Software Engineering

Unit–1: Introduction to Software Engineering

1. What is Software Engineering?

Definition

Software Engineering is the systematic, disciplined, and organized approach used to develop, operate, maintain, and manage software.

According to the Fritz Bauer definition:

"Software Engineering is the establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines."

Explanation

Software engineering applies engineering principles to software development just like civil engineering is used for building bridges.

It includes:

- Planning
- Designing
- Coding
- Testing
- Maintenance
- Documentation

Objectives of Software Engineering

1. Develop high-quality software
2. Reduce development cost
3. Complete projects on time
4. Improve maintainability
5. Increase customer satisfaction

Characteristics of Software Engineering

- Systematic approach
- Team-oriented development
- Documentation based
- Focus on quality
- Uses tools and methodologies

---

2. History of Software Engineering

Early Stage (1950s–1960s)

- Programs were small and simple.
- Coding was done without planning.
- Developers used machine language and assembly language.

Software Crisis (1968)

As software systems became large:

- Projects failed
- Cost increased
- Delivery became late
- Errors increased

This problem was called the **Software Crisis**.

The term "Software Engineering" was introduced during the NATO Software Engineering Conference to solve these problems.

Modern Era

Today software engineering uses:

- Agile methods
- Automation tools
- Cloud computing
- DevOps
- AI-assisted development

---

3. Software Crisis

Definition

Software Crisis refers to the difficulties faced in developing software systems because of increasing complexity and poor development methods.

Causes of Software Crisis

1. Increase in software size
2. Poor project management
3. Lack of documentation
4. Improper testing
5. Changing user requirements
6. Lack of skilled developers

Problems Created

- Late delivery
- Budget overrun
- Low-quality software
- Difficult maintenance
- Unreliable systems

Solution to Software Crisis

Software engineering principles were introduced:

- SDLC models
- Proper planning
- Testing methods
- Documentation
- Project management

---

4. Evolution of a Programming System Product

Definition

Evolution of programming systems means the gradual improvement of software from simple programs to complex software products.

Stages of Evolution

(i) Initial Program

- Written by one programmer
- Small and simple
- No documentation

(ii) Programming Product

- Properly tested
- Well documented
- Usable by others

(iii) Programming System

- Collection of interacting programs
- Supports multiple users and functions

(iv) Programming System Product

- Combination of product quality and system capability
- Commercial software systems

Example

Operating systems like Microsoft Windows and Linux are programming system products.

---

5. Product and Characteristics of Software

Software Product

A software product is a complete software package delivered to customers with documentation and support.

Types of Software Products

1. Generic Products
   Example: MS Office
2. Customized Products
   Example: Banking software developed for a specific bank

Characteristics of Software

1. Software is Developed, Not Manufactured

Software is created through coding and design.

2. Software Does Not Wear Out

Hardware gets damaged physically, but software becomes outdated due to changes.

3. Software is Intangible

Software cannot be touched physically.

4. Software is Complex

Large software contains many modules and interactions.

5. Software Requires Maintenance

Changes and updates are regularly needed.

6. Software is Easy to Copy

A software product can be duplicated at low cost.

---

6. Brooks' No Silver Bullet

Definition

Fred Brooks stated in his famous paper **"No Silver Bullet"** that:
No single technology or method can solve all software development problems completely.

Main Idea

Software development complexity cannot be removed entirely by:

- New programming languages
- Better tools
- Automation

Essential Difficulties

1. Complexity
2. Conformity
3. Changeability
4. Invisibility

Accidental Difficulties

Problems caused by tools or implementation methods.

Conclusion

There is no magical solution (“silver bullet”) to increase software productivity drastically.

---

7. Software Myths

Definition

Software myths are false beliefs about software development.

Types of Software Myths

(i) Management Myths

Example:

- “If a project is late, adding more programmers will help.”

Reality:

- New programmers need training, making projects slower initially.

(ii) Customer Myths

Example:

- “Requirements can change anytime without affecting cost.”

Reality:

- Requirement changes increase cost and time.

(iii) Developer Myths

Example:

- “Once the program works, the job is done.”

Reality:

- Testing, maintenance, and documentation are also important.

---

8. Software Development Life Cycle (SDLC)

Definition

SDLC is a structured process used to develop software systematically.

Phases of SDLC

1. Requirement Analysis
2. System Design
3. Coding
4. Testing
5. Deployment
6. Maintenance

---

9. Software Development Process

Definition

A software development process is a set of activities used to develop software.

Main Activities

1. Specification
2. Design and Implementation
3. Validation
4. Evolution

---

10. Code-and-Fix Model

Definition

A simple software development approach where coding starts immediately without planning.

Process

- Write code
- Fix errors repeatedly

Advantages

- Simple
- Suitable for very small projects

Disadvantages

- Poor quality
- Difficult maintenance
- No documentation

---

11. Waterfall Model

Definition

A linear sequential SDLC model where one phase must complete before the next begins.

Phases

1. Requirement Analysis
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

Diagram

$$Requirement\ Analysis \rightarrow Design \rightarrow Coding \rightarrow Testing \rightarrow Maintenance$$

Advantages

- Simple and easy
- Good documentation

Disadvantages

- Difficult to change requirements
- Not suitable for large dynamic projects

---

12. Evolutionary Model

Definition

Software is developed gradually with continuous user feedback.

Features

- Incremental development
- Frequent improvements

Advantages

- Flexible
- Better customer satisfaction

Disadvantages

- Difficult management
- Requires continuous feedback

---

13. Incremental Implementation Model

Definition

Software is developed in small increments or modules.

Example

A banking system may release:

- Login module first
- Transaction module later
- Loan module afterward

Advantages

- Faster delivery
- Easier testing

Disadvantages

- Requires careful planning

---

14. Prototyping Model

Definition

A prototype (sample version) is developed before the final software.

Types

1. Throwaway Prototype
2. Evolutionary Prototype

Advantages

- Better understanding of requirements
- User involvement

Disadvantages

- May increase development cost

---

15. Spiral Model

Definition

The Spiral Model combines iterative development with risk analysis.

Developed by Barry Boehm.

Phases

1. Planning
2. Risk Analysis
3. Engineering
4. Evaluation

Diagram

$$Planning \rightarrow Risk\ Analysis \rightarrow Engineering \rightarrow Evaluation$$

Advantages

- Risk management
- Suitable for large projects

Disadvantages

- Expensive
- Complex management

---

16. Software Reuse

Definition

Using existing software components to develop new software.

Types

1. Code reuse
2. Component reuse
3. Framework reuse

Advantages

- Saves time
- Reduces cost
- Improves reliability

Disadvantages

- Compatibility issues
- Licensing problems

---

17. Critical Comparison of SDLC Models

Model	Advantages	Disadvantages	Suitable For
Code-and-Fix	Simple	Poor quality	Very small projects
Waterfall	Easy management	No flexibility	Stable requirements
Evolutionary	Flexible	Difficult control	Changing requirements
Incremental	Faster delivery	Planning required	Medium projects
Prototype	Better understanding	Extra cost	Unclear requirements
Spiral	Risk handling	Expensive	Large critical systems

---

18. Rational Unified Process (RUP)

Definition

RUP is an iterative software development process framework developed by IBM.

Phases

1. Inception
2. Elaboration
3. Construction
4. Transition

Features

- Iterative development
- Risk management
- Use-case driven

Advantages

- High-quality software
- Controlled development

Disadvantages

- Complex process
- Requires expertise

---

19. Rapid Application Development (RAD)

Definition

RAD focuses on quick software development using reusable components and user feedback.

Features

- Fast development
- Prototype-based

- User involvement

Advantages

- Reduced development time
- Faster delivery

Disadvantages

- Requires skilled team
- Not suitable for large projects

---

20. Agile Development Process

Definition

Agile is a modern software development methodology focusing on flexibility, collaboration, and continuous delivery.

Agile Principles

1. Customer collaboration
2. Continuous improvement
3. Small releases
4. Team communication

Popular Agile Methods

- Scrum
- Extreme Programming (XP)
- Kanban

Advantages

- Fast delivery
- Flexible changes
- Better customer satisfaction

Disadvantages

- Requires active customer involvement
- Difficult documentation

Conclusion

Software Engineering provides systematic methods for developing reliable and efficient software. Different SDLC models are used according to project requirements. Modern approaches like Agile and RAD focus on flexibility and faster delivery, while traditional models like Waterfall focus on structured development.

Unit – 2: Requirements Engineering and Requirement Analysis

1. Importance of Requirement Analysis

Definition

Requirement Analysis is the process of understanding, identifying, analyzing, and documenting the needs and expectations of users for a software system.
It is one of the most important phases in Software Engineering because incorrect requirements lead to project failure.

Importance of Requirement Analysis

1. Understanding User Needs

Requirement analysis helps developers understand what users actually want from the system.

2. Reduces Development Errors

Clear requirements reduce mistakes during coding and testing.

3. Saves Time and Cost

Fixing errors in the requirement phase is cheaper than fixing them after development.

4. Improves Software Quality

Correct requirements help build reliable and efficient software.

5. Better Communication

It creates proper communication between:

- Customers
- Developers
- Testers
- Managers

6. Basis for Design and Testing

All software design and test cases depend on requirements.

---

2. User Needs

Definition

User needs are the expectations, goals, and problems users want the software system to solve.

Types of User Needs

1. Business Needs

Related to organizational goals.
Example:

- A bank wants secure online transactions.

2. User Needs

Related to actual users.
Example:

- Customers want fast money transfer.

3. Operational Needs

Related to system operation.
Example:

- System should work 24×7.

Characteristics of Good User Needs

- Clear
- Complete
- Understandable
- Consistent
- Feasible

---

3. Software Features and Software Requirements

A. Software Features

Definition

Software features are the visible functions or services provided by software to users.

Examples

- Login system
- Online payment
- Search functionality
- Report generation

B. Software Requirements

Definition

Software requirements describe what the software should do and the constraints under which it should operate.

Difference Between Features and Requirements

Feature	Requirement
Visible capability	Detailed technical description
User-oriented	Developer-oriented
High level	Detailed specification

---

4. Classes of User Requirements

User requirements are divided into two classes:

A. Enduring Requirements

Definition

Requirements that remain stable for a long time.

Examples

- Security
- Data backup
- User authentication

Characteristics

- Stable
- Long-term
- Essential

B. Volatile Requirements

Definition

Requirements that change frequently with time.

Examples

- UI design
- Business policies
- Government regulations

Types of Volatile Requirements

1. Mutable Requirements
2. Emergent Requirements
3. Consequential Requirements
4. Compatibility Requirements

Comparison

Enduring Requirements	Volatile Requirements
Stable	Frequently changing
Long-term	Temporary
Less modification	More modification

---

5. Sub Phases of Requirement Analysis

Requirement Analysis consists of several sub-phases.

---

1. Requirement Gathering

Collecting information from users and stakeholders.

Methods:

- Interviews
- Surveys
- Observation

---

2. Requirement Elicitation

Extracting actual requirements from gathered information.

---

3. Requirement Analysis and Negotiation

Analyzing conflicts and prioritizing requirements.

---

4. Requirement Specification

Documenting requirements formally in SRS.

---

5. Requirement Validation

Checking whether requirements are correct and complete.

Requirement Analysis Flow

$$Requirement\ Gathering \rightarrow Elicitation \rightarrow Analysis \rightarrow Specification \rightarrow Validation$$

---

6. Functional and Nonfunctional Requirements

A. Functional Requirements

Definition

Functional requirements describe what the system should do.

Examples

- User login

- Ticket booking
- Data search
- Report generation

Characteristics

- Describe system behavior
- Directly related to functions

B. Nonfunctional Requirements

Definition

Nonfunctional requirements describe how the system performs its functions.

Examples

- Security
- Performance
- Reliability
- Scalability

Types

1. Performance Requirements
2. Security Requirements
3. Reliability Requirements
4. Usability Requirements
5. Portability Requirements

Comparison

Functional Requirements	Nonfunctional Requirements
What system does	How system performs
Features/functions	Quality attributes
Easier to measure	Difficult to measure

---

7. Barriers to Eliciting User Requirements

Definition

Barriers are problems faced while collecting user requirements.

Common Barriers

1. Communication Gap

Users and developers may not understand each other.

2. Unclear Requirements

Users may not clearly explain their needs.

3. Changing Requirements

Requirements may change frequently.

4. Lack of User Participation

Users may not cooperate actively.

5. Technical Complexity

Complex systems are difficult to explain.

6. Time Constraints

Limited time affects proper requirement gathering.

---

8. Software Requirements Document (SRD)

Definition

A Software Requirements Document contains all software requirements in written form.
It acts as an agreement between customer and developer.

Contents of SRD

1. Introduction
2. Purpose
3. Scope
4. Functional Requirements
5. Nonfunctional Requirements
6. Constraints
7. Assumptions

---

9. SRS (Software Requirement Specification)

Definition

SRS is a formal document that describes complete software requirements in detail.
IEEE provides standard formats for SRS documentation.

IEEE SRS Standard Sections

1. Introduction

- Purpose
- Scope
- Definitions

2. Overall Description

- Product perspective
- Product features
- User characteristics

3. Specific Requirements

- Functional requirements
- Nonfunctional requirements

4. External Interface Requirements

- Hardware interface
- Software interface

5. Performance Requirements

6. Design Constraints

Characteristics of Good SRS

1. Correct
2. Complete
3. Consistent
4. Verifiable
5. Modifiable
6. Traceable

---

10. Requirements Engineering

Definition

Requirements Engineering is the systematic process of:

- Collecting
- Analyzing
- Documenting
- Validating
- Managing requirements

Activities of Requirements Engineering

1. Feasibility Study
2. Requirement Elicitation
3. Requirement Analysis
4. Requirement Specification
5. Requirement Validation
6. Requirement Management

Requirements Engineering Process

$$Feasibility\ Study \rightarrow Elicitation \rightarrow Analysis \rightarrow Specification \rightarrow Validation \rightarrow Management$$

---

11. Case Study of SRS for a Real-Time System

Example: Online Railway Reservation System

Purpose

To provide online train ticket booking services.

Functional Requirements

1. User registration
2. Ticket booking
3. Ticket cancellation
4. Seat availability checking
5. Payment processing

Nonfunctional Requirements

1. High security
2. Fast response time
3. 24×7 availability
4. Database backup

External Interface Requirements

- Web browser interface
- Payment gateway interface

Constraints

- Internet connection required
- Secure payment standards must be followed

---

12. Tools for Requirements Gathering

Requirement gathering tools help analyze and represent system requirements.

A. Document Flow Chart

Definition

A document flowchart shows movement of documents and information between departments or processes.

Uses

- Understand workflow
- Detect delays
- Improve communication

Example Flow

$$Customer\ Request \rightarrow Processing\ Department \rightarrow Approval \rightarrow Database\ Update$$

B. Decision Table

Definition

A decision table is a tabular representation of conditions and corresponding actions.

Structure

1. Conditions
2. Actions
3. Rules

Example

Condition	Rule 1	Rule 2
Valid Login	Yes	No
Action	Allow Access	Deny Access

Advantages

- Easy to understand
- Handles complex logic

C. Decision Tree

Definition

A decision tree is a graphical representation of decisions and possible outcomes.

Features

- Uses branches
- Represents logical conditions

Example

$$Login\ Valid? \rightarrow Yes: Access\ Granted,\ No: Access\ Denied$$

Advantages

- Simple visualization
- Easy decision-making

---

13. Introduction to Nontraditional Requirements

Definition

Nontraditional requirements refer to modern and advanced requirements beyond basic functional needs.

Examples

1. Cloud Compatibility

Software should support cloud platforms.

2. Mobile Accessibility

Applications should work on mobile devices.

3. AI Integration

Software may include artificial intelligence features.

4. Scalability

System should handle increasing users.

5. Cybersecurity

Protection against cyber threats.

6. Sustainability Requirements

Energy-efficient software systems.

Conclusion

Requirement Analysis and Requirements Engineering are the foundation of successful software development. Proper understanding of user requirements helps reduce errors, improve software quality, and ensure customer satisfaction. Tools like decision tables, decision trees, and flowcharts simplify requirement analysis and documentation.

Unit – 3: Software Design

1. Software Design

Definition

Software Design is the process of converting software requirements into a blueprint or model for software construction.

It defines:

- System architecture
- Components
- Modules
- Interfaces
- Data structures

Software design acts as a bridge between:

- Requirement Analysis
- Coding/Implementation

---

2. Goals of Good Software Design

A good software design should achieve the following goals:

---

1. Correctness

The design should satisfy all user requirements.

Example

If the requirement is online payment, the design must support secure transactions.

---

2. Understandability

The design should be easy to understand.

Benefits

- Easier maintenance
- Easier debugging

---

3. Efficiency

The software should use:

- Less memory
- Less processing time

---

4. Maintainability

The design should allow easy modifications and updates.

---

5. Reusability

Components should be reusable in other systems.

---

6. Reliability

The software should work correctly without failure.

---

7. Flexibility

The system should adapt to future changes.

---

8. Modularity

Software should be divided into independent modules.

---

3. Design Strategies and Methodologies
Definition
Design strategies are techniques used to organize and develop software systems.

Types of Design Strategies
A. Function-Oriented Design
Definition
Focuses on system functions and processes.
Characteristics

- Uses DFD
- Top-down approach
- Structured design

B. Object-Oriented Design (OOD)
Definition
Focuses on objects, classes, and their interactions.
Characteristics

- Reusability
- Encapsulation
- Inheritance

Design Methodologies
1. Top-Down Methodology
Starts from the main system and divides it into smaller modules.

---

2. Bottom-Up Methodology
Starts from small modules and combines them into larger systems.

---

4. Data-Oriented Software Design
Definition
Data-oriented design focuses on data structure and flow rather than functions.

Features

- Data is central
- Processes operate on data
- Useful for database applications

Advantages

- Better data management
- Easier database integration

Disadvantages

- Complex for large systems

---

5. Coupling

Definition
Coupling refers to the degree of interdependence between software modules.
Lower coupling is preferred.

Types of Coupling
1. Content Coupling (Worst)
One module directly accesses another module's data.

---

2. Common Coupling
Multiple modules share global data.

---

3. Control Coupling
One module controls another by passing control information.

---

4. Stamp Coupling
Entire data structures are shared.

---

5. Data Coupling (Best)
Modules communicate using simple data parameters.

Coupling Quality Comparison
Quality of Coupling Types
Lower dependency between modules indicates better software design.
02468ContentCommonControlStampData

Advantages of Low Coupling

1. Easier maintenance
2. Better reusability
3. Reduced complexity

---

6. Cohesion
Definition
Cohesion refers to how strongly related the functions inside a module are.
Higher cohesion is preferred.

Types of Cohesion
1. Coincidental Cohesion (Worst)
Unrelated tasks in one module.

---

2. Logical Cohesion
Related logically but not functionally.

---

3. Temporal Cohesion
Tasks executed at the same time.

---

4. Procedural Cohesion
Tasks follow a sequence.

5. Communicational Cohesion
Tasks operate on the same data.

6. Sequential Cohesion
Output of one task becomes input to another.

7. Functional Cohesion (Best)
All tasks contribute to one specific function.

Cohesion Quality Comparison
Quality of Cohesion Types
Higher cohesion indicates stronger module organization.
02468CoincidentalLogicalTemporalProceduralCommunicationalSequentialFunctional

Advantages of High Cohesion
- Better readability
- Easier testing
- Better maintenance

7. Modular Structure
Definition
A modular structure divides software into separate independent modules.

Characteristics
- Each module performs a specific task
- Modules interact through interfaces

Advantages
1. Easier debugging
2. Better maintenance
3. Parallel development
4. Reusability

Example
A library management system may contain:
- Login module
- Book module
- Payment module

8. Packaging
Definition
Packaging is the grouping of related classes and modules into packages.
Used mainly in object-oriented programming.

Advantages
1. Better organization

2. Easier maintenance
3. Access control
4. Reusability

Example in Java
package banking;

---

9. Structured Analysis
Definition
Structured Analysis is a method used to analyze system requirements using graphical tools.

Main Tools of Structured Analysis
1. DFD (Data Flow Diagram)
2. Data Dictionary

---

10. Data Flow Diagram (DFD)
Definition
DFD is a graphical representation of data movement through a system.

Components of DFD

Symbol	Meaning
Circle	Process
Arrow	Data Flow
Rectangle	External Entity
Open Rectangle	Data Store

Levels of DFD
1. Level 0 DFD (Context Diagram)
Shows the entire system as one process.

---

2. Level 1 DFD
Shows major sub-processes.

Example Flow

$$Customer \rightarrow Order\ Processing \rightarrow Database \rightarrow Invoice$$

Advantages of DFD
- Easy visualization
- Better communication
- Helps requirement analysis

---

11. Data Dictionary
Definition
A Data Dictionary stores definitions and descriptions of data elements used in a system.

Contents

1. Data name
2. Data type
3. Description
4. Size
5. Constraints

Example

Data Element	Description
Customer_ID	Unique customer number
Password	User authentication key

---

12. Structured Design

Definition

Structured Design converts DFD into software architecture and module structure.

Goals

- Reduce complexity
- Improve modularity
- Improve maintainability

---

13. Structure Chart

Definition

A structure chart is a hierarchical diagram showing relationships between modules.

Components

1. Module
2. Data flow
3. Control flow

Example Structure

$$Main\ Module \rightarrow Login\ Module \rightarrow Database\ Module$$

Advantages

- Clear hierarchy
- Better modularity
- Easier maintenance

---

14. Object-Oriented Design (OOD)

Definition

OOD is a software design method based on objects and classes.

Important Concepts

1. Class

Blueprint of objects.

---

2. Object

Instance of a class.

---

3. Encapsulation
Binding data and methods together.

---

4. Inheritance
Acquiring properties of another class.

---

5. Polymorphism
One interface with multiple forms.

Advantages of OOD

- Reusability
- Flexibility
- Easy maintenance

---

15. Top-Down and Bottom-Up Approach
A. Top-Down Approach
Definition
Design begins from the main module and divides into submodules.

Advantages

- Better system understanding
- Good planning

Disadvantages

- Lower-level details delayed

B. Bottom-Up Approach
Definition
Design starts with small modules and integrates them.

Advantages

- Reusable modules
- Easier testing

Disadvantages

- System structure may be unclear initially

Comparison

Top-Down	Bottom-Up
Starts from main system	Starts from small modules
Stepwise refinement	Module integration
Better overview	Better reuse

---

16. UML (Unified Modeling Language)

Definition
UML is a standard modeling language used to visualize, specify, and document software systems.
Developed by:

- Grady Booch
- James Rumbaugh
- Ivar Jacobson

Purpose of UML

- Visualize software
- Simplify design
- Improve communication

---

17. UML Diagrams
UML diagrams are divided into two categories:

A. Structural Diagrams
Examples

1. Class Diagram
2. Object Diagram
3. Component Diagram
4. Deployment Diagram

B. Behavioral Diagrams
Examples

1. Use Case Diagram
2. Sequence Diagram
3. Activity Diagram
4. State Diagram

Important UML Diagrams
1. Use Case Diagram
Shows interaction between users and system.

---

2. Class Diagram
Shows classes, attributes, and methods.

---

3. Sequence Diagram
Shows message flow between objects.

---

4. Activity Diagram
Represents workflow of activities.

UML Diagram Usage Distribution
Commonly Used UML Diagrams
Approximate usage distribution of UML diagram types in software design.
Activity Diagram

Class Diagram
Sequence Diagram
Use Case Diagram

---

18. Design Patterns
Definition
Design patterns are reusable solutions to common software design problems.
Introduced by the Gang of Four.

Categories of Design Patterns
1. Creational Patterns
Deal with object creation.
Examples

- Singleton
- Factory Method

---

2. Structural Patterns
Deal with class and object structure.
Examples

- Adapter
- Decorator

---

3. Behavioral Patterns
Deal with communication between objects.
Examples

- Observer
- Strategy

Advantages of Design Patterns

1. Reusable solutions
2. Better software structure
3. Reduced development time
4. Easier maintenance

Conclusion
Software Design is a critical phase in software engineering that transforms requirements into a structured solution. Concepts like coupling, cohesion, modularity, UML diagrams, and design patterns help developers create reliable, maintainable, and efficient software systems.

Unit – 4: Software Project Management

1. Software Project Management

Definition

Software Project Management is the process of planning, organizing, directing, and controlling software projects to achieve specific goals within:

- Time
- Cost
- Quality

It ensures successful software development and delivery.

Objectives of Software Project Management

1. Complete project on time
2. Complete project within budget
3. Maintain software quality
4. Manage risks effectively
5. Improve team coordination

---

2. Overview of Project Manager Responsibilities

Definition

A Project Manager is the person responsible for managing the entire software project.

Main Responsibilities of a Project Manager

1. Project Planning

Preparing project plans, schedules, and strategies.

---

2. Team Management

Managing developers, testers, and designers.

---

3. Resource Management

Allocating:

- Hardware
- Software
- Human resources

---

4. Cost Management

Controlling project budget and expenses.

---

5. Risk Management

Identifying and reducing project risks.

---

6. Communication

Maintaining communication with:

- Clients
- Team members
- Management

---

7. Quality Assurance

Ensuring high-quality software development.

---

8. Scheduling
Monitoring deadlines and milestones.

Skills Required for Project Manager

1. Leadership
2. Communication
3. Technical knowledge
4. Decision-making
5. Problem-solving

---

3. Project Planning
Definition
Project Planning is the process of deciding:

- What to do
- How to do
- When to do
- Who will do

Activities of Project Planning

1. Requirement analysis
2. Effort estimation
3. Cost estimation
4. Scheduling
5. Resource allocation
6. Risk planning

Project Planning Flow

$$Requirement\ Analysis \rightarrow Estimation \rightarrow Scheduling \rightarrow Resource\ Allocation \rightarrow Risk\ Management$$

Advantages of Project Planning

- Better control
- Reduced risk
- Proper resource utilization
- Timely completion

---

4. Software Measurement and Metrics
Definition
Software metrics are quantitative measures used to evaluate software processes, products, and projects.

Importance of Software Metrics

1. Improve quality
2. Estimate cost and effort
3. Measure productivity

4. Track project progress

Types of Software Metrics

1. Product Metrics
2. Process Metrics
3. Project Metrics

---

5. Line of Code (LOC)

Definition

LOC measures software size based on the number of lines of source code.

Formula

$$LOC = Total\ Number\ of\ Source\ Code\ Lines$$

Example

Module	LOC
Login Module	500
Payment Module	800
Report Module	700

Total LOC = 2000

Advantages of LOC

1. Easy to calculate
2. Useful for productivity measurement

Disadvantages of LOC

1. Language dependent
2. Cannot measure quality
3. Difficult in early stages

---

6. Function Point (FP) Based Measures

Definition

Function Point measures software size based on functionality provided to users.
Developed by Allan Albrecht.

Components of FP

1. Inputs
2. Outputs
3. User inquiries
4. Files
5. External interfaces

Formula

$$FP = UFP \times CAF$$

Where:

- UFP = Unadjusted Function Points

- CAF = Complexity Adjustment Factor

Advantages of FP

1. Language independent
2. Better early estimation
3. User-oriented

Disadvantages of FP

1. Complex calculation
2. Requires expertise

LOC vs Function Point

LOC vs Function Point Comparison
Comparison of important characteristics of LOC and Function Point metrics.
02468LOCFunction Point

---

7. Various Size-Oriented Measures

Definition

Size-oriented measures estimate software size using different parameters.
Examples:

- LOC
- Function Points
- Object Points

Importance

- Project estimation
- Productivity analysis
- Cost calculation

---

8. Halstead's Software Science

Definition

Halstead's Software Science is a software metric developed by Maurice Halstead to measure software complexity.

Basic Parameters

Symbol	Meaning
n1	Number of unique operators
n2	Number of unique operands
N1	Total operators
N2	Total operands

Important Formulas

Program Length

$$N = N_1 + N_2$$

Program Vocabulary

$$n = n_1 + n_2$$

Program Volume

$$V = N\log_2 n$$

Advantages

- Measures complexity
- Helps effort estimation

Disadvantages

- Mathematical complexity
- Difficult for large systems

---

9. Project Size Estimation Metrics

Definition

Project size estimation predicts the size of software before development.

Common Metrics

1. LOC
2. Function Points
3. Object Points
4. Use Case Points

Importance

- Cost estimation
- Resource allocation
- Scheduling

---

10. Project Estimation Techniques

Definition

Project estimation techniques predict:

- Time
- Cost
- Effort
- Resources

Techniques

1. Expert Judgment

Experienced experts estimate project size.

---

2. Analogy-Based Estimation

Uses previous similar projects.

---

3. Algorithmic Estimation

Uses mathematical models.

Example:

- COCOMO

4. Top-Down Estimation
Estimate whole project first.

5. Bottom-Up Estimation
Estimate individual modules first.

11. COCOMO Model
Definition
COCOMO (Constructive Cost Model) is an algorithmic software cost estimation model developed by Barry Boehm.

Types of COCOMO
1. Basic COCOMO
2. Intermediate COCOMO
3. Detailed COCOMO

Software Project Categories

Type	Characteristics
Organic	Small simple projects
Semi-detached	Medium complexity
Embedded	Complex real-time systems

Basic COCOMO Formula
Effort Estimation

$$Effort = a(KLOC)^b$$

Development Time

$$Time = c(Effort)^d$$

Advantages of COCOMO
1. Scientific estimation
2. Easy cost prediction

Disadvantages
1. Requires accurate input
2. Less suitable for modern agile projects

12. Staffing Level Estimation
Definition
Staffing estimation determines the number of people required for the project.

Factors Affecting Staffing
1. Project size
2. Project complexity
3. Schedule constraints

4. Team experience

Rayleigh Curve Concept
Staffing usually:

- Starts low
- Increases during development
- Decreases after testing

Staffing Distribution Example
Typical Staffing Level During Software Development
Software staffing generally rises during development and decreases near completion.
051015PlanningDesignDevelopmentTestingMaintenance

---

13. Scheduling
Definition
Scheduling is the process of assigning project activities to time periods.

Objectives of Scheduling

1. Timely completion
2. Better resource utilization
3. Progress tracking

Scheduling Techniques
1. Gantt Chart
Bar chart showing task duration.

---

2. PERT Chart
Program Evaluation and Review Technique.

---

3. CPM
Critical Path Method.

Scheduling Flow

$$Task\ Identification \rightarrow Time\ Estimation \rightarrow Resource\ Allocation \rightarrow Monitoring$$

---

14. Organization and Team Structures
Definition
Team structure defines how project members are organized.

Types of Team Structures
1. Democratic Decentralized (DD)

- Group decision-making
- Open communication

---

2. Controlled Decentralized (CD)

- Leader controls major decisions

---

3. Controlled Centralized (CC)

- Strong central authority

Comparison

Structure	Communication	Decision Making
DD	Open	Group
CD	Mixed	Shared
CC	Limited	Manager

---

15. Staffing

Definition

Staffing is the process of recruiting and managing project personnel.

Staffing Activities

1. Recruitment
2. Training
3. Team formation
4. Performance evaluation

Characteristics of Good Team Members

1. Technical skills
2. Communication
3. Teamwork
4. Creativity

---

16. Risk Management

Definition

Risk Management is the process of identifying, analyzing, and controlling project risks.

Types of Risks

1. Project Risks

- Schedule delay
- Budget problems

---

2. Technical Risks

- Technology failure
- Design issues

---

3. Business Risks

- Market changes
- Customer issues

Risk Management Process

1. Risk Identification

Finding possible risks.

---

2. Risk Analysis
Studying impact and probability.

---

3. Risk Planning
Preparing solutions.

---

4. Risk Monitoring
Tracking risks continuously.

Risk Management Cycle

$$Risk\ Identification \rightarrow Risk\ Analysis \rightarrow Risk\ Planning \rightarrow Risk\ Monitoring$$

Advantages of Risk Management

1. Reduces uncertainty
2. Improves project success
3. Prevents major failures

Conclusion
Software Project Management helps manage software projects efficiently through planning, estimation, scheduling, staffing, and risk management. Metrics such as LOC, Function Point, and COCOMO help estimate project size and effort, while effective scheduling and risk handling ensure successful software delivery.

Unit – 5: Software Development and Testing

1. Software Development

Definition

Software Development is the process of designing, coding, testing, and maintaining software applications.

It converts software design into executable programs.

Main Activities in Software Development

1. Selecting a programming language
2. Writing code
3. Following coding guidelines
4. Documentation
5. Testing
6. Debugging

---

2. Selecting a Language

Definition

Selecting a programming language means choosing the most suitable language for software development.

Factors Affecting Language Selection

1. Application Type

Different applications require different languages.

Application	Suitable Language
Web Development	JavaScript, PHP
System Software	C, C++
Mobile Apps	Kotlin, Swift
AI Applications	Python

---

2. Performance Requirements

Fast systems may require low-level languages.

---

3. Platform Support

Language should support required operating systems.

---

4. Developer Expertise

Team familiarity with language matters.

---

5. Security Requirements

Some languages provide better security features.

Advantages of Proper Language Selection

- Better performance
- Easier maintenance
- Faster development
- Improved security

---

3. Coding Guidelines

Definition

Coding guidelines are rules and standards followed while writing source code.

Objectives of Coding Guidelines

1. Improve readability
2. Reduce errors
3. Simplify maintenance
4. Maintain consistency

Common Coding Guidelines

1. Meaningful Variable Names

Good Example

```
int studentCount;
```

Bad Example

```
int x;
```

---

2. Proper Indentation

```
if(age > 18){
   System.out.println("Eligible");
}
```

---

3. Use Comments

```
# Calculate total marks
```

---

4. Avoid Complex Logic

Keep code simple and modular.

Advantages of Coding Standards

- Easier debugging
- Better teamwork
- Improved software quality

---

4. Writing Code

Definition

Writing code is the implementation phase where programmers convert design into source code.

Steps in Coding

1. Understand requirements
2. Select algorithms
3. Write code
4. Compile and execute
5. Test and debug

Characteristics of Good Code

1. Readable
2. Efficient
3. Reusable
4. Secure
5. Maintainable

Coding Process

$$Requirement\ Understanding \rightarrow Algorithm\ Design \rightarrow Coding \rightarrow Testing \rightarrow Debugging$$

---

5. Code Documentation

Definition

Code documentation explains the functionality and logic of the code.

Types of Documentation

1. Internal Documentation

Comments inside source code.

---

2. External Documentation

Separate manuals and technical documents.

Advantages of Documentation

1. Easier maintenance
2. Better understanding
3. Simplifies future updates

Example

// Function to calculate sum

---

6. Testing Process

Definition

Software Testing is the process of checking software to identify defects and verify correctness.

Objectives of Testing

1. Detect errors
2. Ensure quality
3. Verify requirements
4. Improve reliability

Testing Process Steps

1. Test Planning
2. Test Case Design
3. Test Execution
4. Defect Reporting
5. Retesting

Testing Process Flow

$$Test\ Planning \rightarrow Test\ Case\ Design \rightarrow Test\ Execution \rightarrow Defect\ Reporting \rightarrow Retesting$$

---

7. Design of Test Cases

Definition

A test case is a set of inputs, conditions, and expected outputs used to test software.

Components of Test Case

1. Test Case ID
2. Input Data
3. Expected Output
4. Actual Output
5. Result Status

Characteristics of Good Test Cases

- Simple
- Clear
- Reusable
- Complete

Example

Test Case	Input	Expected Output
Login Test	Correct password	Login successful

---

8. Functional Testing

Definition

Functional testing checks whether software functions according to requirements.
It is also called **Black Box Testing**.

Types of Functional Testing

1. Boundary Value Analysis
2. Equivalence Class Testing
3. Decision Table Testing
4. Cause Effect Graphing

A. Boundary Value Analysis (BVA)

Definition

Tests boundary values where errors commonly occur.

Example

If valid age range is 18–60:
Test values:

- 17
- 18
- 19
- 59

- 60
- 61

Advantages

- Detects edge-case errors
- Reduces test cases

B. Equivalence Class Testing

Definition

Input data is divided into equivalent groups (classes).
Only one value from each class is tested.

Example

Age field:

- Invalid: Below 18
- Valid: 18–60
- Invalid: Above 60

Advantages

- Reduces redundant testing
- Saves time

C. Decision Table Testing

Definition

Decision tables represent combinations of conditions and actions.

Example

Username Valid	Password Valid	Result
Yes	Yes	Login Success
Yes	No	Error
No	Yes	Error
No	No	Error

Advantages

- Handles complex business rules

D. Cause Effect Graphing

Definition

Cause-effect graphing shows relationships between inputs (causes) and outputs (effects).

Example

Cause:

- Valid password

Effect:

- Access granted

Advantages

- Better logical testing
- Identifies combinations

---

9. Structural Testing

Definition

Structural testing checks the internal structure and logic of the code.
It is also called **White Box Testing**.

Types

1. Path testing
2. Loop testing
3. Condition testing

Advantages

- Detects logical errors
- Improves code quality

---

10. Cyclomatic Complexity Measures

Definition

Cyclomatic Complexity measures the complexity of a program by counting independent execution paths.
Developed by Thomas McCabe.

Formula

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

Where:

- E = Number of edges
- N = Number of nodes
- P = Number of connected components

Importance

- Measures complexity
- Determines number of test cases

Complexity Levels

Complexity	Risk
1–10	Low
11–20	Medium
Above 20	High

---

11. Control Flow Graph (CFG)

Definition

CFG is a graphical representation of program control flow.

Components

1. Nodes → Statements
2. Edges → Flow paths

Example Flow

$$Start \rightarrow Condition \rightarrow True\ Path \rightarrow End$$

Advantages

- Helps path testing
- Calculates cyclomatic complexity

---

12. Path Testing

Definition

Path testing ensures all independent paths in a program are tested.

Objectives

- Detect logical errors
- Achieve complete path coverage

Steps

1. Draw CFG
2. Calculate complexity
3. Identify independent paths
4. Design test cases

---

13. Data Flow Testing

Definition

Data flow testing checks the lifecycle of variables:

- Definition
- Usage
- Deletion

Objective

Detect improper variable usage.

Example Errors

- Uninitialized variables
- Unused variables

---

14. Mutation Testing

Definition

Mutation testing evaluates test quality by introducing small code changes (mutations).

Purpose

Check whether test cases detect errors.

Example

Original:

if(a > b)
Mutated:
if(a < b)

Advantages

- Measures test effectiveness

Disadvantages

- Time-consuming

---

15. Unit Testing

Definition

Unit testing tests individual modules or functions independently.

Characteristics

- Smallest testing level
- Usually performed by developers

Advantages

- Early error detection
- Easier debugging

Example

Testing login function separately.

---

16. Integration Testing

Definition

Integration testing checks interaction between combined modules.

Types

1. Top-Down Integration

Testing starts from top modules.

---

2. Bottom-Up Integration

Testing starts from lower modules.

---

3. Big Bang Integration

All modules combined together.

Objectives

- Detect interface errors
- Verify module interaction

---

17. System Testing

Definition

System testing tests the complete integrated software system.

Types

1. Performance testing
2. Security testing
3. Stress testing
4. Recovery testing

Objectives

- Validate complete system
- Ensure requirement satisfaction

---

18. Debugging

Definition

Debugging is the process of finding and fixing software defects.

Steps in Debugging

1. Identify error
2. Locate cause
3. Correct error
4. Retest program

Types of Errors

Error Type	Description
Syntax Error	Grammar mistakes
Logical Error	Incorrect logic
Runtime Error	Error during execution

Debugging Flow

$$Error\ Detection \rightarrow Error\ Localization \rightarrow Error\ Correction \rightarrow Retesting$$

---

19. Alpha and Beta Testing

A. Alpha Testing

Definition

Alpha testing is performed by developers or internal testers before software release.

Features

- Conducted in controlled environment
- Detects major bugs

B. Beta Testing

Definition

Beta testing is performed by real users in a real environment before final release.

Features

- User feedback collection
- Real-world testing

Comparison

Alpha Testing	Beta Testing
Internal testing	External testing
Controlled environment	Real environment
Performed before beta	Performed before final release

---

20. Testing Tools and Standards

A. Testing Tools

Definition

Testing tools automate software testing activities.

Popular Testing Tools

Tool	Purpose
Selenium	Web testing
JUnit	Java unit testing
TestNG	Automated testing
LoadRunner	Load testing

Advantages of Testing Tools

1. Faster testing
2. Better accuracy
3. Automated execution

B. Testing Standards

Definition

Testing standards define guidelines and best practices for software testing.

Common Standards

Standard	Purpose
IEEE 829	Test documentation
ISO 9126	Software quality
ISO/IEC 25010	Software quality model

Advantages of Standards

1. Improved quality
2. Better documentation
3. Standardized testing process

Conclusion

Software development and testing are critical phases of software engineering. Proper coding practices, documentation, and testing techniques ensure high-quality, reliable, and maintainable software. Functional testing, structural testing, debugging, and automated testing tools help identify defects and improve software performance before deployment.

Unit – 6: Software Maintenance, Reliability and Quality Management

1. Software Maintenance

Definition

Software Maintenance is the process of modifying, updating, and improving software after delivery to users.

Maintenance is necessary because:

- User requirements change
- Errors are discovered
- Technology evolves

Objectives of Software Maintenance

1. Correct software defects
2. Improve performance
3. Add new features
4. Adapt software to new environments

Importance of Software Maintenance

- Ensures long-term usability
- Increases software life
- Improves customer satisfaction

---

2. Management of Maintenance

Definition

Maintenance management involves planning, organizing, controlling, and monitoring maintenance activities.

Responsibilities in Maintenance Management

1. Change management
2. Resource allocation
3. Cost control
4. Quality assurance
5. Documentation management

Maintenance Team Responsibilities

- Bug fixing
- Updating software
- Testing modifications
- Supporting users

Challenges in Maintenance Management

1. Poor documentation
2. Complex legacy systems
3. Changing technologies
4. Time constraints

---

3. Maintenance Process

Definition

The maintenance process is a sequence of activities performed to modify software after deployment.

Steps in Maintenance Process

1. Problem Identification

Finding errors or required changes.

---

2. Analysis

Studying the impact of changes.

---

3. Design Modification

Updating software design if required.

---

4. Coding

Implementing modifications.

---

5. Testing

Checking whether modifications work properly.

---

6. Deployment

Releasing updated software.

Maintenance Process Flow

$$Problem\ Identification \rightarrow Analysis \rightarrow Design\ Modification \rightarrow Coding \rightarrow Testing \rightarrow Deployment$$

---

4. Types of Software Maintenance

1. Corrective Maintenance

Definition

Fixing software errors and bugs.

Example

Fixing login failure bug.

---

2. Adaptive Maintenance

Definition

Modifying software to work in a changed environment.

Example

Updating software for a new operating system.

---

3. Perfective Maintenance

Definition

Improving software performance or adding features.

Example

Adding dark mode feature.

---

4. Preventive Maintenance

Definition

Improving software to prevent future problems.

Example

Refactoring complex code.

Maintenance Distribution

Typical Software Maintenance Distribution

Approximate effort distribution among maintenance types.

Adaptive

Corrective

Perfective

Preventive

---

5. Maintenance Models

Definition

Maintenance models describe methods used to manage software maintenance activities.

Types of Maintenance Models

A. Quick-Fix Model

Definition

Directly modifies code without major documentation updates.

Advantages

- Fast solution

Disadvantages

- Poor maintainability

B. Iterative Enhancement Model

Definition

Software is improved gradually through repeated cycles.

Advantages

- Better quality
- Easier maintenance

C. Reuse-Oriented Model

Definition

Uses reusable components during maintenance.

---

6. Regression Testing

Definition

Regression testing ensures that software modifications do not affect existing functionality.

Objectives

1. Verify old features still work
2. Detect side effects of changes

Example

After adding payment functionality:

- Login module

- Registration module
  must still work correctly.

Advantages

- Improves reliability
- Detects hidden bugs

Regression Testing Process

$$Code\ Modification \rightarrow Retesting\ Existing\ Functions \rightarrow Detect\ Side\ Effects$$

---

7. Reverse Engineering

Definition

Reverse Engineering is the process of analyzing existing software to understand its structure and functionality.

Objectives

1. Understand old systems
2. Recover lost documentation
3. Improve maintainability

Activities

- Code analysis
- Design recovery
- Documentation extraction

Advantages

- Better system understanding
- Helps modernization

Disadvantages

- Time-consuming
- Difficult for large systems

---

8. Software Reengineering

Definition

Software Reengineering is the process of modifying and improving existing software to make it better and easier to maintain.

Activities in Reengineering

1. Reverse engineering
2. Code restructuring
3. Data restructuring
4. Forward engineering

Difference Between Reverse Engineering and Reengineering

Reverse Engineering	Reengineering
Understanding software	Improving software

Reverse Engineering	Reengineering
No modification required	Modification required

Reengineering Process

$Existing\ Software \rightarrow Reverse\ Engineering \rightarrow Restructuring \rightarrow Improved\ Software$

Advantages of Reengineering

1. Reduced maintenance cost
2. Improved quality
3. Better performance

---

9. Configuration Management

Definition

Software Configuration Management (SCM) is the process of controlling and tracking software changes.

Objectives of SCM

1. Version control
2. Change management
3. Build management
4. Release management

Activities of SCM

1. Configuration Identification

Identifying software components.

---

2. Version Control

Managing different software versions.

---

3. Change Control

Handling change requests.

---

4. Configuration Audit

Checking consistency of software products.

Popular SCM Tools

Tool	Purpose
Git	Version control
GitHub	Source code hosting
Subversion	Centralized version control

Advantages of SCM

1. Better teamwork
2. Easy rollback
3. Improved change tracking

---

10. Documentation

Definition

Documentation is written information describing software systems and processes.

Types of Documentation

A. User Documentation

Helps end users operate software.

Examples

- User manuals
- Help guides

B. Technical Documentation

Helps developers maintain software.

Examples

- Design documents
- Source code comments

Importance of Documentation

1. Easier maintenance
2. Better understanding
3. Improved communication

---

11. Software Reliability

Definition

Software Reliability is the probability that software works without failure for a specified period under specified conditions.

Characteristics of Reliable Software

1. Correctness
2. Consistency
3. Fault tolerance
4. Stability

Factors Affecting Reliability

1. Software complexity
2. Coding quality
3. Testing quality
4. User environment

---

12. Reliability Metrics

Definition

Reliability metrics measure software reliability quantitatively.

Important Reliability Metrics

1. Mean Time Between Failures (MTBF)

Definition

Average time between two failures.

Formula

$$MTBF = MTTF + MTTR$$

Where:

- MTTF = Mean Time To Failure
- MTTR = Mean Time To Repair

---

2. Failure Rate

Formula

$$Failure\ Rate = \frac{Number\ of\ Failures}{Time}$$

---

3. Availability

Formula

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

Reliability Metrics Comparison

Importance of Reliability Metrics

Relative importance of common software reliability measures.

02468MTBFFailure RateAvailability

---

13. Software Quality Management

Definition

Software Quality Management (SQM) ensures that software products meet required quality standards.

Objectives of SQM

1. Improve software quality
2. Reduce defects
3. Increase customer satisfaction
4. Ensure standard compliance

Activities of Quality Management

1. Quality Planning
2. Quality Assurance
3. Quality Control
4. Continuous Improvement

---

14. ISO 9000

Definition

International Organization for Standardization developed ISO 9000 standards for quality management systems.

Purpose of ISO 9000

- Ensure consistent quality
- Improve organizational processes

Features of ISO 9000

1. Process-oriented approach
2. Customer focus
3. Continuous improvement
4. Documentation standards

Advantages of ISO 9000

1. Better quality management
2. Increased customer trust
3. Improved productivity

ISO 9000 Quality Cycle

$$Planning \rightarrow Implementation \rightarrow Monitoring \rightarrow Improvement$$

---

15. SEI CMM (Capability Maturity Model)

Definition

SEI CMM is a framework developed by the Software Engineering Institute to measure software process maturity.

Objectives of CMM

1. Improve software process quality
2. Standardize development practices
3. Increase productivity

Five Levels of CMM

Level	Name	Description
1	Initial	Unpredictable process
2	Repeatable	Basic project management
3	Defined	Standardized processes
4	Managed	Measured and controlled
5	Optimizing	Continuous improvement

CMM Maturity Progression

SEI CMM Maturity Levels
Progression of software process maturity levels.
0246Level 1Level 2Level 3Level 4Level 5

Advantages of CMM

1. Better software quality
2. Reduced project risk
3. Improved productivity
4. Better process control

Difference Between ISO 9000 and CMM

ISO 9000	CMM
Quality standard	Process maturity model
Applicable to all industries	Mainly software industry
Focus on quality system	Focus on process improvement

Conclusion

Software maintenance and quality management are essential for long-term software success. Maintenance activities such as regression testing, reverse engineering, and reengineering improve software usability and lifespan. Reliability metrics, ISO 9000 standards, and SEI CMM help organizations achieve high-quality, reliable, and maintainable software systems.