CS603 SOFTWARE ARCHITECTURE & DESIGN

SHORT NOTES FOR MIDTERM

SYLLABUS TOPIC 01 TO 76

PREPARED BY ALMAS AFZAL

PREPARED BY ALMAS AFZAL

Topic_1
Design (What & Why)

What is Design?

Design can be defined by two ways:

- ✓ As a Verb
- ✓ As a Noun

As a Verb: It is defined to make draw plans from something, for examples Clothes or buildings.

As a Noun: It is defined as a drawing or sets of drawings showing how a building or product is to be made and how it will work and look.

What is Software Design?

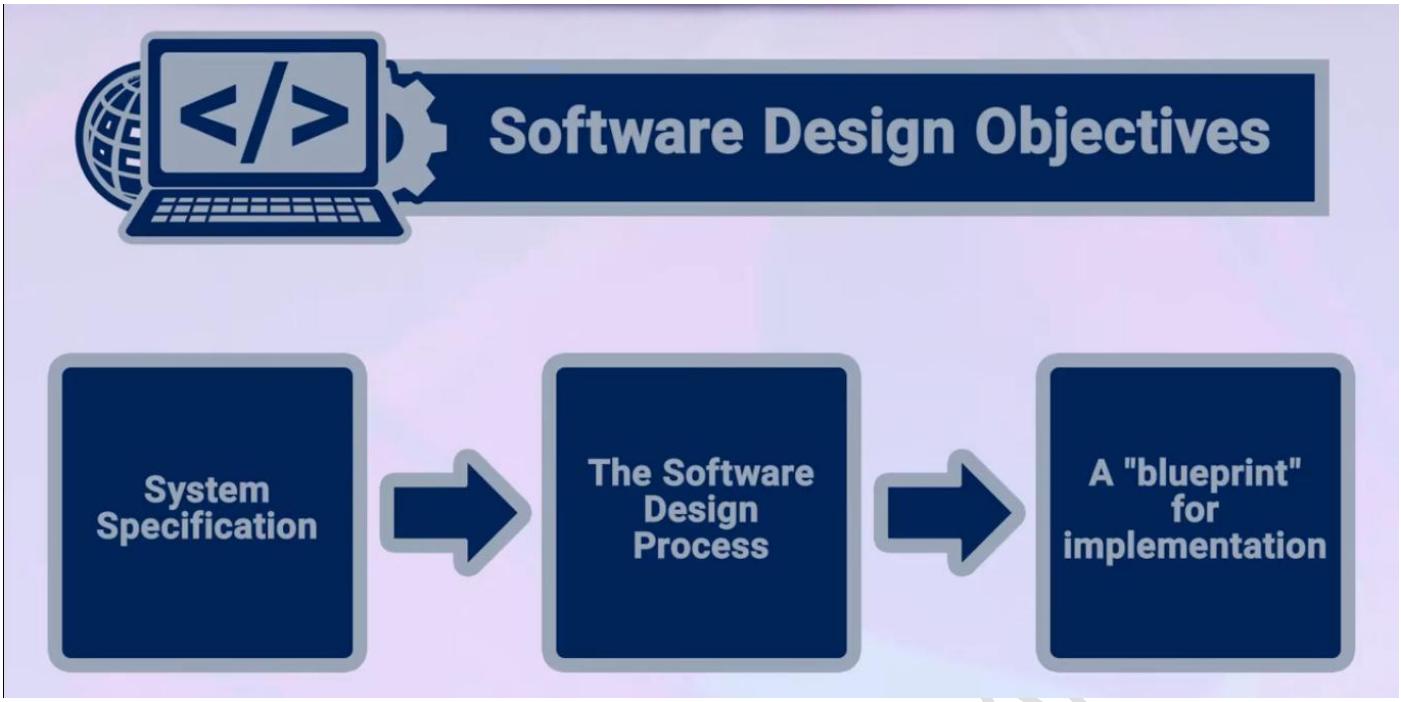
Software Design is the process of defining the architecture, components, modules, interfaces, and data for a software system to satisfy specified requirements. It acts as a blueprint for software development and ensures that the system is structured, maintainable, and efficient.

It includes design principles, patterns, and methodologies to create high-quality software

Why do we use Design?

We design things to improve situations, solve problems ans explore alternatives.

Topic_2
Software Design (Objectives)



1. System Specification

*System specification defines **what** the software should do.*

It includes:

- Functional & Non-functional Requirement
- User Needs & Constraints
- Hardware & Software Requirements

2. The Software Design Process

The software design process is a structured approach that includes:

Requirement Analysis – Understanding what the system should do.

Architectural Design – Defining the system's high-level structure.

Detailed Design – Designing individual components/modules.

Validation & Verification – Checking the design to avoid errors.

3. A Blueprint For implementation

*Software design acts as a **blueprint (plan)** that guides developers.*

PREPARED BY ALMAS

It includes:

Diagrams (UML, Flowcharts) to visualize system components.

Architecture patterns (MVC, Layered Architecture) to organize code.

Coding standards & guidelines to ensure quality.

During the design phase , software engineers apply their knowledge of:

- The problem domain
- Implementation Technologies

Basic Objectives of Software Design:

- ✓ It translates system specification into plans for the technical implementation of the software.
- ✓ It serves as a technical plan for implementation.
- ✓ It specifies the overall structure and organization of the eventual code.

The system or Program should meets the customer's needs.

A Software Should be:

- Easy to Implement
- Efficient
- Easily Extendible to meet new needs

PREPARED BY ALMAS AFZAL

Topic_3 & 4
Software Design (Complexity & Size)

Remember that:

Poorly designed programs are difficult to understand & Modify.

Software design complexity refers to how difficult it is to **understand, modify, test, or maintain** a software system. It depends on:

Code Structure – Messy or well-organized?
Module Interactions – Are components highly dependent on each other?
Algorithms & Logic – Are they simple or too complex?
Scalability – Can it handle growth easily?

Example: A simple **calculator app** has low complexity, while an **AI-based chatbot** has high complexity due to advanced algorithms.

Software Design Size:

Software size refers to **the amount of code, number of modules, and overall system scope**. It is measured using:

Lines of Code (LOC) – Total number of written code lines.
Function Points (FP) – Measures functionality based on inputs, outputs, and interactions.
Cyclomatic Complexity – Measures the number of independent paths in the code.

Impact of Size:

Small-sized software is easier to develop and maintain.

Large-sized software requires careful planning, modularity, and documentation.

Topic_5
Software Design (types of Complexities)

PREPARED BY ALMAS AFZAL

1. Essential Complexities
Complexities that are inherit in the problem.

2. Accidental / incidental Complexities
Complexities that are artifacts of the Solution.

The total amount of complexity in a software design is:
Essential Complexities + Accidental / incidental Complexities

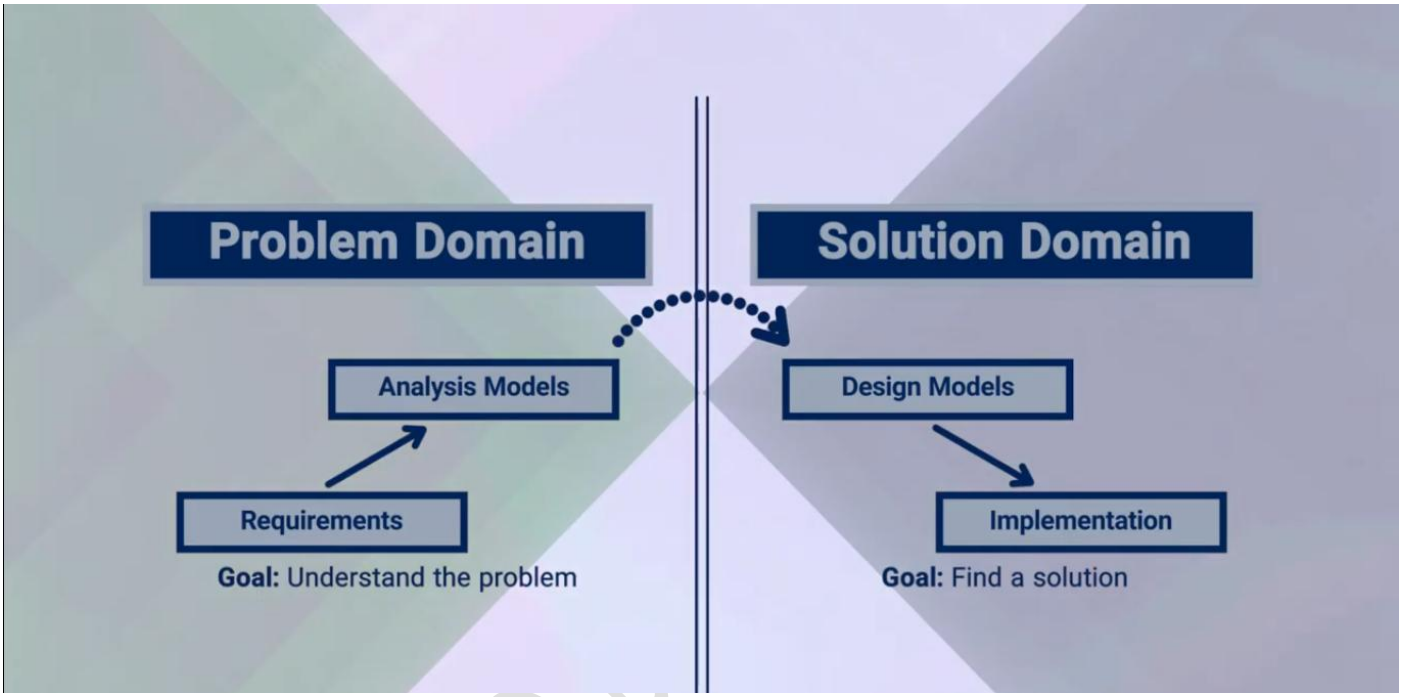
Design: is an Antidote to complexity

Design is a primary tool for managing essential & accidental complexities in Software.

Topic_6
Why is Software Design Hard?

Why is Software Design Hard?

Software Design is hard because design is an abstraction of the solution which has yet to be created.



Topic_7
Software Design
A science or an art?

Software design is considered both **a science and an art** because it involves structured principles (science) and creative problem-solving (art).

- Software design is a **science** because it follows engineering principles.
- It is also an **art** because it requires creativity and problem-solving skills.
- The best software designs balance both aspects for efficiency and user satisfaction.

MCQ:

Science is a **repeatable** process.
For Example: 2 + 2 = 4

Topic_8
Software Design (A Wicked Problem)

A Wicked Problem
A problem that can only be clearly defined by completely solved it.

*Software design is **not just a step-by-step process**; it requires **continuous adaptation, problem-solving, and decision-making**. Since **there is no perfect solution**, designers must focus on **flexibility, scalability, and maintainability**.*

Topic_9
Design Process
More Systematic & Predictable

The design process can be made more systematic & Predictable through the application of:

- Methods
- Techniques
- Patterns
- All applied according to principles & heuristics.

Topic_10
Dealing with Software Complexity
Different rules & Principles

There are two types of Complexities:
Essential Complexity
Accidental Complexity

PREPARED BY ALMAS FEZAL

Remember that:

A good design does not reduce the total amount of essential Complexity in a solution.
But it will reduce the amount of Complexity that a programmer has to deal with at any one time.

Principles to dealing with Software Complexity

1. **Modularity:** We Subdivided the solution into smaller & easier to manage components.
*This technique is also called **Divide & Conquer***

2. **Hierarchical Organization:** Larger components may be composed of smaller components.

3. **Information Hiding:** Hide details and complexity behind the simple interface.

4. **Abstraction:** Use abstractions to suppress details in places where they are unnecessary.

Topic_11
Software Design (Characteristics)

PREPARED BY ALMAS AFZAL

Deterministic Process:

A Deterministic Process is one that produces the same output given the same inputs.

MCQ:

Design is a non-deterministic Process.

Because no two designers or design process are likely to produce the same output.

MCQ:

Design is Heuristic based Process.

Because design is non-deterministic Process , design techniques tend to rely on Heuristics and rules-of-thumb rather than repeatable processes.

Remember that:

Design is a Emergent: The final decision evolves from experience and feedback.

Design is an iterative and incremental process where a complex system arises out of relatively simple interactions.

Topic_12
Software Design (Benefits of Good Design)

Benefits of Good Design:

- A good design reduces the Software Complexities making easier to understand and modify & implement.
- It make easy maintainance.
- Code Reuse
- It improves Software quality
- Good design exposes defects and makes it easier to test the Software

---

Topic_13
Software Design (A Generic Design Process)

Processes are:

- **Software Requirements** (Understand the Problem)
- **System Specification** (Construct a "black-box" model of solution)
- **Architecture & Design Patterns** (Look for exiting solutions that cover some or all of the software design problems identified)
- **Discover missing Requirements**
- **Documents & review design**
- **Refactor** (iterate over solution)

Topic_14
Software Design (Inputs to the Design Process)

There are two inputs:

Domain Knowledge: For Example, if it is a healthcare application the designer will need some knowledge of healthcare systems and concepts.

Implementation Knowledge: capabilities & limitations of eventual execution environment.

PREPARED BY ALMAS AFZAL

Topic_15
Desirable Internal Design (Characteristics)

We know that:

Poorly designed programs are difficult to understand & Modify.
&
Ease of maintenance is one of the prime objective.
Because
Your code will be read more often than it is written.

Desirable Internal Design:

- Minimal Complexity (Keep it Simple)
- Extensibility (Design for today but with an eye toward the future)
- Loose Coupling (Minimize dependencies b/w modules)
- Re-usability

Topic_16
Software Design (A Good Design)

A Good design has:

- Effiecient Code
- Compact Implementation
- Most Maintainable

In Maintainable Design:

- Cost of system changes is minimal
- Readily adaptable to modify existing functionality & enhance functionality
- Design should be understandable
- Changes should be local in effect
- High cohesion & low coupling

PREPARED BY ALMAS AFZAL

Topic_17
Software Design (Coupling)

Coupling:

The measure of dependencies b/w modules & components.

OR:

The degree of inter-dependence b/w modules & components.

A dependency exists b/w two modules/components if a change in one could require a change in the other.

Basically:

Coupling is an inter-component Relationship.

Measuring Coupling:

The degree of coupling b/w two modules is determined by:

- The number of interfaces b/w modules (Quantity)
- Complexity of each interface (determined by the type of communication quality)

Why high degree coupling is bad?

- Highly coupled systems have strong interconnection.
- High dependence of programs unit upon one another.
- Difficult to test, reuse & understand separartly.

So:

Loosely coupled modules are less dependent upon each other and hence are easier to change.

How to reduce Coupling?

Coupling can be reduced by:

- Eliminating unnecessary relationships
- Reducing the number of necessary relationships
- Easing the "tightness" of necessary relationships

PREPARED BY ALMAS AFZAL

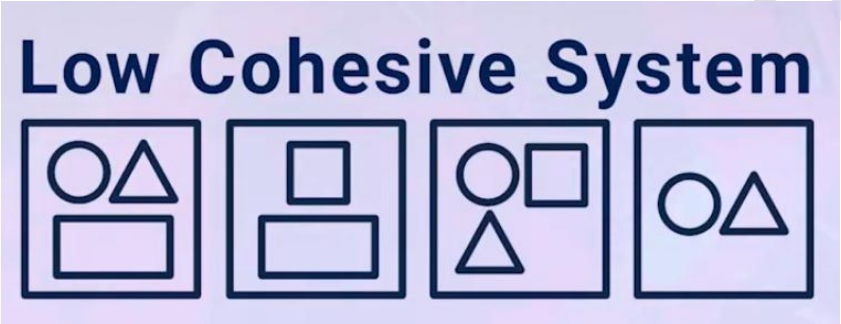
Topic_19
Software Design (Cohesion)

Cohesion:

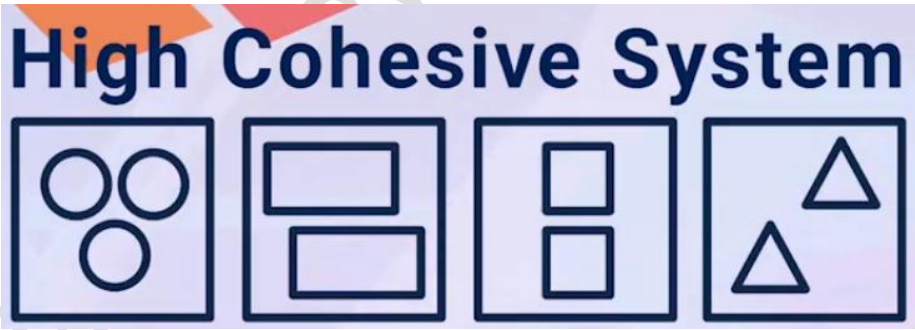
A measure of the strength of relationship b/w:

- The methods and data of a class and some underlying purpose of concept served by that class.
- The class's methods and data themselves.

A module has high cohesion if all of its elements are working towards the same goal.



And we can make it High Cohesion by:



Highly Cohesive Modules are:

- ✓ Easier to understand
- ✓ More Robust
- ✓ More Reliable
- ✓ More Reuseable

Low Cohesive Modules are:

- ✓ Difficult to understand
- ✓ Difficult to Maintain

- ✓ Difficult to Test
- ✓ Difficult to Reuse

Topic_20
Coupling & Cohesion (Relationship)

As we know that:

Highly Cohesive Modules are:

- ✓ Easier to understand
- ✓ More Robust
- ✓ More Reliable
- ✓ More Reuseable

&

Lowly Coupled Modules are:

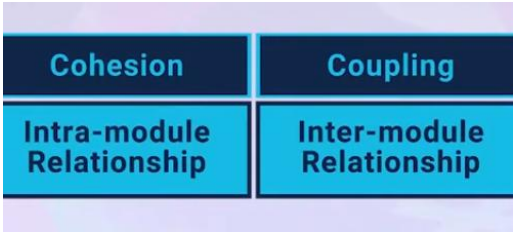
- ✓ Easier to read & understand
- ✓ Easier to Modify
- ✓ More Reliable
- ✓ More Reuseable
- ✓ Easier to deevelop & test

You can see that both are SAME.



So:

PREPARED BY ALMAS AFZAL



Topic_21
Software Design Strategies
Structured & Object-Oriented Design

Structured Design:

- ✓ System is designed from a functional view point.
- ✓ Conceived a a set of functions that are used to solve a problem at hand.
- ✓ The system state is centralized and is shared among the functions operating on that state.

Object-Oriented Design:

- ➢ Based on the idea of information hiding.
- ➢ System as a viewed as a collection of objects.
- ➢ The system intelligence and state is decentralized and each object manages its own state information.

Attributes (State)
Operation (Intelligence)

- ➢ Objects communicate by exchanging information messages.

Conclusion:

Object-Oriented Design is to be considered best & more maintainable as compare to the Structured Design.

Topic_22
Object-Orientation Basic Concepts (Object)

PREPARED BY ALMAS AFZAL

The Object:

- An object is tangible entity that exhibits some well defined behavior.
- An object represents an individual , identifiable item , unit or entity either real or abstract , with a well defined role in the problem domain.

A Object must have:

- State
- Behavior
- Identity

The State:

- ✓ The state of an object is encapsulated within the object.
- ✓ It is the current values of the properties of the object.

The Behavior:

- ✓ How an objects acts and reacts in terms of its state changes and message passing.
- ✓ Part of system intelligence delegated to the object.

The Identity:

- ✓ Identity is that property of an object which distinguishes it from all other objects.

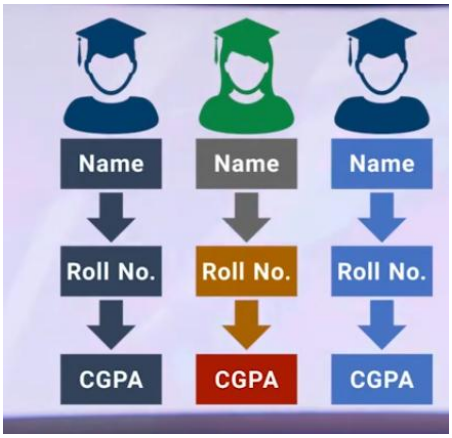
For Example:
Your CNIC , your Phone number & your email adress etc…

Topic_23
Object-Orientation Basic Concepts (Class)

The Class:

- A Class represents similar objects.

PREPARED BY ALMAS AFZAL



- A class is a template of an object that includes properties and behavior.
- It represents an abstraction.

Abstraction has:
Interface = outside view / public part
Implementation = inside view / private part

Topic_24
Relationship b/w Classes & objects (Association)

Objects & Classes are two building blocks of the system.

Relationship b/w Classes:

- Behavior of the system is defined by the manner in which the objects interact with one another.
- In order to Construct a software system , we need mechanism that create connections b/w these building blocks.

Connections can be done by two ways:
Association
Inheritance

Association:

- It indicates that the objects of the two classes are related in some non-hierarchically way.
- It implies that an object of one class is making use of an object of another (or some) class.
- It indicated simply by a solid line connecting the two line icons.
- It does not imply that there is always a link b/w all objects of one class and all objects of the other.
- It does imply that there is a persistent , identifiable connection b/w two classes.

PREPARED BY ALMAS AFZAL

Remember that:

There is a Conceptual Link b.w Classes.
&
There is a Physical Link b.w Objects.

Attributes of Association:

- ✧ Descriptive Name
- ✧ Arity of the connection
- ✧ Direction
- ✧ Roles

Topic_25
Relationship b/w Classes & objects
Aggregation & Composition

PREPARED BY ALMAS AFZAL

Aggregation:

- ✓ It is a form of association that specifies a whole-part relationship b/w an aggregate (a whole) and a constituent part.

It indicated by a "Hollow Diamond (→)"

There is a Sub-Class of aggregation which is called Composite Aggregation , also known as Composition.

Composition:

- ✓ It implies that each instance of the part belongs to only one instance of the whole , and the part cannot exist except as part of the whole.
- ✓ It indicated by a "Filled-in-Diamond"

MCQs:

Aggregation: Loose relationship
Composition: Strong Relationship

Topic_26
Relationship b/w Classes & objects (Inheritance)

Inheritance:
Represents a hierarchy of abstraction in which one class inherits from one or more super-classes.

Inheritance Hierarchy:
It shows the relationships among super-classes & Sub-classes.

Topic_27
Object Oriented Basic Concepts
Abstract Classes & Interfaces

PREPARED BY ALMAS AFZAL

Abstarct Class:

We might know the specifications for a class, but might not have the information needed to implement.
In such cases, we can implement a class partially using what are called abstract classes.

Abstract Class Provides:
A basic info that other classes can inherit from.

MCQ:
An instance of Abstract Class cannot be created.

Interface:

- A mechanism to achieve abstraction
- A collection of abstract methods.
- No state or Implementation.
- It's just a specification of system.

Interface vs Abstract Class:

Interface	Abstract Class

A class that implements an interface must provide an implementation of all the methods of that interface.	Abstract Classes can be inherited without implementing the abstract methods.
Interfaces can have no state or implementation.	Abstract Clases may have contain state.

In General terms:

The differences are not necessarily well-defined b/w and interfaces an Abstract Class.

Topic_28
Relationships Basic Concepts
Object Oriented Programming Defined

PREPARED BY ALMASAFZAL

Object Oriented Programming By Grady Booch:

A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

There are two main Concepts in this definition:

1. **Hierarchy of Classes__** inheritance relationship among classes
2. **Cooperative collections of objects__** Association-relationship among instances of classes.

So we are talking about two types of relationships:

- **Class Relationship**
- **Objects Relationship**

A class relationship is through ***Inheritance.***
An object relationship is through ***Association.***

Topic_29

Relationships Basic Concepts (Association)

According to UML Reference Manual:

- An Association is a relationship among ***two or more*** specified ***classifiers*** that describes ***connection*** among their ***instances.***
- Associations are the ***"Glue"*** that holds together a system.
- Without Associations, there is only a set of unconnected Classes.

Types of Association:

1. Uses Relationship
- Simple Association

2. Whole-Part Relationship
- Aggregation
- Composition

Topic_30
Relationships Basic Concepts
Aggregation & Composition

PREPARED BY ALMAS AFZAL

Aggregation according to UML Reference Manual:

A form of association that specifies a whole-part relationship b/w an aggregate (a whole) and a constituent part.

Remember that:
The distinction b/w aggregation and association is often a matter of taste rather than a differences in semantics.
Think of it as a modeling placebo.

Aggregation according to UML Distilled By Martin Fowler:

"Aggregation is a strictly meaningless;

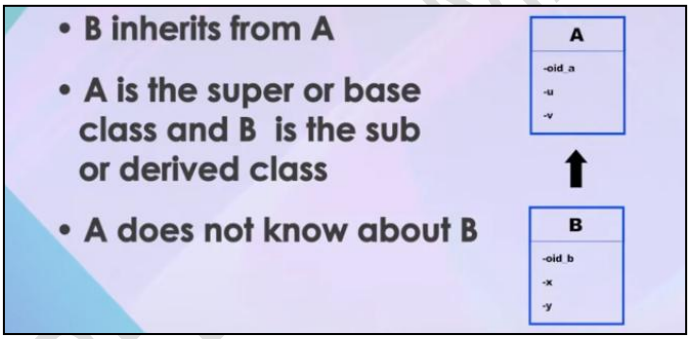
As a result, I recommend that you ignore it in your own diagrams."

Association according to UML Reference Manual:

A form of aggregation association with strong ownership and coincident lifetime of parts by the whole.

Topic_31
Object Lifetime & Visibility (Inheritance)

Inheritance:



Skip Topic_32 & 33

Topic_34
Object to Relational Mapping (Basics)

Object-relational mapping (ORM)

PREPARED BY MAS AFZAL

It is a programming technique that simplifies interactions between object-oriented programming languages and relational databases.

Also Called:

- O2R
- ORM
- O/RM
- O/R

Topic_35 & 36
Object to Relational Mapping
(Mapping Inheritance)

PREPARED BY ALMAS AFZAL

We can Map inheritance relationship in database relational database in three ways:

- Filtered Mapping
- Horizontal Mapping
- Vertical Mapping

In Filtered Mapping:
All the classes of an inheritance hierarchy are mapped to one table.

In Horizontal Mapping:
Each concrete class is mapped to a separate table and each such table includes the attributes and the inherited attributes of the class that it is representing.

In Vertical Mapping:
Each class of the inheritance hierarchy, whether abstract or concrete , is mapped to a separate table.

Topic_37
Object to Relational Mapping
(Mapping Composition)

Mapping Composition in ORM refers to how we map **composite objects** (i.e., objects that contain other objects) from an object-oriented model to a relational database structure.

Which One to Use?

- Use **embedded mapping** if the composed object is **only used inside** the parent and **has no identity of its own**.
- Use **separate table** if the composed object **has meaning or identity independently.**

Topic_38
Object to Relational Mapping
(Mapping Association)

In Object-Oriented Programming (OOP), **association** means a relationship between two or more classes. When we map these relationships to a relational database, it's called **Mapping Association** in ORM.

Types of Associations in OOP:

- One-to-One
- One-to-Many / Many-to-One
- Many-to-Many

Topic_39
Object to Relational Mapping
(Mapping Cardinality of Association)

PREPARED BY ALMAS AFZAL

Cardinality refers to ***how many instances*** of one class are associated with how many instances of another class in an object-oriented model. When we move from OOP to a relational database, we must map this cardinality properly using foreign keys or join tables.

Types of Cardinality in Association

Association Type	Description
One-to-One	One object relates to only one object.
One-to-Many	One object relates to multiple objects.
Many-to-One	Many objects relate to one object.
Many-to-Many	Many objects relate to many objects.

Topic_40
Object to Relational Mapping
(Summary)

So you can skip this topic :)

Topic_41
Object Oriented Programming
(Inheritance & Polymorphism)

Basically OOP is based on 4 pillars:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

PREPARED BY ALMAS AFZAL

Inheritance:

Inheritance is the mechanism by which one class (child/sub-class) **inherits properties and behavior** (fields and methods) from another class (parent/super-class).

Polymorphism:

Polymorphism means **"many forms"**. It allows objects to be treated as instances of their parent class, while behaving differently depending on the actual subclass.

Types of Polymorphism:

- Compile-time Polymorphism (Method Overloading)
- Run-time Polymorphism (Method Overriding)

Difference Between Inheritance and Polymorphism:

inheritance	Polymorphism
Is a mechanism to acquire features from another class	Is a mechanism to perform a single action in different ways
Promotes code re-usability	Promotes flexibility and extensibility
Uses extends or inherits	Uses method overriding or overloading

---

Topic_42
Object Oriented Programming
(Object Creation & Factory Method)

In Object-Oriented Programming, **object creation** is a fundamental concept — it's how we create actual instances of classes so we can use their data and behavior.

Object Creation (Basic Way)

PREPARED BY ALMAS AFZAL

The most common way to create an object is by using the ***new keyword*** (in many languages like Java, C#, C++).

Factory Method (Advanced Way of Object Creation)

Factory Method is a **design pattern** that provides an **indirect way of creating objects**.

Instead of using new everywhere, you **delegate object creation to a method** — often in a separate factory class.

Benefits of Factory Method:

- Centralizes object creation logic
- Makes it easy to switch or add new object types
- Promotes loose coupling
- Encourages use of interfaces/abstract classes

So:

- **Object Creation** is essential in OOP; normally done using new ClassName().
- **Factory Method** is a design pattern used to **encapsulate and control object creation**.
- Factory Method makes your code **more modular, reusable, and easier to maintain**.

Topic_43
Object Oriented Programming
(The Magic Behind Polymorphism)

Polymorphism is one of the **core principles of OOP** and is often called “the magic” that gives OOP its **power and flexibility**. Let's understand what this "magic" actually is.

What is Polymorphism?

Polymorphism means **"many forms"** — the ability of a single function, method, or operator to behave differently based on the context or object type.

PREPARED BY ALMAS AFZAL

Why is this "Magic" Important?

- **Extensibility**: You can add new classes without changing existing code.
- **Flexibility**: One interface, many implementations.
- **Clean code**: Encourages using parent class references with child class behavior.

Topic_44
Object Oriented Programming
(Implementing Composition)

PREPARED BY ALMAS AFZAL

What is Composition?

Composition is a design principle in Object-Oriented Programming where **one class contains an object of another class** and uses it to build more complex functionality.

Why Use Composition?

- Encourages **code reuse** without tight coupling.
- More **flexible** than inheritance.
- Follows **"favor composition over inheritance"** principle.

How to Implement Composition:

- **Create two classes**
- In one class, **create an object** of the other class
- Use that object to **call its methods**

Topic_45
Object Oriented Programming
(Inheritance vs Composition)

What are Inheritance and Composition?

Both are techniques used in Object-Oriented Programming to **reuse code** and build relationships between classes — but they do it in **different ways**.

Feature	Inheritance	Composition
Relationship	is-a	has-a
Coupling	Tight	Loose
Flexibility	Less	More
Re usability	High (for behavior)	High (for components)
Modifiability	Harder to change	Easier to update
Usage	When you want to extend behavior	When you want to combine behaviors

---

Topic_46
Object Oriented Programming
(Implementing Uni-directional Association)

PREPARED BY ALMAS ATAL

What is Uni-directional Association?

Uni-directional association is a **one-way relationship** between two classes where **one class knows about the other**, but **not vice versa**.

- **Uni-directional** = one class has a reference to another.
- Helps keep relationships **simple and less coupled**.
- Common in real-world systems where reverse knowledge isn't required.

When to Use Uni-directional Association:

- When only one class needs to access the other.
- To reduce unnecessary dependencies.

➢ When data flow is only one-way.

Topic_47
Object Oriented Programming
(Implementing Bi-directional Association)

What is Bi-directional Association?

Bi-directional association is a two-way relationship between classes where **both classes know about each other**.

- ❖ Bi-directional = both classes hold references to each other.
- ❖ Ensures mutual communication between objects.
- ❖ Can be more complex, but also more powerful.

When to Use Bi-directional Association:

- ❖ When objects need to interact in both directions.
- ❖ For navigation or data flow from either side.
- ❖ In real-world modeling, like students–departments, teachers–courses, customers–orders.

Topic_48
Solid Design Principles
(Introduction)

What are SOLID Principles?

SOLID is an acronym representing five key principles of **object-oriented design** that help developers create **more maintainable, understandable, and flexible software**.

PREPARED BY ALMAS AFZAL

The Five SOLID Principles:

Solid Principles Originally gathered by Robert C.Martin (aka uncle Bob)

Letter	Principle Name	What It Means
S	Single Responsibility Principle	A class should have only one reason to change.
O	Open/Closed Principle	Software entities should be open for extension but closed for modification.
L	Liskov Substitution Principle	Sub-types must be substitutable for their base types.
I	Interface Segregation Principle	Many client-specific interfaces are better than one general interface.
D	Dependency Inversion Principle	Depend on abstractions, not on concrete implementations.

Topic_49 & 50
SOLID Design principles
Single Responsibility Principle SRP

PREPARED BY ALMAS AFZAL

What is the Single Responsibility Principle (SRP)?

SRP states that **a class should have only one reason to change**. This means each class should have **only one job or responsibility**.

Why SRP is Important?

- ❖ Keeps classes simple and focused.
- ❖ Makes code easier to maintain and understand.
- ❖ Changes in one responsibility don't affect unrelated parts of the code.

So:

- SRP helps reduce complexity.
- Encourages better code organization.
- Makes your system easier to test, maintain, and extend.

Topic_51 & 52
SOLID Design principles
Open Closed Principle OCP

OCP introduced by Bertrand Meyer.

The Open Closed Principle:

A software artifact should be open for extension but closed for Modification.

This means you should be able to add new functionality without changing existing code.

Why OCP is Important?

- Prevents breaking existing code.
- Makes your system easier to maintain and extend.
- Supports flexibility and scalability.

PREPARED BY ALMAS AFZAL

Topic_53 & 54
SOLID Design principles
Liskov's Substitution Principle LCP

LSP states that:

"Objects of a subclass should be replaceable with objects of the super-class without affecting the correctness of the program."

In simple words:

Child classes should behave like their parent classes.

They can extend functionality, but should not break expectations.

Why LSP is Important?

- Ensures that your code works correctly with inheritance.
- Prevents unexpected bugs when replacing base types with derived ones.
- Supports polymorphism safely.

Topic_55
SOLID Design principles
The True Meanings of Is-A Relationship

Validity is not intrinsic.
A model, viewed in isolation, cannot be meaningfully validation.

What is the “Is-A” Relationship?

In Object-Oriented Programming, the "Is-A" relationship refers to inheritance.

If class B inherits from class A, we say "B is-a A".

Example:

- ✓ A Dog is-a Animal.
- ✓ A Car is-a Vehicle.

But to truly satisfy this Is-A relationship, the child class must behave like the parent class without breaking functionality — this is exactly what the Liskov’s Substitution Principle (LSP) is all about.

What’s the True Meaning of Is-A?

- It’s not just about structure or inheritance.
- It’s about behavioral compatibility.

PREPARED BY ALMAS AFZAL

- A child class should truly act like the parent class.

Topic_56 & 57
SOLID Design principles
Interface Segregation Principle ISP

What is Interface Segregation Principle (ISP)?

Clients should not be forced to depend on interfaces they do not use.

In simple terms:

Make small, specific interfaces rather than one big, general one.

A class should only implement what it really needs.

Why ISP is Important?

- Reduces unnecessary code in classes.
- Makes interfaces more focused and meaningful.
- Keeps the code clean, flexible, and easy to maintain.

✓ Do's	✗ Don'ts
Use small, specific interfaces	Don't create one big interface
Keep interfaces focused	Don't force classes to implement unused methods
Make your code more flexible	Avoid tightly coupled systems

PREPARED BY ALMAS AFZAL

Topic_58 & 59

SOLID Design principles
Dependency Inversion Principle DIP

What is Dependency Inversion Principle (DIP)?

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend on details. Details should depend on abstractions."

In simple words:

Instead of depending directly on concrete classes, depend on interfaces or abstract classes.

Why DIP is Important?

- Reduces tight coupling between components.
- Makes the code more flexible, testable, and maintainable.
- Helps in changing implementations without breaking code.

Topic_60 & 61
Good OO Design Principles
Law of Demeter LoD

Law of Demeter:

A style rule of Building Systems.

A Principle of least knowledge

What is Law of Demeter (LoD)?

Each unit (class/method) should only talk to its immediate friends — not strangers.

PREPARED BY ALMAS AFZAL

In simple terms:

- A method should only call:
- Its own methods
- Methods of objects it creates
- Methods of its parameters
- Methods of directly held objects (fields)

Why is it Important?

- Reduces tight coupling
- Improves encapsulation
- Makes code easier to maintain and test

Topic_62
Software Design & Architecture
Action-Oriented Design

PREPARED BY ALMAS AFZAL

What is Action-Oriented Design?

Action-Oriented Design is a procedural programming style where functions or actions are the main focus, rather than objects or data.

In other words:
It focuses on what the system should do, rather than what the system is.

In Action-Oriented Design (Procedural Programming) depends on:

Identity___ Who I am?
Data___ What I know?
Methods___ What I do?
Relationships___ Who I know?

There are a basic challenge during Design:

1. Poorly distributed system intelligence

Topic_63
Software Design & Architecture
Handling Poorly Distributed Sytsem Intelligence

What is Poorly Distributed System Intelligence?

When the logic and decision-making of a software system is unevenly or improperly spread across different parts or components, it leads to:

- Some modules doing too much work (overloaded with responsibilities)
- Other modules doing too little or no work (underutilized)
- Resulting in poor maintainability, debugging difficulty, and inefficiency

Problem	Description
High Coupling	Components are tightly dependent on each other
Low Cohesion	Components have unrelated responsibilities
Difficult to Maintain	Changes in one place affect many others
Poor Performance	Bottlenecks where intelligence is concentrated
Scalability Issues	Hard to scale parts unevenly loaded

PREPARED BY ARIAS AFZAL

Topic_64
Modeling Policy
Jacobson's Heuristic

What is Modeling Policy?

Modeling Policy refers to the set of guidelines or best practices used when designing and modeling software systems, especially in object-oriented analysis and design.

Jacobson's Heuristic:

Ivar Jacobson, one of the pioneers of Object-Oriented Software Engineering, proposed a useful heuristic to help decide what belongs in which object:

"If it moves, it's an object; if it doesn't, it's a policy."

Why is Jacobson's Heuristic useful?

- Helps separate system rules from entities
- Makes system design clearer and more modular
- Promotes better assignment of responsibilities in the design

---

Topic_65
Handling Roles
Player Role Pattern

PREPARED BY ALMAS AFZAL

What is the Player Role Pattern?

The Player Role Pattern is a design approach used in object-oriented design to handle situations where an object can play different roles in different contexts.

Key Concepts:

Player: The object or entity that takes on one or more roles.
Role: The specific behavior or responsibility the player assumes in a particular context.

Why use Player Role Pattern?

- Flexibility: Allows the same object to behave differently in different scenarios without changing its core class.
- Separation of concerns: Roles encapsulate behavior relevant to specific contexts, improving modularity.
- Re-usability: Roles can be reused across different players or contexts.

Topic_66
Handling Multiple Discriminators
Using Player-Role Pattern

A ***Discriminator*** is a label that describes the criteria used in the specialization.

Multiple Discriminators:
In object-oriented design, multiple discriminators occur when a class needs to vary its behavior based on more than one axis or classification.

For example:
Let's say you have a class Vehicle, and you want to classify it based on:

- Type (e.g., Car, Truck, Bike)
- Fuel (e.g., Petrol, Diesel, Electric)

These are two independent discriminators.

Problem:
Using traditional inheritance to handle this would cause combinatorial explosion.
You'd need classes like:

PetrolCar, DieselTruck, ElectricBike...

Too many combinations = too complex to manage.

Topic_67
Software Design & Architecture
Abstraction-Occurrence Pattern

What is Abstraction-Occurrence Pattern?
The Abstraction-Occurrence Pattern is a design pattern used to represent multiple occurrences (instances) of something that share common features — without duplicating data or logic.

It separates:

- Abstraction: The general concept or definition
- Occurrence: The specific, real-world instances of that concept

Purpose:
To avoid redundancy and increase re-usability by separating what is common (abstraction) from what varies (occurrence).

PREPARED BY ALMAS AFZAL

Benefits:

- ✓ No duplication of shared data
- ✓ Easy to create, manage, and track instances
- ✓ Clean separation of general structure vs. specific details

---

Topic_68
Software Design & Architecture
Reflexive Association

What is Reflexive Association?

A reflexive association (also called self-association) is a relationship where a class is associated with itself.

This means an object of a class is related to another object of the same class.

Purpose:

To model hierarchical or peer relationships within the same class.

Benefits:

- ✓ Allows modeling complex relationships within the same object type
- ✓ Useful for representing tree structures (like org charts, file systems)

PREPARED BY ALMAS AFZAL

Topic_69
Software Design & Architecture
Design patterns Introduction

What are Design Patterns?

Design patterns are reusable solutions to common problems that occur repeatedly in software design.

They are not code, but templates or guidelines for how to solve problems effectively and efficiently.

Why Use Design Patterns?

- ✓ **Save time** by using proven solutions
- ✓ **Improve communication** between developers (common vocabulary)
- ✓ **Make code more flexible, reusable, and maintainable**
- ✓ Help **avoid reinventing the wheel**

Where They Come From?

The concept was popularized by the book:

"Design Patterns: Elements of Reusable Object-Oriented Software"
by Gamma, Helm, Johnson, and Vlissides (The Gang of Four - GoF)

Topic_70
Software Design & Architecture
Elements of Design patterns

PREPARED BY ALMAS AFZAL

There are four elements of Design Pattern:

1. Pattern Name
2. Problem
3. Solution
4. Concequence

1. Pattern Name:

A handle used to describe:

- ➢ A design problem
- ➢ Its Solution
- ➢ Its Concequence

Benefits:

- ➢ Increases design vocabulary
- ➢ Makes it possible to design at a higher level of abstraction
- ➢ Enhance commucation

2. Problem:

- Describes when to apply the pattern
- Explain the problem and its context.

3. Solution:

Describes the elements that make up the:

- Design
- Relationships
- Responsibilities
- Collaborations

4. Concequences:

Results and trade-offs of applying the pattern

Critical For:

- Evaluating design alternatives
- Understanding costs
- Understanding benefits of applying the patterns

Topic_71
Software Design & Architecture
Categories of Design patterns

Design Patterns have been divided into three different Categories:

1. Creational Patterns (5)

- Focus on **object creation** mechanisms.
- Make the system **independent of how objects are created, composed, and represented**.

PREPARED BY ALMAS AFZAL

Pattern Name	Purpose
Singleton	Ensure only one instance of a class exists
Factory Method	Create objects without exposing the creation logic
Abstract Factory	Create families of related objects
Builder	Construct complex objects step by step
Prototype	Create new objects by copying an existing one

2. Structural Patterns (8)

- Deal with **how classes and objects are composed** to form larger structures.
- Help ensure that components are **flexibly and efficiently connected**.

Pattern Name	Purpose
Adapter	Allows incompatible interfaces to work together
Decorator	Add behavior to objects dynamically
Composite	Treat a group of objects as a single object
Proxy	Control access to another object
Bridge	Separate abstraction from implementation
Flyweight	Reduce memory usage by sharing objects
Facade	Provide a simplified interface to a complex system

3. Behavioral Patterns (11)

- Focus on **communication between objects**.

PREPARED BY ALI AFZAL

❖ Help assign **responsibility and behavior** clearly.

Pattern Name	Purpose
Observer	Notify objects when state changes
Strategy	Define interchangeable behaviors
Command	Encapsulate a request as an object
State	Change object behavior based on internal state
Iterator	Access elements of a collection sequentially
Mediator	Reduce direct communication between objects
Template Method	Define program steps in a method, allowing overrides
Chain of Responsibility	Pass requests along a chain until handled
Memento	Save and restore object states
Visitor	Separate algorithms from object structures
Interpreter	Interpret language grammar

PREPARED BY ALMAS AFZAL

SO:

They are totally 24 Patterns.

Furthers theses patterns are divided into two pattern types:

- ***Class pattern***
- ***Object pattern***

Remmeber that:

1. ***4 are Class Patterns***
2. ***20 are Object Patterns***

Class Patterns:

- ✧ Deals with relationship b/w classes and their sub-classes
- ✧ Established through inheritance, so they are static-fixed at compile time

Object Patterns:

- ✧ Deal with object relationship

- ✧ Established through Association
- ✧ Can be changes at run time and are more dynamic

Topic_72
Software Design & Architecture
Benefits & Drabacks of Design patterns

PREPARED BY ALMAS AFZAL

Benefits of Design Patterns:

- ❖ Re-usability
- ❖ Standardization
- ❖ Improved Communication
- ❖ Maintainability
- ❖ Best Practices
- ❖ Faster Development
- ❖ Loose Coupling

Drawbacks of Design Patterns

- ❖ Overhead/Complexity
- ❖ Misuse
- ❖ Learning Curve
- ❖ Overengineering
- ❖ Inflexibility

Topic_73
Software Design & Architecture
Singleton Design patterns

What is Singleton Pattern?

The **Singleton pattern** ensures that a **class has only one instance** and provides a **global point of access** to that instance.

Advantages:

- Controlled Access
- Memory efficient
- Consistent state

Disadvantages:

- Global state
- Not easily testable
- Multi-threading issues

Where to Use Singleton Pattern?

- Logging System
- Configuration Managers
- Database Connection
- Catching

Topic_74
Software Design & Architecture
Strategy Design patterns

What is Strategy Pattern?

The **Strategy Pattern** is a behavioral design pattern that **defines a family of algorithms**, encapsulates each one, and makes them **interchangeable**. This pattern lets the algorithm vary independently from clients that use it.

Purpose:

- Allows selecting an algorithm **at runtime**.
- Promotes **flexibility and reusability** of code.
- Eliminates **if-else** or **switch** statements for algorithm selection.

PREPARED BY ALMAS AFZAL

Advantages:

- Open/Closed Principle
- Cleaner code
- Reusability

Disadvantages:

- Complexity
- Indirection

Topic_75
Software Design & Architecture
Design patterns-Introduction

Repeated!
So you can skip it easily :)

Topic_76
Software Design & Architecture
Facade Design patterns

PREPARED BY ALMAS AFZAL

What is Facade Pattern?

The Facade Pattern is a structural design pattern that provides a simple interface to a complex subsystem. It hides the complexity of the system by providing a unified and easy-to-use interface.

Purpose:

- ✓ To make a system easier to use.
- ✓ To hide complexities of the underlying code.

✓ To reduce dependencies between clients and subsystems.

MIDTERM SYLLABUS COMPLETED!

BEST OF LUCK FOR YOUR EXAMS

ALLAH HAFIZ!

PREPARED BY ALMAS AFZAL