Absolutely. For **Software Engineering (SE), CS24353**, I'll follow the **same system as the previous subjects**: complete M1–M5 coverage, no syllabus topic skipped, exam-oriented explanations, memory hooks, active recall, comparisons, diagrams, and PYQ-oriented questions.

I checked your actual BIT Mesra course document. SE is a **3-credit course**, and the official syllabus is divided into five modules.

🧠 SOFTWARE ENGINEERING — COMPLETE MASTER NOTES

CS24353 | M1–M5 | 67 Topics

The study guide you uploaded expands the official syllabus into **67 individual study topics**.

MODULE I — INTRODUCTION TO SOFTWARE ENGINEERING

13 Topics

1. Introduction to Software Engineering
2. Definitions & FAQs
3. Evolving Role of Software
4. Software Process Models
5. Waterfall Model
6. Prototyping Model
7. Spiral Model
8. RAD Model
9. Incremental Model
10. Agile Models
11. Management Activities
12. Project Planning
13. Project Scheduling
14. Risk Management

Note: The study guide groups process models under one heading but expands them into individual models; therefore the checklist counts this module as **13 study areas**.

---

1. What is Software Engineering?

Software Engineering (SE) is the systematic, disciplined and measurable approach to the development, operation and maintenance of software.

Think:

```
Programming
   ↓
Writing code

Software Engineering
   ↓
Requirements
   ↓
Planning
   ↓
Design
   ↓
Implementation
   ↓
Testing
   ↓
Deployment
   ↓
Maintenance
```

🧠 **Memory**

> **Key Insight**
>
> **Software Engineering = Engineering principles applied to software.**

The official course objective specifically emphasizes software process models, project management, requirements, design, quality, estimation, maintenance and evolution.

---

2. Why Software Engineering?

Large software systems create problems involving:

- complexity
- changing requirements
- cost
- deadlines
- reliability
- maintainability
- security
- team coordination

Without engineering discipline:

```
Poor requirements
       ↓
Poor design
       ↓
More bugs
       ↓
More rework
       ↓
Higher cost
       ↓
Project failure
```

3. Evolving Role of Software

Software has evolved from simple programs into systems that control:

- communication
- banking
- transportation
- healthcare
- manufacturing
- entertainment
- cloud systems
- embedded systems
- AI systems

Important idea

Modern software is:

> **Key Insight**
>
> **a product + a service + an evolving system**

Software doesn't simply "finish."

It continuously changes.

---

4. Software Process Model

A **software process model** defines how software development activities are organized.

Common activities:

```
Requirements
     ↓
Design
     ↓
Implementation
     ↓
Testing
     ↓
Deployment
     ↓
Maintenance
```

Different models arrange these activities differently.

---

5. Waterfall Model

Sequential development model.

```
Requirements
     ↓
Design
     ↓
Implementation
     ↓
Testing
     ↓
Deployment
     ↓
Maintenance
```

Advantages

- simple
- structured
- easy to manage
- documentation-heavy

Disadvantages

- difficult to accommodate changes
- testing occurs relatively late
- poor fit for uncertain requirements

Best suited

Stable, well-understood requirements.

---

6. Prototyping Model

Build an early prototype to understand requirements.

```
Initial Requirements
        ↓
     Prototype
        ↓
    User Feedback
        ↓
Refined Requirements
        ↓
Final System
```

Useful when:

> **Key Insight**
> customer doesn't know exactly what they want.

---

7. Spiral Model

Combines iterative development with **risk analysis**.

Each cycle generally involves:

```
Objectives
   ↓
Risk Analysis
   ↓
Development
   ↓
Evaluation
   ↓
Next Spiral
```

🧠 Memory

> **Key Insight**
>
> **Spiral = Iteration + Risk**

Best for large, complex and high-risk projects.

---

8. RAD

Rapid Application Development

Focus:

> **Key Insight**
>
> fast development + reusable components + user feedback.

Typical characteristics:

- rapid prototyping
- component reuse
- iterative development
- strong user involvement

---

9. Incremental Model

Build the system in multiple increments.

```
Increment 1 → Core features
Increment 2 → More features
Increment 3 → More features
Increment 4 → Complete product
```

Difference from waterfall

Waterfall:

> **Key Insight**
> one major sequential development path.

Incremental:

> **Key Insight**
> working versions delivered progressively.

---

10. Agile Models

Agile emphasizes:

- iterative development
- customer collaboration
- responding to change
- frequent delivery
- working software
- continuous feedback

Agile cycle

```
Plan
 ↓
Develop
 ↓
Test
 ↓
Review
 ↓
Feedback
 ↺
```

🧠 Agile memory

> **Key Insight**
> **Build → Show → Learn → Adapt**

Process Model Comparison

Model	Main idea	Best when
Waterfall	Sequential	Stable requirements
Prototype	Discover requirements	Requirements unclear
Spiral	Risk-driven iteration	High-risk projects
RAD	Rapid development	Fast delivery
Incremental	Feature increments	Early partial delivery
Agile	Adaptive iterations	Changing requirements

🔥 **Exam question**

Compare Waterfall, Spiral, Prototyping, Incremental and Agile models.

This is a perfect long-answer question.

---

11. Management Activities

Major software management activities:

- project planning
- scheduling
- risk management
- resource management
- monitoring
- configuration management
- quality management

Think:

```
People
+
Process
+
Time
+
Cost
+
Risk
+
Quality
```

---

12. Project Planning

Project planning determines:

- what must be built
- how it will be built
- who will build it

- how long it will take
- how much it will cost
- what risks exist

Outputs may include:

- project scope
- resource plan
- cost estimates
- schedule
- risk plan

---

13. Project Scheduling

Scheduling determines **when activities occur**.

Important concepts:

- tasks
- dependencies
- duration
- milestones
- deadlines
- resources

Tools:

- Gantt charts
- PERT
- network diagrams
- critical path analysis

---

14. Risk Management

Risk = uncertain event that may negatively affect a project.

Examples:

- requirements changing
- key developer leaving
- technology failure
- budget shortage
- schedule delay
- security issue

Risk management cycle

```
Identify
   ↓
Analyze
   ↓
Prioritize
   ↓
Plan response
   ↓
Monitor
```

🧠 Memory

> **Key Insight**
>
> **I-A-P-M-M**

>

> **Key Insight**
>
> Identify → Analyze → Prioritize → Mitigate → Monitor

MODULE II — SOFTWARE REQUIREMENTS ENGINEERING

12 Topics

1. Functional requirements
2. Non-functional requirements
3. User requirements
4. System requirements
5. SRS
6. IEEE SRS standard
7. Quality of good SRS
8. Feasibility study
9. Requirements elicitation
10. Requirements analysis
11. Requirements validation
12. Requirements management

---

15. Functional Requirements

Describe **what the system should do**.

Example:

> **Key Insight**
> The system shall allow users to reset their password.

Examples:

- login
- registration
- payment
- search
- report generation

---

16. Non-Functional Requirements

Describe **how well** the system should work or constraints on it.

Examples:

- performance
- security
- reliability
- usability
- scalability
- availability

Example:

> **Key Insight**
> The system shall respond within 2 seconds.

Memory

> **Key Insight**
> **Functional = WHAT**

>

> **Key Insight**
> **Non-functional = HOW WELL**

---

17. User Requirements

High-level requirements written for customers/users.

Example:

> **Key Insight**
> Users should be able to book railway tickets online.

Generally understandable without technical implementation details.

---

18. System Requirements

Detailed technical requirements specifying what the system must provide.

Example:

> **Key Insight**
> The system shall authenticate a user using OTP within 30 seconds.

---

19. SRS

Software Requirements Specification

A formal document describing software requirements.

It acts as a bridge:

```
Customer needs
      ↓
     SRS
      ↓
Developers
      ↓
System
```

Typical SRS contents

- introduction
- scope
- definitions
- overall description
- functional requirements
- non-functional requirements
- interfaces
- constraints
- assumptions

---

20. IEEE Standard for SRS

Your official syllabus explicitly includes the **IEEE standard for SRS**.

Know:

- purpose
- organization
- characteristics
- requirements description
- interfaces
- constraints
- verification criteria

---

21. Characteristics of Good SRS

A good SRS should be:

- correct
- unambiguous
- complete
- consistent
- verifiable
- feasible
- traceable
- prioritized
- modifiable

🧠 Memory

> **Key Insight**
>
> **C-U-C-C-V-F-T-P-M**

Don't just memorize the letters—understand what each means.

---

22. Feasibility Study

Determines whether a proposed project is practical.

Common dimensions:

Technical

Can we build it?

Economic

Is it financially worthwhile?

Operational

Will it work in the organization?

Legal

Does it comply with laws/regulations?

Schedule

Can we finish on time?

🧠 Memory

> **Key Insight**
> **TELOS**

>

> **Key Insight**
> Technical

> **Key Insight**
> Economic

> **Key Insight**
> Legal

> **Key Insight**
> Operational

> **Key Insight**
> Schedule

---

23. Requirements Elicitation

Collect requirements from stakeholders.

Techniques:

- interviews
- questionnaires
- observation
- workshops
- brainstorming
- document analysis
- prototyping

---

24. Requirements Analysis

Analyze collected requirements to identify:

- conflicts
- ambiguity
- missing requirements
- dependencies
- feasibility issues
- priorities

---

25. Requirements Validation

Check whether requirements are suitable.

Validation checks:

- validity
- consistency
- completeness
- realism
- verifiability

---

26. Requirements Management

Requirements change.

Therefore:

```
Requirement
    ↓
Track
    ↓
Change request
    ↓
Impact analysis
    ↓
Approval
    ↓
Update
    ↓
Traceability
```

MODULE III — SOFTWARE DESIGN ENGINEERING

13 Topics

1. Design engineering
2. Design process
3. Design quality

4. Design concepts
5. Design models
6. Object-oriented design
7. Cohesion
8. Coupling
9. UML use-case diagram
10. UML class diagram
11. UML activity diagram
12. UML sequence diagram
13. UML collaboration diagram

---

27. Design Engineering

Transforms requirements into an architecture and detailed design.

```
SRS
 ↓
Architecture
 ↓
Components
 ↓
Interfaces
 ↓
Detailed Design
 ↓
Code
```

---

28. Design Process

Typical steps:

1. understand requirements
2. identify architecture
3. identify components
4. design interfaces
5. design data
6. refine components
7. review design

---

29. Design Quality

Good software design should emphasize:

- correctness
- simplicity

- modularity
- maintainability
- efficiency
- reliability
- reusability
- testability

---

30. Design Concepts

Core concepts:

- abstraction
- modularity
- information hiding
- separation of concerns
- refinement
- architecture
- cohesion
- coupling

---

31. Design Models

Common design views:

```
Data/Class Model
       +
Architectural Model
       +
Interface Model
       +
Component Model
       +
Deployment Model
```

---

32. Object-Oriented Design

Models software using objects/classes.

Core concepts:

- class
- object
- encapsulation
- inheritance
- polymorphism
- abstraction

- association

Example:

```
Class: Student

Attributes:
name
rollNo

Methods:
register()
attendClass()
```

---

33. Cohesion

Measures how strongly the elements of a module belong together.

High cohesion = GOOD

Example:

```
PaymentModule
  ├── validatePayment
  ├── processPayment
  └── generateReceipt
```

All related.

🧠 Memory

> **Key Insight**
>
> **Cohesion = inside**

---

34. Coupling

Measures dependency between modules.

Low coupling = GOOD

```
Module A → interface → Module B
```

rather than tightly depending on internal details.

🧠 Perfect exam memory

> **Key Insight**
>
> **High Cohesion + Low Coupling = Good Design**

UML

Unified Modeling Language.

Used to visually represent software systems.

35. Use Case Diagram

Shows:

- actors
- system
- use cases
- relationships

Example:

```
Customer
   |
   ├────→ Login
   |
   ├────→ Place Order
   |
   ├────→ Track Order
```

36. Class Diagram

Shows:

- classes
- attributes
- operations
- relationships

Example:

```
+──────────────────+
| Student          |
+──────────────────+
| name             |
| rollNo           |
+──────────────────+
| register()       |
| attendClass()    |
+──────────────────+
```

37. Activity Diagram

Represents workflow.



38. Sequence Diagram

Shows interactions **over time**.



Key:

> **Key Insight**
> vertical direction = time.

39. Collaboration Diagram

Focuses on interactions between objects and their relationships.

Unlike sequence diagrams, emphasis is more on **object organization and messages** than a vertical timeline.

MODULE IV — VERIFICATION, VALIDATION & TESTING

16 Topics

1. Verification
2. Validation
3. V&V planning
4. Software inspection
5. Static analysis
6. Software testing
7. Testing functions
8. Test-case design

9. White-box testing
10. Black-box testing
11. Basis-path testing
12. Control-structure testing
13. Unit testing
14. Integration testing
15. System testing
16. Reliability

---

40. Verification

Question:

> **Key Insight**
> **Are we building the product right?**

Checks whether artifacts conform to specifications.

Examples:

- reviews
- inspections
- static analysis

---

41. Validation

Question:

> **Key Insight**
> **Are we building the right product?**

Checks whether software satisfies actual user needs.

🧠 V&V memory

> **Key Insight**
> **Verification → specification**

>

> **Key Insight**
> **Validation → user**

---

42. Verification & Validation Planning

Defines:

- activities
- responsibilities
- schedules
- standards
- review strategy
- testing strategy
- acceptance criteria

---

43. Software Inspection

Formal review of software artifacts to find defects.

Can inspect:

- requirements
- design
- code
- documentation

Advantages:

> **Key Insight**
> defects can be found before execution.

---

44. Static Analysis

Analyze software **without executing it**.

Examples:

- syntax checking
- control-flow analysis
- data-flow analysis
- security analysis
- code quality analysis

---

45. Software Testing

Dynamic execution of software to discover defects.

Basic flow:

```
Input
 ↓
Execute
 ↓
Observe output
 ↓
Compare expected vs actual
 ↓
Pass / Fail
```

---

46. Testing Functions

Testing should verify whether functions satisfy requirements.

Example:

For:

```
login(username,password)
```

test:

- valid username/password
- invalid username
- invalid password
- empty values
- boundary cases
- locked account

---

47. Test Case Design

A test case generally contains:

- test ID
- input
- preconditions
- steps
- expected output
- actual output
- status

---

48. White-Box Testing

Tester knows internal implementation.

Focus:

- statements

- branches
- conditions
- paths
- loops

---

49. Black-Box Testing

Tester focuses on external behavior without needing internal code.

Techniques:

- equivalence partitioning
- boundary value analysis
- decision tables
- state-transition testing

🧠 Memory

> **Key Insight**
> **White = inside**

>

> **Key Insight**
> **Black = outside**

---

50. Basis Path Testing

Uses control-flow structure to identify independent execution paths.

Important concept:

Cyclomatic complexity

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

where:

- \(E\) = edges
- \(N\) = nodes

For a connected graph with \(P\) components:

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

Exam importance

Very high.

You should be able to:

1. draw control-flow graph
2. calculate complexity
3. identify independent paths
4. derive test cases

---

51. Control Structure Testing

Tests internal control structures.

Includes:

- condition testing
- loop testing
- data-flow testing
- branch testing

---

52. Unit Testing

Tests individual components/modules.

```
Function
 ↓
Unit Test
 ↓
Pass/Fail
```

---

53. Integration Testing

Tests interactions among modules.

Strategies include:

- top-down
- bottom-up
- sandwich
- big-bang

---

54. System Testing

Tests the complete integrated system against requirements.

Includes:

- functional testing
- performance testing
- security testing
- recovery testing
- usability testing

---

55. Reliability

Reliability = probability that software performs without failure under specified conditions for a specified period.

Important measures/concepts:

- failure rate
- MTTF
- MTBF
- availability

A common relationship:

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

MODULE V — SOFTWARE PROJECT MANAGEMENT & MAINTENANCE

13 Topics

1. Process metrics
2. Software measurement
3. Software project estimation
4. Decomposition techniques
5. Empirical estimation models
6. COCOMO
7. Function points
8. Quality assurance & standards
9. Quality planning
10. Quality control
11. Configuration management
12. Software maintenance
13. Software re-engineering

---

56. Process Metrics

Measure characteristics of the development process.

Examples:

- effort
- schedule
- defect density
- productivity
- cycle time
- rework

---

57. Software Measurement

Quantitative assessment of software/process attributes.

Possible measures:

```
Size
 ↓
Effort
 ↓
Cost
 ↓
Quality
 ↓
Productivity
```

---

58. Software Project Estimation

Estimate:

- effort
- time
- cost
- resources

Common approaches:

- expert judgment
- analogy
- decomposition
- empirical models
- function points
- COCOMO

---

59. Decomposition Techniques

Break a large problem into smaller pieces.

Two common approaches:

Problem decomposition

Break project into manageable tasks.

Process decomposition

Estimate individual activities and combine estimates.

---

60. Empirical Estimation Models

Use observed relationships from historical project data.

General form:

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

The constants depend on the model/calibration.

---

61. COCOMO

Constructive Cost Model

A classic software estimation model associated with Barry Boehm.

Basic COCOMO:

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

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

where:

- KLOC = thousand lines of code
- coefficients depend on project mode

Basic project modes

- Organic
- Semi-detached
- Embedded

🧠 **Memory**

> 💡 **Key Insight**
> **COCOMO = Code → Cost → Months**

---

62. Function Points

Estimate software size based on functionality delivered to users.

Five classic function types:

1. External Inputs
2. External Outputs
3. External Inquiries
4. Internal Logical Files
5. External Interface Files

Then:

```
Count functions
      ↓
Assign complexity weights
      ↓
Unadjusted FP
      ↓
Adjustment
      ↓
Function Points
```

🧯 **Numerical topic**

You should practice complete Function Point calculations.

---

63. Quality Assurance & Standards

Quality Assurance focuses on **process-oriented prevention**.

Standards provide consistent practices and expectations.

Examples of quality-related standards/frameworks may include:

- ISO quality standards
- software process standards
- organizational coding/testing standards

---

64. Quality Planning

Determine:

- quality objectives

- standards
- metrics
- procedures
- responsibilities
- review/testing activities

---

65. Quality Control

Focuses on checking actual outputs against quality requirements.

Think:

```
Quality Planning
      ↓
Define standards
      ↓
Development
      ↓
Measure
      ↓
Compare
      ↓
Correct
```

QA vs QC

QA	QC
Process-oriented	Product/output-oriented
Prevent defects	Detect defects
Proactive	More reactive

---

66. Configuration Management

Controls changes to software artifacts.

Artifacts:

- source code
- requirements
- design
- test cases
- documentation

Core activities:

```
Identify configuration items
          ↓
Version control
          ↓
Change control
          ↓
Status accounting
          ↓
Auditing
```

🧠 Memory

> **Key Insight**
>
> **CM = Control Change**

---

67. Software Maintenance

Software maintenance occurs after delivery.

Types:

Corrective

Fix defects.

Adaptive

Adapt to environmental changes.

Perfective

Improve functionality/performance.

Preventive

Improve maintainability/reduce future problems.

🧠 Memory

> **Key Insight**
>
> **CAP-P**

>

> **Key Insight**
>
> Corrective

> **Key Insight**
>
> Adaptive

> **Key Insight**
> Perfective

> **Key Insight**
> Preventive

Software Re-engineering

Improve/restructure an existing legacy system.

Possible activities:

```
Legacy System
     ↓
Analysis
     ↓
Reverse Engineering
     ↓
Restructuring
     ↓
Forward Engineering
     ↓
Modernized System
```

🔥 COMPLETE SE FORMULA SHEET

Cyclomatic Complexity

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

COCOMO Effort

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

COCOMO Development Time

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

Productivity

$$Productivity = \frac{Output}{Effort}$$

Availability

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

Basic Function Point structure

$$FP = UFP \times VAF$$

where:

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

🧠 ONE-PAGE MEMORY MAP



🎯 WHAT TO MASTER FIRST

For your exam preparation, I'd divide SE into four levels.

🔴 Tier S — Must be able to write/solve

- Waterfall
- Spiral
- Agile
- SRS
- Functional vs non-functional requirements

- Requirements engineering
- Cohesion vs coupling
- UML diagrams
- Verification vs validation
- White-box vs black-box testing
- Basis Path Testing
- Cyclomatic Complexity
- Unit/Integration/System testing
- COCOMO
- Function Points
- Software maintenance
- Configuration management

🟠 Tier A — Strong theory

- Prototyping
- RAD
- Incremental
- Risk management
- Feasibility study
- Requirements validation
- Design concepts
- Static analysis
- Control structure testing
- Reliability
- QA/QC
- Re-engineering

🟡 Tier B

- Definitions/FAQs
- evolving role of software
- process metrics
- detailed management activities
- detailed quality planning

🧠 ACTIVE-RECALL QUESTIONS

Don't reread everything repeatedly. Close the notes and answer these:

M1

1. What is software engineering?
2. Why is software engineering necessary?
3. Explain the Waterfall model.
4. When should a prototype be used?
5. Explain Spiral model.

6. Compare Waterfall and Agile.
7. Compare Incremental and Agile.
8. What is project scheduling?
9. Explain risk management.

M2

1. Functional vs non-functional requirements.
2. User vs system requirements.
3. What is SRS?
4. Characteristics of good SRS.
5. Explain feasibility study.
6. Explain requirement elicitation techniques.
7. Explain requirement validation.
8. Why is requirement management necessary?

M3

1. Explain software design process.
2. Explain abstraction and modularity.
3. Explain cohesion.
4. Explain coupling.
5. Why is high cohesion and low coupling desirable?
6. Draw a use-case diagram.
7. Draw a class diagram.
8. Draw an activity diagram.
9. Draw a sequence diagram.
10. Sequence vs collaboration diagram.

M4

1. Verification vs validation.
2. What is static analysis?
3. Explain software inspection.
4. White-box vs black-box testing.
5. Explain basis path testing.
6. Calculate cyclomatic complexity.
7. Explain unit testing.
8. Explain integration strategies.
9. Explain system testing.
10. Define software reliability.

M5

1. What are process metrics?
2. Explain software estimation.
3. Explain decomposition techniques.
4. Explain COCOMO.
5. Solve a COCOMO numerical.

6. Explain Function Points.
7. Solve a Function Point numerical.
8. QA vs QC.
9. Explain configuration management.
10. Explain four types of maintenance.
11. Explain software re-engineering.

🏆 THE MOST IMPORTANT CONNECTION

Instead of memorizing 67 isolated topics, remember the **software lifecycle**:

```
1. WHAT SHOULD WE BUILD?
          ↓
    REQUIREMENTS
          ↓
2. HOW SHOULD WE BUILD IT?
          ↓
       DESIGN
          ↓
3. HOW DO WE BUILD IT?
          ↓
   IMPLEMENTATION
          ↓
4. DOES IT WORK?
          ↓
     TESTING / V&V
          ↓
5. HOW DO WE MANAGE IT?
          ↓
       PROJECT
          ↓
6. HOW DO WE KEEP IT GOOD?
          ↓
MAINTENANCE + CONFIGURATION
          ↓
7. HOW DO WE IMPROVE IT?
          ↓
   RE-ENGINEERING
```

That single chain gives you a **retrieval scaffold** for almost the entire subject.

The official syllabus confirms that these five modules collectively cover process models/project management, requirements engineering, design/UML, V&V/testing, and estimation/quality/configuration/maintenance.

One caveat: I found the official SE syllabus and the expanded 67-topic study checklist in your uploaded material, but I did **not** find SE-specific PYQ papers among the files currently searchable. So I have not fabricated "actual SE PYQs." If you upload the SE PYQs, I can map every question to these 67 topics and produce a **PYQ-frequency/VVI ranking** just like we did for NLP.