Introduction to Software Engineering

What is Software?

- **Definition**:
  Software is a collection of computer programs, procedures, and associated documentation that enables users to interact with a computer and perform specific tasks.
- **Components of Software**:
  1. **Programs** – Instructions written in programming languages.
  2. **Documentation** – Manuals, help files, technical descriptions.
  3. **Data** – Information used and processed by programs.
- **Difference between Hardware and Software**:
  - **Hardware**: Physical, tangible, wears out with use.
  - **Software**: Logical, intangible, does not wear out but may become obsolete.
- **Examples**:
  - Windows OS, Android, MS Office, Google Chrome, Adobe Photoshop, MATLAB.

Types of Software

1. **System Software**
   - Provides the environment for application execution.
   - Examples:
     - Operating Systems (Windows, Linux, MacOS, Android).
     - Compilers, Interpreters.
     - Device Drivers.
2. **Application Software**
   - User-oriented programs that perform specific tasks.
   - Examples: MS Word, Photoshop, AutoCAD, Banking apps, Web browsers.
3. **Utility Software**
   - Small programs for system maintenance and optimization.
   - Examples: Antivirus software, Disk Defragmenter, WinRAR/7-Zip, Backup tools.
4. **Embedded Software**
   - Built into devices for dedicated control and operation.
   - Examples: Software in microwaves, washing machines, routers, printers.
5. **Web-based Software**
   - Applications running on the internet.
   - Examples: Google Docs, Gmail, Facebook, Amazon, Online Banking.
6. **Artificial Intelligence (AI) Software**

   - Uses ML, neural networks, and reasoning techniques.
   - Examples: Chatbots, Self-driving car software, Recommendation systems.

Characteristics of Software

- **Intangible** – Cannot be physically touched, unlike hardware.
- **Custom-built and Generic** –
  - *Custom-built*: Developed for specific client needs.
  - *Generic*: Developed for general users (MS Word, Photoshop).
- **Engineered not Manufactured** – Software is designed and coded, not assembled in a factory.
- **Does not wear out** – Performance does not degrade with use, but may fail due to design faults (bugs).
- **Evolves over time** – Must adapt to new technologies and requirements.

- **High complexity** – Millions of lines of code, dependencies, and user interactions.
- **Requires maintenance** – Fixing bugs, updating features, adapting to new platforms.

Attributes of Good Software

A good software system should satisfy:

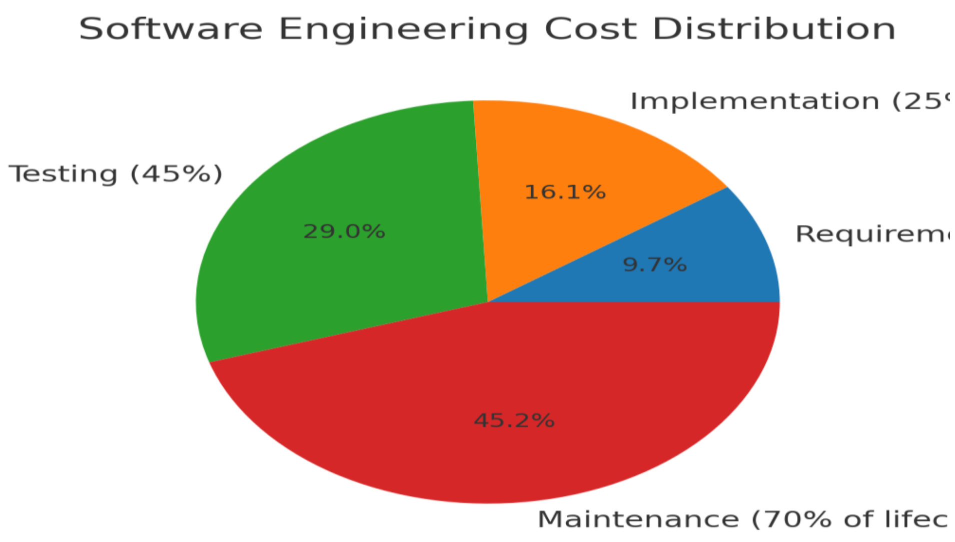
1. **Functionality** – Meets user requirements and performs intended tasks.
2. **Reliability** – Consistent and dependable performance.
3. **Efficiency** – Optimal use of resources (CPU, memory, bandwidth).
4. **Usability** – User-friendly interface, easy to learn and operate.
5. **Maintainability** – Easy to fix, update, and extend features.
6. **Portability** – Ability to run on multiple platforms with minimal changes.
7. **Scalability** – Can handle increasing workload or user base.
8. **Security** – Protects against unauthorized access and data breaches.
9. **Interoperability** – Works with other systems, devices, or software smoothly.

What is Software Engineering?

- **Definition**:
  Software Engineering is the application of engineering principles, methods, and tools to the development and maintenance of high-quality software.
- **Key Features**:
  - Systematic approach.
  - Uses well-defined processes and methodologies.
  - Focuses on quality, cost, and time efficiency.
- **Need for Software Engineering**:
  - To manage **large, complex software systems**.
  - To ensure **quality and reliability**.
  - To **reduce cost and time overruns**.
  - To provide **structured development methodologies** (like Agile, Waterfall, DevOps).

Software Engineering Costs

- **Major cost categories**:
  1. **Development Cost** – Designing, coding, and building the software.
  2. **Testing & Quality Assurance Cost** – Usually **40–50%** of total cost.
  3. **Deployment Cost** – Installation, training, and integration with existing systems.
  4. **Maintenance Cost** – Bug fixing, updates, upgrades (**60–70%** of total cost).
- **Why maintenance is expensive**:
  - Continuous bug fixing.
  - Adapting to new hardware/OS.
  - Adding new features as per user demands.
- **Typical Cost Distribution (in percentage):**
  - Requirements & Design: 10–15%
  - Implementation: 20–25%
  - Testing: 40–50%
  - Maintenance: 60–70%



Key Challenges Facing Software Engineering

1. **Complexity Management**
   - Large projects involve millions of lines of code, multiple teams, and evolving requirements.
2. **Quality Assurance**
   - Ensuring correctness, reliability, and high performance.
3. **Changing Requirements**
   - Clients' needs change frequently → scope creep.
4. **Security & Privacy**
   - Increasing cyber threats, data leaks, ransomware attacks.
5. **Meeting Deadlines & Budgets**
   - Projects often exceed time/cost estimates.
6. **Software Maintenance**
   - Updating, fixing, and enhancing consumes majority of cost.
7. **Scalability**
   - Must support growing number of users, devices, and data sizes.
8. **Integration with New Technologies**
   - Adapting to **AI, Cloud Computing, Big Data, IoT, Blockchain**.
9. **Distributed Development Teams**
   - Global teams face communication & coordination challenges.
10. **Rapid Technological Changes**
    - New frameworks, programming languages, and tools require constant learning.

Systems Engineering vs Software Engineering

Systems Engineering

- **Definition**:
  Systems Engineering is an **interdisciplinary approach** to design, develop, and manage **complex systems** (involving hardware, software, people, processes, and environment) throughout their lifecycle.

- **Key Focus**:
    - Integration of hardware, software, human resources, and processes.
    - Ensures all components work together as a complete system.
    - Deals with *requirements, architecture, design, implementation, testing, deployment, and maintenance* of entire systems.
- **Examples**:
    - Air Traffic Control System
    - Defense Systems
    - Spacecraft Systems (NASA missions)
    - Smart City Infrastructure
- **Phases of Systems Engineering**:
        - Requirements Analysis
        - System Design
        - Implementation (hardware + software)
        - Integration and Testing
        - Deployment and Operations
        - Maintenance and Disposal

Software Engineering

- **Definition**:
  Software Engineering is the application of **engineering principles and systematic methods** to the development and maintenance of **software products**.
- **Key Focus**:
    - Building reliable, maintainable, efficient, and cost-effective software.
    - Manages the **software lifecycle**: requirements → design → coding → testing → deployment → maintenance.
    - Concerned mainly with **programs, data, and documentation**.
- **Examples**:
    - Operating Systems (Windows, Linux)
    - Web Applications (Amazon, Flipkart, Gmail)
    - Mobile Apps (WhatsApp, Instagram)
    - AI Systems (ChatGPT, Recommendation Systems)

Relationship Between Systems & Software Engineering

- Software Engineering is a **subset of Systems Engineering**.
- Every modern system (e.g., aircraft, medical devices, self-driving cars) includes **both hardware and software**.
- **Systems Engineering** ensures all components (hardware + software + people + processes) fit together, while **Software Engineering** ensures the software part is reliable and functional.

Software Development Process Models

A **software process** is a structured set of activities and methods used to **develop, deliver, and maintain software systems**.
It provides a **framework** that defines **what needs to be done, when, how, and by whom** during the software development life cycle (SDLC).

Main Activities in a Software Process

1. **Specification** – Defining what the software should do (requirements analysis).
2. **Design and Implementation** – Designing the system architecture and coding it.
3. **Validation** – Testing and ensuring the software meets requirements.
4. **Evolution (Maintenance)** – Modifying software to adapt to changes (e.g., bug fixes, upgrades).

Importance of a Software Process

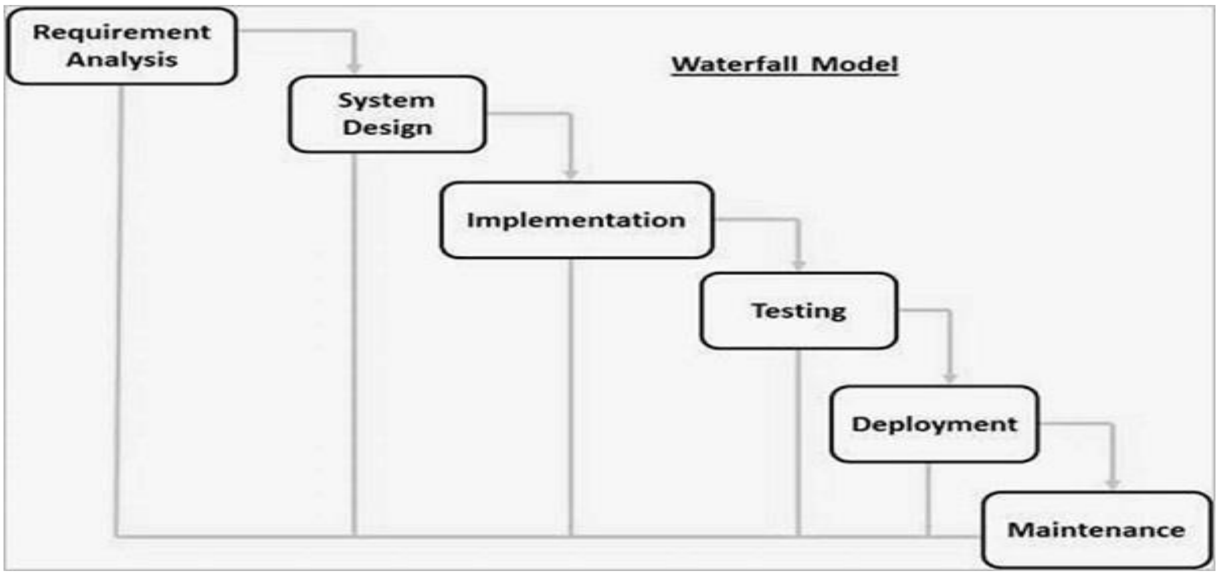
- Ensures **quality** and **reliability** of software.
- Makes development **predictable** and **manageable**.
- Reduces risk of project failure.
- Improves **team coordination** and **customer satisfaction**.

What is a Software Process Model?

A **software process model** is a **standardized representation or framework** that describes how the software process should be organized and carried out.

Major Software Process Models

1. The Waterfall Model



- **Description:**
  - A **linear and sequential** approach.
  - Each phase (requirements → design → implementation → testing → deployment → maintenance) must be completed **before the next begins**.
- **Advantages:**
  - Simple and easy to understand.

    - Works well for projects with **clear, fixed requirements**.
- **Disadvantages:**
    - Inflexible to changes.
    - Problems found late in testing are **expensive to fix**.

2. Evolutionary Development

- **Description:**
    - Software is developed in **increments/iterations**.
    - Starts with an initial version → refined through user feedback → evolves into the final system.
- **Types:**
    - **Prototyping model** – Quick working model shown to user for feedback.
    - **Incremental/Iterative model** – System is built and delivered in parts.
- **Advantages:**
    - Adapts to changing requirements.
    - Early versions available to users.
- **Disadvantages:**
    - Requires close customer involvement.
    - May lead to poorly structured code if not managed well.

3. Component-Based Software Engineering (CBSE)

- **Description:**
    - Focuses on building software by **integrating pre-built, reusable components** rather than developing everything from scratch.
- **Process Steps:**

1. Identify candidate components.
2. Integrate them into the system.
3. Adapt/modify where necessary.
4. Test and deploy.

- **Advantages:**
    - Faster development (reusability).
    - Reduces cost and improves reliability.
- **Disadvantages:**
    - Finding suitable components may be difficult.
    - Dependency on third-party components can create risks.

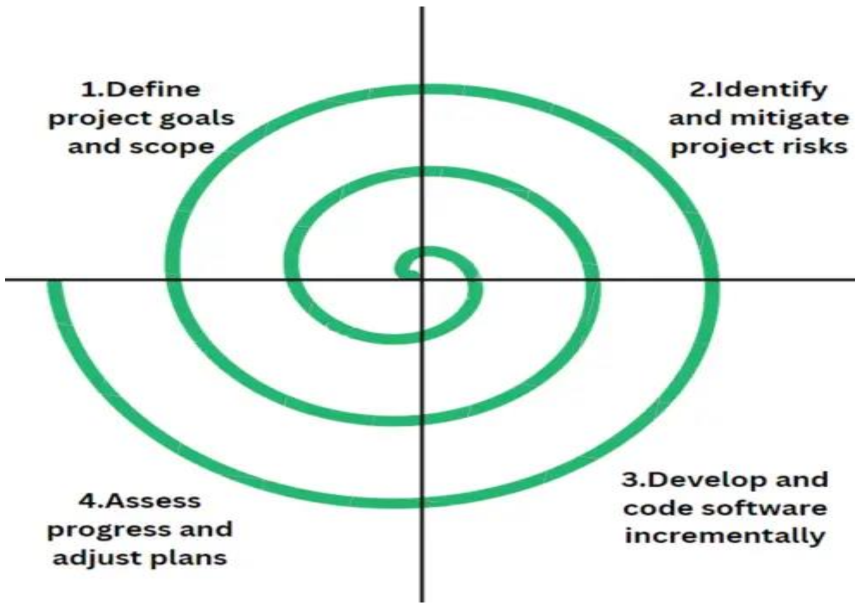
Process Iteration

- **Definition:**
  Iteration means repeating software process activities (requirements, design, implementation, testing) multiple times to refine and improve the system.
- **Idea:** Instead of doing each activity just once (like in Waterfall), the team revisits activities to handle changes or improvements.
- **When Used:**
    - Requirements are unclear at the start.
    - Projects that need frequent feedback and corrections.
- **Advantage:** Flexible, allows continuous improvement.
- **Disadvantage:** Can lead to scope creep (never-ending changes) if not controlled.

Incremental Delivery (Incremental Model)

- **Definition:**
  Software is developed and delivered in **small parts (increments)** rather than building the entire system at once. Each increment adds new features.
- **Process:**
  1. Core/basic system delivered first.
  2. More features added in later increments.
  3. Final system evolves after several increments.
- **Advantages:**
  - Users get usable software early.
  - Feedback from users helps guide next increments.
  - Lower risk than Waterfall.
- **Disadvantages:**
  - System architecture must be designed to accommodate future increments.
  - Planning and management can be complex.

Spiral Development (Spiral Model)


1.Define project goals and scope
2.Identify and mitigate project risks
3.Develop and code software incrementally
4.Assess progress and adjust plans

- **Definition:**
  A **risk-driven iterative model** combining aspects of **Waterfall + Iterative development**.
- **Process Structure (Spirals):**
  Each loop/spiral represents one phase of development, containing:
  1. **Objective setting** (what to achieve).
  2. **Risk analysis & resolution** (identify and reduce risks).
  3. **Development & validation** (build and test).
  4. **Planning** for the next iteration.
- **Advantages:**
  - Strong focus on risk management.
  - Flexible and adaptable to large, complex projects.
- **Disadvantages:**
  - Expensive and requires expertise in risk analysis.
  - May take longer for small projects.

Rapid Software Development (RSD)

- **Definition:**
  An umbrella term for approaches that emphasize **quick development and delivery** of software, often sacrificing some documentation for speed.
- **Key Features:**
  - Short development cycles.
  - Active user involvement.
  - Iterative and incremental process.
- **Goal:** Deliver usable software **faster** and adapt quickly to changing requirements

Agile Methods

- **Definition:**
  Agile is a family of lightweight, iterative methods for **flexible, customer-focused** software development.
- **Principles (from Agile Manifesto):**
  - Individuals and interactions over processes and tools.
  - Working software over documentation.
  - Customer collaboration over contract negotiation.
  - Responding to change over following a plan.
- **Examples:** Scrum, Extreme Programming (XP), Kanban.
- **Advantages:** Flexibility, continuous customer feedback, fast delivery.
- **Disadvantages:** Needs strong customer involvement, can be hard to manage in large projects.

Extreme Programming (XP)

- **Definition:**
  An **Agile method** that emphasizes **customer satisfaction, teamwork, and frequent releases**.
- **Key Practices:**
  - **Pair programming** (two programmers work together at one computer).
  - **Test-driven development (TDD)** – write tests before code.
  - **Continuous integration** – code integrated and tested frequently.
  - **Simple design** – keep design as simple as possible.
  - **Refactoring** – improving existing code continuously.
- **Advantages:** Produces high-quality code, adapts well to changes.
- **Disadvantages:** Requires skilled developers, heavy customer involvement.

Rapid Application Development (RAD)

- **Definition:**
  A software development methodology that emphasizes **quick prototyping and iterative delivery** instead of long development cycles.
- **Phases:**
  1. Requirements planning.
  2. User design (with prototyping).
  3. Construction.
  4. Cutover (deployment).
- **Advantages:** Faster development, active user involvement, reduced risk.
- **Disadvantages:** Not suitable for large, complex systems; depends heavily on strong team and user availability.

Software Prototyping

- **Definition:**
  Building a **simplified, working model (prototype)** of the system to understand requirements and get feedback before final development.
- **Types:**
  - **Throwaway prototyping** – built quickly, then discarded.
  - **Evolutionary prototyping** – gradually refined into the final system.
- **Advantages:** Clearer requirements, user involvement, reduced risk of failure.
- **Disadvantages:** Users may mistake prototype as final system, may lead to poorly structured system if evolved without planning.

Computer Aided Software Engineering (CASE)

Definition-Computer Aided Software Engineering (CASE) **refers to the use of** software tools and automated systems **to support the activities of software development, testing, and maintenance.**

In simple words:
CASE tools = **software that helps in building other software**.

Overview of CASE Approach

- **Goal:** Increase **productivity, quality, and consistency** in software development by automating tasks.
- **Support in SDLC:**
  - Requirements analysis (e.g., modeling tools).
  - Design (e.g., UML diagram tools).
  - Coding (e.g., code generators, IDEs).
  - Testing (e.g., automated test tools).
  - Maintenance (e.g., version control, debugging tools).
- **Benefits:**
  - Faster development.
  - Standardization of design and documentation.
  - Reduced human error.
  - Improved collaboration among developers.

Classification of CASE Tools

CASE tools are usually divided into three categories based on the phase of the **Software Development Life Cycle (SDLC)** they support:

(a) Upper CASE Tools

- Support **early phases** of development (planning, analysis, design).
- Examples:
  - Requirement analysis tools.
  - Modeling tools (ER diagrams, UML).
  - Project management tools.

(b) Lower CASE Tools

- Support **later phases** (implementation, testing, maintenance).
- Examples:
    - Code generators.
    - Testing tools (JUnit, Selenium).
    - Debugging tools.
    - Version control systems (Git).

(c) Integrated CASE Tools (I-CASE)

- Cover **entire SDLC**, integrating upper + lower CASE functionalities.
- Provide **end-to-end automation**.
- Examples:
    - IBM Rational Rose.
    - Visual Paradigm.
    - Enterprise Architect.

Software Requirement Analysis and Specification (SRAS)

Software Requirement Analysis and Specification is the process of **identifying, analyzing, documenting, validating, and managing the needs of users and stakeholders** for a software system. It is a critical phase of the **Software Development Life Cycle (SDLC)**.

System and Software Requirements

System Requirements

These describe the **requirements of the entire system**, including:

- Hardware
- Software
- Network
- People
- Procedures

Example:
"The system must include a server with minimum 16 GB RAM."

Software Requirements

These describe **only the requirements of the software** to be developed.

Example:
"The software must generate monthly reports automatically."

Types of Software Requirements

(A) Functional Requirements

Define **what the system should do**.

Examples:

- User login and authentication
- Data entry and validation
- Report generation
- Search and update records

(B) Non-Functional Requirements

Define **how the system should work**.

Types:

- Performance (response time)
- Security
- Reliability
- Usability
- Scalability

Examples:

- “System should respond within 2 seconds.”
- “System must use encrypted passwords.”

(C) Domain Requirements

Requirements that come from the **application domain**.

Examples:

- Banking rules
- Medical standards
- Government regulations

(D) User Requirements

High-level requirements stated in **simple language** for users.

Examples:

- “The user should be able to book tickets online.”
- “The user should receive confirmation messages.”

Elicitation and Analysis of Requirements

Requirement Elicitation

It is the process of **gathering requirements from stakeholders**.

Requirement Analysis

It is the process of **studying, refining, and modeling requirements**.

Process Modeling with DFDs

(A) Physical DFD

Shows **how the system is physically implemented**.

- People
- Hardware
- Files

(B) Logical DFD

Shows **what the system does logically**.

- Data flow
- Processes
- Data storage

Entity Relationship Diagram (ERD)

Used for **database design**.
Shows:

- Entities
- Attributes
- Relationships

Data Dictionary

A **central repository of data definitions**.

Contains:

- Data name
- Data type
- Size
- Description
- Source

Requirement Validation

Requirement validation ensures that the **documented requirements are correct, complete, consistent, and feasible**.

Validation Techniques

- Requirement reviews
- Prototyping
- Test case generation
- User approval

Objectives

- Remove ambiguity
- Detect errors
- Ensure user satisfaction
- Avoid rework

Requirement Specification

Requirement specification is the process of **writing the requirements clearly and precisely** in a document.

Software Requirement Specification (SRS)

Definition

SRS is a **formal document that describes the complete behavior of the system**.

It acts as a:

- Contract between client and developer
- Reference for design, coding, and testing

Structure and Contents of SRS

1. Introduction
2. Overall description
3. Specific requirements
4. External interface requirements
5. System features
6. Non-functional requirements
7. Constraints
8. Assumptions
9. Appendices

SRS Format (IEEE Standard – Simplified)

1. Purpose
2. Scope
3. Definitions
4. Product perspective
5. Product functions
6. User characteristics
7. Operating environment
8. Functional requirements
9. Non-functional requirements
10. Database requirements
11. External interfaces
12. Future enhancements

Feasibility Study

A **feasibility study** determines whether the proposed software system is **practically and economically viable**.

Types of Feasibility

(A) Technical Feasibility

- Availability of hardware and software
- Technical skills of employees

(B) Economic Feasibility

- Cost vs benefit analysis
- Development cost
- Maintenance cost

(C) Operational Feasibility

- User acceptance
- Organizational support
- Training needs

(D) Schedule Feasibility

- Whether the project can be completed within the time limit

(E) Legal Feasibility

- Copyright laws
- Data protection laws
- Software licensing

Software Design

Software Design is the process of converting **software requirements into a blueprint** that guides the construction of the software system. It acts as a **bridge between requirement analysis and coding**.

Design Concepts

1. Abstraction

Abstraction means **hiding unnecessary details** and showing only essential features.

Types:

- **Data Abstraction** – Hides data representation (e.g., stack, queue)
- **Process Abstraction** – Hides internal processing steps

✅Advantage: Reduces complexity and improves understanding.

2. Architecture

Software architecture defines the **overall structure of the system** and how components interact.

Includes:

- Components
- Data flow
- Control flow
- Interfaces

Examples:

- Layered architecture
- Client–server architecture
- MVC (Model–View–Controller)

3. Design Patterns

A **design pattern** is a **reusable solution to a common design problem**.

Examples:

- Singleton
- Factory
- Observer
- MVC

✅Benefit: Improves reliability and reusability.

4. Modularity

Modularity means dividing the software into **small, manageable modules**.

Each module:

- Performs a specific task
- Can be developed independently

✅Benefits:

- Easy testing
- Easy maintenance
- Better understanding

5. Cohesion

Definition of Cohesion

Cohesion is the measure of the **degree to which the elements inside a module are functionally related to each other**.
It shows how **focused a module is on performing a single task**.

✅**Good software design aims for *high cohesion*.**

Importance of Cohesion

- Improves **maintainability**
- Makes modules **easy to understand**
- Simplifies **testing and debugging**
- Increases **reusability**
- Reduces **errors**

Types of Cohesion (from Worst to Best)

1. Coincidental Cohesion (Worst Type)

Elements in a module are **unrelated** and grouped arbitrarily.

Example:
A single module performing printing, calculation, and file handling together.

❌Very poor design
❌Difficult to maintain

2. Logical Cohesion

Elements perform **similar types of operations**, selected by a control flag.

Example:
A module that handles all input operations based on a flag value.

❌Still not focused on a single task

3. Temporal Cohesion

Elements are grouped because they are **executed at the same time**.

Example:
A module that performs initialization or shutdown tasks.

□ Better than logical cohesion but still not ideal

4. Procedural Cohesion

Elements are grouped because they **follow a specific sequence of execution**.

Example:
A module that reads data → processes data → prints data.

 Sequence-based, not function-based

5. Communicational Cohesion

Elements operate on the **same data set**.

Example:
A module that reads a student record and updates the same record.

✅Data-related grouping
✅Good cohesion

6. Sequential Cohesion

The **output of one part is the input of another part** within the same module.

Example:
A module where one function calculates total and another calculates average using that total.

✅Very good cohesion

7. Functional Cohesion (Best Type)

All elements contribute to **one and only one well-defined task**.

Example:
A module that only calculates salary.

✅Highest quality design
✅Most desirable type

6. Coupling

Definition of Coupling

Coupling is a measure of the **degree of dependency between two software modules**.
It indicates how strongly one module is connected to another.

✅**Good software design aims for *low coupling*.**
Low coupling means:

- Modules are independent
- Easy to modify and maintain
- Easy to test and debug

Importance of Coupling

- Reduces effect of changes
- Improves maintainability

- Increases reusability
- Simplifies debugging
- Enhances reliability

Types of Coupling (from Worst to Best)

1. Content Coupling (Worst Type)

Occurs when **one module directly accesses the internal data or code of another module**.

Examples:

- One module modifies the local variables of another
- One module branches into the middle of another module

❌Disadvantages:

- Very poor design
- High dependency
- Very difficult to maintain

2. Common Coupling

Occurs when **two or more modules share common global data**.

Example:

- Multiple modules using the same global variable

❌Problems:

- Changes in global data affect all modules
- Difficult debugging

3. Control Coupling

Occurs when **one module controls the behavior of another by passing control information** (flags or switches).

Example:

- A module passes a flag to decide what another module should do

❌Disadvantage:

- Called module becomes dependent on the calling module's logic

4. Stamp Coupling (Structure Coupling)

Occurs when **a data structure is passed between modules**, but only **part of it is used**.

Example:

- Passing a full student record when only roll number is needed

❌Problem:

- Unnecessary dependency on data structure

5. Data Coupling (Best Type)

Occurs when **only required data is passed between modules using parameters**.

Example:

- Passing two numbers to a function for addition

✅Advantages:

- Lowest dependency
- Clean communication
- Best design practice

7. Information Hiding

Information hiding means **internal details of a module are hidden from other modules**.

Only necessary details are exposed through interfaces.

✅Benefits:

- Reduces dependency
- Improves security
- Makes changes easier

8. Functional Independence

A module is functionally independent if:

- It has **high cohesion**
- It has **low coupling**

✅Benefits:

- Easy testing
- Easy debugging
- Easy reuse

9. Refinement

Refinement is the process of **breaking a high-level design into detailed lower-level designs** step by step.

Also called **stepwise refinement**.

✅Used to move from:

- Abstract design → Detailed design

Design of Input and Control

(A) Input Design

Input design focuses on:

- How data is **entered into the system**
- How to **reduce input errors**
- How to make input **easy and fast**

Objectives of Input Design:

- Accuracy
- Speed
- Ease of use
- Security
- Validation

Input Design Methods:

- Forms
- Keyboards
- Scanners
- Sensors
- Voice input

Input Validation Checks:

- Range check
- Type check
- Length check
- Format check

(B) Control Design

Control design ensures that the **system functions correctly and securely**.

Types of Controls:

1. **Input Control** – Prevents incorrect data
2. **Processing Control** – Ensures correct processing
3. **Output Control** – Ensures correct output
4. **Security Control** – Prevents unauthorized access
5. **Backup and Recovery Control** – Prevents data loss

✅Examples:

- Password protection
- Access rights
- Logs and audit trails

Design of User Interface (UI Design)

User Interface Design deals with how users **interact with the system**.

A good UI must be:

- Simple
- Attractive
- Consistent
- User-friendly

Elements of Good UI Design

1. **Clarity** – Easy to understand
2. **Consistency** – Same design rules everywhere
3. **Simplicity** – Avoid unnecessary complexity
4. **Feedback** – System must respond to user actions
5. **Error tolerance** – Prevent and handle mistakes
6. **Accessibility** – Usable by all users

CODING

Coding is the process of converting the **software design into executable program instructions** using a programming language. It is one of the most important phases of the **Software Development Life Cycle (SDLC)**.

Coding Standards

Definition of Coding Standards

Coding standards are a set of **rules and guidelines** for writing consistent, readable, and maintainable source code.

Objectives of Coding Standards

- Improve code readability
- Maintain uniformity
- Reduce programming errors
- Make maintenance easy
- Support teamwork

Advantages of Coding Standards

- Easy understanding of code
- Better teamwork
- Easy debugging
- Faster maintenance
- High software quality

SOFTWARE TESTING AND QUALITY ASSURANCE

Software Testing is the process of **executing a program to detect errors**.
Quality Assurance (QA) is the process of **ensuring that software development follows proper standards, processes, and procedures** to produce high-quality software.

Verification and Validation

Verification

Verification checks **“Are we building the product correctly?”**
It is a **static process** (no code execution).

Examples:

- Requirement review
- Design review
- Code inspection

✅Objective: To ensure the product is built as per specifications.

Validation

Validation checks **“Are we building the right product?”**
It is a **dynamic process** (code execution required).

Examples:

- Software testing
- User acceptance testing

✅Objective: To ensure the product satisfies user needs.

Techniques of Testing

(A) Black-Box Testing

Tester checks the **functionality without knowing internal code**.

Focuses on:

- Input
- Output
- Functional behavior

Types:

- Equivalence class testing
- Boundary value testing

✅Used for: Functional testing
✅Performed by: Testers

(B) White-Box Testing

Tester examines the **internal logic and structure of the program**.

Focuses on:

- Code paths
- Loops
- Conditions

Types:

- Statement coverage
- Branch coverage
- Path coverage

✅Used for: Logic testing
✅Performed by: Programmers

Levels of Testing

1. Unit Testing

Testing of **individual program units or modules**.

Performed by: Developer
Purpose: To test each function separately.

2. Integration Testing

Testing of **combined modules after unit testing**.

Purpose: To check **interaction between modules**.

Types:

- Top-down testing
- Bottom-up testing

3. Interface Testing

Testing the **interaction between different systems or modules** through interfaces.

Examples:

- Software ↔ Hardware
- Software ↔ Database

4. System Testing

Testing the **complete integrated system** as a whole.

Includes:

- Functional testing
- Performance testing
- Security testing

5. Alpha Testing

Testing done by **developers at the developer’s site**.

Performed before beta testing.
Users may or may not be involved.

6. Beta Testing

Testing done by **actual users at their location**.

Purpose:

- To get real-world feedback
- To detect remaining defects

7. Regression Testing

Re-testing of the software after:

- Bug fixing
- Adding new features

Purpose:

- To ensure that changes **do not affect existing functionality**

Quality Management Activities

Quality management involves **planning, controlling, and improving software quality**.

Main Quality Management Activities:

1. **Quality Planning**
    - Setting quality standards
    - Selecting processes and metrics
2. **Quality Assurance**
    - Process-oriented
    - Ensures correct development process is followed
3. **Quality Control**
    - Product-oriented
    - Detects defects using testing and inspection
4. **Quality Improvement**
    - Continuous improvement of processes

Product and Process Quality

Product Quality

Refers to the **quality of the final software product**.

Quality attributes:

- Correctness
- Reliability
- Efficiency
- Usability
- Maintainability
- Portability

Process Quality

Refers to the **quality of the software development process**.

Good process quality leads to:

- Fewer defects
- Better product quality
- Reduced cost

✅**Good process → Good product**

Standards

ISO 9000

ISO 9000 is an **international quality management standard** developed by the International Organization for Standardization.

Purpose:

- To ensure consistent quality in products and services

Key Features:

- Documentation of processes
- Continuous improvement
- Customer satisfaction
- Audit-based certification

✅ISO 9001 is the most widely used standard.

Capability Maturity Model (CMM)

CMM is a framework developed by **SEI (Software Engineering Institute)** to improve software process quality.

It defines **5 maturity levels**:

1. **Level 1 – Initial**
   - No defined process
   - Ad-hoc development
2. **Level 2 – Repeatable**
   - Basic project management
   - Past successes can be repeated
3. **Level 3 – Defined**
   - Standard, documented process for all projects
4. **Level 4 – Managed**
   - Quality and performance metrics used
5. **Level 5 – Optimizing**
   - Continuous process improvement

✅Higher CMM level → Better process quality

MANAGING SOFTWARE PROJECTS

Software Project Management (SPM) is the discipline of **planning, organizing, directing, and controlling software projects** to achieve project goals within **time, cost, and quality constraints**.

Need for Software Project Management

- To complete the project **on time**
- To keep the project **within budget**
- To ensure **high software quality**
- To manage **resources effectively**
- To reduce **project risks**
- To satisfy **customer requirements**

Objectives of Project Management

- Complete the project **within time**
- Complete the project **within budget**
- Achieve **required quality**
- Proper use of **resources**
- Reduce **project risks**
- Ensure **customer satisfaction**

Software Size Estimation and Cost Estimation

Software Estimation

Software estimation predicts:

- **Project size**
- **Development effort**
- **Cost**
- **Time**

What is Productivity?

Productivity = Amount of software produced per unit effort.

COCOMO II (Constructive Cost Model)

COCOMO II is an **algorithmic model used for estimating software cost and effort**.

It estimates:

- Effort (person-months)
- Development time
- Cost

COCOMO II considers:

- Product complexity
- Team experience
- Tools and technology

Used for:

- Modern software systems
- Object-oriented and reusable software

Project Scheduling

Project scheduling decides:

- **What tasks will be done**
- **When tasks will be done**
- **Who will do the tasks**

Scheduling

Scheduling assigns:

- Start time
- End time
- Resources

It helps to:

- Monitor progress
- Avoid delays
- Use manpower effectively

Risk Management

Software Risk

A **risk** is an uncertain event that can **cause loss to the project**.

Examples:

- Change in requirements
- Developer leaving the project
- Technology failure
- Budget shortage

Risk Management Process

1. Risk Identification

Identifying possible risks.
Examples:

- Technical risk
- Schedule risk
- Cost risk
- Resource risk

2. Risk Projection

Estimating:

- Probability of risk
- Impact of risk

3. Risk Refinement

Breaking a large risk into **smaller, manageable risks**.

4. Risk Mitigation, Monitoring & Management (RMMM)

- **Mitigation** – Steps to reduce risk impact
- **Monitoring** – Continuous observation of risks
- **Management** – Actions taken when risk occurs

✅This complete process is called the **RMMM Plan**.

Managing People

People are the **most important resource** in a software project.

Objectives of People Management

- Build an effective team
- Encourage team coordination
- Improve productivity
- Reduce conflicts

CURRENT TRENDS IN SOFTWARE ENGINEERING

Software Engineering is continuously evolving due to rapid changes in **technology, user expectations, business needs, and development methods**. Modern software development focuses on **speed, quality, flexibility, and customer satisfaction**.

Current Trends in Software Engineering

- Cloud computing
- Mobile application development
- Artificial Intelligence (AI)
- Machine Learning (ML)
- Internet of Things (IoT)
- DevOps
- Agile and Scrum methodologies
- Cybersecurity focus

Web Engineering

Definition

Web Engineering is a discipline that applies **systematic and disciplined approaches to the development and maintenance of web-based applications**.

It deals with:

- Web application design
- Web development
- Web testing
- Web maintenance

Characteristics of Web Applications

- Run on web browsers
- Distributed architecture
- Multimedia-rich
- Highly interactive
- Fast-changing requirements
- Need for high security

Agile Process

Definition

The **Agile Process** is a **modern software development approach** that focuses on:

- Customer collaboration
- Fast delivery
- Flexibility to change
- Continuous improvement

Agile was introduced to overcome the limitations of the **Waterfall Model**.

Key Principles of Agile

- Working software is delivered frequently
- Customer involvement is continuous
- Changes in requirements are welcomed
- Small development cycles (iterations)
- Face-to-face communication is preferred
- Simplicity and quality focus