|  |
| --- |
| Verification vs. Validation:  Why Do We Need Both?  Verification checks if software is built correctly, while validation ensures it meets users' needs. Both processes together ensure accuracy, user satisfaction, risk reduction, and confidence in the final product. |

|  |
| --- |
| ii. Dynamic Techniques:  Testing: Executing the software to evaluate its behavior against specified requirements. Types of testing include unit, integration, functional, performance, security, and more.  Analysis: Dynamically monitoring and analyzing the software's behavior during runtime to identify performance issues, memory leaks, and other runtime problems.  iii. Test-Driven Development (TDD):  Developers write tests before writing the corresponding code. TDD ensures that the code meets requirements and helps create a more reliable software architecture.  iv. Model-Driven Development:  Using models or diagrams to design software components. The model is translated into code, promoting consistency between design and implementation.  Behavior-Driven Development (BDD): Specifications are written in natural language, guiding development and ensuring alignment with user needs.  Continuous Integration (CI): Frequently integrating code changes into a shared repository, followed by automated testing, to catch integration issues early.  Continuous Delivery (CD): Automating deployment processes to ensure that the software can be deployed to production at any time.  Pair Programming: Two developers collaborate on writing code, reviewing each other's work in real-time to catch errors.  Code Reviews: Peers review code to identify bugs, readability issues, and adherence to coding standards. |

|  |
| --- |
| Software Testing:  a. Levels of Testing:  i. Unit Testing: Testing individual components or modules in isolation to ensure their correctness and functionality.  ii. System Testing:  Integration Testing: Verifying interactions between integrated components to ensure their collective behavior is as expected.  Release Testing: Validating the entire system against its requirements to prepare for deployment.  iii. Acceptance Testing: Testing to ensure that the software meets user requirements and is ready for acceptance by stakeholders.  b. Types of Testing:  i. Functional Testing: Verifying that the software functions according to specifications and user requirements.  ii. Non-Functional Testing:  Performance Testing: Assessing the software's speed, responsiveness, and scalability.  Load Testing: Evaluating software behavior under various loads to determine its capacity and performance limits.  Usability Testing: Ensuring the software is user-friendly and intuitive.  Security Testing: Identifying vulnerabilities and ensuring data protection.  c. Techniques:  i. Regression Testing: Re-running tests to ensure that new code changes have not adversely affected existing functionalities.  ii. Ad-Hoc Testing: Informal testing without a predefined test plan, simulating real-world usage.  iii. Smoke Testing: Preliminary testing to ensure that critical functionalities work before in-depth testing.  iv. ... (Other Techniques): Various other techniques, including exploratory testing, boundary testing, stress testing, and more. |

|  |
| --- |
| d. Concepts  i. Test case  ii. Test steps  iii. Test data  iv. Test results  v. Defects  vi. Test coverage  1. Code coverage  2. Function coverage  3. UI coverage  4. Path coverage |

|  |
| --- |
| Software Testing Techniques:  e. Techniques:  i. Requirement-Based Testing:  Creating test cases based on specified requirements to ensure that all requirements are met.  ii. Equivalence Partition Testing:  Dividing input ranges into equivalent classes.  Example: Generating test cases for the registration function with usernames having at least 8 characters and a special character.  iii. Path Testing:  Designing test cases to cover all possible execution paths in a code snippet.  Given code, create test cases to traverse all paths.  iv. Ad-Hoc Testing:  Testing without a predefined plan, simulating real-world usage.  v. Smoke Testing:  Preliminary testing to ensure essential functionalities work before comprehensive testing. |

|  |
| --- |
| Explain why, in the waterfall model, it is more effective to discover and fix defects in the early phases than in the later phases of the project.  In the waterfall model, it's more effective to discover and fix defects in the early phases due to lower costs, prevention of error propagation, minimal impact on documentation, risk reduction, meeting stakeholder expectations, maintaining progress, ensuring quality, and providing valuable feedback for process improvement. |

|  |
| --- |
| List all responsibilities of a business analyst in a software project. What are the main artifacts produced by this role?  Responsibilities of a Business Analyst:  Gather and analyze requirements.  Communicate with stakeholders.  Define project scope and priorities.  Facilitate user acceptance testing.  Ensure alignment with business goals.  Manage changes and risks.  Document processes and requirements.  Main Artifacts Produced:  Requirement documents.  Use cases and user stories.  Process flow diagrams.  UI mockups.  Business and functional requirement documents.  Change requests and risk assessments. |
| Why do we need to have a test plan? What information does it include?  Why Have a Test Plan?  A test plan guides thorough and organized software testing, setting objectives, defining scope, assigning roles, managing resources, and addressing risks.  Information Included:  Introduction, objectives, and scope.  Test strategy, techniques, and methodologies.  Roles, responsibilities, and schedule.  Test environment, data, and deliverables.  Risk assessment and mitigation.  Testing approaches and metrics. Defect management and exit criteria. | |

|  |
| --- |
| Software Process Models:  Software Process: It's a structured approach involving activities, tasks, and methods for software development, ensuring organized and quality outcomes.  What: Encompasses planning, design, coding, testing, deployment, and maintenance.  Who: Involves software engineers, developers, testers, managers, stakeholders, and users.  When: Begins at project initiation, spans development to maintenance.  Why: Ensures efficiency, quality, risk management, alignment with goals, and successful software products. Facilitates communication, progress tracking, resource management, and adaptation to changes. |

|  |
| --- |
| i. Waterfall model: traditional approach  1. Sequential  2. Making sure one stage completed before doing the next  3. Suitable for projects having stable requirements: it is  eliminating the waste of projects.  4. Stages: requirements definition, design & analysis,  implementation, testing, deployment, maintenance & operation.  ii. RUP: traditional approach  1. Iterative: doing things repeatedly, incrementally  2. Phases: Inception, Elaboration, Construction, and Transition  3. Iterations = cycle  a. One iteration is like one cycle of waterfall  4. Detailed process to do things  5. Detailed roles of team members  6. Many documents to be written  7. Use-case driven development |

|  |
| --- |
| iii. Agile Methods:  1. Four Agile Value Propositions:  People over Processes: Emphasizes collaboration and adaptability.  Working Software over Documentation: Focuses on functional software.  Customer Collaboration over Contracts: Involves customers throughout.  Responding to Change over Plans: Adapts to changing requirements.  2. Scrum:  Sprint: Fixed-length iteration for development.  Backlog: Product, Sprint, and Impediment backlogs.  Roles: Product Owner, Scrum Master, Development Team.  Activities: Daily Scrum, Sprint Review, Planning, Release Planning.  3. XP (Extreme Programming):  Philosophy: Intensifies good practices.  Key Practices: Pair Programming, Small Releases, Coding Standard.  Agile methods prioritize collaboration, working software, customer involvement, and adaptability, offering structures like Scrum for iterative development and XP for rigorous coding practices. |

|  |
| --- |
| Software Process Model:  A software process model is a structured framework outlining stages, tasks, and relationships in software development. It guides how software is planned, designed, built, and maintained. Examples include Waterfall, Agile, and Spiral models, offering different approaches to manage projects effectively. Models help with organization, resource allocation, and adapting to project needs. |

|  |
| --- |
| 2. Software Project Management:  a. Goals:  Delivery: Achieve objectives within scope, time, and budget.  Quality: Ensure high software standards and satisfaction.  Stakeholders: Meet needs, manage risks, optimize resources.  b. Roles:  i. Project Manager (PM):  Plans, monitors, communicates, ensures goals.  ii. Technical Architect (TA):  Designs architecture, guides tech choices.  iii. Business Analyst (BA):  Gathers requirements, bridges business and tech.  iv. Tester:  Tests, reports defects, ensures quality.  v. Maintainer:  Manages post-release, updates, and stability.  Efficient project management balances roles for successful software development. |

|  |
| --- |
| c. Activities:  i. Planning:  Estimation, scheduling, task assignment.  ii. Team Building:  Collaboration, trust, communication.  iii. Human Management:  Well-being, motivation, skill development.  iv. Controlling and Monitoring:  Reporting, problem resolving.  v. Customer Collaboration:  Feedback, requirements alignment.  vi. Risk Management:  Identification, assessment, mitigation.  Effective project management involves planning, team cohesion, stakeholder engagement, and risk mitigation for successful software development. |

|  |
| --- |
| 3. Software Requirements & Requirements Engineering:  a. Software Requirements:  i. User & System Requirements:  User Requirements: Express user needs and concepts of operation.  System Requirements: Include use cases and user stories.  ii. Functional & Non-Functional Requirements:  Functional Requirements: Describe software behavior and features.  Non-Functional Requirements: Cover performance, security, etc.  iii. Domain Requirements:  Functional: Domain-specific functionalities.  Non-Functional: Domain-related performance, legal aspects.  Requirements engineering involves understanding user needs, defining software features, and considering both functional and non-functional aspects within specific domains. |

|  |
| --- |
| b. Requirements Engineering:  i. Requirement Gathering:  Techniques: Interview, survey, observation, record.  ii. Requirement Analysis and Documentation:  Use Case Model: Visualizes interactions for understanding.  iii. Requirement Validation:  Inspection: Systematic review for errors.  TDD: Test generation before coding.  Prototyping: Proof of concept for validation.  iv. Requirement Management:  Tracks changes, ensures alignment with goals.  Requirements engineering involves gathering, analyzing, validating, and managing software requirements using diverse techniques for accurate and aligned specifications. |

|  |
| --- |
| **why do we need to do requirements validation? Why requirement inspection/review is effective?**  Requirements Validation:  Ensures accurate, user-aligned requirements to prevent costly errors and enhance success.  Requirement Inspection/Review:  Effective due to multiple perspectives, error detection, collaboration, and improved quality.  Both practices minimize misunderstandings, reduce risks, and increase the likelihood of a successful software outcome. |
| Software Analysis and Design:  a. Software Architecture:  i. Importance of Detailed Architecture (Log4j): Clear architecture guides component interactions, enhancing efficiency, communication, and future upgrades, e.g., Log4j's logging mechanisms.  ii. Scalability - Horizontal vs. Vertical Scale:  Horizontal Scale: More machines for load distribution.  Vertical Scale: Enhance machine resources.  iii. Architectural Critical Requirements: Fundamental attributes (performance, security) addressed by architecture decisions.  Architectural detailing ensures coherence, scalability options, and meets critical requirements for effective software development. |
| b. Architectural Design:  i. Requirements-Based Architecture: Converts requirements into structured architecture.  ii. Architecture's Performance Impact: Influences speed, efficiency, and responsiveness.  iii. Loosely vs. Tightly Coupled:  Loosely Coupled: Flexibility, independence.  Tightly Coupled: Performance, complexity.  iv. Fine- vs. Coarse-Grained Components:  Fine-Grained: Precision, flexibility.  Coarse-Grained: Management, simplicity. |

|  |
| --- |
| c. Detailed Design (Low-Level Design):  i. High-Level vs. Detailed Design:  High-Level Design: System structure, major components.  Detailed Design: Implementation specifics, interactions.  ii. Class Diagram: Graphical representation of classes, attributes, methods, relationships.  iii. Sequence Diagram: Visualizes object interactions, message flow.  Detailed design refines concepts into specifics using class and sequence diagrams for effective software development. |