UNIT–3 : Software Design & Software Metrics (PYQ Solution)

AKTU · Software Engineering (BCS601) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. Write a note on quality procedure/standards as prescribed by the ISO and Capability Maturity Model (CMM). (May 2026)

ISO 9000 is a series of international quality-management standards; ISO 9001 certifies that an organisation documents its processes, follows them, audits itself and corrects defects, and is checked by an external auditor (ISO 9000-3 applies it to software). CMM, developed by SEI, rates the maturity of a software process on five levels: Initial, Repeatable, Defined, Managed and Optimising, each with Key Process Areas. ISO gives a pass/fail certificate; CMM gives a graded improvement path.

Q2. Give a detailed overview of Size Oriented Measures of software. (May 2025)

Size-oriented metrics use the lines of code (LOC or KLOC) as the normalising value. From the project's records we calculate errors per KLOC, defects per KLOC, cost per LOC (₹/LOC), documentation pages per KLOC and productivity (LOC per person-month). They are easy to compute and directly related to effort, but depend on the programming language, coding style and do not suit non-procedural or design-heavy work. Hence function-oriented measures are also used.

Q3. What do you mean by horizontal and vertical partitioning? (2023)

Partitioning divides a system into manageable modules. Horizontal partitioning splits the program into separate branches of the control hierarchy by function, e.g. input, processing and output modules, so each can be tested and changed independently. Vertical partitioning (factoring) distributes control and work

top-down: decision-making modules sit high in the structure and the working modules sit low. Horizontal gives modularity by function; vertical gives better control and easier maintenance.

Q4. What is the software architecture? (2023)

Software architecture is the high-level structure of a system: its components (modules), their externally visible properties and the relationships (connectors) between them, along with the design rules guiding them. It acts as a blueprint for design, communication with stakeholders and analysis of attributes such as performance, security and modifiability. Common styles are layered, client-server, pipe-and-filter, MVC and repository.



SECTION B Answers (≈250 Words Each)

Q1. Discuss various software reliability metrics. Give a brief description of various software reliability models. (May 2026)

Software reliability is the probability that software works without failure for a specified time in a specified environment.

Reliability metrics

- **MTTF (Mean Time To Failure)** – average time between successive failures.
- **MTTR (Mean Time To Repair)** – average time to fix a failure and restart.
- **MTBF = MTTF + MTTR** – mean time between failures.
- **ROCOF (Rate of Occurrence of Failure)** – failures per unit time; ROCOF = 1/MTTF.
- **POFOD (Probability of Failure On Demand)** – chance that a request for service fails; used for safety systems.
- **Availability** = MTTF / (MTTF + MTTR) × 100%.
- **Failure intensity** – failures per unit of execution time.

Reliability models

- **Basic execution time (Musa) model** – failure intensity falls exponentially as failures are fixed: $\lambda(\mu) = \lambda_0 (1 - \mu/v_0)$.
- **Logarithmic Poisson model** – failure intensity decreases logarithmically; later failures are fixed with less effect.
- **Jelinski-Moranda model** – assumes a fixed number of faults, each fault equally likely to cause failure and removed perfectly, so the failure rate drops by a fixed amount after each fix.
- **Littlewood-Verrall model** – a Bayesian model where fault sizes differ and the failure rate is random.
- **Goel-Okumoto (NHPP) model** – the number of failures follows a non-homogeneous Poisson process.
- **Reliability growth models** – track improvement during debugging; **reliability prediction models** use design data before testing.

Model	Key assumption
Jelinski-Moranda	Fixed faults, equal effect
Musa basic	Exponential decrease
Musa-Okumoto	Logarithmic decrease
Littlewood-Verrall	Random failure rate

Conclusion: the metrics quantify reliability, and the models use failure data to predict how reliability grows and when testing can stop.

Q2. List various drawbacks of the Line of Code (LOC) based size measure of software. Discuss the Albrecht Function Point count method. Show that the Complexity Adjustment Factor (CAF) adjusts the unadjusted value of Function Point (UFP) to ± 35%. (May 2025)

Drawbacks of LOC

- Depends on the **programming language** and coding style.
- Known exactly only **after coding**, so it is poor for early estimation.
- Rewards long code and penalises efficient, short code.
- Ignores design, documentation and quality effort.
- Cannot measure non-coding work and is hard to compare across projects.
- Different counting rules (comments, blank lines).

Albrecht Function Point method: FP measures the functionality delivered, independent of language.

1. Count the five **function types**: External Inputs (EI), External Outputs (EO), External Inquiries (EQ), Internal Logical Files (ILF), External Interface Files (EIF).
2. Rate each as simple, average or complex and multiply by its weight to get **UFP** (e.g. EI 3/4/6, EO 4/5/7, EQ 3/4/6, ILF 7/10/15, EIF 5/7/10).
3. Rate **14 General System Characteristics** (data communication, performance, reusability etc.), each from 0 to 5.
4. **CAF = 0.65 + 0.01 × ΣF_i**, where ΣF_i is the total score.
5. **FP = UFP × CAF.**

Why CAF changes UFP by ±35%:

- Minimum $\Sigma F_i = 14 \times 0 = 0 \rightarrow$ CAF = 0.65 + 0 = **0.65**, so FP = 0.65 UFP, a **35% decrease**.
- Maximum $\Sigma F_i = 14 \times 5 = 70 \rightarrow$ CAF = 0.65 + 0.70 = **1.35**, so FP = 1.35 UFP, a **35% increase**.
- Hence $0.65 \leq CAF \leq 1.35$ and FP lies within **UFP ± 35%**.

Conclusion: FP is a language-independent size measure whose adjustment is limited to ± 35% of UFP.

Q3. Explain about the structure chart and all its types with suitable example. (2023)

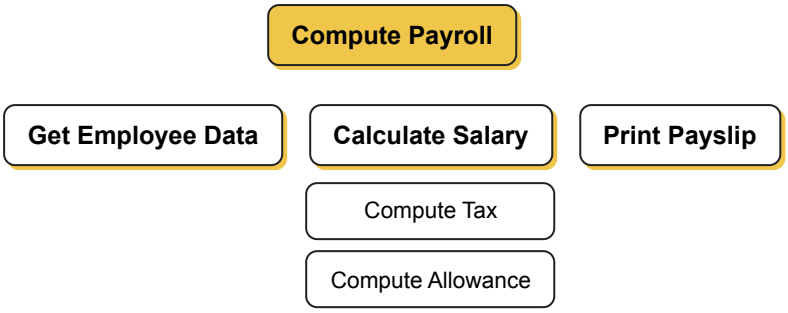
A **structure chart** is a design tool showing the **modular hierarchy** of a program: modules as rectangles, calls as arrows, and data/control flow between modules shown by small arrows with circles.

Symbols

- **Module** – a rectangle with its name.
- **Arrow** – a call from a parent (invoking) module to a child (invoked) module.
- **Data couple** – arrow with an open circle for data passed.
- **Control couple (flag)** – arrow with a filled circle for a control signal.
- **Selection (decision)** – a diamond; the parent calls one of several children.
- **Iteration (loop)** – a curved arrow; the parent calls a child repeatedly.

Types of modules by role: input (afferent) modules bring data in, **output (efferent)** modules send data out, **transform (central)** modules process the data and **coordinate** modules manage the flow.

Example – Payroll: the top module "Compute Payroll" calls "Get Employee Data" (input), "Calculate Salary" (transform) and "Print Payslip" (output). "Calculate Salary" calls "Compute Tax" and "Compute Allowance", with data couples such as basic pay and tax. "Print Payslip" is called inside a loop for each employee.



Structure chart of a payroll program

Use: it shows module hierarchy, coupling and cohesion, and guides coding and integration.

Conclusion: the structure chart is the main design output of structured design, showing how modules are organised and communicate.



SECTION C Answers (≈500 Words Each)

Q1. Write down various characteristics of a good software design. Discuss the procedure for determination of strength of design using concept of Coupling, Cohesion and Dependence Matrix. (May 2026, May 2025)

Software design converts the SRS into a blueprint (architecture, modules, interfaces and data) from which the code can be written.

Characteristics of a good design

- **Correct and complete** – implements every requirement in the SRS.
- **Modular** – divided into independent modules, each with one clear job.
- **High cohesion, low coupling** – the strongest sign of a good design.
- **Understandable and simple** – readable by developers and maintainers.
- **Traceable** to requirements and **consistent** in notation.
- **Flexible and maintainable** – changes affect few modules.
- **Reusable and portable** components.
- **Efficient and robust** – good performance and handles errors.
- **Testable** and **well documented**.

Coupling is the degree of interdependence between modules; **cohesion** is the degree to which elements inside a module belong together. Good design **= low coupling + high cohesion**.

Coupling (best → worst)	Cohesion (best → worst)
Data	Functional
Stamp	Sequential
Control	Communicational
External	Procedural
Common	Temporal
Content	Logical, Coincidental

Dependence matrix – a square matrix with the modules as rows and columns. The entry **D[i][j] = 1** if module i depends on (calls or uses data of) module j, otherwise 0.

Procedure for design strength

1. List the n modules and build the n × n dependence matrix.
2. For each module, count its **fan-out** (row sum) and **fan-in** (column sum); the diagonal is zero.
3. **Coupling measure** = total number of 1s / [n(n − 1)]. A value near 0 means low coupling (strong design); near 1 means tight coupling.
4. Check the coupling **type** of each dependency (data, control, common...) and prefer data coupling.
5. Find the **cohesion level** of each module from its description and give it a score (functional = 7 ... coincidental = 1).
6. Combine: design is **strong** when average cohesion is high and the dependence density is low; modules that are mutually dependent (symmetric 1s) are candidates for merging or redesign.

Example: modules A, B, C with A → B, A → C only: matrix has two 1s out of n(n−1) = 6 possible, so coupling = 0.33, which is low.

Conclusion: a good design is modular, simple and traceable, and its strength is judged by low coupling (from the dependence matrix) and high cohesion.

Q2. Write down various methods of estimation. Discuss the COCOMO hierarchy of models for estimation of Effort and Cost. (May 2026)

Software project estimation predicts size, effort, time and cost before and during development.

Methods of estimation

- **Expert judgement** and the **Delphi technique** – estimates by experienced people, refined in rounds.
- **Estimation by analogy** – compare with similar past projects.
- **Work Breakdown Structure (bottom-up)** – estimate each task and add up.
- **Size-based methods** – LOC or Function Points × productivity.
- **Algorithmic/empirical models** – **COCOMO**, Putnam (SLIM), Use Case Points.
- **Top-down** estimation from total budget or size.

COCOMO (Constructive Cost Model) by Barry Boehm estimates effort in **person-months (PM)** from size in **KLOC**. It has a **hierarchy of three models**.

1. Basic COCOMO – a quick estimate using only size.
E = a × (KLOC)^b PM, Tdev = c × E^d months

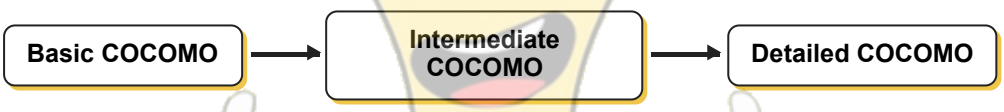
Project class	a	b	c	d
Organic (small, simple, experienced team)	2.4	1.05	2.5	0.38
Semi-detached (medium)	3.0	1.12	2.5	0.35
Embedded (tight constraints)	3.6	1.20	2.5	0.32

2. Intermediate COCOMO – adds **15 cost drivers** (product, hardware, personnel and project attributes) rated very low to extra high; their product is the **Effort Adjustment Factor (EAF)**: **E = a × (KLOC)^b × EAF**, with a = 3.2, 3.0, 2.8 for the three classes.

3. Detailed (Complete) COCOMO – applies the cost drivers **phase-wise** (plan, design, code, integration and test) and at module/subsystem level, giving a more accurate estimate.

Cost: Cost = Effort × average monthly salary; staff = E / Tdev.

Example: 32 KLOC organic project: E = 2.4 × 32^1.05 = 2.4 × 38.0 ≈ 91 PM; Tdev = 2.5 × 91^0.38 ≈ 2.5 × 5.55 ≈ 13.9 months; staff ≈ 6.6 persons.

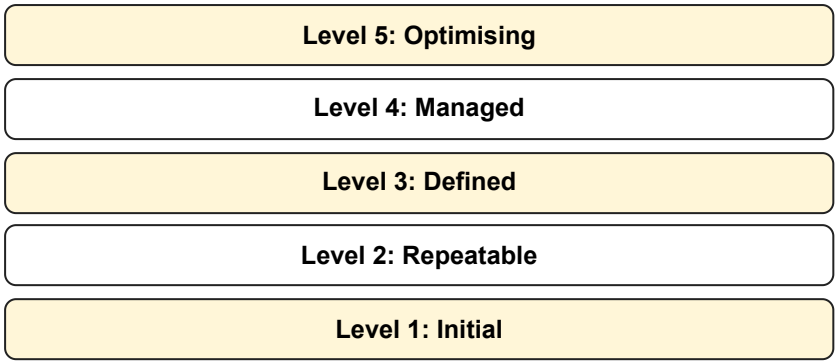


COCOMO hierarchy (increasing accuracy)

Conclusion: COCOMO gives effort, time and cost from size, and moving from Basic to Detailed adds cost drivers for greater accuracy.

Q3. Discuss the SEI Capability Maturity Model and compare it with the ISO 9000 certification giving various distinctive features of each model. Explain the SEI-CMM model. What do you mean by state of fire fighting. (May 2025, 2023)

SEI CMM (Capability Maturity Model) was developed by the Software Engineering Institute (Carnegie Mellon) to assess and improve the maturity of an organisation's software process. It has **five maturity levels**, each with **Key Process Areas (KPAs)**.



SEI CMM maturity levels

- **Level 1 – Initial:** ad hoc, chaotic process; success depends on individual heroes; no KPAs.
- **Level 2 – Repeatable:** basic project management; KPAs: requirements management, project planning, tracking, subcontract management, SQA, configuration management.
- **Level 3 – Defined:** the process is documented and standardised organisation-wide; KPAs: process definition, training, integrated management, peer reviews, intergroup coordination.
- **Level 4 – Managed:** process and product quality are measured quantitatively; KPAs: quantitative process management, software quality management.
- **Level 5 – Optimising:** continuous improvement using feedback and new technology; KPAs: defect prevention, technology and process change management.

State of fire fighting: an organisation at **Level 1** works in a "fire-fighting" mode: there is no planning, crises are handled as they arise, deadlines are missed, schedules and costs are unpredictable, and the team keeps rushing to fix urgent problems rather than preventing them.

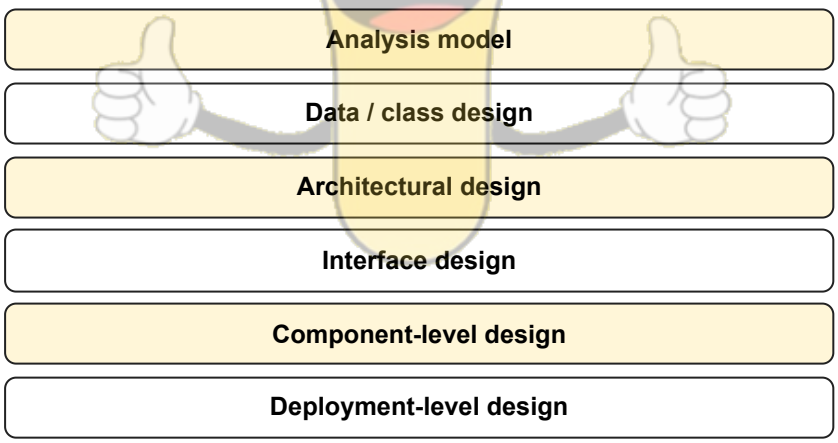
ISO 9000 is a set of international quality-management standards (ISO 9001 for design, development and service) that certifies that a documented quality system exists and is followed.

Feature	SEI CMM	ISO 9000
Origin	SEI, USA (software)	International, any industry
Result	Maturity level 1-5	Pass/fail certificate
Focus	Continuous improvement	Meeting minimum quality system
Scope	Software process	Whole quality system
Assessment	By SEI-authorised assessors	By accredited registrars
Validity	Time-limited rating	Periodic surveillance (3 years)
Approach	Improvement path with KPAs	Documented, auditable system

Conclusion: CMM gives a ladder of improvement for software processes, whereas ISO 9000 certifies a basic quality system; organisations often use both.

Q4. Draw the software design framework and discuss the elements of design model. (2023)

The **design model** transforms the analysis model into a blueprint for construction. Design flows from the analysis model (scenario, class, flow and behaviour elements) to the four design elements.



Software design framework

Elements of the design model

- **Data (class) design** – converts the analysis classes and data objects into data structures and database design at different levels of detail.
- **Architectural design** – defines the overall structure: major components, their relations and the architectural style (layered, client-server), derived from the analysis model and application domain.

- **Interface design** – describes how the software communicates: **user interface** (screens, menus), **external interfaces** (with other systems and devices) and **internal interfaces** (between components).
- **Component-level design** – describes the internal detail of each component: algorithms, data structures and interfaces, using flowcharts or pseudocode.
- **Deployment-level design** – shows how software is arranged on the physical hardware and nodes.

Design quality guidelines: the design should be evolutionary, modular, distinct in components, represent data, architecture, interfaces and components, and use repeatable methods.

Process: requirements → data design → architectural design → interface design → component design → review.

Conclusion: the design model has data, architectural, interface and component elements, each refining the analysis model into a form that can be coded.

Q5. Illustrate the principles of software design. Discuss the characteristics of good software design. (2023)

Principles of software design

- **Abstraction** – hide details and show only the essential; procedural and data abstraction.
- **Modularity** – divide the software into separately named, addressable modules that can be integrated to meet requirements.
- **Information hiding** – a module's internals are hidden from others; communication through interfaces.
- **Functional independence** – modules with a single purpose and little interaction: high cohesion, low coupling.
- **Refinement** – stepwise refinement from a high-level statement to detailed steps.
- **Architecture** – overall structure that organises components.
- **Separation of concerns** – each part addresses one concern.
- **Refactoring** – improving the internal structure without changing behaviour.
- **Design traceability** to requirements and **reuse**.

Characteristics of good design

- **Correct and complete** according to the SRS.

- **Understandable** and simple.
- **Modular** with low coupling and high cohesion.
- **Maintainable and flexible**; changes affect few modules.
- **Efficient** in time and space.
- **Reliable and robust** in handling errors.
- **Testable, portable and reusable.**
- **Well documented and traceable.**

Principle	Benefit
Abstraction	Reduces complexity
Modularity	Parallel development, easy testing
Information hiding	Safe changes
Independence	Easy maintenance
Refinement	Gradual detail

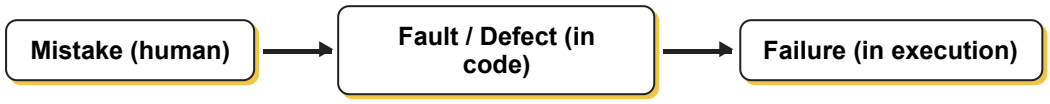
Conclusion: applying these principles gives a modular, simple design whose quality is judged by the characteristics listed.

Q6. Write short notes on: (i) Function Point (ii) COCOMO (iii) Defect, Fault, Failure (2023)

(i) Function Point (FP) – a **size measure** based on the functionality delivered to the user, independent of language. Steps: count five items: **external inputs, external outputs, external inquiries, internal logical files and external interface files**, weight each as simple, average or complex to get the **Unadjusted FP (UFP)**, then multiply by the **Complexity Adjustment Factor CAF = 0.65 + 0.01 × ΣFi** (14 factors rated 0 to 5). **FP = UFP × CAF.** Example: UFP = 100, ΣFi = 40 → CAF = 1.05 → FP = 105. Effort = FP / productivity (FP per person-month). It can be used early in the life cycle, but is subjective.

(ii) COCOMO – the **Constructive Cost Model** (Boehm) estimates effort and time from size in KLOC. **Basic:** E = a(KLOC)^b PM and Tdev = c·E^d, with coefficients for organic (2.4, 1.05), semi-detached (3.0, 1.12) and embedded (3.6, 1.20) projects. **Intermediate** multiplies by an effort adjustment factor from 15 cost drivers. **Detailed** applies the drivers phase-wise. Example: 10 KLOC organic: E = 2.4 × 10^1.05 ≈ 26.9 PM.

(iii) Defect, Fault, Failure – a **mistake (error)** is a human action. A **fault or defect** is an incorrect instruction or design element in the software caused by the mistake. A **failure** is the observed deviation from expected behaviour that occurs when the faulty code executes. Sequence: mistake → fault/defect → failure. Defect and fault are often used for the same thing; defects are counted for quality metrics such as defects per KLOC.



Mistake, fault and failure

Conclusion: FP measures size, COCOMO converts size into effort and cost, and the fault-to-failure chain explains the origin of software errors.

