|  |  |
| --- | --- |
| software engineering & challenge facing SE | Software Engineering is the systematic approach to designing, building, and maintaining software systems. It involves using engineering principles to create reliable and efficient software.  CHALLENGES FACING SOFTWARE ENGINEERING  1. The legacy challenge: The majority of software systems that are in use today were developed many years yet they perform critical business functions. The legacy challenge is the challenge of maintaining and updating this software in such a way that excessive costs are avoided and essential business services continue to be delivered.  2. Heterogeneity challenge:Increasingly, systems are required to operate as distributed systems across networks that include different types of computers and with different kinds of support systems. The heterogeneity challenge is the challenge of developing techniques to build dependable software which is flexible enough to cope with this heterogeneity.  3. The delivery challenge: Many traditional software engineering techniques are time-consuming. The time they take is required to achieve software quality. However, businesses today must be responsive and change very rapidly. Their supporting software must change equally rapidly. The delivery challenge is the challenge of shortening delivery times for large and complex systems without compromising system quality.  4. Trust challenge: As software is intertwined with all aspects of our lives, it is essential that we can trust that software, so, the trust challenge is to develop techniques that demonstrate that software can be trusted by its users.  5. Risk challenge: In safety-critical areas such as space, aviation, nuclear power plants, etc. the cost of software failure can be massive because lives are at risk. Dealing with the increased complexity of software needed for new applications. |
| Software Specifications / SRS | Software specifications serves as a blueprint for the software development process and provide a clear understanding of what the software should do and how it should perform like functionality, behavior etc. Here are some key aspects of software specifications:  1. Functional Requirements: These specify what the software is supposed to do. They describe the system's functions, features, and capabilities in detail. Functional requirements often include use cases, scenarios, and user stories that outline the interactions between users and the software.  2. Non-functional Requirements: These specify how the software should perform rather than what it should do. Non-functional requirements encompass aspects like performance, reliability, scalability, security, and usability. Examples include response times, system availability, and data encryption requirements.  3. User Interfaces: Software specifications may include details about the user interfaces, such as screen mockups, wireframes, or descriptions of the user interface elements and their behavior.  4. Data Requirements: Information about the data the software will handle, including data models, database schemas, data storage, and data validation rules, is often part of the specifications.  5. Constraints: Any constraints that affect the development or use of the software should be documented. This may include hardware limitations, regulatory compliance requirements, or third-party software dependencies.  6. Use Cases: Use cases describe various interactions between users (or external systems) and the software. They outline specific scenarios and how the system should respond to different inputs and actions.  7. System Architecture: High-level descriptions of the software's architecture, including components, modules, and their interactions, can be included in the specifications. |
| Software validation | Software validation is a critical quality assurance step in the software development life cycle. It helps ensure that the software meets the intended purpose, functions correctly, and delivers value to users while also complying with any regulatory or industry-specific requirements. Validation is typically performed in conjunction with verification, which focuses on ensuring that the software is built correctly, according to its specifications. Together, verification and validation contribute to overall software quality. Here's a concise summary of software validation with three testing stages:  1. Unit Testing:     - Tests individual code components.     - Focuses on functionality within components.     - Detects and fixes defects early.  2. System Testing:     - Validates the entire system.     - Tests interactions between components.     - Ensures system functionality and performance.     - Includes various testing types.  3. Acceptance Testing:     - Validates against user or business requirements.     - Conducted by end-users or stakeholders.     - Determines if the software meets real-world needs.     - Final approval for software release. |
| Software Evolutions | Software evolution is an inevitable and essential aspect of the software development lifecycle. As software systems are deployed and used, they undergo a continuous process of change and improvement to stay aligned with evolving requirements, technological advancements, and user expectations. This evolution encompasses various aspects, including accommodating changing user needs, responding to environmental shifts, addressing errors and security vulnerabilities, adapting to new hardware configurations, and enhancing system functionality. Through these adaptations and refinements, software evolves to remain effective, secure, and competitive in the dynamic landscape of technology and user demands, ensuring its continued relevance and utility. Software evolution involves several factors, including:  1. Requirement Changes:     - Evolving user needs and market demands can lead to changes in software requirements.     - Adaptation to new features or functionality is essential to remain relevant.  2. Environment Changes:     - Changes in the deployment environment, such as operating system updates or cloud migrations, may necessitate software modifications.     - Compatibility with new hardware or software platforms may be required.  3. Errors or Security Breaches:     - The discovery of defects, vulnerabilities, or security breaches necessitates corrective actions.     - Patches and updates are crucial to fix issues and maintain system integrity.  4. New Equipment Added or Removed:     - Integration with new hardware or the removal of obsolete equipment may require software adjustments.     - Compatibility with different devices or configurations must be ensured.  5. Improvement to the System:     - Software evolution often includes enhancements for performance, usability, and user experience.     - Continuous improvement ensures the software remains competitive and efficient. |
| What is CASE? Explain the importance of CASE tools in software development life cycle? | CASE stands for Computer-Aided Software Engineering. It refers to a set of tools and software designed to assist in various activities throughout the software development life cycle. These tools are used to automate and streamline tasks, improve collaboration, and enhance the overall efficiency and quality of software development. The importance of CASE tools in the software development life cycle can be explained as follows:  1. Requirements Analysis: CASE tools can help in capturing, documenting, and managing requirements effectively. They enable developers and stakeholders to create and maintain clear and structured requirements, reducing the chances of misunderstandings and scope changes later in the project.  2. Design and Modeling: CASE tools provide graphical interfaces and templates for creating system architectures, data models, and process flows. This visual representation makes it easier for developers to design and communicate their ideas, leading to better-designed software systems.  3. Code Generation: Some CASE tools offer code generation capabilities, which can significantly speed up the coding process. Developers can generate code based on the design and modeling diagrams, reducing the chances of coding errors and inconsistencies.  4. Testing and Debugging: CASE tools often include debugging and testing features, helping developers identify and fix issues more efficiently. Automated testing tools can run test cases and report on the results, ensuring software quality.  5. Documentation: Comprehensive documentation is crucial for maintaining and evolving software systems. CASE tools assist in generating documentation automatically from the design and code, saving time and ensuring that documentation is always up-to-date.  6. Version Control and Configuration Management: Many CASE tools offer version control and configuration management features. These tools help manage changes to the software, track revisions, and ensure that developers are working on the correct version of the code and documentation.  7. Collaboration: CASE tools facilitate collaboration among team members, even if they are geographically dispersed. Team members can access and work on the same project simultaneously, ensuring that everyone is on the same page.  8. Project Management: Some CASE tools include project management features like task tracking, scheduling, and resource allocation. These features help project managers keep projects on schedule and within budget.  9. Quality Assurance: CASE tools can enforce coding standards and best practices, helping maintain code quality and consistency across the project. This can reduce the likelihood of defects and make it easier to maintain and update the software.  10. Maintenance and Evolution: After the initial development, software requires ongoing maintenance and updates. CASE tools can assist in tracking changes, managing issues, and ensuring that the software remains robust and reliable over time.  In summary, CASE tools play a crucial role in improving the efficiency, quality, and collaboration within the software development life cycle. They help streamline various tasks, reduce errors, and ensure that software projects are delivered on time and within budget. By automating and enhancing many aspects of the development process, CASE tools contribute to the success of software development projects. |
| Software Development Models | Software development process models are frameworks that guide the software development process. They provide a structured approach to managing tasks, activities, and resources throughout the software development lifecycle. Different models suit various project types and goals. Here are some notable software development process models:  1. Waterfall Model:     - Sequential and linear approach.     - Phases: Requirements, Design, Implementation, Testing, Deployment, Maintenance.     - Progression to the next phase only after completing the previous one.     - Well-suited for projects with well-defined requirements and stable scope.  2. Agile Model:     - Iterative and incremental development.     - Emphasis on flexibility, customer collaboration, and responding to changes.     - Key methods include Scrum, Kanban, and Extreme Programming (XP).     - Ideal for dynamic projects with evolving requirements.  3. Iterative Model:     - Repeatedly cycles through phases with each iteration.     - Each iteration results in an improved version of the software.     - Useful for projects where requirements are not fully understood initially.  5. Software prototyping:   * Purpose: Software prototyping creates preliminary versions to visualize, test, and refine software concepts and designs. * User Involvement: It involves users and stakeholders early, gathering feedback for better alignment with their needs. * Iterative: Prototyping is an iterative process, with multiple versions refined based on feedback. * Rapid Development: Tools and methods used in prototyping facilitate quick user interface and functionality development. * Early Validation: Prototyping helps validate requirements, reducing misunderstandings and enhancing final product quality.   6. Rational Unified Process (RUP):   * RUP promotes an iterative and incremental approach to software development, dividing the project into manageable cycles to allow for flexibility and early delivery. * Phases and Workflows: It organizes the development process into distinct phases, with each phase having defined activities and objectives. Workflows guide how activities are carried out within each phase. * Artifact-Driven: RUP places a strong emphasis on creating and maintaining documentation artifacts throughout the development process, ensuring clarity and documentation of requirements, designs, and other aspects. * Roles and Responsibilities: RUP defines specific roles and responsibilities for team members, clarifying their roles in areas like architecture, development, testing, and project management. * Customizable and Tailorable: RUP is highly customizable, allowing organizations to adapt its processes, guidelines, and templates to suit the specific needs and constraints of their software projects.   7. Spiral Model:     - Combines iterative development with risk assessment.     - Phases include planning, risk analysis, engineering, and evaluation.     - Well-suited for complex projects with high levels of uncertainty.  8. V-Model (Verification and Validation Model):     - Emphasizes validation and verification activities at each development phase.     - Matches each development stage with a corresponding testing phase.     - Ensures alignment between development and testing processes.  9. RAD Model (Rapid Application Development):     - Focuses on rapid prototyping and quick feedback from users.     - Emphasizes user involvement and iterative development.     - Suitable for projects with short timelines and frequent changes.  10. Incremental Model:     - Breaks the project into smaller, manageable parts or increments.     - Each increment adds new functionality to the system.     - Useful for large projects with complex requirements. |
| explain how the principles of agile method accelerate the development of software application | The principles of Agile software development, as outlined in the Agile Manifesto, accelerate the development of software applications in several ways:  Customer Collaboration Over Contract Negotiation: Agile places a strong emphasis on involving customers and stakeholders throughout the development process. By maintaining open lines of communication with end-users, development teams can better understand their needs and preferences, leading to a more relevant and valuable end product.  Individuals and Interactions Over Processes and Tools: Agile values the contributions of individual team members and encourages collaboration among them. This focus on people fosters creativity, problem-solving, and efficient communication, which can lead to faster and more effective development.  Working Software Over Comprehensive Documentation: While documentation is important, Agile prioritizes delivering working software over extensive documentation. This approach allows developers to focus on building functional features and getting them into the hands of users quickly, rather than spending excessive time on documentation that may become outdated.  Responding to Change Over Following a Plan: Agile recognizes that change is inevitable in software development. Instead of rigidly adhering to a fixed plan, Agile teams are flexible and can adjust their priorities and features based on new information and changing requirements. This adaptability accelerates the development process by avoiding unnecessary delays caused by resistance to change.  Sustainable Development: Agile principles encourage a sustainable pace of work. By avoiding overloading team members with excessive workloads or unrealistic deadlines, Agile promotes long-term productivity and minimizes burnout, ultimately contributing to a more efficient development process. |
| principles of Extreme Programming | The principles of Extreme Programming include:  1. Customer Satisfaction: XP places a strong emphasis on understanding and satisfying customer needs. Close collaboration with customers is encouraged to ensure that the software meets their requirements and expectations.  2. Incremental and Iterative Development: XP advocates for short development cycles, typically lasting one to three weeks, where small increments of functionality are delivered in each iteration. This incremental approach allows for frequent feedback and the early delivery of valuable features.  3. Continuous Testing: Automated testing is a fundamental practice in XP. Developers write unit tests before coding and run automated tests continuously throughout the development process to catch defects early and ensure that changes do not introduce regressions.  4. Simple Design: XP promotes a simple and clean code design. Developers are encouraged to make the simplest thing that could possibly work and to refactor code continuously to keep it maintainable and free from unnecessary complexity.  5. Pair Programming: XP encourages pair programming, where two developers work together at the same computer. This practice fosters collaboration, knowledge sharing, and higher code quality, as code is reviewed and discussed in real-time. |
| How does extreme programming support the principles of agile development? | Extreme Programming supports the principles of agile development by aligning with the core values of the Agile Manifesto and addressing many of the challenges faced in software development:  1. Customer Collaboration: XP's emphasis on close collaboration with customers ensures that development efforts are aligned with customer needs, a key principle of agile development.  2. Iterative Development: XP's short iterations align with the agile principle of delivering working software frequently, allowing for rapid adaptation to changing requirements and continuous feedback.  3. Continuous Testing and Quality: XP's focus on automated testing and clean code supports the agile principle of delivering high-quality software.  4. Adaptability: XP's practices, such as simple design and incremental development, enable teams to respond to changes in requirements and priorities, a core aspect of agility.  5. Team Collaboration: Pair programming, collective code ownership, and on-site customer involvement foster collaboration and communication, which are essential elements of agile development. |
| What is scrum? Explain scrum roles and terminologies | Scrum is an agile framework for managing and delivering complex projects, primarily in software development but applicable to various fields. It provides a structured and iterative approach to project management and product development, emphasizing collaboration, transparency, and adaptability. Scrum is guided by a set of roles, events, and artifacts that help teams organize their work and deliver value to customers efficiently.  Here are the key roles and terminologies in Scrum:  **Scrum Roles:** The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master Scrum Teams are self-organizing and cross-functional. Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross- functional teams have all competencies needed to accomplish the work without depending on others not part of the team.  **Sprint backlog:** Sprint Backlog is the subset of Product Backlog items selected for the Sprint together with a plan for delivering the product Increment and realizing the Sprint Goal. It is a forecast by the development team about what functionality will be in the next Increment and the work needed to deliver that functionalityAs new work is required, the development team adds it to the sprint backlog. As work is performed or completed, the estimated remaining work is updated. Only the development team can change its sprint backlog during a Sprint. It is a highly visible, real-time picture of the work that the development team plans to accomplish during the Sprint, and it belongs solely to the development team.  **Increment:** The Increment is the sum of all the Product Backlog items completed during a Sprint and all previous Sprints. At the end of a Sprint, the new Increment must be "Done," which means it must be in useable condition and meet the Scrum Team's Definition of "Done" It must be in useable condition regardless of whether the Product Owner decides to actually release it  **Scrum Events:** Prescribed events are used in Scrum to create regularity and to minimize the need for meetings not defined in Scrum. Scrum uses time-boxed events, such that every event has a maximum duration. This ensures an appropriate amount of time is spent planning without allowing waste in the planning process. Other than the Sprint itself, which is a container for all other events, each event in Scrum is a formal opportunity to inspect and adapt something. These events are specifically designed to enable critical transparency and inspection. Failure to include any of these events results in reduced transparency and is a lost opportunity to inspect and adapt. |
| Explain different perspective on scrum scaling agile methods | Scaling Agile Methods  Agile methods were specially developed for the development of small and medium-sized systems where there is a small co-located development team and all the members work together in the same place. Agile methodologies are becoming increasingly popular for software development. However, the most popular agile methodology, called Scrum, is best suited for a single, five- to nine-person development team. In order to adopt agile processes for larger organizations, longer projects, and more complex environments, scaling is necessary. There are two perspectives on the scaling of agile methods  **Scaling up Perspective:** This method of scaling agile concerns using these methods for developing large software systems that cannot be developed by a small team. In this method the critical adaptation that has to be introduced is: for large system developments, it is not possible to focus only on the code of the system so, software architecture has to be designed and documents have to be produced to describe critical aspects of the system such as database schema. Cross team communication mechanism has to be designed and used where teams update each other on progress.  **Scaling Out Perspective:** In this method, a major concern is with how agile methods can be introduced across a large organization with many years of software development experience. |
| Define requirement validations and It’s techniques | Requirement validation is a critical phase in the software development process that involves the systematic assessment and confirmation that gathered requirements are accurate, complete, and aligned with stakeholder expectations, ensuring that they serve as a solid foundation for subsequent project phases.  Types of Requirement Validation Techniques:  1. Requirements Reviews/Inspections: Collaborative reviews of requirement documents to uncover issues, such as ambiguities and inconsistencies.  2. Prototyping: Creation of working prototypes to validate requirements and gather early feedback.  3. User Acceptance Testing (UAT): End-users or client representatives conduct tests to validate that the system meets their needs and business goals.  4. Use Case Validation: Review and validation of use cases with stakeholders to ensure they accurately represent system interactions.  5. Requirement Workshops: Facilitated sessions to discuss, clarify, and validate requirements among project stakeholders.  6. Simulation and Modeling: Use of visual models and simulations to illustrate system behavior and validate requirements.  7. Requirement Traceability Matrix (RTM): Establishes links between requirements, their sources, and design and test artifacts.  8. Formal Verification and Validation: Rigorous mathematical or logical methods to formally verify the correctness of requirements.  9. Requirements Management Tools: Specialized software tools for managing, tracking, and validating requirements throughout the project lifecycle. |
| Difference between Functional and Non-functional requirements | **Functional Requirements || Non-Functional Requirements**  A functional requirements defines a system or its component|| A non-functional requirement defines the quality of a software system.  It specifies “What the system does (features/functions)”  || It places constraints on “How well the system performs (quality attributes)”.  It is captured in use case || It is captured as a quality attribute.  These requirements are typically testable through functional testing like End to End, API testing, etc. || These requirements may require specialized testing like performance, security, usability, etc.  These requirements directly impacts user interactions || These requirements indirectly impacts user satisfaction.  It helps you to verify the functionality of the software || It helps you to verify the performance of the software.  Examples || User registration, data validation, report generation || Response time, scalability, security standards. |
| Domain Requirements | Domain requirements, specific to particular industries or fields, capture the unique needs, rules, and regulations relevant to that domain. They ensure compliance with industry-specific standards and guide the software in supporting specialized business processes and terminology. Domain requirements encompass data handling, security, performance, and user experience expectations unique to the industry. Additionally, they may specify interoperability with other systems, scalability considerations, and documentation/reporting requirements. Collaboration with domain experts is essential for accurately capturing and validating these requirements, which may evolve as industry regulations change. |
| User requirements | User requirements are the core expectations and needs of software users and stakeholders. They define what the software should do, both in terms of features and how it should function. User requirements are essential for guiding software development and ensuring that the final product aligns with user needs and business objectives. |
| Software Maintenance and its types | Software maintenance refers to the process of managing and updating software after its initial development and deployment. It includes making modifications, fixing defects, and enhancing the software to ensure it continues to meet evolving user needs and performance standards. There are four primary types of software maintenance:  1. Corrective Maintenance: Corrective maintenance focuses on identifying and fixing defects or issues in the software. This type of maintenance is essential for resolving errors, bugs, and any unexpected problems that arise in the system.  2. Adaptive Maintenance: Adaptive maintenance involves modifying the software to adapt it to changing environments, hardware, or software dependencies. It ensures that the software remains compatible and functional in evolving conditions.  3. Perfective Maintenance: Perfective maintenance aims to improve and enhance the software's performance, reliability, and usability. It involves optimizing code, adding new features, and enhancing existing functionality to meet evolving user requirements and expectations.  4. Preventive Maintenance: Preventive maintenance proactively identifies and addresses potential issues before they lead to problems or defects. It involves activities like code refactoring, security updates, and performance tuning to prevent future issues.  Effective software maintenance is crucial for the longevity and success of software systems, as it ensures that they remain reliable, secure, and aligned with user needs over time. These four types of maintenance are often applied in various combinations throughout a software system's lifecycle to ensure its continued effectiveness and value. |
| What is cmm? describe its levels and compare it with ISO 9001 | The Capability Maturity Model (CMM) is a framework used primarily in software development and engineering to assess and improve an organization's processes and capabilities. It was originally developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. CMM provides a structured approach for organizations to enhance their processes and achieve higher levels of maturity and effectiveness.  CMM consists of five maturity levels, which represent the progression of an organization's process capabilities:  Initial (Level 1): At this level, processes are ad hoc, chaotic, and unpredictable. There is little standardization, and success relies heavily on individual skills and efforts.  Managed (Level 2): In this stage, organizations start to define and document their processes. Processes become more repeatable and predictable, often with a focus on project management and process discipline.  Defined (Level 3): At this level, processes are well-defined and standardized across the organization. They are documented, consistently followed, and tailored to specific project needs.  Quantitatively Managed (Level 4): Organizations at this level use quantitative data and metrics to manage and control their processes. The emphasis is on continuous process improvement and optimization.  Optimizing (Level 5): The highest level of CMM maturity, organizations here are committed to continuous process improvement and innovation. They actively seek out best practices and employ advanced technologies to optimize processes.  The difference between cmm and ISO 9001 are:  CMM:  Focus: Process improvement and maturity, primarily in software.  Structure: Five-level maturity model.  Applicability: Originally for software but adaptable.  Certification: Does not provide certification.  ISO 9001:  Focus: Quality management across various industries.  Structure: Structured requirements organized into clauses.  Applicability: Widely applicable to diverse industries.  Certification: Offers certification for compliance with quality standards. |
| explain system modeling and its types | System modeling is the process of creating abstract representations of a system to analyze, design, understand, or communicate its various aspects. These models help stakeholders gain insights into a system's structure, behavior, and interactions. System modeling is a fundamental activity in system engineering, software development, and various other fields. Here are different types of system models, including context modeling, interaction modeling, structural modeling, and behavioral modeling:  1. Context Modeling:     - Context Diagram: A context diagram is a high-level representation of a system that shows its boundaries and interactions with external entities, often referred to as "actors." It provides an overview of what the system does without delving into internal details.  2. Interaction Modeling:     - Sequence Diagram: Sequence diagrams illustrate the interactions and sequences of messages or actions between objects or components within a system. They are particularly useful for visualizing the flow of behavior over time in response to different scenarios.     - Communication Diagram : Communication diagrams also show interactions between objects or components but focus on the relationships and associations between them. They are helpful in understanding the structural aspects of these interactions.     - Use Case Diagram: While primarily used for context modeling, use case diagrams can also capture interactions between actors and a system. They depict the various ways in which actors interact with the system to achieve specific goals or use cases.  3. Structural Modeling:     - Class Diagram: Class diagrams represent the structure of a system in terms of classes, their attributes, methods, and associations. They are commonly used in object-oriented design to define the static structure of software systems.     - Component Diagram: Component diagrams focus on the physical or logical components of a system and their dependencies. They are useful for modeling the organization of a system into modular components.     - Deployment Diagram: Deployment diagrams depict the physical deployment of software components and hardware nodes in a distributed system, such as servers, databases, and communication channels.  4. Behavioral Modeling:     - State Diagram : State diagrams capture the different states that an object or system can be in and the transitions between these states. They are used to model the behavior of systems with distinct states and state transitions.     - Activity Diagram: Activity diagrams represent the dynamic aspects of a system, showing the flow of activities or actions within a process or workflow. They are commonly used in business process modeling and workflow design.     - Behavioral Models with Formal Methods: In some cases, formal methods and notations, such as finite state machines or temporal logic, are used to model and analyze complex system behavior formally, ensuring correctness and reliability. |
| What do you mean by coding standards? why do we need to follow them? explain its purpose | Coding standards, also known as coding conventions or coding guidelines, are a set of rules, guidelines, and best practices that define how source code should be written and formatted in a software development project. These standards aim to promote consistency, readability, maintainability, and collaboration among developers. Here's why following coding standards is crucial:  1. Consistency: Coding standards ensure a consistent code style across a project.  2. Readability: They enhance code readability, making it easier to understand.  3. Error Reduction: Following standards reduces common programming errors.  4. Collaboration: They facilitate collaboration among team members.  5. Maintainability: Standards simplify code maintenance and updates.  6. Quality Assurance: They contribute to higher software quality and reliability.  7. Scalability: Coding standards help manage project growth and complexity effectively. |
| Milestones | Milestones are specific points or events in a project's timeline that mark significant achievements, stages, or deadlines. They play a crucial role in project management by serving as key reference points for assessing progress, recognizing important accomplishments, making decisions, managing risks, and facilitating communication among project teams and stakeholders. Milestones help ensure that the project stays on course, and they provide a clear framework for monitoring and controlling the project's advancement toward its goals. |
| Deliverables | Deliverables are tangible outputs, results, or products produced during the course of a project. They represent the culmination of specific tasks, phases, or efforts within the project and are typically the key items that need to be completed and delivered to meet project objectives. Deliverables can take various forms, including documents, software components, prototypes, reports, or any other tangible results that demonstrate progress and contribute to the project's goals. These tangible outcomes are not only essential for achieving project success but also serve as a means of measuring and verifying the completion of project work and fulfilling stakeholder expectations. |
| Software  verification and validation | Software Verification || Software Validation  Software verification ensures that the software meets its specified requirements and behaves correctly according to its design. || Software validation ensures that the software satisfies the needs and expectations of the end-users and the intended environment.  Software verification is typically performed during the development phase before the software is released. || Software validation is typically performed after the development phase, during testing and prior to deployment.  Verification mainly focuses on assessing the software against its documented requirements, specifications, and design. || Validation mainly focuses on assessing the software's real-world behavior, usability, and fitness for its intended purpose.  It involves methods like code reviews, inspections, static analysis, and unit testing. || It involves methods like system testing, integration testing, user acceptance testing, and usability testing.  Outputs of software verification includes verification reports, test cases, and code that conforms to specified requirements. || Outputs include validation reports, user feedback, and software that meets user needs and expectations.  Software verification gives question "Are we building the product right?" (i.e., correctness) | "Are we building the right product?" (i.e., relevance)  Main goal of verification is to ensures that the software is error-free, adheres to specifications, and functions as intended. | Ensures that the software solves the right problem and meets user needs effectively.  It is typically done by developers, testers, and quality assurance teams. | It is typically done by end-users, customers, and product owners.  | Example | Verifying that a login page correctly validates user credentials. | Validating that the login process is user-friendly and meets security requirements. |
|  |  |
|  |  |
|  |  |
| Process of testing software / testing levels / METHOD | In software testing, different levels of testing are performed to ensure the quality and correctness of a software application. These levels progressively test the software from smaller, more granular components to the entire system and its alignment with user requirements. Here's an explanation of the software testing process at various levels: unit testing, module testing, subsystem testing, system testing, and acceptance testing.  1. Unit Testing:Unit testing focuses on the smallest testable parts of the software, often at the function or method level.The primary purpose is to verify that individual units or components of code work as intended.Developers write unit tests to validate each component's correctness, ensuring that it produces the expected output for given inputs.Early detection of defects and code quality improvement.  2. Module Testing:Module testing expands the scope to test multiple related units or components that work together to perform a specific function or module.It verifies the interactions between these units and the correct functionality of the module as a whole.Test cases are designed to assess the integration and functionality of the module.Ensures that modules can work together cohesively.  3. Subsystem Testing:Subsystem testing tests a group of modules or components that collectively provide a specific subsystem or feature of the software.It validates the interactions and interfaces between subsystems and ensures that the subsystem delivers the intended functionality. Test cases assess the integration and functionality of the subsystem and may involve testing across multiple modules.Provides confidence that subsystems work together as expected.  4. System Testing:System testing examines the entire software application or system as a whole.It verifies the system's functionality, performance, and behavior in alignment with its requirements.Test cases assess functional, non-functional, and system-wide requirements, often covering various use cases and scenarios.Identifies any issues or discrepancies in the complete system.  5. Acceptance Testing:  Acceptance testing assesses whether the software meets the specified business requirements and is ready for deployment. It validates that the software fulfills user expectations and is acceptable for use in the real-world environment. End-users or stakeholders may perform user acceptance testing (UAT) to evaluate the software based on real-world scenarios.: Ensures alignment with business needs and user satisfaction, signifying readiness for deployment.  **White-Box Testing:** White-box testing is a software testing method that examines the internal code structure, logic, and data flows of a software application. Testers with knowledge of the application's source code design test cases to assess code correctness, logic flow, and adherence to coding standards, aiming to uncover defects and vulnerabilities at the code level.     - Types:    - Static White-Box Testing: Analyzing code without executing it (e.g., code reviews).    - Dynamic White-Box Testing: Examining code behavior through execution (e.g., unit testing).     - Use Cases: Identifying code-level defects and vulnerabilities.  **Black-Box Testing:** Black-box testing is a software testing approach where testers evaluate the functionality of a software application without knowledge of its internal code or structure. Instead, they focus on verifying that the software behaves correctly according to specified requirements and user expectations.     - Types:    - Functional Testing: Assessing software functionality without knowledge of its internal code.    - Non-Functional Testing: Evaluating non-functional aspects like performance, usability, and security.     - Use Cases: Validating user requirements, system behavior, and quality attributes. |
| What do you mean by risk management? Explain risk management process in software engineering. | Risk management in software engineering refers to the systematic process of identifying, assessing, prioritizing, and mitigating risks that could impact a software project's success. These risks can be anything that may lead to project failure, such as budget overruns, schedule delays, quality issues, or unexpected technical challenges. Effective risk management aims to minimize the negative impact of these risks and increase the likelihood of project success.  The software risk management process typically involves the following steps:  1. Risk Identification:In this step, project stakeholders identify and document potential risks that could affect the software project.Brainstorming, documentation review, interviews, and expert judgment are commonly used techniques to identify risks.A list of identified risks along with their descriptions and potential impacts.  2. Risk Assessment:Once risks are identified, they are assessed to determine their likelihood of occurrence and their potential impact on the project. Qualitative assessment involves assigning subjective values to the likelihood and impact (e.g., low, medium, high). Quantitative assessment uses data and modeling to calculate risk levels.A prioritized list of risks, often in the form of a risk matrix, where risks are ranked based on their severity.  3. Risk Mitigation Planning:For each identified and assessed risk, a mitigation plan is developed to address and reduce the risk's impact or likelihood.Risk mitigation strategies may include risk avoidance (eliminating the risk), risk reduction (taking actions to minimize the risk), risk sharing (transferring risk to third parties), or risk acceptance (accepting the risk with a contingency plan).Detailed mitigation plans for high-priority risks, including actions, responsibilities, and timelines.  4. Risk Monitoring and Tracking:Throughout the software project's lifecycle, risks are continually monitored to assess their status, evaluate the effectiveness of mitigation strategies, and identify new risks.Regular status updates, risk assessments, and reviews are conducted to track the progress of risk mitigation efforts. Ongoing updates to risk registers and status reports, as well as adjustments to mitigation plans as needed.  5. Contingency Planning:In cases where a risk materializes and becomes an issue, contingency plans are enacted to manage and mitigate the impact of the risk.Contingency plans may involve invoking backup procedures, allocating additional resources, or implementing alternative approaches to mitigate the risk's effects.Well-documented contingency plans that guide actions in response to specific risk events. |
|  |  |
| Software Reengineering and it’s objectives | Software Reengineering, also known as software reverse engineering, is the process of analyzing, understanding, and often restructuring or modernizing an existing software system to improve its quality, maintainability, performance, or other attributes without changing its external behavior. This is typically done when dealing with legacy software, outdated technology stacks, or poorly documented code. The primary objectives of software reengineering include:  1. Code Understanding: Analyzing and comprehending the existing source code and documentation to gain insights into the system's structure and behavior.  2. Improvement: Enhancing the software by making changes to its internal structure, which may involve refactoring code, optimizing algorithms, or redesigning components to improve performance or maintainability.  3. Migration: Adapting the software to run on new platforms, programming languages, or technology stacks while preserving its functionality and data.  4. Documentation: Updating or creating documentation to accurately reflect the current state of the software, making it easier for future maintenance and development.  5. Re-testing: Re-validating the software to ensure that changes introduced during reengineering do not introduce new defects and that the software continues to meet its requirements.  6. Legacy System Modernization: Transforming older software systems to align with modern technology standards and practices. |
|  |  |
|  |  |
| What is Quality Assurance? discuss the activities in detail | Software Quality Assurance (SQA) is a comprehensive approach to ensure the quality of software products and processes throughout their development life cycle. SQA involves a range of activities and processes aimed at preventing defects, detecting and addressing issues, and improving the overall quality of software. Here are some key SQA activities in detail:  **Quality Planning:**Establishing a plan that outlines the quality goals, standards, and processes for a specific project.Determining quality objectives, defining processes, selecting appropriate quality standards and metrics, and creating a Quality Assurance Plan (QAP).  **Process Definition and Implementation:**Defining and implementing standardized processes and procedures for software development and testing.Documenting the software development lifecycle, creating process workflows, and ensuring that processes are followed consistently across the organization.  **Quality Standards and Metrics:**Defining and enforcing quality standards, guidelines, and performance metrics.Establishing coding standards, design guidelines, and testing procedures. Defining metrics to measure code quality, defect density, and process performance.  **Reviews and Audits:**Evaluating work products and processes to identify defects and ensure compliance with quality standards.Conducting code reviews, design reviews, and process audits to find issues early in the development cycle and rectify them promptly.  **Testing and Test Planning:**: Ensuring that software functions correctly and meets requirements.Developing a comprehensive test plan, designing test cases, executing tests, and reporting and tracking defects. This includes functional, integration, performance, and regression testing.  **Documentation:**Creating and maintaining accurate and comprehensive documentation throughout the software development process.Documenting requirements, design specifications, coding standards, test plans, and user manuals.  **Training and Education:**Ensuring that the development team is knowledgeable about and follows quality processes and standards.Providing training sessions and resources to educate team members on best practices, new tools, and quality guidelines.  **Configuration Management:**Managing and controlling changes to software and its related assets.Identifying and tracking software configuration items, version control, and managing changes through a well-defined process.  **Risk Management:**Identifying and mitigating risks that could affect software quality and project success.Conducting risk assessments, developing risk mitigation plans, and monitoring risks throughout the project lifecycle.  **Continuous Improvement:**Fostering a culture of ongoing enhancement of processes and practices.Collecting and analyzing data and feedback to identify areas for improvement. Implementing lessons learned from previous projects to enhance quality in future ones.  **Quality Reporting and Metrics Analysis:** Providing stakeholders with visibility into the quality of the software and the progress of quality improvement efforts.Generating quality reports, analyzing metrics, and communicating findings to project teams and management. |
|  | Note small Q & A  3 wota  then finish 2 page a4 paper |
| Reengineering phases | 1. **Understanding and Documentation:**  * **Objective:** Gain a comprehensive understanding of the existing software system, including its functionality, architecture, codebase, and documentation (if available). * **Activities:**   + Review existing documentation, if any.   + Conduct interviews with stakeholders, including developers and users.   + Perform code analysis to understand the structure and dependencies within the software.  1. **Reverse Engineering:**    * **Objective:** Create detailed models, representations, or documentation of the existing software system's design and code.    * **Activities:**      + Reverse engineer the code to produce high-level models or diagrams (e.g., UML diagrams) that represent the software's architecture.      + Generate or update documentation, including data flow diagrams, class diagrams, and sequence diagrams.      + Identify and document software components and their relationships. 2. **Assessment and Risk Analysis:**    * **Objective:** Identify weaknesses, vulnerabilities, and areas of improvement within the existing software.    * **Activities:**      + Perform a risk assessment to identify security, performance, and maintenance risks.      + Evaluate the software against modern software engineering standards and best practices.      + Identify code smells, technical debt, and areas requiring refactoring. 3. **Re-architecting and Refactoring:**    * **Objective:** Restructure the software to improve its maintainability, scalability, and other quality attributes.    * **Activities:**      + Plan and implement architectural changes to align the software with modern design patterns and principles.      + Refactor code to improve its clarity, modularity, and readability.      + Optimize algorithms and data structures for better performance. 4. **Re-implementation or Migration:**    * **Objective:** Rewrite or migrate parts of the software to newer platforms, languages, or frameworks.    * **Activities:**      + Rewrite components using more up-to-date technologies or languages.      + Migrate data and functionality to new platforms or environments.      + Ensure compatibility with modern hardware and software requirements. 5. **Testing and Validation:**    * **Objective:** Verify that the reengineered software meets its objectives, adheres to requirements, and functions correctly.    * **Activities:**      + Design and execute test cases to validate the reengineered components.      + Conduct integration testing to ensure the new and existing parts of the software work together seamlessly.      + Perform regression testing to ensure that existing functionality remains intact. 6. **Documentation and Knowledge Transfer:**    * **Objective:** Document the reengineered software and transfer knowledge to the development team.    * **Activities:**      + Update and create documentation for the reengineered components.      + Provide training and knowledge transfer sessions to team members who will maintain the software. 7. **Deployment and Monitoring:**    * **Objective:** Deploy the reengineered software into production and monitor its performance.    * **Activities:**      + Plan and execute the deployment of reengineered components.      + Monitor the software in the production environment, addressing any issues that may arise. 8. **Maintenance and Evolution:**    * **Objective:** Continuously maintain, improve, and evolve the reengineered software.    * **Activities:**      + Establish ongoing maintenance procedures and practices.      + Address bug fixes, updates, and new feature requests as they arise. 9. **Project Closure:**    * **Objective:** Officially close the reengineering project, document lessons learned, and assess the project's success.    * **Activities:**      + Conduct a final project review and assessment.      + Document lessons learned and best practices for future projects.      + Hand over the reengineered software to the maintenance and support team. |
|  | Re engneerin |
|  |  |
|  |  |