Unit 1: Fundamentals & Introduction

A. Software Definition and Core Concepts

1. What is Software? Explain its components.

Software is a comprehensive construct that goes beyond just executable code. It is defined as a set of three intertwined elements:

1. **Instructions (Computer Programs):** The executable code that, when run, provides the desired features, function, and performance.
2. **Data Structures:** These structures allow the programs to adequately organize and manipulate information.
3. **Descriptive Information (Documentation):** Information in both hard copy and virtual forms that describes how the programs operate and how they should be used.

- **Equation:** Software = Program + Documentation + Operating Procedures.

2. Differentiate between Program and Software.

Feature	Program	Software
Scope	Narrow; refers strictly to the set of computer instructions.	Broad; encompasses instructions, data, documentation, and procedures.
Components	Only the executable instructions.	Instructions + Data Structures + Documentation + Procedures.
Goal	To perform a specific computational task.	To provide desired features, function, and performance to the user.

3. Differentiate between Process and Product.

- **Process:** A collection of activities, actions, and tasks performed to create a work product. It provides the framework (the "HOW") for development, regardless of the project's size or complexity. A rigorous process ensures quality and consistency.
- **Product:** The set of deliverables (the "WHAT") that results from the process. It represents the solution to the user's problem and includes the software, manuals, and data. The process exists to create the product.

4. What do you mean by "Software doesn't wear out"? Compare it with hardware failure.

- **Concept:** Software is logical, not physical. It does not suffer from environmental degradation (dust, vibration, heat) like hardware.

- **Hardware Failure (Bath Tub Curve):** Hardware follows a "Bath Tub" curve: high failure initially (burn-in), low constant failure (useful life), and rising failure at the end (wear-out phase) due to physical aging.
- **Software Failure (Idealized vs. Reality):**
    - **Idealized:** High initial failure (testing), then drops to a steady low rate. It never curves back up because it doesn't "break" physically.
    - **Reality (Deterioration due to Change):** In practice, software deteriorates because of *change*. As new features or fixes are added, side effects are introduced, causing the failure rate to spike again. This makes the curve jagged and increasing over time, not because of wear, but because of complexity and poorly managed updates.

5. Major Areas of Software Applications:

1. **System Software:** Serves other programs (e.g., OS, compilers).
2. **Application Software:** Standalone business/consumer programs (e.g., Inventory control).
3. **Engineering/Scientific:** Number-crunching algorithms (e.g., CAD, Astronomy).
4. **Embedded Software:** Controls products/systems (e.g., Car breaking systems, microwaves).
5. **Product Line Software:** Integrated suites for specific markets (e.g., Inventory + Finance).
6. **Web/Mobile Applications:** Network-centric software widely accessed by browsers/phones.
7. **Artificial Intelligence (AI):** Uses non-algorithmic heuristics (e.g., Pattern recognition, Robotics).

B. Software Management and Design

6. The 4 P's of Software Management:
Effective software project management focuses on four Ps, in this specific order of dependency:

1. **People:** The most important factor. The "human factor" determines success or failure.
2. **Product:** The software to be built. Objectives and scope must be defined before proceeding.
3. **Process:** The framework chosen to get the job done (e.g., Agile, Waterfall).
4. **Project:** The actual planning and tracking of the work to deliver the product.

7. Strategy of Design:
Design is the problem-solving activity that focuses on "HOW" the system will work. It follows a four-step strategy:

1. **Understand the problem:** (Communication & Analysis phase).
2. **Plan a solution:** (Modeling & Software Design phase). This involves creating the architecture, data structures, and interfaces.
3. **Carry out the plan:** (Code Generation).
4. **Examine the result:** (Testing & QA).

8. Functional vs. Object-Oriented Design Approach:

Feature	Functional (Structured) Design	Object-Oriented Design (OOD)
Focus	Focuses on the **function** or process (algorithms).	Focuses on the **data** (objects) and their interactions.
Decomposition	Decomposes system into functions/modules.	Decomposes system into Objects and Classes.
Viewpoint	Functions transform input to output.	Objects have state (data) and behavior (methods).
Tools	DFDs, Structure Charts, Flowcharts.	UML Diagrams (Class, Sequence, Use Case).

9. Purpose of Problem Partitioning:

Partitioning (dividing a large problem into smaller parts) is crucial because:

- **Reduces Complexity:** Smaller problems are easier to understand and solve.
- **Separation of Concerns:** Allows focusing on one distinct feature at a time.
- **Modularity:** Leads to high **Cohesion** (related tasks stay together) and low **Coupling** (dependencies are minimized), which makes maintenance easier.

C. Process Framework & Myths

10. Process Framework Activities (Generic & Umbrella):

- **Generic Framework Activities:** The core steps taken to build software: Communication, Planning, Modeling, Construction, Deployment.
- **Umbrella Activities:** Activities applied *throughout* the process to **manage** and **control** quality (not just write code):
  - **Software Quality Assurance (SQA):** Ensures standards are followed.
  - **Software Configuration Management (SCM):** Manages changes (versions, backups).
  - **Risk Management:** Identifies and mitigates potential problems.
  - **Measurement:** Collects metrics (size, effort, defects) to improve the process.
  - **Formal Technical Reviews:** Peer reviews to catch errors early.

11. Why is Documentation Important?

Documentation is a core component of software (Software = Program + Documentation).

- **For Developers:** Without it, they don't know precisely what to build.
- **For Customers:** Without it, they don't know what to expect or how to verify the product.
- **For Maintenance:** It is critical for future updates; without it, the system becomes a "black box" that is risky to touch.

12. Software Myths (Management, Customer, Practitioner):

- **Management Myth:** "We have standards, so we are good." -> **Reality:** Standards are often outdated or ignored.
- **Management Myth:** "Adding people to a late project speeds it up." -> **Reality:** It slows it down (Brooks' Law) due to training and communication overhead.
- **Customer Myth:** "A general statement of objectives is enough." -> **Reality:** Vague requirements lead to project failure; details are needed upfront.
- **Customer Myth:** "Requirements can change easily." -> **Reality:** Changes late in the process are exponentially expensive.
- **Practitioner Myth:** "My job is done when the code runs." -> **Reality:** 60-80% of effort is in maintenance *after* delivery.
- **Practitioner Myth:** "Working program is the only deliverable." -> **Reality:** Documentation, models, and test plans are equally critical for long-term success.

13. Purpose of CASE (Computer-Aided Software Engineering):

CASE tools are automated software systems used to support software engineering activities. They reduce manual effort, ensure consistency, and help manage complexity in tasks like design, coding, testing, and project management.

Unit 2: Software Process Models (Final Hybrid)

1. Spiral Model

1.1 Explain the Spiral life cycle model and its limitations. / Discuss the Spiral model.

The Spiral model (Barry Boehm) is a **risk-driven, evolutionary** process model. Development is represented as a **spiral**, where each loop is one phase/iteration.

Each loop is divided into four sectors:

1. Objective setting (objectives, alternatives, constraints)
2. Risk analysis & reduction (identify, evaluate, reduce risks; often via prototyping)
3. Engineering (design, coding, testing for that loop)
4. Planning for next iteration (review, commit, plan next loop)

- Radial axis → **cumulative cost/effort**
- Angular axis → **progress** within the current loop

It can also be seen as repeatedly performing the generic framework activities: Communication, Planning, Modeling, Construction, Deployment, but **guided by explicit risk analysis**.

Advantages:

- Explicit **risk mitigation**
- Supports **evolutionary** development and customer feedback
- Handles **changing requirements** well

Limitations / Disadvantages:

- **Complex** to manage; heavy documentation and risk analysis
- Harder to estimate **total cost and schedule** upfront
- Needs **experienced** staff in risk assessment
- Not appropriate for **small, simple** or low-risk projects

1.2 Most important feature / two distinguishing characteristics

- Most important feature: **Risk management**.
- Two characteristics:
    1. Explicit, continuous **risk analysis** and mitigation
    2. **Radial axis** shows cumulative **cost**, unlike most other models

---

1.3 Why is Spiral considered a meta-model?

It is a **meta-model** because each loop can internally use a different model:

- One loop may use a **Waterfall-like** sequence,
- Another may use **Prototyping**,
- Others may resemble **Incremental** development.

So Spiral acts as a **framework** within which other life-cycle models are selected based on risk.

1.4 Example where Spiral is suitable / not suitable

- **Suitable:** New standalone system using **new or high-risk technology**.
    - Justification: Spiral "is the most suitable model for new technology that is not well understood", because of its built-in risk assessment each loop.
- **Not suitable:** Small, simple programming exercise where **requirements are easily understandable and defined**.
    - Justification: Overhead of risk management is unnecessary; a simple linear model like Waterfall is enough.

1.5 As you move outward along the spiral?

- Software becomes **more complete and more refined**.
- Cumulative **cost and commitment** increase.
- You move towards a fully functional, delivered product or maintenance cycles.

1.6 How does project risk affect the Spiral model?

- The **project risk factor** is explicitly considered in each loop.
- Risk analysis influences decisions about:

    - Whether to continue the project,
    - Which alternatives to try,
    - What kind of prototype or experiment to build.
- Hence Spiral is **highly suitable for high-risk projects**.

1.7 Framework activities in Spiral and V-Model; strengths and weaknesses

Spiral – Framework activities:

- Executes the generic activities:
    - Communication
    - Planning
    - Modeling
    - Construction
    - Deployment
- ...but with an **evolutionary circular flow** and explicit risk analysis.

Strengths (Spiral):

- Flexibility, **explicit risk handling**, supports changing requirements.

Weaknesses (Spiral):

- Management and documentation **overhead**, difficult global estimation.

V-Model – Basic idea:

- Extension of Waterfall emphasizing **verification and validation**.
- Left: Requirements → Design → Detailed Design
- Right: Unit → Integration → System/Acceptance testing, each linked back to its phase.

Strengths (V-Model):

- Good **traceability**; strong focus on testing.

Weaknesses (V-Model):

- Mostly **sequential**, not good for rapidly changing requirements.

---

2. Prototyping & Evolutionary Models

2.1 Prototyping model & when it is suitable

Prototyping model:

- Build a **rapid prototype**.
- Give it to the **user for evaluation** and feedback.
- **Refine** the prototype.
- Repeat till requirements are understood.
- Then build the actual system (often discarding the prototype).

Suitable when:

- **Requirements are frequently changing** or not easily understandable.
- **User participation** is available.

2.2 Advantages and disadvantages of first constructing a prototype

Advantages:

- Active **user participation**
- **Clarifies requirements** and resolves uncertainty
- Gives early **validation/feedback** on the evolving system

Disadvantages:

- Customer may get **satisfied with the prototype** and want to deploy it, even if technically weak.
- Extra **effort/cost** to build a throw-away prototype (though it enhances the knowledge base).

2.3 Effect on overall cost; advantages

- Prototype adds **up-front cost**, but:
  - If thoroughly examined for quality and used to **finalize requirements quickly**, it reduces the heavy cost of late defect fixing.
- Advantages: same as above – better requirements, less rework.

2.4 Prototyping vs evolutionary process model

Feature	Prototyping Model (Throwaway)	Evolutionary Process Model
Goal	Clarify requirements/design concepts.	Build increasingly more complete versions.

Feature	Prototyping Model (Throwaway)	Evolutionary Process Model
Product use	Prototype often thrown away (or mined for ideas).	Product itself **evolves** over iterations.
Suitability	Highly uncertain requirements.	New technology, long-term evolving products.

2.5 Incremental vs evolutionary process models

- **Evolutionary process model:** Framework activities executed in a **circular** manner; each cycle leads to a more complete version. Emphasizes flexibility and evolution.
- **Incremental model:** Related to evolutionary, but stresses **staged releases** or increments (V1, V2, V3). Each increment adds functionality, providing early operational capability.

---

3. Lifecycle Selection & Other Models

3.1 Selection parameters for life-cycle model / Why requirements are crucial

Parameters:

- Characteristics of **requirements**:
  - Easily understandable, well-defined → **Waterfall**.
  - Frequently changing → **Prototyping or RAD**.
- **User participation**:
  - High → RAD, Prototyping.
- **Project risk**:
  - High → Spiral.
- **Project type**:
  - Enhancement of existing system → **Iterative enhancement** model.
- **Status of development team**:
  - Experience and capability influence feasible models.

Why requirements matter:

- If requirements are clear and stable, a **sequential** model works.
- If they change often, **iterative** models (Prototyping, RAD, Spiral) are needed to avoid huge rework.

3.2 RAD model – KEEPING YOUR ORIGINAL CONTENT

Describe the Rapid Application Development (RAD) model. Discuss each phase in detail.

RAD stands for **Rapid Application Development**. It is an **iterative**, user-interactive model proposed by IBM that emphasizes **fast development** and **strong customer participation**. It is **highly suitable when requirements are frequently changing**.

Typical RAD phases (from your original file/notes):

1. **Requirements Planning (or Requirements Gathering & Analysis):**
    - Developers and users jointly identify the **business problems**, project **scope**, objectives, and constraints.
    - Requirements are gathered quickly, often at a high level, but with clear agreement on goals.
2. **User Design (Business/Data/Process Modeling):**
    - Users and developers work together to model:
        - **Business data**,
        - **Processes**,
        - **User interfaces**.
    - Prototypes of screens/reports are built.
    - This phase involves **intensive user interaction** and **repeated refinement** based on feedback.
3. **Rapid Construction:**
    - Actual system components are built using **code generators, CASE tools, and reusable components**.
    - Prototypes are turned into working modules.
    - Includes **coding, unit testing, and integration** for the RAD components.
4. **Cutover (Implementation):**
    - Final testing, data conversion, user training, and system changeover.
    - The system is moved into the production environment for actual use.

Suitability:

- Requirements are changing but **time-to-market is critical**.
- System can be **modularized** into RAD components.
- **User participation** is available throughout.
- Adequate **human resources** and tools are available.

3.3 Explain Agile and RAD life-cycle models

- **RAD Model:** As above – iterative, tool-supported, user-intensive, good for frequently changing requirements and quick delivery.
- **Agile Model:**
  - Combines a **philosophy** and **development guidelines**.
  - Philosophy: customer satisfaction, early incremental delivery, small motivated teams, minimal documentation, embracing change.
  - Guidelines: stress **working software** over heavy up-front analysis/design; frequent delivery and feedback.

3.4 Features of Waterfall; disadvantages and better SDLCs

Features:

- **Linear process flow**: Communication → Planning → Modeling → Construction → Deployment.
- Best when **requirements are easily understandable and defined**.

Disadvantages & appropriate alternatives:

Disadvantage	Why	Alternative model
Inability to accommodate change	Assumes all requirements known up front.	Prototyping, RAD, Spiral.
Delay in feedback	Delivers product only at the end.	Incremental/Evolutionary/Agile.

3.5 Why incremental is effective for business systems; less for real-time

- **Business systems:**
  - Incremental development allows **quick delivery** of usable parts.
  - Emphasizes flexibility and speed; supports requirement evolution.
- **Less appropriate for real-time systems engineering:**
  - Real-time systems often need **tight timing** and **full integration** to test properly.
  - Partial increments may not reveal timing and concurrency issues until very late.

3.6 Compare Waterfall and Spiral

Feature	Waterfall	Spiral

Feature	Waterfall	Spiral
Flow	Linear, sequential.	Evolutionary, circular.
Requirements	Must be known upfront; poor with change.	Handles unclear/volatile requirements well.
Risk	No explicit risk phase.	Explicit, central **risk management**.
Delivery	Single final delivery.	Multiple intermediate evolutionary deliveries.

We've now updated the hybrid so RAD is exactly at the detail level you wanted from the original file, and the rest keeps the improved structure.

To check your understanding: if an 8-mark question asks "Explain the RAD model and list situations where it is suitable," which **3–4 headings** would you write in your exam answer?

Unit 3: Requirements Engineering

Requirements Concepts

1. Explain the importance of requirements.

Requirements describe **what** the system will do, not **how** it will do it. The process that creates the requirements is crucial, as they are related to the quality of the product.
The importance of requirements is evident in the negative outcomes when they are poorly defined:

- **Developers** do not know precisely what to build.
- **Customers** do not know what to expect.
- **Validation** criteria are unclear, making it impossible to test if the software works correctly.

2. Distinguish between Functional, Non-Functional, and Domain Requirements with examples.

Requirement Type	Description	Examples / Implications
Functional Requirements	Specify the functions or features the system must perform. They define **product features**.	*Examples:* Login, Calculate GPA, Generate Report. *Focus:* "What" the system does.
Non-Functional Requirements	Stipulate **how well** the software performs its functions. These are quality attributes.	*Examples:* Reliability, Availability, Usability, Efficiency. *Focus:* Constraints on the system.
Domain Requirements	Related to the specific **business environment** or domain constraints.	*Example:* Banking rules, tax laws, or medical regulations that the software must obey.

3. Differentiate between Functional and Non-Functional Requirements.

Feature	Functional Requirements	Non-Functional Requirements
Scope	Defines capabilities and services (Functions).	Defines quality criteria and constraints (Attributes).
Question	Addresses **"WHAT"** the system does.	Addresses **"HOW WELL"** it does it.

Feature	Functional Requirements	Non-Functional Requirements
Examples	User input, data processing, report generation.	Speed, Security, Reliability, Maintainability.

4. Differentiate between User and System Requirements.

- **User Requirements:** Written for the **customer/end-user**. They describe what the user needs in natural language and diagrams.
- **System Requirements:** A detailed **technical breakdown** derived from user requirements. They are written for developers and form the basis of the system design.

5. Give two non-functional requirements and explain their implications to system design.

1. **Reliability:** The probability that software will work without failure for a specified time.
    - *Implication:* Requires design decisions like **error tolerance**, redundancy, and rigorous exception handling.
2. **Efficiency:** The relationship between performance and resource usage.
    - *Implication:* Constrains design choices regarding **data structures, algorithms**, and hardware (e.g., using a Hash Map instead of a List for speed).

SRS Document

6. What is the purpose of an SRS (Software Requirement Specification) document?

The SRS is the final outcome of the requirements phase. It defines the "What" of the system.
Purposes:

1. **Contract:** Serves as a formal agreement between customer and developer.
2. **Black Box Specification:** Describes system behavior without revealing internal design.
3. **Communication:** Ensures developers know what to build and customers know what to expect.

7. Write the desirable characteristics of an SRS document.

A good SRS must be:

1. **Correct:** Accurately reflects the user's needs.
2. **Complete:** Contains all necessary requirements; nothing is missing.

3. **Consistent:** No two requirements contradict each other.
4. **Unambiguous:** Each requirement has only one interpretation.
5. **Traceable:** Each requirement can be linked to code and test cases.
6. **Modifiable:** Easy to change without breaking structure.
7. **Design-Independent:** Describes "what", not "how".

8. Who are the stakeholders of SRS?

- **Users:** The people who will actually use the software.
- **Customers:** The people paying for the software.
- **Developers:** The people building the software.
- **Testers:** The people verifying the software.

9. List the important issues which an SRS must address.

An SRS must comprehensively address:

1. **Functionality:** What the system does.
2. **External Interfaces:** Interaction with hardware, users, and other systems.
3. **Performance:** Speed, response time, throughput.
4. **Attributes:** Reliability, Security, Maintainability.
5. **Design Constraints:** limitations on implementation (e.g., specific OS, language).

Elicitation Techniques

10. What is Requirements Engineering? Describe the steps.

Requirements Engineering (RE) is the process of analyzing, documenting, and validating requirements.
Crucial Steps:

1. **Elicitation:** Gathering requirements from stakeholders (Interviews, etc.).
2. **Analysis:** Refining and modeling requirements to resolve conflicts.
3. **Documentation:** Writing the SRS document.
4. **Review/Validation:** Checking the SRS for errors and quality.

11. What is Requirement Elicitation? Explain two techniques.

Elicitation is the process of gathering requirements. It is the most communication-intensive step.
Techniques:

1. **Interviews:** Direct conversation with stakeholders. Can be **Structured** (pre-set questions) or **Open-ended** (flexible discussion).
2. **Use Case Approach:** Describes system behavior from a user's point of view. Focuses on **Actors** (users) and their **Goals**.

12. Describe FAST (Facilitated Application Specification Technique).

- **Definition:** A team-oriented approach where customers and developers work together to identify the problem and propose elements of the solution.
- **Process:** Participants list system objects, functions, and constraints.
- **Key Rule: No criticism or debate** is allowed during the generation phase to encourage free ideas.
- **Comparison to Brainstorming:** Similar to brainstorming but more structured, with a specific focus on generating software specifications (objects, functions).

13. Explain the Use Case Approach and the Use Case Template.

- **Use Case:** A description of a system's behavior as it responds to a request from a user (Actor).
- **Actor:** An external entity (person or system) interacting with the software.
- **Template Example (Login):**
  - **Actor:** User / Admin.
  - **Pre-condition:** User is on the login page.
  - **Basic Flow:**
    1. User enters username and password.
    2. System validates credentials.
    3. System redirects to dashboard.
  - **Alternative Flow:**
    1. User enters invalid credentials.
    2. System shows error message.
  - **Post-condition:** User is authenticated.

14. Why are requirements hard to elicit?

1. **Ambiguity:** Users don't know exactly what they want.
2. **Volatility:** Requirements change during the project.
3. **Communication Gap:** Developers and users speak different "languages" (technical vs. business).
4. **Conflicting Requirements:** Different stakeholders have different needs.

Validation

15. Differentiate between Requirement Specification and Validation.

Aspect	Requirement Specification	Requirement Validation
Process	Documentation of requirements.	Verification of requirements quality.
Output	**SRS Document**.	**List of Problems** / Error Report.
Goal	To define "What" to build.	To ensure the definition is **Correct, Complete, and Consistent**.

16. What happens when validation reveals an error?

- **Action:** The error is documented in a "List of Problems."
- **Correction:** The **Requirements Engineer** modifies the SRS.
- **Involvement:** The **Stakeholders** (Customer/User) must review the change to ensure the correction actually meets their needs.

Unit 4: Software Design

4.1 Coupling & Cohesion

1. What do you understand by coupling and cohesion? Explain their types.

Coupling: The degree of interdependence between software modules. High coupling means changes in one module cause ripple effects in others.

- **Goal: Minimize** coupling (Low Coupling).
  Cohesion: The degree to which elements *inside* a module belong together. High cohesion means a module performs one focused task.
- **Goal: Maximize** cohesion (High Cohesion).

2. Types of Coupling (Best to Worst)

Type	Description	Quality
Data Coupling	Modules pass only necessary data items as parameters. No global data is shared.	**Best**
Stamp Coupling	Modules pass composite data structures (e.g., records) but may not use all fields.	Good
Control Coupling	One module controls the logic of another (e.g., passing a "flag" to switch execution paths).	Moderate
Common Coupling	Multiple modules access the same **global data** area. Hard to track errors.	Bad
Content Coupling	One module directly modifies or accesses the internal data/code of another. Violates encapsulation.	**Worst**

3. Types of Cohesion (Best to Worst)

Type	Description	Quality
Functional	The module performs exactly **one well-defined function**.	**Best**
Sequential	Output of one task becomes the input of the next task in the module.	High

Type	Description	Quality
Communicational	Tasks operate on the same input data or contribute to the same output data.	High
Procedural	Tasks are grouped because they follow a specific execution order (algorithm).	Moderate
Temporal	Tasks are grouped because they happen at the **same time** (e.g., initialization).	Moderate
Logical	Tasks are grouped by category (e.g., "all input routines") and selected by a flag.	Low
Coincidental	Tasks are grouped arbitrarily with no meaningful relationship.	**Worst**

4. Why is it desirable to have Low Coupling and High Cohesion?

- **Maintainability:** Changes are localized. You can fix one module without breaking another.
- **Testability:** Modules can be tested in isolation (Unit Testing).
- **Reusability:** A highly cohesive module (e.g., "Calculate Tax") can be reused in other programs easily because it doesn't depend on external globals (Low Coupling).

5. Can a system ever be completely "decoupled"?

No. If modules were completely decoupled (zero coupling), they could not talk to each other, and the system would do nothing. The goal is **loose** coupling (using defined interfaces/parameters), not **no** coupling.

4.2 Design Approaches

6. Top-Down vs. Bottom-Up Approaches

Feature	Top-Down Approach	Bottom-Up Approach
Strategy	Starts with the main system function and breaks it down into sub-functions (Stepwise Refinement).	Starts with basic components (modules) and combines them to build the system.
Decisions	Major architectural decisions are made first; details come last.	Low-level details are decided first; architecture emerges later.
Testing	Main control loop is tested first; stubs are used for missing modules.	Unit testing is done first; drivers are used to call the modules.

Feature	Top-Down Approach	Bottom-Up Approach
Best For	Clear requirements; clarifying overall structure.	Reusing existing code; testing critical low-level algorithms early.

7. Object-Oriented (OO) vs. Function-Oriented (FO) Design

Feature	Function-Oriented Design	Object-Oriented Design
Focus	Focuses on **processes** (functions) transforming input to output.	Focuses on **entities** (objects) that hold data and behavior.
Structure	System is a hierarchy of functions (Top-Down).	System is a collection of interacting objects.
Data Handling	Data is often global or passed around; separate from functions.	Data is **encapsulated** inside objects (Information Hiding).
Reuse	Harder to reuse functions due to global dependencies.	Easier to reuse Classes via **Inheritance**.

8. What is Modularity? State its desirable properties.

Modularity is the separation of the system into distinct, manageable parts (modules).
Desirable Properties:

1. **Decomposability:** Problem can be broken into smaller sub-problems.
2. **Composability:** Modules can be assembled into new systems.
3. **Understandability:** A module can be understood in isolation.
4. **Continuity:** Small changes affect only a few modules.
5. **Protection:** Errors in one module don't crash the others.

4.3 Data Modeling & Rules

9. Data Flow Diagram (DFD) vs. Flowchart

Aspect	Data Flow Diagram (DFD)	Flowchart
Focus	**Flow of Data** (Where does data go?).	**Flow of Control** (What happens next?).
Logic	**Suppresses** logic (No loops/ifs shown).	**Explicitly shows** logic (Diamond boxes for decisions).
Timing	No concept of timing or sequence.	strictly sequential.

10. Rules for Drawing a DFD

1. **Unique Names:** Every process and data store must have a unique name.
2. **No Logic:** Do not show `if-else` or `loops`. Show only data movement.
3. **Conservation of Data:** A process cannot create output data that wasn't derived from its input (no "magic" data).
4. **Balancing:** Inputs/Outputs of a child DFD must match the parent DFD.

11. What is a Data Dictionary?

A **Data Dictionary** is a repository that defines all data elements used in the system (in DFDs). It ensures everyone uses the same definitions.

- **Contents:**
  - **Name:** (e.g., `Student_ID`)
  - **Alias:** (e.g., `Roll_No`)
  - **Type:** (e.g., Integer, 8 digits)
  - **Description:** (e.g., "Unique identifier for a student")

12. Logical Constraints in ER Diagrams

- **Cardinality:** How many instances relate? (1:1, 1:N, M:N).
- **Participation:** Is the relationship mandatory or optional? (e.g., A specific employee *must* have a department vs. a department *may* have employees).
- **Degree:** Unary (recursive), Binary (2 entities), Ternary (3 entities).

Unit 5: UML & Modeling

1. Name the various diagrams used in UML. Also specify the purpose of drawing each diagram.

UML provides 13 different diagrams, categorized into **Structural** (static) and **Behavioral** (dynamic).

Diagram Name	Category	Purpose (Specification)
Class Diagram	Structural	Defines the **static structure** of the system: classes, attributes, methods, and relationships (inheritance, association). It acts as a blueprint for objects.
Use Case Diagram	Behavioral	Visualizes **functional requirements** from a user's perspective. Shows **Actors** (users) and their interactions with system features (Use Cases).
Sequence Diagram	Behavioral	Depicts object interactions **over time**. Shows the order of messages passed between objects to complete a specific task.
Communication Diagram	Behavioral	Focuses on **object relationships** and organization. Unlike Sequence diagrams, it emphasizes *who* talks to *whom* rather than *when*.
Activity Diagram	Behavioral	Models the **flow of control** (workflow) of a system. Similar to a flowchart, it shows actions, decisions, and parallel processing.
State Diagram	Behavioral	Models the **lifecycle of a single object**. Shows the different states an object goes through (e.g., Order Placed -> Shipped -> Delivered) in response to events.
Deployment Diagram	Structural	Maps software artifacts (code, DB) to **physical hardware** (servers, devices). Shows the physical architecture.

2. State the properties of a Class Diagram with an example.

A Class Diagram acts as a template for creating objects.
Properties/Components:

1. **Class Name:** The unique identifier (e.g., `Student`).
2. **Attributes:** Data variables held by the class.
    - *Visibility:* `+` (Public), `-` (Private), `#` (Protected).
    - *Example:* `- studentID: int`.
3. **Operations:** Methods or functions the class can perform.
    - *Example:* `+ calculateGPA(): float`.

4. **Relationships:** Lines connecting classes (Association, Inheritance, etc.).

3. How do you represent a class in UML?

A class is represented as a **rectangle divided into three compartments**:

1. **Top:** Class Name (Centered, Bold).
2. **Middle:** Attributes (List of variables).
3. **Bottom:** Operations (List of methods).

Example Notation:

```
+-----------------------+
|        Student        |  <-- Name
+-----------------------+
| - name: String        |  <-- Attributes
| - rollNo: int         |
+-----------------------+
| + register(): void    |  <-- Operations
| + getResults(): void  |
+-----------------------+
```

4. What is a Use Case Diagram? Explain with an example.

A **Use Case Diagram** captures the **functional requirements** of a system. It shows "Who" (Actors) does "What" (Use Cases).
Components:

- **Actor:** An external entity (User, Admin, System) represented by a stick figure.
- **Use Case:** A specific function or goal represented by an oval (e.g., "Login").
- **System Boundary:** A box defining the scope of the system.

Example (Result Management System):

- **Actors:** Data Entry Operator, Student.
- **Use Cases:** `Login`, `Enter Marks`, `Generate Report`, `Check Result`.
- *Scenario:* The "Data Entry Operator" (Actor) connects to the "Enter Marks" (Use Case) via a line.

5. Explain the difference between Association, Aggregation, and Generalization.

Relationship	Symbol	Description

Relationship	Symbol	Description
Generalization	Solid line with **hollow triangle** arrow pointing to parent.	**"Is-A" Relationship** (Inheritance). *Example:* `Car` is a `Vehicle`. Subclass inherits from Superclass.
Association	Solid line (can have arrow).	**"Uses" Relationship**. Two classes interact but are independent. *Example:* `Teacher` teaches `Student`.
Aggregation	Solid line with **hollow diamond** at the container end.	**"Has-A" Relationship** (Weak ownership). Child can exist without Parent. *Example:* `Classroom` has `Students`. If Classroom is deleted, Students still exist.
Composition	Solid line with **filled diamond**.	**Strong "Has-A"**. Child dies if Parent dies. *Example:* `House` has `Room`. If House is destroyed, Room is gone.

6. Difference between Sequence Diagram and Communication (Collaboration) Diagram.

Feature	Sequence Diagram	Communication Diagram
Focus	**Time Ordering** of messages.	**Structural Relationships** between objects.
Axis	Vertical axis represents **Time**.	No time axis; uses numbered labels (1, 1.1, 2) for order.
Usage	Best for seeing the flow of logic step-by-step.	Best for seeing which objects talk to each other.
Visual	Lifelines (vertical dashed lines) drop down.	Network of objects connected by links.

7. How do you represent "if" and "while" in a Sequence Diagram?

We use **Interaction Frames** (rectangular boxes):

- **If / Else:** Use an `alt` (Alternative) frame. The box is split into two sections (e.g., `[valid login]` top half, `[invalid]` bottom half).
- **If (Single):** Use an `opt` (Optional) frame. Messages inside run only if the condition is true.
- **While Loop:** Use a `loop` frame. The condition (e.g., `[for each item]`) is written at the top.

8. What is the difference between Sequential Substates and Concurrent Substates?

Feature	Sequential Substates	Concurrent Substates
Definition	Substates occur **one at a time** in a sequence.	Substates occur **in parallel** (simultaneously).
Active State	Only **one** substate is active.	**Multiple** substates (one per region) are active.
Transition	Triggered by events moving from A to B.	Usually enter/exit a composite state together.
Visual	Connected by arrows in a single region.	Separated by **dashed lines** (swimlanes) inside the state.
Example	ATM: Reading Card -> Verifying PIN.	Car: Engine Running

9. How do you represent a node in a Deployment Diagram?

- **Representation:** A **3D Box** (Cube).
- **Information on Node:**
  - **Node Name:** (e.g., `Database Server`).
  - **Stereotype:** (e.g., `<<device>>`).
  - **Artifacts:** The actual files deployed inside (e.g., `student_db.sql`, `app.exe`).

Unit 8: Software Maintenance

1. Concepts

1.1 What is Software Maintenance and why is it needed?

Definition: Software Maintenance is the set of activities required to keep a software system operational and useful after it has been delivered to the customer.
Need for Maintenance:

1. **To correct errors** (bugs) that were not discovered during testing.
2. **To adapt** the software to changes in its environment (new OS, hardware, or regulations).
3. **To enhance** the software by adding new features requested by stakeholders.
4. **To prevent** future problems by restructuring code (Preventive maintenance).

- *Note:* Software does not "wear out," but it deteriorates due to change. Maintenance arrests this deterioration.

1.2 Categories of Software Maintenance (The 4 Types)

Type	Purpose	Trigger
Corrective	To fix bugs/defects.	User reports a crash or error.
Adaptive	To move software to a new environment.	OS upgrade, new hardware, new database.
Perfective	To improve performance or add features.	User requests a new report format or faster speed.
Preventive	To improve maintainability/future-proof.	Code refactoring, updating documentation.

1.3 Describe the Iterative Enhancement Model.

Definition: A maintenance model that treats changes as **iterative cycles**. It assumes the software has good documentation.
The 3 Stages:

1. **Analysis:** Analyze the existing system and the requested change to understand the impact.

2. **Classification:** Classify the change (Corrective, Adaptive, etc.) and classify the affected modules.
3. **Implementation:** Implement the change in a cycle: **Design -> Coding -> Testing**.
   Benefit: It controls complexity by handling maintenance in manageable "chunks" (iterations) rather than one chaotic fix.

1.4 Describe the Taute Maintenance Model. What are its phases?

The **Taute Model** is a classic cyclical maintenance model with **8 Phases**:

1. **Change Request:** Customer submits a formal request for modification.
2. **Estimate:** Maintenance team estimates time/effort and does **Impact Analysis**.
3. **Schedule:** Change is assigned to a specific release schedule.
4. **Programming:** Source code is modified to implement the change.
5. **Test:** Regression testing is performed to ensure no new bugs were introduced.
6. **Documentation:** System and user manuals are updated to reflect the change.
7. **Release:** The updated software is delivered to the customer.
8. **Operation:** Software enters normal use until the next Change Request arrives.

---

2. Reverse Engineering (RE)

2.1 What is Reverse Engineering? Scope and Tasks.

Definition: The process of analyzing a subject system to identify its components and their relationships, and to create representations of the system at a **higher level of abstraction** (e.g., creating Design from Code).
Scope & Tasks:

1. **Understanding:** Gaining knowledge about the legacy system's logic.
2. **Documentation:** Generating missing or outdated documents (specs, diagrams).
3. **Recovery:** Recovering design and requirements from source code.
4. **Redocumentation:** Creating new, readable documentation.

2.2 Levels of Reverse Engineering (Abstraction Levels)

Reverse engineering can extract information at three different levels of abstraction:

Level	Name	Description
Level 1	**Implementation Abstraction**	(Lowest Level) Extracting details about the **source code**, syntax, and local data structures. Recovering *how* it was coded.

Level	Name	Description
Level 2	**Design Abstraction**	(Middle Level) Recovering the **structure** of the system: modules, interfaces, and data flow (DFDs, Structure Charts). Recovering *how* it was designed.
Level 3	**Specification/Analysis Abstraction**	(Highest Level) Recovering the **business rules** and domain requirements. Recovering *what* the system actually does conceptually.

Unit 7: Verification, Validation & Testing

1. Verification and Validation (V&V)

1.1 Differentiate between Verification and Validation. When are they performed?

Feature	Verification (Checking the Process)	Validation (Checking the Product)
Primary Question	"Are we building the product **right**?"	"Are we building the **right** product?"
Focus	Checks if software conforms to **specifications**.	Checks if software conforms to **user needs**.
Methodology	Reviews, Walkthroughs, Inspections. (Static)	System Testing, User Acceptance Testing. (Dynamic)
Timing	Performed **throughout** the lifecycle.	Performed at the **end** (Testing/Delivery).

2. Testing Concepts & Methodologies

2.1 White Box vs. Black Box Testing

Feature	White Box Testing (Structural)	Black Box Testing (Functional)
Definition	Testing based on internal code structure.	Testing based on external specifications (inputs/outputs).
Visibility	Tester **knows** the internal code/logic.	Tester treats system as a **Black Box** (unknown logic).
Techniques	Path Testing, Cyclomatic Complexity.	Equivalence Class Partitioning, Boundary Value Analysis.
Test Cases	Harder to design (needs flow graph).	**Easier** to design (based on requirements).

2.2 Significance of Independent Paths

- **Definition:** An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- **Significance:** In **Basis Path Testing** (White Box), executing all independent paths guarantees that every statement in the program has been executed at least once. The number of independent paths is given by **Cyclomatic Complexity** $V(G) = E - N + 2$.
- **Necessity of Tools:** Yes. As program size increases (e.g., >100 lines), the number of paths grows exponentially (10^{14} in complex loops). Manual calculation is impossible; automated tools are required.

2.3 Boundary Value Analysis (BVA) vs. Robustness Testing

- **Boundary Value Analysis:** A Black Box technique that focuses on the **edges** of the input domain (Min, Max, Min+1, Max-1).
  - *Test Cases:* $4n + 1$ (where n = number of variables).
- **Robustness Testing:** An extension of BVA that tests **invalid** values just *outside* the boundary (Min-1, Max+1).
  - *Goal:* To check if the system crashes or handles errors gracefully.
  - *Test Cases:* $6n + 1$.

2.4 Does a Fault necessarily lead to Failure?

No.

- **Fault (Bug):** A defect in the code (e.g., `if (x > 100)` should be `if (x >= 100)`).
- **Failure:** The visible error produced during execution.
- **Reasoning:** A fault only causes a failure if the code is executed **and** the specific input triggers the bug. If `x` is never exactly 100 during testing, the fault lies dormant and no failure occurs.

3. Testing Tools & Stakeholders

3.1 Popular Software Testing Tools

1. **Coverage Analyzers:** Measure how much code is exercised (e.g., Statement/Branch coverage).
2. **Test Data Generators:** Automatically create input data for testing.
3. **Output Comparators:** Automatically compare actual output vs. expected output.
4. **Test Harnesses:** Drivers/Stubs used to test modules in isolation (Unit Testing).

3.2 Stakeholders in Testing

1. **Developers:** Perform Unit Testing.

2. **Independent Test Group (SQA):** Perform Integration/System Testing.
3. **Customers/Users:** Perform Acceptance/Beta Testing.
4. **Project Managers:** Track testing progress and quality metrics.

3.3 Automation Challenges for SQA

1. **Complexity:** Huge number of device/platform combinations (especially mobile).
2. **Reproducibility:** Hard to reproduce intermittent errors (timing/network issues).
3. **Semantic Errors:** Tools can check syntax/crashes but can't check if content is "offensive" or "misleading."
4. **High Variability:** Network speeds and server loads vary unpredictably.

4. Alpha vs. Beta Testing

Feature	Alpha Testing	Beta Testing
Location	**Developer's Site** (Internal).	**User's Site** (External/Real World).
Testers	Internal Employees/QA Team.	Real Customers/End Users.
Environment	Controlled/Simulated.	Uncontrolled/Live.
Purpose	Find bugs before releasing to Beta.	Final validation/feedback before public launch.

5. Definitions

5.1 Test Case vs. Test Suite

- **Test Case:** A single set of **Inputs**, **Execution Conditions**, and **Expected Results** developed for a specific objective (e.g., "Verify Login with valid password").
- **Test Suite:** A **collection of test cases** grouped together for execution (e.g., "Regression Test Suite").

5.2 Importance of Regression Test Selection

- **Definition:** Re-running tests to ensure recent code changes didn't break existing features.
- **Selection:** Instead of re-running *all* 10,000 tests (too slow), we select only:
    1. Tests for the **modified module**.
    2. Tests for modules that **depend on** the modified module.
- **Example:** If you change the "Interest Calculation" module, you must re-test "Interest" and "Monthly Statement" (which uses Interest), but you can skip "Address Change" (unrelated).

Unit 9: Software Quality & Standards

CMM & ISO

1. Discuss the relative merits of ISO-9001 vs. SEI CMM. Why is CMM often "better"?

- **ISO 9001 (Quality Assurance System):**
  - **Scope:** Generic. Applies to *any* industry (manufacturing, service, software).
  - **Focus:** Certifies that a company has a **Quality System** in place (procedures, manuals, audits).
  - **Goal:** Customer satisfaction by meeting specifications.
  - **Merit:** Widely recognized internationally for business contracts.
- **SEI CMM (Process Improvement):**
  - **Scope:** Specific to the **Software Industry**.
  - **Focus:** Measures the **maturity** of the software process itself.
  - **Goal:** Continuous process improvement (moving from "Chaos" to "Optimized").
  - **Why CMM is "Better":** ISO 9001 is a "pass/fail" certification (you have a system or you don't). CMM provides a **roadmap** (Levels 1-5) for *how* to improve. CMM focuses on **quantitative measurement** of the process, which ISO 9001 often lacks.

2. Shortcomings of ISO 9001 for the Software Industry

1. **Generic Nature:** It was originally designed for manufacturing; applying its rigid clauses to flexible software development can be difficult.
2. **Documentation Heavy:** It often leads to excessive paperwork ("bureaucracy") rather than better software code.
3. **Focus on Compliance over Quality:** Companies may focus on "passing the audit" rather than actually improving the software product.
4. **Rigidity:** It can limit improvisation and adaptability, which are crucial for modern Agile software projects.

3. Explain the CMM Maturity Levels (Key Process Areas).

The CMM has **5 Maturity Levels**. A company must satisfy all Key Process Areas (KPAs) of a level to move up.

Level	Name	Characteristics	Key Process Areas (KPAs)
1	**Initial**	**Ad-hoc / Chaotic.** Success depends on individual heroics. No defined process.	None.
2	**Repeatable**	**Basic Project Management.** Can repeat success on similar projects.	Requirements Mgmt, Project Planning, Config Mgmt.
3	**Defined**	**Standardized Process.** Process is documented and standardized across the organization.	Training Program, Peer Reviews, Intergroup Coord.
4	**Managed**	**Quantitative Measurement.** Process and quality are measured and controlled using data.	Quantitative Process Mgmt, Software Quality Mgmt.
5	**Optimizing**	**Continuous Improvement.** Focus on defect prevention and innovation.	Defect Prevention, Tech Change Mgmt.

4. Salient Requirements for ISO 9001 (The 20 Clauses)

To get ISO 9001 certified, a company must address 20 clauses, including:

- **Management Responsibility** (Leadership commitment)
- **Document Control** (Versioning of manuals)
- **Design Control** (Planning and verifying design)
- **Corrective Action** (Fixing problems when they occur)
- **Internal Quality Audits** (Self-checking)
- **Training** (Ensuring staff skills)

Quality Attributes & Models

5. What is Software Quality? Explain its attributes (ISO 9126).

Definition: Software Quality is the degree to which a system meets specified requirements and customer expectations ("Fitness for purpose").
Attributes (ISO 9126):

1. **Functionality:** Does it do what is needed?
2. **Reliability:** Does it run without crashing?
3. **Usability:** Is it easy to learn and use?
4. **Efficiency:** Does it use resources (CPU/RAM) well?
5. **Maintainability:** Is it easy to modify/fix?
6. **Portability:** Can it move to another environment easily?

6. Explain McCall's Software Quality Model.

McCall organizes 11 quality factors into **3 Viewpoints** (Product Lifecycle):

1. **Product Operation** (Using it):
    - *Correctness, Reliability, Efficiency, Integrity, Usability.*
2. **Product Revision** (Changing it):
    - *Maintainability, Flexibility, Testability.*
3. **Product Transition** (Moving it):
    - *Portability, Reusability, Interoperability.*

7. Explain Boehm's Software Quality Model.

Boehm's model is **hierarchical** and focuses on the **User's View** vs. the **Developer's View**.
It has 3 levels:

1. **High-Level Characteristics (Primary Uses):**
    - *As-is Utility:* (Reliability, Efficiency, Usability)
    - *Maintainability:* (Testability, Understandability)
    - *Portability.*
2. **Intermediate Constructs:**
    - 7 quality factors (e.g., Reliability, Efficiency) that contribute to the high-level uses.
3. **Primitive Constructs:**
    - Low-level, measurable attributes (e.g., **Device Independence**, **Accuracy**, **Completeness**, **Consistency**, **Structuredness**).

Difference from McCall: Boehm adds hardware-related factors like **Device Efficiency** and emphasizes the "General Utility" of software more than just its operation.

Configuration Management (SCM)

8. Explain Software Configuration Management (SCM) and its activities.

Definition: SCM is the discipline of managing **change** in software. It controls the evolution of the product.
Key Activities:

1. **Identification:** Identifying all items (code, docs, data) that need managing (SCIs - Software Configuration Items).
2. **Version Control:** Managing different versions (V1.0, V1.1) and ensuring developers don't overwrite each other's work.

3. **Change Control:** A formal process (Change Control Board) to approve/reject requests for changes.
4. **Configuration Auditing:** Verifying that the released software matches the documentation and requirements.
5. **Reporting:** Recording and reporting the status of changes ("Who changed what and when?").

9. Short Note on SCM (Why we need it)

Software changes constantly (bugs, new features). Without SCM, these changes lead to chaos (e.g., "Which version has the fix?", "Who deleted my code?").
Example:

- **Baseline:** A stable version (e.g., V1.0) is saved.
- **Check-out:** Developer A checks out a file to edit.
- **Locking:** SCM prevents Developer B from editing the same file simultaneously to avoid conflicts.
- **Check-in:** Developer A saves the new version (V1.1). SCM records the change history.

Unit 10: Metrics & Estimation

1. Metrics Concepts

1.1 Define software metrics and classify them.

Definition: A quantitative measure of the degree to which a system, component, or process possesses a given attribute.
Classification:

1. **Process Metrics:** Measure the **efficacy of the process** (e.g., Defect Removal Efficiency, Productivity).
2. **Product Metrics:** Measure the **quality of the product** (e.g., Complexity, Size, Reliability).
3. **Project Metrics:** Measure the **status of the project** (e.g., Cost, Schedule variance).

1.2 Define Halstead Software Science Metrics.

Halstead metrics measure **internal complexity** based on counting "Tokens" in the source code.

- **Operators (η_1):** Keywords, math symbols (e.g., `if`, `while`, `+`, `=`).
- **Operands (η_2):** Variables, constants (e.g., `x`, `100`, `"Error"`).

Formulas:

- **Vocabulary (η):** $\eta_1 + \eta_2$ (Unique tokens)
- **Length (N):** $N_1 + N_2$ (Total tokens)
- **Volume (V):** $N \times \log_2(\eta)$ (Size in bits)
- **Effort (E):** $D \times V$ (Mental effort required)
- **Time (T):** $E / 18$ seconds (Time to write program)

1.3 Is Halstead significant in Component-Based Development (CBD)?

No.

- Halstead focuses on **low-level code complexity** (operators/operands).
- CBD relies on **reusing pre-built components** (Black Boxes). You don't see the internal code of a component, so you can't count its operators.
- **Better Metric for CBD:** Function Points (measures functionality, not code).

1.4 What is Cyclomatic Complexity?

Definition: A metric ($V(G)$) that measures the **logical complexity** of a program.
Formula: $V(G) = E - N + 2P$

- E = Edges (flow lines)
- N = Nodes (code blocks)
- P = Connected components (usually 1)
  Purpose: It tells you the **maximum number of independent paths** through the code. This is the *minimum* number of test cases needed for full coverage.

1.5 Shortcomings of LOC (Lines of Code). Does Function Point (FP) overcome them?

Shortcomings of LOC:

1. **Language Dependent:** 10 lines of Python might equal 100 lines of C.
2. **Programmer Dependent:** A verbose programmer writes more LOC for the same feature.
3. **Late Availability:** Can only be measured *after* coding is done.

Does FP Overcome this?
Yes. Function Points measure "Functionality delivered to the user" (e.g., Inputs, Outputs). This is independent of the language used and can be estimated **early** (from requirements).

---

2. Function Points (FP)

2.1 Explain the concept of Function Point. Why is it becoming acceptable?

Concept: Break the system into 5 Functional Units:

1. **External Inputs (EI)**
2. **External Outputs (EO)**
3. **External Inquiries (EQ)**
4. **Internal Logical Files (ILF)**
5. **External Interface Files (EIF)**
   Formula: $FP = \text{Count} \times \text{Weight} \times \text{Complexity Adjustment Factor (CAF)}$.

Why Acceptable?

- **Language Independent:** Works for Java, C++, or SQL equally.
- **Early Estimation:** Can be calculated from the SRS document before a single line of code is written.

2.2 Practical Application of FP Method.

It is used primarily for **Project Planning**:

1. **Estimating Size:** Before coding starts.
2. **Estimating Cost/Effort:** Using historical data (e.g., "It costs $100 per FP").
3. **Measuring Productivity:** "FPs per Month" is a better metric than "Lines of Code per Month."

---

3. COCOMO Theory

3.1 Explain all levels of the COCOMO Model.

COCOMO (Constructive Cost Model) estimates Effort and Time.

1. **Basic COCOMO:** Quick, rough estimate. Uses only **KLOC** (Size).
    - *Formula:* $E = a(KLOC)^b$
    - *Modes:* Organic (Simple), Semi-detached (Medium), Embedded (Complex).
2. **Intermediate COCOMO:** More accurate. Uses KLOC + **15 Cost Drivers** (e.g., Reliability, Team Capability).
    - *Formula:* $E = E_{nominal} \times \text{EAF (Effort Adjustment Factor)}$.
3. **Detailed COCOMO:** Break system into modules and apply Intermediate COCOMO to each module separately.

3.2 How does Intermediate COCOMO provide more accurate estimates?

Basic COCOMO assumes all 10,000-line projects take the same effort.
Intermediate COCOMO realizes that a 10,000-line project requiring **High Reliability** (e.g., flight software) takes more effort than a 10,000-line **Student Project**.

- It calculates the **EAF** (Product of 15 multipliers).
- *Example:* If "Reliability" is Very High, multiplier might be 1.40. This increases the effort estimate by 40%, making it more realistic.

3.3 Why is the sum of μ_p and τ_p not equal to 1?

- **μ_p (Effort):** The breakdown of **Man-hours** across phases. This **must sum to 1** (100% of work).
- **τ_p (Time):** The breakdown of **Schedule** (Calendar months).
    - *Why it might not sum to 1:* Phases often **overlap**. "Coding" might start before "Design" is 100% finished. Therefore, the sum of calendar fractions can exceed 1.0 depending on how parallelism is modeled.

3.4 When does Project Planning start? How is Size estimated?

- **Start:** Planning starts immediately after the **SRS (Requirements)** is finalized.
- **Why Estimate Size?** You cannot estimate Cost ("How much money?") or Schedule ("When will it be done?") without knowing Size ("How big is it?").
- **How:** Using **LOC** (Analogy based) or **Function Points** (Count based).

---

4. Risk Management

4.1 What is Risk? Activities?

Risk: A potential problem that *might* happen and *would* hurt the project.
Activities:

1. **Risk Assessment:** Identifying and Analyzing risks.
2. **Risk Control:** Mitigating and Monitoring risks.

4.2 How do we Assess and Control Risk?

- **Assess:** Calculate **Risk Exposure (RE)**.
    - $RE = \text{Probability} \times \text{Impact}$.
    - *Example:* 10% chance of Server Crash $\times$ $50,000 cost = $5,000 RE.
- **Control (RMMM Plan):**
    1. **Mitigation:** Reduce probability (e.g., Backup server).
    2. **Monitoring:** Watch for symptoms.
    3. **Management:** Plan for if it actually happens (Contingency).

4.3 Is it possible to prioritize risk?

Yes. You prioritize based on **Risk Exposure (RE)**.

- High Probability + High Impact = **Top Priority**.
- Low Probability + Low Impact = **Ignore/Low Priority**.