

Software engineering summary

Slide 1

What is Software?

- Software is more than just program code.
- A program is executable code that performs tasks.
- Software includes the program code, libraries, and documentation.
- Software made for a specific need is called a software product.

What is Engineering?

Engineering means developing products using scientific principles and methods.

What is Software Engineering?

- Software engineering applies engineering principles to develop software products.
- It uses scientific, systematic methods to create efficient and reliable software.
- **IEEE Definition:** Applying a systematic, disciplined, and measurable approach to develop, operate, and maintain software.

Why Software Engineering is Important?

- Large programs are difficult to develop without software engineering principles.
- As software grows, complexity increases exponentially.
- Software engineering reduces complexity using two main techniques:
 - Abstraction
 - Decomposition

What is Abstraction?

- Simplifying a problem by ignoring irrelevant details.
- Focus on only what is important at each step.
- Solve the problem layer by layer, adding details gradually.
- Helps reduce complexity effectively.

What is Decomposition?

- Breaking a complex problem into smaller, manageable parts.
- Each smaller problem is solved independently.
- These solutions are then combined to solve the original problem.
- Good decomposition minimizes interaction between parts to reduce complexity.

Need for Software Engineering

- **Large Software:** Like building a house, large software needs scientific processes.
- **Scalability:** It's easier to build on existing software if proper processes are followed.
- **Cost:** Software development is expensive unless a proper engineering process is used.
- **Dynamic Nature:** Software keeps changing to meet new needs, so it needs a structured approach.
- **Quality Management:** Better development processes lead to better quality software.

Characteristics of Good Software

Good software can be judged based on how well it works, how easy it is to move or update, and how easy it is to maintain:

Operational Characteristics

- **Budget:** Cost-effective.
- **Usability:** Easy to use.
- **Efficiency:** Uses resources well.
- **Correctness:** Works as expected.
- **Functionality:** Provides the required features.
- **Dependability:** Reliable and available when needed.
- **Security:** Protected against threats.
- **Safety:** Does not cause harm.

Transitional Characteristics

- **Portability:** Can run on different platforms.

- **Interoperability:** Can work with other systems.
- **Reusability:** Parts can be reused in other software.
- **Adaptability:** Can adjust to changes easily.

Maintenance Characteristics

- **Modularity:** Divided into separate components for easier management.
- **Maintainability:** Easy to fix and update.
- **Flexibility:** Can accommodate new requirements.
- **Scalability:** Can grow in size or scope without problems.

Slide 2

What is a Life Cycle Model?

- It's a **diagram or description** showing all the activities needed to develop software from start to finish.
- Shows the **order of activities** from **inception** to **retirement** of software.
- Different models arrange these activities differently but all include the basic phases.

Why do we need a Life Cycle Model?

- To develop software **systematically and disciplined**.
- To ensure everyone in the team knows **what to do and when**.
- To set **clear entry and exit criteria** for each phase (you start a phase only when the previous one is complete).
- Helps **project managers monitor progress** effectively.

Different Life Cycle Models

- **Classical Waterfall Model**
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model

- Spiral Model

Classical Waterfall Model

- The **oldest and simplest** SDLC model.
- Works in a **sequential step-by-step** approach.
- Although **theoretical and less practical** alone, other models are **derived from it**.

Phases of Classical Waterfall Model:

1. Feasibility Study
2. Requirements Analysis and Specification
3. Design
4. Coding and Unit Testing
5. Integration and System Testing
6. Maintenance

1. Feasibility Study

- Check if the project is **financially and technically possible**.
- **Understand requirements** by meeting the client and **studying inputs/outputs**.
- **Explore possible solutions** and **estimate cost, resources, and time**.
- **Choose the best feasible solution**.

2. Requirements Analysis and Specification

- **Collect detailed requirements** from users and customers (via interviews, discussions).
- **Resolve any contradictions** or ambiguities in requirements.
- Write a clear **Software Requirements Specification (SRS)** document including:
 - **Functional requirements**
 - **Non-functional requirements**
 - **Implementation goals**

3. Design

- Convert SRS into a blueprint for software implementation.
- Two design approaches:
 - **Traditional Design:** Structured analysis and structured design.
 - **Object-Oriented Design:** Identify objects and relationships, then detail design.

4. Coding and Unit Testing

- Write the actual source code for each module.
- Test each module individually (unit testing) to find bugs early.

5. Integration and System Testing

- Combine modules step-by-step (incremental integration).
- Test the integrated system for errors and functionality.
- Types of system testing:
 - **Alpha Testing:** Performed by developers.
 - **Beta Testing:** Performed by friendly users.
 - **Acceptance Testing:** Performed by the customer to approve or reject the product.

6. Maintenance

- After delivery, fix any undiscovered bugs (**corrective maintenance**).
- Improve or add new features (**perfective maintenance**).
- Adapt software to new environments or platforms (**adaptive maintenance**).

Shortcomings of Classical Waterfall Model

- Assumes no errors occur during phases, which is **unrealistic**.
- Errors found late cause costly rework.
- Not flexible to change once a phase is completed.

Software Development Life Cycle (SDLC) Models

1. Iterative Waterfall Model

- Improvement of Classical Waterfall with feedback loops.
- Errors found later in a phase can be corrected within the same phase.
- Advantage: Early working version helps catch design or function problems early.
- Disadvantage: Suitable only for large projects — small projects are hard to split into increments.

2. Prototyping Model (Toy implementation of system)

- Build a simple, quick version (prototype) of the system with limited features.
- Prototype is often incomplete and less reliable but helps understand requirements.
- Useful when user requirements are unclear or technical issues are unknown.
- Helps users see how input, output, and interface might look.
- The first prototype is often discarded; lessons learned help build the real product.

3. Evolutionary Model (Incremental Model) (successive version model)

- Build a basic working system first and then add features in increments.
- Customers can start using core features early.
- Common in large projects and object-oriented development.
- Advantage: Early partial system delivery; reduces errors as core tested thoroughly.
- Disadvantage: Hard to split software into usable increments that customers accept.

4. Spiral Model

- Combines elements of waterfall and prototyping with a focus on risk management.
- The development process is a spiral with loops; each loop is a phase (e.g., feasibility, design).

- Each loop has 4 parts:
 1. **Objective Setting:** Define goals and risks.
 2. **Risk Assessment & Reduction:** Analyze and reduce risks (e.g., prototype if needed).
 3. **Development & Validation:** Build and test next version.
 4. **Review & Planning:** Review progress and plan next cycle.
- Best for **complex, high-risk projects**.
- **Disadvantage:** Complex and costly, so less used for simple projects.

5. Rapid Application Development (RAD) Model

- Focuses on **very fast development** using reusable components and powerful tools.
- Project divided into small modules, developed in parallel by teams.
- Time frame (time-box) is short, usually **60-90 days**.
- **When to use:**
 - Requirements are well understood.
 - Tight deadlines.
 - Small to medium projects.
 - High user involvement is possible.
- **Advantages:**

Fast delivery, reusable components, customer feedback early, lower cost.
- **Disadvantages:**
 - Needs highly skilled staff and tools.
 - Not good for projects that can't be modularized or very small projects.

6. Agile Model

- Focuses on **flexibility, collaboration, and customer satisfaction**.

- Based on **Agile Manifesto** with four core values:
 - Individuals and interactions over processes and tools.
 - Working software over comprehensive documentation.
 - Customer collaboration over contract negotiation.
 - Responding to change over following a plan.
- Uses **iterative and incremental development** with frequent releases.
- **Customers and developers work closely throughout.**
- **Twelve principles** include early delivery, welcoming change, face-to-face communication, simplicity, self-organizing teams, and continuous improvement.

Agile Process Steps

1. **Requirements Gathering:** Collect and prioritize.
2. **Planning:** Decide what to deliver in each iteration.
3. **Development:** Build software in short cycles.
4. **Testing:** Ensure quality and correctness.
5. **Deployment:** Release to users.
6. **Maintenance:** Keep updating based on feedback.

Advantages of Agile

- Faster delivery and increased customer trust.
- **Adapts quickly to changing requirements.**
- Frequent feedback improves software quality.
- People and collaboration prioritized over rigid processes.
- Encourages good design and continuous improvement.
- **Promotes teamwork** and high morale.

Sure! Here's an easy-to-understand explanation of **Requirements Analysis and Specification** without focusing on exam jargon:

Requirements Analysis and Specification

What is it?

Before starting to build software, it's very important to **understand exactly what the customer wants**. This is done by experienced team members called **system analysts**.

What does the analyst do?

- Talk to the customer and gather all necessary information about the software.
- Study the information carefully to remove any confusion or contradictions.
- Make sure the team understands clearly what the software must do.

Important questions the analyst asks (W/H Questions):

- What is the problem to be solved?
- Why is it important to solve this problem?
- What are possible ways to solve it?
- What exactly will be the input and output of the system?
- What challenges might come up?
- Does the software need to connect with other hardware or software? If yes, how will they exchange data?

The Software Requirements Specification (SRS) Document

This document records **everything the system must do** and important details. It usually has three parts:

1. **Functional requirements:** What the system **should do** (the features and functions).
2. **Non-functional requirements:** How the system **should behave** (like being easy to maintain, portable, user-friendly).
3. **Goals of implementation:** Extra notes about future changes or improvements to keep in mind while building the system.

Functional Requirements — What are they?

- These describe **specific actions or tasks** the system performs.

- For example, a "Search Book" function in a library system:
 - Input: Author's name
 - Output: List of books by that author and their locations.

Documenting Functional Requirements — Example: ATM Withdraw Cash

- The user selects "Withdraw Cash" option.
- User selects the account type.
- User enters the amount to withdraw (e.g., between 100 and 10,000 in multiples of 100).
- The system checks if there is enough balance.
- If yes, it gives the cash and prints a receipt; if no, it shows an error message.

What makes a good SRS Document?

- **Concise:** Clear and to the point, no unnecessary information.
- **Structured:** Organized well, so it's easy to read and update.
- **Black-box view:** Only describes **what** the system does, not **how** it does it.
- **Conceptual integrity:** Easy to understand overall, with consistent ideas.
- **Response to undesired events:** Specifies how the system handles errors or problems.
- **Verifiable:** You should be able to check if the system meets the requirements.

Why is SRS important?

- Without SRS, the software **might not meet what the customer wants.**
- Developers **won't know exactly what to build.**
- Maintenance teams **won't understand how the system works later.**
- User manuals **can't be written** properly without SRS.

Problems with unstructured or missing SRS:

- Hard to understand what the system should do.
- Difficult to make changes.
- Confusing and inconsistent requirements.

- Poor communication among team members and customers.

Slide 5

Decision Tree

- A **decision tree** is like a flowchart that shows how decisions are made step-by-step.
- It uses **branches** (called edges) for different conditions to check.
- The **end points** (called leaf nodes) show what action to take based on the condition results.

Example: Library Membership System (LMS)

There are three options for members:

- **New Member**
- **Renew Membership**
- **Cancel Membership**

New Member:

- Decision: When you choose “new member,” the system asks for personal details (name, phone, address).
- Action: If the details are correct, a new membership record is created and a bill is printed for fees.

Renew Membership:

- Decision: When you choose “renewal,” the system asks for member’s name and membership number to check validity.
- Action: If valid, the expiry date is updated and bill printed; if not, an error message is shown.

Cancel Membership:

- Decision: When you choose “cancel membership,” the system asks for member’s name and number.
- Action: The membership is canceled, refund cheque is printed, and record deleted.

Decision Table

- A **decision table** is a table that organizes complex decision logic in rows and columns.
- The **top rows** list the conditions to check (like “Is member valid?”).
- The **bottom rows** list the actions to perform depending on those conditions.
- Each column in the table is called a **rule**, which means “If these conditions are true, then do these actions.”

Example: Library Membership System (LMS)

- The table shows all possible conditions and what to do for each.
- For example, if the "valid selection" condition is **false**, the system will display an **error message**.
- You can look at any column to know exactly what to do for any combination of conditions.

In simple words:

- A **decision tree** is like a map of choices that shows how you get from questions to answers.
 - A **decision table** is a neat checklist that says, "If this happens, do this."
-

Slide 6

Software Design (Simple Description)

- Software design means turning user requirements into a clear plan that helps programmers write the code.
- Before coding, we have an SRS document that explains what the software should do. Software design makes this more detailed and ready for programming.
- It is the first step in the Software Development Life Cycle (SDLC), moving focus from “what the problem is” to “how to solve it.”

Levels of Software Design

1. Architectural Design:

This is the big picture showing the whole system and how different parts interact.

2. High-level Design:

Breaks the system into smaller subsystems and modules and shows how these modules work together.

3. Detailed Design:

Focuses on the details inside each module, like how they are structured and how they communicate with each other.

Modularization

- Dividing the software into small, independent modules that can work and be tested separately.
- This makes managing the project easier and follows the "divide and conquer" strategy.

Concurrency

- Originally, software ran one step at a time (sequentially).
- **Concurrency** means running multiple parts of the program at the same time (in parallel).
- Designers should identify which modules can run concurrently for better performance.

Coupling and Cohesion

When software is divided into modules, we measure how well modules are organized and interact:

Cohesion: How well the parts inside a module work together.

- High cohesion is good — everything inside a module belongs together and does one task.

- Types of cohesion from worst to best:
 - Coincidental (random grouping)
 - Logical (grouped by logic but different tasks)
 - Temporal (grouped by timing)
 - Procedural (grouped by sequence)
 - Communicational (share same data)
 - Sequential (output of one is input to another)
 - Functional (all parts work toward one function) — best!

Coupling: How much modules depend on each other.

- Low coupling is good — modules are independent.
- Types of coupling from worst to best:
 - Content (one module accesses the inside of another) — worst
 - Common (sharing global data)
 - Control (one controls the flow of another)
 - Stamp (modules share parts of data structures)
 - Data (modules only pass data) — best

Design Verification

- The design produces documents like diagrams and pseudocode before coding.
 - It's very important to check and review the design carefully to find mistakes early.
 - Fixing design problems early saves effort later in testing and implementation.
-

Slide 7

Software Design Strategies (Simple Version)

Software design means turning user needs into a clear plan to create software.

Designers try to find the best way to build the software by breaking the problem down and organizing solutions.

Types of Software Design

1. Structured Design:

- Breaks a problem into smaller parts (modules) that solve smaller problems.
- Uses "divide and conquer" strategy.
- Modules are arranged in a hierarchy and communicate with each other.
- Good design means **high cohesion** (related things grouped together) and **low coupling** (modules don't depend too much on each other).

2. Function-Oriented Design:

- The system is made of many small functions.
- Like structured design, it breaks the system into functions.
- Functions work on inputs and don't depend on the system state too much.
- Uses data flow diagrams (DFD) to show how data moves through the system.

3. Object-Oriented Design (OOD):

- Focuses on **objects** (things/entities) and their characteristics instead of just functions.
- **Class:** Blueprint for objects, defining attributes (data) and methods (actions).
- Key OOD features:
 - **Encapsulation:** Bundling data and methods together, hiding internal details.
 - **Inheritance:** Classes can inherit features from other classes, making reuse easier.
 - **Polymorphism:** Methods with the same name can work differently depending on context.

Design Process Steps (OOD example)

- Identify objects and group similar ones into classes.
- Define relationships and hierarchies between classes.
- Plan the overall structure (framework) of the application.

Design Approaches

Top-Down Design:

- Start with the whole system as one big piece, then break it down into smaller parts step-by-step until you get to basic components.
- Best for new systems with unknown details.

Bottom-Up Design:

- Start with small, basic components and combine them to form bigger parts until the full system is built.
- Useful when building on existing components.

Usually, a mix of both is used.

Analysis vs Design

- **Analysis** focuses on understanding what the system should do (requirements).
- **Design** focuses on how to implement those requirements.
- Different models are used: e.g., data flow diagrams (DFD) for analysis and structure charts for design in function-oriented approaches; UML is used for both in object-oriented approaches.

Comparing Modularity

- Good design has **high cohesion** (modules focused on one task) and **low coupling** (modules don't rely heavily on each other).
- This reduces complexity and improves productivity.

Layered Design

- Modules arranged in layers like a hierarchy (tree structure).
- A module can only call modules in the layer directly below it.
- The top layer manages the layers below; the bottom layer does the actual work.
- Helps with control abstraction and simplifies debugging.

- Important terms:
 - **Superordinate/Subordinate**: Controller module and controlled module.
 - **Visibility**: Which modules can call others (only immediate lower layer).
 - **Fan-out**: How many modules a module controls (should not be too high).
 - **Fan-in**: How many modules call a module (high fan-in means good reuse).

Functional Independence

- Modules **should do one clear task** and **interact very little with others**.
- Benefits:
 - Easier to **isolate errors**.
 - Easier to **reuse modules** in other software.
 - Easier to **understand the system** since modules are independent.

Slide 8

Function-Oriented Software Design (Simple Version)

- Function-oriented design started about 40 years ago and is **still widely used today**.
- It looks at the **system as a black box** that offers services (called **high-level functions**) to users.
- During design, **these high-level functions are broken down step-by-step into more detailed, smaller functions**.
- After breaking down, **each function is turned into a module**, and a **module structure is created**.

SA/SD Design Methodology

- **Structured Analysis (SA):**
 - Converts the Software Requirements Specification (SRS) into a Data Flow Diagram (DFD).
 - Analyzes major tasks (functions) and how data flows between them.
 - Uses top-down decomposition and divide and conquer strategies.
 - Shows results graphically with DFDs.

- **Structured Design (SD):**

Converts the DFD model into a structure chart which shows module relationships.

What is a Data Flow Diagram (DFD)?

- A DFD shows how data moves through a system:
 - Inputs, processes (functions), and outputs.
- Simple to create and understand, helps manage complex problems.
- Focuses only on data flow, not control flow or execution order.
- Differs from flowcharts:
 - DFD: Shows how data moves.
 - Flowchart: Shows control/sequence of execution.

Symbols in DFDs

- **Process (Function):** Circle or bubble. Named for the function it performs.
- **External Entity:** Rectangle. Represents outside users or systems interacting with the software.
- **Data Flow:** Arrow showing direction of data movement between processes or entities.
- **Data Store:** Two parallel lines. Represents stored data (files, databases).
- **Output Symbol:** Used for things like printed reports.

Synchronous vs Asynchronous Operations

- **Synchronous:** Two processes connected directly by data flow operate at the same speed; one waits for the other.

- **Asynchronous:** Processes connected through a data store operate independently; one can produce data before the other consumes it.

Importance of DFD

- Simple and easy to understand.
- Shows system as a hierarchy from simple to detailed functions.
- Helps everyone understand the system clearly.
- Useful beyond software — can show how documents or items flow in an organization.

Data Dictionary

- A list of all data items used in the DFDs (data flows and data stores).
- Provides a common vocabulary for developers so everyone uses consistent terms.
- Helps define data structures and understand their impact on the system.
- Important for clear communication and avoiding confusion in large projects.

Data Definition Symbols (Used in Data Dictionary)

- + : Combines data items (e.g., a + b means a and b together).
- [,] : Select one from the list (e.g., [a,b] means either a or b).
- () : Optional data (e.g., a + (b) means a, or a and b).
- { } : Repetition (e.g., {name}5 means five names, {name}* means zero or more names).
- = : Defines equivalence (e.g., a = b + c means a consists of b and c).
- / ... / : Comments (text inside is ignored, just notes).

Slide 9

Constructing a DFD Model of a System (Simplified)

What is a DFD Model?

- It shows how input data transforms into output data through different functions of the system.
- DFDs are built in layers (levels), from the simplest (most abstract) to detailed.

Levels of DFD

1. Level 0 DFD (Context Diagram):

- Shows the entire system as a single process (bubble).
- Simplest, highest-level view.
- Named after the system (usually a noun, e.g., "Supermarket Software").
- Shows system's interaction with external entities.

2. Level 1 DFD:

- Breaks down the system into 3 to 7 major functions (bubbles).
- Represents key high-level functions from the requirements.
- If more than 7, combine some functions; if fewer than 3, split functions further.

3. Lower-Level DFDs (Level 2, Level 3, ...):

- Each function is broken down into smaller subfunctions (called decomposition or factoring).
- Each bubble typically splits into 3 to 7 sub-bubbles to keep clarity.
- Decompose until the function is simple enough to explain with an algorithm.

Steps to Construct DFDs

• Start with Level 0 (Context Diagram):

Identify the system as one process and its interactions with external entities.

• Build Level 1 DFD:

Identify high-level functions from the requirements and represent each as a bubble.

• Develop Lower Levels:

For each bubble, find subfunctions, their data inputs and outputs, and how data flows between them. Draw these as lower-level DFDs.

- **Number the Bubbles:**

- Level 0 bubble: 0
 - Level 1 bubbles: 0.1, 0.2, 0.3, ...
 - Level 2 bubbles: 0.1.1, 0.1.2, 0.1.3, ...
- Numbering shows hierarchy and relation.

Balancing DFDs

- Ensure inputs and outputs of a parent bubble match the combined inputs and outputs of its child bubbles.
- This keeps the model consistent and accurate.

Example: RMS Calculating Software

- **Context Diagram:**

Shows system taking 3 numbers as input and returning the RMS result.

- **Level 1 DFD:**

Shows 4 major functions:

1. Accept input numbers
2. Validate numbers
3. Calculate RMS
4. Display result

- **Level 2 DFD:**

Break down "Calculate RMS" into:

- Calculate squares of input numbers
- Calculate mean
- Calculate root

Slide 10

Structure Chart: Simple Explanation

What is a Structure Chart?

- A **structure chart** is a graphical representation of the **software architecture**.
- It shows how **modules** (parts of the program) are arranged and how they **depend** on each other (which module calls which).
- It also shows what **data** is passed between modules.
- It focuses on **module structure and interactions**, but **not** on the detailed steps inside each module (procedural details).

Basic Elements of a Structure Chart

1. Rectangular Boxes:

- Each box represents a **module** (a self-contained unit of the software).
- The box is labeled with the module's name.

2. Module Invocation Arrows:

- Arrows connecting modules show **control flow**: the module at the tail calls the module at the head.
- This does not show how many times a module is called or the order of calls.

3. Data Flow Arrows:

- Small arrows next to invocation arrows represent **data passed** from one module to another.
- These arrows are labeled with the data name.

4. Library Modules:

- Represented by a **rectangle with double edges**.
- These are frequently used modules called by many other modules.

5. Selection (Decision):

A **diamond symbol** means a module chooses to call one of several modules based on some condition.

6. Repetition (Loop):

A **loop symbol** (like a curved arrow around an arrow) means some modules are called repeatedly.

Rules and Properties of Structure Charts

- There is **one root module** at the top — this is the main starting module.
- There is at most **one control relationship** between any two modules (if A calls B, then B cannot call A).
- This means the chart forms a **hierarchy** (layers of modules).
- **Lower-level modules do not know** about higher-level modules (principle of abstraction).
- Two higher-level modules **can call the same lower-level module** (shared utility).

Examples of Layered Designs

- **Properly layered design:** Modules are organized in levels with clear parent-child relationships and minimal cross-layer calls.
- **Poorly layered design:** Modules call each other in a messy, cyclic way, making the design confusing and hard to maintain.

Aspect	Flowchart	Structure Chart
Purpose	Shows flow of control inside a program	Shows module hierarchy and data flow between modules
Identification of modules	Difficult to identify modules clearly	Modules are clearly identified as boxes
Data interchange	Not shown	Clearly shown with data flow arrows
Sequence of tasks	Shown explicitly	Sequence is suppressed/abstracted

Transforming DFD into Structure Chart

You can convert a **Data Flow Diagram (DFD)** into a **structure chart** using two approaches:

1. **Transform Analysis**
2. **Transaction Analysis**

When to Use Which Analysis?

- Check the **input data** in the DFD.
- **If all input data is processed similarly** (all inputs flow to the same process bubble), use **Transform Analysis**.
- **If input data is processed differently in multiple ways**, use **Transaction Analysis**.

- Usually, **Transform Analysis** is for simple, lower-level DFDs.
- Each distinct processing way is a **transaction** representing a functional task for users.

Transform Analysis Details

Break the DFD into three parts:

- **Input portion:** Converts physical input to logical data (called **afferent branch**).
- **Processing portion:** The core transformation of data (called **central transform**).
- **Output portion:** Converts logical output data to physical form (called **efferent branch**).

Slide 11

Object Modelling Using UML — Simple Explanation

What is a Model?

- A **model** is a **simplified representation of a system** that **focuses on important aspects** and **ignores unnecessary details**.
- In software, models can be **graphical, textual, mathematical, or code-based**.
- Models help document design and analysis results and assist during development.
- **Graphical models** (like UML diagrams) are popular because they are easy to understand.

Why Do We Need Models?

- Software systems can be very complex.
- Models help **manage this complexity** by breaking down the system into **understandable parts**.

- They can be used for:
 - **Analysis** of the system
 - **Specification** of requirements
 - **Code generation**
 - **Design**
 - **Visualizing and understanding** how the system works
 - **Testing**
- Models serve not just to document but also to derive solutions.

What is UML (Unified Modeling Language)?

- UML is a **standard modeling language** used to visualize, specify, construct, and document software systems.
- It provides a set of **symbols and notations** (rectangles, lines, ellipses, etc.) to create graphical models.
- UML has its own **syntax (rules)** and **semantics (meaning)** for these symbols.
- UML is not a **software development methodology** but a tool to **document and communicate** object-oriented designs.

UML Diagrams and Views

UML provides **9 types of diagrams** that capture **5 different views** of a system. Each view shows the system from a different perspective:

1. User's View:

- Shows what functions the system offers to its users.
- This is a **black-box view** (internal details hidden).
- Focuses on **functionalities** from the user's perspective.

2. Structural View:

- Shows the system's **static structure** — important classes or objects and their relationships.
- Also called the **static model** since it doesn't show changes over time.

3. Behavioral View:

- Captures how objects **interact and behave over time**.
- Shows **dynamic aspects** of the system.

4. Implementation View:

- Shows system components and their **dependencies**.
- Helps understand the software's organization.

5. Environmental View:

Shows how components are mapped onto **hardware devices** or physical systems.

Use Case Model (User's View in UML)

- **Use cases** represent the different ways users interact with the system.
- A simple question to find use cases is:
“What can users do with the system?”
- Example for a Library System:
 - issue-book
 - return-book
 - query-book
 - add-book
 - create-member
- Use cases correspond to **high-level functional requirements**.
- Each use case describes a **transaction or interaction** between the user and the system.
- Use cases **do not reveal internal structure or algorithms** — they focus on **what** the system does, not **how**.
- A use case usually involves a **sequence of interactions/messages** between the user and the system.

Object Modelling Using UML

What is a Model?

- A **model** shows important parts of a system and ignores unimportant details (abstraction).
- Models help in **documenting design and analysis** and guide development.
- Models can be **graphical, textual, mathematical, or code-based**.
- UML (Unified Modeling Language) is mainly a **graphical modeling tool** but often needs text explanation.

Why Use Models?

- Models help to **manage complexity** in software development.
- They are useful for:
 - **Analysis** (understanding the problem)
 - **Specification** (defining requirements)
 - **Code generation**
 - **Design** (planning structure)
 - **Visualization** (understanding how the system works)
 - **Testing**
- Models not only document but help **derive solutions**.
- Different models are made for different purposes.

What is UML?

- UML is a **standard language to visualize, specify, construct, and document** software.
- It uses symbols like rectangles, lines, ellipses.
- UML has rules (syntax) and meanings (semantics).

- UML is not a development method but a way to represent object-oriented designs.

UML Diagrams and Views

UML uses **9 diagram types** to show **5 views** of a system, similar to how a building is viewed from different perspectives:

1. User's View

- Shows what the system does for users (functionalities).
- It is a **black box** — users see only inputs/outputs, not internals.

2. Structural View

- Shows the **static structure**: classes/objects and relationships.
- Called the **static model**.

3. Behavioral View

Shows how objects **interact dynamically** over time.

4. Implementation View

Shows software components and their dependencies.

5. Environmental View

Shows how software components run on hardware.

Use Case Model (User's View)

What are Use Cases?

- Use cases describe **how users interact with the system**.
- To find use cases, ask: “**What can users do with the system?**”
- Example for Library System:
 - issue-book, return-book, query-book, add-book, create-member
- Use cases **represent high-level functional requirements**.
- They **group system behavior into transactions or meaningful user actions**.
- A use case **is a sequence of interactions** between user and system.
- Use cases **do not** show internal workings or algorithms.

Main Line Sequence & Variations

- The **main line sequence** is the normal, most common user-system interaction.
- Example: ATM withdraw cash — enter PIN, select amount, get cash.
- There are also **alternative paths** for exceptions or special conditions (e.g., invalid PIN, insufficient balance).
- A use case is made of **scenarios** or **instances** showing different possible sequences.

Representing Use Cases

- Use cases **are shown with ellipses**, named inside.
- The system boundary is shown **with a rectangle** enclosing all use cases.
- **Actors** (users or external systems) are shown as **stick figures**.
- Lines between actors and use cases show **communication** (who uses what).
- **External systems as actors have special notation** (e.g., «external»).

Text Description of Use Cases

- Each use case ellipse should have a text description.
- Description includes:
 - Mainline sequence
 - Alternative scenarios
 - System responses
 - Exceptions or errors
- Typically written as a conversational flow between user and system.

- Extra info in description may include:
 - Contact persons
 - Actors details
 - Preconditions (system state before use case)
 - Postconditions (system state after completion)
 - Non-functional requirements (e.g., response time)
 - Error conditions related to domain (e.g., invalid input)
 - Sample dialogues (example interactions)
 - User interface needs (forms, screens)
 - Document references

Example: Supermarket Prize Scheme

- Customers register with details like address, phone, license.
- Each customer gets a unique customer number (CN).
- Purchases are credited against CN.
- Yearly prizes for top buyers.
- Use cases: **register-customer**, **register-sales**, **select-winners**.

Sample use case description:

U1: **register-customer**

- Customer selects register option
- System prompts for info
- Customer enters info
- System confirms registration or shows errors (already registered or missing info).

- System responses, exceptional conditions.
- Contact persons involved in discussions.
- Actors and their roles/details.
- Preconditions (system state before use case).
- Postconditions (system state after completion).
- Non-functional requirements (performance, environment).
- Error handling (domain-related only).
- Sample dialogs illustrating use.
- User interface requirements (screens, forms).
- Document references for further info.

Example: Supermarket Prize Scheme

- Customers register with personal details and receive a unique Customer Number (CN).
- Purchases credited to CN.
- Annual rewards for top customers and those crossing a purchase threshold.
- Use cases:
 - register-customer
 - register-sales
 - select-winners

How to Identify Use Cases

- Review **SRS document** or perform actor-based analysis.
- **Identify actors** (roles users play) and their functional needs.
- Example: Library Automation System actors — member, librarian, accountant — each with specific tasks.

Factoring Use Cases

Why Factor Use Cases?

factoring of use cases are required under two situations

- To decompose complex use cases into simpler parts for clarity and manageability.
- To reuse common behavior across multiple use cases.

Relationship	Description	Example
Generalization	One use case is a specialized form of another (inherits behavior).	"Admin login" is a specialized "User login".
Includes	Base use case compulsorily includes behavior of another (common reusable chunk).	"Issue book" and "Renew book" include "Check reservation".
Extends	Optional/conditional behavior extending a base use case at specific extension points.	"Place order" extended by "Apply discount" only if discount applicable.

Includes Example

Create Account use case includes **Validate Email** — email validation always happens during account creation.

Extends Example

- **Place Order** use case can be extended by **Apply Discount** optionally.
- **Update Delivery Address** can extend **Place Order** but also be used independently.

Slide 14

Unified Modeling Language (UML) — Class Diagram

Purpose of Class Diagram:

- Describes the static structure of a system, showing how it is organized rather than its behavior.
- Complex systems may have multiple class diagrams.
- Each diagram shows classes and their relationships.

Classes

Definition:

- A **class** represents a set of objects with common attributes (data) and operations (methods).
- Objects with similar features belong to the same class.

Representation:

- Drawn as a **solid rectangle divided into compartments**.
- Compartments (top to bottom):
 1. **Class name** (mandatory, centered, bold, e.g., `LibraryMember`)
 2. Attributes (optional)
 3. Operations (optional)

Naming Conventions:

- Class names: **Singular nouns**, start with uppercase (e.g., `LibraryMember`).
- Object names: start with lowercase (e.g., `studentMember`).

Attributes

- Named properties representing data stored by an object.
- Format: `attributeName : Type` (e.g., `sensorStatus : Int`)
- Attributes start with lowercase letters.
- Can specify **multiplicity** with brackets, e.g., `sensorStatus[10]` means 10 instances per object.
- Can include initial values or constraints.

Relationships Between Classes

1. Association

- A general relationship where classes are connected.
- Represented by a **line** between classes.

- May include:
 - **Name** near the line to describe the relationship.
 - **Arrowhead** indicating reading direction.
 - **Multiplicity** at each end, e.g., **1**, ***** (many), or ranges like **1..5**.
- Example: “Many books may be borrowed by a LibraryMember, and each book is borrowed by exactly one member.”

2. Aggregation

- A **whole-part relationship** where the whole contains parts.
- The whole **owns and manages** parts but parts can exist independently.
- Represented by an **empty diamond** on the whole side.
- Example: A **document** aggregates **paragraphs**; paragraphs can exist independently.

3. Composition

- A **stronger whole-part relationship** with **lifecycle dependency**.
- **Parts cannot exist independently** of the whole.
- When the whole is destroyed, parts are destroyed too.
- Represented by a **filled (black) diamond** on the whole side.
- Example: An **order** composed of **order items** — deleting the order deletes the items.
- Components are created and destroyed with the whole object.

Feature	Aggregation	Composition
Lifecycle dependency	Parts can exist independently	Parts depend on the whole's lifecycle
Diamond symbol	Empty diamond	Filled diamond
Dynamic addition/removal	Allowed	Not allowed
Ownership	Whole manages parts, but parts exist separately	Whole fully owns parts

4. Inheritance (Generalization)

-
- Models "is-a" relationships, where a subclass inherits from a superclass.
 - Represented by an empty arrow pointing from subclass to superclass.
 - Multiple subclasses can share one superclass.
 - Useful for specialization and abstraction.
 - Example: Some library books are **issuable**, others are for **reference**. Both inherit from the general **Book** class.
-

Slide 15

UML: Additional Diagram Types & Concepts

1. Dependency Relationship

- Represents that one element (the **client**) depends on or uses another (the **supplier**).
- Changes in the supplier might require changes in the client.
- Shown as a dotted arrow from client to supplier.

Example:

In an e-commerce system, **Cart** depends on **Product** because **Cart** uses **Product** objects.

2. Constraints

- Define conditions or rules on elements, e.g., attribute value ranges, operation pre/post-conditions.
- Written in curly braces {}.
- Example: {sorted} on a **Book** class to indicate books are sorted by ISBN.

3. Object Diagram

- Shows a snapshot of objects at a specific moment.

- Represents **instances** of classes (unlike class diagrams which show class templates).
- Objects are shown as **rounded rectangles**.
- Can show just names or include attribute values.

4. Interaction Diagrams

Describe how objects collaborate to realize behavior (e.g., use case execution).

Types:

a) Sequence Diagram

- Objects are displayed at the top with lifelines (vertical dashed lines).
- Time flows from **top to bottom**.
- Messages shown as arrows between lifelines.
- Activation bars (rectangles on lifelines) indicate when an object is active.
- Objects created during the interaction are shown at the point of creation.
- Lifeline ends when an object is destroyed.

b) Collaboration Diagram

- Shows both **structural** (objects and links) and **behavioral** (messages) aspects.
- Links between objects show associations.
- Messages are labeled and numbered to show sequence.
- Emphasizes **relationships among objects** as well as message order.

5. Activity Diagram

- Models **workflow** or activities and their sequencing.
- Similar to flowcharts but supports **parallelism and synchronization**.
- Uses **swim lanes** to show responsibility for activities by different components.
- Useful for describing which part of the system performs which activity.

6. State Chart Diagram (State Machine Diagram)

- Models **state changes** of an object over time.

- Based on **finite state machine (FSM)** theory.
- Elements:
 - **Initial state:** filled circle.
 - **Final state:** filled circle inside a larger circle.
 - **States:** rounded rectangles.
 - **Transitions:** arrows labeled with event and optional guard conditions.
- **Guard:** Boolean condition controlling the transition.
- **Pseudo transitions:** transitions without event or guard, happen automatically.

Example: Order states in an automation system, including transitions from "Rejected" to "End".

If you want, I can also help illustrate these diagrams or provide examples tailored to your projects!

why you slip some topic? give all topic with simple explain

Sure! Here's a **complete and simple explanation** of **all the UML topics** you shared, organized clearly and covering everything step-by-step:

Complete Simple Explanation of UML Topics

1. Dependency Relationship

- One class (client) **depends on** another class (supplier).
- If the supplier changes, the client might also need to change.
- Represented by a **dotted arrow** from client to supplier.
- Example: **Cart** depends on **Product** because Cart uses Product objects.

2. Constraints

- Rules or conditions that must be true for attributes, operations, etc.
- Written inside curly braces **{}**.
- Example: **{sorted}** means books are kept sorted by ISBN.

3. Object Diagram

- Shows a **snapshot of objects** at a moment in time.

- Objects are instances of classes.
- Represented by **rounded rectangles** with object name and class.
- Useful to see actual data and object relationships during execution.

4. Interaction Diagrams

Describe how objects communicate to perform behavior (e.g., use case).

Two types:

- a) **Sequence Diagram**
- b) **Collaboration Diagram**

4a. Sequence Diagram

- Objects appear at top with lifelines (vertical dashed lines).
- Time flows **top to bottom**.
- Messages shown as arrows from sender to receiver.
- Activation bars (rectangles) show when object is active.
- Objects created during interaction are shown when created.
- Lifeline ends when object destroyed.

4b. Collaboration Diagram

- Shows objects and their links (associations).
- Messages are numbered arrows on links showing interaction order.
- Emphasizes **which objects interact and how**.
- Shows structure and behavior together.

5. Activity Diagram

- Shows flow of activities (process steps).
- Similar to flowcharts but supports **parallel activities**.
- Uses **swim lanes** to show who (which component) is responsible for each activity.
- Useful for workflows and parallel processing.

6. State Chart Diagram (State Machine Diagram)

- Models different **states** of an object and **transitions** between states.

- Uses:
 - **Initial state:** filled black circle.
 - **Final state:** filled circle inside bigger circle.
 - **States:** rounded rectangles.
 - **Transitions:** arrows labeled with event and optional guard condition.
 - Guards are conditions that must be true for transition.
 - Pseudo transitions happen automatically without event or guard.
 - Example: Order state changing from "New" → "Processed" → "Rejected".
-

Slide 16

Coding Phase in Software Development

What is Coding?

- Coding means **converting your design into an actual program using a high-level programming language.**
- After coding, you do **unit testing** to check if the code works correctly.
- Programmers **follow a coding standard** (a set of rules) to write code consistently and clearly.

Why Use Coding Standards?

- Gives code a **uniform look** even if many programmers work on it.
- Makes the code **easier to read and understand.**
- Encourages **good programming practices.**

Characteristics of a Good Programming Language

1. **Readability:**

Code should look **somewhat like English, easy to read and self-explanatory.**

2. **Portability:**

Code should work on different machines without many changes.

3. **Generality:**

The language should allow writing many kinds of programs.

4. **Brevity:**

It should let you write programs with fewer lines of code compared to low-level languages.

5. **Error Checking:**

The language should catch many errors during compilation and runtime.

6. **Cost:**

Consider the overall cost of using the language (tools, training, etc.).

7. **Familiar Notation:**

Syntax should be familiar so programmers can learn it quickly.

8. **Quick Translation:**

The language should compile or interpret quickly.

9. **Efficiency:**

The generated code should run efficiently.

10. **Modularity:**

Should support writing programs as modules that can be compiled separately.

11. **Widely Available:**

The language should be available on many systems and platforms.

Coding Standards and Guidelines

Organizations create their own rules to maintain code quality and consistency. Here are common rules:

1. **Limiting Global Variables:**

organization Define what data can be global and what cannot.

2. Headers in Code Files:

Each module/file should have a header with:

- Module name
- Creation date
- Author's name
- Modification history
- Summary of module functions
- Inputs/outputs of functions
- Global variables used or changed

```
/*
Module: SensorReadings
Created: 2025-08-10
Author: A. Developer
History: 2025-08-10: Initial version

Synopsis: Reads sensor data and
converts it to usable values.

Functions:
readSensor(inputPin) -> value
calibrate() -> status

Globals:
SENSOR_SCALE, lastReadTime
*/
```

3. Naming Conventions:

- Global variables start with **Capital letters** (e.g., `GlobalCount`)
- Local variables start with **small letters** (e.g., `counter`)
- Constants are written in **ALL CAPS** (e.g., `MAX_SIZE`)

4. Error Handling:

Standard way of returning errors (e.g., 0 for success, 1 for error).

Coding Guidelines

These help improve the quality and maintainability of code:

1. Avoid Clever but Hard-to-Understand Code:

Code should be simple and clear, not tricky or cryptic.

2. Avoid Obscure Side Effects:

- Side effects are things like modifying a global variable or performing hidden I/O.
- These should be clear and obvious, not hidden inside functions.

3. Don't Reuse Identifiers for Different Purposes:

- Each variable should have one clear purpose and a meaningful name.
- Reusing variables for different tasks can confuse readers and make future changes hard.

4. Well-Documented Code:

There should be about **1 comment line for every 3 lines of code** to explain what the code does.

5. Keep Functions Short:

- Functions should be **no longer than 10 lines** if possible.
- Short functions are easier to understand and less error-prone.

6. Avoid **goto** Statements:

- Using **goto** makes code jump around unpredictably.
 - This leads to unstructured, hard-to-read programs.
-

Slide 17

Coding Phase: Code Review, Clean Room Testing & Software Documentation

1. Code Review

After a module is coded and free from syntax errors, the code review process starts to **improve code quality and reduce errors**. There are two main types:

Code Walkthrough

A review meeting where the author of the code guides peers through the code to explain logic, design, and implementation, aiming to find defects and improve understanding

- Informal review done by a small team (3 to 7 members).
- Team reads the code before the meeting.
- They pick **test cases and simulate the code execution by hand** (manually tracing logic).
- In the meeting, they discuss **errors found** but **not how to fix them**.
- The coder is present to explain if needed.
- Managers should **not attend** to keep the environment friendly and open.

Code Inspection

-
- More formal and thorough than walkthroughs.
 - Focuses on finding common programming mistakes and checking adherence to coding standards.
 - Instead of running the code, reviewers carefully examine the code for errors such as:
 - Use of uninitialized variables.
 - Jumping into loops incorrectly.
 - Infinite (non-terminating) loops.
 - Wrong assignments or incompatible types.
 - Array index out of bounds.
 - Wrong memory allocation or deallocation.
 - Wrong parameters in function calls.
 - Incorrect logical operators or operator precedence.
 - Modifying loop variables improperly.
 - Comparing floating-point variables incorrectly.
 - Good companies keep statistics on common errors to focus inspection efforts.

2. Clean Room Testing

- Developed by IBM.
- A rigorous approach aiming to avoid defects without running unit tests.
- Programmers can only test syntax via the compiler; no other direct testing allowed.
- Relies heavily on code inspections, walkthroughs, and formal verification.
- Named “clean room” like in semiconductor manufacturing where defects are minimized by a clean environment.

Five Key Features of Clean Room Approach:

1. Formal Specification:

Software requirements are formally defined, often using state transition models.

2. Incremental Development:

Software is broken into small increments, developed and **verified separately with customer input**.

3. Structured Programming:

Code **uses limited, well-defined control structures**; development happens through stepwise refinement.

4. Static Verification:

No traditional unit testing; code is verified through inspections and proofs.

5. Statistical Testing:

Integrated software is **tested statistically based on usage profiles** to measure reliability.

3. Software Documentation

Besides the code, **various documents** are produced during software development, which are vital for success and maintenance.

Why Good Documentation Matters:

- Makes the software easier to understand and maintain.
- Helps users operate the system effectively.
- Helps new team members quickly learn about the project.
- Helps managers track project progress effectively.

Types of Software Documentation:

Internal Documentation:

- **Comments** and explanations inside the source code.
- **Module headers** describing purpose, author, date, etc.
- Clear **variable and function names**.
- Code formatting like **indentation**.
- Use of constants and meaningful data types.

External Documentation:

- **User manuals**.
- Software Requirements Specification (**SRS**) documents.
- **Design documents**.

- Test plans and reports.
- Installation manuals.

Good development processes ensure all these documents are created and reviewed properly.

Slide 18

Testing in Software Development

What is Testing?

Testing means giving a program some inputs (called **test cases**) and checking if it behaves as expected. If it doesn't, the problem (called a **failure**) is recorded to fix later.

Important Terms:

- **Failure:** When the program does not behave as expected (caused by an error or bug).
- **Test Case:** A group of three things:
 - Input data (I)
 - The system state when input is given (S)
 - Expected output (O)
- **Test Suite:** A set of all test cases used to test the software.

Aim of Testing

The goal is to find as many defects (bugs) as possible in the software. However, it's impossible to test **every** possible input because inputs can be huge or infinite.

Even so, testing is important because it reduces bugs and increases user confidence in the software.

Verification vs Validation

- **Verification:** Check if each phase of software development meets the requirements of the previous phase. (Are we building the product right?)
- **Validation:** Check if the finished software meets the user requirements and works as expected. (Are we building the right product?)

Designing Test Cases

Testing every input is impossible. So, we design a **test suite** that is:

- Small enough to be practical
- Designed to find as many bugs as possible

Random test cases often miss important bugs. Bigger test suites do not always mean better testing unless they are **carefully designed**.

Example of Test Case Design:

c

```
if (x > y)
    max = x;
else
    max = x; // Bug: Should be max = y here
```

- A test suite with $(x=3, y=2)$ and $(x=2, y=3)$ will catch this bug.
- A bigger test suite without $(x=2, y=3)$ may not detect this bug.

Functional Testing vs Structural Testing

- **Functional Testing (Black-box testing):** Test based on software functions and expected output without looking at code.
- **Structural Testing (White-box testing):** Test based on internal code structure, logic, and paths.

Black-Box Testing Techniques

1. Equivalence Class Partitioning:

- Input data is divided into groups (classes) where all inputs are expected to behave the same way.

- Test one value from each class instead of all values.

Guidelines:

- For input ranges, create 1 valid and 2 invalid classes.
- For discrete values, create valid and invalid classes.

Example:

- For a function calculating square root for inputs 0 to 5000:
 - Negative numbers (invalid class)
 - Numbers between 0 and 5000 (valid class)
 - Numbers greater than 5000 (invalid class)
- Test values: -5, 500, 6000

2. Boundary Value Analysis:

- Errors often happen at the edges of equivalence classes.
- Test inputs at boundaries, just below, and just above.

Example:

- For square root function (0 to 5000), test: 0, -1, 5000, 5001
- For age input field (18 to 56):
 - Valid inputs: 18, 19, 55, 56
 - Invalid inputs: 17, 57

Slide 19

White-Box Testing (WBT)

White-box testing is a type of testing where the internal structure, code, and logic of the program are known and used to design test cases. It is often done during unit testing to check the smallest parts of code.

Types of White-Box Testing Strategies

1. Fault-Based Testing:

Targets finding specific types of faults (bugs).

Example: Mutation testing, where small changes (mutations) are made to the code to check if the tests detect the changes.

2. Coverage-Based Testing:

Focuses on making sure certain parts of the program are executed at least once during testing.

Common types of coverage:

- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage

Coverage-Based Testing Criteria

- The goal is to cover all program elements defined by the testing strategy.
- A test suite is adequate if it covers all elements required by that coverage criterion.
- Testing strategies can be:
 - Stronger: Covers more elements than another.
 - Weaker: Covers fewer elements.
 - Complementary: Cover different elements, neither is stronger.

Control Flow Testing

- Focuses on the execution order of statements in a program using its control structure.
- Uses a Control Flow Graph (CFG) that represents all possible paths through the program.
- CFG consists of:
 - Nodes: Represent actions or statements.
 - Edges: Show flow from one node to another.
 - Decision Nodes: Points where decisions (if-else) are made.
 - Junction Nodes: Points where different paths merge.

Control Flow Graph Example (Voting Eligibility)

- Node 1: Start
- Node 2: Assign age value
- Decision Node: Check if age ≥ 18
- Node 3: Print "Eligible" if true
- Node 4: Print "Not Eligible" if false
- Node 5: Stop

Test cases are designed to cover all paths in this graph.

Statement Coverage Testing

- **Objective:** Execute every statement in the program at least once.
- Measures how many statements run compared to total statements.
- Formula:

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total statements}} \times 100\%$$

Example:

c

```
print(int a, int b) {  
    int sum = a + b;  
    if (sum > 0)  
        print("Positive");  
    else  
        print("Negative");  
}
```

- Test 1: a=5, b=4 → sum=9 → "Positive"
Coverage = 5/7 statements executed ≈ 71%
- Test 2: a=-2, b=-7 → sum=-9 → "Negative"
Coverage = 6/7 statements executed ≈ 85%

Branch Coverage Testing

- Focuses on covering all possible branches (true and false outcomes) of decision points.
- Ensures each condition outcome is tested at least once.
- Branch coverage is slightly different from decision coverage but related.
- Uses Control Flow Graph to identify branches.

Branch Coverage Calculation Example

Code:

```
mathematica
```

```
ReadXReadYIFX+Y>100THENPrint"Large"ENDIFIFX+Y<100THENPrint"Small"ENDIF
```

Control Flow Graph Paths:

- Path 1 (Yes for first IF): A1-B2-C4-D6-E8
Covers edges: 1,2,4,5,6,8
- Path 2 (No for first IF, Yes for second IF): A1-B3-5-D7
Covers edges: 3,7

To cover all branches, both paths must be tested.

Branch Coverage = Number of paths covering all branches = 2

Slide 20

White-Box Testing: Condition Coverage & Multiple Condition Coverage

1. Condition Coverage (also called Predicate Coverage)

- Tests each condition in decision statements to make sure all possible outcomes (true and false) happen at least once.
- Condition coverage tests each condition independently of other conditions.

Difference from Branch Coverage:

- Branch coverage tests if each branch (true or false) of the decision is executed at least once.
- Condition coverage tests if each individual condition inside the decision is true and false at least once.

Example:

cpp

```
int a = 10;
if (a > 0) {
    cout << "a is positive";
}
```

- Branch coverage only requires the condition `a > 0` to run once (true).
- Condition coverage requires both `a > 0` true and false cases to be tested.

Condition Coverage Testing Example:

cpp

```
int num1 = 0;
if(num1 > 0) {
    cout << "valid input";
} else {
    cout << "invalid input";
}
```

Test cases:

Test Case	num1 > 0	Output
1	True	valid input
2	False	invalid input

2. Multiple Condition Coverage (MCC)

- MCC requires testing **all possible combinations** of true/false for multiple conditions inside a compound decision.
- For a decision with n conditions, **2^n test cases** are required.
- This can quickly become impractical when there are many conditions.
- MCC can detect faults that branch or simple condition coverage might miss.

Example of MCC:

If condition is:

scss

(c1 AND c2) OR c3

Test cases must cover all 8 combinations of true/false for **c1**, **c2**, and **c3**.

MCC Fault Example (Detected by MCC, but not branch coverage):

cpp

```
if (temperature > 150 || temperature > 50) {
    setWarningLightOn();
}
```

- Bug: Second condition should be **temperature < 50**.
- Branch coverage with **{temperature=160, temperature=40}** won't detect this.

- MCC testing detects it because it tests all condition combinations.

3. Modified Condition/Decision Coverage (MC/DC)

- MC/DC requires each condition in a decision to **independently affect** the decision's outcome.
- Usually requires **fewer test cases** than MCC.
- Typically, **$n + 1$ test cases** are enough for a decision with n conditions.

Designing MC/DC test cases

Example 1:

For decision:

css

```
if (A AND B)
```

Truth table:

Test Case	A	B	Outcome
1	True	True	True
2	False	True	False
3	True	False	False

Test cases 1, 2, and 3 achieve MC/DC.

Example 2:

For decision:

css

```
if ((A AND B) OR C)
```

Possible MC/DC test sets:

{2,3,4,6} or {2,3,4,7} or {1,2,3,4,5}

(Refer to the truth table to see how these cover all conditions independently.)

Condition Coverage Examples with multiple conditions:

cpp

```
if ((num1 > 0 || num2 < 10))
```

Test cases:

Test Case	num1 > 0	num2 < 10	Output
1	True	Don't care	valid input
2	False	True	valid input
3	False	False	invalid input

cpp

```
if ((num1 > 0) && (num1 + num2 < 15))
```

Test cases:

Test Case num1 > 0 num1 + num2 < 15 Output

1	True	True	valid input
2	True	False	invalid input
3	False	Don't care	invalid input

cpp

```
if ((num1 > 0 || num2 < 10) && (num1 + num2 < 15))
```

Test cases:

Test Case num1 > 0 num2 < 10 num1 + num2 < 15 Output

1	True	Don't care	True	valid input
2	True	Don't care	False	invalid input
3	False	True	True	valid input
4	False	True	False	invalid input
5	False	False	Don't care	invalid input

Slide 21**White-Box Testing: Path Coverage****Path Coverage Definition**

- **Path coverage** means designing test cases so that **every linearly independent path** (or basis path) in a program's Control Flow Graph (CFG) is executed at least once.

- A **linearly independent path** is a path through the program that introduces at least one new edge or node not included in any other path.

Control Flow Graph (CFG)

- A CFG is a **directed graph** where:
 - **Nodes (N)** represent individual program statements.
 - **Edges (E)** represent possible flow of control from one statement to another.
- To create a CFG:
 1. Number each program statement.
 2. Nodes correspond to these numbered statements.
 3. Draw edges between nodes if control can transfer from one statement to the next.
- CFG represents program control flow including:
 - **Sequence:** Simple flow from one statement to the next.
 - **Selection:** Conditional branches (if-else).
 - **Iteration:** Loops (while, for).

Example of CFG Types:

- **Sequence:** Straight line flow from statement to statement.
- **Selection:** Decision nodes with branches for **if** or **else**.
- **Iteration:** Looping edges back to the start of the loop until the exit condition is met.

Path

- A **path** is any sequence of nodes and edges starting from the start node to a terminal node (program exit).
- Programs may have multiple terminal nodes (due to multiple return/exit points).
- **All possible paths** may be infinite in presence of loops.
- **Testing all paths is impractical** due to potential infinity or very large number.

Linearly Independent Paths (Basis Paths)

- A set of paths is **linearly independent** if each path introduces at least one edge not covered by other paths.

- Testing these basis paths ensures good coverage without needing to test all possible paths.

Example Paths:

Path 1: 1 → 2 → 3 → 5 → 6 → 7

Path 2: 1 → 2 → 4 → 5 → 6 → 7

Path 3: 1 → 6 → 7

McCabe's Cyclomatic Complexity Metric

Purpose

- Helps compute the **number of linearly independent paths** for any program.
- Provides an **upper bound** for the number of independent paths to test.
- Does **not** directly find the paths but guides how many paths to test.

Computing Cyclomatic Complexity V(G)

Given a CFG with **N nodes** and **E edges**:

Method 1:

$$V(G) = E - N + 2$$

Example: For CFG with $E=7$ edges and $N=6$ nodes,

$$V(G) = 7 - 6 + 2 = 3.$$

Method 2:

$$V(G) = \text{Number of bounded (enclosed) regions in CFG} + 1$$

- Count enclosed areas in the CFG.
- For example, if CFG has 2 bounded areas,
 $V(G) = 2 + 1 = 3.$

Method 3:

$$V(G) = \text{Number of decision and loop statements} + 1$$

- Count all if-else, switch decisions, and loops.
- Example: If 2 decision/loop statements,
 $V(G) = 2 + 1 = 3.$

Meaning of Cyclomatic Complexity

- Number of linearly independent paths to test.
- Indicates complexity: higher values mean more complex code.

- Useful for estimating minimum number of test cases needed for path coverage.

Steps for Path Coverage Testing

1. Draw Control Flow Graph (CFG) for the program.
 2. Calculate McCabe's Cyclomatic Complexity $V(G)$ to find the minimum number of test cases.
 3. Design test cases to cover each of the linearly independent paths.
 4. Run tests and monitor path coverage using dynamic analysis tools.
 5. Repeat and refine test cases until at least 90% path coverage is achieved.
-

Slide 22

Debugging

What is Debugging?

- The process of identifying the exact location of errors in a program and fixing them.
- It involves tracing, locating, and correcting code defects.

Common Debugging Approaches

1. Brute Force Method

- Insert print statements to observe intermediate values.
- Use symbolic (source code) debuggers with breakpoints and watchpoints for systematic examination.

2. Backtracking

- Trace backward from the point where an error is observed to find its origin.
- Can become complex with large codebases due to multiple backward paths.

3. Cause Elimination Method

- List all possible causes for the error and test each to eliminate until the cause is found.
- Software fault tree analysis is related to this.

4. Program Slicing

- Reduce the search space by focusing only on relevant slices of code that affect a variable at a statement.
- Similar to backtracking but more focused.

Debugging Guidelines

- Have a thorough understanding of the system design before debugging.
- Avoid fixing symptoms instead of the actual errors.
- Beware that fixing one error might introduce others.
- Always perform regression testing after bug fixes to ensure stability.

Integration Testing

Purpose

- To test interfaces between modules after unit testing is complete.
- Ensures correct communication and data passing between modules.

Process

- Modules integrated stepwise according to an integration plan based on module dependencies (structure chart).
- Test after each integration step to isolate errors early.

Integration Strategies

1. Big-Bang Testing

- Integrate all modules simultaneously and test.
- Suitable only for small systems.
- Debugging is difficult due to lack of isolation.

2. Bottom-Up Testing

- Integrate and test lower-level modules first, then move upward.
- Allows testing of low-level critical modules thoroughly.
- Can lead to complexity if many small subsystems exist.

3. Top-Down Testing

- Start testing from the top module and integrate lower-level modules progressively.
- Uses **stubs** to simulate lower modules.
- Easier to write stubs than drivers, but limited by unimplemented lower modules.

4. Mixed (**Sandwiched**) Approach

- Combines top-down and bottom-up.
- Testing can start as soon as any module is ready.
- Uses both stubs and drivers.
- Most commonly used.

System Testing

- Performed after integration testing on the complete system.
- Test cases are based solely on the Software Requirements Specification (SRS).
- Implementation details (procedural or object-oriented) don't affect test design.

Types of System Testing

1. Alpha Testing

Performed by internal testing teams within the development organization.

2. Beta Testing

Conducted by selected external users (friendly customers).

3. Acceptance Testing

Performed by the customer to decide on system acceptance.

Smoke Testing

-
- Preliminary testing to check if the major functionalities work.
 - Conducted before system testing to ensure the software is stable enough for detailed testing.
 - Example: In a library system, test creating/deleting books and member records, loaning/returning books.

Performance Testing

- Focuses on non-functional requirements documented in the SRS.
- Generally black-box testing.

Types of Performance Testing

1. Stress Testing (Endurance Testing)

- Test system behavior under extreme conditions beyond normal capacity.
- Checks handling of abnormal inputs, high data volume, processing time, memory usage, etc.

2. Volume Testing

Tests handling of very large data volumes (e.g., large symbol table in a compiler).

3. Configuration Testing

Tests software on different hardware/software configurations specified in requirements.

4. Compatibility Testing

Verifies system interfaces with external systems work correctly (e.g., database communication).

5. Regression Testing

Ensures that recent changes (bug fixes or enhancements) haven't introduced new bugs.

6. Recovery Testing

Tests system recovery after faults like power loss, device failure, or data loss.

7. Maintenance Testing

Tests diagnostic tools and procedures required for ongoing maintenance.

8. Documentation Testing

Checks completeness and consistency of manuals and documentation.

9. Usability Testing

Tests user interface elements to ensure they meet user requirements and ease of use.

10. Security Testing

Tests software for protection against security threats like hacking, password cracking, penetration testing.

Slide 23

Project Scheduling

Scheduling project tasks is crucial for effective project planning and management. The project manager ensures tasks are completed on time and handles any delays.

Steps in Scheduling

1. **Identify Major Activities:** List all key activities required to complete the project.
2. **Break Down Activities:** Decompose activities into smaller, manageable tasks.
3. **Determine Dependencies:** Find relationships and dependencies among tasks (which tasks must precede others).
4. **Estimate Durations:** Assign time estimates for each task.
5. **Create Activity Network:** Represent tasks, durations, and dependencies visually.
6. **Determine Start and End Dates:** Calculate when each task starts and finishes based on dependencies.
7. **Find Critical Path:** Identify the sequence of tasks that determines the minimum project duration.
8. **Allocate Resources:** Assign people, tools, or equipment to each task.

Work Breakdown Structure (WBS)

- A hierarchical decomposition of the project into smaller components.
- Represents activities, sub-activities, and tasks in a tree form.
- The root is the project name; nodes represent activities broken down into children.
- Decomposition continues until:
 - Tasks are about two weeks of work.
 - Hidden complexities are exposed.
 - Tasks can be assigned as units to developers.
 - Opportunities for software reuse are identified.

Activity Network

- Visualizes tasks, durations, and dependencies.
- Two common representations:
 - **Activity on Node (AoN)**: Activities are nodes; edges represent dependencies.
 - **Activity on Edge (AoE)**: Tasks are edges; nodes represent milestones.
- AoN is more popular due to ease of understanding and modification.

Critical Path Method (CPM)

- Used to find the longest sequence of dependent tasks (critical path) that dictates project duration.
- Key quantities in CPM:
 - **Minimum Time (MT)**: Project duration — maximum of all path durations from start to finish.
 - **Earliest Start (ES)**: Earliest time a task can start; max of earliest finishes of predecessor tasks.
 - **Earliest Finish (EF)**: ES + task duration.
 - **Latest Start (LS)**: Latest time a task can start without delaying project; MT minus max path length from task to finish.
 - **Latest Finish (LF)**: LS + task duration or MT minus max path length.
 - **Slack Time (ST)**: Flexibility in task timing — ST = LS - ES = LF - EF.

- Tasks with zero slack are on the critical path; delays affect the project finish date.

Example Calculations (General Process)

- Calculate **ES** and **EF** forward from start.
- Calculate **LS** and **LF** backward from project completion.
- Calculate **Slack** as difference between latest and earliest timings.

Gantt Charts

- Visual bar charts for project scheduling.
- Vertical axis lists tasks; horizontal bars show duration.
- Bars divided into:
 - **Shaded Part:** Estimated task duration.
 - **Unshaded Part:** Slack or lax time (flexibility).
- Useful for tracking progress and identifying critical tasks visually.

Slide Easy Explain:

| Made using chatgpt By **Masud Rana Mushfiq**