BCU BCA 5TH SEM — SOFTWARE ENGINEERING SOLVED QUESTION BANK

Solved Question Bank by Nikhil P

PART I: SECTION A — 2 MARKS QUESTIONS (PAPERS 1 TO 5)

MODEL QUESTION PAPER – 1

Q1. What is software engineering?

IEEE Definition: According to IEEE 610.12, software engineering is the systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering principles to software.
Core Objective: To build reliable, maintainable, efficient, and cost-effective software products that satisfy client requirements within defined schedule and budget constraints.
Core Elements: Tools, Methods, Processes, and a Quality commitment.

Q2. What is Requirement Engineering?

Definition: Requirement Engineering (RE) is the systematic, iterative process of discovering, eliciting, analyzing, documenting, validating, and managing the functional services and operational constraints required of a software system.
Key Phases: Requirements Elicitation → Requirements Analysis & Negotiation → Requirements Specification (SRS) → Requirements Validation → Requirements Management.
Primary Deliverable: Software Requirements Specification (SRS) document.

Q3. What is software design?

Definition: Software design is the iterative engineering process of translating customer requirements into an architectural blueprint, data structures, component interfaces, and procedural algorithms before programming begins.
Two Major Levels:

- **High-Level (Architectural) Design:** Outlines overall system architecture, subsystems, modules, and database structures.
- **Low-Level (Detailed) Design:** Defines internal module logic, algorithmic pseudocode, data structures, and interface specifications.

Q4. What is cohesion?

Definition: Cohesion is a software design metric that measures the internal strength, functional relatedness, and degree of unity among the elements or responsibilities contained within a single software module.
Engineering Rule: High cohesion is highly desirable because it ensures each module performs a single, focused, well-defined task, enhancing readability, reusability, testability, and maintainability.
Ideal Level: Functional Cohesion (all module operations contribute strictly to one single function).

Q5. What is software testing?

Definition: Software testing is the systematic process of verifying and validating software artifacts by executing programs under controlled conditions with the explicit intent of uncovering defects, bugs, and requirement omissions.
Dual Objectives:

- **Verification:** Ensures the software is built correctly according to design specifications ("Are we building the product right?").
- **Validation:** Ensures the software satisfies actual user needs and business requirements ("Are we building the right product?").

Q6. What is white-box testing?

Definition: White-box testing (also known as structural, glass-box, or clear-box testing) is a software testing methodology where the internal logic, code paths, conditional statements, and control flows of the program are directly analyzed and tested.
Core Coverage Techniques: Statement Coverage, Branch/Decision Coverage, Condition Coverage, and Path Coverage (using McCabe's Cyclomatic Complexity).
Prerequisite: Testers must possess detailed knowledge of the underlying source code and algorithm design.

Q7. What are product metrics?

Definition: Product metrics are quantitative, objective measurements used to assess the internal and external attributes, size, complexity, reliability, and performance of software deliverables and work products across lifecycle phases.
Key Examples:

- **Size Metrics:** Lines of Code (LOC), Function Points (FP).
- **Complexity Metrics:** McCabe's Cyclomatic Complexity `V(G) = E - N + 2P`.
- **Quality Metrics:** Defect density (defects per KLOC), Mean Time Between Failures (MTBF).

Q8. What is software maintenance?

Definition: Software maintenance is the modification of a software system after delivery to correct latent faults, improve operational performance, or adapt the software to a changing business, regulatory, or computing environment.
Four Standard Types:

- **Corrective Maintenance:** Diagnosing and fixing discovered software defects and bugs.
- **Adaptive Maintenance:** Modifying software to accommodate hardware, OS, or compiler changes.
- **Perfective Maintenance:** Adding new user features or optimizing performance based on user requests.
- **Preventive Maintenance:** Re-engineering code and documentation to prevent future maintenance problems.

Q9. Define software project management.

Definition: Software Project Management (SPM) is the discipline of planning, organizing, staffing, monitoring, and controlling all software development activities to ensure that quality software is delivered on schedule, within budget, and meeting customer specifications.
Core Triple Constraints: Scope, Time, Cost, and Quality.
Key Responsibilities: Effort estimation, project scheduling, risk analysis, resource allocation, and team coordination.

Q10. What is software project planning?

Definition: Software project planning is the initial, preparatory phase of software project management in which project scope is demarcated, feasibility is evaluated, risks are assessed, and required resources, costs, and timelines are formally estimated prior to engineering execution.
Key Deliverables: Software Project Management Plan (SPMP), Work Breakdown Structure (WBS), and Gantt chart milestone schedules.
Standard Estimation Models: COCOMO (Constructive Cost Model), Function Point Analysis (FPA), and Expert Delphi judgment.

MODEL QUESTION PAPER – 2

Q1. What is software crisis?

Definition: Software crisis refers to a historical problem (originating in the late 1960s) where software projects consistently exceeded financial budgets, missed delivery deadlines, contained numerous residual bugs, and proved extremely difficult to maintain.
Key Symptoms: Cost overruns, schedule slippages, low user satisfaction, poor software quality, and complex maintenance.

Significance: The crisis served as the primary catalyst for establishing Software Engineering as a formal, disciplined engineering domain.

Q2. What is the Classical Waterfall Model?

Definition: The Classical Waterfall Model is a theoretical, linear-sequential software life cycle model where development progresses strictly in a downward cascading flow across distinct, non-overlapping phases.
Core Sequence: Feasibility Study → Requirements Analysis → Design → Coding → Testing → Maintenance.
Distinguishing Rule: Each phase must be completely finished and formally signed off before the next phase begins, with strictly no feedback loops permitted.

Q3. What is the Prototyping Model?

Definition: The Prototyping Model is an evolutionary software development approach where an initial working mock-up or prototype of the system is rapidly constructed, demonstrated to users for evaluation, and iteratively refined before full-scale coding begins.
Core Process: Quick Design → Build Prototype → Customer Evaluation → Refine Requirements → Engineer Final Product.
Best Suited For: Projects where user requirements are ambiguous, vague, or frequently evolving.

Q4. What is a module?

Definition: A module is a logically separable, independently identifiable, and self-contained programmatic unit that performs a specific, cohesive sub-task within a larger software system.
Key Characteristics: High internal cohesion, low external coupling, clean interface boundaries, and hidden internal implementation details.
Examples: Functions, subroutines, classes, or packages (e.g., `PaymentProcessor`, `AuthModule`).

Q5. What is unit testing?

Definition: Unit testing is the lowest-level verification testing phase where individual software units, functions, procedures, or methods are tested in complete isolation from the rest of the application.
Execution Role: Conducted directly by developers during the implementation phase using automated testing frameworks (e.g., JUnit, PyTest).
Support Harness: Employs testing stubs and drivers to simulate missing subroutines or calling modules.

Q6. What is software quality?

IEEE Definition: Software quality is the degree to which a software system, component, or process conforms to explicit functional specifications, documented development standards, and customer expectations.
Major Quality Models: McCall's Quality Model, Boehm's Model, and ISO/IEC 9126.
Core Quality Factors: Correctness, Reliability, Usability, Efficiency, Maintainability, Portability, and Security.

Q7. What is a Function Point?

Definition: A Function Point (FP) is a synthetic, standardized, language-independent unit of measurement used to quantify the functional size and business capability delivered by a software application to its users.
Five User Function Types: External Inputs (EI), External Outputs (EO), External Inquiries (EQ), Internal Logical Files (ILF), and External Interface Files (EIF).
Advantage: Measures software size directly from user requirements before source code is written, overcoming the language bias of Lines of Code (LOC).

Q8. What is corrective maintenance?

Definition: Corrective maintenance is reactive maintenance carried out after software delivery to identify, diagnose, patch, and repair residual faults, bugs, and defects uncovered during operational use.

Scope of Repairs: Logic bugs, algorithmic mistakes, runtime crashes, and syntax or design flaws.
Maintenance Share: Represents approximately 20% to 25% of all post-delivery software maintenance activities.

Q9. What is adaptive maintenance?

Definition: Adaptive maintenance refers to modifications made to a software system to keep it operational and compatible with a continually changing external environment without altering its core functionality.
Environmental Triggers: Hardware upgrades, operating system migrations, new database versions, or updated legal/tax regulatory mandates.
Example: Modifying payroll software to incorporate newly revised tax slabs or updating a desktop app to run on a new 64-bit operating system.

Q10. Define software cost estimation.

Definition: Software cost estimation is the analytical process of predicting the realistic financial expenditure, human effort (measured in person-months), and project schedule required to develop a software product.
Key Input Parameters: Software size (LOC or Function Points), project complexity, team experience, development tools, and delivery timeline.
Prominent Techniques: Algorithmic models (COCOMO Model), Expert Judgment (Delphi technique), and Estimation by Analogy.

MODEL QUESTION PAPER – 3

Q1. What is Software Development Life Cycle (SDLC)?

Definition: Software Development Life Cycle (SDLC) is a structured framework that outlines the systematic sequential phases through which a software product progresses from initial conception to final deployment and maintenance.
Sequential Phases: Feasibility Study → Requirements Analysis → System Design → Coding (Implementation) → Testing → Deployment → Maintenance.
Core Objective: To produce high-quality, reliable software that satisfies client expectations within estimated cost and time constraints.

Q2. What is a software process framework?

Definition: A software process framework establishes the organizational foundation for an engineering process by defining a small set of generic framework activities applicable to all software projects regardless of scale or complexity.
Five Generic Framework Activities: Communication, Planning, Modeling (Analysis and Design), Construction (Code and Test), and Deployment.
Umbrella Activities: Complemented across the lifecycle by umbrella activities including software quality assurance (SQA), configuration management, and risk tracking.

Q3. What is the Spiral Model?

Definition: Developed by Barry Boehm in 1986, the Spiral Model is an evolutionary, risk-driven software development process model that couples the iterative prototyping approach with the systematic, controlled structure of the Waterfall model.
Four Spiral Quadrants: Objective Determination, Risk Assessment and Reduction, Engineering and Validation, and Review and Next-Phase Planning.
Best Suited For: Large-scale, mission-critical, and high-risk software systems requiring continuous risk evaluation.

Q4. What is Functional Requirement?

Definition: Functional requirements are explicit statements detailing the specific services, functionalities, data transformations, calculations, and operational behaviors that a software system must provide to its end users.
Primary Focus: Defines "what the system should do" in response to specific user inputs and system events.

Examples: User login with credential validation, generating monthly sales reports, calculating student grade point averages, and automated cart checkout processing.

Q5. What is Non-functional Requirement?

Definition: Non-functional requirements (NFRs), also known as software quality attributes, specify the operational characteristics, performance criteria, and architectural constraints under which the system must operate.
Primary Focus: Defines "how well the system performs" its functions (judging quality rather than specific behaviors).
Key Dimensions: Response time latency (performance), data encryption (security), 99.9% uptime (reliability), usability, and horizontal scalability.

Q6. What is a Data Flow Diagram?

Definition: A Data Flow Diagram (DFD) is a structured graphical modeling tool that illustrates how data moves, transforms, and is stored throughout an information system, without depicting control logic or decision branching.
Four Notation Elements:

- **External Entity (Source/Sink):** Rectangles representing external origins or consumers of data.
- **Process:** Circles or rounded rectangles representing actions that transform data inputs into outputs.
- **Data Store:** Open-ended rectangles representing files, tables, or repositories at rest.
- **Data Flow:** Directed arrows depicting the direction of data movement.

Q7. What is integration testing?

Definition: Integration testing is a systematic level of software testing where individually tested unit modules are combined and tested collectively to expose interface, data communication, and interaction defects between integrated components.
Integration Approaches:

- **Top-Down Integration:** Modules are integrated from top control levels downward using dummy subordinate stubs.
- **Bottom-Up Integration:** Low-level atomic modules are tested upward using test drivers.
- **Big Bang / Sandwich:** Non-incremental simultaneous integration or combined hybrid testing.

Q8. What is software reengineering?

Definition: Software reengineering is the systematic examination, restructuring, and alteration of an existing legacy software system to reconstitute it in a new architectural form, improving maintainability, quality, and performance without changing its core business functionality.
BPR Process Stages: Inventory Analysis → Document Restructuring → Reverse Engineering → Code & Data Restructuring → Forward Engineering.
Core Benefit: Extends legacy software operational lifetime while reducing ongoing maintenance overhead.

Q9. Define project scheduling.

Definition: Project scheduling is the project management procedure of allocating estimated software engineering effort across a planned calendar timeline by decomposing tasks, establishing sequential dependencies, and allocating resources.
Standard Techniques: Work Breakdown Structure (WBS), Gantt Charts, Critical Path Method (CPM), and Program Evaluation and Review Technique (PERT).
Core Objective: To define measurable project milestones, track task progress, and guarantee software delivery within contracted completion dates.

Q10. Define software risk and name its main categories.

Definition: A software risk is a potential future event, uncertainty, or vulnerability that, if it occurs, will cause adverse impacts on project timelines, budget expenditures, software quality, or operational success.
Main Risk Categories:

- **Project Risks:** Threaten budget, resources, or schedule (e.g., loss of experienced staff).
- **Technical Risks:** Threaten software quality and implementation viability (e.g., unproven technologies).
- **Business Risks:** Threaten commercial product viability (e.g., market competition, loss of executive sponsorship).
- **Predictability Classes:** Known risks, Predictable risks, and Unpredictable risks.

MODEL QUESTION PAPER – 4

Q1. What is meant by software engineering ethics?

Definition: Software engineering ethics refers to the moral framework, professional obligations, and legal standards that guide software practitioners in making responsible technical and business decisions throughout the software lifecycle.
Core Principles (IEEE-CS/ACM Code): Public health and safety, confidentiality of client data, product quality integrity, professional competence, and refusal to participate in deceptive or harmful computing practices.
Objective: To ensure that software technology serves the public welfare and protects individual privacy.

Q2. What is the Incremental Delivery Model?

Definition: The Incremental Delivery Model is an evolutionary software development process where system requirements are prioritized and developed, tested, and released to the customer as a succession of usable, partial product increments.
Mechanism: The highest-priority core functions are developed and deployed in the initial increment, with subsequent increments progressively adding supplementary functionality based on user feedback.
Key Advantage: Provides rapid delivery of early business value and mitigates overall project risk.

Q3. What is Agile software development?

Definition: Agile software development is a lightweight, adaptive, and human-centric software engineering philosophy governed by the four core values and twelve principles of the Agile Manifesto (2001).
Core Philosophy: Prioritizes individuals and interactions, working software, customer collaboration, and rapid responsiveness to evolving requirements over rigid upfront planning.
Common Frameworks: Scrum, Extreme Programming (XP), Kanban, and Lean Software Development.

Q4. What is Software Requirement Specification (SRS) Document?

Definition: The Software Requirement Specification (SRS) is a formal, comprehensive document that completely defines the intended external behavior, functional capabilities, and operational constraints of a software system before development begins.
Core Contents: Functional requirements, non-functional requirements (performance, reliability, security), system interface definitions, design constraints, and data requirements.
Role: Serves as a contractual agreement between stakeholders and developers, and acts as the official validation baseline for acceptance testing.

Q5. Define UML.

Definition: UML (Unified Modeling Language) is a standardized, general-purpose graphical modeling language used in object-oriented software engineering to specify, visualize, construct, and document the structural and behavioral artifacts of software systems.
Standard Body: Standardized and managed by the Object Management Group (OMG).
Two Major Categories:

- **Structural Diagrams:** Class diagrams, Object diagrams, Component diagrams, Deployment diagrams.
- **Behavioral Diagrams:** Use Case diagrams, Sequence diagrams, Activity diagrams, State Machine diagrams.

Q6. What is a coding standard?

Definition: A coding standard is an agreed-upon set of programming rules, syntactic conventions, formatting styles, and architectural guidelines that developers within an organization must strictly adhere to when authoring source code.
Key Elements: Identifier naming conventions (e.g., camelCase for variables, PascalCase for classes), indentation rules, maximum function length, file header comments, and defensive exception handling.
Primary Benefits: Enhances code readability, reduces software defect density, and simplifies long-term maintenance.

Q7. What is system testing?

Definition: System testing is the high-level phase of black-box testing where the complete, fully integrated software product is rigorously evaluated against specified system and business requirements in an environment simulating real-world production.
Scope: Evaluates end-to-end functionality, system security, performance, stress resistance, and recovery under peak operational workloads.
Major Types: Functional system testing, Performance testing (Load, Stress), Security testing, and Usability testing.

Q8. What is black-box testing?

Definition: Black-box testing (also known as behavioral or functional testing) is a software testing methodology where test cases are designed and executed based strictly on external system specifications without any knowledge or examination of the internal code structures, logic paths, or implementation algorithms.
Focus: Verifying that specified inputs produce expected outputs and that interface requirements are satisfied.
Core Techniques: Equivalence Class Partitioning (ECP), Boundary Value Analysis (BVA), and Decision Table testing.

Q9. What is Software Maturity Index?

Definition: The Software Maturity Index (SMI) is an established software quality metric used to quantify the architectural stability and maintenance maturity of a software product release based on the volume of modifications made between successive releases.
Mathematical Formula: `SMI = [Mt - (Fa + Fc + Fd)] / Mt`
Notation:

- `Mt` : Total number of modules/functions in the current software release.
- `Fa` : Number of modules/functions added to this release.
- `Fc` : Number of modules/functions changed in this release.
- `Fd` : Number of modules/functions deleted from the preceding release.

Interpretation: As the software product stabilizes, SMI approaches 1.0 (indicating minimal changes and high stability).

Q10. What is PERT? State its expected-time formula.

Definition: PERT (Program Evaluation and Review Technique) is a statistical project scheduling and network modeling tool used in software project management to plan, schedule, and estimate project durations when individual task completion times are subject to high uncertainty.
Three Time Estimates:

- **Optimistic Time (To):** Minimum possible time required under ideal conditions.
- **Most Likely Time (Tm):** Most realistic duration estimate under normal conditions.
- **Pessimistic Time (Tp):** Maximum time required under worst-case circumstances.

Expected Time Formula: `Te = (To + 4Tm + Tp) / 6`

MODEL QUESTION PAPER – 5

Q1. What is software engineering? State any two goals.

IEEE Definition: According to IEEE 610.12, software engineering is the systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering principles to software.
Two Major Goals:

- To develop high-quality, reliable, and fault-tolerant software that completely satisfies client requirements.
- To deliver software solutions on schedule and within allocated budget constraints while facilitating future maintainability.

Q2. What is an SRS document?

Expansion & Definition: SRS stands for Software Requirements Specification. It is an official, structured technical document that completely describes the expected external behavior, functional capabilities, performance criteria, and operational constraints of a software system.
Core Contents: Functional requirements, non-functional requirements (security, portability, reliability), system design constraints, and external interface descriptions.
Role: Serves as a contractual baseline between clients and developers, forming the foundation for architectural design, project cost estimation, and validation testing.

Q3. What is requirement elicitation?

Definition: Requirement elicitation (also known as requirements gathering) is the initial, collaborative phase of Requirements Engineering focused on discovering, collecting, and understanding customer expectations, business objectives, and system constraints from stakeholders, domain experts, and end-users.
Common Techniques: One-on-one stakeholder interviews, surveys and questionnaires, Joint Application Development (JAD) sessions, on-site task observation, and user story workshops.

Q4. What is internal documentation?

Definition: Internal documentation refers to the technical descriptions, explanatory comments, and structural readability mechanisms embedded directly within the program source code itself to assist software engineers with program maintenance and code reviews.
Key Elements: Meaningful identifier and variable naming conventions, file/module header blocks (author, version, date), inline comments explaining non-trivial logic, and function docstrings detailing parameters and return types.

Q5. What is regression testing?

Definition: Regression testing is the quality verification process of re-executing a selective suite of previously passed test cases following code modifications (such as bug fixes, software patches, refactoring, or feature additions) to ensure that existing system functionalities have not been broken or degraded.
Core Objective: To preserve baseline software stability and verify that fixing one software defect does not inadvertently introduce new regression defects elsewhere.
Execution Mode: Commonly automated through continuous integration (CI/CD) test suites.

Q6. What is Defect Removal Efficiency?

Definition: Defect Removal Efficiency (DRE) is an essential software quality and process metric that quantifies the percentage of software defects identified and successfully resolved by the development team prior to commercial release.
Mathematical Formula: `DRE = [ E / (E + D) ] × 100`
where `E` is the number of errors discovered internally before software delivery (via reviews, inspections, and testing), and `D` is the number of defects reported by customers post-release.
Significance: A higher DRE percentage (approaching 100%) indicates superior pre-release verification and testing effectiveness.

Q7. What is Cleanroom Software Development?

Origin & Definition: Developed by Harlan Mills at IBM, Cleanroom Software Development is a formal software engineering process model that emphasizes defect prevention through mathematical formal specifications, formal correctness verification, and statistical quality control to deliver certified zero or near-zero defect software.
Core Characteristics:

- Replaces traditional developer unit testing and ad-hoc debugging with rigorous mathematical verification of correctness.
- Employs independent statistical usage testing based on operational usage profiles to certify software reliability.

Q8. What is software reliability?

IEEE Definition: Software reliability is the probability that a software system will perform its required functions correctly and without failure under specified environmental conditions for a specified period of time.
Primary Measurement Metrics:

- **Mean Time To Failure (MTTF):** The average uptime duration of the system prior to encountering a failure.
- **Mean Time Between Failures (MTBF):** Total operational cycle time defined as `MTBF = MTTF + MTTR` (Mean Time To Repair).
- **Failure Intensity / Rate (λ):** The frequency of failures occurring per unit execution time.

Q9. What is Software Quality Assurance (SQA)?

Definition: Software Quality Assurance (SQA) is an umbrella, systematic pattern of organizational actions, reviews, audits, and monitoring activities designed to provide adequate management confidence that the software development process adheres to established engineering standards, processes, and guidelines.
Key Activities: Formal technical reviews (FTR), process audits, compliance verification (e.g., ISO 9001, CMMI models), coding standard enforcement, and defect tracking.
Distinction: SQA is process-oriented (focused on defect prevention), whereas software testing is product-oriented (focused on defect detection).

Q10. What is a CASE tool?

Expansion & Definition: CASE stands for Computer-Aided Software Engineering. A CASE tool is a specialized software application or integrated software environment that provides automated or semi-automated support for engineering activities across the software development life cycle (SDLC).
Major Classifications:

- **Upper CASE Tools:** Support early planning, requirement engineering, and architectural UML modeling (e.g., Enterprise Architect, Rational Rose).
- **Lower CASE Tools:** Support downstream implementation, code generation, testing, and debugging (e.g., Eclipse IDE, Selenium, Git).
- **Integrated CASE (I-CASE):** Provides continuous, unified support spanning the entire SDLC life cycle.

PART II: SECTION B — 6 MARKS QUESTIONS (PAPERS 1 TO 5)

MODEL QUESTION PAPER – 1

Q11. Explain the characteristics of software in detail.

1. Core Concept

Software is a logical rather than a physical system element. Its engineering design, operational failure profile, and lifecycle evolution differ fundamentally from physical manufactured hardware.

2. Failure Curve Comparison

```
Failure Rate         Hardware Bathtub             Actual Software Curve
     ^              \               /            \   /\    /\    /\   /
     |               \_____________/              \_/  \__/  \__/  \_/
     +------------------------------> Time      (Spikes from maintenance patches)
```

3. Primary Characteristics

- **Engineered, not manufactured:** Development costs reside in intellectual design and testing; replication and deployment cost virtually zero.
- **Does not wear out:** Software is immune to physical wear and friction; failures arise from design defects and logical bugs present from inception.
- **Deteriorates via changes:** Repeated maintenance patches introduce side-effects and code entropy, elevating the baseline failure rate over time.
- **Custom-built nature:** While component reuse is growing, most enterprise software systems remain custom-tailored to specific business logic.
- **Inherent complexity:** Software encompasses millions of execution branches and states that are invisible and cannot be physically inspected.

4. Summary Comparison

Feature	Hardware	Software
Production	Factory assembly of physical parts.	Intellectual design and coding.
Wear Out	Wears out physically over time.	Deteriorates logically via patches.
Components	Standard off-the-shelf ICs.	Custom-built programmatic modules.

Q12. Explain software crisis and need for software engineering.

1. Core Concept

The Software Crisis was identified at the 1968 NATO Conference to describe the systemic failure of the software industry to build reliable software on schedule and within budget.

2. Crisis vs Engineering Resolution

```
[Historical Crisis: 1968]                      [Engineering Solution]
- Budget overruns (200-500%)   === Adopt ===>   - Structured SDLC Models
- Schedule slippages (years)   Software Engg.   - Formal Quality & Testing
- Software cost reached 90%                     - Modular & Maintainable Code
```

3. Major Symptoms of Software Crisis

- **Severe cost overruns:** Projects routinely exceeded original financial estimates by 200% to 500% due to ad-hoc development.

- **Chronic delivery delays:** Software schedules slipped by months or years, often rendering systems obsolete before deployment.
- **Poor quality and reliability:** Delivered software was riddled with latent defects, crashes, and unhandled runtime exceptions.
- **Unmaintainable spaghetti code:** The absence of modular design made software nearly impossible to maintain without introducing new bugs.
- **Cost ratio inversion:** Hardware costs dropped while software development and maintenance costs exploded to over 90% of total IT budgets.

4. Need for Software Engineering

Software engineering provides disciplined SDLC processes, empirical cost estimation (COCOMO), modular design, and formal testing to guarantee on-time, budget-compliant, and reliable systems.
Real-World Example: The cancellation of the 1992 London Ambulance Dispatch System highlighted the disaster of building software without engineering discipline.

Q13. Explain software process models and their classification.

1. Core Concept

A Software Process Model (SDLC model) is a structured framework defining the sequence of activities, deliverables, and milestones from project conception to maintenance.

2. Classification Hierarchy



3. Primary Model Categories

- **Linear-Sequential (Waterfall, V-Model):** Executes stages in a strict linear order without feedback loops; suited for stable, well-understood requirements.
- **Prototyping Model:** Rapidly builds a working mock-up to clarify ambiguous requirements via early user feedback.
- **Spiral Model (Boehm):** Risk-driven evolutionary model combining prototyping iterations with waterfall control across four rotating quadrants.
- **Incremental Model:** Delivers software in staggered, usable releases, deploying core features early.
- **Agile Models (Scrum, XP):** Lightweight, iterative frameworks focusing on 2–4 week sprints, continuous collaboration, and rapid adaptability.

4. Summary Comparison Table

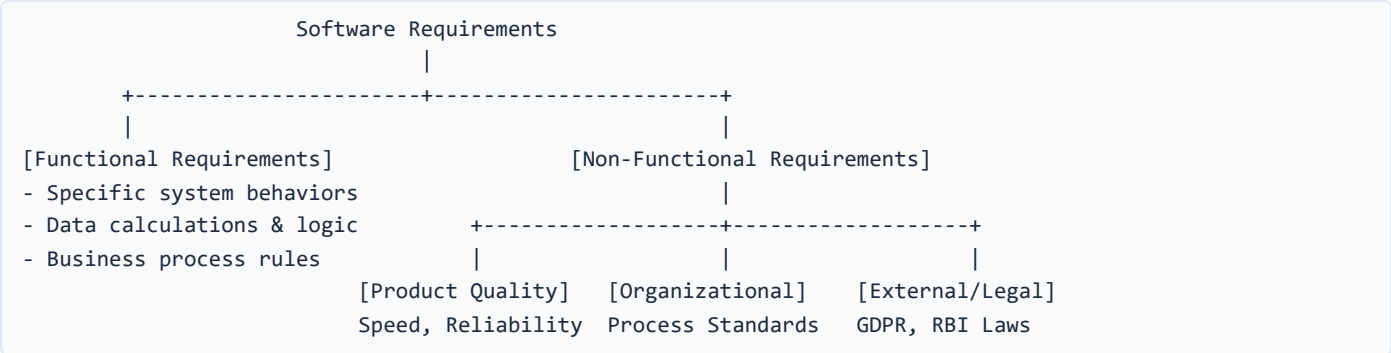
Category	Key Models	Strengths	Ideal Project Fit
Linear	Waterfall, V-Model	Simple, disciplined milestones.	Stable, clear requirements.
Evolutionary	Prototyping, Spiral	Early feedback, strong risk analysis.	Complex, high-risk systems.
Agile	Scrum, XP, Kanban	Fast releases, high adaptability.	Dynamic apps with changing needs.

Q14. Explain the Types of Requirements.

1. Core Concept

A software requirement is an essential capability or operational constraint that a system must satisfy to fulfill user goals.

2. Requirements Taxonomy



3. Key Requirement Categories

- **User vs System Requirements:** User requirements are high-level goals for clients; system requirements are detailed specifications for engineers.
- **Functional Requirements (FRs):** Specific services and computations the system must execute (e.g., calculate tax and generate invoices).
- **Non-Functional Requirements (NFRs):** Quality constraints defining performance, security, and reliability (e.g., response time < 1 second).
- **Organizational Requirements:** Constraints from corporate policies, such as using Java 17 and Agile Scrum.
- **External & Domain Requirements:** Mandates from legal regulations (GDPR) or domain rules (banking interest formulas).

4. Summary Comparison Table

Type	Core Focus	Verification	Concrete Example
Functional	What system does.	Test execution.	Transfer funds between accounts.
Non-Functional	How well it performs.	Load & stress tests.	Support 10,000 concurrent users.
Domain	Domain rules.	Expert audit.	Calculate interest per laws.

Q15. Explain metrics for the Analysis Model.

1. Core Concept

Analysis model metrics provide quantitative measurements during requirements analysis to estimate system size, functional scope, and development effort prior to architectural design.

2. Function Point Formula

```
Albrecht Function Point Formula:
  FP = UFP * [0.65 + 0.01 * SUM(Fi)]
  - UFP: Unadjusted Function Points (weighted sum of 5 domain elements)
  - Fi: 14 Value Adjustment Factors rated 0 to 5 (scaling factor 0.65 to 1.35)
```

3. 5 Information Domain Characteristics

- **External Inputs (EI):** User transactions and input data that update internal database tables (e.g., registration form).
- **External Outputs (EO):** Reports, calculations, or messages generated for users (e.g., printed salary slip).
- **External Inquiries (EQ):** Search requests that retrieve stored data without modifying internal tables (e.g., balance check).
- **Internal Logical Files (ILF):** Logical database tables maintained entirely within application boundaries (e.g., Customer Table).
- **External Interface Files (EIF):** Reference data files maintained by external systems across boundaries (e.g., credit rating file).

4. Complexity Weight Matrix

Element	Low Weight	Average Weight	High Weight
Inputs (EI) / Inquiries (EQ)	3	4	6
Outputs (EO)	4	5	7
Internal Files (ILF)	7	10	15
External Interfaces (EIF)	5	7	10

Q16. Explain the meaning, goals, and importance of software maintenance.

1. Core Concept

Software Maintenance is the engineering process of modifying, optimizing, and supporting a software product after its operational deployment, spanning the entire remaining lifespan of the system.

2. Maintenance Effort Distribution

```
+-----------------------------------------------------------+
|          Distribution of Software Maintenance Effort      |
|  [Perfective: ~50%]  User feature additions & performance |
|  [Adaptive:   ~25%]  OS, DB, and regulatory updates       |
|  [Corrective: ~20%]  Runtime bug fixing and crash patches |
|  [Preventive: ~5%]   Code refactoring & re-engineering    |
+-----------------------------------------------------------+
```

3. Four Standard Maintenance Categories

- **Corrective Maintenance (~20%):** Reactive patching of latent bugs, computational errors, and crashes reported by live users.
- **Adaptive Maintenance (~25%):** Modifying the software to remain compatible with evolving hardware platforms, operating systems, or legal regulations.
- **Perfective Maintenance (~50%):** Proactively implementing new user features, workflow refinements, and performance optimizations requested by clients.
- **Preventive Maintenance (~5%):** Re-engineering and refactoring source code to improve maintainability and prevent future system failures.
- **Economic Dominance:** Maintenance consumes 60% to 80% of total software lifecycle costs, far exceeding original development budgets.

4. Lehman's Laws of Software Evolution

- **Law of Continuing Change:** A real-world system must continually adapt to changing environments, or it becomes progressively less useful.
- **Law of Increasing Complexity:** Ongoing maintenance increases system complexity unless explicit refactoring effort is invested to simplify it.

MODEL QUESTION PAPER – 2

Q11. Explain software products and their types with suitable examples.

1. Core Concept:

A **Software Product** is a complete software deliverable comprising executable code, data structures, and documentation delivered to end users. Products are divided into Generic (off-the-shelf) and Customized (bespoke) systems.

2. Classification Framework:



3. Key Points & Characteristics:

- **Generic Software Products:** Standalone systems developed by an organization and sold openly on the commercial market to any customer who needs their capabilities.
- **Customized (Bespoke) Products:** Software commissioned and engineered specifically for a single client to satisfy their unique operational workflows.
- **Requirement Control:** Generic product requirements are determined and controlled by the developer; customized requirements are dictated strictly by the purchasing client.
- **Ownership & IP:** The vendor retains intellectual property rights in generic software, whereas bespoke software IP is typically transferred to the client.

4. Comparison Table & Examples:

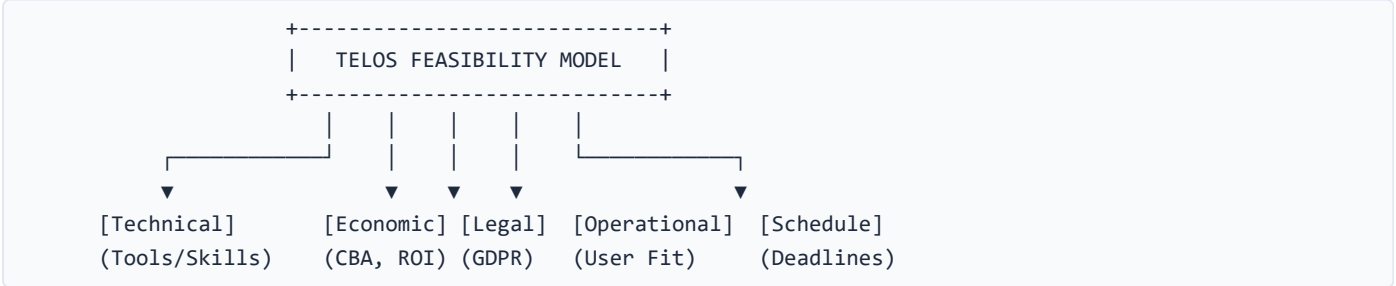
Product Type	Target Audience	Cost Model	Real-World Examples
Generic (COTS)	Mass public market	Low cost per seat	Microsoft Windows 11, Adobe Photoshop, Oracle Database
Customized (Bespoke)	Single enterprise	High development cost	SBI Core Banking Portal, Air Traffic Control, Hospital ERP

Q12. Explain the Types of Feasibility Study.

1. Core Concept:

A **Feasibility Study** is a preliminary evaluation conducted during SDLC inception to assess whether a proposed software project is technically practical, financially sound, legally compliant, and worth corporate investment.

2. The TELOS Feasibility Framework:



3. Five Key Dimensions of Feasibility:

- **Technical Feasibility:** Evaluates whether the system can be built using existing hardware, software, development frameworks, and the engineering team's current technical skills.
- **Economic Feasibility:** Conducts a Cost-Benefit Analysis (CBA) comparing development expenditures against financial returns, evaluating Return on Investment (ROI) and Break-Even Point.
- **Legal Feasibility:** Ensures the software complies with intellectual property rights, open-source licenses, labor laws, and data privacy regulations (e.g., GDPR, IT Act).
- **Operational Feasibility:** Evaluates how effectively the software solves business problems and determines end-user readiness and organizational adoption.

- **Schedule Feasibility:** Assesses whether the project can be realistically developed and delivered within customer-imposed calendar deadlines.

4. Primary Deliverable:

Feasibility Report: Summarizes analytical findings and provides an executive recommendation to **Proceed (Go)**, **Revise Scope (Modify)**, or **Abandon (No-Go)**.

Q13. Explain software design and its importance.

1. Core Concept:

Software Design is the technical engineering phase that transforms user requirements from the SRS document into an architectural blueprint, data structures, interface specifications, and procedural algorithms before programming begins.

2. Three Levels of Software Design:

```
[SRS Requirements] ──> [1. Architectural Design]  (Subsystems & frameworks)
                                    │
                                    ▼
                           [2. High-Level Design]     (Modules & interfaces)
                                    │
                                    ▼
                           [3. Detailed Design]       (Algorithms & data structures)
                                    │
                                    ▼
                           [Source Code Coding]
```

3. Importance of Software Design:

- **Quality Baseline:** Embeds critical quality factors (robustness, security, efficiency) into the software architecture before coding, preventing expensive structural rework.
- **Modularity:** Decomposes complex systems into independent, cohesive modules with low coupling, enabling parallel development across engineering teams.
- **Maintainability:** Simplifies bug isolation and localized code modifications without triggering unintended side-effect defects across other subsystems.
- **Software Reusability:** Standardized architectural patterns and component designs can be repurposed across other corporate projects, reducing development costs.
- **Cost & Risk Reduction:** Identifies architectural flaws and resource bottlenecks early, ensuring accurate schedule and cost estimation.

4. Practical Example:

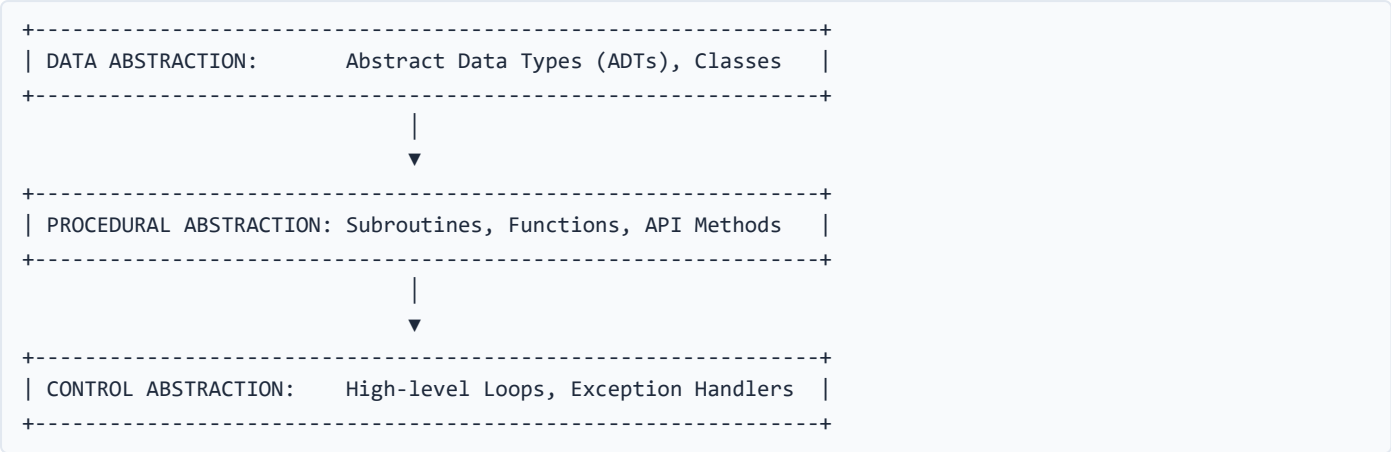
In an e-commerce platform, architectural design partitions the system into `Catalog`, `Payment`, and `Shipping` microservices, allowing the payment gateway to be upgraded independently without disrupting catalog browsing.

Q14. Explain abstraction with suitable examples.

1. Core Concept:

Abstraction is a fundamental design principle that focuses on the essential, high-level behavioral characteristics of an entity while hiding non-essential, complex implementation details to manage system complexity.

2. Levels of Abstraction Model:



3. Key Points & Levels of Abstraction:

- **Data Abstraction:** Defines data entities based on allowable operations rather than internal storage representation (e.g., a `Stack` class exposing `push()` and `pop()` while hiding array vs linked-list memory structures).
- **Procedural Abstraction:** Encapsulates a sequence of programmatic instructions under a descriptive name (e.g., calling `sort()` or `Math.sqrt()` without knowing the underlying algorithm).
- **Control Abstraction:** Provides high-level control flow mechanisms (e.g., `for-each` loops, `try-catch` blocks) without manual machine-level register manipulation.
- **Complexity Management:** Allows developers to conceptualize systems in layered, manageable tiers without cognitive overload.

4. Real-World Practical Example:

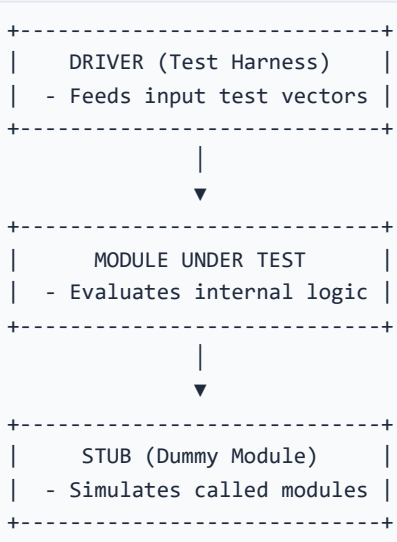
Automobile Interface: A driver operates a car using a steering wheel, accelerator, and brake pedals (high-level abstraction) without needing to understand fuel injection timing or hydraulic fluid transmission mechanics.

Q15. Describe unit testing and the important unit test considerations.

1. Core Concept:

Unit Testing is the lowest-level software verification phase where individual software components, subroutines, or classes are tested in complete isolation to verify that their internal logic operates strictly according to detailed design specifications.

2. Unit Test Environment (Test Harness):



3. Five Important Unit Test Considerations:

- **Module Interface:** Verifies that input parameter count, order, data types, and return values strictly match specifications.

- **Local Data Structures:** Checks for uninitialized variables, default value faults, type mismatches, and memory leaks within the unit.
- **Boundary Conditions:** Tests edge limits and off-by-one conditions (e.g., maximum/minimum array bounds, empty strings, null pointers).
- **Independent Paths:** Employs basis path testing to ensure every execution branch (e.g., all `if-else` paths, `case` statements) runs at least once.
- **Error Handling Paths:** Ensures unexpected inputs trigger graceful error messages and exceptions rather than system crashes.

4. Execution Role:

Conducted white-box by developers during the coding phase using automated frameworks (e.g., JUnit, PyTest) with drivers and stubs to discover defects at the earliest, least costly stage.

Q16. Explain software project planning and its purpose.

1. Core Concept & Purpose:

Software Project Planning is the initial project management phase that defines project scope, estimates required resources, evaluates risks, and creates a realistic schedule to deliver quality software within budget and deadline constraints.

2. Planning Process Flow:

```
[1. Scope Definition] ──> [2. Size & Cost Estimation] ──> [3. Resource Allocation]
                                                                    |
[6. SPMP Plan Document] <── [5. Project Scheduling]   <── [4. Risk Analysis]
```

3. Essential Planning Activities:

- **Project Scope Definition:** Establishes functional boundaries, deliverables, and performance goals to prevent project scope creep.
- **Software Estimation:** Predicts software size (LOC or Function Points) and estimates required effort and cost using empirical models like Boehm's COCOMO.
- **Resource Allocation:** Identifies and schedules human engineering talent, software tools, hardware testbeds, and cloud environments.
- **Risk Analysis & Management:** Identifies potential project, technical, and business risks and constructs a formal Risk Mitigation, Monitoring, and Management (RMMM) plan.
- **Project Scheduling:** Decomposes work into tasks using a Work Breakdown Structure (WBS) and establishes delivery milestones using Gantt Charts and PERT networks.

4. Primary Deliverable:

SPMP Document: The Software Project Management Plan formalizes the baseline operational roadmap that guides development and manages client expectations.

MODEL QUESTION PAPER – 3

Q11. Differentiate between the Classical Waterfall Model and the Iterative Waterfall Model.

1. Core Concept

Definition: The Classical Waterfall Model is an idealistic, strictly linear-sequential SDLC model without feedback loops, whereas the Iterative Waterfall Model introduces backward feedback paths between adjacent phases to rectify defects discovered downstream.

2. Architecture Comparison Workflow

```
Classical Waterfall (Strictly One-Way):
[ Requirements ] --> [ Design ] --> [ Coding ] --> [ Testing ] --> [ Maintenance ]

Iterative Waterfall (With Adjacent Feedback Loops):
[ Requirements ] <--> [ Design ] <--> [ Coding ] <--> [ Testing ] --> [ Maintenance ]
```

3. Key Differences

- **Workflow Structure:** Classical is strictly unidirectional; Iterative incorporates feedback cycles between adjacent phases.
- **Defect Rectification:** Classical provides no mechanism to fix defects from earlier phases; Iterative allows backtracking to the immediate previous stage.
- **Practical Viability:** Classical is theoretical and rarely feasible; Iterative is practical and widely used in commercial software.
- **Requirement Flexibility:** Classical assumes stable requirements with zero change; Iterative accommodates minor requirement refinements.
- **Risk Exposure:** Classical carries high risk because integration flaws appear only at the end; Iterative lowers risk via early defect isolation.

4. Comparison Summary Table

Comparison Parameter	Classical Waterfall Model	Iterative Waterfall Model
Feedback Loops	Completely absent	Present between adjacent phases
Practical Applicability	Theoretical / Idealistic	Practical and industry-usable
Error Recovery	No error recovery	Backtracking to prior phase
Project Risk	Very high risk	Moderate risk

Q12. Explain functional independence with high cohesion and low coupling.

1. Core Concept

Definition: Functional independence is a software design principle where each module performs a single, specific task with minimal dependence on other modules. It is achieved by maximizing cohesion (internal functional strength) and minimizing coupling (inter-module interdependence).

2. Modular Design Spectrum Diagram

```
[ Low / Worst Cohesion ]  ======================>  [ High / Best Cohesion ]
(Coincidental, Logical, Temporal)                   (Sequential, Functional)

[ High / Worst Coupling ]  ======================>  [ Low / Best Coupling ]
(Content, Common, Control)                          (Stamp, Data Coupling)
```

3. Key Points and Classifications

- **Cohesion Concept:** Measures how closely related the internal tasks of a single module are. Higher cohesion indicates better design.
- **Cohesion Levels (Worst to Best):** Coincidental → Logical → Temporal → Procedural → Communicational → Sequential → Functional.
- **Coupling Concept:** Measures the degree of interconnectivity and data sharing between distinct modules. Lower coupling indicates better design.
- **Coupling Levels (Worst to Best):** Content → Common → Control → Stamp → Data → Uncoupled.
- **Engineering Benefits:** Functional independence limits fault propagation (ripple effect), simplifies unit testing, and enhances code reusability.

4. Best Practice Design Target

Design Target	Level to Achieve	Practical Example
Target Cohesion	Functional Cohesion	Module executes one single task: `calculateTax(income)`.
Target Coupling	Data Coupling	Modules exchange only primitive arguments through parameter lists.

Q13. Describe integration testing and explain why it is necessary.

1. Core Concept

Definition: Integration testing is a systematic testing phase in which individually unit-tested modules are combined and evaluated as a group to uncover defects in component interfaces, parameter passing, and inter-module communication.

2. Test Harness Hierarchy Diagram

```
Top-Down vs Bottom-Up Test Harnesses:
            [ Top Module (Driver Harness) ]
                          |
                          v
              [ Module Under Evaluation ]
                          |
                          v
            [ Subordinate (Stub Routine) ]
```

3. Why Integration Testing is Necessary

- **Interface Incompatibilities:** Calling and receiving modules may mismatch on data types, argument ordering, or variable formats.
- **Shared Data Corruption:** Global variables or shared database tables modified by one module can cause side-effects in other modules.
- **Timing and Race Conditions:** Asynchronous function calls and thread locks only manifest when multiple components interact simultaneously.
- **Cumulative Approximations:** Minor arithmetic rounding errors in individual unit functions compound into significant discrepancies across components.

4. Integration Strategies Summary Table

Strategy	Integration Approach	Harness Used	Major Advantage
Big Bang	All modules combined at once	None	Fast for small programs; debugging is hard.
Top-Down	From control module downward	Stubs (dummy routines)	Verifies high-level architecture early.
Bottom-Up	From low-level leaf modules upward	Drivers (caller harnesses)	Verifies critical utility calculations early.

Q14. Describe validation testing and explain its importance.

1. Core Concept

Definition: Validation testing evaluates whether the completed, integrated software complies with customer requirements and business workflows. It answers Barry Boehm's fundamental question: "*Are we building the right product?*"

2. Verification vs Validation Comparison Matrix

Comparison Parameter	Verification	Validation
Fundamental Question	"Are we building the product right?"	"Are we building the right product?"
Primary Focus	Conformance to design specifications	Conformance to customer operational needs
Evaluation Method	Static (Reviews, walkthroughs, inspections)	Dynamic execution using live test cases
Lifecycle Timing	Continuous across early SDLC phases	Conducted toward the end of development

3. Key Levels of Validation Testing

- **Validation against SRS:** Checks all functional requirements, input validations, and error handlers against the formal specification.
- **Alpha Testing:** Conducted at the developer's site by internal quality engineers or representative users in a controlled environment.
- **Beta Testing:** Conducted at external client locations in live, uncontrolled environments by end users without developer presence.
- **User Acceptance Testing (UAT):** Formal evaluation by the client to confirm contractual compliance before final deployment.

4. Strategic Importance

A software product can pass all unit tests and technical verification checks, yet fail commercially if it misinterprets customer business workflows. Validation ensures customer satisfaction and prevents expensive post-release rework.

Q15. Explain metrics for the Design Model.

1. Core Concept

Definition: Design metrics are quantitative measures applied to architectural models, module hierarchies, and procedural logic prior to coding to evaluate complexity, modular independence, maintainability, and expected defect proneness.

2. Architectural Design Complexity Metrics (Card & Agresti)

- **Structural Complexity:** `S(i) = (f_out(i))^2`, based on the fan-out of module `i`.
- **Data Complexity:** `D(i) = v(i) / (f_out(i) + 1)`, where `v(i)` is the count of input/output data variables.
- **System Complexity:** `C(i) = S(i) + D(i) = (f_out(i))^2 + [v(i) / (f_out(i) + 1)]`.

3. Component-Level Design Metrics

- **Fan-In:** Count of calling modules invoking module `X`; high fan-in indicates good functional reusability.
- **Fan-Out:** Count of subordinate modules called by module `X`; fan-out > 7 indicates excessive complexity.
- **Cohesion & Coupling Metrics:** Measures module independence through parameter ratios and global variable references.
- **McCabe Cyclomatic Complexity:** Evaluates procedural decision logic using the Control Flow Graph (CFG): `V(G) = E - N + 2P = P_nodes + 1`

4. McCabe Complexity Thresholds

Cyclomatic Complexity V(G)	Risk Level	Engineering Implication
1 – 10	Low Risk	Simple, structured module; easy to test and maintain.
11 – 20	Moderate Risk	Moderately complex; requires additional test cases.
> 20	High Risk	Complex and error-prone; candidates for immediate refactoring.

Q16. Describe the major challenges in software maintenance.

1. Core Concept

Definition: Software maintenance is the modification of a software system following delivery to correct bugs, improve performance, or adapt it to changing environments. It represents the longest SDLC phase, consuming **60% to 80%** of total software expenditure.

2. The Software Maintenance Cost Iceberg

```
Software Lifecycle Cost Distribution:
   +------------------------------+
   |   Development (20% - 40%)    |  <-- Above Waterline (Initial visible cost)
===+==============================+=== Waterline
   |                              |
   |   Maintenance (60% - 80%)    |  <-- Below Waterline (Ongoing expenditure:
   |                              |      Corrective, Adaptive, Perfective, Preventive)
   +------------------------------+
```

3. Major Technical and Managerial Challenges

- **Legacy Code and Outdated Docs:** Systems often lack up-to-date documentation, forcing maintainers to reverse-engineer code to understand design intent.
- **Developer Turnover:** Original system authors depart, resulting in the loss of critical unwritten architectural knowledge.
- **Ripple Effect of Changes:** Modifying code in tightly coupled modules often triggers unexpected defects in seemingly unrelated subsystems.
- **Decay of Test Suites:** Existing test suites become obsolete, making manual regression testing slow, expensive, and error-prone.
- **Disproportionate Cost Burden:** Maintaining aging systems absorbs the majority of IT budgets, starving new product innovation.

4. Engineering Mitigations Summary Table

Maintenance Challenge	Engineering Mitigation Strategy
Outdated documentation	Automated reverse engineering tools and living code comments.
Ripple effect defects	Modular refactoring to enforce high cohesion and low coupling.
Regression testing overhead	Automated Continuous Integration (CI/CD) regression test suites.

MODEL QUESTION PAPER – 4

Q11. Explain the role of a software engineer in software development.

1. Core Concept

A software engineer is a technical professional who applies computer science principles, engineering methods, and mathematical analysis to design, develop, test, maintain, and evaluate software systems systematically across the SDLC.

2. Lifecycle Responsibilities Workflow

```
[Requirements] ──▶ [Architecture] ──▶ [Clean Coding] ──▶ [Automated Testing] ──▶ [CI/CD Deploy]
```

3. Key Responsibilities

- **Requirements Analysis:** Evaluates user stories and feasibility to create unambiguous, verifiable functional specifications.
- **Architectural Design:** Designs scalable system blueprints, modular interfaces, and database schemas ensuring loose coupling.
- **Clean Implementation:** Authors modular, readable code following coding standards, defensive programming, and design patterns.

- **Automated Testing:** Implements unit and integration test suites, conducts peer code reviews, and patches bugs early.
- **DevOps & Maintenance:** Manages CI/CD deployment pipelines, monitors production logs, and performs periodic refactoring.

4. Role Summary Matrix

SDLC Phase	Core Responsibility	Primary Deliverable
Analysis & Design	System architecture & interface modeling.	SRS notes, UML class diagrams.
Coding & Testing	Clean modular coding and test automation.	Tested source code, unit test suites.
Operations	Continuous delivery & defect remediation.	CI/CD build scripts, bug patches.

Q12. Explain the different ways of Requirements Elicitation or Gathering.

1. Core Concept

Requirements elicitation is the initial phase of Requirement Engineering focused on discovering and extracting the operational constraints and needs of a software system from diverse stakeholders.

2. Elicitation Architecture Workflow

```
[Stakeholder Needs] ──▶ [Interviews / JAD / Surveys / Observation] ──▶ [Verified Requirements]
```

3. Major Elicitation Techniques

- **Interviews:** Formal structured or open-ended unstructured conversations with stakeholders to uncover in-depth domain expectations and pain points.
- **Questionnaires & Surveys:** Pre-designed forms distributed to a large, geographically dispersed population for quick statistical data.
- **Joint Application Development (JAD):** Intensive collaborative workshops uniting users and developers under a facilitator to reach consensus.
- **Observation (Ethnography):** Direct shadowing of end-users in their operational environment to capture unstated workflows and tacit habits.
- **Prototyping:** Building interactive throwaway wireframes to give users concrete visual models and validate ambiguous requirements.

4. Comparative Evaluation Table

Elicitation Technique	Best Suited For	Primary Advantage	Major Limitation
Interviews	Domain experts	In-depth qualitative insight.	Time-consuming to execute.
Questionnaires	Broad audience	Fast, cost-effective data.	No clarifying follow-ups.
JAD Sessions	Conflicting needs	Rapid consensus resolution.	High scheduling cost.

Q13. Explain software measurement, measures, metrics, and indicators.

1. Core Concept

In software engineering, quantitative management follows Roger Pressman's hierarchy: Measure, Measurement, Metric, and Indicator, converting raw counts into actionable decisions.

2. Measurement Hierarchy Flow

```
[MEASURE] ──▶ [MEASUREMENT] ──▶ [METRIC] ──▶ [INDICATOR]
(Raw Count)   (Counting Act)   (Calculated Ratio) (Actionable Insight)
```

3. Key Terms and Formulas

- **Measure:** A direct quantitative count of a single attribute (e.g., 50 defects or 10,000 LOC).
- **Measurement:** The formal procedural act of collecting a measure using standardized counting rules.
- **Metric:** A calculated ratio relating two or more measures to quantify a property (e.g., `Defect Density = Total Defects / KLOC`).
- **Indicator:** A metric evaluated against known thresholds to provide actionable management insight.
- **Decision Role:** Triggers engineering interventions (e.g., defect density exceeding 1.0 defect/KLOC halts deployment).

4. Comparative Distinction Matrix

Term	Formal Definition	Concrete Example	Operational Role
Measure	Raw quantitative value.	25 defects identified.	Baseline raw count.
Metric	Relational calculated ratio.	2.5 defects / KLOC.	Standardized quality metric.
Indicator	Insight guiding action.	Metric breaches threshold.	Triggers corrective process action.

Q14. Explain abstraction level, completeness, interactivity, and directionality in reverse engineering.

1. Core Concept

Reverse engineering is the process of analyzing an existing software system to identify its components and recover design representations at higher levels of abstraction without altering source code.

2. Recovery Workflow Flow

```
Source Code / Binaries ──[Reverse Engineering]──▶ Syntax Trees ──▶ Architecture Models
```

3. Four Core Characteristics

- **Abstraction Level:** Conceptual sophistication of recovered models, ranging from low (syntax trees, control flow) to high (architectural block diagrams, ER schemas).
- **Completeness:** Percentage of total system information recovered. High completeness captures all execution paths; low completeness recovers partial database schemas.
- **Interactivity:** Degree of human guidance required. Automated tools extract syntax, while human analysts resolve semantic ambiguities and business intent.
- **Directionality:** Engineering workflow scope. Unidirectional extracts models strictly for comprehension; bidirectional (round-trip) synchronizes design changes back into code.

4. Characteristics Summary Matrix

Dimension	Spectrum / Scale	Key Operational Focus	Practical Value
Abstraction Level	Low to High	Syntax trees to Architecture.	System comprehensibility.
Completeness	Partial to Complete	Subsystem vs Full Coverage.	Recovery reliability.
Interactivity	Automated to Manual	Tool execution vs Human input.	Semantic accuracy.

Q15. Explain the basic COCOMO model with its three project modes and equations.

1. Core Concept

Basic COCOMO (Constructive Cost Model), developed by Barry Boehm, is an empirical algorithmic model predicting software effort (Person-Months) and duration (Months) from size in KLOC.

2. Model Equations

```
Effort:   E = a_b * (KLOC)^b_b  (Person-Months)
Duration: D = c_b * (E)^d_b     (Months)
Staff:    SS = E / D            (Persons)
```

3. Three Development Modes

- **Organic Mode:** Small projects with experienced teams, familiar requirements, and minimal constraints (Size: `< 50 KLOC`).
- **Semidetached Mode:** Intermediate projects with mixed teams, moderate constraints, and novel requirements (Size: `50 - 300 KLOC`).
- **Embedded Mode:** Complex, safety-critical projects developed under tight operational and hardware constraints (Size: `> 300 KLOC`).

4. Coefficients Table & Numerical Example

Mode	a_b	b_b	c_b	d_b	Scale
Organic	2.4	1.05	2.5	0.38	Simple (< 50 KLOC)
Semidetached	3.0	1.12	2.5	0.35	Medium (50-300 KLOC)
Embedded	3.6	1.20	2.5	0.32	Complex (> 300 KLOC)

Example (Organic, 20 KLOC):
`Effort E = 2.4 * (20)^1.05 ≈ 56 Person-Months`
`Duration D = 2.5 * (56)^0.38 ≈ 11.55 Months`

Q16. Explain the Gantt chart and its uses in software project scheduling.

1. Core Concept

A Gantt chart is a specialized horizontal bar chart developed by Henry Gantt, used in software project scheduling to plan, allocate, sequence, and track tasks across project timelines.

2. Architectural Layout Diagram

```
WBS Activity              W1   W2   W3   W4   W5   W6   W7   W8   W9
──────────────────────────────────────────────────────────────────────
1. Requirements Analysis  [=====]
2. SRS Milestone                    ◆ (M1)
3. Architectural Design            [======]
4. Implementation                         [========]
5. System Testing                                  [======]
6. Release Milestone                                        ◆ (M2)
──────────────────────────────────────────────────────────────────────
```

3. Structural Features and Uses

- **Task Duration Mapping:** Each software activity is plotted as a horizontal bar where start, end, and length represent scheduled task duration.
- **Milestone Tracking:** Significant zero-duration project events (e.g., SRS Signoff, Release) are marked with diamond symbols (`◆`).
- **Dependency Visualization:** Dependency links illustrate precedence (e.g., Finish-to-Start), showing which tasks must conclude before successor tasks begin.
- **Progress Monitoring:** Shading inside task bars depicts completed work percentage, immediately exposing project slippages.

4. Benefits Summary Table

Scheduling Feature	Gantt Mechanism	Project Benefit
Timeline Tracking	Horizontal time bars.	Prevents deadline slippage.
Precedence Mapping	Activity dependency links.	Pinpoints critical bottlenecks.
Progress Auditing	Shading percent complete.	Provides visual real-time status.

MODEL QUESTION PAPER – 5

Q11. Explain software engineering as a layered technology.

1. Core Concept

Software engineering is a layered technology that provides a disciplined approach to developing quality software. Formulated by Roger Pressman, it rests on an organizational commitment to quality that drives the process, defines methods, and is automated by tools.

2. Layered Architecture Diagram

```
+-------------------------------------------------------------+
|                           TOOLS                             |  <-- Automated CASE support
+-------------------------------------------------------------+
|                         METHODS                             |  <-- Technical "how-to" tasks
+-------------------------------------------------------------+
|                         PROCESS                             |  <-- Framework binding layers
+-------------------------------------------------------------+
|                      QUALITY FOCUS                          |  <-- Foundational bedrock
+-------------------------------------------------------------+
```

3. Key Layers

- **Quality Focus:** The foundational bedrock rooted in Total Quality Management (TQM) fostering continuous process improvement.
- **Process:** The structural glue that binds layers together, establishing milestones, deliverables, quality gates, and change control.
- **Methods:** Provides technical "how-to" procedures across the SDLC, including requirements analysis, modeling, coding, and testing.
- **Tools:** Provides automated Computer-Aided Software Engineering (CASE) support to execute processes and methods efficiently.

4. Summary Table

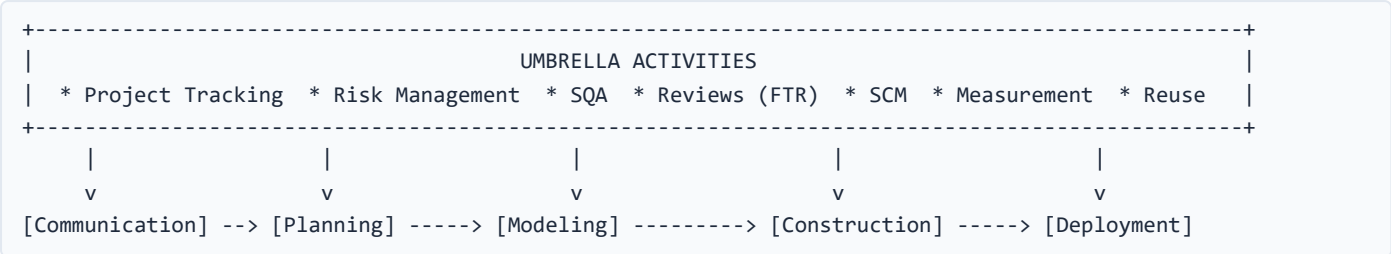
Layer	Metaphor	Primary Role	Example
Quality Focus	Bedrock	Zero-defect commitment	TQM, Six Sigma
Process	Glue	Defines life cycle roadmap	Scrum, Waterfall
Methods	How-to	Technical modeling rules	OOAD, UML
Tools	Leverage	Automates repetitive tasks	Jira, Git, Jenkins

Q12. Explain umbrella activities and their role in software development.

1. Core Concept

Umbrella activities are management and quality assurance functions applied uniformly throughout the software life cycle. They operate continuously across all phases to ensure software quality, risk control, and schedule integrity.

2. Lifecycle Alignment



3. Key Umbrella Activities

- **Project Tracking:** Compares progress against schedule baselines, taking corrective action on deviations.
- **Risk Management:** Identifies and assesses technical risks to prevent project disruptions.
- **SQA:** Conducts planned audits to guarantee conformance to engineering standards.
- **Formal Reviews (FTR):** Peer reviews of work products to uncover defects early.
- **SCM:** Controls changes across documents, code repositories, and release baselines.

4. Summary Table

Activity	Objective	Impact
Project Tracking	Monitor milestone velocity	Prevents budget and schedule slippage
Risk Management	Mitigate vulnerabilities	Proactive risk avoidance
SCM	Coordinate concurrent changes	Version control and baseline traceability
Formal Reviews	Peer inspection	Uncovers defects before release

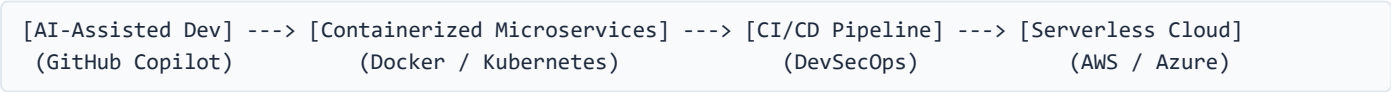
---

Q13. Explain the emerging trends in software development.

1. Core Concept

Modern software development is transitioning from monolithic, manual workflows toward automated, cloud-native, and intelligence-driven engineering paradigms to meet demands for rapid delivery, massive scalability, and continuous deployment.

2. Modern Engineering Ecosystem



3. Key Emerging Trends

- **DevOps & CI/CD:** Integrates development and IT operations into continuous pipelines (Jenkins, GitHub Actions) to automate building, testing, and deployment.
- **Cloud-Native & Serverless:** Architectures built to exploit cloud elasticity (AWS, Azure) and serverless computing (AWS Lambda), abstracting server management.
- **AI-Assisted Engineering:** Generative AI tools (GitHub Copilot) automate routine boilerplate coding, unit test generation, and vulnerability scanning.
- **Microservices Architecture:** Decomposes large monolithic applications into small, independently deployable services communicating via REST APIs or Kafka event streams.
- **DevSecOps:** Integrates automated security analysis and vulnerability scanning directly into early CI/CD pipeline stages.

4. Comparison Table

Dimension	Traditional Paradigm	Emerging Modern Paradigm
Architecture	Monolithic codebase	Distributed microservices
Deployment	Monthly / quarterly releases	Automated continuous releases
Infrastructure	On-premise physical servers	Cloud-native / serverless
Security	Late audit before launch	DevSecOps shift-left automated

---

Q14. Explain coding standards, coding guidelines, and the advantages of coding standards.

1. Core Concept

Source code is read and maintained far more frequently than it is written. Coding standards are mandatory, non-negotiable rules governing code formatting and structure, while coding guidelines are advisory recommendations that guide programmers toward writing clean, maintainable software.

2. Standards vs. Guidelines Structure

```
+-----------------------------------------------------------------------------------------+
| CODING STANDARDS (Mandatory)   --> Naming conventions, indentation, file organization, safety |
+-----------------------------------------------------------------------------------------+
| CODING GUIDELINES (Advisory)   --> Design patterns, code readability, performance advice     |
+-----------------------------------------------------------------------------------------+
```

3. Key Elements & Rules

- **Naming Conventions:** Enforces standardized casing (camelCase for variables userAge, PascalCase for classes InvoiceManager, UPPERCASE for constants MAX_SIZE).
- **Indentation & Formatting:** Mandates uniform spacing (e.g., 4 spaces, no mixed tabs) and character line limits (80–120 characters) for visual clarity.
- **Internal Documentation:** Enforces descriptive function docstrings detailing parameters, return types, and exceptions thrown.
- **Exception Handling:** Prohibits empty catch blocks and mandates structured try-catch-finally error management.

4. Advantages Summary Table

Advantage	Benefit to Development Team
Enhanced Readability	Code written by multiple developers appears uniform and clean.
Lower Maintenance Cost	Speeds up debugging, refactoring, and code comprehension.
Streamlined Code Reviews	Reviews focus on logic and architecture rather than formatting debates.
Defect Prevention	Bounds checking and initialization rules prevent runtime crashes.

Q15. Explain software testing and its importance in software development.

1. Core Concept

Software testing is the process of executing a program with the intent of finding defects (IEEE). It encompasses Verification ("Are we building the product right?") and Validation ("Are we building the right product?").

2. Software Testing Life Cycle (STLC) Workflow

```
[Requirements Analysis] ---> [Test Planning] ---> [Test Case Design]
                                                          |
[Test Closure] <--- [Defect Tracking] <--- [Test Execution] <--- [Environment Setup]
```

3. Key Reasons for Importance

- **Defect Cost Reduction:** Resolving defects during requirements or unit testing is 10 to 100 times cheaper than fixing bugs in production.
- **Quality & Reliability:** Verifies that critical functions operate reliably under boundary inputs and unexpected conditions.
- **Security Hardening:** Identifies vulnerabilities before systems are exploited in production.
- **User Trust & Reputation:** Prevents costly commercial downtime, software crashes, and customer dissatisfaction.
- **Regulatory Compliance:** Essential for mission-critical software requiring safety certifications.

4. Summary Table of Testing Levels

Testing Level	Target Focus	Primary Responsibility
Unit Testing	Individual functions / modules	Developers (JUnit, PyTest)
Integration Testing	Interfaces between modules	Developers & QA Engineers
System Testing	End-to-end integrated application	Independent QA Team
Acceptance Testing	User operational readiness	Clients / End-Users (UAT)

Q16. Explain team management, team composition, and the responsibilities involved in managing a software team.

1. Core Concept

Software team management is the leadership discipline of organizing human engineering talent, assigning roles, establishing communication pathways, and resolving conflicts to deliver quality software within budget and schedule constraints.

2. Team Organizational Models (Mantei Model)

```
[ Democratic Decentralized ]       [ Controlled Decentralized ]       [ Controlled Centralized ]
    (Consensus-driven)                  (Leader + Sub-teams)               (Chief Programmer)
     * Open horizontal                   * Modular coordination             * Strict top-down
     * High satisfaction                 * Industry standard                * Fast, routine tasks
```

3. Key Roles in Team Composition

- **Project Manager (PM):** Tracks project milestones, allocates budgets, manages resources, and coordinates stakeholders.
- **Software Architect:** Defines high-level system structures, module boundaries, and technology stack choices.
- **Lead Developer:** Implements complex modules, conducts peer code reviews, and mentors junior engineers.
- **QA Engineer:** Designs automated test suites, validates acceptance criteria, and manages defect tracking.
- **DevOps Engineer:** Manages CI/CD pipelines, cloud deployments, and runtime monitoring.

4. Summary of Team Paradigms

Paradigm	Authority Style	Communication	Optimal Project Type
Democratic (DD)	Consensus (No boss)	Horizontal (All-to-all)	Complex research, exploratory tasks
Controlled (CD)	Leader with sub-teams	Hierarchical & horizontal	Large modular enterprise systems
Centralized (CC)	Strict Chief Programmer	Vertical only	Simple, routine, tight-deadline projects

---

PART III: SECTION C — 8 MARKS QUESTIONS (PAPERS 1 TO 5)

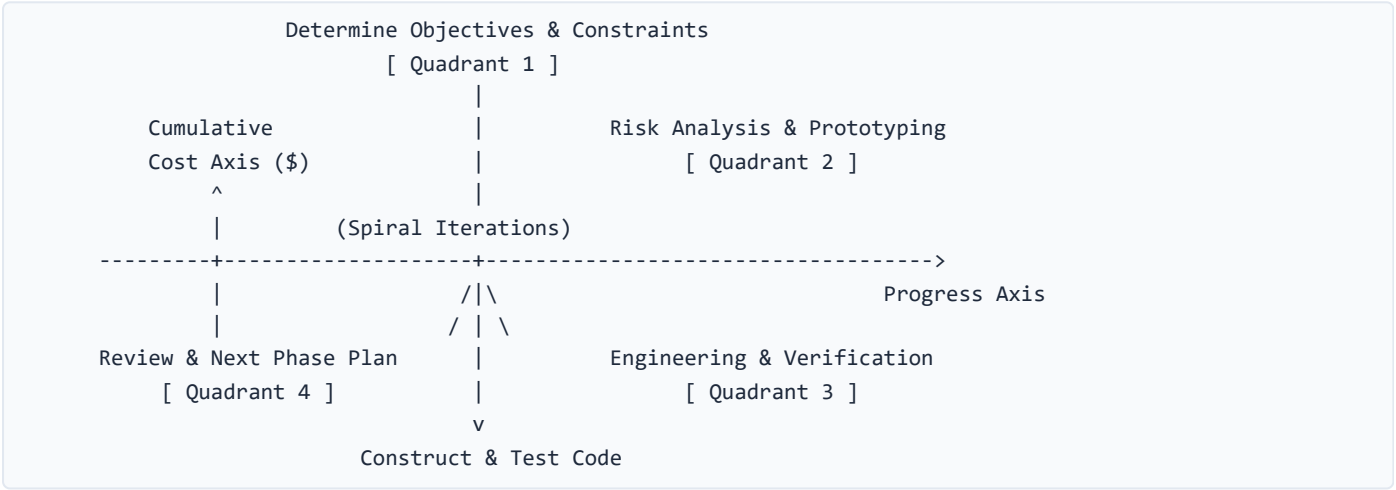
MODEL QUESTION PAPER – 1

Q17. Explain the Spiral Model with its four phases.

1. Core Concept

Proposed by Barry W. Boehm in 1988, the Spiral Model is an evolutionary, risk-driven software development process model that couples iterative prototyping with controlled, systematic aspects of the Waterfall model. It organizes software production into repeated concentric cycles around an origin point. The radial dimension from the origin represents cumulative development cost incurred, while the angular dimension represents progress completed within each lifecycle phase. The model was specifically engineered to accommodate rapid evolution and risk management in mission-critical applications.

2. Four-Quadrant Architecture Diagram



3. Four Core Operational Phases

- **Determine objectives and constraints:** Defines project goals, performance targets, alternative implementation paths, and strict technical constraints such as cost, platform compatibility, and operational environments.
- **Risk analysis and resolution:** Evaluates technical and managerial risks, resolving uncertainty through analytic modeling, benchmarking, simulation, and disposable prototype construction.
- **Engineering and product development:** Translates verified requirements into detailed designs, production code, unit verification, and component integration using conventional lifecycle approaches.
- **Customer evaluation and planning:** Assesses product releases against stakeholder expectations, tracks project status, secures budget commitments, and plans the succeeding spiral iteration.

4. Spiral Dimensions and Characteristics

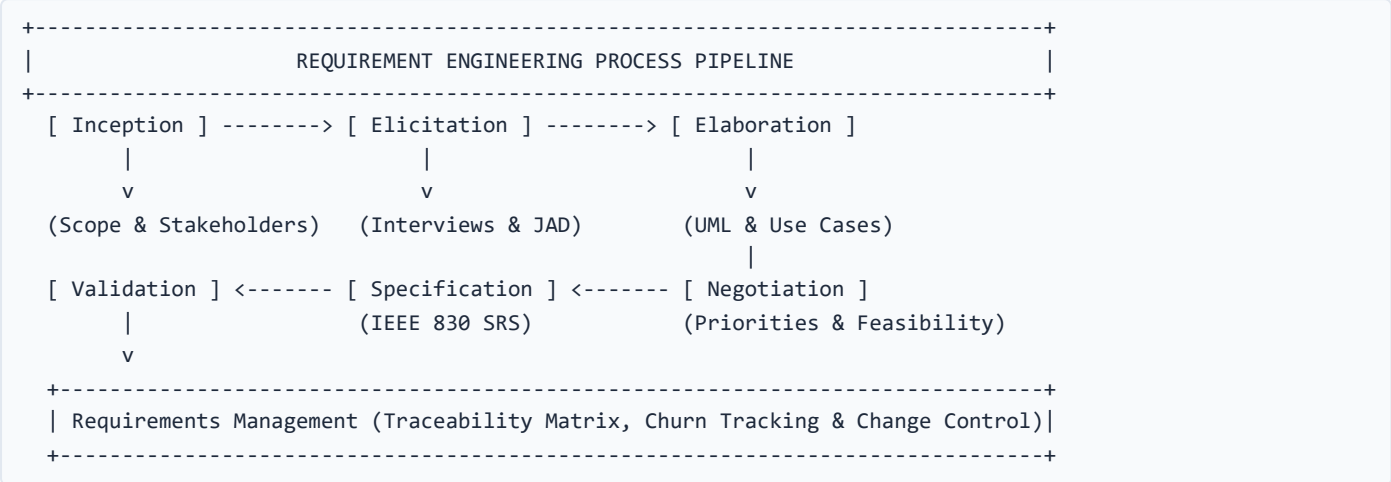
Spiral Dimension	Operational Meaning	Practical Function
Radial Distance (r)	Measures total cumulative cost	Tracks cumulative expenditure and human labor hours expended
Angular Displacement (theta)	Measures progress toward completion	Advances through objectives, risks, engineering, and planning
Risk Analysis Pivot	Mandatory checkpoint per spiral	Builds working prototypes to confirm feasibility before construction
Project Suitability	High-uncertainty systems	Ideal for large, mission-critical enterprise systems with volatile requirements

Q18. Explain Requirement Engineering Process Activities.

1. Core Concept

Requirement Engineering (RE) is the systematic, disciplined process of discovering, analyzing, modeling, documenting, validating, and managing requirements throughout the software product lifecycle. Governed by IEEE 610.12 and ISO/IEC/IEEE 29148 standards, RE establishes a formalized bridge between client business goals and software architectural designs, creating an unambiguous baseline for contracting, system construction, and acceptance testing.

2. Process Pipeline Diagram



3. Detailed Process Activities

- **Inception:** Establishes initial business understanding, defines overall project boundaries, identifies key stakeholders, and validates high-level feasibility.
- **Elicitation:** Gathers business needs and operational constraints from users using collaborative workshops, interviews, questionnaires, and rapid prototyping.
- **Elaboration:** Refines gathered requirements into formal domain models, activity workflows, data dictionaries, and structural use case models.
- **Negotiation:** Resolves conflicting stakeholder desires regarding cost, delivery schedules, and technical constraints through objective trade-off prioritization.
- **Specification:** Formalizes agreed functional, non-functional, and interface capabilities into an unambiguous Software Requirements Specification (SRS) document.
- **Validation:** Inspects the SRS for completeness, consistency, technical feasibility, and conformance to standards prior to architectural sign-off.
- **Management:** Tracks requirements churn, evaluates change requests via Change Control Boards (CCB), and maintains Requirements Traceability Matrices (RTM).

4. Activity Summary and Deliverables

RE Activity	Primary Technique / Mechanism	Key Deliverable / Output
Elicitation	Stakeholder interviews and JAD sessions	Raw user stories and feature wish-lists
Elaboration	Object-oriented and behavioral analysis	UML use cases and domain object models
Specification	IEEE 830 / ISO 29148 structural authoring	Baseline Software Requirements Specification (SRS)
Management	Bidirectional traceability tracking	Updated Requirements Traceability Matrix (RTM)

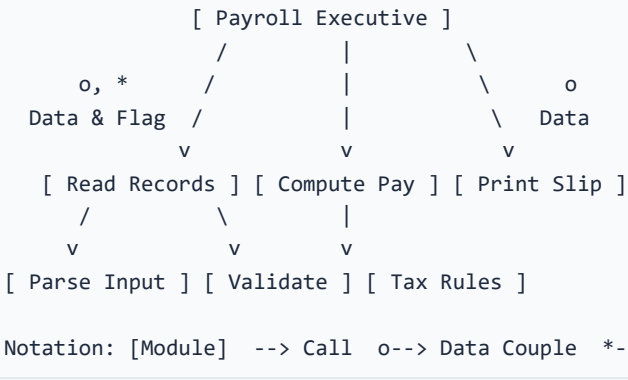
Q19. Explain structured design and the different structured design tools.

1. Core Concept

Structured Design is a traditional software engineering methodology based on top-down functional decomposition, transforming data flow representations into a well-defined modular software architecture. Formulated by Edward Yourdon

and Larry Constantine, it partitions complex problems into manageable, single-purpose software modules characterized by high internal cohesion and low inter-module coupling.

2. Architecture / Structure Chart Diagram



3. Core Structured Design Tools

- **Structure Charts:** Tree diagrams depicting module call hierarchies, parameter-passing mechanisms (data couples), and conditional decision flags (control couples) across software subsystems.
- **Data Flow Diagrams (DFD):** Graphical representations depicting the flow, storage, and transformation of data within a system without illustrating procedural sequence or hardware logic.
- **Data Dictionaries:** Centralized structured catalogs defining schemas, data types, relationships, and allowable constraints for all data elements flowing through system processes.
- **Decision Tables and Trees:** Tabular and branched matrices that document complex logical decision spaces and condition-action combinations, eliminating algorithmic ambiguity in business rules.
- **Structured English / Pseudocode:** Controlled subsets of natural language utilizing standardized programming keywords (IF-THEN-ELSE, REPEAT-UNTIL) to specify module internal logic.

4. Structured Design Tools Comparison

Design Tool	Structural Focus	Primary Notational Symbols
Structure Chart	Architectural module hierarchy	Module boxes, directed call arrows, data and control couples
Data Flow Diagram	Data flow and transformation	Process bubbles, entity squares, data stores, directed arrows
Decision Table	Complex logic and conditional rules	Condition stubs, action stubs, condition entries, action entries
Structured English	Detailed algorithmic procedural logic	Standardized programming verbs, indentation, sequential constructs

---

Q20. Describe the different strategies of testing conventional software.

1. Core Concept

Testing conventional software is a systematic, hierarchical roadmap that integrates test-case design techniques into a planned sequence of verification and validation activities. Organized as an expanding spiral or V-model hierarchy, testing begins at the microscopic source code level and progresses outward to validate integrated modules, full subsystem behaviors, and overall business system compliance.

2. Architecture / Testing Hierarchy Spiral

```
   +---------------------------------------------------------------+
   |                  SOFTWARE TESTING STRATEGY SPIRAL              |
   +---------------------------------------------------------------+
      / \        [ 4. System Testing ]     -> Recovery, Security, Stress
     /   \       [ 3. Validation Testing ] -> Alpha, Beta & Acceptance
    /     \      [ 2. Integration Testing] -> Top-Down & Bottom-Up Harness
   /_______\     [ 1. Unit Testing ]       -> Module Logic, White-Box Paths

   Focus: Code Level (Inward) -------------> Complete System (Outward)
```

3. Four Conventional Testing Strategies

- **Unit Testing:** Validates microscopic software components (classes, functions, procedures) in isolation, examining algorithmic execution paths, boundary values, and local data structures using white-box techniques.
- **Integration Testing:** Combines verified unit modules to uncover interface defects and communication faults, utilizing Top-Down (employing stub functions) or Bottom-Up (employing test driver programs) strategies.
- **Validation Testing:** Confirms that the integrated application satisfies all functional, behavioral, and performance requirements specified in the SRS through Alpha (in-house developer environment) and Beta (field user environment) tests.
- **System Testing:** Exercises the fully assembled software across targeted hardware, network, and operating system environments, evaluating critical non-functional parameters including recovery, penetration security, stress limits, and performance throughput.

4. Summary Comparison Table

Testing Strategy	Focus & Target Scope	Primary Testing Mechanism
Unit Testing	Individual code modules and methods	White-box logic path and boundary value coverage
Integration Testing	Inter-module interfaces and parameter passing	Incremental harnesses using test drivers and stubs
Validation Testing	Conformance to customer SRS requirements	Black-box functional validation, Alpha and Beta testing
System Testing	Complete end-to-end operational platform	Stress, recovery, security, and load benchmarking

---

Q21. Explain Halstead Software Science and its basic measures.

1. Core Concept

Introduced by Maurice H. Halstead in 1977, Halstead Software Science is an analytical measurement theory that evaluates the structural complexity, memory volume, and mental development effort of a software program. It treats computer programs as tokenized texts comprising distinct and repeated operators (procedural keywords and symbols) and operands (variables, constants, and data identifiers).

2. Primitive Measures and Derived Formulas

```
Primitive Token Counts:
  eta1 = Number of unique operators      eta2 = Number of unique operands
  N1   = Total occurrences of operators  N2   = Total occurrences of operands

Derived Software Science Formulas:
  Program Vocabulary:   eta = eta1 + eta2
  Program Length:       N   = N1 + N2
  Program Volume:       V   = N * log2(eta)  (measured in bits)
  Program Difficulty:   D   = (eta1 / 2) * (N2 / eta2)
  Development Effort:   E   = D * V  (elementary mental discriminations)
  Estimated Time:       T   = E / 18  (seconds, where Stroud number S = 18)
```

3. Core Halstead Metrics Explained

- **Program Vocabulary (eta):** Sum of distinct operators and operands, measuring vocabulary size required to comprehend the algorithm.
- **Program Length (N):** Sum of total operator and operand occurrences, quantifying program scale independent of line formatting.
- **Program Volume (V):** Minimum binary bits required to store the source algorithm in computer memory.
- **Program Difficulty (D):** Measures error-proneness and cognitive reading effort, increasing with operand reuse.
- **Mental Effort (E):** Measures elementary cognitive decisions required by a programmer to construct the code.

4. Concrete Numerical Calculation Example

Token Category	Elements Present	Count
Operators (eta1, N1)	int, (), {}, return, >, ?:, ,, ;	eta1 = 8, N1 = 10
Operands (eta2, N2)	max, a, b	eta2 = 3, N2 = 6
Vocabulary & Length	eta = 8 + 3 = 11 tokens	N = 10 + 6 = 16 tokens
Volume, Difficulty, Effort	V = 16 * log2(11) = 55.35 bits; D = (8/2)*(6/3) = 8	E = 8 * 55.35 = 442.8 discriminations

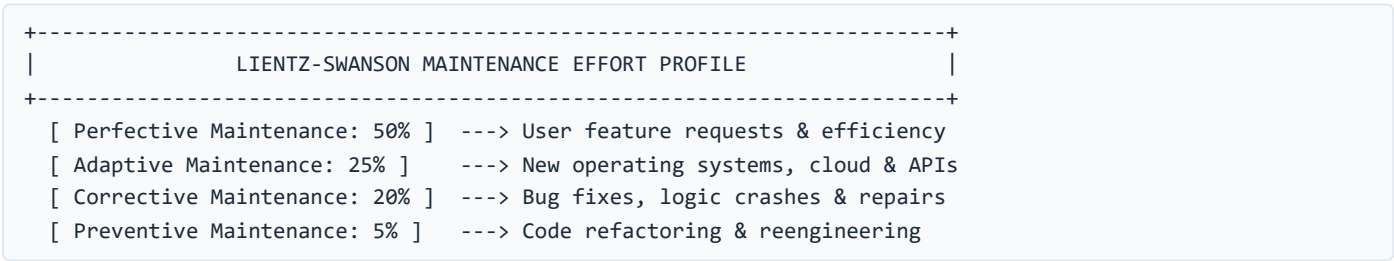
---

Q22. Explain the different types of software maintenance with suitable examples.

1. Core Concept

According to ISO/IEC 14764, software maintenance is the totality of engineering processes required to provide cost-effective modifications and operational support to a delivered software system throughout its post-delivery lifecycle. Maintenance is essential to preserve operational viability, adapt systems to evolving hardware and regulatory frameworks, fulfill new user demands, and mitigate structural decay.

2. Maintenance Effort Distribution Diagram



3. Four Core Maintenance Categories with Examples

- **Corrective Maintenance:** Reactive troubleshooting to repair latent logical defects, runtime crashes, and calculation errors identified in production (e.g., patching a null pointer exception in a credit card billing checkout routine).
- **Adaptive Maintenance:** Modifications required to keep software functional when underlying operational environments, hardware platforms, or external regulations change (e.g., updating database connectivity drivers for Linux kernel upgrades or adopting GST tax rules).
- **Perfective Maintenance:** Proactive enhancements driven by user requests to add new functional capabilities, improve UI responsiveness, and optimize system speed (e.g., implementing biometric login and dark-mode display options on a mobile banking application).
- **Preventive Maintenance:** Re-engineering, restructuring, and code hardening to eliminate latent vulnerabilities and prevent future failures before occurrence (e.g., converting legacy monolithic functions into decoupled microservices and updating deprecated libraries).

4. Summary Comparison Table

Maintenance Category	Trigger Nature	Effort Share	Concrete Industry Example
Corrective	Reactive	20%	Fixing buffer overflow or division-by-zero runtime error
Adaptive	Environmental	25%	Upgrading payment gateway to comply with TLS 1.3 protocol
Perfective	Proactive Enhancement	50%	Adding multi-currency checkout to an e-commerce website
Preventive	Quality Hardening	5%	Refactoring undocumented spaghetti code to reduce complexity

---

Q23. Explain the meaning, need, and objectives of software project management.

1. Core Concept

Software Project Management (SPM) is the discipline of planning, organizing, monitoring, and controlling human, financial, and technical resources to deliver high-quality software that meets agreed requirements. Governed by IEEE 1058 and the Project Management Body of Knowledge (PMBOK), SPM balances the Triple Constraints of Scope, Time, and Cost while ensuring rigorous quality control throughout the engineering lifecycle. It systematically coordinates human teams, engineering processes, and technological tools to minimize project failure rates and deliver reliable software within constraints.

2. Triple Constraints Triangle Diagram



3. Need for Software Project Management

- **Overcoming Software Crisis:** Prevents the chronic pitfalls of budget overruns, missed release schedules, unmaintainable code, and abandoned development projects.
- **Managing Intangibility:** Unlike civil construction, software progress cannot be tracked visually; SPM establishes measurable milestones, metrics, and deliverable baselines.
- **Coordinating Triad Dimensions:** Harmonizes the 3Ps (People, Product, and Process) to ensure team members execute optimal development methodologies against customer specifications.
- **Managing Project Risks:** Anticipates technological dependencies, requirements churn, staff turnover, and schedule bottlenecks before they trigger catastrophic delivery delays.
- **Optimizing Resource Allocation:** Distributes development tools, computing environments, and human talent efficiently across parallel lifecycle engineering activities.

4. Core Objectives and Governance Summary

Project Objective	Operational Focus	Key Control Tool / Metric
On-Time Delivery	Adherence to scheduled project deadlines	Gantt charts, Milestone tracking, Critical Path Method (CPM)
Cost Containment	Completing development within approved budget	COCOMO estimation, Earned Value Analysis (EVA)
Quality Conformance	Fulfilling all functional and ISO 25010 benchmarks	Defect Removal Efficiency (DRE), Formal Technical Reviews
Stakeholder Satisfaction	Aligning deliverables with client expectations	Scope baseline control, Change Control Board (CCB) reviews
Risk Mitigation	Proactive containment of schedule and cost risks	Continuous risk logging, contingency reserves, and mitigation plans

---

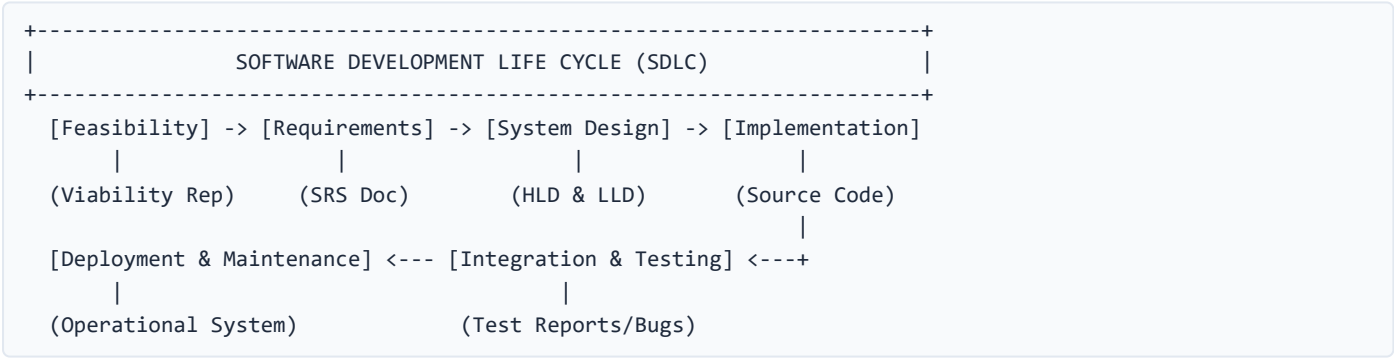
MODEL QUESTION PAPER – 2

Q17. Describe the different phases of the Software Development Life Cycle.

1. Core Concept

The Software Development Life Cycle (SDLC) is a structured, systematic framework defining the comprehensive series of stages executed to plan, design, build, test, deploy, and maintain high-quality software systems. According to IEEE standards, SDLC establishes clear technical milestones, deliverables, and quality gates across the entire system existence from initial feasibility assessment through final software decommissioning.

2. Architecture & Process Framework



3. Key Engineering Dimensions & Activities

- **Feasibility Study:** Evaluates technical, economic, legal, operational, and schedule viability (TELOS framework) to determine if project execution is commercially and technologically practical.
- **Requirements Analysis & Specification:** Gathers end-user business expectations, resolves inconsistencies, and establishes a formal Software Requirement Specification (SRS) baseline document.
- **System Design:** Translates SRS specifications into detailed technical blueprints, including high-level architecture (HLD), modular decomposition, database schemas, and low-level algorithms (LLD).
- **Implementation & Coding:** Software developers write, refactor, and compile executable source code using designated programming languages, adhering to formal coding conventions and unit-testing standards.
- **Integration & Testing:** Combines compiled modules and verifies functional, performance, security, and interface behavior against SRS requirements to isolate and remediate software defects.
- **Deployment & Maintenance:** Releases the verified software package into the client operational production environment, followed by ongoing corrective, adaptive, and perfective maintenance.

4. Technical Comparison & Deliverables Matrix

SDLC Phase	Primary Engineering Activity	Key Deliverable / Work Product
Feasibility Study	Viability & cost-benefit analysis	Feasibility Assessment Report
Requirements Analysis	User elicitation & functional modeling	Formal SRS Document (IEEE 830)
System Design	Architectural & database modeling	Design Document (HLD & LLD)
Implementation	Source code translation & compilation	Executable Modules & Unit Tests
Testing & QA	Verification, validation & defect fixing	Test Summary & Defect Logs
Deployment & Maint.	Production release & user support	Operational Release & Patches

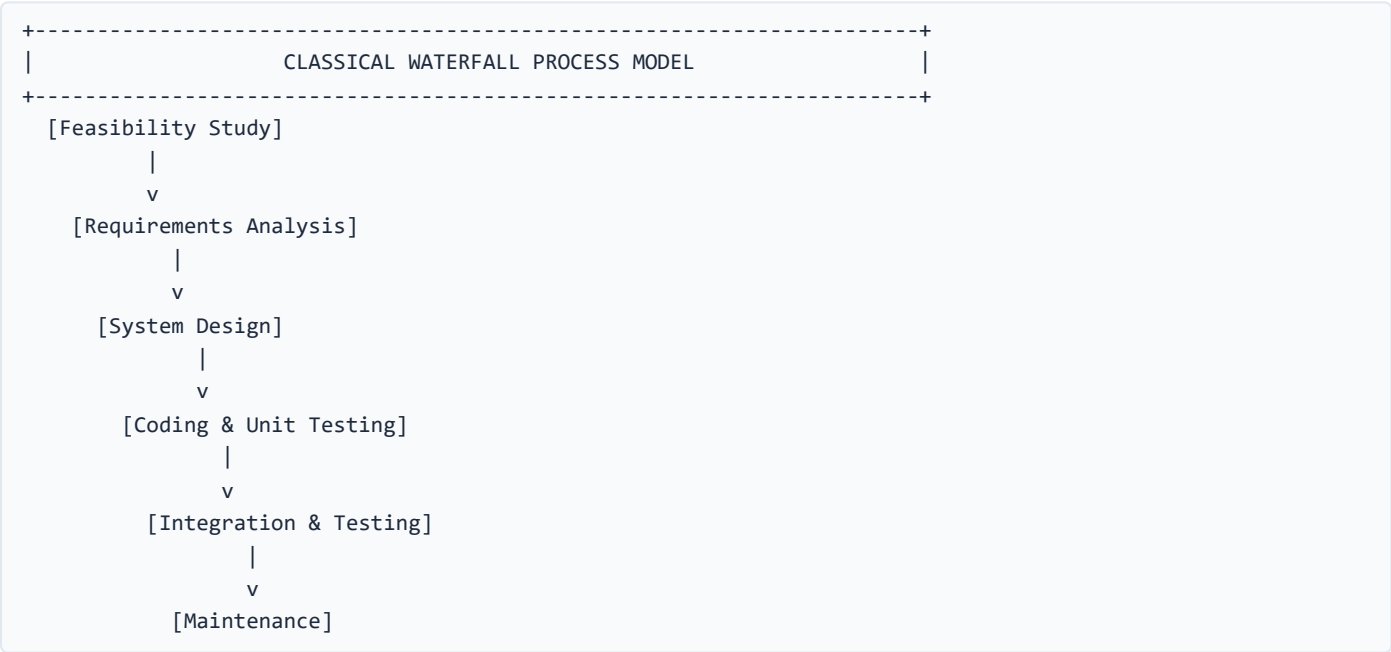
---

Q18. Explain the Classical Waterfall Model with its different phases.

1. Core Concept

The Classical Waterfall Model, introduced by Winston Royce in 1970, is the fundamental linear-sequential software process lifecycle model where development activities cascade downward through distinct phases in a rigid, non-overlapping sequence. In this theoretical model, a subsequent phase commences only after the preceding phase achieves 100% completion, verification, and formal documentation sign-off, featuring strictly zero feedback loops to prior stages.

2. Architecture & Process Framework



3. Key Engineering Dimensions & Activities

- **Feasibility Study:** Assesses financial cost-benefit, technological capability, and organizational feasibility to determine if development is practical and economically viable.
- **Requirements Analysis & Specification:** Gathers complete customer expectations and freezes them into a definitive Software Requirement Specification (SRS) before any engineering starts.
- **Design Phase:** Converts frozen requirements into high-level modular architecture, component data structures, database schemas, and procedural interfaces.
- **Coding & Unit Testing:** Programmers translate low-level design specifications into executable source code, testing each individual module in strict isolation.
- **Integration & System Testing:** Integrates standalone software modules and systematically evaluates complete system functionality against original SRS specifications to detect interface defects.
- **Maintenance Phase:** The longest lifecycle stage, focusing on repairing latent software bugs, adapting software to changing operational environments, and enhancing features.

4. Technical Comparison & Deliverables Matrix

Engineering Dimension	Classical Waterfall Strengths	Critical Engineering Limitations
Process Structure	Highly disciplined, sequential, and simple to understand	Inflexible; cannot accommodate changing user requirements
Milestone Tracking	Clear phase boundaries, deliverables, and sign-offs	Working software is delivered extremely late in lifecycle
Documentation	Comprehensive documentation at every project phase	High project risk; customer sees product only at delivery
Defect Management	Early detection during requirements prevents design flaws	Zero feedback loops; late defect discovery causes disaster

---

Q19. Explain Software Requirement Specification Document. Why we need SRS Document?

1. Core Concept

A Software Requirement Specification (SRS) is a formal, comprehensive document conforming to the IEEE 830 standard that explicitly defines the functional capabilities, performance criteria, constraints, and operational behavior of a target software system. It specifies completely what the software system must do without dictating how it should be designed or implemented, establishing the definitive legal agreement between stakeholders and developers.

2. Architecture & Process Framework

```
+-------------------------------------------------------------------------+
|                 IEEE 830 STANDARD SRS DOCUMENT STRUCTURE                |
+-------------------------------------------------------------------------+
  Section 1: Introduction
    |--> 1.1 Purpose  |  1.2 Scope  |  1.3 Definitions  |  1.4 References
  Section 2: Overall Description
    |--> 2.1 Product Perspective  |  2.2 User Characteristics  |  2.3 Constraints
  Section 3: Specific Requirements
    |--> 3.1 External Interfaces  |  3.2 Functional  |  3.3 Performance  |  3.4 Security
  Section 4: Appendices & Supporting Information
```

3. Key Engineering Dimensions & Activities

- **Contractual Baseline Agreement:** Formalizes shared understanding between customers and development teams, serving as a legally binding foundation that prevents scope creep and contractual disputes.
- **Input for Architectural Design:** Provides system architects and database designers with explicit functional specifications, data models, and interface constraints necessary to structure system architecture.
- **Benchmark for Verification & Validation:** Serves as the primary validation reference against which quality assurance teams design test cases, acceptance criteria, and system verification suites.
- **Accurate Project Estimation:** Furnishes project managers with granular requirement metrics to estimate project effort, financial budget, staffing requirements, and delivery milestones accurately.
- **Knowledge Transfer & Maintenance:** Acts as authoritative operational documentation enabling future maintenance engineers to understand core system constraints, business logic, and operational interfaces.

4. Technical Comparison & Deliverables Matrix

IEEE 830 Section	Document Component Focus	Primary Information Delivered
1. Introduction	Scope, Purpose & Vocabulary	Defines system boundaries, objectives, and terms
2. Overall Description	Context & Operational Constraints	Hardware platforms, user profiles, design assumptions
3. Specific Requirements	Granular Functional & Non-Functional	Inputs, outputs, throughput, security, UI interfaces
4. Appendices	Supporting Data & Schemas	Data flow models, acronym glossaries, sample forms

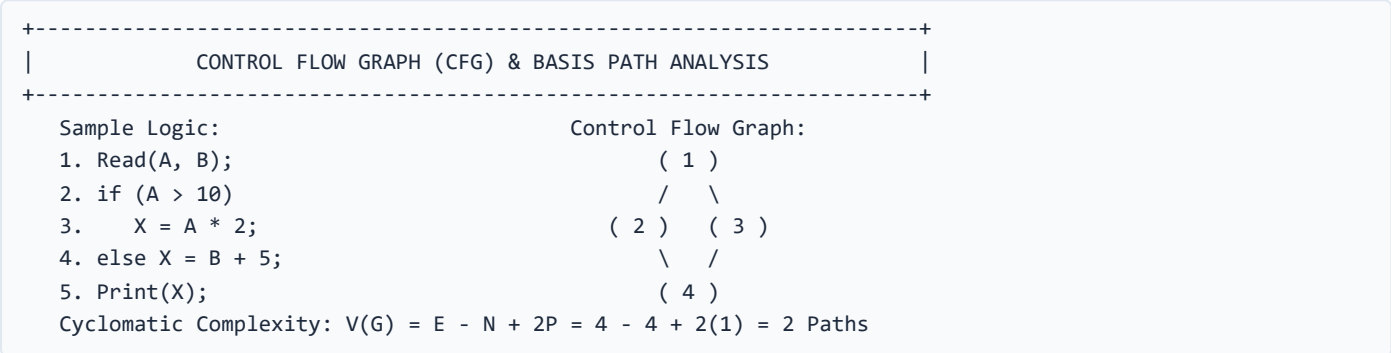
---

Q20. Explain white-box testing and its techniques in detail.

1. Core Concept

White-box testing (structural or glass-box testing) is a software verification method where internal code structure, control paths, logic conditions, and data flows are systematically inspected. It verifies that internal program statements execute as specified, eliminating unreached dead code and detecting logic path anomalies.

2. Architecture & Process Framework



3. Key Engineering Dimensions & Activities

- **Statement Coverage:** Executes every executable code statement at least once: Statement Coverage = (Executed Statements / Total Statements) * 100%.
- **Branch / Decision Coverage:** Traverses every True and False branch from each decision point: Branch Coverage = (Traversed Branches / Total Branches) * 100%.
- **Condition Coverage:** Evaluates each atomic boolean condition in compound decisions independently to True and False states.
- **Basis Path Testing (McCabe):** Uses Cyclomatic Complexity V(G) = E - N + 2P to compute the number of linearly independent paths, ensuring full execution coverage.

4. Technical Comparison & Deliverables Matrix

White-Box Technique	Coverage Metric Formula	Rigor Level	Primary Defect Focus
Statement Coverage	Executed Statements / Total Statements	Basic	Unreachable dead code statements
Branch Coverage	Traversed Branches / Total Decisions * 2	Moderate	Untested conditional outcomes
Condition Coverage	Evaluated Conditions / Total Conditions	High	Faulty boolean operator logic
Basis Path Testing	V(G) = Edges - Nodes + 2(Components)	Comprehensive	Complex logic interactions & loops

---

Q21. Explain metrics for testing.

1. Core Concept

Software testing metrics are quantitative measurements systematically gathered during test engineering to evaluate the progress, quality, coverage, defect severity, and overall effectiveness of software testing activities. According to software

engineering standards, these metrics provide project managers and quality assurance teams with objective, data-driven indicators to track test execution, identify high-risk components, and authorize production release.

2. Architecture & Process Framework

```
+---------------------------------------------------------------------+
|                CLASSIFICATION OF SOFTWARE TESTING METRICS            |
+---------------------------------------------------------------------+
                      /                              [Test Process Metrics]              [Test Product Met
                    |                                |
    * Test Case Execution Rate (TCER)      * Defect Density (DD = Def/KLOC)
    * Defect Removal Efficiency (DRE)      * Defect Severity Distribution
    * Test Turnaround & Fix Time           * Cyclomatic Complexity & Risk
    * Execution Pass / Fail Rate           * Code Coverage (Stmt/Branch %)
```

3. Key Engineering Dimensions & Activities

- **Defect Density (DD):** Quantifies discovered defects normalized against software size: DD = Total Defects / Size (KLOC), measuring relative module defect proneness.
- **Defect Removal Efficiency (DRE):** Measures testing filter effectiveness before release: DRE = [E / (E + D)] * 100%, where E is pre-release bugs and D is post-release customer defects.
- **Test Case Execution Rate (TCER):** Tracks testing progress and schedule velocity: TCER = (Executed Tests / Planned Tests) * 100%.
- **Test Pass / Fail Rate:** Assesses build stability and release readiness: Pass Rate = (Passed Tests / Executed Tests) * 100%.
- **Cyclomatic Complexity Correlation:** Relates McCabe metric V(G) to bug likelihood; modules with V(G) > 10 indicate high structural risk requiring focused re-testing.

4. Technical Comparison & Deliverables Matrix

Testing Metric Name	Mathematical Computation Formula	Key Engineering Objective	Industry Benchmark / Target
Defect Density	Total Defects / KLOC	Quantify code quality and defect rate	< 1.0 Defect / KLOC
Defect Removal Eff.	[E / (E + D)] * 100%	Measure quality filter effectiveness	>= 90% DRE before release
Test Execution Rate	(Executed Tests / Planned) * 100%	Track test execution progress against plan	100% planned tests executed
Test Pass Rate	(Passed Tests / Executed) * 100%	Determine build stability for deployment	>= 98% with zero P1 defects

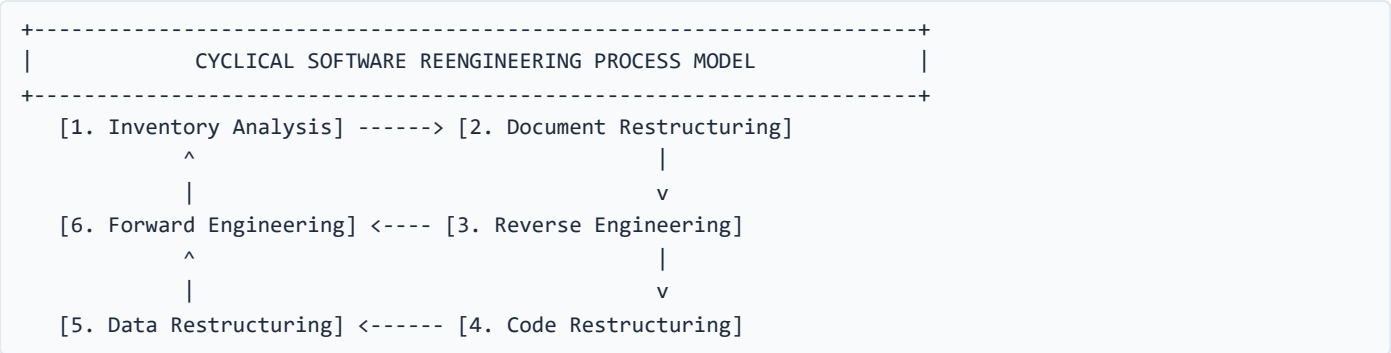
---

Q22. Explain the Software Reengineering Process Model with its different activities.

1. Core Concept

Software Reengineering is an iterative, disciplined software modernization process that re-examines, reconstructs, and modifies existing legacy systems into higher-quality, maintainable, and scalable software architectures without altering core business capabilities. It addresses legacy system obsolescence, preserving past capital investments and critical institutional business logic while dramatically cutting long-term maintenance overhead compared to high-risk greenfield redevelopment.

2. Architecture & Process Framework



3. Key Engineering Dimensions & Activities

- **Inventory Analysis:** Catalogs and assesses all active legacy software assets, evaluating program age, defect logs, maintenance expenses, and business value to prioritize candidate modules for reengineering.
- **Document Restructuring:** Updates, generates, and standardizes outdated or missing technical manuals, architectural designs, interface specifications, and user documentation to match current system reality.
- **Reverse Engineering:** Analyzes compiled modules and source code to extract high-level architectural models, database schemas, procedural control flows, and underlying business rules.
- **Code Restructuring:** Restructures and optimizes spaghetti source code by removing redundant logic, restructuring dead branches, and improving modularity without changing external program behavior.
- **Data Restructuring:** Modernizes legacy data stores by cleaning, normalizing, and migrating unindexed flat files or legacy hierarchical databases into modern relational or distributed schemas.
- **Forward Engineering:** Applies modern software engineering principles, component frameworks, and modern technologies to the recovered specifications, delivering an updated, maintainable, and scalable target application.

4. Technical Comparison & Deliverables Matrix

Evaluation Dimension	Software Reengineering	Greenfield (Fresh) Development
Business Logic Risk	Very Low; preserves verified enterprise rules	High; risk of omitting subtle domain rules
Development Cost	Moderate; leverages existing assets and testing	High; requires full-scale ground-up engineering
Time to Deployment	Rapid; modernized in phased module iterations	Protracted; long delay before functional release
User Retraining	Minimal; core workflows and business concepts preserved	Significant; new paradigms require complete retraining

---

Q23. Explain project scheduling and describe the major steps involved in preparing a project schedule.

1. Core Concept

Project scheduling is the essential project management discipline of organizing, sequencing, and allocating project activities, required resources, and technical deliverables across a calendar timeline. It establishes definitive delivery baselines, maps task dependencies, and designates intermediate project milestones, enabling engineering managers to track team progress, control budget expenditures, and ensure on-time delivery within scope constraints.

2. Architecture & Process Framework

```
+-------------------------------------------------------------------------+
|              PROJECT SCHEDULING FORMULATION PIPELINE                    |
+-------------------------------------------------------------------------+
  [Work Breakdown Structure] ---> [Activity Sequencing & Dependencies]
                                                 |
                                                 v
  [Critical Path Analysis (CPM)] <--- [Effort & Duration Estimation]
                |
                v
  [Resource Allocation & Leveling] -> [Baseline Schedule & Gantt Chart]
```

3. Key Engineering Dimensions & Activities

- **Work Breakdown Structure (WBS):** Decomposes total project scope hierarchically into manageable, discrete work packages and assignable task units, ensuring no vital activity is overlooked.
- **Activity Sequencing & Dependency Identification:** Defines chronological relationships between activities using Precedence Diagramming Methods (Finish-to-Start, Start-to-Start, Finish-to-Finish), determining logical order.
- **Effort & Duration Estimation:** Calculates required person-months and calendar days for each task using estimation models such as COCOMO II, Function Points, or Three-Point (PERT) estimates.
- **Critical Path Method (CPM) Analysis:** Calculates early/late starts and finishes across the network to identify the Critical Path (longest sequence with zero float), determining minimum project duration.
- **Resource Allocation & Leveling:** Assigns qualified human personnel, software licenses, and hardware assets to scheduled activities, smoothing resource utilization to prevent team burnout.
- **Baseline Schedule Formulation & Tracking:** Creates authoritative Gantt charts and milestone checkpoints against which actual ongoing project velocity and schedule variances are monitored.

4. Technical Comparison & Deliverables Matrix

Scheduling Stage / Technique	Primary Input Information	Core Output Deliverable	Management Utility / Purpose
Work Breakdown Structure	Project Scope & SRS	Hierarchical WBS Tree	Defines all discrete work packages
Activity Sequencing	WBS Work Packages	Network Dependency Diagram	Determines prerequisite work flow
Critical Path Analysis	Durations & Dependencies	Critical Path & Task Float	Identifies bottleneck tasks with zero slack
Gantt Chart Formulation	CPM Network & Resources	Baseline Project Schedule	Visualizes calendar deadlines & milestones

---

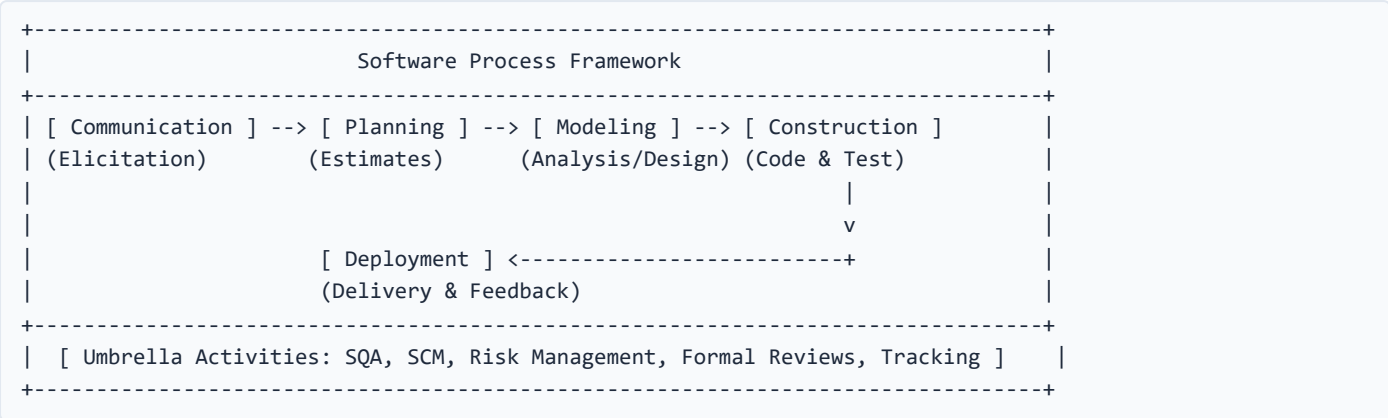
MODEL QUESTION PAPER – 3

Q17. Explain the software process framework and its major framework activities.

1. Core Concept

Definition: According to Roger Pressman, a software process framework establishes the organizational foundation for an entire software engineering life cycle by identifying a small set of generic framework activities that apply to all software projects regardless of their scale, technical complexity, or application domain.

2. Framework Architecture Workflow



3. Five Generic Framework Activities

- **Communication:** Involves heavy customer collaboration to understand business goals, define project scope, and elicit initial requirements.
- **Planning:** Formulates the software project plan by defining technical tasks, risk factors, required resources, cost estimates, and delivery schedules.
- **Modeling:** Creates analytical models (data flow, use cases) and architectural designs to conceptualize how the system will satisfy requirements.
- **Construction:** Combines programmatic code generation in chosen programming languages with rigorous unit and integration testing.
- **Deployment:** Delivers completed operational software increments to end users for evaluation, feedback, and operational maintenance.

4. Framework vs Umbrella Activities Summary Table

Comparison Dimension	Generic Framework Activities	Umbrella Activities
Execution Nature	Sequential or iterative developmental stages	Continuous across the entire lifecycle
Primary Objective	Direct engineering and construction of software	Management control, quality assurance, and tracking
Core Examples	Communication, Modeling, Construction	SQA, SCM, Risk Management, FTR

Q18. Explain the Prototyping Model with its different phases or stages.

1. Core Concept

Definition: The Prototyping Model is an evolutionary software development process model designed for projects where customer requirements are vague, unstable, or poorly understood. Developers rapidly build an interactive mock-up (prototype) to elicit user feedback, progressively refining system specifications prior to full production coding.

2. Prototype Lifecycle Loop Diagram



3. Six Sequential Phases of the Prototyping Model

- **Requirements Gathering:** Developers and clients meet to identify broad system objectives and outline known operational requirements.
- **Quick Design:** Focuses rapidly on visible interface screens, menu flows, input dialogues, and report outputs without internal algorithmic details.
- **Build Prototype:** Developers assemble a functional mock-up rapidly using high-level GUI builders, component libraries, or simulated data.
- **Customer Evaluation:** End users test the working prototype to evaluate usability, uncover missing requirements, and suggest functional modifications.
- **Prototype Refinement:** Feedback is iteratively incorporated to refine the design and rebuild prototype iterations until customer consensus is reached.
- **Final Product Engineering:** Once requirements freeze, the system is fully engineered, coded, tested, and optimized for production deployment.

4. Throwaway vs Evolutionary Prototyping

Feature	Throwaway Prototyping	Evolutionary Prototyping
Prototype Destiny	Discarded once requirements are clarified	Gradually refactored into the final production product
Primary Objective	Requirement validation and risk exploration	Rapid delivery of core functioning increments
Code Rigor	Quick, unoptimized mock-up code	Robust, documented, production-grade architecture

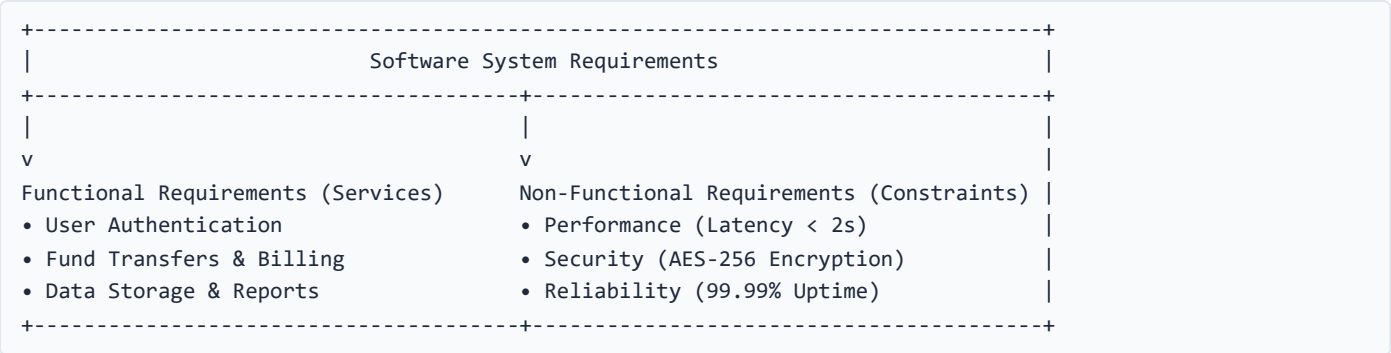
---

Q19. Describe the difference between functional and non-functional requirements with examples.

1. Core Concept

Definition: Software requirements define the capabilities and constraints of a proposed software system. Functional requirements specify **what the system must do** (explicit services and behaviors), whereas Non-functional requirements (NFRs or quality attributes) specify **how well the system must perform** (operational constraints, performance benchmarks, and security rules).

2. Requirements Architecture Diagram



3. Comprehensive Comparison Table

Comparison Dimension	Functional Requirements	Non-Functional Requirements (NFRs)
Primary Focus	Specific tasks and behavioral services	Quality attributes and operational constraints
User Visibility	Directly visible and interactive for users	Pervasive system qualities experienced indirectly
Verification Method	Black-box functional and unit test cases	Stress, load, penetration, and benchmark tests
Measurement Metric	Boolean pass/fail (feature works or doesn't)	Quantitative values (seconds, uptime %, throughput)
Standard Hierarchy	Use Cases, User Stories, Functional specs	ISO/IEC 25010 Quality Model (Security, Performance)
Architectural Impact	Dictates module logic and database tables	Dictates hardware, clustering, caching, and security layers
Online Banking Example	User can transfer funds via NEFT/UPI	Fund transfer completes in under 1.5 seconds
Hospital System Example	Doctor can view and update patient charts	Patient medical records encrypted using AES-256

4. Engineering Significance

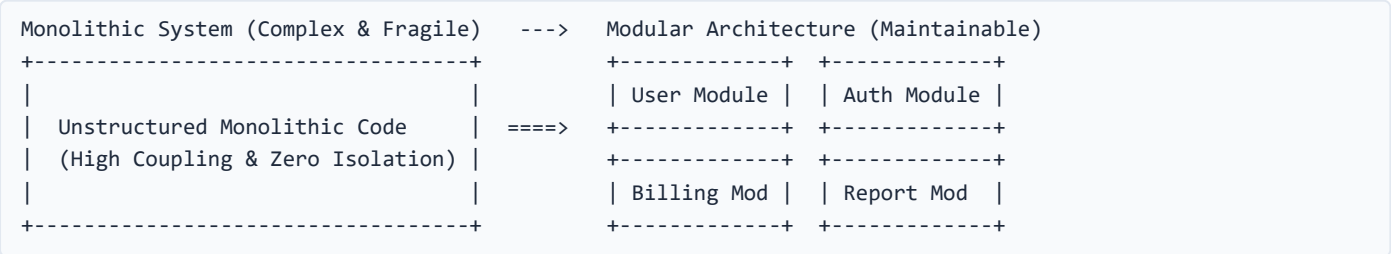
Functional requirements ensure that software satisfies business logic, while non-functional requirements ensure that the system remains scalable, secure, and resilient under production workloads.

Q20. Explain modular design and the need for modular design.

1. Core Concept

Definition: Modular design is the software architectural practice of decomposing a large, complex monolithic software system into distinct, self-contained, and manageable units called modules. Each module encapsulates a specific sub-function, adheres to information hiding, and interacts with other modules through clean, standardized interfaces.

2. Modularization Architecture Diagram



3. Five Key Reasons Why Modular Design is Essential

- **Cognitive Manageability:** Breaks down overwhelming system complexity into intellectually manageable pieces that individual programmers can comprehend.

- **Concurrent Team Development:** Different engineering teams can develop, test, and debug separate modules in parallel without code collisions.
- **Localized Fault Isolation:** Defects occurring within one module are encapsulated, preventing catastrophic system-wide ripple failures.
- **Component Reusability:** Cohesive, independent modules (e.g., encryption utilities, payment gateways) can be reused across different projects.
- **Maintenance and Evolution:** Individual modules can be refactored, upgraded, or replaced without rewriting the entire application.

4. Effective Modular Design Criteria

Design Principle	Engineering Definition	Architectural Impact
Functional Independence	Module executes one single, distinct sub-function	High cohesion and low coupling
Information Hiding	Internal data structures and algorithms hidden from callers	Prevents unauthorized external state tampering
Modular Decomposability	System can be systematically broken into sub-problems	Enables clean top-down architectural decomposition

---

Q21. Explain Software Maturity Index with formula and example.

1. Core Concept

Definition: The Software Maturity Index (SMI) is an IEEE standard metric (IEEE Standard 982.1) used to assess the architectural stability, quality maturity, and release readiness of a software product based on the volume of changes made between sequential releases.

2. Mathematical Formula and Parameters

```
SMI = [ M_t - (F_a + F_c + F_d) ] / M_t
```

- `M_t:` Total number of modules in the current software release.
- `F_a:` Number of modules newly added in the current release.
- `F_c:` Number of existing modules changed or modified in the current release.
- `F_d:` Number of modules deleted from the previous release.
- **Stabilization Target:** As software matures, the volume of changes approaches zero, causing the SMI value to approach **1.00**. An SMI close to 1.0 confirms production stability.

3. Step-by-Step Multi-Release Numerical Example

Consider an enterprise banking system tracked across three development releases:

- **Release 1 (Initial Baseline):** Total modules `M_t = 100`.
- **Release 2:** `M_t = 110`, with `F_a = 15`, `F_c = 20`, and `F_d = 5`.
  Total Changes = `15 + 20 + 5 = 40`
  `SMI_2 = (110 - 40) / 110 = 70 / 110 = 0.636`
- **Release 3:** `M_t = 115`, with `F_a = 5`, `F_c = 6`, and `F_d = 1`.
  Total Changes = `5 + 6 + 1 = 12`
  `SMI_3 = (115 - 12) / 115 = 103 / 115 = 0.896`

4. Release Stability Tracking Table

Software Release	Total Modules (M_t)	Total Changes (F_a + F_c + F_d)	Calculated SMI	Release Maturity Assessment
Release 2	110	40	**0.636**	High volatility; extensive regression testing required
Release 3	115	12	**0.896**	Rapidly stabilizing toward baseline maturity
Target Goal	—	Approaches 0	→ **1.00**	Mature, stable product ready for deployment

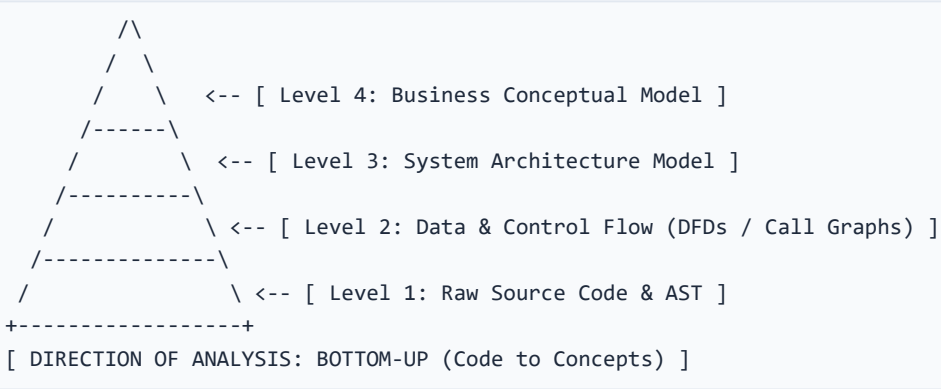
---

Q22. Explain the reverse engineering process in detail.

1. Core Concept

Definition: As defined by Chikofsky and Cross, reverse engineering is the systematic process of analyzing an existing target software system to identify its structural components and interrelationships, and to reconstruct representations of the system at higher levels of abstraction without modifying the underlying source code.

2. Reverse Engineering Abstraction Pyramid



3. Four Core Phases of Reverse Engineering

- **Code Parsing & Data Extraction:** Syntactic analyzers and lexers parse raw source code and binaries into Abstract Syntax Trees (AST) and symbol tables.
- **Information Abstraction:** Strips low-level language idioms to reconstruct control-flow graphs, call hierarchies, and database entity schemas.
- **Concept Formulation:** Synthesizes extracted models to identify architectural design patterns, component boundaries, and core business rules.
- **Architectural Redocumentation:** Generates clean, updated structural documentation, UML class models, and interface specifications for maintainers.

4. Reverse vs Forward Engineering Comparison Table

Comparison Parameter	Reverse Engineering	Forward Engineering
Development Direction	Bottom-up (Source Code → Design → Requirements)	Top-down (Requirements → Design → Source Code)
Starting Point	Existing legacy source code and binaries	Customer business needs and problem statements
Primary Objective	Understand and document existing architecture	Build a brand-new working software system
Tool Support	Decompilers, code parsers, profilers	IDEs, compilers, CASE modeling tools

---

Q23. Explain the software risk management process.

1. Core Concept

Definition: Software risk management is a formal, proactive management discipline that identifies, assesses, monitors, and mitigates prospective uncertainties that could threaten project delivery schedules, budgets, software quality, or operational success. It is operationalized through the **RMMM (Risk Mitigation, Monitoring, and Management)** plan.

2. Risk Management Lifecycle Loop



3. Five Key Stages of the Risk Management Process

- **Risk Identification:** Catalogs potential project, technical, and business threats using checklists, team audits, and historical defect logs.
- **Risk Analysis & Projection:** Evaluates each risk across two quantitative dimensions: **Probability (P)** and **Impact (I)** to compute Risk Exposure: `RE = P × I`.
- **Risk Prioritization:** Ranks risks on a severity matrix to focus managerial effort on high-probability, high-impact hazards.
- **Risk Mitigation Planning:** Formulates specific proactive strategies to reduce risk likelihood, alongside fallback contingency plans if risks materialize.
- **Risk Monitoring:** Continuously tracks risk indicator metrics across sprint milestones to adjust mitigation tactics dynamically.

4. Sample Risk Information Table (RMMM Matrix)

Identified Risk	Category	Probability	Impact	Mitigation & Contingency Plan
Staff Turnover	Project Risk	High (70%)	Critical	Cross-training, paired programming, and modular documentation.
Scope Creep	Business Risk	Medium (50%)	High	Formal Change Control Board (CCB) and baseline requirement freezes.
Unproven Cloud API	Technical Risk	Low (20%)	Moderate	Early technical spikes, benchmarking, and alternative vendor fallbacks.

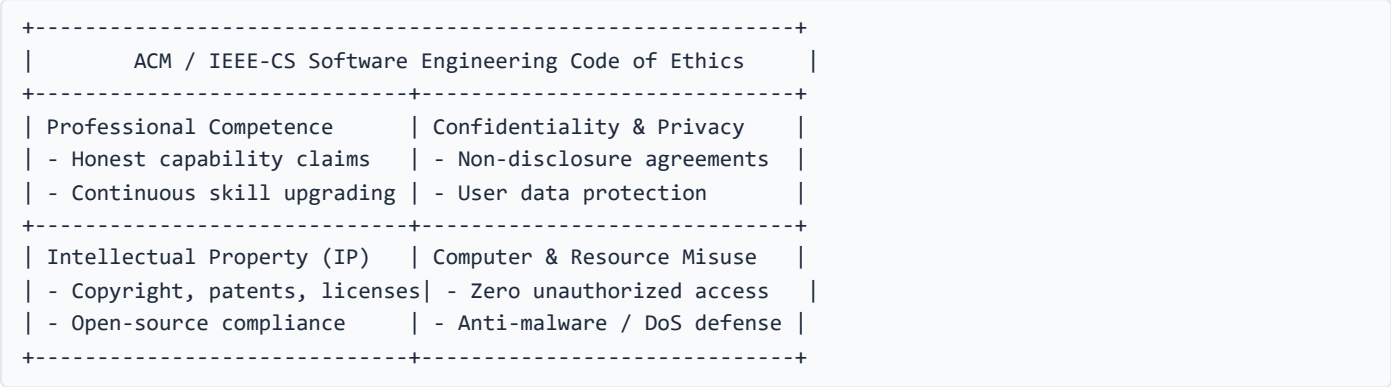
MODEL QUESTION PAPER – 4

Q17. Explain software engineering ethics and describe confidentiality, competence, intellectual property rights, and computer misuse.

1. Core Concept

Software engineering ethics encompasses the moral principles and professional responsibilities governing practitioners to ensure software systems safeguard public welfare, privacy, and safety. Jointly formulated by the ACM and IEEE Computer Society (ACM/IEEE-CS), the Code of Ethics establishes eight fundamental principles (Public, Client/Employer, Product, Judgment, Management, Profession, Colleagues, and Self) committing engineers to the highest ethical and professional integrity.

2. Ethical Practice Framework



3. Detailed Ethical Dimensions

- **Confidentiality:** Engineers must scrupulously protect sensitive employer and client information, respecting non-disclosure agreements and data privacy mandates regardless of formal contractual boundaries.
- **Competence:** Practitioners must not misrepresent their technical expertise, accept assignments beyond their competence, or approve safety-critical systems without verified qualification.
- **Intellectual Property Rights:** Professionals must respect copyrights, patents, trade secrets, and open-source licenses, refusing unauthorized asset cloning or proprietary source-code piracy.
- **Computer Misuse:** Engineers must never leverage privileged system access to compromise infrastructure, plant malicious backdoors, access private accounts, or launch denial-of-service attacks.

4. Ethical Obligations and Violation Consequences

Ethical Area	Core Professional Mandate	Potential Violation Scenario	Real-World Consequence
Confidentiality	Protect proprietary trade secrets and user PII	Exposing client source code or customer data	Severe litigation, loss of professional trust
Competence	Accept tasks strictly within verified skill limits	Falsifying readiness on critical safety systems	Catastrophic system failure and legal liability
Intellectual Property	Respect software patents, trademarks, and OSS terms	Embedding GPL-licensed libraries into closed SaaS	Copyright infringement and asset forfeiture
Computer Misuse	Protect computing environments from unauthorized access	Deploying unauthorized data-harvesting logic	Criminal prosecution and credential revocation

---

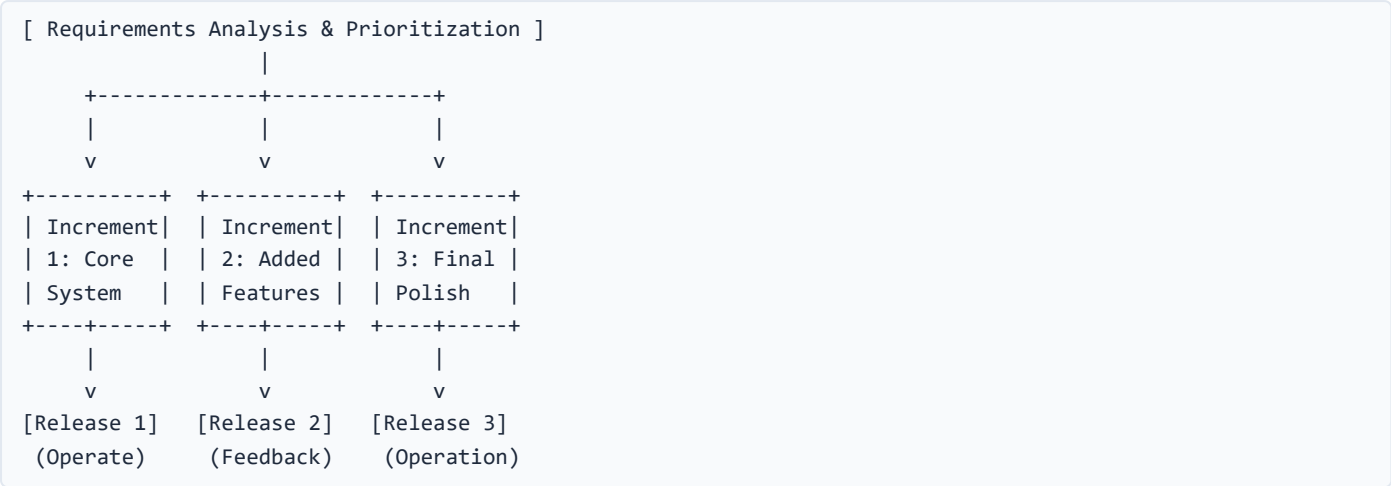
Q18. Explain the Incremental Delivery Model and its development process.

1. Core Concept

The Incremental Delivery Model is an iterative software process where system requirements are prioritized and delivered to customers in successive, functioning releases termed increments. Rather than postponing delivery until complete system realization, Increment 1 delivers the core baseline product (must-have capabilities), while subsequent increments

systematically add enhancements, supplementary services, and user-driven refinements. This approach is widely adopted in commercial products to enable early market entry and reduce delivery risk.

2. Incremental Delivery Architecture



3. Development Process Activities

- **Requirements Prioritization:** System requirements are analyzed and categorized into prioritized tiers, placing architectural core services into the earliest delivery increment.
- **Iterative Design and Coding:** Each increment undergoes a complete mini-lifecycle, including detailed specification, component design, implementation, and rigorous unit testing.
- **Continuous Integration and Regression:** Completed increments are integrated directly into the deployed operational baseline, using regression testing to ensure existing capabilities remain unbroken.
- **Customer Feedback Incorporation:** Deployed increments undergo actual operational use, generating empirical user feedback that steers and refines future increment specifications.

4. Model Comparison and Trade-offs

Evaluation Dimension	Incremental Delivery Model	Traditional Waterfall Model
Value Realization	Early operational delivery of core capabilities to end users	Zero business value delivered until entire final product cutover
Risk Mitigation	Early exposure of architecture reduces critical late-stage failures	High risk; catastrophic integration defects discovered late
Change Accommodation	Customer feedback seamlessly integrated into subsequent increments	Requirements locked early; alterations cause substantial rework
Stakeholder Engagement	Continuous customer evaluation fosters active collaboration	Limited to requirements sign-off and final acceptance testing

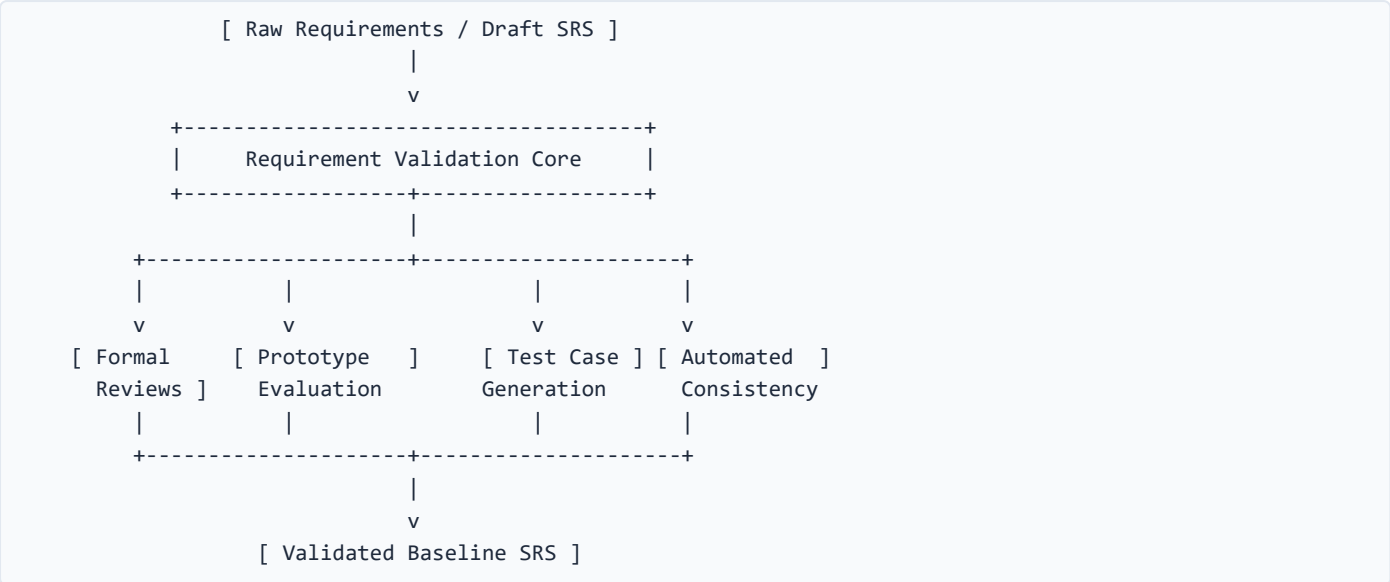
---

Q19. Explain the Activities of Requirement Validation.

1. Core Concept

Requirement validation is the formal quality assurance process that verifies whether the Software Requirements Specification (SRS) accurately defines the system that stakeholders truly expect and need. While verification evaluates specification conformance ("Are we building the product right?"), validation asks "Are we building the right product?", resolving ambiguities, omissions, and conflicts before design begins. Industry benchmarks demonstrate that discovering requirement errors during validation costs up to 100 times less than fixing them post-deployment.

2. Requirement Validation Pipeline



3. Essential Validation Activities

- **Formal Requirements Reviews:** Multidisciplinary teams (analysts, clients, architects) systematically inspect the SRS against checklists to detect ambiguities, inconsistencies, and omissions.
- **Prototyping Evaluation:** Interactive mock-ups and wireframes are presented to end-users to validate workflow expectations, resolve fuzzy needs, and clarify requirements.
- **Test-Case Generation:** Designing acceptance test cases directly from functional specifications exposes untestable, vague, or non-deterministic requirements early in the project.
- **Automated Consistency Analysis:** Computer-Aided Software Engineering tools parse formal specification models to identify circular dependencies, conflicting rules, and undefined terms.
- **Traceability Matrix Verification:** Requirements Traceability Matrices ensure each requirement maps directly to business objectives without orphan or extraneous features.

4. Validation Quality Checklist

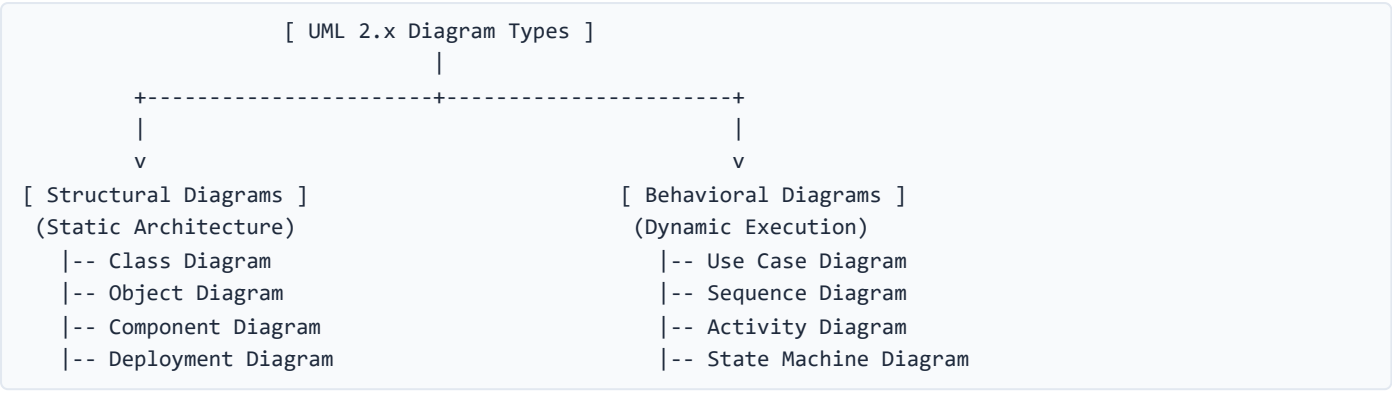
Quality Criterion	Validation Objective	Typical Defect Detected
Validity / Correctness	Confirms requirements align with real operational workflows	Misinterpreted user expectations and domain logic
Completeness	Ensures all operational states, inputs, and exceptions are specified	Missing boundary cases, exceptions, and security limits
Consistency	Verifies no internal contradictions exist across functional statements	Clashing functional rules and conflicting data definitions
Verifiability	Confirms every requirement can be objectively proven via testing	Vague subjective phrases such as "user-friendly" or "fast"

Q20. Explain UML, its features, and the types of UML diagrams.

1. Core Concept

The Unified Modeling Language (UML) is the industry-standard, object-oriented visual modeling language governed by the Object Management Group (OMG) for specifying, visualizing, constructing, and documenting software artifacts. Created by Grady Booch, James Rumbaugh, and Ivar Jacobson, UML 2.x defines 14 diagram types categorized into structural and behavioral views, enabling architects and developers to represent complex software blueprints independently of programming languages.

2. UML Diagram Classification



3. Core Features of UML

- **Standardized Visual Notation:** Establishes unambiguous graphical primitives (nodes, connectors, stereotypes) recognized universally across modern software development.
- **Language Independence:** Provides platform-agnostic models readily translated into diverse target programming languages like Java, C++, Python, or C#.
- **Dual-Perspective Modeling:** Accurately represents both static structural layouts (classes, physical nodes) and runtime dynamic behaviors (messages, state shifts).
- **Extensibility Profiles:** Accommodates domain-specific customizations through stereotypes, tagged values, and formal constraints via Object Constraint Language.
- **Tool Interoperability:** Standardized metamodels enable seamless model exchange across Computer-Aided Software Engineering environments.

4. Comparison of Primary Structural vs. Behavioral Diagrams

Diagram Name	Classification	Primary Modeling Focus	Key Notational Elements
Class Diagram	Structural	Static class structures, attributes, methods, and relationships	Classes, methods, generalization, and association
Deployment Diagram	Structural	Hardware execution topology and software artifact deployment	Hardware nodes, artifacts, communication paths
Use Case Diagram	Behavioral	System functional boundary and external actor interactions	Actors, use cases, boundary box, include/extend links
Sequence Diagram	Behavioral	Chronological message interactions exchanged between objects	Object lifelines, activation bars, sync/async messages

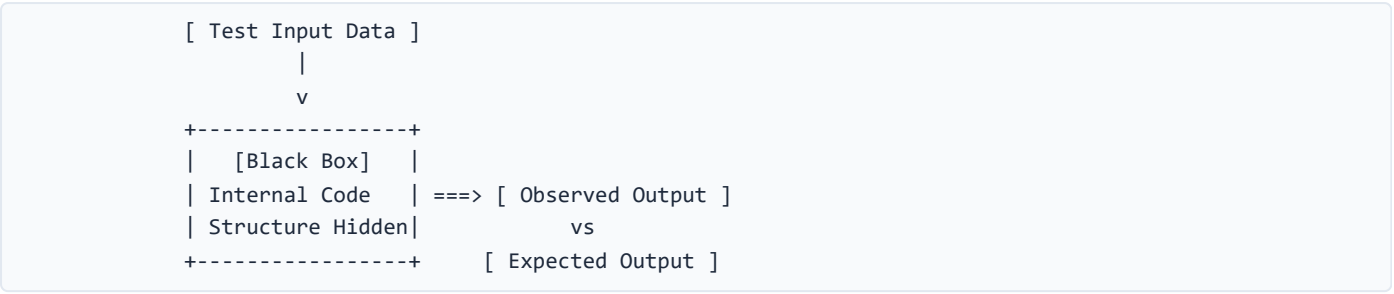
---

Q21. Describe black-box testing and explain its testing techniques with suitable examples.

1. Core Concept

Black-box testing (also known as behavioral or functional testing) evaluates software functionality purely against requirements specifications without inspecting internal source code, logic structures, or internal data paths. The system is treated as an opaque box where testers apply targeted inputs and verify whether actual outputs match expected specifications. It is extensively applied during system testing and user acceptance testing (UAT) to validate compliance with business needs.

2. Black-Box Testing Model



3. Primary Black-Box Testing Techniques

- **Equivalence Class Partitioning (ECP):** Divides input data into valid and invalid partitions, selecting representative test values assuming all elements within a partition yield identical results.
- **Boundary Value Analysis (BVA):** Focuses testing on boundary extremes (minimum, minimum+1, maximum-1, maximum) where coding errors statistically concentrate.
- **Decision Table Testing:** Maps complex business logic combinations into tabular condition-action matrices, ensuring thorough coverage of multi-condition branches.
- **State Transition Testing:** Tests system behavioral transitions between discrete states in response to valid and invalid triggering events.

4. Practical BVA Example: Password Length (6 to 12 Characters)

Consider an authentication module requiring passwords between 6 and 12 characters inclusive. BVA selects inputs directly at the boundary thresholds to prevent off-by-one errors:

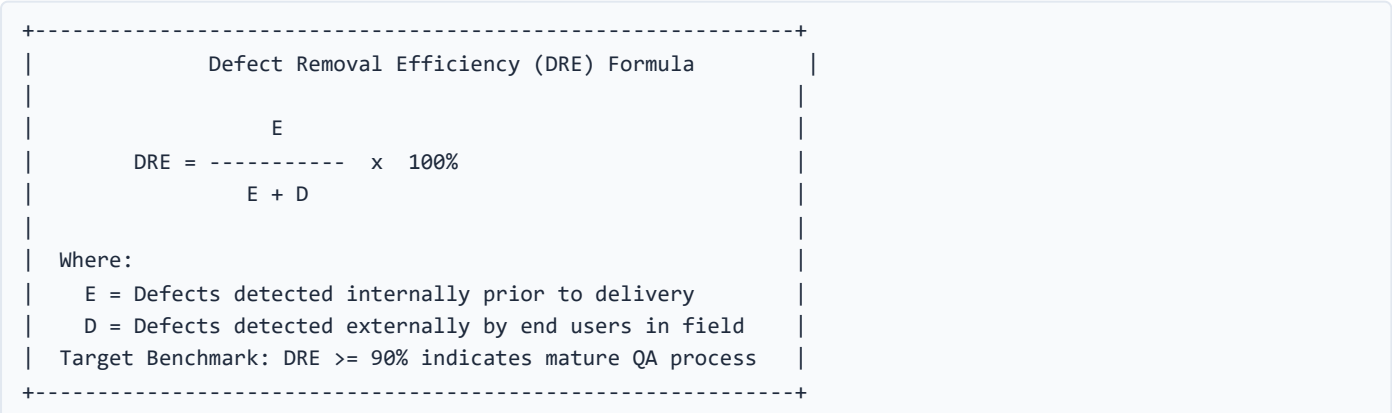
Test Case ID	Password Length	Boundary Position	Partition Type	Expected System Action
TC-01	5 characters	Min − 1	Invalid Partition	Rejected with validation error: "Password too short"
TC-02	6 characters	Min (Lower Edge)	Valid Partition	Accepted as valid password
TC-03	12 characters	Max (Upper Edge)	Valid Partition	Accepted as valid password
TC-04	13 characters	Max + 1	Invalid Partition	Rejected with validation error: "Password too long"

Q22. Explain software quality metrics and Defect Removal Efficiency.

1. Core Concept

Software quality metrics provide quantitative measures of the degree to which a software system, component, or development process satisfies specified customer requirements and engineering standards. Formulated by Capers Jones, Defect Removal Efficiency (DRE) is a paramount process quality metric that measures the capability of quality assurance activities (inspections, reviews, and test phases) to eliminate defects prior to production release.

2. Defect Removal Efficiency (DRE) Formulation



3. Categories of Software Quality Metrics

- **Product Quality Metrics:** Quantifies functional attributes of deliverables, such as defect density per KLOC, cyclomatic complexity, and test code coverage.
- **In-Process Quality Metrics:** Evaluates ongoing development phase health, measuring defect discovery rates, phase containment, and review inspection speed.
- **Maintenance Quality Metrics:** Assesses operational stability and maintainability, including Backlog Management Index, mean time to patch, and fix response time.
- **Phase Containment Leverage:** Higher DRE prevents defect amplification, where early design flaws escape into production, escalating rework costs exponentially and causing project schedule slippage.

4. Concrete Numerical DRE Calculation Example

Scenario: An enterprise software team detects and resolves `E = 450` defects during pre-release code reviews and integration testing. After production deployment, end users report `D = 50` defects during the initial release quarter.

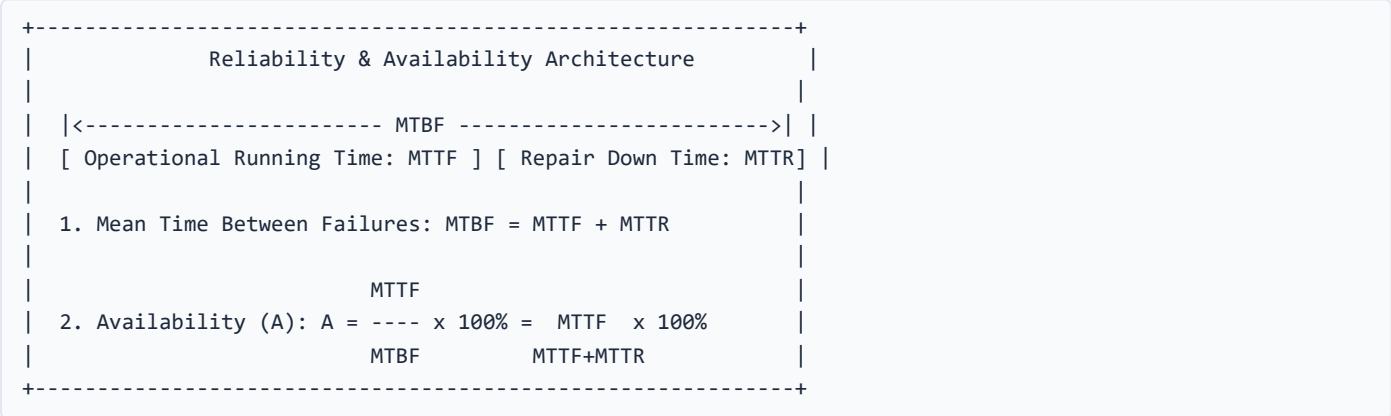
- **Total Defect Volume:** `Total = E + D = 450 + 50 = 500 defects`
- **DRE Calculation:** `DRE = (450 / 500) × 100% = 90.0%`
- **Process Evaluation:** The achieved DRE of **90.0%** satisfies industry benchmark standards (≥ 90%), confirming that 9 out of 10 defects were filtered before customer deployment.

Q23. Explain the measures of software reliability and availability, including MTTF, MTTR, and MTBF.

1. Core Concept

Software reliability is the probability of failure-free software execution for a specified duration in a specified environment (expressed as `R(t) = e^(-λt)`). Software availability (`A`) is the probability that a system is operational and accessible to deliver required services at any given point in time, combining failure-free operation with rapid service restoration.

2. Reliability and Availability Architecture



3. Detailed Metric Definitions

- **Mean Time To Failure (MTTF):** The expected operational uptime of the software between successive failures; reflects underlying code reliability and robustness.
- **Mean Time To Repair (MTTR):** The average downtime required to diagnose, debug, patch, retest, and restore a failed application back to full operation.
- **Mean Time Between Failures (MTBF):** The total operational cycle duration between consecutive failure events, mathematically defined as `MTBF = MTTF + MTTR`.
- **Operational Availability (A):** The percentage of overall operational time the system remains functional; mission-critical systems target "five nines" (99.999% availability).

4. Concrete Numerical Calculation and Summary Comparison

Numerical Example: A cloud payment microservice operates for `MTTF = 995 hours` before encountering an unhandled crash. Automated recovery restores service in `MTTR = 5 hours`.

- `MTBF = MTTF + MTTR = 995 + 5 = 1000 hours`

- `Availability (A) = (MTTF / MTBF) × 100% = (995 / 1000) × 100% = 99.5%`

Reliability Metric	Primary Measurement Focus	Mathematical Formula	Target Objective
MTTF	Continuous operational failure-free uptime	Total Operating Time / Total Failures	Maximize (thousands of hours)
MTTR	Downtime spent in diagnosis and repair	Total Repair Downtime / Total Failures	Minimize (minutes / seconds)
Availability (A)	Percentage of time system is fully operational	[MTTF / (MTTF + MTTR)] × 100%	Maximize (≥ 99.9%)

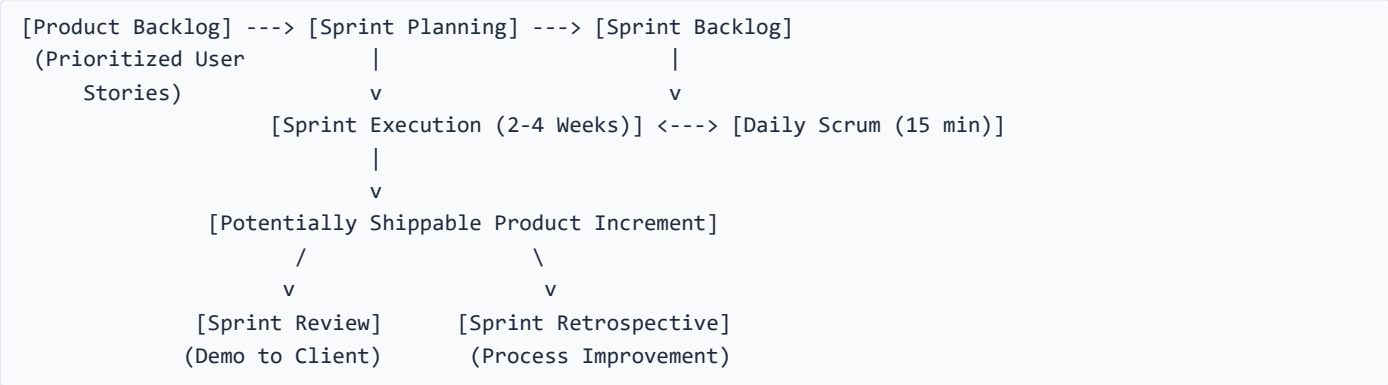
MODEL QUESTION PAPER – 5

Q17. Explain Agile software development and its basic concepts.

1. Core Concept

Agile software development is an iterative, incremental software engineering approach governed by the 2001 Agile Manifesto. It prioritizes customer collaboration, adaptive planning, rapid delivery of working software increments, and flexible responsiveness to changing requirements over rigid procedural documentation.

2. Scrum Sprint Lifecycle Architecture Diagram



3. Core Concepts & Framework Pillars

- **Agile Manifesto Values:** Values individuals and interactions over processes, working software over documentation, customer collaboration over contract negotiation, and responding to change over following a plan.
- **Timeboxed Sprints:** Partitions development into fixed 2-to-4 week iterations yielding a verified, potentially shippable product increment at each cycle end.
- **Scrum Artifacts:** Maintains a prioritized Product Backlog owned by the Product Owner and an actionable Sprint Backlog managed autonomously by the self-organizing engineering team.
- **Scrum Ceremonies:** Facilitates structured team events: Sprint Planning, 15-minute Daily Scrum synchronization, Sprint Review stakeholder demonstrations, and Sprint Retrospective process refinements.
- **Empirical Process Control:** Relies on transparency, inspection, and adaptation, tracking progress via burn-down charts and real-time sprint velocity metrics.

4. Agile vs. Traditional Waterfall Comparison Table

Dimension	Agile (Scrum)	Traditional Waterfall
Development Model	Iterative and incremental sprints	Linear, sequential phases
Requirement Flexibility	Welcomes changing requirements late	Rigid, frozen requirements baseline
Customer Engagement	Continuous throughout every sprint	Limited to initial and final stages
Delivery Frequency	Working increments every 2–4 weeks	Single final monolithic delivery

---

Q18. What is Requirements Management ? Explain the activities involved in Requirement Management.

1. Core Concept

Requirements Management is a systematic engineering process of eliciting, documenting, organizing, and controlling evolving software requirements throughout the entire system life cycle. Because project requirements change constantly due to emerging market needs and technical constraints, it ensures that all modifications are systematically evaluated, approved, and tracked across all design and testing artifacts.

2. Requirements Change Management Workflow Diagram



3. Key Requirements Management Activities

- **Requirements Version Control:** Establishes unambiguous identifiers and formal version numbers for each requirement document, preventing concurrent editing conflicts and preserving audit baselines.
- **Change Control Management (CCB):** Evaluates formal change requests via a multidisciplinary Change Control Board that analyzes cost, schedule, and architectural feasibility before approving modifications.
- **Requirements Traceability (RTM):** Constructs a bidirectional Requirements Traceability Matrix linking each user requirement forward to architectural design modules and test cases, and backward to original business needs.
- **Impact Analysis:** Assesses technical dependencies, estimating the ripple effects, testing overhead, and budget adjustments required if a proposed change is integrated into the baseline.
- **Requirement Status Tracking:** Monitors and categorizes requirements through defined operational states (Proposed, Under Review, Approved, Implemented, Verified, or Deferred) to ensure transparent progress reporting.

4. Summary Table

Activity	Primary Mechanism	Strategic Project Benefit
Version Control	Baseline labeling & changelogs	Prevents unauthorized modifications
CCB Governance	Multidisciplinary approval gate	Eliminates uncontrolled scope creep
Traceability (RTM)	Bidirectional linkage mapping	Guarantees test coverage & zero missing features
Impact Analysis	Dependency evaluation graph	Prevents regression bugs & unexpected delays

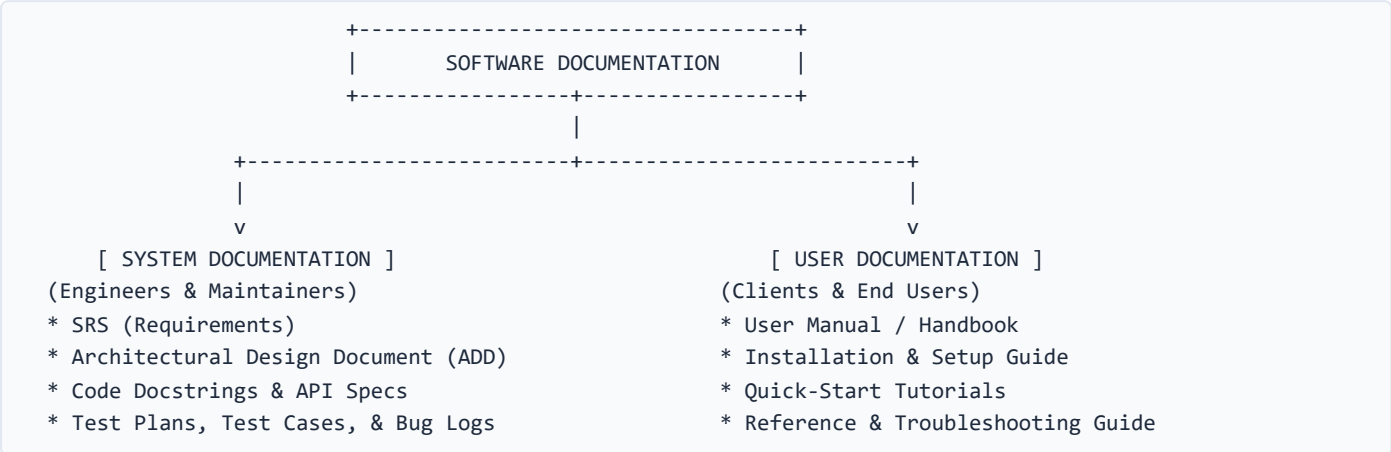
---

Q19. Write a note on software documentation.

1. Core Concept

Software documentation encompasses all written materials, diagrams, and digital artifacts that describe the architecture, operational capabilities, design rationale, and maintenance procedures of a software system. High-quality documentation reduces maintenance costs, facilitates seamless knowledge transfer among developers, and enables end users to utilize the application effectively.

2. Documentation Architecture Hierarchy



3. Core Categories & Best Practices

- **System Documentation:** Internal technical records created during the SDLC for architects and maintainers, including the Software Requirements Specification (SRS), architectural diagrams, API schemas, and test suites.
- **User Documentation:** External instructional materials tailored for non-technical end users and system administrators, including installation runbooks, user manuals, FAQs, and troubleshooting guides.
- **Continuous Synchronization:** Documentation must be maintained concurrently alongside code changes via version control (Docs-as-Code) to avoid dangerous obsolescence.
- **Clarity & Audience Targeting:** Information must use unambiguous terminology tailored to the reader's expertise, employing diagrams and procedural checklists rather than ambiguous prose.
- **Automated Generation:** Employs automated tools (Swagger/OpenAPI for REST APIs, Javadoc, Sphinx) to generate living documentation directly from synchronized code comments.

4. System vs. User Documentation Comparison Table

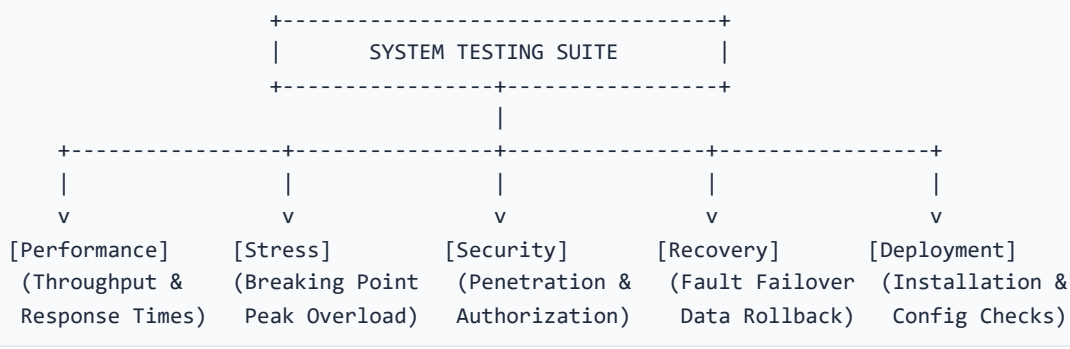
Dimension	System Documentation	User Documentation
Target Audience	Software engineers, maintainers, QA	End users, system administrators, operators
Primary Purpose	Guide system evolution, debugging, refactoring	Guide software operation, configuration, setup
Key Artifacts	SRS, UML models, API schemas, Test Plans	User manuals, tutorials, troubleshooting guides
Technical Depth	High (algorithms, schemas, source code links)	Low to moderate (task workflows, UI screens)

Q20. Describe the different types of system testing such as recovery testing, security testing, stress testing, performance testing, and deployment testing.

1. Core Concept

System testing is a black-box testing phase that evaluates a completely integrated software system against specified business requirements. Conducted by an independent QA team in a production-like environment, it verifies end-to-end functionality, security, performance, and recovery capabilities.

2. System Testing Execution Framework



3. Detailed Breakdown of Key System Testing Types

- **Performance Testing:** Evaluates system responsiveness, throughput, and transaction latency under predictable normal operational workloads using tools like Apache JMeter.
- **Stress Testing:** Evaluates software robustness by subjecting the system to extreme conditions beyond rated capacity to identify breaking points and graceful degradation.
- **Security Testing:** Verifies protective controls by conducting penetration testing and vulnerability scans to prevent unauthorized data access, SQL injection, and exploits.
- **Recovery Testing:** Forces software to fail artificially via fault injection (e.g., server crash, network failure) to verify automated failover and data integrity.
- **Deployment / Installation Testing:** Validates that software installs, upgrades, and uninstalls cleanly across target operating systems without configuration errors.

4. Summary Comparison Table

System Testing Type	Primary Objective	Key Evaluation Metric
Performance Testing	Validate speed and efficiency under expected load	Response time (ms), TPS throughput
Stress Testing	Determine breaking point and recovery behavior	Failure threshold, resource exhaustion
Security Testing	Detect exploitable vulnerabilities and loopholes	Authorization breaches, OWASP Top 10
Recovery Testing	Validate automated failover and data integrity	MTTR (Mean Time to Recover), zero data loss
Deployment Testing	Confirm smooth installation and environment setup	Clean install logs, zero configuration errors

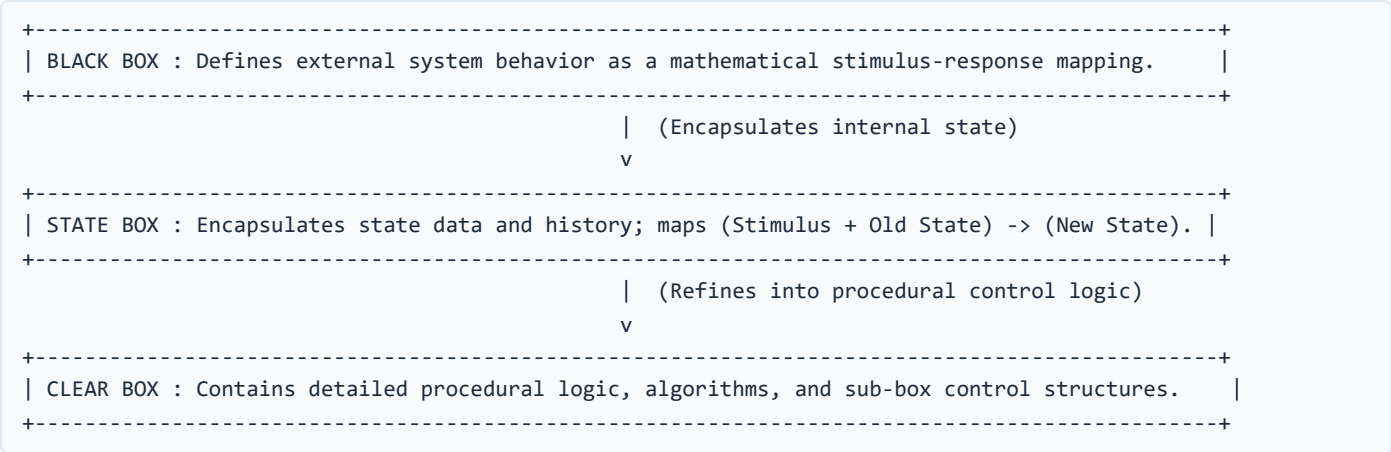
---

Q21. Explain Cleanroom Software Development, its process, box structures, testing, and advantages and disadvantages.

1. Core Concept

Cleanroom Software Development is a formal engineering methodology developed by Harlan Mills emphasizing mathematical verification over defect detection to produce zero-defect software. It replaces traditional developer unit testing with formal specification, mathematical correctness proofs, and independent statistical usage testing.

2. Cleanroom Box Structure Hierarchy



3. Key Process Elements & Testing Strategy

- **Box Structure Refinement:** Progressively transforms user requirements from Black Box (stimulus-response) to State Box (state transitions) to Clear Box (procedural logic).
- **Formal Correctness Proofs:** Engineers perform rigorous mathematical verification to certify that Clear Box designs fulfill State Box specifications before compilation.
- **Statistical Usage Testing:** Prohibits developer unit testing; an independent testing team executes test cases derived from operational usage profiles to certify statistical MTTF.
- **Core Advantages:** Reduces defect density to near-zero levels, provides certified statistical reliability metrics, and significantly lowers maintenance costs.
- **Key Limitations:** Requires advanced mathematical training, increases initial engineering effort, and can be difficult to scale on rapidly changing projects.

4. Summary Table of Box Structures

Box Structure	Information Focus	Viewpoint	Mathematical Basis
Black Box	External system behavior	User	Stimulus -> Response history
State Box	Internal data & persistent state	Architect	(Stimulus + Old State) -> (New State + Response)
Clear Box	Algorithmic & control flow	Programmer	Stepwise procedural refinement & correctness proofs

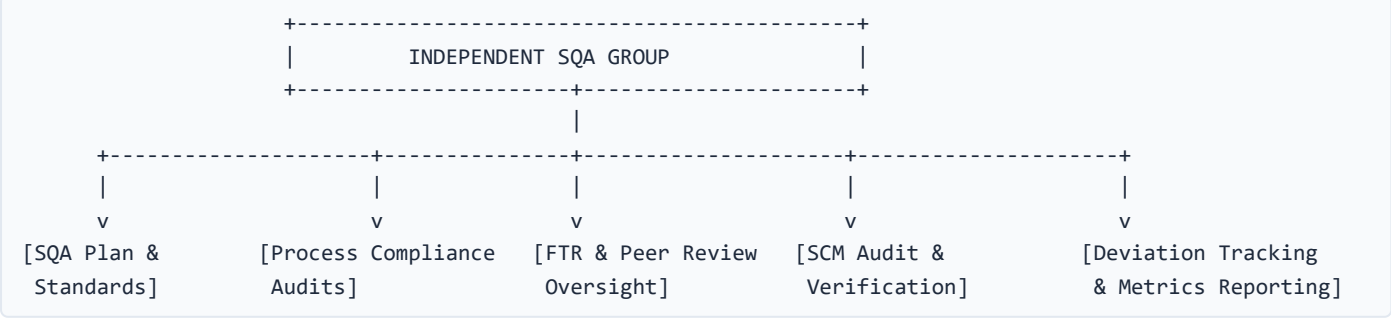
---

Q22. Explain Software Quality Assurance and describe the major activities performed by the SQA group.

1. Core Concept

Software Quality Assurance (SQA) is a planned, systematic management pattern ensuring software processes and work products conform to organizational standards. While software testing detects defects in executable code, SQA is a process-oriented prevention framework aligned with ISO 9001 and CMMI standards.

2. SQA Organizational Framework Diagram



3. Major Activities Performed by the SQA Group

- **Developing the SQA Plan:** Formulates the project quality charter detailing engineering standards, audit milestones, review schedules, and acceptance criteria.
- **Conducting Process Compliance Audits:** Evaluates project activities against organizational SDLC processes (e.g., CMMI Level 5), documenting deviations for corrective action.
- **Participating in Formal Technical Reviews (FTR):** Monitors peer reviews across requirements, designs, and code to ensure review rigor and evaluate defect containment.
- **Auditing Configuration Management (SCM):** Verifies that software baselines, branch merges, and change requests conform to configuration control protocols.
- **Tracking Non-Compliance:** Documents process non-conformance issues, tracking deviations until resolved and escalating risks to executive management.
- **Analyzing Quality Metrics:** Gathers and evaluates statistical metrics (Defect Removal Efficiency [DRE], defect density) to guide continuous process improvement.

4. SQA vs. Software Testing Comparison Table

Dimension	Software Quality Assurance (SQA)	Software Testing
Primary Focus	Process-oriented (How software is built)	Product-oriented (Finding bugs in built code)
Objective	Defect prevention across the entire SDLC	Defect detection in executable artifacts
Execution Phase	Continuous umbrella activity across all phases	Conducted during construction and verification phases
Responsibility	Independent SQA Group & Project Leadership	Developers and dedicated QA test engineers

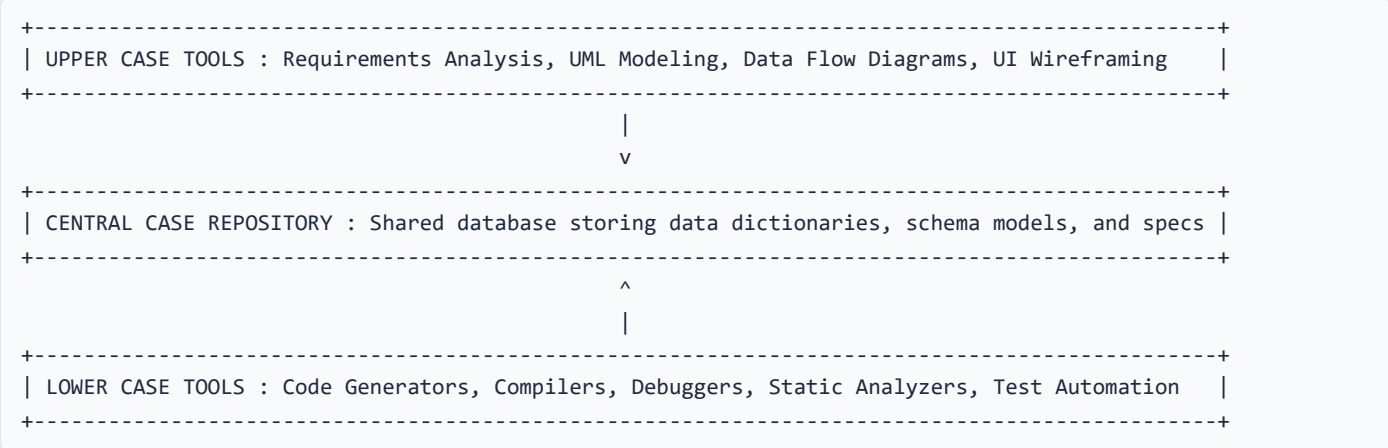
---

Q23. Explain CASE tools, their objectives, categories of support, CASE environment architecture, benefits, limitations, and selection criteria.

1. Core Concept

Computer-Aided Software Engineering (CASE) refers to a collection of automated and semi-automated software tools that provide structured support for software engineering activities across all SDLC phases. CASE tools aim to enhance developer productivity, improve software quality, enforce engineering standards, and reduce life cycle delivery costs.

2. Integrated CASE (I-CASE) Environment Architecture



3. Core Categories, Benefits, and Selection Criteria

- **Upper CASE Tools:** Automates early front-end life cycle phases (Planning, Requirements Analysis, and Architectural Design), including UML modeling tools (Enterprise Architect) and requirements management platforms.
- **Lower CASE Tools:** Automates back-end technical execution phases (Implementation, Testing, and Maintenance), including compilers, automated test frameworks (Selenium, JUnit), and code refactoring linters.
- **Integrated CASE (I-CASE):** Unifies Upper and Lower tools through a common Central Repository, providing end-to-end requirement traceability and automated code synthesis.

- **Key Benefits:** Accelerates development velocity, eliminates manual drawing errors, maintains standardized documentation, and lowers long-term maintenance costs.
- **Key Limitations:** High upfront licensing and infrastructure costs, steep developer learning curves, and potential tool incompatibility across multivendor toolchains.
- **Selection Criteria:** Evaluates tool portability, vendor support, repository integration, ease of learning, and compatibility with organizational SDLC standards.

4. Summary Comparison Table

CASE Category	SDLC Phases Supported	Primary Deliverables	Typical Tools
Upper CASE	Planning, Analysis & Design	SRS documents, UML diagrams, ERDs	Enterprise Architect, StarUML
Lower CASE	Implementation & Testing	Executable code, test scripts, bug logs	Eclipse, GCC, Selenium, JUnit
Integrated CASE	Full end-to-end SDLC	Complete traceable project repository	Jira + Git + Jenkins + SonarQube

---