Software Engineering

(LECTURE NOTES)

Prepared by:

Er. AKSHAY KUMAR PATRA

(Assistant Professor)

DEPT. OF COMPUTER SCIENCE. & ENGINEERING

Modern Engineering and Management Studies

Banaparia,Kuruda,Balasore,Odisha.

INTODUCTION TO SOFTWARE:-

Software is a set of programs used to operate computers and excute specific task.It is a generic term used to refer to applications,scripts and programs that run on a device.

Generally software are two types

1.System software **2.Application software**

System software-

System software are design to run a computers hardware and provide platform for application to run of computer system.

Example-Operating system,Drivers

Application software-

An application software that fullfils a specific need or performs task.

Example-MS-Office,Media player,Photo editor app.

CHARACTERSTICS OF SOFTWARE:

1.Functionality

2.Reliability

3.Efficiency

4.Usability

5.Maintainability

6.Portability

Module I: Software Process Models

1. Software Product

A software product is a set of programs and related documentation that performs a specific set of functions for a user.

Types:

- **Generic Products: Developed to be sold to a wide range of customers (e.g., MS Word, Photoshop).**
- **Customized Products: Developed for a specific client or organization (e.g., banking software for a particular bank).**

Characteristics:

- **Intangible**
- **Can be replicated at low cost**
- **Needs regular maintenance and updates**

2. Software Crisis:

The term "software crisis" was coined in the late 1960s to describe the growing difficulty of writing correct, reliable, and efficient software.

Major Issues:

- **Late delivery**
- **Exceeding budget**
- **Unreliable software**
- **Difficulty in maintenance**
- **Lack of skilled professionals**

Causes:

- **Increasing complexity of software systems**
- **Poor project management**
- **Inadequate testing**
- **Lack of structured development approaches**

Solution:

Adoption of structured software development models and engineering principles to manage complexity, improve quality, and meet timelines.

---

3. Handling Complexity through Abstraction and Decomposition

Abstraction

- **Simplifies complexity by focusing on essential details.**
- **Hides internal implementation.**
- **Examples: Abstract Data Types, APIs.**

Decomposition

- **Divides a system into smaller, manageable parts (modules or components).**
- **Each part handles a specific subtask.**
- **Encourages parallel development and easier maintenance.**

These are the foundation principles of software engineering that make large systems manageable.

---

4. Overview of Software Development Activities

Phases of Software Development Lifecycle (SDLC):

1. **Requirement Analysis**
    - **Understanding user needs and documenting requirements.**
2. **System Design**
    - **Architectural and detailed design.**
3. **Implementation (Coding)**
    - **Translating design into code.**
4. **Testing**
    - **Verifying correctness and identifying defects.**
5. **Deployment**
    - **Delivering software to users.**
6. **Maintenance**
    - **Bug fixing, enhancements, and updates.**

Software Process Models

A software process model is a standardized format for planning, organizing, and running a software development project.

---

1. Classical Waterfall Model

Phases:

1. **Feasibility Study**
2. **Requirement & Specification**
3. **System Design**
4. **Implementation/Design**
5. **Coading**
6. **Testing(Unit/Integration)**
7. **Maintenance**

Features:

- **Linear and sequential.**
- **Each phase must be completed before the next begins.**

Advantages:

- **Simple and easy to manage.**
- **Well-suited for small, well-defined projects.**

Disadvantages:

- **No room for changes once a phase is complete.**
- **Late detection of errors.**

2. Iterative Waterfall Model

Features:

- **Similar to waterfall, but allows feedback loops between phases.**
- **Errors can be corrected in earlier phases.**

Phases:

1. **Feasibility Study**
2. **Requirement & Specification**
3. **System Design**
4. **Implementation/Design**
5. **Coading**
6. **Testing(Unit/Integration)**
7. **Maintenance**

Advantages:

- **More flexible than classical waterfall.**
- **Allows rework and refinement.**

Disadvantages:

- **Still not ideal for dynamic or high-risk projects.**

3. Prototyping Model

Process:

1. **Gather basic requirements.**
2. **Build a prototype.**
3. **Get user feedback.**
4. **Refine the prototype.**
5. **Develop final system.**

Types:

- **Throwaway Prototype**
- **Evolutionary Prototype**

Advantages:

- **User feedback is incorporated early.**
- **Better understanding of requirements.**

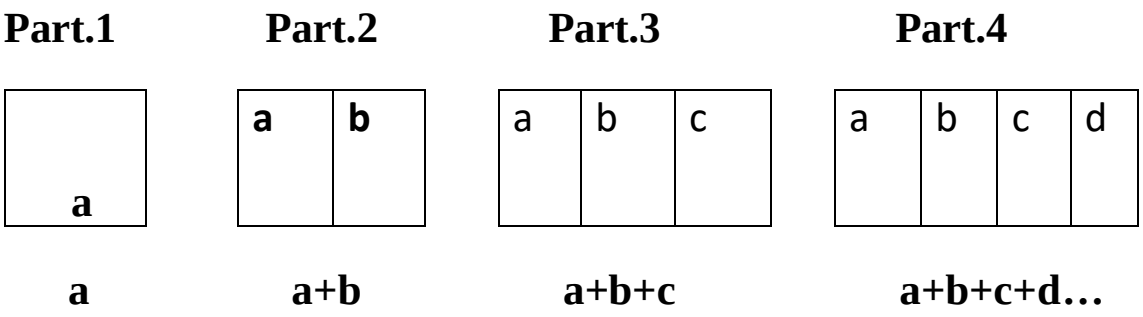
Disadvantages:

- **Misunderstanding that prototype is the final system.**
- **Poorly built prototypes can mislead users.**

4. Evolutionary Model

Features:

- **Builds the system in small, incremental versions.**
- **Requirements evolve over time.**



Advantages:

- **Adaptable to changing requirements.**
- **Quick delivery of working software.**

Disadvantages:

- **Design issues may emerge later.**
- **Hard to manage system architecture.**

5 Spiral Model (by Barry Boehm)

Phases in Each Spiral:

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

Features:

- **Combines iterative development with risk management.**
- **Ideal for large and complex projects.**

Advantages:

- **Focus on early risk identification.**
- **Frequent revisions improve quality.**

Disadvantages:

- **Complex and costly to implement.**
- **Requires expertise in risk assessment.**

6. RAD (Rapid Application Development) Model

Features:

- **Emphasizes quick development and delivery.**
- **Uses component-based construction.**

Phases:

1. **Requirements Planning**
2. **User Design**
3. **Construction**
4. **Cutover**

Advantages:

- **Short development cycles.**
- **High user involvement.**

Disadvantages:

- **Requires skilled developers.**
- **Not suitable for large, complex systems.**

6. Agile Models

Agile methodologies are iterative and incremental, focusing on customer collaboration and quick delivery.

6.1 Extreme Programming (XP)

Key Practices:

- **Pair Programming**
- **Test-Driven Development (TDD)**
- **Continuous Integration**
- **Refactoring**
- **Short release cycles**

Advantages:

- **High code quality**
- **Fast response to change**

Disadvantages:

- **Demands high customer involvement**
- **Not suitable for large teams**

6.2 Scrum

Roles:

- **Product Owner: Manages the product backlog.**
- **Scrum Master: Facilitates the Scrum process.**
- **Development Team: Cross-functional members who build the product.**

Events:

- **Sprint (2–4 weeks)**
- **Sprint Planning**
- **Daily Scrum**
- **Sprint Review**
- **Sprint Retrospective**

Artifacts:

- **Product Backlog**
- **Sprint Backlog**
- **Increment**

Advantages:

- **Quick feedback loop**
- **High adaptability**

Disadvantages:

- **Requires disciplined and self-organizing teams**
- **Scope creep if not well-managed**

Comparison of Software Process Models

Model	Flexibility	Risk Management	Delivery Speed	Ideal Use Case
Waterfall	Low	Low	Slow	Simple, well-defined projects
Iterative Waterfall	Medium	Low	Medium	Medium-size projects
Prototyping	High	Medium	Fast	Projects with unclear requirements
Evolutionary	High	Medium	Fast	Projects needing quick iterations
Spiral	High	High	Medium	High-risk, large projects
RAD	High	Low	Very Fast	Small-to-medium business apps
Agile (XP, Scrum)	Very High	Medium	Fast	Projects with changing needs

---

Conclusion

Understanding various software process models helps in selecting the right development approach based on project needs. modern methodologies like Agile provide flexibility and fast feedback.

Module II:

Software Requirements Engineering & Structured Design

Software Requirements Engineering

1. Requirement Gathering and Analysis

Requirement Gathering

- **The process of collecting the needs and expectations of stakeholders.**
- **Techniques:**
  - **Interviews**
  - **Surveys/Questionnaires**
  - **Brainstorming**
  - **Observation**
  - **Workshops**

Requirement Analysis

- **Evaluates and organizes gathered requirements.**
- **Goals:**
  - **Identify inconsistencies and ambiguities**
  - **Prioritize requirements**
  - **Classify as functional or non-functional**

---

2. Functional and Non-Functional Requirements

Functional Requirements

- **Describe what the system should do.**
- **Include user interactions, data manipulation, business rules.**
- **Examples:**
  - **Login functionality**
  - **Generating invoices**
  - **File upload system**

Non-Functional Requirements

- **Describe how the system should perform.**
- **Focus on quality attributes such as:**
  - **Performance**
  - **Scalability**
  - **Security**
  - **Usability**
  - **Reliability**
- **Examples:**
  - **Response time should be less than 2 seconds**
  - **System should support 1,000 concurrent users**

3. Software Requirement Specification (SRS)

Definition

- **A formal document that captures all software requirements in detail.**

Objectives:

- **Serve as a contract between client and developer**
- **Guide design, implementation, and testing**
- **Reduce misunderstandings**

Characteristics of a Good SRS:

- **Correct**
- **Unambiguous**
- **Complete**
- **Consistent**
- **Modifiable**
- **Verifiable**
- **Traceable**

---

4. IEEE 830 Guidelines for SRS

IEEE 830 is a standard that provides a template for writing SRS documents.

□ Structure of IEEE 830 SRS:

1. **Introduction**
    - **Purpose**
    - **Scope**
    - **Definitions**
    - **References**
2. **Overall Description**
    - **Product perspective**
    - **Product functions**
    - **User characteristics**
    - **Constraints**
    - **Assumptions and dependencies**
3. **Specific Requirements**
    - **Functional requirements**
    - **Non-functional requirements**
    - **External interfaces**

---

5. Decision Tables and Decision Trees

Decision Table

- **A tabular representation of conditions and actions.**
- **Useful for representing complex business logic.**

Example Table:

Conditions	Rule 1	Rule 2
Age > 18	Yes	No
Has ID	Yes	Yes
Action	Allow	Deny

Decision Tree

- **A graphical representation of decisions.**
- **Each internal node: a condition**
- **Each leaf node: an action**

Part B: Structured Analysis and Design

6. Overview of Design Process

Design Phase

Transforms requirements into a blueprint for software construction.

Steps:

1. **Architectural Design (high-level)**
2. **Detailed Design (module-level)**
3. **Interface Design**
4. **Data Design**

---

7. High-Level and Detailed Design

High-Level Design (HLD)

- **Also called architectural design.**
- **Focuses on:**
  - **Module decomposition**
  - **Interfaces between modules**

Detailed Design (LLD)

- **Focuses on:**
  - **Internal logic of modules**
  - **Algorithms**
  - **Data structures**

---

8. Cohesion and Coupling

Cohesion – Measure of how strongly related functions within a module are.

Types (Best to Worst):

- **Functional**
- **Sequential**
- **Communicational**
- **Procedural**
- **Temporal**
- **Logical**
- **Coincidental**

High cohesion is desirable.

Coupling – Measure of interdependence between modules.

Types (Worst to Best):

- **Content**
- **Common**
- **Control**
- **Stamp**
- **Data**
- **No Coupling**

Low coupling is desirable.

---

9. Modularity and Layering

Modularity

- **Divides the system into independent modules.**

- **Improves:**
  - **Maintainability**
  - **Reusability**
  - **Testability**

Layering

- **Organizes software into layers (e.g., UI, Business Logic, Data Access).**
- **Promotes separation of concerns.**

---

10. Function-Oriented Design

A. Structured Analysis using DFD (Data Flow Diagram)

DFD:

- **Represents data flow through a system.**

Symbols:

- **Process: Circle or rounded rectangle**
- **Data Store: Open-ended rectangle**
- **External Entity: Square**
- **Data Flow: Arrow**

DFD Levels:

- **Level 0: Context diagram**
- **Level 1: Top-level processes**
- **Level 2+: Detailed subprocesses**

B. Structured Design using Structure Chart

Structure Chart:

- **Represents hierarchical module structure.**
- **Shows:**
  - **Module calls**
  - **Data transfer**
  - **Control flow**

Elements:

- **Rectangles: Modules**
- **Arrows: Control/data flow**

Part C: Object-Oriented Analysis and Design (OOAD)

---

11. Basic Concepts of OOAD

Object-Oriented Analysis (OOA)

- **Identifies system objects and their interactions based on requirements.**

Object-Oriented Design (OOD)

- **Defines system architecture using objects and classes.**

OOAD Concepts:

- **Object: An entity with state and behavior**
- **Class: Blueprint for objects**
- **Encapsulation: Data hiding**
- **Abstraction: Focus on essential details**
- **Inheritance: Code reuse**

- **Polymorphism: Different implementations of the same interface**

Part D: User Interface (UI) Design

---

12. User Interface Design

Goals of UI Design:

- **Easy to learn**
- **Efficient to use**
- **Error-resistant**
- **Visually pleasing**

Principles:

- **Consistency**
- **Feedback**
- **Visibility**
- **User control**
- **Error prevention**

---

13. Types of Interfaces

A. Command Language Interface (CLI)

- **User types commands.**
- **Examples: Terminal, Command Prompt**
- **Pros: Powerful, precise**
- **Cons: Hard for beginners**

B. Menu-based Interface

- **Users select options from lists.**
- **Pros: Easy to navigate**
- **Cons: Limited flexibility**

C. Iconic (Graphical) Interface

- **Uses icons, buttons, visual elements.**
- **Examples: Windows, Android, iOS**
- **Pros: Intuitive and user-friendly**
- **Cons: Resource-intensive**

Summary Table

Topic	Key Points
Requirement Gathering	**Interviews, surveys, document analysis**
Functional Requirements	**Define what system should do**
Non-Functional Requirements	**Define how system performs**
SRS	**Formal document capturing all requirements**
IEEE 830	**Standard format for SRS**
Decision Table/Tree	**Represent complex decision logic**
High/Low-Level Design	**Architecture vs module logic**
Cohesion and Coupling	**Good design = high cohesion + low**

Topic	Key Points
	coupling
DFD	Data-centric modeling technique
Structure Chart	Function-oriented module hierarchy
OOAD	Object-focused system modeling
UI Design	Design user-friendly and accessible interfaces

Conclusion

This module emphasizes clear requirement specification, strong system architecture, and effective user interface design. Mastering these principles is essential for creating maintainable, user-friendly, and robust software systems.

Module III: Coding and Software Testing Techniques

1. Coding

Definition:
Coding is the process of translating design into a machine-readable format using a programming language.

Key Aspects:

- **Readability: Code should be clean, readable, and well-commented.**
- **Maintainability: Use consistent naming conventions, indentation, and modular programming.**
- **Reusability: Code should be reusable across different modules or projects.**
- **Error Handling: Include proper exception and error-handling mechanisms.**

---

2. Code Review

Definition:
A systematic examination of source code by developers other than the author.

Types of Code Reviews:

- **Formal Reviews: Includes inspection meetings and detailed checklists.**
- **Informal Reviews: Peer reviews, over-the-shoulder reviews.**
- **Automated Reviews: Using tools like SonarQube, CodeClimate, etc.**

Benefits:

- **Improves code quality.**
- **Identifies bugs early.**
- **Promotes knowledge sharing among developers.**

---

3. Documentation

Definition:
Creating supporting documents for the codebase to help future maintenance and scalability.

Types:

- **Internal Documentation: Code comments, inline explanations.**
- **External Documentation: User manuals, system design, API docs.**

---

4. Software Testing

Purpose:
To ensure the software product is reliable, meets requirements, and is bug-free.

4.1 Unit Testing

Definition:
Testing individual units/components of software.

- **Performed by developers.**
- **Tools: JUnit (Java), NUnit (.NET), PyTest (Python).**
- **Focus: Functions, methods, classes.**

4.2 Black-box Testing

Definition:
Testing without knowledge of internal code structure.

- **Based on input-output behavior.**
- **Techniques: Equivalence partitioning, boundary value analysis.**
- **Suitable for: System and acceptance testing.**

4.3 White-box Testing

Definition:
Testing with knowledge of the internal workings of the application.

- **Focus: Path coverage, branch coverage.**
- **Techniques: Statement testing, condition testing.**
- **Suitable for: Unit testing.**

4.4 Cyclomatic Complexity

Definition:
A metric to measure the complexity of a program by counting independent paths.

- **Formula: $M = E - N + 2P$**
  (E = edges, N = nodes, P = connected components)
- **Lower complexity = better maintainability.**
- **Helps in determining the minimum number of test cases.**

4.5 Coverage Analysis

Definition:
Determines which parts of the code were executed (covered) during testing.

Types:

- **Statement Coverage**
- **Branch Coverage**
- **Path Coverage**
- **Function Coverage**

4.6 Mutation Testing

Definition:
Involves modifying program code slightly (creating mutants) to test the effectiveness of test cases.

- **If test cases fail for mutant versions → strong test cases.**
- **Helps ensure robustness of testing suite.**

4.7 Debugging Techniques

Definition:
The process of finding and fixing bugs in the code.

Methods:

- **Print debugging: Using print/log statements.**
- **Breakpoint debugging: Using IDE breakpoints.**
- **Backtracking: Tracing the code from output to source.**
- **Automated debugging tools: GDB, Visual Studio Debugger.**

4.8 Integration Testing

Definition:
Testing combined parts/modules of an application to verify they work together.

- **Top-down: Test from main module downwards.**
- **Bottom-up: Start with lower-level modules.**
- **Big Bang: All components integrated and tested at once.**

4.9 System Testing

Definition:
Testing the complete, integrated system to verify it meets the specified requirements.

- **Conducted by independent testers.**
- **Includes: Functional, non-functional, performance, security testing.**

4.10 Regression Testing

Definition:
Testing the software after changes (bug fixes, updates) to ensure existing functionality still works.

- **Ensures that new changes haven’t introduced new bugs.**
- **Can be automated for efficiency.**

---

5. Software Reliability

Definition:
The probability that software will work without failure under given conditions for a specified period.

Key Concepts:

- **MTTF (Mean Time to Failure)**
- **MTTR (Mean Time to Repair)**
- **MTBF (Mean Time Between Failures)**
- **Fault Tolerance: Ability to continue operation despite faults.**
- **Redundancy: Using backup systems to increase reliability.**

Summary Table

Concept	Key Idea	Tools/Examples
Coding	Writing functional code	Any IDE/language
Code Review	Peer review of code	GitHub, Bitbucket
Documentation	Internal & external descriptions	Markdown, Doxygen
Unit Testing	Test individual functions	JUnit, PyTest
Black-box	Test with no internal view	Functional Testing
White-box	Test with internal knowledge	Statement/Branch Testing
Cyclomatic	Measure of code	Manual or tools

Concept	Key Idea	Tools/Examples
Complexity	**complexity**	
Coverage Analysis	**Measure of code tested**	**Coverage.py, Istanbul**
Mutation Testing	**Test quality of test cases**	**PIT, MutPy**
Debugging	**Finding and fixing bugs**	**GDB, Print logs**
Integration Testing	**Testing module interaction**	**Top-down, Bottom-up**
System Testing	**Whole system validation**	**End-user testing**
Regression Testing	**Re-test after changes**	**Selenium, JUnit**
Software Reliability	**Probability of failure-free operation**	**MTTF, MTBF metrics**

Module IV: Maintenance – Detailed

1. Basic Concepts in Software Reliability

Definition:

Software reliability refers to the probability of a software system operating without failure under given conditions for a specified period of time.

Importance:

- **Critical in safety systems (e.g., aviation, medical).**
- **Enhances user trust and system longevity.**

Key Factors:

- **Code quality.**
- **Testing completeness.**
- **Fault tolerance.**
- **Development process maturity.**

---

2. Reliability Measures

Common Metrics:

Measure	Description
MTTF (Mean Time To Failure)	**Average time the software operates before failure.**
MTTR (Mean Time To Repair)	**Average time taken to repair a system after failure.**
MTBF (Mean Time	**MTTF + MTTR; often used to assess**

Measure	Description
Between Failures)	system availability.
Failure Rate	Number of failures per unit of time.

---

3. Reliability Growth Modeling

Purpose:

To estimate and improve software reliability over time based on observed failures during testing.

Common Models:

- **Jelinski-Moranda Model: Assumes fixed number of initial faults.**
- **Goel-Okumoto Model: Uses exponential distribution to model fault detection.**
- **Musa Model: Considers execution time rather than calendar time.**

---

4. Quality and SEI Capability Maturity Model (CMM)

SEI CMM Overview:

Developed by the Software Engineering Institute (SEI) to improve software process maturity.

CMM Levels:

Level	Description

Level	Description
1 – Initial	Ad hoc and chaotic process.
2 – Repeatable	Basic project management processes established.
3 – Defined	Standardized and documented processes.
4 – Managed	Quantitative quality goals established.
5 – Optimizing	Continuous process improvement in place.

---

5. Characteristics of Software Maintenance

Definition:

The process of modifying a software system after delivery to correct faults, improve performance, or adapt to a changed environment.

Types of Maintenance:

- **Corrective: Fixing bugs.**
- **Adaptive: Updating software for a new environment (e.g., OS update).**
- **Perfective: Enhancing features or performance.**
- **Preventive: Improving future maintainability.**

Challenges:

- **Poor documentation.**
- **Code complexity.**
- **Developer turnover.**

---

6. Software Reverse Engineering

Definition:

Analyzing software to identify components and their interrelationships to extract design and specifications.

Applications:

- **Understanding legacy systems.**
- **Security analysis.**
- **Migration and documentation.**

Steps:

1. **Code analysis.**
2. **Abstraction extraction.**
3. **Design recovery.**

---

7. Software Reengineering

Definition:

Rebuilding existing software to improve maintainability or performance without changing its functionality.

Process:

1. **Reverse engineering.**
2. **Analysis.**
3. **Redesign.**
4. **Forward engineering.**

Benefits:

- **Cost-effective.**

- **Reduces complexity.**
- **Improves quality.**

---

8. Software Reuse

Definition:

Using existing software artifacts (code, designs, documentation) in new software systems.

Levels:

- **Code-level reuse: Libraries, functions.**
- **Design-level reuse: Architecture, patterns.**
- **Component-level reuse: Services, modules.**

Advantages:

- **Faster development.**
- **Lower costs.**
- **Higher quality through tested components.**

Emerging Topics in Software Engineering

---

9. Client-Server Software Engineering

Definition:

A software architecture model where clients request services and servers provide them.

Features:

- **Distributed computing.**
- **Centralized server logic.**
- **Examples: Web applications, email systems.**

Advantages:

- **Easier maintenance.**
- **Better resource sharing.**

---

10. Service-Oriented Architecture (SOA)

Definition:

A design pattern where software components provide services to other components via well-defined interfaces.

Key Concepts:

- **Loose coupling: Services are independent.**
- **Reusability: Services can be reused across systems.**
- **Interoperability: Based on open standards (SOAP, REST).**

Benefits:

- **Scalability.**
- **Flexibility.**
- **Easy integration.**

---

11. Software as a Service (SaaS)

Definition:

A cloud-based delivery model where software is hosted centrally and accessed over the Internet.

Examples:

- **Google Workspace.**
- **Microsoft 365.**
- **Salesforce.**

Characteristics:

- **Subscription-based.**
- **No need for local installation.**
- **Automatic updates.**

Advantages:

- **Reduced IT costs.**
- **Accessibility.**
- **Scalability.**

□ Summary Table

Topic	Key Focus
Software Reliability	Failure-free operation over time
Reliability Measures	MTTF, MTTR, MTBF
Growth Models	Predicting reliability improvement
SEI CMM	Process maturity model
Maintenance	Corrective, Adaptive, Perfective, Preventive
Reverse Engineering	Understanding legacy systems
Reengineering	Enhancing existing systems

Topic	Key Focus
Reuse	**Using existing components**
Client-Server	**Distributed model**
SOA	**Modular service interaction**
SaaS	**Web-based software delivery**

Thank you...