1. Introduction to UML

- **What is a Model in Software Engineering?**
    - **A high-level abstraction of a system to aid in development.**
    - **Often represented as graphs or flowcharts for better visualization.**
- **Why Use UML?**
    - **Reduces errors early in development.**
    - **Finding issues in requirements & design is cost-effective.**
    - **Standardized visual language for system documentation.**
- **What is UML?**
    - **A standardized modeling language for software systems.**
    - **Helps developers specify, visualize, construct, and document software.**
    - **UML diagrams can be converted into code manually or with tools.**

---

2. Pros and Cons of UML

✅ Pros

- **Increased Productivity – Helps visualize and structure software effectively.**
- **Better Design & Consistency – Encourages structured and modular design.**
- **Improved Communication – Common language between developers, designers, and stakeholders.**
- **Early Issue Detection – Helps identify issues before coding.**
- **Clear Specification – Test case generation and traceability between design and requirements.**
- **Tool Support – UML has commercial tools that integrate into development workflows.**

❌ Cons

- **Code Optimization Issues – UML-generated code may not be optimized for performance.**
- **Learning Curve – Developers need time to learn UML modeling tools.**
- **High Maintenance Overhead – Keeping UML diagrams in sync with code is challenging.**
- **Not Always Precise – UML is an abstraction, meaning low-level system details may be missing.**
- **Scalability Issues – Large UML diagrams become hard to interpret.**

```
// A simple class implementation via UML
+-----------------------------------------------+
|   Rectangle                                   |
+-----------------------------------------------+
| - width : double                              |
| - height: double                              |
+-----------------------------------------------+
| + Rectangle(w: double, h: double)             |
| + getWidth() : double                         |
| + getHeight() : double                        |
| + getArea() : double                          |
+-----------------------------------------------+

class Rectangle {
private:
    double width;
    double height;

public:
    // Constructor
    Rectangle(double w, double h) : width(w), height(h) {}

    // Getter for width
    double getWidth() const {
        return width;
    }

    // Getter for height
    double getHeight() const {
        return height;
    }
```

```
    double getArea() const {
        return width * height;
    }
};
```

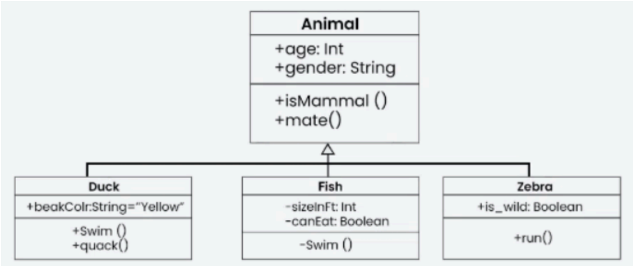
UML models the flow of actions in the “Place Order”

```
      [Start]
         |
         v
+---------------------+
| Customer Places Order |
+---------------------+
         |
         v
+---------------------+
| System Checks Stock |
+---------------------+
         |
         v
+-----------------------+
| [All Items In Stock?] |
+-----------------------+
        /              \
      Yes               No
       |                 \
       v                  v
+----------------------+   +----------------------+
| Confirm Order        |   | Reject Order         |
+----------------------+   +----------------------+
       |                          |
       v                          v
+-----------------------+   +---------------------------+
| Update Stock          |   | Send Rejection Message    |
+-----------------------+   +---------------------------+
   |                                 |
   v                                 v
+---------------------------+      [End]
| Send Confirmation Message|
+---------------------------+
   |
   v
[End]
```

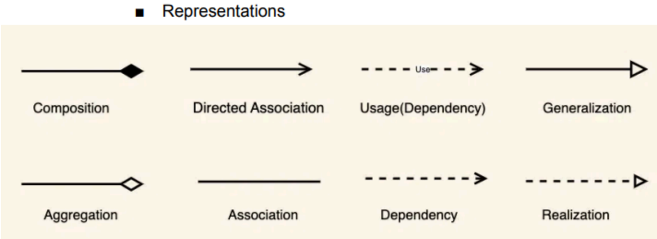
3. Types of UML Diagrams

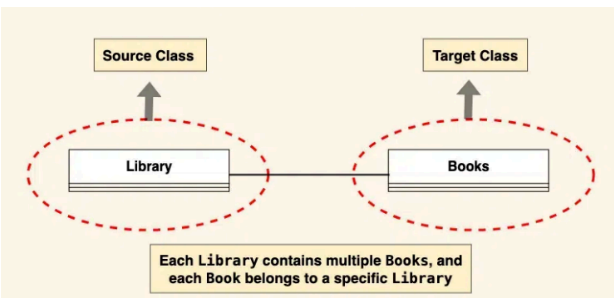
Structural Diagrams (Static)

Class Diagrams – Show classes, attributes, methods, and relationships.

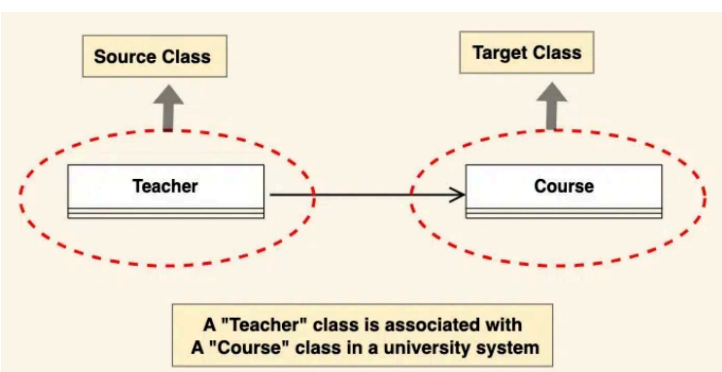


- Intra-class notion
  - Class Name
  - Attributes: data members
  - Methods
    - can add input/output parameters
  - Visibility Notation
    - + public
    - - private
    - # protected
    - ~ visible to classes in the same package/library
- Inter-class relationship
  - Representations

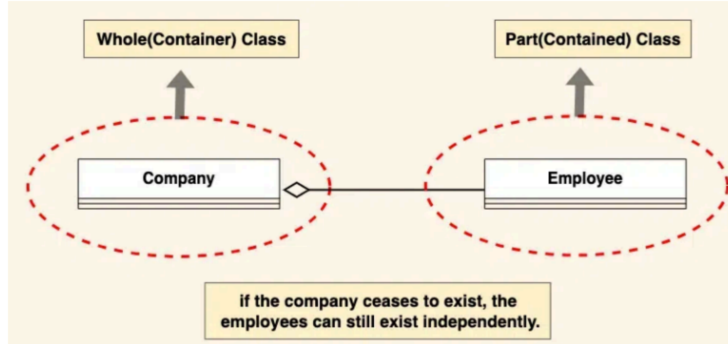




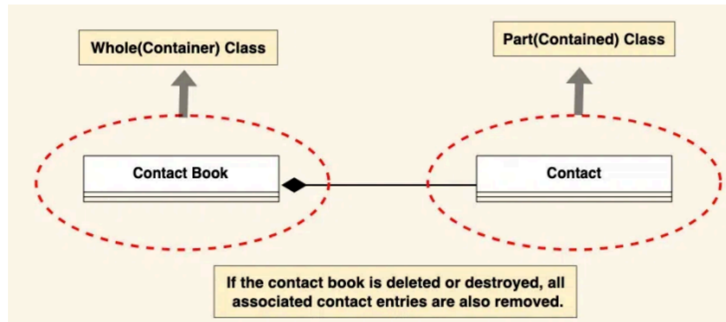
- Association
  - bi-directional relationship between two classes
  - instances of one class are connected to instances of another class
  - class like Library below is a folded representation. i.e. hiding non-class name notions.



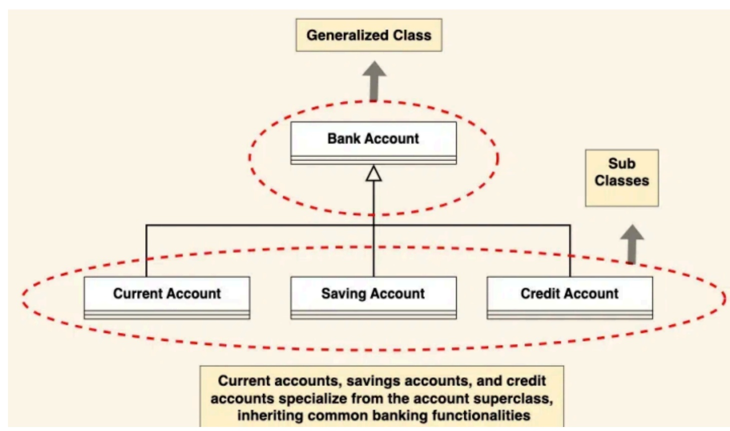
- Directed association
  - one class is associated with another in a specific way/direction.
  - the arrow points from the class that initiates the association
  - the arrow points to the class affected by the association.
  - Example. I teach specific course classes such as CS35L & CS 130.



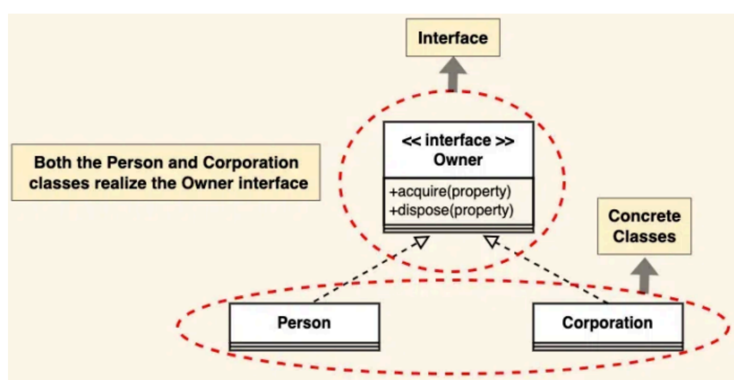
- Aggregation
  - a stronger relationship where one class (the whole) contains or is composed of another class (the part).



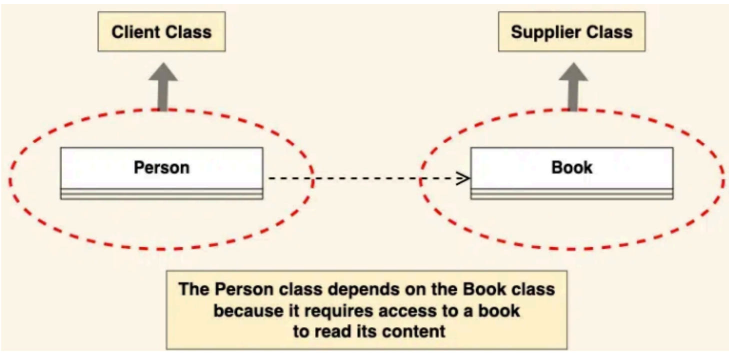
- Composition
  - stronger form of aggregation
  - the part class cannot exist independently of the whole class



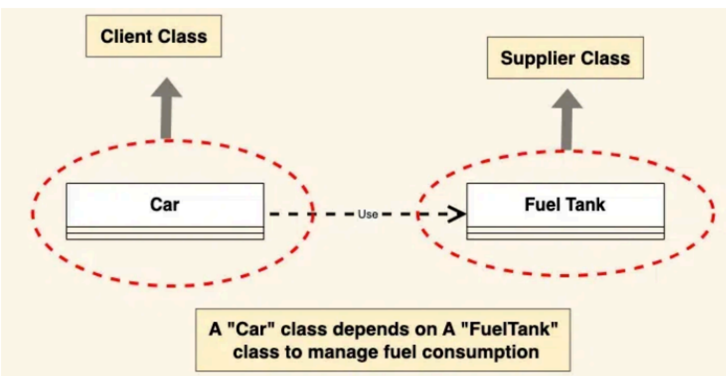
- Inheritance/Generation (Frequently Use)
  - one class (the subclass or child) inherits the properties and behaviors of another class (the generalized class or parent).
  - is-a relation
  - inherits can be recursive/multiple layers



- Realization
    - a class implements the features of an interface.
    - can-do relation
    - Owner Interface: This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
    - Person Class (Realization): a person can acquire ownership of a house or dispose of a car.
    - Corporation Class (Realization): a corporation can acquire ownership of real estate properties or dispose of company vehicles.

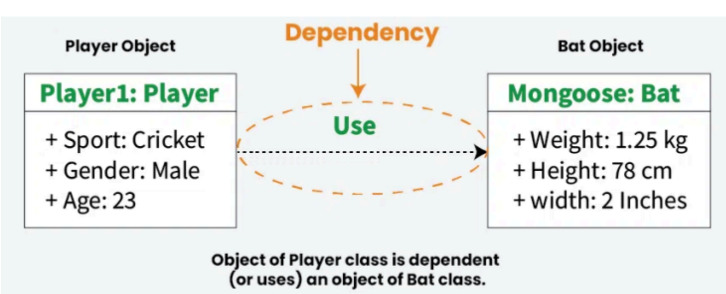


- Dependency
    - one class relies on another
    - not as strong as association or inheritance



- Usage
    - one class (the client) utilizes or depends on another class (the supplier) to perform certain tasks or access certain functionality

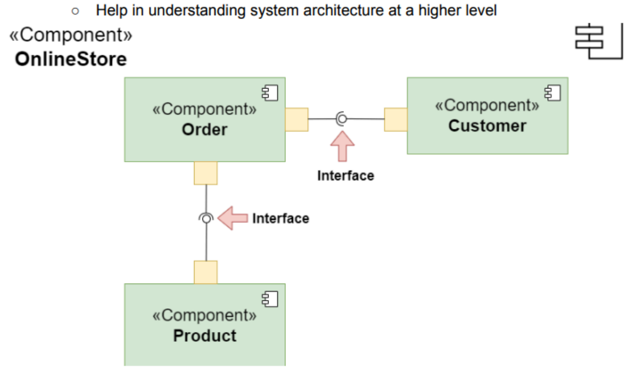
Object Diagrams – Represent real instances of a system at a specific time.

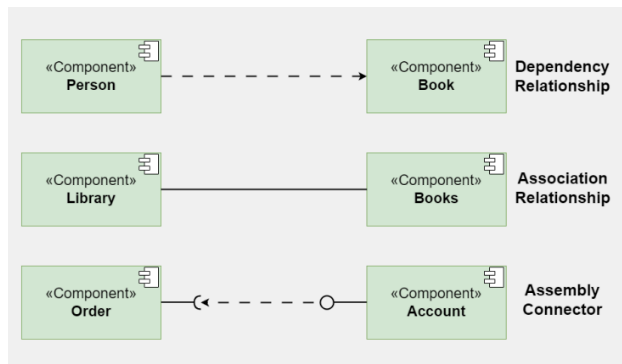


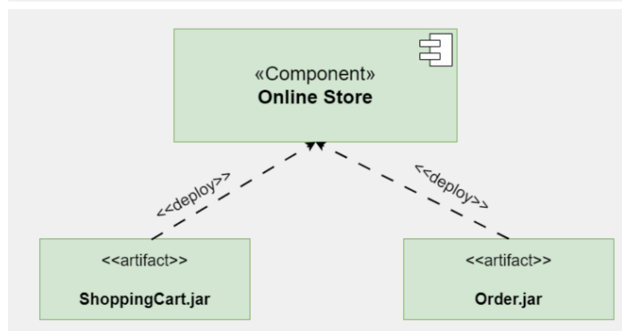
- Object Diagrams
    - Provide snapshots of **instances** of class diagrams at a specific point in time.
    - a detailed view of how objects interact with each other in specific scenarios
    - promoting a shared understanding of specific instances and their relationships.
    - relationships in class diagrams apply.

Component Diagrams – Illustrate system components and dependencies.

- Component Diagrams
    - Illustrate how different components of a system are wired together
    - A set of diagrams talking about components, interfaces, relationships, ports, artifacts, nodes. etc.
    - Focus on the physical aspects of an object-oriented system
    - Show dependencies between different software components
    - Help in understanding system architecture at a higher level

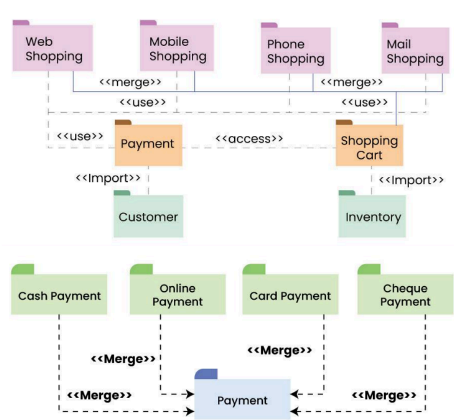






Package Diagrams – Show modular organization of a system.

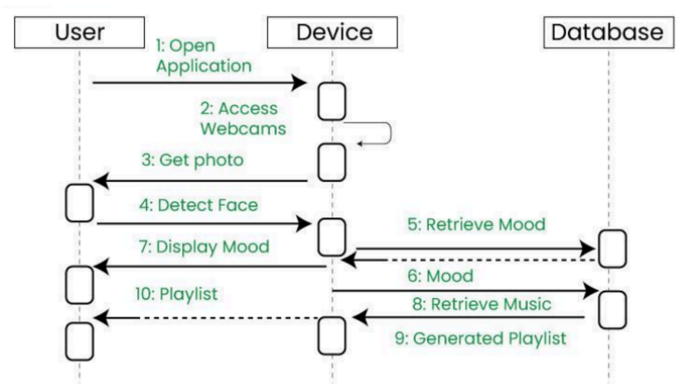
- Package Diagrams: Demonstrate the organization and layering of various elements in a system
    - Used to show dependencies between different packages
    - Help in managing large-scale system architecture
    - Essential for maintaining system modularity



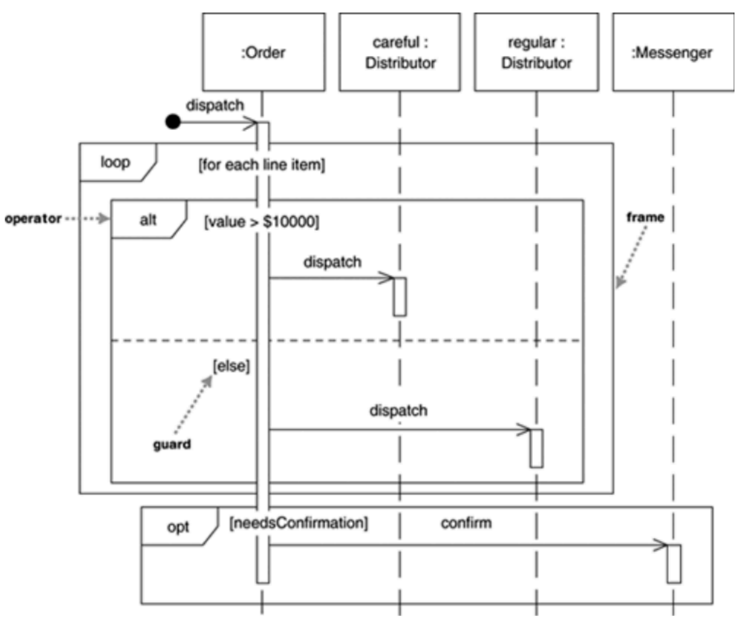
Behavioral Diagrams (Dynamic)

Sequence Diagrams – Show object interactions over time.

- Sequence Diagrams (Used often in design)
  - Illustrate interactions between objects in sequential order
  - Show the chronological flow of messages between objects
  - Essential for understanding complex method calls and system behavior
  - Communication between objects is depicted using messages.
    - Lifeline elements are individual participants in a sequence diagram.
      - located at the top in a sequence diagram
    - represent messages using arrows.
    - lifelines and messages form the core of a sequence diagram.
    - little rectangle among dashed lines: invocation lifetime

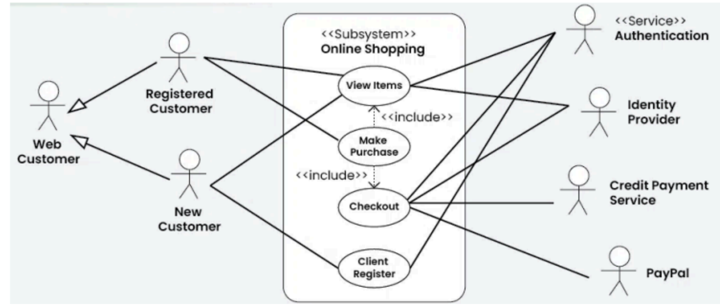


  - Some fancier sequence diagrams with found message and loop/alt/opt frames
    - loop: a loop
    - alt: multiple alternatives of which exactly one will be executed
    - opt: something or nothing happens
    - We may ask you in the exam with essential background provided.



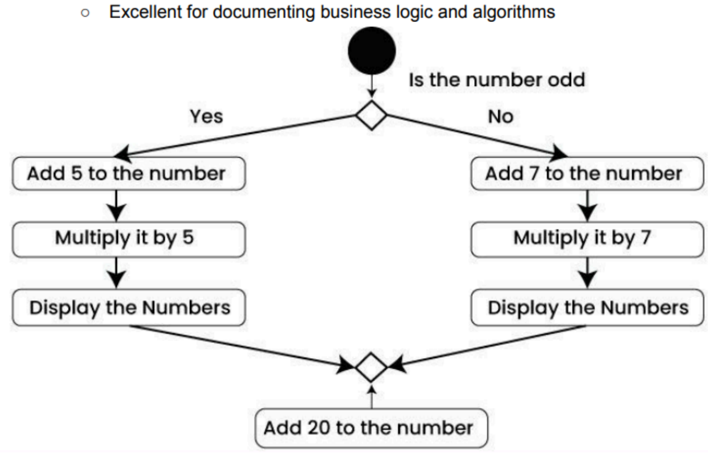
Use Case Diagrams – Represent system interactions with users or other systems.

- Usecase Diagrams
  - interaction between different actors/parties
  - visualize how each party interact with the system
  - non-tech stakeholders like them
  - Frequent used relationships
    - association / include / extend / generalization
    - no need to bounded by above, as long as all parties on the same page



Activity Diagrams – Model workflow and business logic (similar to flowcharts).

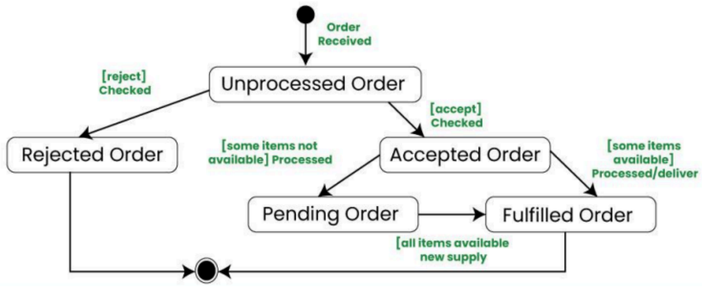
- Activity Diagrams
  - Model workflows and business processes
  - Similar to traditional flowcharts but with additional capabilities
  - Support parallel processing and complex decision paths
  - Excellent for documenting business logic and algorithms



State Machine Diagrams – Describe object states and transitions.

- State Machine Diagrams
  - Show different states an object can be in and transitions between states
  - Document the lifecycle of objects
  - Essential for understanding complex state-dependent systems
  - Useful for modeling reactive systems
  - Also very useful in verification (if taken automata course)
  - Components
    - state
      - a condition of a modeled entity for which some action is performed, some stimulus is received, or some condition is met elsewhere in the system
    - action
      - an atomic execution
      - Atomic means it completes without interruption
    - activity
      - a more complex collection of behavior that may run for a long duration
    - transition between two states
      - an arc from one state to another
      - transitions can have triggers, guard conditions, and actions

- transitions can be labeled with the event or action that creates the entity
  - E.g., trigger [guard] / effect
- initial state: a solid black circle
- end state: a circle wrapping a solid black circle



---

4. UML in Software Verification

- **UML can help in creating test cases for various scenarios:**
  - ✅ **Successful transactions.**
  - ❌ **Errors like out-of-stock items.**
  - 🔄 **Edge cases (e.g., invalid inputs, empty orders).**

✅ Pros of UML in Testing

- **Clear Specifications – Ensures unambiguous system behavior.**
- **Automated Test Case Generation – Helps create comprehensive test cases.**
- **Early Bug Detection – Identifies issues before development progresses.**
- **Traceability – Links requirements, design, and verification.**

❌ Cons of UML in Testing

- **Cannot guarantee a bug-free system.**
- **High abstraction – UML diagrams may omit important low-level details.**
- **Model Maintenance Required – Keeping UML models aligned with code is essential**

5. UML Example: Order Processing Flow

Example: UML Activity Diagram for an Order System
Customer Places Order → `System Checks Stock`
If Items Available → `Confirm Order` → `Update Stock` → `Send Confirmation`
If Items Not Available → `Reject Order` → `Send Rejection Message`
This flow ensures clear process visualization for software design.

Diagram Type	Purpose	Key Elements	Key Differences
Class Diagram	Models the **static structure** of a system	**Classes, attributes, methods, relationships (inheritance, association, composition, etc.)**	- **Most commonly used UML diagram** in OOP design. - Defines how objects interact **at the type level**. - Helps in designing **architecture and object-oriented systems**.
Object Diagram	Represents **specific instances** of a class diagram at a **particular point in time**	**Objects (instances of classes), attribute values, links between objects**	- **Snapshot of runtime objects**. - Useful for **debugging and testing**. - Helps **validate class diagrams by showing real-world examples**.
Component Diagram	Shows **physical components** of a system and their dependencies	**Components (modules, libraries, APIs), interfaces, dependencies**	- Focuses on **modularity and reusability**. - **Used in system-level design** (e.g., how microservices communicate). - Often used with **deployment diagrams**.
Package Diagram	Organizes large systems into **manageable modules**	**Packages (grouped elements), dependencies**	- Helps in **managing complexity**. - Useful for **large software projects**. - Similar to **component diagrams** but focuses on **logical structure** instead of physical.
Sequence Diagram	Represents **step-by-step interactions between objects** over time	**Actors, objects, messages (method calls), lifelines, activation bars**	- Shows **chronological order of events**. - Essential for **understanding method calls and communication**. - Used for **use case scenarios**.
Use Case Diagram	Describes how **users (actors) interact** with the system	**Actors, use cases, associations, relationships (include, extend)**	- High-level view of **what the system does**, not how. - Used in **requirement gathering and stakeholder communication**. - Different from **sequence diagrams**, which focus on **event flow**.
Activity Diagram	Models **workflows and business logic**	**Actions, decisions, forks, joins, start/end nodes**	- Similar to **flowcharts** but more advanced. - Supports **parallel processes** (forks/joins). - Useful for **process modeling**.
State Machine Diagram	Represents **different states** of an object and transitions between them	**States, transitions, events, guards, initial/final states**	- Used for **objects with distinct lifecycle states**. - Essential for **modeling reactive systems** (e.g., ATM, traffic lights). - Different from **activity diagrams**, which model **processes** rather than **states**.

Key Differences Between Sequence and Activity Diagrams

Aspect	Sequence Diagram	Activity Diagram
Focus	**Interaction between objects** over time	**Process flow of activities**
Represents	**Message exchanges** between system components	**Workflow steps & conditions**
Best for	**System interactions, API calls, method execution order**	**Business logic, user flows, decision-making**
Key Elements	Actors, Objects, Messages (arrows), Lifelines	Actions, Decisions (diamonds), Forks/Joins, Start/End
Parallel Execution	Not explicit	Supports **parallel processing** (fork/join)
Time Consideration	Explicit (left-to-right time flow)	Less focus on exact time