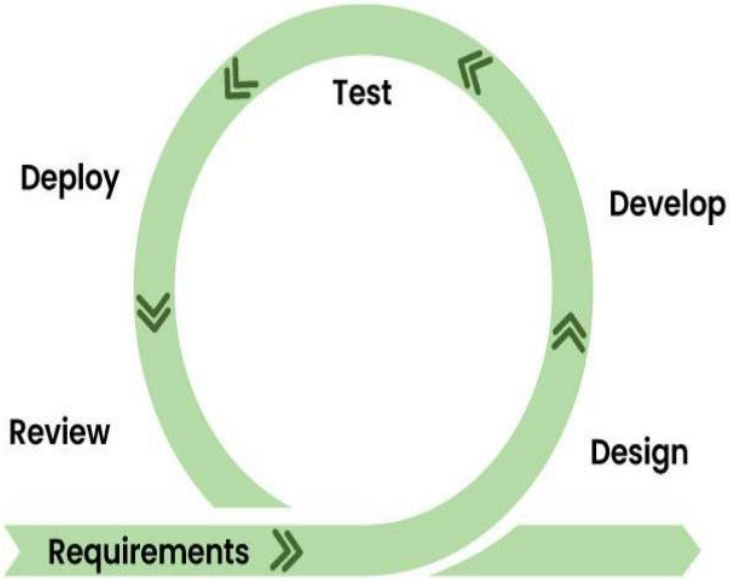
Agile model is a combination of iterative and incremental models, that is, it is made up of iterative and incremental models.

In Agile model, focus is given to process adaptability and customer satisfaction.

In earlier times, iterative waterfall model was used to create software. But in today's time developers have to face many problems. The biggest problem is that in the middle of software development, the customer asks to make changes in the software. It takes a lot of time and money to make these changes.

So to overcome all these shortcomings, the agile model was proposed in the 1990s.

The agile model was created mainly to make changes in the middle of software development so that the software project can be completed quickly.



In the agile model, the software product is divided into small incremental parts. In this, the smallest part is developed first and then the larger one.

And each incremental part is developed over iteration.

Each iteration is kept small so that it can be easily managed. And it can be completed in two-three weeks. Only one iteration is planned, developed and deployed at a time.

Principles of Agile model:-

There is a customer representative in the development team to maintain contact with the customer during software development and to understand the requirement. When an iteration is completed, stakeholders and customer representatives review it and re-evaluate the requirements.

Demo of working software is given to understand the customer's requirements. That is, it does not depend only on documentation.

Incremental versions of the software have to be delivered to the customer representative after a few weeks.

In this model it is advised that the size of the development team should be small (5 to 9 people) so that the team members can communicate face to face.

Agile model focuses on the fact that whenever any changes have to be made in the software, it should be completed quickly.

In agile development, two programmers work together. One programmer does the coding and the other reviews that code. Both the programmers keep changing their tasks, that is, sometimes one does coding and sometimes someone reviews.

Agile has the following models:-

- Scrum
- Crystal methods
- DSDM
- Feature driven development (FDD
- Lean software development
- Extreme programming (xp)

Advantages of Agile Model:-

- In this, two programmers work together due to which the code is error free and there are very few mistakes in it.
- In this the software project is completed in a very short time.
- In this the customer representative has an idea of each iteration so that he can easily change the requirement.
- This is a very realistic approach to software development.
- In this, focus is given on teamwork.
- There are very few rules in this and documentation is also negligible.
- There is no need for planning in this.
- It can be managed easily.
- It provides flexibility to developers.

Disadvantages of Agile Model:-

- It cannot handle complex dependencies.
- Due to lack of formal documentation in this, there is confusion in development.
- It mostly depends on the customer representative, if the customer representative gives any wrong information then the software can become wrong.
- Only experienced programmers can take any decision in this. New programmers cannot take any decision.
- In the beginning of software development, it is not known how much effort and time will be required to create the software.

Extreme Programming (XP):

Extreme Programming (XP) is an agile software development methodology that focuses on delivering high-quality software quickly and continuously adapting to changing requirements. XP was introduced by Kent Beck in the late 1990s and emphasizes values such as simplicity, communication, feedback, and courage. Key practices of XP include:

Pair Programming: Two programmers work together at one computer, continuously reviewing each other's work and providing immediate feedback.

Test-Driven Development (TDD): Developers write automated tests before writing code, guiding the design and ensuring that the code meets requirements.

Continuous Integration: Developers integrate their code frequently into a shared repository, with automated builds and tests triggered after each integration.

Refactoring: Regularly improving the design and structure of the code without changing its external behavior, keeping the codebase clean and maintainable.

Simple Design: Focusing on the simplest solution that meets current requirements, avoiding unnecessary complexity.

Collective Code Ownership: All team members are responsible for the quality and maintenance of the codebase, encouraging collaboration and knowledge sharing.

XP is well-suited for small to medium-sized teams working on projects with rapidly changing requirements, where flexibility, adaptability, and close collaboration are essential.

Scrum:

Scrum is an agile framework for managing and delivering complex projects, introduced by Jeff Sutherland and Ken Schwaber in the early 1990s. Scrum emphasizes iterative development, transparency, inspection, and adaptation. In Scrum, work is organized into fixed-length iterations called sprints, typically lasting 1-4 weeks. Key roles, artifacts, and ceremonies in Scrum include:

Scrum Team: A cross-functional team consisting of developers, a Scrum Master, and a Product Owner, collectively responsible for delivering a potentially shippable product increment at the end of each sprint.

Product Backlog: A prioritized list of user stories or features representing the requirements of the product, maintained by the Product Owner.

Sprint Planning: A meeting at the beginning of each sprint where the Scrum Team plans the work to be done during the sprint, based on the items from the Product Backlog.

Daily Stand-up (Daily Scrum): A brief daily meeting where team members synchronize their activities, discuss progress, and identify any impediments.

Sprint Review: A meeting at the end of each sprint where the Scrum Team demonstrates the completed work to stakeholders and gathers feedback.

Sprint Retrospective: A meeting at the end of each sprint where the Scrum Team reflects on their process and identifies opportunities for improvement.

Scrum provides a framework for fostering collaboration, transparency, and continuous improvement within teams, enabling them to deliver value to customers more effectively.

In summary, Extreme Programming (XP) emphasizes technical practices such as pair programming, test-driven development, and continuous integration, while Scrum provides a

framework for project management and collaboration through roles, artifacts, and ceremonies. Both methodologies promote agility, flexibility, and responsiveness to change, but they differ in their focus and approach to achieving those goals.

Unit– 2: Software Requirement Analysis and Design

Requirement Gathering and Analysis:

Requirement gathering and analysis is a critical phase in the software development lifecycle where the needs and expectations of stakeholders are identified, documented, and analyzed to establish the foundation for designing and building the software system. This phase is essential for ensuring that the software solution meets the desired objectives, functionalities, and constraints. Here's a breakdown of the process:

Identifying Stakeholders: The first step in requirement gathering is identifying all stakeholders who have an interest or role in the software project. Stakeholders may include end-users, customers, business owners, managers, subject matter experts, and technical teams.

Eliciting Requirements: Requirement elicitation involves actively engaging with stakeholders to understand their needs, preferences, and expectations regarding the software system. Techniques such as interviews, surveys, workshops, focus groups, and observation are used to gather information effectively.

Analyzing Requirements: Once requirements are gathered, they need to be analyzed to ensure clarity, consistency, completeness, and feasibility. This involves breaking down complex requirements into smaller, manageable units, identifying dependencies and conflicts, and prioritizing requirements based on their importance and impact on the system.

Documenting Requirements: Requirements need to be documented in a clear, concise, and unambiguous manner to serve as a basis for communication and decision-making throughout the project. Requirements documentation typically includes functional requirements (what the system should do) and non-functional requirements (qualities or constraints the system should meet), along with any assumptions, constraints, or acceptance criteria.

Validating Requirements: Requirement validation involves verifying that the documented requirements accurately reflect the needs and expectations of stakeholders and that they are feasible and achievable within the project constraints. Techniques such as reviews, walkthroughs, prototyping, and simulations may be used to validate requirements and ensure their alignment with the project goals.

Managing Requirements: Requirements management involves tracking, prioritizing, and controlling changes to requirements throughout the project lifecycle. A requirements management plan is established to document how requirements will be managed, traced, and communicated, and a change control process is implemented to evaluate and approve proposed changes to requirements.

Requirement gathering and analysis is an iterative and collaborative process that requires effective communication, active listening, empathy, and critical thinking skills. It lays the groundwork for successful software development by establishing a shared understanding of

the project scope, objectives, and deliverables among stakeholders and project team members.

Software Requirement Specification (SRS) :

The Software Requirement Specification (SRS) is a comprehensive document that serves as a blueprint for the development team, outlining the detailed requirements and specifications of the software system to be developed. Some of the characteristics of a well-written SRS document include:

Characteristic:

- Clarity: The document should be clear and easy to understand for all stakeholders, including developers, testers, project managers, and customers.
- Completeness: It should capture all relevant requirements, functionalities, constraints, and assumptions related to the software system.
- Consistency: The requirements should be consistent with each other and align with the overall goals and objectives of the project.
- Correctness: The requirements should accurately reflect the needs and expectations of stakeholders and be free from errors or ambiguities.
- Traceability: Requirements should be traceable to their sources, such as customer needs, business goals, or regulatory requirements, to ensure accountability and compliance.
- Modifiability: The document should be flexible and adaptable to changes in requirements or project scope over time.
- Verifiability: Requirements should be testable and verifiable, allowing for validation and verification throughout the development process.

Functional Requirement:

Functional requirements specify the specific behaviors, capabilities, and interactions of the software system, describing what the system should do to fulfill customer requirements. These requirements are typically expressed as detailed, precise statements of desired system functionality and can be categorized into various functional areas. Examples of functional requirements include:

"The system shall allow users to log in with a username and password."

"Users shall be able to add items to their shopping cart and proceed to checkout."

"The system shall send email notifications to users when their orders are shipped."

Functional requirements are essential for guiding the design, implementation, and testing of the software system, ensuring that it meets the intended functionality and user needs. They are typically documented using techniques such as use cases, user stories, and functional specifications within the SRS document.

Customer Requirement:

Customer requirements, also known as user requirements or business requirements, represent the needs, expectations, and objectives of the stakeholders who will be using or benefiting from the software system. These requirements are typically expressed in non-technical language and describe the high-level functionality and features desired from the software. Examples of customer requirements include:

"The system should allow users to create and edit their profiles."

"Users should be able to search for products based on various criteria."

"The system should generate monthly reports for sales analysis."

Customer requirements serve as the foundation for developing more detailed functional and non-functional requirements and provide the context for understanding the business value and user needs driving the project.

Characteristics of good software:

Correctness: Good software design ensures that the system behaves as expected and produces the desired results under all conditions. It accurately implements the requirements specified in the software requirements specification (SRS) and is free from bugs, errors, and inconsistencies.

Reliability: Reliable software design ensures that the system operates consistently and predictably over time and in various environments. It minimizes the occurrence of failures, crashes, or unexpected behavior, providing users with a dependable and trustworthy experience.

Maintainability: Maintainable software design facilitates ease of maintenance, modification, and evolution of the software system over its lifecycle. It employs clear, modular, and well-structured code, documentation, and design patterns, allowing developers to understand, update, and extend the system efficiently.

Scalability: Scalable software design enables the system to handle increasing workload, user base, and data volume without significant degradation in performance or functionality. It employs techniques such as modularization, abstraction, and optimization to support growth and accommodate changing requirements and usage patterns.

Performance: Good software design optimizes system performance to ensure that it meets or exceeds specified performance goals, such as response time, throughput, and resource utilization. It minimizes bottlenecks, inefficiencies, and unnecessary overhead, employing efficient algorithms, data structures, and optimization techniques.

Usability: Usable software design focuses on providing an intuitive, user-friendly interface and experience that meets the needs, preferences, and expectations of end-users. It employs principles of user-centered design, accessibility, and human-computer interaction to ensure that the system is easy to learn, use, and navigate.

Security: Secure software design incorporates mechanisms and safeguards to protect the system and its data from unauthorized access, manipulation, or disclosure. It employs encryption, authentication, authorization, and other security measures to mitigate risks and vulnerabilities and ensure the confidentiality, integrity, and availability of information.

Flexibility: Flexible software design enables the system to adapt to changing requirements, environments, and technologies without significant rework or disruption. It employs abstraction, encapsulation, and design patterns to promote loose coupling, separation of concerns, and modularity, allowing for easy configuration, extension, and customization.

Portability: Portable software design ensures that the system can run on different platforms, operating systems, and hardware configurations with minimal adaptation or modification. It employs platform-independent technologies, standards, and abstraction layers to maximize compatibility and interoperability across diverse environments.

Efficiency: Efficient software design optimizes resource usage, such as memory, CPU, disk space, and network bandwidth, to minimize waste and maximize performance. It employs algorithms, data structures, and caching strategies to achieve optimal runtime and resource efficiency without sacrificing functionality or usability.

Analysis v/s design:

Aspect	Analysis	Design
Purpose	Understand and define the problem domain, requirements, and constraints of the software system.	Translate requirements into a blueprint or architecture for the software solution.
Abstraction Level	Higher level of abstraction, focusing on the "what" rather than the "how" of the system.	Lower level of abstraction, focusing on the "how" of the system's implementation.
Output	Software Requirements Specification (SRS) document or similar artifacts, describing functional and non-functional requirements.	Software Design Specification (SDS) document or similar artifacts, detailing technical architecture, components, and implementation plans.
Scope	Define the scope and boundaries of the software system, identifying what needs to be built and why it is needed.	Elaborate on the details and specifics of how the software system will be built.
Activities	Requirements elicitation, analysis, validation, and prioritization.	Architectural design, detailed design, database design, user interface design, and algorithm design.
Tools and Techniques	Interviews, surveys, use case modeling, user stories, requirements workshops, and prototyping.	Architectural diagrams (e.g., UML), data flow diagrams, entity-relationship diagrams, design patterns, and modeling languages.
Focus	Understand and capture user needs, expectations, and constraints to ensure the system addresses the int ed problem effectively.	Create a technical blueprint or plan for implementing the system, ensuring it is scalable, maintainable, and meets quality attributes.

Cohesion:

Cohesion refers to the degree of relatedness and focus within a module or component of a software system. It measures how well the elements within a module are logically connected

and work together to achieve a single, well-defined purpose or responsibility. High cohesion is desirable as it leads to better maintainability, reusability, and understandability of the codebase.

Classification of Cohesion:

Functional Cohesion:

Functional cohesion occurs when elements within a module perform a single, well-defined task or function, without significant variation in functionality.

Example: A module that calculates the square root of a number or sorts an array exhibits functional cohesion.

Sequential Cohesion:

Sequential cohesion occurs when elements within a module are organized in a sequential order, with the output of one element serving as the input to the next.

Example: A module that reads data from a file, processes it, and then writes the results to another file demonstrates sequential cohesion.

Communicational Cohesion:

Communicational cohesion occurs when elements within a module operate on the same input data or share common state information.

Example: A module that formats and displays customer information received from a database exhibits communicational cohesion.

Procedural Cohesion:

Procedural cohesion occurs when elements within a module are grouped together based on their execution sequence or procedural flow.

Example: A module that handles user authentication, authorization, and session management demonstrates procedural cohesion.

Temporal Cohesion:

Temporal cohesion occurs when elements within a module are grouped together because they are executed at the same time, such as during a specific event or timeframe.

Example: A module that performs initialization tasks when the system starts up exhibits temporal cohesion.

Coupling:

Coupling refers to the degree of interdependence and connectivity between modules or components of a software system. It measures how closely two or more modules are interconnected or rely on each other to accomplish their tasks. Low coupling is desirable as it promotes modularity, flexibility, and maintainability of the codebase.

Classification of Coupling:

Data Coupling:

Data coupling occurs when modules communicate with each other through parameters or data structures, passing only the necessary data needed for interaction.

Example: A function that accepts input parameters and returns a result without accessing global variables exhibits data coupling.

Stamp Coupling:

Stamp coupling occurs when modules communicate with each other through complex data structures, such as arrays or records, passing large amounts of data.

Example: A function that receives a large data structure containing multiple fields and selectively uses certain fields exhibits stamp coupling.

Control Coupling:

Control coupling occurs when modules communicate with each other by sharing control information, such as flags or status variables, to coordinate their behavior.

Example: A function that uses a global variable to indicate whether a certain condition has been met before executing a specific code path exhibits control coupling.

Common Coupling:

Common coupling occurs when modules share a common global data or resource, such as a global variable or file, which is accessed by multiple modules.

Example: Multiple modules accessing and modifying the same global configuration file or database connection exhibit common coupling.

Content Coupling:

Content coupling occurs when modules are highly dependent on each other's internal implementation details, such as accessing each other's variables or functions directly.

Example: A function that directly accesses and modifies the internal variables of another module exhibits content coupling.

Understanding and managing cohesion and coupling are crucial for designing modular, maintainable, and scalable software systems. High cohesion and low coupling are key principles in achieving robust and flexible software architectures.

Function Oriented Software Design:

Function-Oriented Software Design (FOSD) is an approach to software design that focuses on organizing the system's functionality into a set of cohesive and reusable functions or procedures. In FOSD, the emphasis is on breaking down the problem into smaller, manageable tasks or functions and then designing the software system based on these functional components.

Here are the key principles and characteristics of Function-Oriented Software Design:

Decomposition: FOSD involves decomposing the system's requirements into smaller, more manageable functions or procedures. This decomposition process helps in breaking down the complexity of the system into smaller, more understandable components.

Modularity: Functions in FOSD are designed to be modular, meaning that they are self-contained units with well-defined inputs, outputs, and functionality. Each function performs a specific task or operation, and functions can be combined or reused to build larger, more complex systems.

Reuse: FOSD promotes the reuse of functions across different parts of the system or even across different projects. By designing functions to be modular and independent, they can be easily reused in various contexts, leading to increased productivity and efficiency.

Top-Down Design: FOSD typically follows a top-down design approach, where the system's functionality is progressively decomposed into smaller, more detailed functions. This approach allows designers to focus on high-level abstractions initially and then refine them into more concrete functions as the design process progresses.

Structured Programming: FOSD is closely associated with structured programming techniques, such as using control structures like loops, conditionals, and subroutines to organize and control the flow of program execution. Structured programming helps in creating clear, readable, and maintainable code.

Data Independence: In FOSD, functions are designed to be independent of specific data structures or implementation details. This data independence allows functions to be reused with different data types or structures, promoting flexibility and adaptability.

Encapsulation: Functions in FOSD are encapsulated units of functionality that hide their internal details and only expose well-defined interfaces to the outside world. Encapsulation helps in managing complexity, reducing dependencies, and promoting modular design.

Testing and Debugging: FOSD facilitates testing and debugging by isolating individual functions and making it easier to verify their correctness and behavior independently. This modular approach to design makes it easier to identify and fix bugs, as changes to one function are less likely to affect others.

Data Flow Diagram(DFD):

A Data Flow Diagram (DFD) is a graphical representation of the flow of data within a system. It visualizes how data moves between processes, data stores, and external entities in a system, providing a high-level overview of the system's data processing and interactions. DFDs are commonly used in software engineering and system analysis to model and understand the information flow in a system.

Here are the key components and symbols used in a Data Flow Diagram:

Processes: Processes represent the various activities or transformations that occur within the system. Each process in a DFD takes input data, performs some processing or manipulation, and produces output data. Processes are typically represented by circles or ovals in a DFD.

Data Flows: Data flows represent the movement of data between processes, data stores, and external entities in the system. They show how data is input, processed, and output by different components of the system. Data flows are typically represented by arrows in a DFD, indicating the direction of data movement.

Data Stores: Data stores represent the repositories or storage locations where data is stored within the system. They can include databases, files, tables, or any other storage medium used to hold data. Data stores are typically represented by rectangles in a DFD.

External Entities: External entities represent the external sources or destinations of data that interact with the system. They can include users, other systems, devices, or organizations that send or receive data to or from the system. External entities are typically represented by squares or rectangles with rounded corners in a DFD.

Data Flows: Data flows represent the movement of data between processes, data stores, and external entities in the system. They show how data is input, processed, and output by different components of the system. Data flows are typically represented by arrows in a DFD, indicating the direction of data movement.

Data Dictionary: A Data Dictionary is a supplementary document that accompanies a DFD, providing detailed descriptions of the data elements, data structures, data flows, and data stores depicted in the diagram. It defines the meaning, format, and characteristics of each data item used in the system.

DFDs are hierarchical in nature, meaning that they can be decomposed into multiple levels of detail. The highest level DFD, known as the Context Diagram, provides an overview of the entire system and its interactions with external entities. Lower-level DFDs provide more detailed views of specific subsystems or processes within the system, breaking down complex processes into smaller, more manageable components.

Overall, Data Flow Diagrams are powerful tools for modeling and analyzing the flow of data within a system, helping stakeholders understand how information is processed, stored, and exchanged to support the system's functionality.

Unit– 3: Software Project Estimation & Scheduling

Responsibility of software project Manager:

The role of a Software Project Manager is multifaceted and critical to the success of a software development project. Here are some of the key responsibilities:

Project Planning:

Developing project plans that outline the scope, objectives, timelines, resources, and deliverables of the project.

Creating schedules, milestones, and work breakdown structures to guide project execution.

Estimating costs, budgeting, and resource allocation to ensure project feasibility and alignment with organizational goals.

Team Management:

Building and leading a high-performing project team by recruiting, selecting, and training team members.

Assigning roles and responsibilities, delegating tasks, and fostering collaboration and communication among team members.

Providing coaching, mentoring, and support to team members, and resolving conflicts or issues that arise during the project.

Stakeholder Communication:

Serving as the primary point of contact and liaison between the project team and stakeholders, including clients, sponsors, and senior management.

Communicating project status, progress, risks, and issues to stakeholders through regular updates, reports, meetings, and presentations.

Managing stakeholder expectations and addressing their concerns or feedback to ensure alignment with project goals and objectives.

Risk Management:

Identifying, assessing, and prioritizing project risks and uncertainties that could impact project outcomes.

Developing risk mitigation strategies, contingency plans, and response mechanisms to minimize the impact of risks on project success.

Monitoring and evaluating risk throughout the project lifecycle and adjusting plans and actions as necessary to address emerging threats or opportunities.

Quality Assurance:

Establishing quality standards, metrics, and processes to ensure that project deliverables meet specified requirements and expectations.

Implementing quality assurance activities such as reviews, inspections, testing, and validation to verify the integrity, functionality, and usability of the software product.

Continuously monitoring and improving project quality and performance based on feedback and lessons learned from previous projects.

Change Management:

Managing changes to project scope, requirements, schedules, and resources in a controlled and systematic manner.

Assessing the impact of changes on project objectives, timelines, and budgets, and obtaining approval from stakeholders before implementing changes.

Communicating changes effectively to the project team and stakeholders and ensuring that everyone is aware of their implications.

Documentation and Reporting:

Maintaining accurate and up-to-date project documentation, including plans, schedules, budgets, requirements, specifications, and reports.

Generating regular progress reports, status updates, and performance metrics to track project progress, identify issues, and make data-driven decisions.

Archiving project documentation and lessons learned for future reference and knowledge sharing.

Overall, a Software Project Manager plays a pivotal role in planning, organizing, leading, and controlling software development projects to ensure they are completed on time, within budget, and to the satisfaction of stakeholders. They must possess a combination of technical expertise, leadership skills, communication abilities, and project management knowledge to effectively navigate the complexities of software development and deliver successful outcomes.

Project Estimation Techniques using COCOMO model:

Project estimation techniques using the COCOMO (Constructive Cost Model) model involve predicting the effort, time, and resources required to complete a software project based on various factors such as project size, complexity, and development environment. COCOMO was developed by Barry Boehm in the late 1970s and has since been refined into several versions, with COCOMO II being the most widely used.

Here's an overview of the COCOMO model and its estimation techniques:

Basic COCOMO: Basic COCOMO is the original version of the model and is used for estimating the effort required for small to medium-sized projects. It is based on the size of the software product in terms of lines of code (LOC) and uses a simple equation to calculate the effort in person-months:

$$Effort = a \times (KLOC)^b$$

where:

Effort is the effort required in person-months.

KLOC is the size of the software product in thousands of lines of code.

a and b are constants that depend on the project type and development environment.

Intermediate COCOMO: Intermediate COCOMO extends the basic model by considering additional factors such as product attributes, hardware constraints, personnel experience, and project complexity. It uses a set of cost drivers to adjust the effort estimate based on these factors. The effort estimation equation is:

$$Effort = a \times (KLOC)^b \times EAF$$

where EAF is the Effort Adjustment Factor calculated based on the selected cost drivers.

Detailed COCOMO: Detailed COCOMO is the most comprehensive version of the model and is used for estimating effort, schedule, and cost for large and complex projects. It breaks down the development process into multiple phases (e.g., requirements analysis, design, implementation, testing) and estimates effort and resources required for each phase separately. It also considers factors such as personnel capability, development flexibility, and project constraints.

Estimation Techniques: Estimating effort and resources using the COCOMO model typically involves the following steps:

Size Estimation: Determine the size of the software product in terms of lines of code (LOC) or function points (FP).

Selecting Cost Drivers: Identify and assess the various cost drivers such as product attributes, hardware constraints, personnel experience, and project complexity.

Calculating Effort: Use the appropriate COCOMO equation (Basic, Intermediate, or Detailed) to calculate the effort required based on the size and cost drivers.

Adjusting Estimates: Adjust the effort estimate based on additional factors such as project risk, management strategy, and organizational culture.

Iterative Refinement: Continuously refine and update the estimates as more information becomes available throughout the project lifecycle.

COCOMO provides a systematic and structured approach to project estimation, helping project managers and stakeholders make informed decisions about resource allocation, scheduling, and budgeting. However, it's essential to remember that COCOMO estimates are only predictions and may vary based on actual project conditions and uncertainties. Regular monitoring and adjustment of estimates are necessary to ensure project success.

Risk Management:

Risk management is a systematic process of identifying, assessing, and controlling risks to minimize their potential negative impact on a project or organization. It involves proactive planning, monitoring, and mitigation strategies to address potential threats and exploit opportunities. Here's an explanation of each phase of risk management:

Risk Identification:

Risk identification is the process of systematically identifying, documenting, and analyzing potential risks that could affect the project's objectives. It involves identifying both threats (negative risks) and opportunities (positive risks) that could impact the project's scope, schedule, cost, quality, or other key parameters. Risk identification techniques may include:

Brainstorming sessions with project stakeholders to generate a list of potential risks.

Reviewing project documentation, historical data, and lessons learned from past projects.

Using checklists, risk registers, and risk databases to systematically identify common project risks.

Conducting interviews, surveys, and expert consultations to gather insights and perspectives from subject matter experts.

The output of risk identification is a comprehensive list of identified risks, along with their descriptions, potential impacts, and likelihood of occurrence.

Risk Assessment:

Risk assessment involves evaluating and analyzing the identified risks to prioritize them based on their potential impact and likelihood of occurrence. It helps in understanding the significance of each risk and determining which risks require further attention and resources for mitigation. Risk assessment typically includes two key dimensions:

Impact: Assessing the potential consequences or effects of each risk on the project objectives, such as schedule delays, cost overruns, quality issues, or reputational damage.

Likelihood: Assessing the probability or likelihood of each risk occurring based on historical data, expert judgment, and other relevant information.

Risk assessment techniques may include qualitative analysis (e.g., risk matrix, risk scoring) and quantitative analysis (e.g., Monte Carlo simulation, sensitivity analysis) to assign risk ratings and prioritize risks for further action.

Risk Control:

Risk control involves developing and implementing strategies to mitigate, transfer, avoid, or accept identified risks to reduce their impact and likelihood of occurrence. It aims to manage risks effectively and ensure that the project remains on track to achieve its objectives. Risk control activities may include:

Risk Mitigation: Developing and implementing proactive measures to reduce the likelihood or impact of identified risks, such as implementing quality control processes, adding contingency reserves, or optimizing project schedules.

Risk Transfer: Transferring the responsibility for managing certain risks to third parties, such as insurance companies or subcontractors, through contractual agreements or insurance policies.

Risk Avoidance: Taking actions to eliminate or avoid the occurrence of high-risk activities or situations that could negatively impact the project, such as avoiding complex technologies or changing project scope.

Risk Acceptance: Acknowledging and accepting certain risks that cannot be mitigated or transferred and developing contingency plans to address them if they occur.

Risk control also involves monitoring and reviewing the effectiveness of risk management strategies over time, updating risk registers, and communicating risk-related information to stakeholders regularly.

By effectively managing risks throughout the project lifecycle, organizations can minimize potential threats, capitalize on opportunities, and enhance their ability to achieve project success. Risk management is an ongoing process that requires continuous attention and adaptation to changing project conditions and environments.

Metrics for Size Estimation:

Metrics for size estimation in software development are essential for quantifying the size or scale of a software project. Two common metrics used for size estimation are Lines of Code (LOC) and Function Points (FP).

Lines of Code (LOC):

Definition: Lines of Code (LOC) is a simple metric that measures the size of a software project based on the number of lines of code written in the source code files.

Usage: LOC is commonly used in traditional software development methodologies where code is the primary deliverable. It provides a straightforward measure of the size of the software system.

Advantages:

- Simple and easy to understand.
- Widely applicable to various programming languages and technologies.
- Provides a direct correlation between effort and project size.

Disadvantages:

- Not language-independent; the same functionality can be implemented in different languages with different LOC counts.

- Does not account for differences in complexity or functionality between lines of code.
- Prone to manipulation and inaccuracies, as developers may use coding shortcuts or redundant code to inflate LOC counts.

Function Points (FP):

Definition: Function Points (FP) is a standardized metric that quantifies the size of a software project based on its functional requirements and complexity.

Usage: FP is commonly used in modern software development methodologies such as agile and iterative approaches. It focuses on the functionality delivered by the software rather than the implementation details.

Advantages:

- Language-independent; it provides a consistent measure of software size across different programming languages and technologies.
- Reflects the complexity and functionality of the software system, not just the amount of code written.
- Helps in estimating effort, cost, and schedule based on the functional requirements of the software.

Disadvantages:

- Requires specialized knowledge and training to perform FP analysis accurately.
- Subjective; different analysts may assign different weights to the same function types, leading to variability in estimates.
- Time-consuming; FP analysis can be labor-intensive, especially for large and complex systems.

In summary, both Lines of Code (LOC) and Function Points (FP) are metrics used for size estimation in software development, but they differ in their focus and approach. LOC measures the size of the software based on the lines of code written, while FP measures the size based on the functional requirements and complexity of the software system. Each metric has its advantages and disadvantages, and the choice between them depends on the specific needs and context of the software project.