MAA OMWATI DEGREE COLLEGE (HASSANPUR)

SOFTWARE ENGINEERING

Program : BCA 5TH SEM

IMPORTANT NOTES FOR EXAM

UNIT I – Introduction to Software Engineering

1. Software Engineering

--

Software Engineering is a systematic, disciplined, and measurable approach to the development, operation, testing, maintenance, and management of software systems. It applies engineering principles, methods, tools, and techniques to produce reliable, efficient, secure, and high-quality software. Software engineering focuses on developing software within time, cost, and

resource constraints while satisfying user requirements. It includes activities such as requirement analysis, design, coding, testing, deployment, and maintenance. The main objective of software engineering is to overcome software development challenges, reduce failures, improve productivity, and ensure that software products are dependable, maintainable, and capable of adapting to future changes.

---

2. Software and Its Characteristics

--

Software is a collection of programs, procedures, data, and related documentation that instructs a computer system to perform specific tasks. Unlike hardware, software is a logical product that does not physically wear out but can become outdated due to changing requirements and technology. Software is developed through engineering processes and can be modified, improved, and maintained throughout its life cycle. Important characteristics of software include reliability, efficiency, usability, maintainability, portability, security, and reusability. Software plays an essential role in modern computing systems by controlling hardware operations, processing information, and providing solutions for various user and organizational needs.

Detailed Notes

What is Software?

Software consists of:

1. **Programs** – Set of instructions executed by computers.
2. **Data** – Information processed by programs.
3. **Documentation** – Manuals and instructions related to software.

Types of Software

1. System Software

Controls computer hardware and provides a platform for applications.

Examples:

- Operating systems
- Device drivers
- Compilers

---

2. Application Software

Designed to perform user-specific tasks.

Examples:

- Banking applications
- Word processors
- Web browsers

---

3. Embedded Software

Software integrated into hardware devices.

Examples:

- Automobile control systems
- Medical devices

---

4. Web Applications

Software accessed through web browsers.

Examples:

- Online shopping websites
- Social media applications

Characteristics of Software

1. Intangible

Software cannot be physically touched.

---

2. Developed, Not Manufactured

Hardware is manufactured, but software is developed through programming.

---

3. Does Not Wear Out

Software does not physically degrade but becomes outdated.

---

4. Highly Complex

Large software systems contain millions of lines of code.

---

5. Easy to Modify
Software can be changed according to requirements.

---

6. Maintenance Intensive
Software requires continuous updates and improvements.

---

7. Reusability
Software components can be reused in different applications.

---

3. Evolving Role of Software
--
The **Evolving Role of Software** refers to the continuous expansion and transformation of software from simple computational programs to complex systems that support communication, automation, business operations, artificial intelligence, and decision-making. Earlier, software was mainly used for scientific calculations and data processing. Today, software has become a critical component of almost every industry, including healthcare, education, finance, transportation, and entertainment. Modern software supports cloud computing, mobile applications, IoT, artificial intelligence, and digital transformation. The evolving role of software requires improved development methods, security practices, scalability, and maintenance approaches to meet changing technological and societal needs.

Detailed Notes
Evolution of Software
1. Early Era

- Software used mainly for scientific calculations.
- Programs were written directly in machine language.

---

2. Business Era

- Software used for business data processing.
- Database systems became popular.

---

3. Personal Computer Era

- Desktop applications increased.
- User-friendly software developed.

---

4. Internet Era

- Web applications and online services became common.

---

5. Mobile and Cloud Era

- Mobile apps, cloud computing, AI, and IoT became important.

Modern Role of Software
Software is used for:

- Automation
- Communication
- Data analysis
- Artificial intelligence
- Digital services
- Business management
- Cybersecurity

---

4. Software Product
--
A **Software Product** is a complete software system developed to satisfy specific user requirements and deliver value to customers or organizations. It includes executable programs, documentation, configuration files, user manuals, and supporting materials required for operation and maintenance. Software products can be commercial, customized, open-source, or embedded systems. A good software product should provide functionality, reliability, usability, efficiency, security, and maintainability. Software products are developed through systematic processes involving requirements analysis, design, implementation, testing, deployment, and maintenance. Successful software products solve user problems, adapt to changing requirements, and provide continuous improvements throughout their operational life cycle.

Components of Software Product

1. Software Programs
Executable code that performs required functions.

---

2. Documentation
Contains:

- User manuals
- Technical documents
- Installation guides

---

3. Configuration Data
Settings required for software operation.

---

4. Support Materials
Includes maintenance and training resources.

Types of Software Products
1. Generic Products
Developed for general users.
Examples:

- MS Office
- Web browsers

---

2. Customized Products
Developed for specific customers.
Examples:

- Banking systems
- Hospital management systems

Characteristics of Good Software Product

- Reliable
- Secure
- Efficient
- User-friendly
- Maintainable
- Portable
- Scalable

---

5. Software Processes
--
A **Software Process** is a structured set of activities, methods, and practices used to develop, deliver, and maintain software systems. It defines how software requirements are analyzed, designed, implemented, tested, deployed, and improved. A software process provides a framework for managing software development projects effectively by defining tasks, responsibilities, resources, and timelines. Common software process activities include specification, development, validation, and evolution. Software processes help organizations produce high-quality software within planned cost and time limits. Well-defined processes improve productivity, reduce risks, ensure consistency, and support continuous improvement throughout the software development life cycle.

Main Activities of Software Process
1. Software Specification
Determining customer requirements.
Activities:

- Requirement gathering
- Analysis
- Documentation

---

2. Software Development
Creating software.
Includes:

- Design

- Coding
- Integration

---

3. Software Validation
Checking software correctness.
Includes:

- Testing
- Reviews

---

4. Software Evolution
Modifying software after delivery.
Includes:

- Updates
- Maintenance

Common Software Process Models
1. Waterfall Model
Sequential development approach.
Phases:
Requirement → Design → Coding → Testing → Maintenance

---

2. Incremental Model
Software developed in small increments.

---

3. Spiral Model
Risk-driven iterative development.

---

4. Agile Model
Flexible development with continuous customer involvement.

---

6. Software Crisis
--
Software Crisis refers to the difficulties and failures experienced in software development during the early years of computing due to increasing software complexity, poor development practices, and inability to meet growing demands. The software crisis resulted in delayed projects, cost overruns, unreliable software, and maintenance difficulties. As software systems became larger and more complex, traditional programming methods were unable to handle development challenges effectively. Common problems included incorrect requirements, poor planning, insufficient testing, and lack of documentation. Software engineering emerged as a solution to the software crisis by introducing systematic methods, development processes, quality standards, and management techniques.

Causes of Software Crisis
1. Increasing Complexity
Software systems became larger and harder to manage.

---

2. Poor Project Management
Lack of proper planning caused delays.

---

3. Changing Requirements
Frequent changes increased development difficulty.

---

4. Lack of Standards
No common development practices existed.

---

5. Inadequate Testing
Caused unreliable software.

---

6. High Maintenance Cost
Poor design increased maintenance effort.

Effects of Software Crisis

- Project failures

- Cost overruns
- Delayed delivery
- Low-quality software
- Customer dissatisfaction

Solutions to Software Crisis

- Software engineering principles
- Better development methodologies
- Software testing
- Quality assurance
- Project management techniques

---

7. Software Engineering Approach and Challenges

--

The **Software Engineering Approach** is a systematic method of developing software by applying engineering principles, processes, tools, and techniques throughout the software life cycle. It focuses on producing high-quality software while controlling cost, time, and resources. The approach includes requirement analysis, design, implementation, testing, deployment, and maintenance. Software engineering addresses challenges such as managing complexity, changing requirements, security threats, reliability issues, and project constraints. Modern software development requires continuous improvement, collaboration, automation, and quality management. The software engineering approach helps organizations develop reliable, scalable, maintainable, and cost-effective software solutions.

Software Engineering Approach

1. Requirement Analysis

Understand user needs.

---

2. Design

Create software architecture and models.

---

3. Implementation

Write and integrate code.

---

4. Testing

Verify software quality.

---

5. Maintenance

Modify and improve software.

Challenges in Software Engineering

1. Complexity Management

Large systems are difficult to design.

---

2. Requirement Changes

Customer needs may change frequently.

---

3. Security

Protecting software from attacks.

---

4. Time and Cost Management

Completing projects within limits.

---

5. Quality Assurance

Ensuring reliable software.

---

6. Scalability

Supporting increasing users and data.

---

8. Principles of Software Engineering

--

Principles of Software Engineering are fundamental guidelines that help developers create reliable, maintainable, efficient, and high-quality software systems. These principles provide a foundation for effective software development and help manage

complexity throughout the software life cycle. Important software engineering principles include abstraction, modularity, information hiding, separation of concerns, simplicity, reusability, continuous improvement, and quality assurance. Following these principles improves software design, reduces errors, increases maintainability, and supports future changes. Software engineering principles encourage systematic development practices and help teams deliver software products that meet customer expectations within available resources and project constraints.

Important Software Engineering Principles

1. Abstraction

Focus on important details while hiding unnecessary information.
Example:
Using functions without knowing internal implementation.

---

2. Modularity

Divide software into smaller independent modules.
Advantages:

- Easy testing
- Easy maintenance

---

3. Separation of Concerns

Different problems should be handled separately.

---

4. Information Hiding

Internal details of modules should remain hidden.

---

5. Reusability

Existing components should be reused.
Benefits:

- Saves time
- Reduces cost

---

6. Simplicity

Keep design and code simple.

---

7. Maintainability

Software should be easy to modify.

---

8. Quality Focus

Quality should be considered throughout development.

---

9. Documentation

Maintain proper records of software design and operation.

Quick Revision Table

Topic	Key Points
Software	Programs, data, and documentation that perform tasks
Software Characteristics	Intangible, complex, modifiable, maintainable, reusable
Evolving Role	From calculation programs to AI, cloud, IoT, and digital systems
Software Product	Complete software solution with code, documents, and support
Software Process	Specification, development, validation, and evolution activities
Software Crisis	Problems caused by complexity, cost, delays, and poor quality
Software Engineering Approach	Systematic development using engineering principles
Principles	Abstraction, modularity, reuse, simplicity, quality, maintainability

1. Software Development Process Model

--

A **Software Development Process Model** is a structured framework that defines the activities, tasks, methods, and sequence followed during the development of software. It provides a systematic approach for planning, designing, implementing, testing, deploying, and maintaining software systems. Process models help software teams manage complexity, improve quality, control

costs, and complete projects within scheduled time limits. Different models are selected based on project requirements, complexity, risk factors, customer involvement, and available resources. Common software development models include Waterfall, Prototype, Iterative Enhancement, Time Boxing, Spiral, RAD, and Object-Oriented models. Each model has unique advantages and limitations.

Importance of Software Process Models

- Provides systematic development approach.
- Improves software quality.
- Reduces development risks.
- Helps in project management.
- Defines team responsibilities.
- Improves communication between developers and customers.

---

1. Waterfall Model

--

The **Waterfall Model** is a traditional linear sequential software development model where software development activities are performed step-by-step in a fixed sequence. Each phase must be completed before moving to the next phase. The model begins with requirement analysis and proceeds through design, implementation, testing, deployment, and maintenance. It emphasizes documentation and planning before development begins. The waterfall model is suitable for projects where requirements are clearly understood and unlikely to change. However, it is less flexible because changes in later stages are difficult and expensive. It provides simplicity, clear milestones, and effective project control.

Phases of Waterfall Model

```
Requirement Analysis
        ↓
System Design
        ↓
Implementation
        ↓
Testing
        ↓
Deployment
        ↓
Maintenance
```

Phases Explanation

1. Requirement Analysis

- Collect customer requirements.
- Prepare requirement specification document.

---

2. System Design

- Design software architecture.
- Define database and interfaces.

---

3. Implementation

- Developers write source code.

---

4. Testing

- Detect and fix defects.

---

5. Deployment

- Deliver software to users.

---

6. Maintenance

- Modify and improve software after delivery.

Advantages of Waterfall Model

- Simple and easy to understand.
- Clear development stages.
- Good documentation.

- Easy project management.
- Suitable for stable requirements.

Disadvantages

- Difficult to handle requirement changes.
- Testing occurs late.
- High risk for complex projects.
- Customer feedback comes late.

Applications

- Government projects.
- Banking systems with fixed requirements.
- Large documentation-based projects.

---

2. Prototype Model

--

The **Prototype Model** is a software development approach in which a working model or prototype of the software is created before developing the final system. The prototype helps users and developers understand requirements, identify missing features, and receive early feedback. It is especially useful when requirements are unclear or continuously changing. The prototype is evaluated by customers, improved based on feedback, and then used as a foundation for developing the final product. This model improves communication between users and developers, reduces requirement errors, and increases customer satisfaction. However, excessive prototype changes may increase development time and cost.

Prototype Model Process

```
Requirement Collection
        ↓
Quick Design
        ↓
Build Prototype
        ↓
Customer Evaluation
        ↓
Refinement
        ↓
Final System Development
```

Types of Prototypes

1. Throwaway Prototype

- Created only to understand requirements.
- Discarded after analysis.

---

2. Evolutionary Prototype

- Continuously improved into final software.

Advantages

- Better understanding of requirements.
- Early customer involvement.
- Reduces requirement errors.
- Improves user satisfaction.

Disadvantages

- Users may expect prototype as final product.
- Extra development effort.
- Poor design decisions may occur.

Applications

- User interface systems.
- Web applications.
- New innovative projects.

3. Iterative Enhancement Model

--

The **Iterative Enhancement Model** is a software development approach where the complete system is developed and delivered in multiple iterations or increments. Each iteration produces an improved version of the software by adding new features, correcting errors, and enhancing existing functionality. The initial version contains basic functionality, and subsequent iterations gradually increase system capabilities. Customer feedback is collected after each iteration and used for further improvements. This model reduces development risks, allows requirement changes, and provides early delivery of useful software. It is suitable for large projects where requirements evolve over time and continuous improvement is required.

Process

```
Initial Version
       ↓
Iteration 1
       ↓
Testing & Feedback
       ↓
Iteration 2
       ↓
Final Product
```

Characteristics

- Incremental development.
- Continuous feedback.
- Multiple releases.
- Flexible requirements.

Advantages

- Early working software delivery.
- Handles changing requirements.
- Reduces risks.
- Continuous customer involvement.

Disadvantages

- Requires good planning.
- Higher management complexity.
- Integration problems may occur.

Applications

- Large software systems.
- Web applications.
- Enterprise software.

---

4. Time Boxing Model

--

The **Time Boxing Model** is an iterative software development approach where development activities are divided into fixed time periods called time boxes. Each time box has a specific duration, objectives, resources, and deliverables. The development team focuses on completing selected features within the given time limit. If all planned features cannot be completed, lower-priority features are postponed to future time boxes. This model emphasizes rapid delivery, strict scheduling, customer involvement, and continuous improvement. It is commonly used in agile development environments where frequent releases and quick responses to changing requirements are important.

Features of Time Boxing

- Fixed development period.
- Prioritized features.
- Regular releases.
- Continuous feedback.

Time Box Structure

```
Planning
   ↓
Development
   ↓
Testing
   ↓
Release
   ↓
Next Time Box
```

Advantages

- Faster delivery.
- Better time management.
- Encourages teamwork.
- Supports changing requirements.

Disadvantages

- May reduce quality due to time pressure.
- Difficult feature estimation.
- Requires experienced teams.

Applications

- Agile projects.
- Mobile applications.
- Internet-based applications.

---

5. Spiral Model

--

The **Spiral Model** is a risk-driven software development model that combines iterative development with systematic risk analysis. It represents software development as a series of spiral cycles, where each cycle includes planning, risk analysis, engineering, and evaluation activities. The main focus of this model is identifying and reducing risks before proceeding further. Customer feedback is obtained regularly, and prototypes may be developed during each cycle. The spiral model is suitable for large, complex, and high-risk projects where requirements may change. Although it provides better risk management and flexibility, it requires expert risk analysis and can be expensive.

Spiral Model Phases

```
Planning
    ↓
Risk Analysis
    ↓
Engineering
    ↓
Customer Evaluation
    ↓
Next Spiral
```

Activities

1. Planning

Identify objectives and requirements.

---

2. Risk Analysis

Identify possible risks.

---

3. Engineering

Develop and test software.

---

4. Evaluation

Customer reviews progress.

Advantages

- Excellent risk management.

- Handles changing requirements.
- Customer feedback included.
- Suitable for complex projects.

Disadvantages

- Expensive.
- Requires risk expertise.
- Complex management.

Applications

- Defense systems.
- Banking applications.
- Large enterprise systems.

---

6. RAD Model (Rapid Application Development)

--

The **RAD Model (Rapid Application Development)** is a software development model that focuses on rapid development, prototype creation, user involvement, and quick delivery of software products. It uses reusable components, automated tools, and parallel development activities to reduce development time. RAD emphasizes continuous customer feedback and frequent improvements during development. The model divides the project into smaller modules that can be developed and integrated quickly. RAD is suitable for projects requiring fast delivery and where requirements are well understood. However, it requires skilled developers, active user participation, and sufficient resources to achieve successful results.

RAD Phases

```
Requirements Planning
        ↓
User Design
        ↓
Construction
        ↓
Cutover
```

Features

- Fast development.
- Reusable components.
- Continuous feedback.
- Prototype-based development.

Advantages

- Reduces development time.
- High customer involvement.
- Faster product delivery.
- Better user satisfaction.

Disadvantages

- Requires skilled developers.
- Not suitable for large systems.
- Depends on user involvement.

Applications

- Business applications.
- Database applications.
- Web-based systems.

---

7. Object-Oriented Models

--

Object-Oriented Models are software development approaches that organize software systems around objects rather than functions or procedures. An object represents real-world entities containing data and behavior. Object-oriented models use concepts such as classes, objects, inheritance, encapsulation, abstraction, and polymorphism to design and develop software.

These models improve software organization, reusability, flexibility, and maintainability. Development activities include object-oriented analysis, object-oriented design, implementation, and testing. Object-oriented models are widely used for complex software systems because they support modular design and easier modification. Examples of object-oriented languages include Java, C++, Python, and C#.

Main Concepts

1. Object

An entity containing data and operations.
Example:
Student object.

---

2. Class

Blueprint for creating objects.
Example:
Student class.

---

3. Encapsulation

Combining data and methods together.

---

4. Inheritance

Reusing properties of existing classes.

---

5. Polymorphism

One interface with multiple implementations.

---

6. Abstraction

Showing important features and hiding details.

Advantages

- High reusability.
- Better maintainability.
- Easier modification.
- Real-world representation.

Disadvantages

- Requires skilled developers.
- More design effort.
- Complex for beginners.

Applications

- Enterprise software.
- Mobile applications.
- Real-time systems.

Comparison of Software Development Models

Model	Approach	Requirement Changes	Customer Involvement	Risk Handling
Waterfall	Sequential	Very difficult	Low	Low
Prototype	Prototype-based	Easy	High	Medium
Iterative Enhancement	Incremental	Easy	High	Medium
Time Boxing	Time-based iterations	Easy	High	Medium
Spiral	Risk-driven	Very easy	High	Excellent
RAD	Rapid development	Moderate	High	Medium
Object-Oriented	Object-based	Easy	Medium	Good

Selection of Models

Project Type	Suitable Model
Fixed requirements	Waterfall

Project Type	Suitable Model
Unclear requirements	Prototype
Large evolving projects	Iterative Model
Fast delivery projects	RAD
High-risk projects	Spiral
Agile projects	Time Boxing
Complex object-based systems	Object-Oriented Model

. Software Project Management

--

Software Project Management is the process of planning, organizing, monitoring, controlling, and coordinating software development activities to achieve project objectives within specified time, cost, and quality constraints. It involves managing resources, schedules, risks, teams, requirements, and software processes throughout the project life cycle. The main goal of software project management is to deliver high-quality software successfully while minimizing risks and maximizing productivity. It includes activities such as project planning, estimation, scheduling, resource allocation, risk management, quality control, and progress tracking. Effective software project management ensures smooth execution, better communication, and successful completion of software projects.

Importance of Software Project Management

- Ensures timely software delivery.
- Controls project cost.
- Improves resource utilization.
- Reduces development risks.
- Maintains software quality.
- Improves communication among stakeholders.
- Helps achieve customer satisfaction.

Software Project Management Activities

--

Software Project Management Activities are the set of tasks performed by project managers to successfully plan, execute, monitor, and control software projects. These activities include project planning, effort estimation, scheduling, resource management, team coordination, risk management, quality assurance, progress monitoring, and reporting. Project managers are responsible for making decisions, solving problems, managing changes, and ensuring that the project meets customer requirements. Effective management activities provide a structured approach for software development and help organizations complete projects within planned time, budget, and quality standards. These activities are essential for reducing failures and improving overall project performance.

Major Management Activities

1. Project Planning

Determines:

- Project objectives.
- Scope.
- Resources.
- Schedule.
- Cost estimation.

---

2. Project Organization

Includes:

- Creating project teams.
- Assigning responsibilities.
- Defining roles.

---

3. Resource Management

Manages:

- Human resources.
- Hardware.
- Software tools.

- Budget.

---

4. Project Monitoring and Control
Tracks:

- Progress.
- Cost.
- Quality.
- Schedule.

---

5. Risk Management
Identifies and handles possible project problems.

---

6. Communication Management
Ensures proper communication between:

- Developers
- Customers
- Managers
- Stakeholders

---

7. Quality Management
Ensures software meets required standards.

Role of Software Project Manager
A project manager is responsible for:

- Planning project activities.
- Managing development teams.
- Allocating resources.
- Monitoring progress.
- Handling risks.
- Communicating with customers.
- Ensuring successful delivery.

---

2. Project Planning
--
Project Planning is the process of defining project goals, identifying required activities, estimating resources, preparing schedules, and developing strategies to complete a software project successfully. It is the first major activity in software project management and provides a roadmap for development. Project planning includes scope definition, effort estimation, cost estimation, resource allocation, risk identification, and quality planning. A proper project plan helps teams understand responsibilities, manage time effectively, control costs, and handle unexpected problems. It improves coordination among team members and increases the probability of successful project completion within the required constraints.

Objectives of Project Planning

- Define project scope.
- Estimate project cost.
- Allocate resources.
- Create development schedule.
- Identify risks.
- Establish quality standards.

Project Planning Activities
1. Scope Definition
Defines:

- What will be developed.
- Project boundaries.
- Deliverables.

---

2. Effort Estimation
Determines:

- Development effort.
- Required manpower.
- Project duration.

---

3. Cost Estimation
Calculates:

- Development cost.
- Resource cost.
- Operational cost.

---

4. Resource Planning
Identifies:

- Developers.
- Tools.
- Hardware.
- Infrastructure.

---

5. Risk Planning
Identifies possible problems and solutions.

---

6. Quality Planning
Defines:

- Testing methods.
- Standards.
- Quality goals.

Components of Project Plan
A project plan includes:
1. Introduction
Project overview and objectives.
2. Organization
Team structure and responsibilities.
3. Risk Analysis
Possible risks and solutions.
4. Hardware and Software Requirements
Required resources.
5. Work Schedule
Timeline and milestones.
6. Quality Plan
Quality assurance activities.

Advantages of Project Planning

- Reduces uncertainty.
- Improves coordination.
- Controls project cost.
- Helps meet deadlines.
- Reduces risks.

---

3. Project Scheduling
--
Project Scheduling is the process of defining project activities, arranging them in a logical sequence, assigning resources, and determining the time required to complete each task. It converts the project plan into a detailed timeline that guides development activities. Scheduling helps project managers monitor progress, identify delays, allocate resources effectively, and complete projects within deadlines. Important scheduling techniques include Gantt charts, Program Evaluation and Review Technique (PERT), and Critical Path Method (CPM). A well-prepared schedule improves coordination among team members, ensures efficient resource utilization, and increases the chances of successful software project completion.

Objectives of Project Scheduling

- Define task sequence.

- Estimate completion time.
- Allocate resources.
- Monitor project progress.
- Identify critical activities.

Scheduling Activities
1. Identify Activities
Break project into smaller tasks.
Example:

- Requirement analysis.
- Design.
- Coding.
- Testing.

---

2. Determine Dependencies
Identify relationships between tasks.
Example:
Testing depends on coding completion.

---

3. Estimate Duration
Calculate time required for each activity.

---

4. Assign Resources
Allocate:

- People.
- Tools.
- Equipment.

---

5. Create Schedule
Prepare timeline.

Scheduling Techniques

---

1. Gantt Chart
Definition
A Gantt chart is a graphical representation of project activities against time.
Example:

Activity	Week 1	Week 2	Week 3
Design	■■		
Coding		■■	■■
Testing			■■

Advantages

- Easy to understand.
- Shows progress clearly.
- Helps track deadlines.

---

2. PERT (Program Evaluation and Review Technique)
Definition
PERT is a scheduling technique used for projects where activity duration is uncertain.
Formula:
Expected Time=Optimistic+4(Most Likely)+Pessimistic6Expected\ Time=\frac{Optimistic+4(Most\ Likely)+Pessimistic}{6}Expected Time=6Optimistic+4(Most Likely)+Pessimistic

---

3. CPM (Critical Path Method)
Definition
CPM identifies the longest sequence of dependent activities that determines the minimum project completion time.

Critical Path

The path with:

- Longest duration.
- Zero slack time.

Factors Affecting Scheduling

- Project size.
- Team experience.
- Resource availability.
- Requirement changes.
- Technical complexity.

---

4. Risk Management Activities

--

Risk Management Activities are systematic processes used to identify, analyze, plan, monitor, and control risks that may affect software project success. A risk is an uncertain event that can negatively impact project cost, schedule, quality, or performance. Risk management helps project teams prepare solutions before problems occur. Major activities include risk identification, risk analysis, risk prioritization, risk planning, risk monitoring, and risk control. Effective risk management reduces project failures, improves decision-making, and increases confidence in project execution. It ensures that potential problems are handled proactively rather than causing unexpected project disruptions.

Software Project Risks

Definition

A software risk is a possible future event that may negatively affect project objectives.

Types of Software Risks

1. Project Risks

Affect project schedule and resources.

Examples:

- Staff shortage.
- Budget problems.

---

2. Technical Risks

Related to technology.

Examples:

- New technology failure.
- Performance issues.

---

3. Business Risks

Affect business goals.

Examples:

- Market changes.
- Customer rejection.

Risk Management Process

```
Risk Identification
        ↓
Risk Analysis
        ↓
Risk Prioritization
        ↓
Risk Planning
        ↓
Risk Monitoring
```

Risk Management Activities

---

1. Risk Identification

Meaning:

Finding possible risks that may affect the project.

Methods:

- Brainstorming.
- Expert opinion.
- Previous project analysis.

---

2. Risk Analysis
Meaning:
Evaluating the probability and impact of risks.
Factors:

- Probability of occurrence.
- Damage level.

---

3. Risk Prioritization
Risks are ranked according to importance.
High-risk items receive immediate attention.

---

4. Risk Planning
Develop strategies to handle risks.
Strategies:
Avoidance
Remove the cause of risk.
Example:
Change technology.

Reduction
Reduce probability or impact.
Example:
Additional testing.

Transfer
Move responsibility to another party.
Example:
Insurance.

Acceptance
Accept risk when impact is low.

---

5. Risk Monitoring
Continuous tracking of risks during development.
Activities:

- Review risks.
- Update risk status.
- Take corrective actions.

Risk Management Advantages

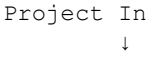
- Prevents project failure.
- Improves decision-making.
- Reduces unexpected problems.
- Controls project cost.
- Improves software quality.

Comparison: Project Planning vs Project Scheduling vs Risk Management

Project Planning	Project Scheduling	Risk Management
Defines project activities	Defines timeline	Handles uncertainties
Determines resources	Assigns duration	Identifies threats
Creates project roadmap	Creates task sequence	Creates solutions

---

Software Project Management Life Cycle



```
Planning
        ↓
Scheduling
        ↓
Development
        ↓
Monitoring
        ↓
Risk Management
        ↓
Project Completion
```

Quick Revision Table

Topic	Key Points
Software Project Management	Planning, organizing, monitoring, controlling software projects
Management Activities	Planning, resources, communication, quality, risk control
Project Planning	Defines scope, resources, cost, schedule, and risks
Project Scheduling	Arranges tasks and creates timeline using Gantt, PERT, CPM
Risk Management	Identifies, analyzes, plans, and controls project risks

UNIT II – Software Requirements Engineering

1. Software Requirements Engineering (SRE)

--

Software Requirements Engineering is the systematic process of identifying, analyzing, documenting, validating, and managing the needs and expectations of stakeholders for a software system. It is one of the most important phases of software development because it defines what the software should do and the constraints under which it must operate. Requirements engineering involves activities such as feasibility study, requirements elicitation, analysis, specification, validation, and management. The main objective of requirements engineering is to develop a clear understanding between customers and developers, reduce misunderstandings, avoid development errors, and ensure that the final software product satisfies user and business requirements.

Importance of Requirements Engineering

- Defines software objectives.
- Reduces development errors.
- Improves communication between customers and developers.
- Helps estimate cost and time.
- Provides a foundation for design and testing.
- Reduces project risks.

Requirements Engineering Process

--

The **Requirements Engineering Process** is a structured set of activities performed to discover, document, validate, and manage software requirements throughout the software life cycle. It transforms customer needs into a detailed requirement specification that guides software development. The process begins with feasibility studies and continues through requirements elicitation, analysis, specification, validation, and management. It ensures that requirements are complete, consistent, realistic, and understandable. A well-defined requirements engineering process improves software quality, reduces development costs, prevents requirement conflicts, and helps developers build systems that meet customer expectations. It is essential for successful software project planning and implementation.

Activities of Requirements Engineering Process

```
Feasibility Study
        ↓
Requirements Elicitation
        ↓
Requirements Analysis
```

```
        ↓
Requirements Specification
        ↓
Requirements Validation
        ↓
Requirements Management
```

---

1. Feasibility Study

--

A **Feasibility Study** is an initial evaluation process used to determine whether a proposed software project is practical, achievable, and beneficial. It analyzes technical, economic, operational, legal, and scheduling factors before development begins. The purpose of a feasibility study is to identify possible problems, estimate required resources, evaluate alternatives, and decide whether the project should proceed. It helps organizations avoid investing in unsuccessful projects by analyzing costs, benefits, risks, and available technologies. A feasibility study provides decision-makers with sufficient information to determine whether the software system can be developed successfully within available constraints.

Objectives of Feasibility Study

- Determine project possibility.
- Identify risks.
- Estimate cost and benefits.
- Analyze available resources.
- Select the best solution.

Types of Feasibility

1. Technical Feasibility

Meaning:

Checks whether required technology and skills are available.

Questions:

- Is required hardware available?
- Do developers have necessary skills?

Example:

Can the organization develop an AI-based system?

---

2. Economic Feasibility

Meaning:

Determines whether the project is financially beneficial.

Includes:

- Development cost.
- Maintenance cost.
- Expected benefits.

---

3. Operational Feasibility

Meaning:

Checks whether the system will work effectively in the organization.

Factors:

- User acceptance.
- Organizational support.

---

4. Legal Feasibility

Meaning:

Ensures the system follows laws and regulations.

Examples:

- Data privacy laws.
- Software licensing.

---

5. Schedule Feasibility

Meaning:

Determines whether the project can be completed within the required time.

Feasibility Study Report

Contains:

- Problem definition.
- Proposed solution.
- Cost estimation.
- Risk analysis.
- Recommendations.

---

2. Requirements Elicitation and Analysis
--
Requirements Elicitation and Analysis is the process of collecting, understanding, organizing, and analyzing the needs of stakeholders for a software system. It involves communication between customers, users, developers, and other stakeholders to identify functional and non-functional requirements. During elicitation, information is gathered using interviews, questionnaires, observations, workshops, and document analysis. Requirement analysis focuses on examining collected requirements, resolving conflicts, identifying dependencies, and prioritizing needs. The main goal is to produce clear, complete, and consistent requirements that can be used for software design and development. Effective elicitation and analysis reduce misunderstandings and improve software quality.

Requirements Elicitation Process

```
Identify Stakeholders
        ↓
Gather Requirements
        ↓
Analyze Requirements
        ↓
Resolve Conflicts
        ↓
Document Requirements
```

Sources of Requirements
1. Customers
Provide business requirements.

---

2. End Users
Provide practical usage needs.

---

3. Managers
Provide organizational requirements.

---

4. Existing Systems
Provide information about current functionality.

Requirements Elicitation Techniques

---

1. Interviews
Direct discussion with stakeholders.
Advantages:

- Detailed information.
- Better understanding.

---

2. Questionnaires
Collect information from many users.
Advantages:

- Saves time.
- Suitable for large groups.

---

3. Observation
Study users while performing tasks.
Advantages:

- Identifies actual user behavior.

---

4. Workshops

Group discussions between stakeholders.
Advantages:

- Quick requirement gathering.

---

5. Prototyping
Creating sample models for feedback.
Advantages:

- Clarifies unclear requirements.

Requirements Analysis Activities
1. Requirement Classification
Classify requirements into categories.

---

2. Requirement Prioritization
Identify important requirements.

---

3. Conflict Resolution
Remove contradictions between requirements.

---

4. Requirement Modeling
Represent requirements using diagrams and models.

---

3. Requirements Validation
--
Requirements Validation is the process of checking whether documented software requirements accurately represent customer needs and are suitable for software development. It ensures that requirements are correct, complete, consistent, realistic, and testable before implementation begins. Validation helps identify errors, missing information, conflicts, and unrealistic expectations at an early stage. It involves reviews, inspections, prototyping, and testing of requirements documents. The main purpose of requirements validation is to prevent costly changes during later development stages and ensure that the final software system satisfies stakeholder expectations. Effective validation improves software quality and reduces project risks.

Objectives of Requirements Validation

- Verify correctness.
- Detect missing requirements.
- Remove conflicts.
- Ensure feasibility.
- Confirm customer satisfaction.

Requirement Validation Checks
1. Validity Check
Does the requirement represent real user needs?

---

2. Consistency Check
Are requirements free from conflicts?

---

3. Completeness Check
Are all necessary requirements included?

---

4. Realism Check
Can requirements be implemented within available resources?

---

5. Verifiability Check
Can requirements be tested?

Requirements Validation Techniques
1. Requirements Reviews
Experts examine requirement documents.

---

2. Prototyping
Creates sample system for evaluation.

---

3. Test Case Generation
Checks whether requirements can be tested.

---

4. Automated Analysis
Uses tools to identify errors.

Advantages of Requirements Validation

- Reduces errors.
- Saves development cost.
- Improves communication.
- Prevents requirement failures.

---

4. Requirements Management
--
Requirements Management is the process of controlling, organizing, tracking, and managing changes to software requirements throughout the development life cycle. Since requirements often change due to business needs, technology changes, or customer feedback, proper management is necessary to maintain consistency and project control. Requirements management includes requirement identification, documentation, version control, change management, traceability, and impact analysis. It ensures that all stakeholders understand requirement changes and that modifications are properly evaluated before implementation. Effective requirements management reduces confusion, controls project scope, improves communication, and helps deliver software that continues to meet customer expectations.

Need for Requirements Management
Requirements change because of:

- Business changes.
- Customer feedback.
- New technologies.
- Market competition.
- Regulatory changes.

Requirements Management Activities

---

1. Requirement Identification
Assign unique identifiers to requirements.
Example:
REQ-001, REQ-002

---

2. Requirement Documentation
Maintain requirement records.
Documents include:

- SRS document.
- Change requests.

---

3. Change Management
Controls requirement modifications.
Steps:

```
Change Request
      ↓
Impact Analysis
      ↓
Approval
      ↓
Implementation
```

---

4. Requirement Traceability
Tracks relationship between:

- Requirements
- Design
- Code
- Test cases

5. Version Control
Maintains different versions of requirements.

Requirements Traceability Matrix (RTM)
Definition
A Requirements Traceability Matrix is a document that maps requirements with design components, source code, and test cases to ensure that all requirements are implemented and tested.

Benefits of Requirements Management

- Controls scope changes.
- Improves project tracking.
- Maintains documentation.
- Reduces development risks.
- Improves software quality.

Difference Between Requirements Validation and Verification

Requirements Validation	Requirements Verification
Checks whether requirements meet user needs	Checks whether requirements are correctly documented
"Are we building the right product?"	"Are we building the product correctly?"
Focuses on customer satisfaction	Focuses on correctness

Requirements Engineering Process Summary

Activity	Purpose
Feasibility Study	Determines project practicality
Elicitation	Collects stakeholder requirements
Analysis	Studies and organizes requirements
Specification	Documents requirements
Validation	Checks correctness
Management	Controls requirement changes

Quick Revision Table

Topic	Key Points
Requirements Engineering	Identifying, analyzing, documenting, validating, and managing requirements
Feasibility Study	Checks technical, economic, operational, legal, and schedule possibility
Elicitation & Analysis	Collecting and understanding stakeholder needs
Validation	Ensures requirements are correct, complete, and realistic
Management	Controls requirement changes throughout developmen

Software Requirements Analysis & Specifications

1. Software Requirements
--
Software Requirements are the descriptions of services, functions, features, and constraints that a software system must satisfy to meet the needs of users and stakeholders. They define what the software should do and the conditions under which it must operate. Requirements are collected during the requirements engineering process and serve as the foundation for software design, development, testing, and maintenance. They help developers understand customer expectations and provide a clear direction for building the system. Software requirements are generally classified into functional requirements and non-functional requirements. Well-defined requirements reduce errors, control project scope, and improve software quality.

Types of Software Requirements
1. Functional Requirements
Definition:

Functional requirements describe the specific services and operations that a software system must perform.
They define:

- What the system should do.
- User interactions.
- System behavior.

Examples:

- User login.
- Generate reports.
- Process payments.
- Search records.

---

2. Non-Functional Requirements
Definition:
Non-functional requirements describe quality attributes and constraints of the software system.
They define:

- How the system should perform.
- System limitations.

Examples:

- Security.
- Performance.
- Reliability.
- Usability.
- Scalability.

Difference Between Functional and Non-Functional Requirements

Functional Requirements	Non-Functional Requirements
Describe system functions	Describe system qualities
Define what system does	Define how system works
Easier to test	Often measured indirectly
Example: Login system	Example: Response time

---

2. Software Requirements Analysis
--
Software Requirements Analysis is the process of examining, organizing, and understanding the collected software requirements to identify what the system should accomplish. It involves studying user needs, resolving conflicts, identifying relationships between requirements, and preparing models that represent system behavior. Requirements analysis transforms informal customer expectations into clear and structured requirements that guide software design and implementation. It includes activities such as requirement classification, prioritization, modeling, feasibility checking, and validation. Effective requirements analysis helps prevent misunderstandings, reduces development risks, controls project scope, and ensures that the final software system meets stakeholder expectations.

Objectives of Requirements Analysis

- Understand customer needs.
- Identify system functions.
- Remove requirement conflicts.
- Define system boundaries.
- Prepare design foundation.

Activities of Requirements Analysis
1. Requirement Classification
Requirements are grouped into:

- Functional requirements.
- Non-functional requirements.
- User requirements.
- System requirements.

---

2. Requirement Prioritization

Requirements are ranked based on importance.
Example:

- High priority.
- Medium priority.
- Low priority.

---

3. Requirement Modeling
Represents requirements using:

- Data Flow Diagrams.
- UML diagrams.
- Models.

---

4. Requirement Negotiation
Resolves conflicts among stakeholders.

---

3. Structured Analysis
--
Structured Analysis is a traditional software analysis technique that focuses on understanding system functions and data flow before software design begins. It uses graphical modeling tools to represent system processes, data movement, and data storage. The main objective of structured analysis is to divide a complex system into smaller, manageable processes and understand how information moves through the system. It commonly uses tools such as Data Flow Diagrams (DFD), data dictionaries, process specifications, and structure charts. Structured analysis improves communication between users and developers, provides clear system documentation, and helps create accurate software requirements before implementation.

Features of Structured Analysis

- Process-oriented approach.
- Focuses on data movement.
- Uses graphical models.
- Supports top-down analysis.
- Improves documentation.

Structured Analysis Tools

1. Data Flow Diagram (DFD)
2. Data Dictionary
3. Process Specification
4. Decision Tables

---

4. Data Flow Diagram (DFD)
--
A **Data Flow Diagram (DFD)** is a graphical representation of how data moves through a software system. It shows the flow of information between external entities, processes, and data stores. DFDs are used during structured analysis to understand system functionality and represent system requirements clearly. They focus on input, processing, and output of data without describing implementation details. DFDs help analysts communicate system behavior with customers and developers. The main components of a DFD are processes, data flows, data stores, and external entities. DFDs can be created at different levels, such as Level 0, Level 1, and Level 2.

Components of DFD
1. Process
Represents an activity that transforms data.
Symbol:
Circle or rounded rectangle.
Example:
Calculate salary.

---

2. Data Flow
Represents movement of data.
Symbol:
Arrow.
Example:
Customer information.

3. Data Store
Stores information.
Symbol:
Open rectangle.
Example:
Database.

---

4. External Entity
Source or destination of data.
Symbol:
Rectangle.
Example:
Customer.

Levels of DFD

Level 0 DFD (Context Diagram)

- Highest-level view.
- Shows entire system as one process.

Example:
`Customer → Banking System → Customer`

Level 1 DFD

- Divides system into major processes.

Example:
Banking System:

- Account Management
- Transaction Processing

Level 2 DFD

- Further breaks processes into smaller activities.

Advantages of DFD

- Easy to understand.
- Improves communication.
- Shows data movement clearly.
- Helps identify missing processes.
- Useful for documentation.

Limitations of DFD

- Does not show program logic.
- Cannot represent timing.
- Large systems create complex diagrams.

---

5. Data Dictionary
--
A **Data Dictionary** is a centralized collection of information about data elements used in a software system. It describes the meaning, structure, format, source, and usage of data within the system. A data dictionary supports structured analysis by providing detailed descriptions of data flows, data stores, and data items shown in Data Flow Diagrams. It helps developers, analysts, and users maintain consistency in data definitions and improves communication among project members. A well-prepared data dictionary reduces confusion, prevents duplicate data definitions, and serves as an important documentation resource throughout software development and maintenance.

Contents of Data Dictionary
1. Data Name
Name of data element.
Example:
Customer_ID

---

2. Description
Meaning of data.
Example:
Unique customer identification number.

---

3. Data Type
Defines format.
Examples:

- Integer
- String
- Date

---

4. Size
Defines storage length.
Example:
10 characters.

---

5. Source
Where data comes from.

---

6. Destination
Where data is used.

Example Data Dictionary

Data Element	Type	Description
Customer_ID	Integer	Customer identification
Name	String	Customer name
Balance	Float	Account balance

Advantages of Data Dictionary

- Maintains data consistency.
- Improves documentation.
- Helps database design.
- Reduces confusion.
- Supports maintenance.

---

6. Object-Oriented Analysis (OOA)
--
Object-Oriented Analysis (OOA) is a software analysis approach that focuses on identifying objects, classes, relationships, and behaviors required in a software system. It models a system as a collection of interacting objects that represent real-world entities. OOA uses concepts such as objects, classes, inheritance, encapsulation, abstraction, and polymorphism to understand system requirements. The main purpose of object-oriented analysis is to create a clear representation of system structure and behavior before design and implementation. It improves software flexibility, reusability, maintainability, and communication between developers and stakeholders. UML diagrams are commonly used for object-oriented analysis.

Basic Concepts of OOA
1. Object
An entity having:

- Data.
- Behavior.

Example:
Student object.

---

2. Class
Blueprint for objects.
Example:
Student class.

---

3. Encapsulation

Combining data and methods together.

---

4. Inheritance
Creating new classes from existing classes.

---

5. Polymorphism
Same operation with different behaviors.

---

6. Abstraction
Showing essential features while hiding details.

UML Models Used in OOA

- Use Case Diagram.
- Class Diagram.
- Sequence Diagram.
- Activity Diagram.
- State Diagram.

Advantages of Object-Oriented Analysis

- Better system understanding.
- Supports reuse.
- Easier maintenance.
- Real-world modeling.
- Reduces complexity.

---

7. Software Requirement Specification (SRS)
--
A **Software Requirement Specification (SRS)** is a formal document that describes all functional and non-functional requirements of a software system. It defines what the software should do, system constraints, interfaces, performance requirements, and quality attributes. The SRS acts as an agreement between customers and developers by providing a clear understanding of the expected software behavior. It serves as a foundation for software design, coding, testing, and maintenance activities. A good SRS document should be complete, consistent, correct, understandable, and verifiable. It reduces requirement misunderstandings and helps developers build software according to customer expectations.

Need of SRS
--
The **Need of SRS** refers to the importance of preparing a Software Requirement Specification document before software development begins. SRS provides a clear understanding of customer requirements and acts as a communication medium between customers, developers, testers, and managers. It reduces ambiguity, prevents requirement conflicts, and provides a reference for design and testing activities. SRS helps estimate project cost, time, and resources accurately. It also supports future maintenance by documenting system behavior and constraints. A properly prepared SRS improves software quality, reduces development risks, controls project scope, and ensures that the final product satisfies user and business requirements.

Importance of SRS

- Provides clear requirements.
- Acts as a contract between customer and developer.
- Helps software design.
- Supports testing.
- Reduces development errors.
- Helps maintenance.

Characteristics of Good SRS
1. Correct
Requirements should accurately represent user needs.

---

2. Complete
All necessary requirements should be included.

---

3. Consistent
Requirements should not conflict.

4. Unambiguous
Each requirement should have only one meaning.

---

5. Verifiable
Requirements should be testable.

---

6. Traceable
Requirements should be linked with design and testing.

---

7. Modifiable
Easy to update when changes occur.

Components of SRS
1. Introduction
Contains:

- Purpose.
- Scope.
- Definitions.
- References.

---

2. Overall Description
Contains:

- Product perspective.
- Product functions.
- User characteristics.
- Constraints.

---

3. Functional Requirements
Describes system operations.
Examples:

- Login.
- Report generation.

---

4. Non-Functional Requirements
Describes quality attributes.
Examples:

- Security.
- Performance.

---

5. External Interface Requirements
Includes:

- User interfaces.
- Hardware interfaces.
- Software interfaces.

---

6. System Constraints
Defines limitations.
Examples:

- Hardware restrictions.
- Legal requirements.

Structure of SRS Document

```
1. Introduction
      |
2. Overall Description
      |
3. Functional Requirements
      |
4. Non-Functional Requirements
```

```
      |
5. External Interface Requirements
      |
6. System Constraints
      |
7. Appendices
```

Advantages of SRS

- Clear communication.
- Better planning.
- Easier testing.
- Reduces errors.
- Improves software quality.
- Helps maintenance.

Comparison: Structured Analysis vs Object-Oriented Analysis

Structured Analysis	Object-Oriented Analysis
Process-oriented	Object-oriented
Focuses on functions and data flow	Focuses on objects and classes
Uses DFD	Uses UML diagrams
Less reusable	Highly reusable
Traditional approach	Modern approach

Important Exam Questions

2 Marks Questions

1. Define Software Requirement Specification.
2. What is DFD?
3. Define Data Dictionary.
4. What is Object-Oriented Analysis?
5. List characteristics of SRS.
6. What are functional requirements?

---

5 Marks Questions

1. Explain Structured Analysis with tools.
2. Explain Data Flow Diagram and its components.
3. Describe Data Dictionary.
4. Explain Object-Oriented Analysis concepts.
5. Explain the need and characteristics of SRS.
6. Describe components and structure of SRS.
7. Compare Structured Analysis and Object-Oriented Analysis.

Quick Revision Table

Topic	Key Points
Software Requirements	Defines what software should do
Structured Analysis	Process-oriented analysis using DFD and data dictionary
DFD	Shows movement of data through system
Data Dictionary	Describes data elements and their properties
OOA	Models system using objects and classes
SRS	Formal document containing complete software requirements

Software Metrics and Measures

---

1. Software Metrics and Measures

--

Software Metrics are quantitative measures used to evaluate different characteristics of a software product, process, or project. They provide numerical information about software size, complexity, quality, performance, productivity, and development effort. Software metrics help software engineers make better decisions, estimate project costs, monitor progress, identify problems, and improve software quality. Metrics can be applied during all phases of the software development life cycle, including

requirements, design, coding, testing, and maintenance. Common software metrics include size metrics, complexity metrics, quality metrics, and project metrics. Effective use of software metrics improves management control and supports successful software development.

---

2. Software Measures

--

Software Measures are numerical values assigned to software attributes to quantify and evaluate different aspects of software development. A measure represents a direct observation or calculation of a software characteristic, such as the number of lines of code, number of defects, development effort, or execution time. Software measures provide basic data from which software metrics are derived. They help developers and managers analyze software performance, estimate resources, control development activities, and improve productivity. Examples of software measures include code size, number of modules, number of test cases, and number of errors detected during testing.

Difference Between Metrics and Measures

Software Measures	Software Metrics
Direct numerical values	Derived information from measures
Basic data collection	Analysis of collected data
Example: Number of lines of code	Example: Productivity per LOC

---

3. Need of Software Metrics

--

The **Need of Software Metrics** arises from the requirement to measure, control, and improve software development activities. Software metrics provide objective information about software size, quality, cost, complexity, and productivity. They help project managers estimate effort, schedule tasks, allocate resources, and monitor project progress. Metrics allow developers to identify defects, evaluate software quality, and make improvements during development. Without metrics, software projects depend mainly on assumptions and subjective judgments. Software metrics support better decision-making, reduce project risks, improve customer satisfaction, and help organizations achieve reliable and cost-effective software development.

Importance of Software Metrics

1. Project Estimation

Helps estimate:

- Cost
- Time
- Resources

---

2. Quality Improvement

Measures:

- Defects
- Reliability
- Maintainability

---

3. Project Monitoring

Tracks:

- Progress
- Productivity
- Performance

---

4. Risk Management

Identifies possible project problems early.

---

5. Decision Making

Provides measurable information for management decisions.

---

4. Benefits of Software Metrics

--

Benefits of Software Metrics include improved planning, better control, higher software quality, and efficient resource management during software development. Metrics provide measurable information that helps organizations understand project performance and identify areas requiring improvement. They support accurate estimation of development effort, cost, and schedule. Software metrics help detect defects, evaluate productivity, compare development processes, and improve decision-

making. They also provide historical data that can be used for future project planning. By applying appropriate metrics, organizations can reduce risks, increase software reliability, improve team performance, and deliver high-quality software products within planned constraints.

Major Benefits
1. Better Estimation
Provides accurate cost and effort prediction.

---

2. Improved Quality
Helps detect and reduce defects.

---

3. Increased Productivity
Measures developer performance.

---

4. Better Project Control
Tracks project progress.

---

5. Process Improvement
Helps improve development methods.

---

5. Size Metrics
--
Size Metrics are software metrics used to measure the size or volume of a software system. They provide information about the amount of software produced and help estimate development effort, cost, complexity, and maintenance requirements. Size metrics are important because larger software systems generally require more resources, testing, and maintenance activities. Common size measurement techniques include Lines of Code (LOC), Token Metrics, and Function Point Metrics. These measures help project managers compare software products, estimate productivity, and predict project requirements. Size metrics are widely used in software estimation models such as the COCOMO model.

Types of Size Metrics
1. Line of Code (LOC)
2. Token Metrics
3. Function Point Metrics

---

6. Line of Code (LOC)
--
Line of Code (LOC) is a software size metric that measures the size of a program by counting the number of source code lines written during development. It is one of the simplest and oldest software measurement techniques. LOC is used to estimate development effort, productivity, maintenance cost, and project complexity. A higher number of lines generally indicates a larger software system requiring more development effort. However, LOC depends on programming language and coding style, making comparisons between different languages difficult. Comments and blank lines may or may not be included depending on the measurement standard used.

LOC Types
1. Physical LOC
Counts actual lines in source files.
Includes:
- Code lines
- Comments
- Blank lines

---

2. Logical LOC
Counts executable statements.
Example:
One instruction = one LOC.

Advantages of LOC
- Simple measurement.
- Easy to calculate.
- Useful for productivity analysis.

Disadvantages of LOC

- Language dependent.
- Does not measure functionality.
- Difficult before coding begins.

Example
Program contains:
`5000 executable lines`
Therefore:
LOC=5000LOC = 5000LOC=5000

---

7. Token Metrics
--
Token Metrics are software size metrics that measure program size based on the number of tokens used in source code rather than counting lines. A token represents a basic element of a program, such as keywords, operators, identifiers, constants, and special symbols. Token metrics provide a more detailed measurement of software complexity because they analyze the actual programming elements used in the code. They are less dependent on programming style compared to LOC. Token metrics are useful for evaluating program complexity, comparing software components, and estimating development effort. They provide better understanding of software structure and coding characteristics.

Types of Tokens
1. Operators
Symbols representing operations.
Examples:

- , - , *

---

2. Operands
Data values and variables.
Examples:
x, y, number

Halstead Metrics
Token metrics are commonly associated with **Halstead Software Metrics**.
Important measures:
n1 = Number of unique operators
n2 = Number of unique operands
N1 = Total operators
N2 = Total operands

Program Vocabulary
n=n1+n2n=n1+n2n=n1+n2

Program Length
N=N1+N2N=N1+N2N=N1+N2

Advantages of Token Metrics

- More language independent.
- Measures complexity.
- Considers program structure.

Disadvantages

- Difficult calculation.
- Requires code analysis.

---

8. Function Point Metrics
--
Function Point Metrics are software size measurement techniques that measure software functionality from the user's perspective rather than counting lines of code. Function points evaluate the amount of functionality delivered by a software system based on inputs, outputs, files, queries, and interfaces. They are independent of programming language and can be estimated before coding begins. Function point analysis helps in effort estimation, cost prediction, productivity measurement, and

project comparison. It is widely used in business applications where user functionality is more important than code size. Function points provide a better measure of software size for modern software development environments.

Function Point Components
1. External Inputs (EI)
Data entered into the system.
Example:
Login information.

---

2. External Outputs (EO)
Information generated by system.
Example:
Reports.

---

3. External Queries (EQ)
User requests for information.
Example:
Search operation.

---

4. Internal Logical Files (ILF)
Data maintained by system.
Example:
Database tables.

---

5. External Interface Files (EIF)
Files used from external systems.

Function Point Formula
FP=UFP×VAFFP = UFP \times VAFFP=UFP×VAF
Where:

- FP = Function Points
- UFP = Unadjusted Function Points
- VAF = Value Adjustment Factor

Advantages of Function Points

- Language independent.
- Can be estimated early.
- Measures user functionality.
- Useful for business applications.

Disadvantages

- Calculation is complex.
- Requires expert judgment.

Comparison of Size Metrics

Metric	Basis	Advantage	Limitation
LOC	Number of code lines	Simple	Language dependent
Token Metrics	Program tokens	Measures complexity	Difficult calculation
Function Points	User functionality	Language independent	More complex

---

9. Software Project Estimation Models
--
Software Project Estimation Models are mathematical techniques used to predict the effort, cost, time, resources, and size required to develop a software system. Estimation models help project managers prepare realistic plans and allocate resources effectively. They use historical project data, software size measurements, complexity factors, and development characteristics to calculate required effort. Accurate estimation reduces project risks, prevents cost overruns, and improves project scheduling. Common estimation techniques include algorithmic models, expert judgment, estimation by analogy, and models such as COCOMO. These models support better decision-making and help organizations complete software projects successfully within planned constraints.

10. COCOMO Model

--

COCOMO (Constructive Cost Model) is a software cost estimation model developed by Barry Boehm to estimate software development effort, cost, and schedule based on the size of the software measured in Lines of Code (LOC). It uses mathematical formulas to calculate the amount of effort required for a project. COCOMO classifies software projects into different modes based on complexity and development environment. The three basic categories are Organic, Semi-detached, and Embedded modes. COCOMO helps project managers predict resources, estimate development time, and create realistic project plans. It is widely used for software project estimation and management.

COCOMO Project Modes

1. Organic Mode

Characteristics:

- Small projects.
- Experienced teams.
- Simple requirements.

Examples:

Small business applications.

---

2. Semi-Detached Mode

Characteristics:

- Medium-size projects.
- Mixed experience teams.
- Moderate complexity.

Examples:

Database management systems.

---

3. Embedded Mode

Characteristics:

- Complex systems.
- Strict constraints.
- Hardware interaction.

Examples:

Real-time systems.

Basic COCOMO Formula

Effort Calculation

E=a(KLOC)bE = a(KLOC)^bE=a(KLOC)b

Where:

- E = Effort (person-months)
- KLOC = Thousand lines of code
- a,b = Constants

Development Time

T=c(E)dT = c(E)^dT=c(E)d

Where:

- T = Development time
- c,d = Constants

COCOMO Models

1. Basic COCOMO

Uses only program size.

---

2. Intermediate COCOMO

Uses:

- Size
- Cost drivers

---

3. Detailed COCOMO

Includes phase-wise estimation.

Advantages of COCOMO

- Provides systematic estimation.
- Easy project planning.
- Uses historical data.
- Helps cost prediction.

Limitations of COCOMO

- Depends on accurate size estimation.
- Less suitable for modern agile projects.
- Requires historical data.

Quick Revision Table

Topic	Key Points
Software Metrics	Quantitative measurement of software attributes
Need	Estimation, quality control, decision-making
LOC	Measures size by counting code lines
Token Metrics	Measures size using operators and operands
Function Points	Measures user functionality
COCOMO	Estimates effort, cost, and schedule using software size

UNIT III – Software Design

---

1. Software Design

--

Software Design is the process of transforming software requirements into a detailed blueprint that describes the structure, components, interfaces, and data organization of a software system. It defines how the system will be developed and how different parts of the software will interact with each other. Software design acts as a bridge between requirements analysis and implementation. It includes architectural design, database design, interface design, and component-level design. A good software design improves software quality, maintainability, reliability, and performance. The main objective of software design is to create an efficient, understandable, reusable, and flexible software system.

Importance of Software Design

- Converts requirements into implementation plans.
- Reduces development complexity.
- Improves software quality.
- Supports maintenance and modification.
- Provides better communication among developers.
- Reduces development risks.

Basic Software Design Process

```
Requirements Analysis
          ↓
Architectural Design
          ↓
Detailed Design
          ↓
Interface Design
          ↓
Implementation
```

---

2. Basic Concepts of Software Design

--

Basic Concepts of Software Design are fundamental principles used by software engineers to create effective and maintainable software systems. These concepts provide guidelines for organizing software components and managing complexity. Important design concepts include abstraction, refinement, modularity, software architecture, information hiding, separation of concerns,

and design patterns. These principles help developers divide complex systems into smaller manageable parts and improve software flexibility and reusability. Good design concepts ensure that software is easy to understand, test, modify, and maintain. They provide a systematic approach for developing high-quality software that satisfies functional and non-functional requirements.

Major Design Concepts

---

1. Abstraction
Definition:
Abstraction focuses on showing essential features of a system while hiding unnecessary details.
Example:
ATM interface hides internal banking operations.
Advantages:

- Reduces complexity.
- Improves understanding.

---

2. Refinement
Definition:
Refinement is the process of gradually transforming a high-level design into detailed design.
Example:
System → Module → Function → Code

---

3. Modularity
Definition:
Modularity divides a large software system into smaller independent modules.
Benefits:

- Easy development.
- Easy testing.
- Better maintenance.

---

4. Information Hiding
Definition:
Information hiding hides internal details of a module and exposes only necessary information.
Benefits:

- Reduces dependency.
- Improves security.

---

5. Separation of Concerns
Definition:
Dividing software into separate sections where each section handles a specific responsibility.
Example:
User interface separated from database logic.

---

6. Software Architecture
Defines the overall structure of software components and their relationships.

---

3. Architectural Design
--
Architectural Design is the process of defining the overall structure, organization, and interaction of major components of a software system. It describes how software components are arranged, how they communicate, and how responsibilities are distributed among them. Architectural design provides a high-level view of the software system before detailed coding begins. It includes selecting architectural styles, designing system layers, identifying components, and defining interfaces. A good architecture improves software performance, scalability, reliability, security, and maintainability. It acts as a foundation for detailed design and implementation. Common architectural styles include layered architecture, client-server architecture, and repository architecture.

Objectives of Architectural Design

- Define system structure.
- Identify major components.
- Improve scalability.

- Support future changes.
- Manage complexity.

Types of Software Architecture

---

1. Layered Architecture
System is divided into layers.
Example:

```
Presentation Layer
        ↓
Business Logic Layer
        ↓
Database Layer
```

Advantages:

- Easy maintenance.
- Better organization.

---

2. Client-Server Architecture
Client requests services from server.
Example:
Web applications.

---

3. Repository Architecture
All data is stored in a central repository.
Example:
Database systems.

---

4. Pipe and Filter Architecture
Data passes through processing components.
Example:
Compiler systems.

---

4. Low-Level Design (Detailed Design)
--
Low-Level Design (LLD) is the process of designing the internal details of individual software components after completing architectural design. It describes the logic, algorithms, data structures, interfaces, and interactions of each module. Low-level design provides enough detail for programmers to write actual code. It focuses on module-level implementation and explains how each component performs its responsibilities. LLD includes class design, database structures, flowcharts, pseudocode, and detailed algorithms. A well-prepared low-level design improves coding efficiency, reduces errors, supports testing, and ensures that software components work correctly according to the system architecture.

Components of Low-Level Design

- Module design.
- Data structure design.
- Algorithm design.
- Database design.
- Interface specification.

Difference Between High-Level Design and Low-Level Design

High-Level Design	Low-Level Design
Defines overall architecture	Defines module details
System-level view	Component-level view
Identifies major components	Defines internal logic
Used by architects	Used by programmers

---

5. Modularization
--
Modularization is a software design technique that divides a large software system into smaller independent units called modules. Each module performs a specific function and communicates with other modules through well-defined interfaces.

Modularization reduces software complexity by allowing developers to design, implement, test, and maintain individual components separately. It improves reusability, reliability, and team productivity. Good modular design ensures high cohesion within modules and low coupling between modules. Examples of modules include login module, payment module, and reporting module. Modularization is one of the most important principles for developing flexible and maintainable software systems.

Advantages of Modularization

- Reduces complexity.
- Supports parallel development.
- Easier testing.
- Improves maintenance.
- Encourages reuse.

Characteristics of Good Modules

- Independent.
- Clearly defined purpose.
- High cohesion.
- Low coupling.
- Easy to understand.

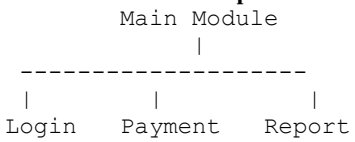
---

6. Design Structure Charts

--

A **Design Structure Chart** is a hierarchical diagram used to represent the organization and relationship of modules in a software system. It shows how a system is divided into modules and how these modules communicate with each other. Structure charts are mainly used in structured design methods to represent module calling relationships, data flow, and control flow. They help designers understand system architecture and improve modular design. A structure chart consists of modules, connections, and data/control information passed between modules. It provides a clear representation of software structure before implementation and supports better program organization.

Structure Chart Example



Components of Structure Chart

1. Module

Represents a functional unit.

---

2. Connection Line

Shows relationship between modules.

---

3. Data Couple

Shows data passed between modules.

---

4. Control Couple

Shows control information.

Advantages

- Shows module hierarchy.
- Improves communication.
- Helps coding.
- Supports maintenance.

---

7. Flow Charts

--

A **Flow Chart** is a graphical representation of an algorithm or process using standard symbols to show the sequence of operations and decision points. It describes the flow of control and data through a program or system. Flowcharts help developers understand program logic before implementation and identify errors in design. They are commonly used for representing algorithms, processes, and workflows. Standard symbols such as rectangles, diamonds, arrows, and ovals are used to represent

processing steps, decisions, flow direction, and start/end points. Flowcharts improve communication, documentation, debugging, and maintenance of software systems.

Common Flowchart Symbols

Symbol	Meaning
Oval	Start/End
Rectangle	Process
Diamond	Decision
Arrow	Flow direction
Parallelogram	Input/Output

Advantages

- Easy understanding.
- Helps debugging.
- Improves documentation.

Disadvantages

- Time-consuming for large systems.
- Difficult to modify.

---

8. Coupling and Cohesion Measures

--

Coupling and Cohesion are important software design measures used to evaluate the quality of modular design. Coupling refers to the degree of dependency between different modules, while cohesion refers to the degree to which elements within a single module are related. Good software design aims for low coupling and high cohesion because independent modules are easier to understand, test, modify, and reuse. High coupling increases complexity because changes in one module affect others. High cohesion improves module functionality by keeping related tasks together. These measures help developers create flexible, maintainable, and reliable software systems.

Coupling
Definition:
Coupling measures the dependency between modules.
Goal:
Low Coupling

Types of Coupling (Worst to Best)

1. Content Coupling
2. Common Coupling
3. External Coupling
4. Control Coupling
5. Stamp Coupling
6. Data Coupling

Cohesion
Definition:
Cohesion measures how closely related elements inside a module are.
Goal:
High Cohesion

Types of Cohesion (Worst to Best)

1. Coincidental Cohesion
2. Logical Cohesion
3. Temporal Cohesion
4. Procedural Cohesion
5. Communicational Cohesion
6. Sequential Cohesion
7. Functional Cohesion

---

9. Design Strategies

--

Software Design Strategies are approaches used by software engineers to transform requirements into an effective software design. These strategies define how system components are identified, organized, and developed. Different strategies are selected based on project size, complexity, and requirements. Common design strategies include function-oriented design, object-oriented design, top-down design, and bottom-up design. These approaches help manage complexity, improve software structure, support reuse, and increase maintainability. A suitable design strategy ensures efficient development and produces software that is reliable, flexible, and easy to modify. Proper selection of design strategy is important for successful software implementation.

---

10. Function-Oriented Design

--

Function-Oriented Design is a software design approach that focuses on identifying functions or processes required to perform system operations. The system is divided into a hierarchy of functions and sub-functions, where each function performs a specific task. This approach follows a process-centered view and commonly uses techniques such as data flow diagrams and structure charts. Function-oriented design emphasizes algorithms and data transformation rather than objects. It is suitable for systems with well-defined processes. However, it provides limited support for reuse and maintenance compared to object-oriented design because data and functions are usually separated.

Features

- Process-oriented.
- Uses DFD.
- Uses structure charts.
- Focuses on functions.

---

11. Object-Oriented Design (OOD)

--

Object-Oriented Design (OOD) is a software design approach that organizes software around objects and classes instead of functions. It identifies objects, their attributes, methods, and relationships to create a flexible software structure. OOD uses concepts such as encapsulation, inheritance, abstraction, and polymorphism. It improves software reusability, maintainability, and scalability. Object-oriented design models real-world entities and their interactions, making complex systems easier to understand and modify. UML diagrams such as class diagrams, sequence diagrams, and use case diagrams are commonly used during object-oriented design. OOD is widely used in modern software development.

Advantages of OOD

- Reusable components.
- Easy maintenance.
- Better scalability.
- Real-world modeling.

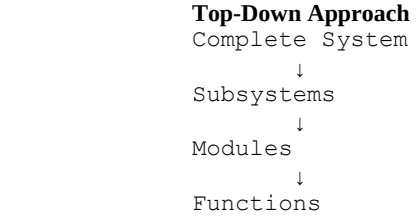
---

12. Top-Down Design

--

Top-Down Design is a design approach in which a complex system is divided into smaller modules by starting from the overall system level and gradually moving toward detailed components. It begins with the main system function and breaks it into smaller sub-functions until each module becomes manageable. This approach focuses on overall system structure before implementation details. Top-down design improves planning, reduces complexity, and provides clear system organization. It is commonly used in structured programming and function-oriented design. However, lower-level details may be ignored during the early stages, causing difficulties in some complex projects.

Top-Down Approach



---

13. Bottom-Up Design

--

Bottom-Up Design is a software design approach in which development begins by designing small reusable components and gradually combining them to create a complete system. It focuses on solving smaller problems first and integrating them into higher-level structures. This approach encourages component reuse, flexibility, and practical implementation. Bottom-up design

is commonly used in object-oriented programming where classes and objects are developed first and then combined into larger systems. It allows developers to test individual components early. However, it may lack a clear overall system view during the initial stages of development.

Bottom-Up Approach

```
Modules
   ↓
Subsystems
   ↓
Complete System
```

Difference Between Top-Down and Bottom-Up Design

Top-Down	Bottom-Up
Starts from system level	Starts from components
Focuses on decomposition	Focuses on integration
Used in structured design	Used in object-oriented design
Less reusable	Highly reusable

---

14. User Interface Design

--

User Interface Design is the process of designing the interaction between users and software systems. It focuses on creating interfaces that are easy to understand, efficient to use, and visually clear. A good user interface allows users to perform tasks easily while reducing errors and learning effort. UI design includes screen layout, navigation, input methods, colors, menus, buttons, and feedback mechanisms. The main goal of UI design is to improve user experience and satisfaction. Effective user interface design follows principles such as simplicity, consistency, user control, error prevention, and accessibility.

Principles of Good UI Design

1. Simplicity

Keep interface easy to understand.

---

2. Consistency

Maintain similar layouts and operations.

---

3. Feedback

Provide responses to user actions.

---

4. Error Prevention

Avoid user mistakes.

---

5. User Control

Allow users to control actions.

Types of User Interfaces

1. Command Line Interface (CLI)

Users enter commands.

Example:

Terminal.

---

2. Graphical User Interface (GUI)

Uses:

- Windows
- Icons
- Buttons

Example:

Desktop applications.

---

3. Web Interface

Browser-based interaction.

Quick Revision Table

Topic	Key Point
Software Design	Blueprint for software development
Architectural Design	Defines overall system structure
Low-Level Design	Defines module implementation details
Modularization	Divides system into independent modules
Structure Chart	Shows module hierarchy
Flow Chart	Represents process logic
Coupling	Dependency between modules
Cohesion	Relationship within module
Function-Oriented Design	Focuses on functions
Object-Oriented Design	Focuses on objects
UI Design	Improves user interaction

Coding and Structured Programming

---

1. Coding Phase

--

Coding Phase is the stage of software development in which the software design is converted into actual program code using a programming language. It is one of the most important phases of the Software Development Life Cycle (SDLC) because it transforms design specifications into an executable software product. During coding, programmers write, debug, and optimize source code according to coding standards and design guidelines. The main objectives of the coding phase are to produce efficient, readable, maintainable, and error-free programs. Proper coding practices improve software quality, reduce defects, simplify testing, and make future modifications easier.

Coding Phase in SDLC

```
Requirement Analysis
          ↓
Design
          ↓
Coding
          ↓
Testing
          ↓
Deployment
          ↓
Maintenance
```

Activities of Coding Phase

1. Code Writing

- Developers convert design documents into programs.
- Programming languages are selected according to project requirements.

Examples:

- Java
- Python
- C++
- C#

---

2. Code Review

- Code is examined by other developers.
- Errors and coding standard violations are identified.

---

3. Debugging

- Finding and removing programming errors.

Types of errors:

- Syntax errors.
- Logical errors.

- Runtime errors.

---

4. Unit Testing

- Individual modules are tested separately.
- Ensures each component works correctly.

---

2. Goals of Coding Phase

--

The **Goals of the Coding Phase** describe the objectives that developers should achieve while converting software design into executable programs. The primary goal is to produce correct, efficient, reliable, and maintainable source code that satisfies software requirements. Coding should follow proper programming standards, naming conventions, documentation practices, and error-handling techniques. Good coding reduces defects, improves readability, simplifies testing, and supports future maintenance. The coding phase also focuses on optimization, security, portability, and reusability of software components. Achieving these goals results in high-quality software that performs effectively and meets user expectations.

Major Goals of Coding Phase

---

1. Correctness

Meaning:

Code should perform exactly according to specified requirements.

Example:

A banking system should calculate balances correctly.

---

2. Readability

Meaning:

Code should be easy for developers to understand.

Achieved through:

- Proper naming.
- Comments.
- Indentation.

---

3. Maintainability

Meaning:

Code should be easy to modify and update.

---

4. Efficiency

Meaning:

Code should use resources effectively.

Resources:

- Memory.
- Processing time.

---

5. Reliability

Meaning:

Software should operate correctly under different conditions.

---

6. Reusability

Meaning:

Existing code components should be usable in other applications.

---

7. Security

Meaning:

Code should protect data and prevent unauthorized access.

---

3. Programming Style

--

Programming Style refers to the set of coding practices, conventions, and guidelines followed by programmers while writing software programs. It determines how code is organized, formatted, documented, and structured. A good programming style improves readability, understanding, debugging, and maintenance of software. It includes meaningful variable names, proper indentation, consistent formatting, comments, modular design, and error-handling practices. Programming style helps teams maintain uniform coding standards when multiple developers work on the same project. A well-designed programming style

reduces programming errors, improves communication among developers, and increases the overall quality and reliability of software systems.

Importance of Programming Style

- Improves code readability.
- Makes debugging easier.
- Reduces maintenance cost.
- Supports teamwork.
- Improves software quality.

Elements of Good Programming Style

---

1. Meaningful Names
Variables and functions should have descriptive names.
Example:
Good:
```
total_salary
```
Bad:
```
ts
```

---

2. Proper Indentation
Correct spacing improves readability.
Example:
```
if(condition)
{
   statement;
}
```

---

3. Comments and Documentation
Comments explain complex logic.
Example:
```
// Calculate employee salary
```

---

4. Modular Code
Programs should be divided into small functions/modules.

---

5. Consistent Formatting
Maintain similar coding patterns throughout the program.

---

6. Error Handling
Programs should handle unexpected situations properly.

Common Programming Style Rules

Rule	Purpose
Use meaningful names	Better understanding
Avoid unnecessary complexity	Easier maintenance
Write comments	Better documentation
Follow standards	Consistent development
Avoid duplicate code	Improve reuse

---

4. Structured Programming
--
Structured Programming is a programming methodology that emphasizes writing programs using clear logical structures and systematic control flow. It avoids the use of uncontrolled jumps such as the GOTO statement and uses three basic control structures: sequence, selection, and iteration. Structured programming divides a large program into smaller modules or functions, making the software easier to understand, test, and maintain. It follows a top-down approach where complex problems are gradually divided into simpler tasks. The main objective of structured programming is to improve program reliability, readability, and development efficiency by applying disciplined programming techniques and proper organization.

Basic Control Structures of Structured Programming

1. Sequence Structure
Definition:
Statements are executed one after another in a fixed order.
Example:

```
Step 1
  ↓
Step 2
  ↓
Step 3
```

2. Selection Structure
Definition:
Allows decision making based on conditions.
Examples:

- if statement
- if-else statement
- switch statement

Example:

```
Condition
   |
Yes/No
```

3. Iteration Structure
Definition:
Repeats statements until a condition is satisfied.
Examples:

- for loop
- while loop
- do-while loop

5. Objectives of Structured Programming
--
The **Objectives of Structured Programming** are to develop software programs that are easy to understand, verify, modify, and maintain. Structured programming aims to reduce program complexity by dividing large problems into smaller modules and using simple control structures. It improves program reliability by eliminating unnecessary jumps and encouraging logical program flow. The approach focuses on clarity, modularity, readability, and systematic development. It helps programmers identify errors easily and reduces maintenance efforts. Structured programming also promotes code reuse and standard programming practices. Its main objective is to create organized, efficient, and high-quality software that can be easily managed.

Main Objectives

1. Reduce Complexity
Large problems are divided into smaller manageable parts.

2. Improve Readability
Programs should be easy to understand.

3. Increase Reliability
Logical structure reduces programming errors.

4. Simplify Testing
Individual modules can be tested separately.

5. Improve Maintainability
Changes can be made easily.

6. Encourage Reusability
Modules can be reused in different programs.

6. Principles of Structured Programming

--

The **Principles of Structured Programming** are guidelines that help programmers create well-organized and maintainable software. These principles focus on logical program structure, modular design, and controlled execution flow. Structured programming emphasizes dividing programs into smaller modules, using standard control structures, avoiding unnecessary jumps, and maintaining clear documentation. The main principles include top-down design, modularity, use of sequence-selection-iteration structures, single entry and exit points, and proper data organization. Following these principles improves program readability, reduces errors, simplifies testing, and makes software easier to modify and maintain throughout its life cycle.

Major Principles

---

1. Top-Down Design
Meaning:
A complex problem is divided into smaller sub-problems.
Example:
System
↓
Modules
↓
Functions

---

2. Modularity
Meaning:
Programs are divided into independent modules.
Benefits:

- Easier testing.
- Easier maintenance.

---

3. Use of Basic Control Structures
Structured programming uses:

- Sequence.
- Selection.
- Iteration.

---

4. Avoid GOTO Statement
Meaning:
Uncontrolled jumps make programs difficult to understand.

---

5. Single Entry and Exit
Meaning:
Each module should have one starting point and one ending point.

---

6. Stepwise Refinement
Meaning:
Detailed solution is developed gradually from a general solution.

---

7. Proper Documentation
Meaning:
Programs should include comments and explanations.

---

7. Advantages of Structured Programming
--

The **Advantages of Structured Programming** include improved readability, easier debugging, better software reliability, and simplified maintenance. By dividing programs into smaller modules and using clear control structures, structured programming reduces complexity and improves program organization. It allows developers to understand, test, and modify individual parts of software without affecting the entire system. Structured programs are easier to document and maintain because their logic follows a systematic approach. This methodology improves teamwork because programmers can follow common standards. Structured programming also reduces development time and increases software quality by minimizing errors and encouraging disciplined programming practices.

Advantages

1. Easy Understanding
Programs have clear logical flow.

---

2. Easy Debugging
Errors can be identified quickly.

---

3. Better Maintenance
Changes can be made easily.

---

4. Improved Reliability
Logical structure reduces mistakes.

---

5. Code Reuse
Modules can be reused.

---

6. Better Testing
Individual modules can be tested independently.

---

7. Increased Productivity
Developers work more efficiently.

---

8. Disadvantages of Structured Programming
--
The **Disadvantages of Structured Programming** are limitations associated with the structured approach to software development. Although structured programming improves program organization, it may become difficult to manage for very large and complex systems. It focuses mainly on functions and procedures rather than real-world objects, making it less suitable for object-oriented applications. Reusing existing code can be difficult compared to object-oriented approaches. Large structured programs may require extensive planning and documentation. Changes in data structures may require modifications in multiple functions. Therefore, structured programming is less flexible for modern software systems that require high scalability and continuous evolution.

Disadvantages
1. Difficult for Large Systems
Large programs become complex.

---

2. Limited Reusability
Less support for component reuse.

---

3. Data Security Problems
Data and functions are often separate.

---

4. Less Flexible
Changes may affect many modules.

---

5. Not Suitable for Modern Applications
Complex systems often require object-oriented approaches.

Structured Programming vs Object-Oriented Programming

Structured Programming	Object-Oriented Programming
Function-based approach	Object-based approach
Focuses on procedures	Focuses on objects
Uses functions/modules	Uses classes and objects
Less reusable	Highly reusable
Suitable for small systems	Suitable for complex systems

Quick Revision Table

Topic	Key Points
Coding Phase	Converts design into program code
Coding Goals	Correctness, readability, efficiency, maintainability

Topic	Key Points
Programming Style	Rules for writing quality code
Structured Programming	Uses sequence, selection, and iteration
Objectives	Reduce complexity and improve quality
Principles	Modularity, top-down design, no GOTO
Advantages	Easy testing, maintenance, reliability
Disadvantages	Less flexible and reusable

Software Testing

---

1. Software Testing

--

Software Testing is the process of evaluating and verifying a software system to ensure that it meets specified requirements and works correctly. It involves executing software programs to identify defects, errors, missing functions, and unexpected behavior. The main objective of testing is to improve software quality, reliability, security, and performance before delivery to users. Testing helps developers find problems early and reduce the risk of software failure. It includes various techniques such as structural testing, functional testing, unit testing, integration testing, system testing, and acceptance testing. Effective software testing ensures that the final product satisfies customer expectations and quality standards.

Objectives of Software Testing

- Find software defects.
- Verify software requirements.
- Improve software reliability.
- Ensure quality standards.
- Reduce maintenance costs.
- Increase customer satisfaction.

---

2. Impracticality of Testing

--

Impracticality of Testing refers to the fact that complete and exhaustive testing of software is usually impossible because software systems contain a large number of possible inputs, execution paths, and operating conditions. Testing every possible combination requires unlimited time, cost, and resources. Therefore, testers use selective testing techniques to identify the most important defects within available constraints. The goal of testing is not to prove that software is completely error-free but to increase confidence that the software works correctly. Effective testing focuses on risk areas, critical functions, and high-impact defects to achieve maximum quality with limited resources.

Why Complete Testing is Impossible?

1. Large Input Space

Software can accept thousands or millions of possible inputs.

Example:

A banking application can have countless transaction combinations.

---

2. Multiple Execution Paths

Programs may have many possible paths.

Example:

A complex program may contain thousands of decision combinations.

---

3. Time Limitations

Testing every possibility requires excessive time.

---

4. Cost Limitations

Complete testing requires large resources and manpower.

---

5. Changing Requirements

Software requirements may change during development.

Approaches to Handle Testing Limitations

1. Risk-Based Testing

Focus testing on high-risk areas.

---

2. Test Case Prioritization
Execute important test cases first.

---

3. Automation Testing
Use tools to increase testing efficiency.

---

4. Regression Testing
Retest software after modifications.

---

3. Levels of Testing
--
Levels of Testing represent different stages at which software testing is performed during the development process. Each testing level focuses on specific components and objectives to ensure software quality. Testing begins with individual program units and gradually moves toward complete system evaluation. The major levels of testing include unit testing, integration testing, system testing, and acceptance testing. These levels help identify errors at different stages of development and ensure that software components work correctly individually as well as together. A systematic testing approach improves reliability, reduces defects, and ensures that the final software product satisfies user requirements.

Testing Levels Diagram

```
Unit Testing
      ↓
Integration Testing
      ↓
System Testing
      ↓
Acceptance Testing
```

---

1. Unit Testing
Definition
Unit testing is the testing of individual software components or modules independently.
Purpose:

- Verify each module works correctly.
- Identify programming errors.

Example:
Testing a login function separately.
Performed By:
Developers.

Advantages

- Finds errors early.
- Easy debugging.
- Reduces development cost.

---

2. Integration Testing
Definition
Integration testing combines individual modules and tests their interaction.
Purpose:

- Detect interface errors.
- Verify communication between modules.

Example:
Testing interaction between login and database modules.

Types of Integration Testing
1. Top-Down Integration
Testing starts from the highest-level module.

---

2. Bottom-Up Integration
Testing starts from lower-level modules.

---

3. Big Bang Integration
All modules are combined at once.

Advantages

- Finds communication errors.
- Checks module compatibility.

---

3. System Testing
Definition
System testing evaluates the complete integrated software system to verify whether it satisfies specified requirements.
Includes:

- Functional testing.
- Performance testing.
- Security testing.
- Usability testing.

Advantages

- Tests complete system behavior.
- Validates requirements.

---

4. Acceptance Testing
Definition
Acceptance testing is performed to determine whether the software is acceptable to customers or end users before final delivery.
Types:

1. User Acceptance Testing (UAT)
2. Alpha Testing
3. Beta Testing

Advantages

- Ensures customer satisfaction.
- Confirms business requirements.

---

4. Structural Testing (White Box Testing)
--
Structural Testing, also known as **White Box Testing**, is a software testing technique that examines the internal structure, logic, and implementation details of a program. In this approach, testers have knowledge of the source code and test the internal working of software components. White box testing focuses on control flow, data flow, paths, conditions, and loops within the program. It helps identify coding errors, logical mistakes, and unused sections of code. Structural testing is mainly performed by developers during unit testing. Common techniques include statement coverage, branch coverage, path testing, and condition testing.

Characteristics of White Box Testing

- Requires programming knowledge.
- Tests internal logic.
- Examines source code.
- Usually performed by developers.
- Focuses on code coverage.

White Box Testing Techniques

---

1. Statement Coverage
Definition:
Ensures every statement in the program is executed at least once.
Example:
If a program has 100 statements, testing should execute all 100 statements.

---

2. Branch Coverage
Definition:
Checks whether all decision branches are executed.
Example:

For:
`if(condition)`
Both:

- True branch
- False branch

are tested.

---

3. Path Testing
Definition:
Tests different execution paths in a program.

---

4. Condition Testing
Definition:
Tests individual conditions within decision statements.

---

5. Loop Testing
Definition:
Tests loops for:

- Zero iterations.
- One iteration.
- Multiple iterations.
- Maximum iterations.

Advantages of White Box Testing

- Finds logical errors.
- Improves code quality.
- Provides high code coverage.
- Detects hidden errors.
- Helps optimize code.

Disadvantages of White Box Testing

- Requires programming knowledge.
- Expensive for large systems.
- Cannot detect missing requirements.
- Time-consuming.

---

5. Functional Testing (Black Box Testing)
--
Functional Testing, also known as **Black Box Testing**, is a software testing technique that evaluates the functionality of software without examining its internal code structure. Testers focus on inputs, outputs, and expected behavior of the system according to requirements. The tester does not need programming knowledge because testing is performed from the user's perspective. Functional testing verifies whether the software performs required operations correctly. It is commonly used during system testing and acceptance testing. Techniques include equivalence partitioning, boundary value analysis, decision table testing, and cause-effect graphing. Black box testing ensures that software meets user expectations and functional requirements.

Characteristics of Black Box Testing

- No knowledge of source code required.
- Tests software functionality.
- Based on requirements.
- Performed by testers.
- Focuses on user behavior.

Black Box Testing Techniques

---

1. Equivalence Partitioning
Definition:
Divides input data into valid and invalid groups called equivalence classes.
Example:
Age field:

- Valid: 18–60
- Invalid: Below 18

---

2. Boundary Value Analysis
Definition:
Tests values at the boundaries of input ranges.
Example:
For range 1–100:
Test:

- 0
- 1
- 100
- 101

---

3. Decision Table Testing
Definition:
Tests combinations of conditions and actions.
Used for:

- Business rules.
- Complex decisions.

---

4. State Transition Testing
Definition:
Tests system behavior when moving between different states.
Example:
Login states:

- Logged out.
- Logged in.

---

5. Cause-Effect Graphing
Definition:
Shows relationships between input conditions and outputs.

Advantages of Black Box Testing

- No programming knowledge required.
- Tests user requirements.
- Suitable for large systems.
- Finds missing functionality.
- Independent testing.

Disadvantages of Black Box Testing

- Limited code coverage.
- Difficult to identify internal errors.
- Duplicate test cases possible.
- Cannot test hidden logic.

Difference Between White Box and Black Box Testing

White Box Testing	Black Box Testing
Tests internal code	Tests external behavior
Requires programming knowledge	No coding knowledge required
Done mainly by developers	Done mainly by testers
Focuses on logic and paths	Focuses on requirements
High code coverage	Functional coverage

Difference Between Structural and Functional Testing

Structural Testing	Functional Testing
Also called White Box Testing	Also called Black Box Testing

Structural Testing	Functional Testing
Examines internal structure	Examines external behavior
Code knowledge required	Code knowledge not required
Finds logical errors	Finds requirement failures
Uses coverage techniques	Uses input-output techniques

Testing Process

```
Test Planning
      ↓
Test Case Design
      ↓
Test Execution
      ↓
Defect Reporting
      ↓
Defect Fixing
      ↓
Regression Testing
      ↓
Test Completion
```

Quick Revision Table

Topic	Key Points
Software Testing	Process of finding defects and verifying quality
Testing Impracticality	Complete testing is impossible due to time and cost limits
Unit Testing	Tests individual modules
Integration Testing	Tests module interaction
System Testing	Tests complete system
Acceptance Testing	Validates customer requirements
White Box Testing	Tests internal code structure
Black Box Testing	Tests functionality without code knowledge

UNIT IV – Software Maintenance

---

1. Software Maintenance

--

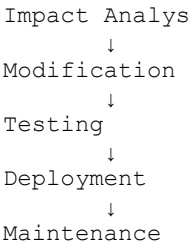
Software Maintenance is the process of modifying, updating, improving, and managing a software system after its delivery to users. It is performed to correct errors, adapt software to changing environments, improve performance, and add new features according to user requirements. Maintenance is an important phase of the Software Development Life Cycle (SDLC) because software continues to evolve after deployment. It includes activities such as bug fixing, performance enhancement, security updates, and system modifications. Effective software maintenance increases software reliability, extends the useful life of the system, reduces failures, and ensures that software continues to satisfy user and business needs.

Importance of Software Maintenance

- Fixes software defects.
- Improves software performance.
- Adapts software to new environments.
- Adds new features.
- Enhances security.
- Extends software life.
- Improves user satisfaction.

Software Maintenance Process

```
Change Request
      ↓
```



---

2. Need for Software Maintenance

--

The **Need for Software Maintenance** arises because software systems must continue to operate effectively after deployment in a changing environment. User requirements, business rules, operating systems, hardware platforms, and security threats may change over time. Maintenance allows developers to correct errors, improve software performance, add new functionality, and ensure compatibility with new technologies. Without proper maintenance, software becomes outdated, unreliable, and difficult to use. Software maintenance also helps organizations protect their investment by extending the operational life of software systems. Therefore, maintenance is necessary to maintain software quality, reliability, security, and user satisfaction throughout the software life cycle.

Reasons for Software Maintenance

---

1. Correct Errors

Software may contain hidden defects discovered after release.
Example:
Fixing calculation errors.

---

2. Adapt to Changes

Software must work with:

- New operating systems.
- New hardware.
- New business environments.

---

3. Improve Performance

Enhances:

- Speed.
- Efficiency.
- Resource usage.

---

4. Add New Features

New user requirements require software modifications.
Example:
Adding online payment options.

---

5. Improve Security

Updates protect against new security threats.

---

6. Maintain Compatibility

Ensures software works with other systems.

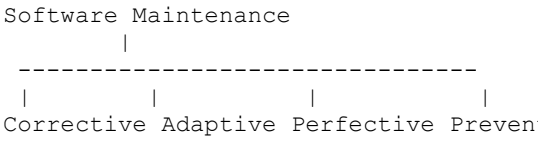
---

3. Categories of Software Maintenance

--

Categories of Software Maintenance classify maintenance activities according to the purpose and type of changes made to software. Software maintenance is generally divided into four major categories: corrective, adaptive, perfective, and preventive maintenance. Corrective maintenance focuses on fixing errors and defects found after software deployment. Adaptive maintenance modifies software to work in new environments. Perfective maintenance improves software performance and adds new features based on user needs. Preventive maintenance improves software structure and reduces future maintenance problems. These categories help organizations manage software changes effectively and maintain software quality throughout its operational life.

Types of Software Maintenance



---

4. Corrective Maintenance

--

Corrective Maintenance is the process of identifying and fixing errors, faults, and defects discovered after software deployment. It ensures that software performs according to its original requirements. Corrective maintenance is usually initiated when users report problems or when testing identifies failures in the operational system. Examples include fixing programming bugs, correcting incorrect calculations, and resolving system crashes. It is one of the most common types of maintenance because software defects may appear after real-world usage. Corrective maintenance improves software reliability, restores proper functionality, and prevents failures from affecting users.

Examples of Corrective Maintenance

- Fixing login errors.
- Correcting database problems.
- Removing software bugs.
- Repairing calculation mistakes.

Advantages

- Improves reliability.
- Removes defects.
- Increases user satisfaction.

---

5. Adaptive Maintenance

--

Adaptive Maintenance is the process of modifying software to operate correctly in a changed environment. It is performed when external conditions such as operating systems, hardware platforms, database systems, or business rules change. Adaptive maintenance ensures that existing software remains compatible with new technologies and organizational requirements. For example, updating an application to work with a new operating system or changing software according to new government regulations are adaptive maintenance activities. This type of maintenance helps organizations continue using existing software without completely replacing it and increases the software system's operational life.

Examples

- Updating software for a new operating system.
- Changing database compatibility.
- Supporting new hardware.

---

6. Perfective Maintenance

--

Perfective Maintenance refers to modifications made to improve software performance, usability, efficiency, and functionality after deployment. It focuses on enhancing existing features and adding new capabilities based on changing user needs. Unlike corrective maintenance, perfective maintenance is not performed because of errors but to improve the software experience. Examples include improving user interface design, increasing processing speed, adding new reports, and enhancing system performance. Perfective maintenance helps software remain competitive, useful, and aligned with business requirements. It improves user satisfaction and ensures that software continues to provide value over time.

Examples

- Adding new features.
- Improving interface design.
- Optimizing performance.
- Adding new reports.

---

7. Preventive Maintenance

--

Preventive Maintenance is the process of modifying software to prevent future problems and improve maintainability. It focuses on improving internal software quality rather than fixing current errors. Preventive maintenance activities include code restructuring, documentation improvement, removing unused code, and updating software architecture. The main purpose is to

reduce future maintenance costs and prevent possible failures. It improves software readability, reliability, and flexibility. Preventive maintenance is usually performed before problems occur and helps organizations manage software effectively throughout its life cycle. It increases system stability and makes future modifications easier.

Examples

- Code refactoring.
- Updating documentation.
- Improving software design.
- Removing unnecessary components.

Comparison of Maintenance Types

Type	Purpose	Example
Corrective	Fix errors	Bug removal
Adaptive	Adjust to environment changes	OS update
Perfective	Improve/add features	New functionality
Preventive	Avoid future problems	Code improvement

---

8. Cost of Software Maintenance

--

The **Cost of Software Maintenance** refers to the expenses required to modify, update, improve, and support software after its initial delivery. Maintenance cost is often a major part of the total software life cycle cost because software continues to evolve for many years. The cost depends on factors such as software complexity, system age, documentation quality, developer experience, and frequency of changes. Poor design and lack of documentation increase maintenance expenses. Effective software engineering practices, modular design, and proper documentation help reduce maintenance costs. Organizations must carefully manage maintenance activities to control expenses and maintain software quality.

Factors Affecting Maintenance Cost

---

1. Software Complexity

Complex systems require more effort to modify.

---

2. Documentation Quality

Poor documentation increases understanding time.

---

3. Programmer Experience

Experienced developers reduce maintenance effort.

---

4. Software Age

Older systems are harder to maintain.

---

5. Programming Language

Old technologies may increase cost.

---

6. Frequency of Changes

More changes increase maintenance cost.

Maintenance Cost Distribution

Approximate distribution:

Maintenance Type	Cost Contribution
Perfective	Highest
Adaptive	Medium
Corrective	Medium
Preventive	Lower

---

9. Software Re-Engineering

--

Software Re-Engineering is the process of analyzing, modifying, and improving an existing software system to increase its quality, maintainability, and usability without completely developing a new system. It involves transforming old software into a

modern and improved version while preserving its functionality. Re-engineering activities may include code restructuring, data conversion, documentation improvement, reverse engineering, and system modernization. It is used when legacy systems become difficult to maintain or operate with new technologies. Software re-engineering reduces development costs, extends software life, improves performance, and allows organizations to continue using valuable existing software systems.

Objectives of Software Re-Engineering

- Improve maintainability.
- Reduce maintenance cost.
- Modernize old systems.
- Improve documentation.
- Extend software life.
- Increase performance.

Software Re-Engineering Process

```
Existing Software
        ↓
Reverse Engineering
        ↓
Code/Data Improvement
        ↓
Forward Engineering
        ↓
Modern Software
```

Activities of Software Re-Engineering

1. Reverse Engineering

Understanding existing software.

---

2. Code Restructuring

Improving program structure.

---

3. Data Restructuring

Improving database organization.

---

4. Documentation Improvement

Updating software documents.

---

5. Forward Engineering

Developing improved software.

Advantages of Software Re-Engineering

- Lower cost than redevelopment.
- Extends software life.
- Improves quality.
- Reduces risks.
- Preserves existing knowledge.

---

10. Reverse Engineering

--

Reverse Engineering is the process of analyzing an existing software system to recover its design, architecture, requirements, and internal structure from source code or executable software. It is performed when original documentation is missing, outdated, or incomplete. Reverse engineering helps developers understand legacy systems before making modifications or performing re-engineering. It involves code analysis, database analysis, program understanding, and documentation generation. The main objective of reverse engineering is not to create new software but to understand and represent existing software in a better form. It supports maintenance, migration, modernization, and improvement of old software systems.

Reverse Engineering Activities

1. Code analysis.
2. Design recovery.
3. Database analysis.

4. Documentation generation.
5. System understanding.

Advantages

- Helps understand old systems.
- Recovers missing documentation.
- Supports maintenance.
- Helps migration.

Difference Between Re-Engineering and Reverse Engineering

Reverse Engineering	Re-Engineering
Understands existing software	Improves existing software
Focuses on analysis	Focuses on modification
Creates documentation	Creates improved system
First step of re-engineering	Complete improvement process

---

11. Software Documentation

--

Software Documentation is the collection of written information that describes software requirements, design, development, operation, and maintenance procedures. It provides important knowledge about the software system and helps developers, users, and maintainers understand how the software works. Documentation includes requirement documents, design documents, user manuals, installation guides, and maintenance records. Good documentation improves communication, reduces maintenance difficulties, supports training, and helps future modifications. It is an essential part of software engineering because software without proper documentation becomes difficult to understand and maintain. Effective documentation increases software reliability and ensures long-term usability.

Types of Software Documentation

---

1. Requirement Documentation

Describes:

- User requirements.
- System requirements.
- Functional requirements.

Example:
SRS document.

---

2. Design Documentation

Contains:

- Architecture.
- Design diagrams.
- Database design.

---

3. Technical Documentation

Used by developers.
Includes:

- Source code explanation.
- APIs.
- Algorithms.

---

4. User Documentation

Used by end users.
Includes:

- User manuals.
- Help guides.

---

5. Maintenance Documentation

Contains:

- Change history.

- Bug reports.
- Updates.

Importance of Documentation

- Improves understanding.
- Helps maintenance.
- Supports training.
- Reduces dependency on individuals.
- Provides future reference.

Quick Revision Table

Topic	Key Point
Software Maintenance	Modification after software delivery
Corrective	Fixes errors
Adaptive	Handles environment changes
Perfective	Adds improvements/features
Preventive	Prevents future problems
Maintenance Cost	Expense of modifying software
Re-Engineering	Improving existing software
Reverse Engineering	Understanding existing software
Documentation	Written software information

Software Quality and Software Quality Assurance
Detailed Exam Notes with 100-Word Definitions, Concepts, Quality Attributes, Standards & Important Questions

---

1. Software Quality

--

Software Quality refers to the degree to which a software product satisfies specified requirements, user expectations, and quality standards. It represents the ability of software to perform correctly, reliably, efficiently, securely, and maintainably throughout its life cycle. Software quality is not limited to error-free operation but also includes usability, performance, flexibility, and customer satisfaction. Quality software should meet functional requirements as well as non-functional requirements. Software quality is achieved through proper requirements analysis, good design, disciplined development practices, testing, and quality assurance activities. Measuring and improving software quality helps organizations deliver reliable and cost-effective software products.

Importance of Software Quality

- Increases customer satisfaction.
- Reduces software failures.
- Improves reliability.
- Reduces maintenance cost.
- Enhances security.
- Increases software life.
- Improves organization reputation.

---

2. Software Quality Attributes

--

Software Quality Attributes are the characteristics or properties that determine the quality of a software system. They describe how well software performs its functions and how easily it can be used, maintained, and modified. Quality attributes are mainly related to non-functional requirements and define the overall behavior and performance of software. Important software quality attributes include reliability, usability, efficiency, maintainability, portability, security, and functionality. These attributes help developers and managers evaluate software quality and ensure that the final product satisfies user expectations. A balance among different quality attributes is necessary to develop successful and dependable software systems.

Major Software Quality Attributes

---

1. Functionality
Definition:
Functionality refers to the ability of software to provide required services according to user requirements.
Includes:

- Correctness.
- Completeness.
- Interoperability.
- Security.

Example:
A banking application should correctly perform transactions.

---

2. Reliability
Definition:
Reliability is the ability of software to perform consistently without failure for a specified period.
Measures:

- Failure rate.
- Availability.
- Recovery capability.

Example:
An online payment system should work without unexpected failures.

---

3. Usability
Definition:
Usability refers to how easily users can learn, operate, and understand software.
Includes:

- User-friendly interface.
- Easy navigation.
- Simple operation.

---

4. Efficiency
Definition:
Efficiency measures how effectively software uses system resources.
Includes:

- Execution speed.
- Memory usage.
- Response time.

---

5. Maintainability
Definition:
Maintainability is the ability of software to be modified, corrected, or improved easily.
Includes:

- Modifiability.
- Testability.
- Analyzability.

---

6. Portability
Definition:
Portability is the ability of software to run on different platforms or environments.
Example:
Software working on Windows and Linux.

---

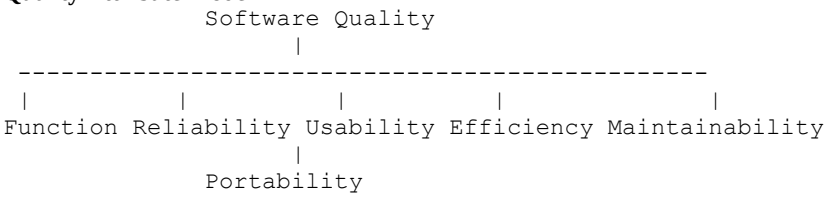
7. Security
Definition:
Security is the ability of software to protect information from unauthorized access.
Includes:

- Authentication.
- Authorization.
- Data protection.

---

8. Scalability
Definition:
Scalability is the ability of software to handle increasing users, data, or workload.

Quality Attribute Model



---

3. Software Quality Assurance (SQA)
--
Software Quality Assurance (SQA) is a systematic process used to ensure that software development activities and products meet specified quality standards. It focuses on preventing defects rather than only detecting them after development. SQA includes activities such as quality planning, reviews, audits, testing, process monitoring, and documentation control. The main objective of SQA is to improve software reliability, maintainability, and performance by following defined standards and procedures. SQA ensures that software processes are properly managed and that the final product satisfies customer requirements. It is an essential part of software engineering that helps deliver high-quality and dependable software systems.

Objectives of SQA

- Ensure software quality.
- Prevent defects.
- Follow development standards.
- Improve development process.
- Reduce development risks.
- Increase customer satisfaction.

Activities of Software Quality Assurance

---

1. Quality Planning
Definition:
Quality planning defines the quality goals, standards, methods, and procedures to be followed during software development.
Includes:

- Quality objectives.
- Quality standards.
- Review procedures.
- Testing plans.

---

2. Quality Reviews
Definition:
Reviews are systematic examinations of software documents and products to identify defects.
Types:

- Requirement reviews.
- Design reviews.
- Code reviews.

---

3. Software Audits
Definition:
Audits verify whether software processes follow defined standards and procedures.

---

4. Testing Activities
Testing ensures:

- Correctness.
- Reliability.
- Performance.

---

5. Configuration Management

Controls changes in:

- Software versions.
- Documents.
- Code.

---

6. Defect Management
Includes:

- Defect identification.
- Recording.
- Tracking.
- Removal.

---

7. Documentation Control
Ensures software documents are:

- Complete.
- Updated.
- Accurate.

---

4. Software Quality Assurance Plan (SQAP)
--
A **Software Quality Assurance Plan (SQAP)** is a document that describes the quality activities, responsibilities, standards, and procedures that will be followed during software development. It defines how quality will be measured, monitored, and maintained throughout the software life cycle. The SQAP identifies quality objectives, review methods, testing approaches, standards to be followed, and reporting mechanisms. It provides a systematic approach for achieving software quality and ensures that all team members understand their quality responsibilities. A properly prepared SQAP reduces defects, improves development processes, and helps deliver software that satisfies customer and organizational quality requirements.

Contents of SQAP
1. Purpose
Defines quality objectives.

---

2. Scope
Defines software areas covered.

---

3. Quality Standards
Specifies standards to follow.
Examples:

- ISO standards.
- CMM guidelines.

---

4. Reviews and Audits
Defines review procedures.

---

5. Testing Strategy
Defines testing methods.

---

6. Problem Reporting
Defines defect reporting methods.

---

7. Roles and Responsibilities
Defines team responsibilities.

---

5. Software Quality Standards
--
Software Quality Standards are established guidelines and models that define procedures, practices, and requirements for developing high-quality software. These standards help organizations improve software processes, maintain consistency, reduce defects, and ensure customer satisfaction. Quality standards provide frameworks for managing software development activities, documentation, testing, and process improvement. They allow organizations to measure their maturity level and compare their development practices with industry requirements. Important software quality standards include ISO 9001 and SEI Capability

Maturity Model (SEI-CMM). Following these standards helps organizations produce reliable, efficient, maintainable, and internationally acceptable software products.

---

6. ISO 9001

--

ISO 9001 is an international quality management standard that specifies requirements for establishing, implementing, maintaining, and improving a quality management system (QMS). It focuses on ensuring that organizations consistently provide products and services that satisfy customer requirements and regulatory standards. In software development, ISO 9001 provides guidelines for controlling processes, documentation, reviews, testing, and continuous improvement. It does not define how software should be developed but ensures that development processes are properly managed and controlled. ISO 9001 certification demonstrates an organization's commitment to quality, customer satisfaction, and continuous process improvement.

Objectives of ISO 9001

- Establish quality management systems.
- Improve customer satisfaction.
- Maintain consistent processes.
- Reduce errors.
- Encourage continuous improvement.

ISO 9001 Quality Principles

1. Customer Focus

Understanding and satisfying customer needs.

---

2. Leadership

Management provides direction and support.

---

3. Process Approach

Managing activities as processes.

---

4. Continuous Improvement

Regular improvement of processes.

---

5. Evidence-Based Decisions

Using data for decisions.

Benefits of ISO 9001

- International recognition.
- Improved quality control.
- Better customer confidence.
- Reduced defects.
- Improved documentation.

---

7. SEI-CMM (Software Engineering Institute Capability Maturity Model)

--

SEI-CMM (Software Engineering Institute Capability Maturity Model) is a process improvement framework developed by the Software Engineering Institute to evaluate and improve the maturity of software development processes in organizations. It provides guidelines for managing, measuring, and improving software processes. CMM classifies organizations into different maturity levels based on their process capability. The model helps organizations move from poorly managed processes to disciplined and optimized development practices. It focuses on areas such as project management, quality assurance, process control, and continuous improvement. SEI-CMM helps organizations produce higher-quality software with predictable cost, schedule, and performance.

Objectives of SEI-CMM

- Improve software processes.
- Increase productivity.
- Reduce project risks.
- Improve quality.
- Provide process maturity measurement.

SEI-CMM Maturity Levels

```
Level 5 - Optimizing
        ↑
Level 4 - Managed
        ↑
Level 3 - Defined
        ↑
Level 2 - Repeatable
        ↑
Level 1 - Initial
```

Level 1: Initial
Characteristics:

- Unorganized process.
- Success depends on individual effort.
- Poor management control.

Level 2: Repeatable
Characteristics:

- Basic project management practices exist.
- Previous successes can be repeated.

Key areas:

- Requirements management.
- Project planning.
- Configuration management.

Level 3: Defined
Characteristics:

- Standard software processes are documented.
- Organization-wide practices are followed.

Level 4: Managed
Characteristics:

- Processes are measured and controlled.
- Quantitative management is used.

Level 5: Optimizing
Characteristics:

- Continuous process improvement.
- Use of innovation and technology improvement.

Comparison Between ISO 9001 and SEI-CMM

ISO 9001	SEI-CMM
Quality management standard	Process maturity model
International standard	Software process improvement framework
Focuses on quality systems	Focuses on process maturity
Used for certification	Used for process assessment
Applicable to many industries	Mainly software organizations

Difference Between SQA and Software Testing

SQA	Testing
Prevents defects	Finds defects
Process-oriented	Product-oriented
Covers entire SDLC	Focuses on software execution
Includes standards and reviews	Includes test execution

Quick Revision Table

Topic	Key Points
Software Quality	Ability of software to satisfy requirements
Quality Attributes	Reliability, usability, efficiency, maintainability, portability
SQA	Ensures software quality through planned activities
SQAP	Document describing quality activities
ISO 9001	Quality management standard
SEI-CMM	Software process maturity model