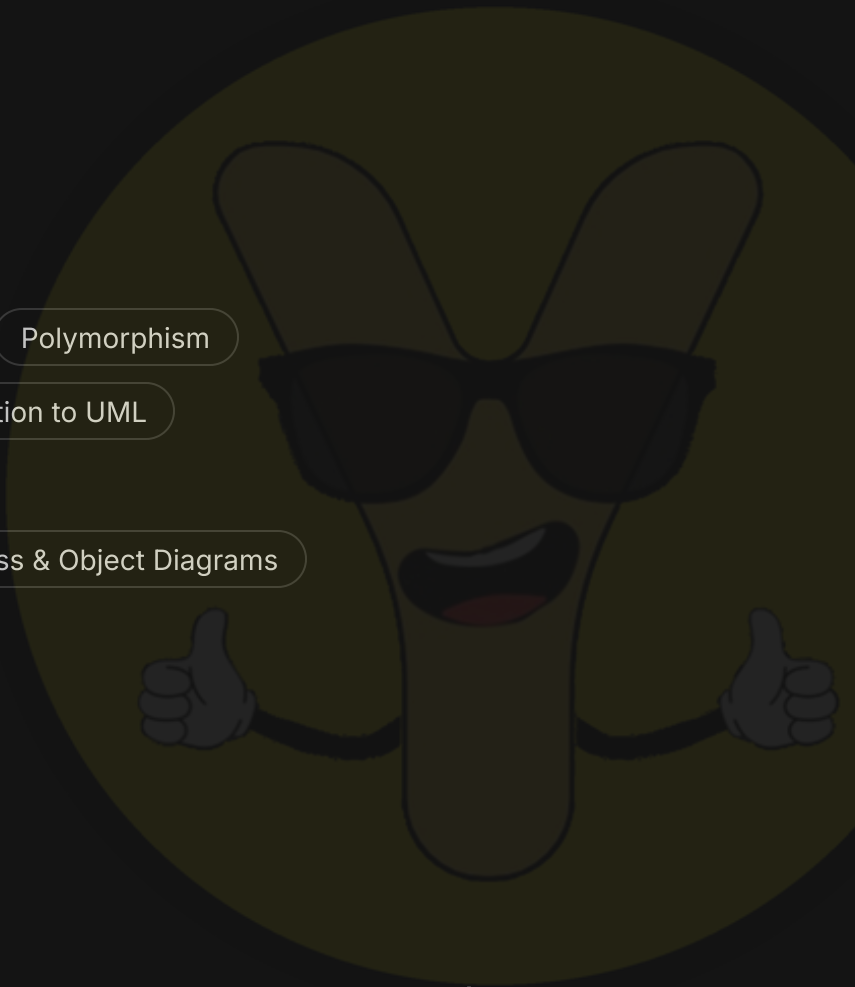
AKTU · 3RD YEAR · SEMESTER 5 · BCS054

Object Oriented System Design with C++

Unit 1 — Introduction and Basic Structural Modeling

- Introduction: The meaning of Object Orientation
- Object identity
- Encapsulation
- Information hiding
- Polymorphism
- Generosity
- Importance of modelling
- Principles of modelling
- Object-oriented modelling
- Introduction to UML
- Conceptual model of the UML
- Architecture
- Basic Structural Modeling: Classes
- Relationships
- Common Mechanisms and diagrams
- Class & Object Diagrams: Terms, concepts, modelling techniques for Class & Object Diagrams



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Introduction to Object Orientation

OBJECT ORIENTATION (OO)

Object Orientation is a software development paradigm that models real-world entities as software objects containing both data state (attributes) and behavior (methods).

- Organizes software design around data and objects rather than procedural logic.
- Core pillars include Abstraction, Encapsulation, Inheritance, and Polymorphism.
- Improves software modularity, reusability, maintainability, and scalability.

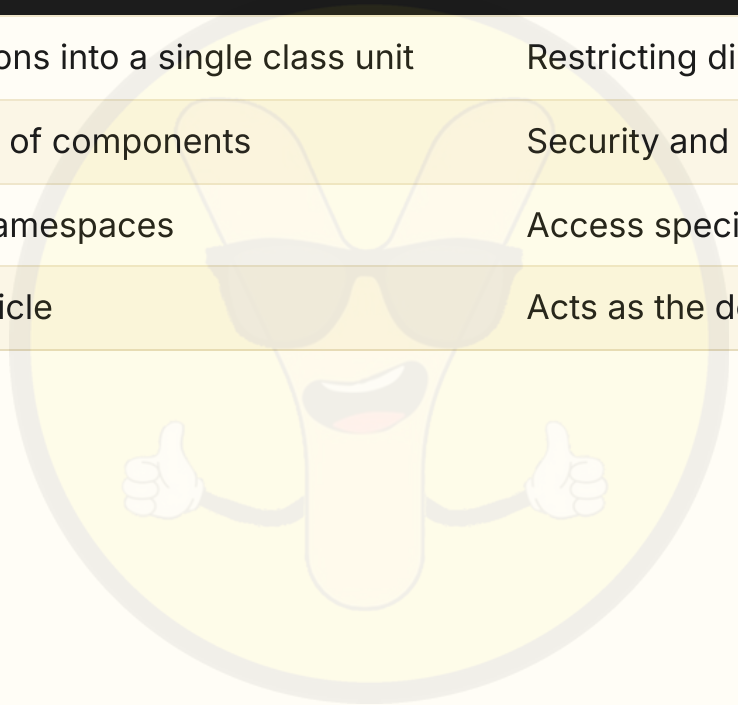
Object Identity, Encapsulation, & Hiding

Fundamental concepts governing object state, structure, and accessibility.

- Object Identity
  - Unique property of an object distinguishing it from all other objects.
  - Independent of attribute values; represented via memory addresses or unique IDs.
- Encapsulation
  - Bundling data attributes and methods operating on that data into a single unit.
  - Prevents external code from direct unauthorized access to internal state.
- Information Hiding
  - Hiding internal implementation details behind a clear public interface.
  - Achieved in C++ using access specifiers: private, protected, and public.

Encapsulation vs Information Hiding

Feature	Encapsulation	Information Hiding
Definition	Packaging data and functions into a single class unit	Restricting direct access to internal components
Primary Focus	Structure and organization of components	Security and decoupling by hiding implementation
Mechanism	Classes, structures, and namespaces	Access specifiers (private, protected)
Relationship	Acts as the packaging vehicle	Acts as the design principle achieved via encapsulation



Polymorphism and Generosity

Advanced object-oriented mechanisms for flexibility and code reusability.

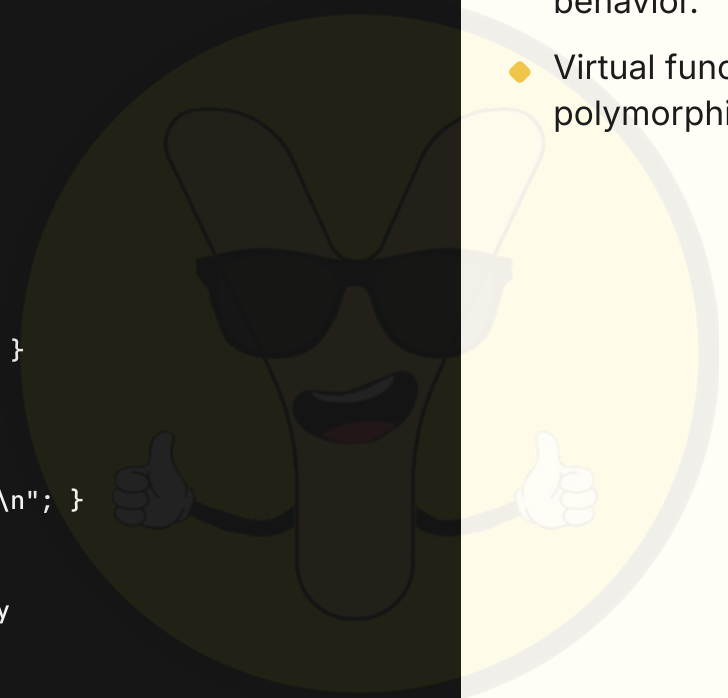
- Polymorphism
  - Ability of a single interface to represent different underlying forms.
  - Compile-Time: Function and operator overloading.
  - Run-Time: Virtual functions and method overriding using dynamic dispatch.
- Generosity (Genericity)
  - Ability to define parameterized software components (templates).
  - Allows algorithms and data structures to operate on any data type without code duplication.
  - Implemented in C++ using Class and Function Templates.

C++ Polymorphism and Genericity

```cpp
#include <iostream>
using namespace std;

// Genericity via Template
template <typename T>
T add(T a, T b) { return a + b; }

// Dynamic Polymorphism via Virtual Function
class Base {
public:
    virtual void show() { cout << "Base Class\n"; }
};
class Derived : public Base {
public:
    void show() override { cout << "Derived Class\n"; }
};
int main() {
    cout << add<int>(5, 10) << endl; // Genericity
    Base* ptr = new Derived();
    ptr->show(); // Polymorphism
    delete ptr;
    return 0;
}
```

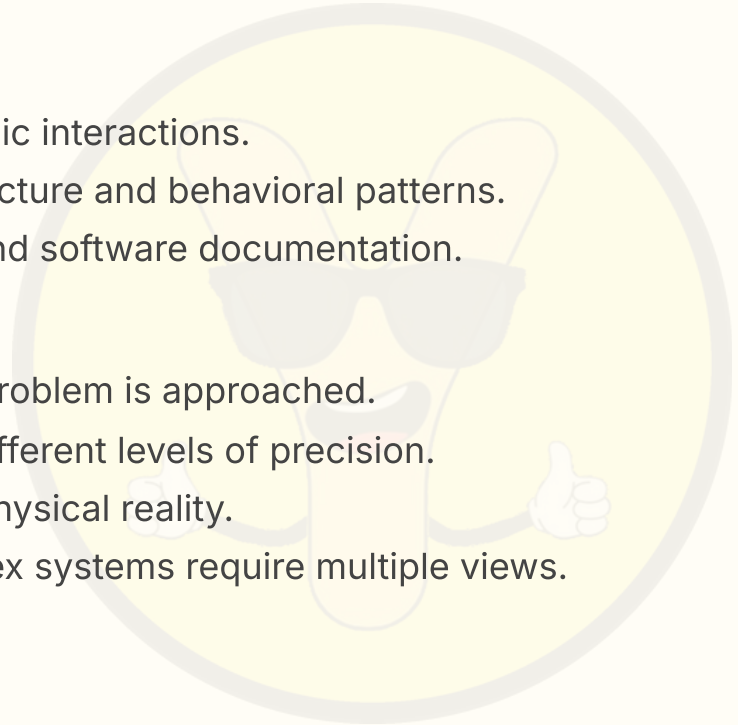


- Template function demonstrates type-independent generic behavior.
- Virtual function `show()` enables dynamic runtime polymorphism via vtable.

Importance & Principles of Modeling

Modeling is the construction of abstractions before building complex systems.

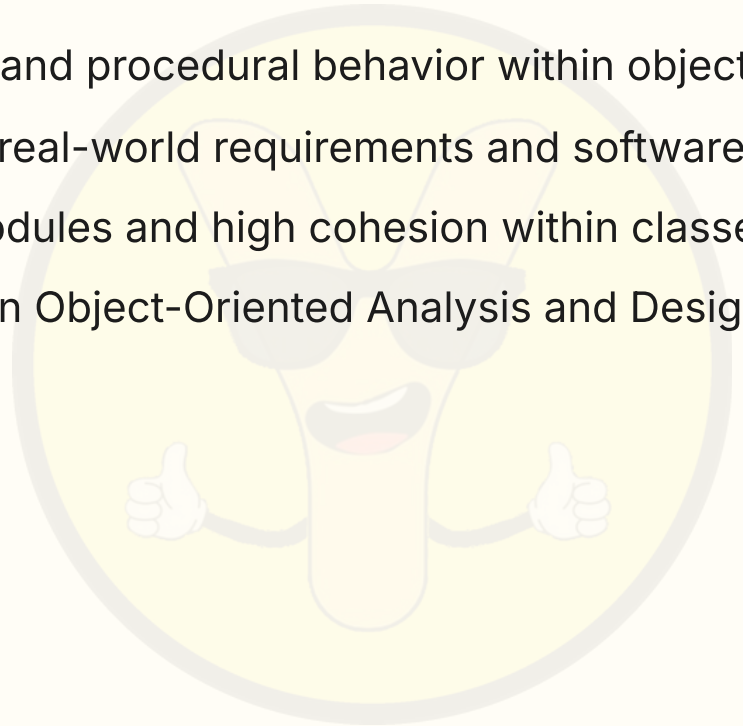
- Importance of Modeling
  - Visualizes system structure and dynamic interactions.
  - Permits specification of system architecture and behavioral patterns.
  - Provides a blueprint for construction and software documentation.
- Four Principles of Modeling
  - 1. Choice of model determines how a problem is approached.
  - 2. Every model may be expressed at different levels of precision.
  - 3. The best models are connected to physical reality.
  - 4. No single model is sufficient; complex systems require multiple views.



Object-Oriented Modeling (OOM)

A design methodology modeling systems as interacting object abstractions.

- Combines dynamic data structures and procedural behavior within object models.
- Bridges the semantic gap between real-world requirements and software code.
- Promotes low coupling between modules and high cohesion within classes.
- Serves as the foundation for modern Object-Oriented Analysis and Design (OOAD).



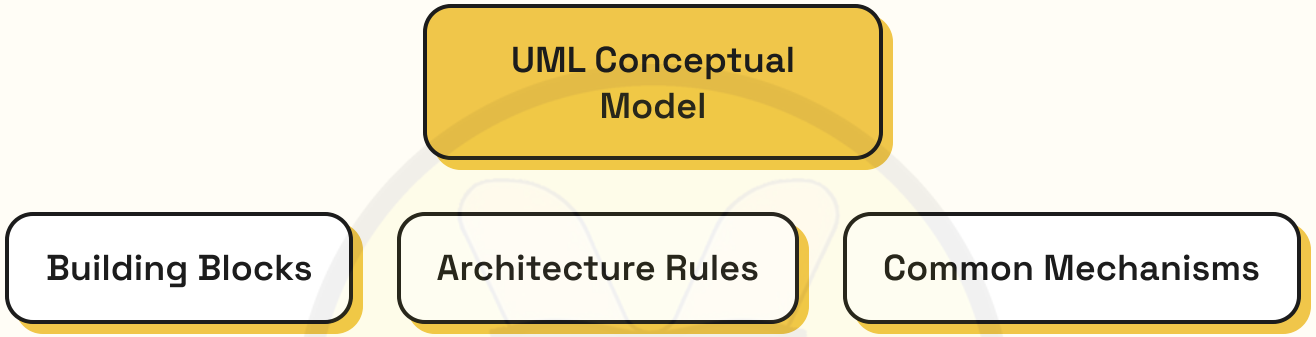
Introduction to UML

UNIFIED MODELING LANGUAGE (UML)

UML is a standardized visual modeling language used to specify, visualize, construct, and document the artifacts of software-intensive systems.

- Standardized by the Object Management Group (OMG) and ISO.
- Created by Grady Booch, James Rumbaugh, and Ivar Jacobson (The Three Amigos).
- Language-independent blueprinting standard for software architectures.

Conceptual Model of the UML

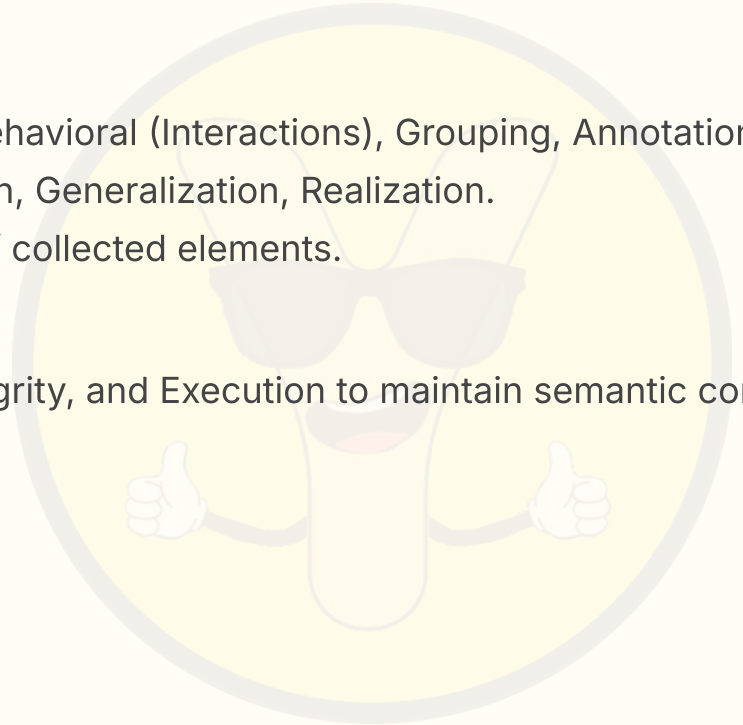


The three core pillars forming the conceptual foundation of UML.

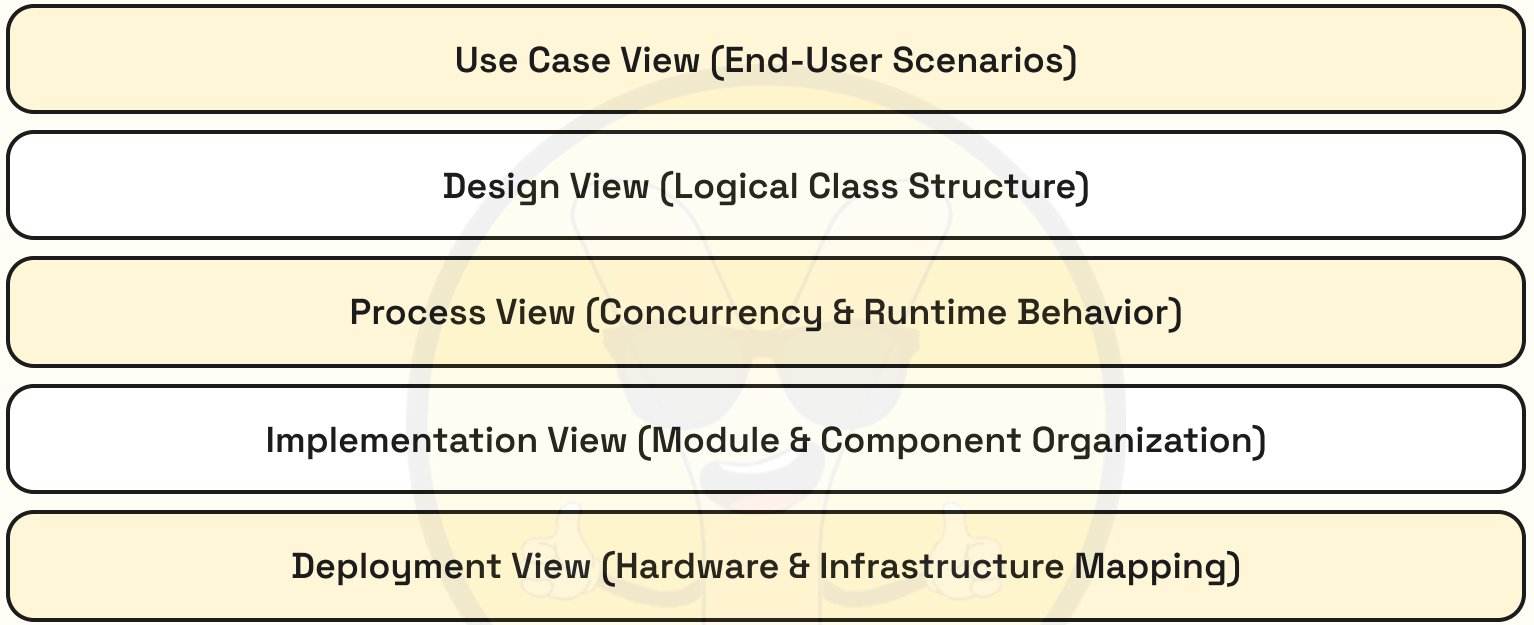
UML Building Blocks and Rules

Foundational components used to construct complete visual models.

- Building Blocks
  - Things: Structural (Classes, Nodes), Behavioral (Interactions), Grouping, Annotational.
  - Relationships: Dependency, Association, Generalization, Realization.
  - Diagrams: Graphical representations of collected elements.
- Architecture Rules
  - Rules for Names, Scope, Visibility, Integrity, and Execution to maintain semantic consistency.



Software Architecture: 4+1 View Model



Kruchten's 4+1 View Model describing software system architecture.

Basic Structural Modeling: Classes

Classes are the primary structural building blocks in object-oriented systems.

- Class Notation
  - Rendered as a rectangle with 3 compartments: Name, Attributes, and Operations.
- Attribute Syntax
  - `[visibility] name : type [= initial-value]`
  - Example: `- balance : double = 0.0`
- Operation Syntax
  - `[visibility] name(parameter-list) : return-type`
  - Example: `+ deposit(amount : double) : bool`
- Visibility Symbols
  - `+` Public, `-` Private, `#` Protected, `~` Package.

Relationships in UML

Connectors defining structural and behavioral dependencies between elements.

- Dependency (`⟶`)
  - Semantic relationship where a change in target affects the source element.
- Association (`--`)
  - Structural connection specifying objects are linked to other objects.
- Aggregation (`<>--`)
  - Asymmetrical 'has-a' relationship with weak whole-part lifetime coupling.
- Composition (`<#>--`)
  - Strong whole-part relationship where parts cannot exist without the whole.
- Generalization (`--▷`)
  - 'Is-a' relationship connecting a sub-element to a super-element.

Association vs Aggregation vs Composition

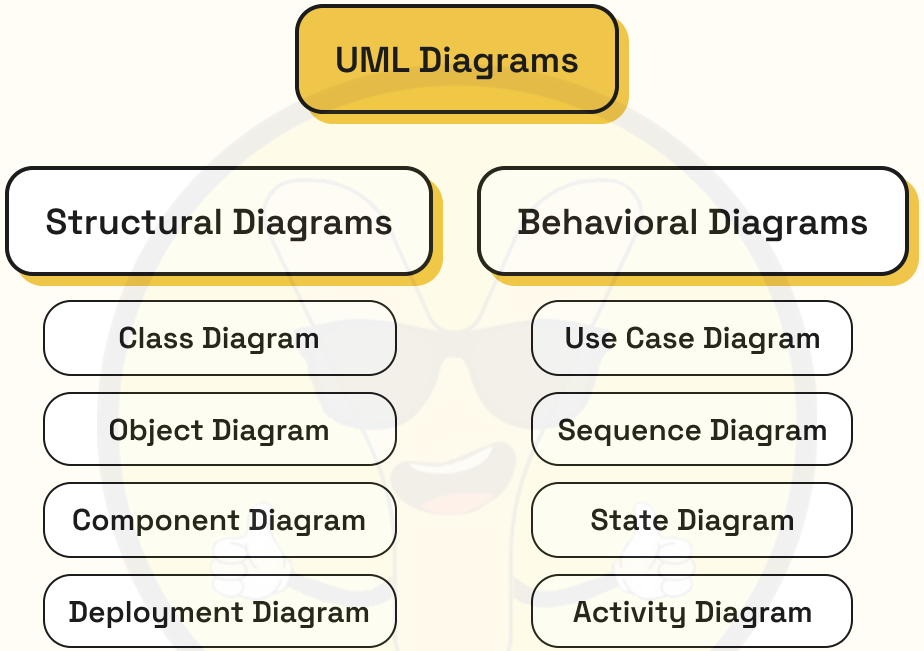
Property	Association	Aggregation	Composition
Relationship Type	Peer-to-Peer link	Weak Whole-Part	Strong Whole-Part
Ownership	No ownership	Shared ownership	Exclusive single owner
Lifetime Coupling	Independent	Part survives Whole	Part destroyed with Whole
UML Notation	Solid Line	Hollow Diamond	Filled Diamond
Code Example	Pointer reference	Member pointer/ref	Direct member variable

Common Mechanisms in UML

Four universal mechanisms applied consistently across all UML diagrams.

- Specifications
  - Textual descriptions defining exact syntax and semantics of graphical elements.
- Adornments
  - Visual details added to items (e.g., multiplicity, visibility, role names).
- Common Distinctions
  - Clear separation between Type (Class) vs Instance (Object), and Interface vs Implementation.
- Extensibility Mechanisms
  - Stereotypes (`<<interface>>`), Tagged Values (`{author="AKTU"}`), and Constraints (`{ordered}`).

Classification of UML Diagrams



Hierarchical structural and behavioral breakdown of UML diagram types.

Class Diagrams: Concepts & Techniques

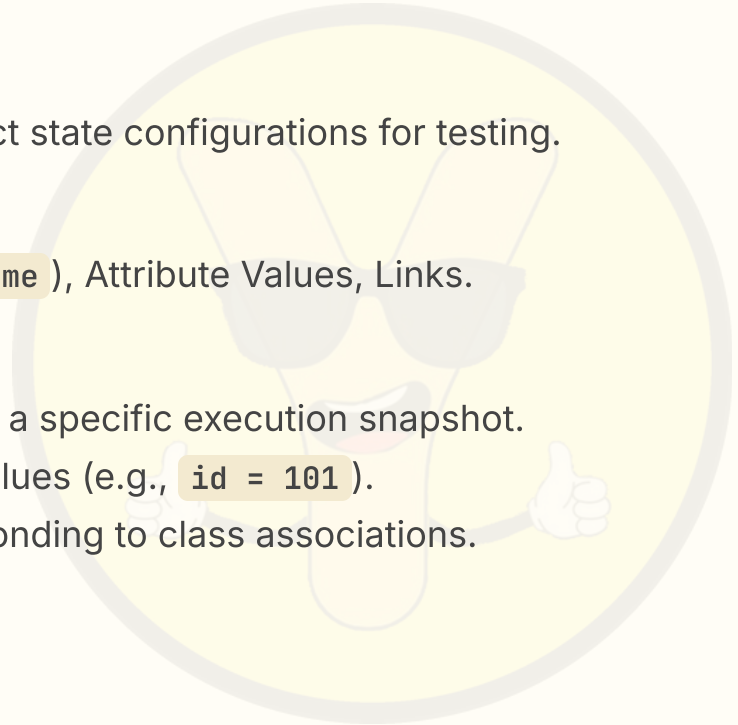
Static structural diagrams showing system classes, attributes, and relationships.

- Key Purpose
  - Provides static view of software architecture for analysis and code generation.
- Modeling Technique
  - 1. Identify key domain classes from system specifications.
  - 2. Define class attributes, operations, and visibility specifiers.
  - 3. Identify relationships (generalization, association, dependency).
  - 4. Specify multiplicities (`0..1`, `1..*`) and role names on associations.

Object Diagrams: Concepts & Techniques

Snapshot of instantiated class structures at a specific moment in runtime.

- Key Purpose
  - Models static data structures and object state configurations for testing.
- Core Concepts
  - Object Name (`instanceName : ClassName`), Attribute Values, Links.
- Modeling Technique
  - 1. Identify instances required to explain a specific execution snapshot.
  - 2. Populate slots with actual runtime values (e.g., `id = 101`).
  - 3. Connect objects using links corresponding to class associations.



Class Diagram vs Object Diagram

Feature	Class Diagram	Object Diagram
Abstraction Level	Abstract template / Blueprint	Concrete execution snapshot
Time Dimension	Static across entire system life	Specific point in execution time
Key Elements	Classes, Interfaces, Associations	Objects, Attribute values, Links
Multiplicity	Explicitly defined (e.g., 1..*)	Not applicable (single instances)
Name Notation	Plain class name (`Customer`)	Underlined object name (`c1:Customer`)
Primary Purpose	Architectural design & coding	Debugging & prototype validation

Modeling Technique: Library System Class Diagram

Design a class diagram for a Library System with Books, Readers, and Librarians.

1. 1. Identify entities: Book, User, Librarian, Member.
2. 2. Apply Generalization: Librarian and Member inherit from abstract superclass User.
3. 3. Define Attributes: Book (title: String, isbn: String), User (id: int, name: String).
4. 4. Define Relationships: Member borrows Book (Association 0..* to 0..*); Librarian manages Book (1 to *).
5. 5. Define Aggregation: Library contains Books (Aggregation whole-part relationship).

A Class Diagram featuring superclass User, derived classes Member and Librarian, connected via Association and Aggregation relationships to Book.

Exam Corner: High-Yield AKTU Questions

- **Q1** Explain the fundamental concepts of Object Orientation: Encapsulation, Information Hiding, and Polymorphism with suitable C++ examples.
- **Q2** Differentiate between Aggregation, Composition, and Association with real-world examples and UML symbols.
- **Q3** Describe the Conceptual Model of UML detailing its building blocks, architecture rules, and common mechanisms.
- **Q4** Explain Kruchten's 4+1 Architectural View Model and its significance in software modeling.
- **Q5** Compare Class Diagrams and Object Diagrams. Explain the step-by-step modeling techniques for designing Class Diagrams.
- **Q6** Write short notes on UML extensibility mechanisms: Stereotypes, Tagged Values, and Constraints.
- **Q7** Define Genericity (Generosity). Explain how compile-time genericity is implemented in C++ using templates.

Unit 1 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 2 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

