UNIT–II

Software Project Planning & Software Design

Student-Friendly • Exam-Oriented • Relatable Examples

🏫 **RUNNING CASE STUDY — COLLEGE CANTEEN ORDERING SYSTEM**

We will use one continuous example throughout the unit: Student → Selects Food → Places Order → Makes Payment → Canteen Receives Order. This makes each technical concept easier to connect with a real system.

What you will learn

Part	Topics
A. Project Planning	Size Estimation (LOC, Function Point), Cost Estimation, Static Single/Multivariable Models, COCOMO, COCOMO-II
B. Software Design	Cohesion, Coupling, Function-Oriented Design, Object-Oriented Design, User Interface Design

⭐ **EXAM TIP**

For university answers, write in this order whenever possible: Definition → Key points → Example/Diagram → Advantages/Limitations → One-line conclusion.

1. Software Project Planning

Software Project Planning is the process of deciding what will be built, how much work is involved, how long it may take, how many people/resources are needed, and how much it may cost before development begins.

💡 **RELATABLE EXAMPLE**

For the canteen system, the team first decides the required modules—Login, Food/Menu, Order, Payment and Admin—then estimates software size, effort, time, people and cost.

🧠 **MEMORY AID**

Planning = Size + Cost + Time + People + Resources → think: “Plan before you press Go.”

1.1 Why is Project Planning Needed?

- To estimate the size of the software.
- To estimate effort, time and cost.
- To decide the required team and resources.
- To organize development activities.
- To reduce the risk of unrealistic schedules and budgets.

2. Size Estimation

Before estimating effort or cost, we need an approximate measure of software size. Two syllabus methods are LOC and Function Point.

Think of it this way

LOC asks about the amount of code. Function Point asks about the amount of functionality delivered to the user.

2.1 LOC — Lines of Code

LOC measures software size by counting the number of lines of source code required to build the software.

💡 **RELATABLE EXAMPLE**

Suppose the canteen system is estimated as: Login = 500 lines, Food = 800, Order = 700, Payment = 600, Admin = 400. Total = 3,000 LOC = 3 KLOC.

$$KLOC = LOC \div 1000$$

1 KLOC = 1000 LOC

- 2,000 LOC = 2 KLOC
- 10,000 LOC = 10 KLOC
- 25,000 LOC = 25 KLOC

Advantages	Disadvantages
Easy to understand	Language dependent
Easy to calculate after coding	Difficult to estimate accurately before coding
Useful in effort estimation	More lines do not mean better software
Simple numerical measure	May vary with programmer and coding style

⭐ **EXAM TIP**

LOC measures software size by counting lines of source code.

2.2 Function Point (FP)

Function Point Analysis measures software size from the functionality provided to users rather than from the number of source-code lines.

💡 **RELATABLE EXAMPLE**

In the canteen system, placing an order, receiving an order confirmation, checking order status and maintaining order data represent user-visible/system functions.

Component	Meaning	Canteen Example
EI	External Input	Student places order
EO	External Output	Bill / order confirmation
EQ	External Inquiry	Check order status
ILF	Internal Logical File	Student / Food / Order data
EIF	External Interface File	External payment system

🧠 **MEMORY AID**

EI → Enter/Input, EO → Output, EQ → Query, ILF → Internal File, EIF → External Interface File.

2.2.1 Function Point Calculation — Basic Flow

The usual calculation flow is:

Count Functions → Assign Complexity → Calculate UFP → Apply Adjustment Factor → FP

$$FP = UFP \times VAF$$

UFP = Unadjusted Function Points; VAF = Value Adjustment Factor

⭐ **EXAM TIP**

One-line difference: LOC asks “How much code?” while Function Point asks “How much functionality?”

2.3 LOC vs Function Point

Basis	LOC	Function Point
Measures	Lines of code	Functionality
Focus	Code-oriented	User/function-oriented
Language dependency	Language dependent	Mostly language independent
Early estimation	Difficult before coding	Can be estimated earlier
Main idea	Amount of source code	Amount of user-visible functionality

3. Cost Estimation Models

A Cost Estimation Model is a mathematical method used to estimate the effort, development time, resources and cost required to develop software.

Typical relationship:

Software Size → Effort → Development Time → Cost

💡 **RELATABLE EXAMPLE**

Imagine planning a canteen party. You estimate the number of people, menu complexity and cooks before deciding the budget. Software estimation works similarly: size and project factors help justify the expected effort and cost.

3.1 Static Single Variable Model

A Static Single Variable Model estimates effort or cost mainly from one major variable, generally software size.

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

E = Effort, S = Software Size, a and b = constants based on historical data

Basic idea: Bigger software generally requires more effort.

- Simple to understand.
- Uses one major size-related variable.
- May ignore developer experience, complexity, tools, reliability and technology.

3.2 Static Multivariable Model

A Static Multivariable Model uses multiple factors to estimate effort/cost, rather than relying only on software size.

- Software size
- Complexity
- Developer capability/experience
- Required reliability
- Technology and tools

💡 **RELATABLE EXAMPLE**

Two projects may both be 10 KLOC but still need different effort. Project A has an experienced team and simple requirements; Project B has a new team, high complexity and strict security requirements. A multivariable approach captures these differences better.

⭐ **EXAM TIP**

If a question emphasizes several factors affecting cost, think Static Multivariable Model.

3.3 Static Single vs Static Multivariable

Point	Single Variable	Multivariable
Main basis	One major variable	Several variables
Typical example	Software size	Size + complexity + experience + other factors
Complexity	Simpler	More realistic but more involved
Limitation	Ignores many project factors	Requires more project information

4. COCOMO

COCOMO stands for Constructive Cost Model. It was developed by Barry Boehm and is used to estimate software effort and development time, mainly using KLOC.

4.1 Basic COCOMO Equations

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

Effort is measured in Person-Months (PM)

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

TDEV = Development Time

The values of a, b, c and d depend on the project mode.

What is a Person-Month?

10 Person-Months can roughly mean 1 person working for 10 months or 2 people working for 5 months. In real projects, effort and calendar time are not perfectly interchangeable because communication and coordination also matter.

4.2 COCOMO Project Modes

Mode	Size / Nature	Typical Characteristics	Example
Organic	Small	Simple, familiar technology, small team	Library system / small canteen system
Semi-Detached	Medium	Moderate complexity, mixed experience	University management / banking application
Embedded	Large	Highly complex, strict constraints, high reliability	Aircraft control / medical control

🧠 **MEMORY AID**

Organic → Small & Simple; Semi-Detached → Medium & Moderate; Embedded → Large & Complex.

4.3 Basic COCOMO Constants

Mode	Effort	Development Time
Organic	2.4(KLOC)^1.05	2.5(E)^0.38
Semi-Detached	3.0(KLOC)^1.12	2.5(E)^0.35
Embedded	3.6(KLOC)^1.20	2.5(E)^0.32

⭐ **EXAM TIP**

Remember the progression: Organic → Semi-Detached → Embedded and the effort multipliers 2.4 → 3.0 → 3.6.

4.4 Worked Example — Organic Project

Given: Project size = 10 KLOC; Mode = Organic.

$$Effort = 2.4 \times (10)^{1.05}$$

This gives the estimated effort in Person-Months.

For a numerical university problem, substitute the KLOC value, calculate the power, and report the result in Person-Months. Then use TDEV = 2.5(Effort)^0.38 to estimate development time.

For 10 KLOC: Effort ≈ 26.93 Person-Months; Development Time ≈ 8.74 months.

⭐ **EXAM TIP**

Always write the unit: Effort → Person-Months; Development Time → Months.

4.5 COCOMO-II

COCOMO-II is an updated version of COCOMO designed for modern software development environments. It is better suited to development approaches involving reuse, object-oriented development, incremental development, prototyping and newer technologies.

COCOMO	COCOMO-II	Why it matters
Older model	Updated model	Better fit for modern development
Mainly KLOC-based	Supports broader modern estimation approaches	Useful when development is not simply new code from scratch
Traditional development	Modern development practices	Supports contemporary project realities
Less suitable for reuse-heavy work	Considers software reuse	Relevant to components and reusable software

RELATABLE EXAMPLE

If a project builds its system by combining reusable components and developing new modules incrementally, COCOMO-II is designed to handle modern estimation situations better than the original model.

5. Software Design

Software Design converts requirements into a structured solution: modules, responsibilities, relationships, objects, functions and user interfaces.

Design Goal

A good design should divide a large problem into manageable parts, keep related work together, reduce unnecessary dependencies, and make the system easier to understand and maintain.

5.1 Cohesion

Cohesion is the degree to which the elements inside a single module are related to one another.

RELATABLE EXAMPLE

A CalculateBill() module that calculates food price, tax, total amount and prepares the bill has high cohesion because its activities all belong to billing.

MEMORY AID

Cohesion = connection WITHIN one module. High Cohesion = Good.

5.1.1 Types of Cohesion — Worst to Best

No.	Type	Simple Meaning	Example
1	Coincidental	Unrelated tasks in one module	PrintBill + CalculateSalary + SendEmail
2	Logical	Similar-category tasks grouped	Keyboard / Mouse / File input handling
3	Temporal	Tasks grouped because they occur at the same time	Load config + initialize DB + create logs at startup
4	Procedural	Tasks grouped because they follow a sequence	Login → Validate user → Display dashboard

5	Communicational	Parts work on the same data	Read / Update / Display Student Record
6	Sequential	Output of one part becomes input to the next	Enter Order → Calculate Total → Generate Bill
7	Functional	All parts contribute to one specific function	CalculateOrderTotal()

MEMORY AID

C-L-T-P-C-S-F → Coincidental, Logical, Temporal, Procedural, Communicational, Sequential, Functional.

⭐ EXAM TIP

Functional cohesion is the best; Coincidental cohesion is the worst. Good design aims for high cohesion.

5.2 Coupling

Coupling is the degree of dependency between different modules.

💡 RELATABLE EXAMPLE

If the Order module directly depends on many internal details of the Payment module, the coupling is high. Better design keeps modules as independent as practical.

🧠 MEMORY AID

Coupling = connection BETWEEN modules. Low Coupling = Good.

5.2.1 Types of Coupling — Worst to Best

No.	Type	Simple Meaning	Example
1	Content	One module accesses/modifies another module's internal data or code	Directly changing another module's internal details
2	Common	Modules share common/global data	Two modules read/write the same global variable
3	External	Modules depend on an external format/protocol/device	Modules tied to the same external payment protocol
4	Control	One module passes control information to another	processOrder(type), where type controls the action
5	Stamp	A whole data structure is passed when only part is needed	Passing full Student object only to read rollNo
6	Data	Only required data is passed	calculatePrice(quantity, price)

🧠 MEMORY AID

C-C-E-C-S-D → Content, Common, External, Control, Stamp, Data.

⭐ EXAM TIP

Data coupling is the best in this list; Content coupling is the worst. Good design aims for low coupling.

5.3 Cohesion vs Coupling

Point	Cohesion	Coupling
Relationship	Within a module	Between modules
Focus	How strongly module parts belong together	How much one module depends on another
Desired level	High	Low
Example	Billing functions stay together	Order module passes only required data to Payment

🏆 GOLDEN RULE

HIGH COHESION + LOW COUPLING = GOOD SOFTWARE DESIGN.

6. Function-Oriented Design

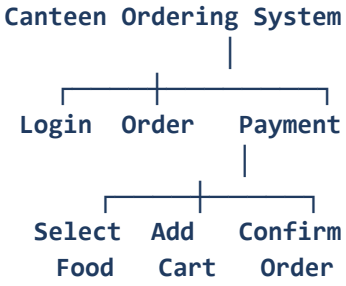
Function-Oriented Design divides the system into functions or procedures and focuses on what functions the system performs.

💡 RELATABLE EXAMPLE

Canteen system: Login → Select Food → Add to Cart → Confirm Order → Make Payment. Each major operation can be represented as a function/process.

6.1 Characteristics

- Focuses on functions or procedures.
- Data and functions are generally treated separately.
- DFD (Data Flow Diagram) is commonly used.
- Structure charts and flowcharts can also represent the design.



7. Object-Oriented Design (OOD)

Object-Oriented Design organizes software around objects and classes. An object combines data (state) and functions/methods (behavior).

Object	Example Data	Example Functions
Student	Name, Roll Number, Email	Login(), PlaceOrder(), ViewOrder()
Food	Food Name, Price, Quantity	DisplayFood(), UpdatePrice()
Order	Order ID, Items, Total	AddItem(), CalculateTotal(), ConfirmOrder()

7.1 Key OOP Concepts in Design

Concept	Meaning	Canteen Analogy
Class	Blueprint/template for objects	Student class is a template for student objects
Object	Actual instance of a class	A particular student object
Encapsulation	Combining data and methods into one unit	Student data + student operations
Inheritance	One class acquires properties/behavior from another	A specialized user type can reuse common user features
Polymorphism	One interface/function can take different forms	Same operation can behave differently for different objects
Abstraction	Show important details and hide unnecessary implementation	User sees “Pay Now” without seeing internal payment processing

🧠 MEMORY AID

OOD = Objects + Classes + Data + Methods. Think LEGO: each object carries its own properties and abilities.

7.2 Function-Oriented vs Object-Oriented Design

Point	Function-Oriented	Object-Oriented
Main focus	Functions / procedures	Objects / classes
View of system	Process-oriented	Object-oriented
Data & functions	Generally separate	Combined in objects
Common representation	DFD, structure chart, flowchart	UML diagrams
Reuse	Function reuse	Class/object/component reuse is emphasized
Example	calculateBill()	Order.calculateBill()

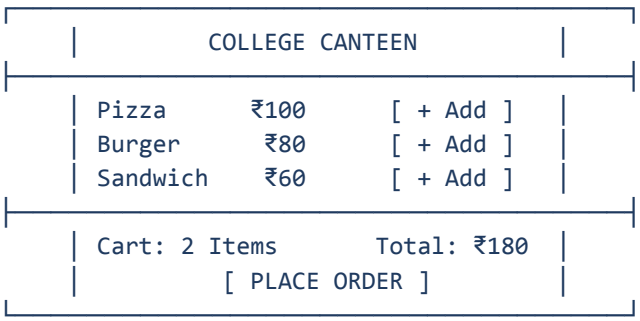
⭐ **EXAM TIP**

One-line distinction: Function-Oriented asks “What does the system DO?”; Object-Oriented asks “What OBJECTS are involved?”

8. User Interface (UI) Design

User Interface Design is the process of designing the interface through which users interact with software—the part the user sees and uses.

8.1 Example — Canteen Ordering Screen



8.2 Goals of Good UI Design

Goal	Meaning
Simple	Easy to understand
Consistent	Buttons, menus and interactions behave consistently
Easy to Learn	A new user can understand it quickly
Fast	Users can complete tasks efficiently
Error-Tolerant	Prevents errors or explains them clearly
User-Friendly	Comfortable and intuitive
Accessible	Usable by people with different abilities

8.3 Basic UI Design Principles

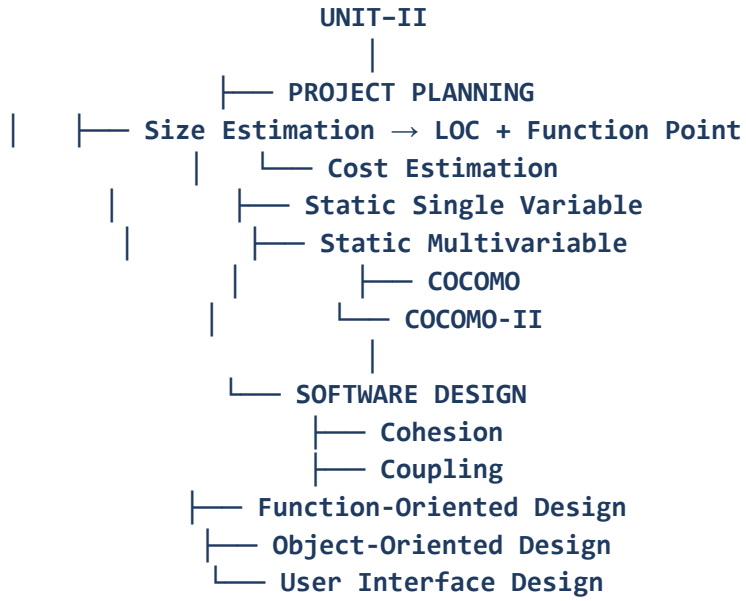
Principle	Meaning	Canteen Example
Visibility	Important options are easy to see	PLACE ORDER button is clearly visible
Feedback	System tells the user what happened	“Order placed successfully.”
Consistency	Similar controls behave similarly	Every Add to Cart button works consistently

Error Prevention	Prevent common mistakes	Do not allow checkout with an empty cart
User Control	User can cancel/go back when appropriate	Cancel Order / Back to Menu

⭐ **EXAM TIP**

A good UI is simple, consistent, easy to learn, fast, user-friendly and able to prevent/handle errors.

9. Unit-II — Quick Revision Map



10. Most Important One-Liners

Topic	Exam-ready definition
LOC	Measures software size by counting lines of source code.
Function Point	Measures software size based on functionality provided to users.
Cost Estimation	Estimates effort, time, resources and cost required for software development.
Static Single Variable	Uses one major variable, generally software size, for estimation.
Static Multivariable	Uses multiple factors such as size, complexity and developer capability.
COCOMO	Constructive Cost Model used to estimate software effort and development time.
COCOMO-II	Updated COCOMO model designed for modern software development environments.
Cohesion	Degree to which elements within a module are related.
Coupling	Degree of dependency between modules.
Function-Oriented Design	Designs software around functions/procedures.
Object-Oriented Design	Designs software around objects containing data and operations.
UI Design	Designs the interface through which users interact with software.

11. Last-Minute Memory Sheet

SIZE

LOC + Function Point

COST

Static Single → Static Multi → COCOMO → COCOMO-II

DESIGN

Cohesion + Coupling → Function-Oriented → Object-Oriented → UI Design

🏆 **GOLDEN RULE**

HIGH COHESION + LOW COUPLING = GOOD SOFTWARE DESIGN.

🧠 **MEMORY AID**

COCOMO Modes: Organic → Semi-Detached → Embedded = Small/Simple → Medium/Moderate → Large/Complex.

🧠 **MEMORY AID**

Cohesion: C-L-T-P-C-S-F = Worst → Best. Coupling: C-C-E-C-C-S-D = Worst → Best.

12. Practice Questions

1. 1. Define software project planning. Why is it required?
2. 2. Explain LOC and KLOC with an example.
3. 3. Differentiate between LOC and Function Point.
4. 4. Explain the five components of Function Point.
5. 5. What is a cost estimation model?
6. 6. Differentiate Static Single Variable and Static Multivariable models.
7. 7. What is COCOMO? Write its basic equations.
8. 8. Explain Organic, Semi-Detached and Embedded project modes.
9. 9. Calculate COCOMO effort and development time for a given KLOC value.
10. 10. What is COCOMO-II? Why is it useful for modern development?
11. 11. Define cohesion. Explain its types from worst to best.
12. 12. Define coupling. Explain its types from worst to best.
13. 13. Differentiate cohesion and coupling.
14. 14. Explain Function-Oriented Design with a suitable example.
15. 15. Explain Object-Oriented Design and key OOP concepts.
16. 16. Differentiate Function-Oriented and Object-Oriented Design.
17. 17. What is User Interface Design? Explain goals and principles of a good UI.

⭐ **EXAM TIP**

For a 5–10 mark answer: start with a definition, explain 4–7 key points, add a relatable example/diagram, and finish with a short conclusion or comparison.