Object Oriented Analysis and Design

Deepika Kamboj
Assistant professor
School of Computer Science
UPES

Object-Oriented Analysis and Design (OOAD) is a structured software engineering approach that models real-world systems using **objects**, their **attributes**, **behaviors**, and **interactions**.

Key Concepts:

- Models systems based on real-world concepts
- Breaks down complex systems into manageable components
- Improves scalability, maintainability, and development efficiency

Benefits:

- Mirrors real-world behavior
- Enhances modularity and reusability
- Bridges the gap between requirements and implementation

Analysis and Design in OOAD

OOAD consists of two key phases:

1. **Analysis Phase**
   - Understands the **problem domain**
   - Identifies system **requirements**
   - Models what the system should do — without deciding how to do it
   - Focuses on user needs and system functionality

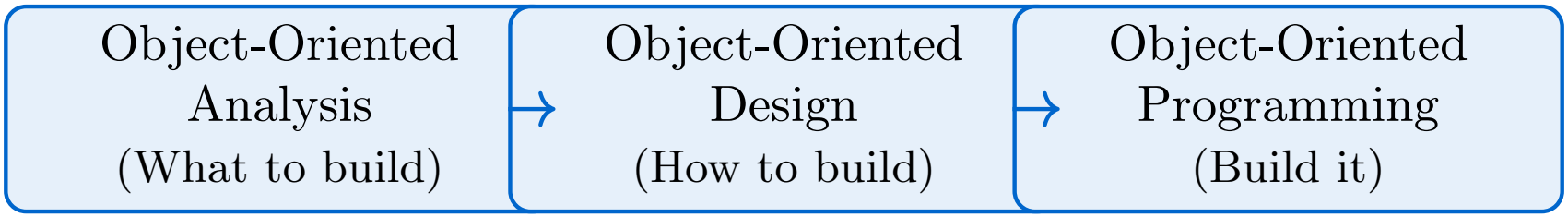
2. **Design Phase**
   - Converts analysis into a technical blueprint
   - Defines **software architecture**, components, and layers
   - Selects appropriate **data structures** and **algorithms**
   - Designs object structures and interactions for implementation

Overview of Object-Oriented System Development

- Object-Oriented System Development (OOSD) is a method of developing software by modeling it as a group of cooperating objects.
- These objects are instances of classes, which bundle both data and behavior.
- OOSD is aligned with real-world concepts, improving modularity, reusability, and maintainability.
- It follows a structured lifecycle: Analysis, Design, and Implementation.

Overview of Object-Oriented System Development

Object-Oriented System Development follows a structured lifecycle consisting of three key stages:

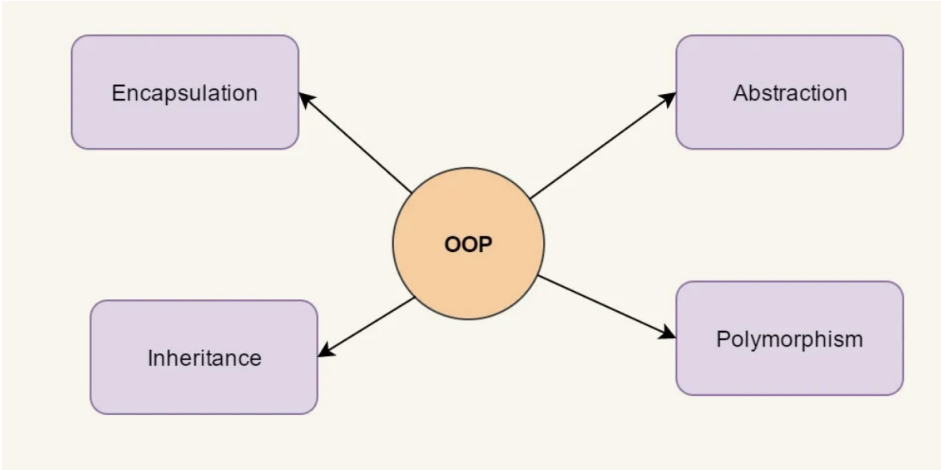


Each phase transitions smoothly into the next, supporting iterative refinement and modular growth.

Overview of Object-Oriented System Development

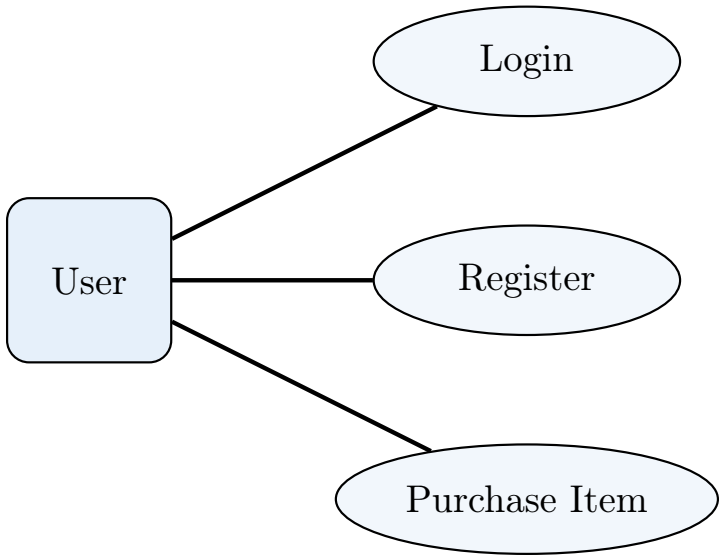
OOSD is based on four key principles:

- **Encapsulation**: Bundling data with methods.
- **Abstraction**: Hiding internal details, exposing only what's necessary.
- **Inheritance**: Mechanism to create new classes from existing ones.
- **Polymorphism**: One interface, multiple implementations.



Overview of Object-Oriented System Development

- Focus on “what” the system should do.
- Identify:
  - Key objects and classes
  - Their attributes and behaviors
  - Relationships among them
- Tools used:
  - Use case diagrams
  - Class diagrams (initial versions)



Overview of Object-Oriented System Development

- Focuses on "how" to implement the system.
- Refines analysis model with implementation-level details.
- Defines class interfaces, relationships, responsibilities.
- Tools include:
  - Class Diagrams
  - Sequence Diagrams
  - Collaboration Diagrams

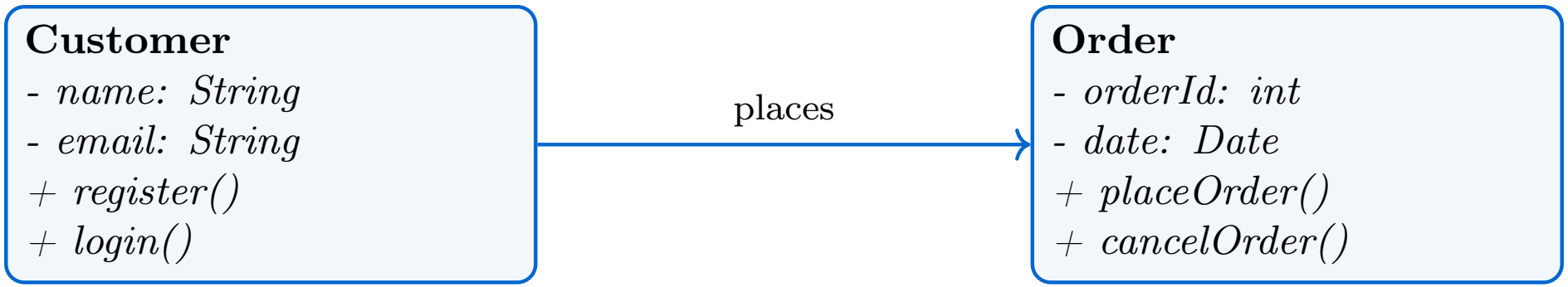


Figure: Sample class diagram showing Customer placing an Order.

Object

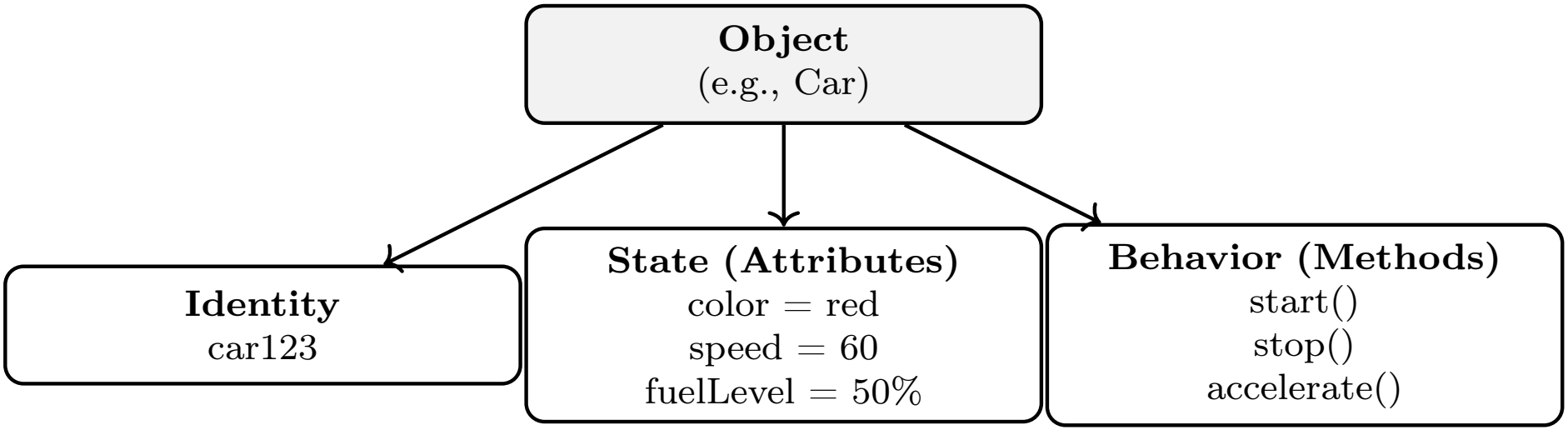
In Object-Oriented System Development, an **object** is a core entity that represents a real-world concept. It encapsulates data and behavior and interacts through messages.

- An **Object** = Identity + State (Attributes) + Behavior (Methods)
- **Identity**: Unique reference to the object
- **State**: Defined by attribute values (e.g., color, speed)
- **Behavior**: Defined by methods (e.g., start(), stop())

Example: A Car

- State: color = red, speed = 60, fuelLevel = 50%
- Behavior: start(), stop(), accelerate()

Object


Object
(e.g., Car)
Identity
car123
State (Attributes)
color = red
speed = 60
fuelLevel = 50%
Behavior (Methods)
start()
stop()
accelerate()

Class in OOP

Class: A class is a blueprint for creating objects. It defines the structure and behavior of those objects.

- A class contains **attributes** (state) and **methods** (behavior).
- An **object** is an instance of a class.
- Real-world analogy: **Class** = House Plan, **Object** = Actual House

Class Structure Example

Key Components:

- **Attributes**: Define the object's state
- **Methods**: Define the object's behavior
- **Constructor**: Initializes object properties

Class with Attributes and Methods

Python Example:

```
class Car:
  def __init__(self, color, speed):
    self.color = color
    self.speed = speed

  def drive(self):
    print(f"Driving at {self.speed} km/h")

  def show_color(self):
    print(f"The car color is {self.color}")
```

Why Use Classes?

- **Code Reusability:** Define once, use many times
- **Encapsulation:** Hides internal complexity
- **Modularity:** Organize code into logical units
- **Maintainability:** Easier to update and scale

Summary:
A class allows bundling of data and functions together, making Object-Oriented Programming more powerful and structured.

Benefits of Object-Oriented Analysis and Design

- Models real-world problems using objects
- Increases modularity and code organization
- Enables reusability of classes and components
- Supports easy scalability and future expansion
- Improves maintainability and debugging
- Facilitates team collaboration and parallel development
- Encourages use of design patterns and best practices

Unified Process (UP) is a **software development process framework** that is:

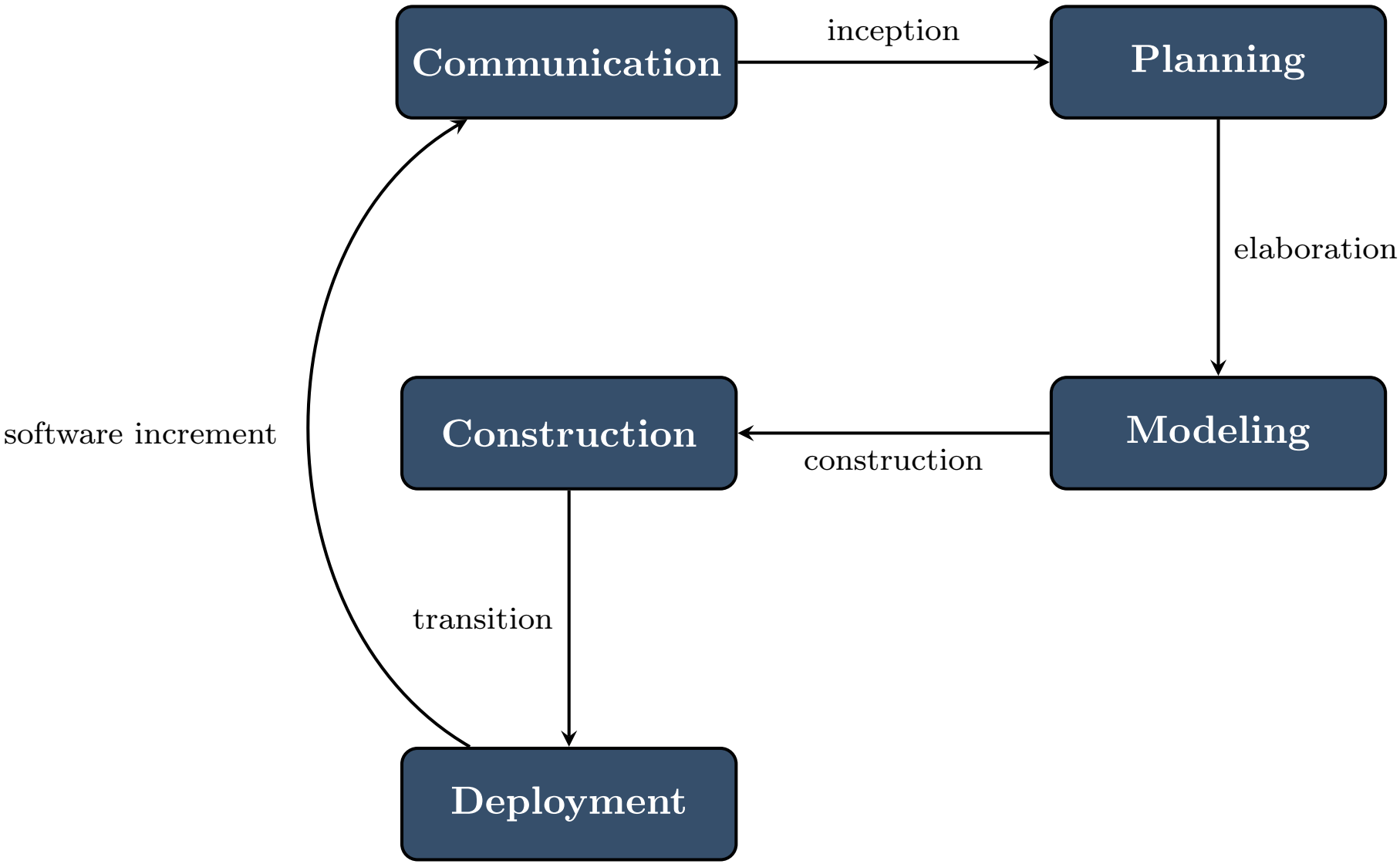
- **Iterative**
- **Incremental**
- **Object-oriented**

It guides teams in:

- Analyzing
- Designing
- Implementing
- Managing software systems using **UML**

Note: UP was developed as a foundation for the **Rational Unified Process (RUP)** and supports object-oriented modeling principles.

Object Basics – The Unified Process



The Unified Process divides the software lifecycle into **four phases**:

Phase	**Goal**
Inception	Understand the project scope, key requirements, and feasibility
Elaboration	Refine the system architecture and resolve high-risk elements
Construction	Build the software product in iterations
Transition	Deploy the system to users, handle feedback and bugs

Inception and Elaboration Phases

Inception Phase

- Identify major use cases and actors.
- Discover key **business objects** and their interactions.
- Define object relationships at a high level.

Elaboration Phase

- Develop detailed **object models**.
- Assign responsibilities using **GRASP** principles.
- Identify classes, interfaces, and hierarchies.
- Create UML diagrams: **class, sequence, state**.

Construction and Transition Phases

Construction Phase

- Implement classes and their methods.
- Ensure encapsulation and object collaboration.
- Build code modules based on the object model.

Transition Phase

- Test object behavior in real-world scenarios.
- Adjust object interactions based on feedback.
- Finalize documentation and deployment.

Object Basics – Unified Process Core Principles

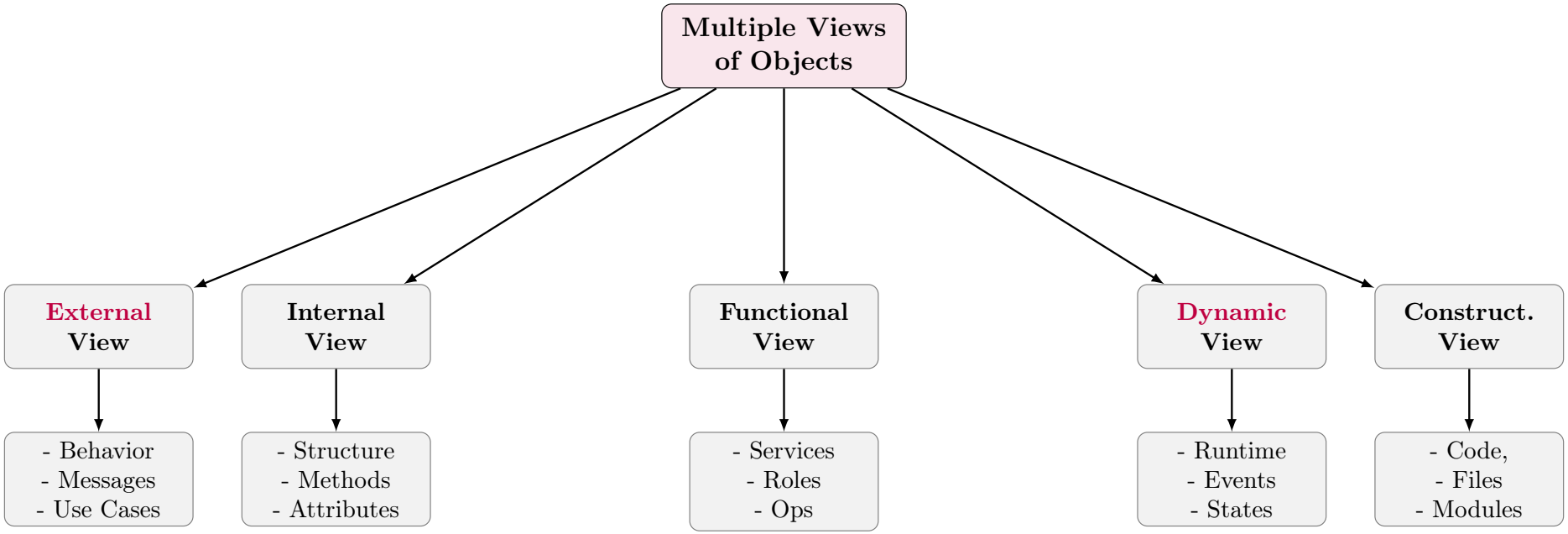
- **Iterative and Incremental:** Development is divided into multiple iterations, with each iteration delivering incremental functionality.
- **Use Case Driven:** Focuses on identifying and prioritizing *use cases* that capture the system's functionality from the user's perspective.
- **Architecture-Centric:** Emphasizes continuous definition and refinement of the system architecture throughout the lifecycle.
- **Risk Management:** Actively identifies, assesses, and mitigates project risks to improve project stability and success.
- **Continuous Validation:** Requirements, design, and implementation are validated continuously through testing, reviews, and stakeholder feedback.

Key Roles in Object-Oriented Development

- **Project Manager:** Oversees planning, scheduling, resource management, and ensures timely delivery within scope and budget.
- **Architect:** Designs system architecture, ensuring scalability, maintainability, and adherence to quality standards.
- **Analyst:** Gathers and analyzes requirements; creates specifications and identifies project risks.
- **Designer:** Develops detailed system designs (e.g., UML diagrams) aligned with the architecture.
- **Developer:** Implements the system as per design, performs unit testing, and integrates components.
- **Tester:** Validates the system against requirements through test case design, execution, and defect reporting.

Multiple Views of Objects

In Object-Oriented Analysis and Design (OOAD), an object can be better understood by viewing it from different perspectives. These views capture an object's structure, behavior, responsibilities, runtime dynamics, and implementation — enabling a complete and modular design.



Overview of UML (Unified Modeling Language)

- UML is a standardized visual modeling language used in software engineering.
- Developed by the Object Management Group (OMG) in the mid-1990s.
- Helps describe, design, and document software systems.
- Provides diagramming techniques for both structural and behavioral aspects.
- Promotes a common understanding among developers, stakeholders, and analysts.
- Includes various diagram types: class, use case, sequence, activity, and state diagrams.

Importance of UML in Software Development

- **Enhanced Communication:** Bridges gaps between technical and non-technical stakeholders.
- **System Visualization:** Helps visualize structure and relationships in complex systems.
- **Improved Documentation:** Serves as comprehensive and maintainable system documentation.
- **Requirement Clarification:** Use case diagrams clarify user requirements.
- **Methodology Support:** Adaptable to both waterfall and agile models.
- **Quality Assurance:** Aids testing and validation by modeling behavior and interactions.

UML Diagrams

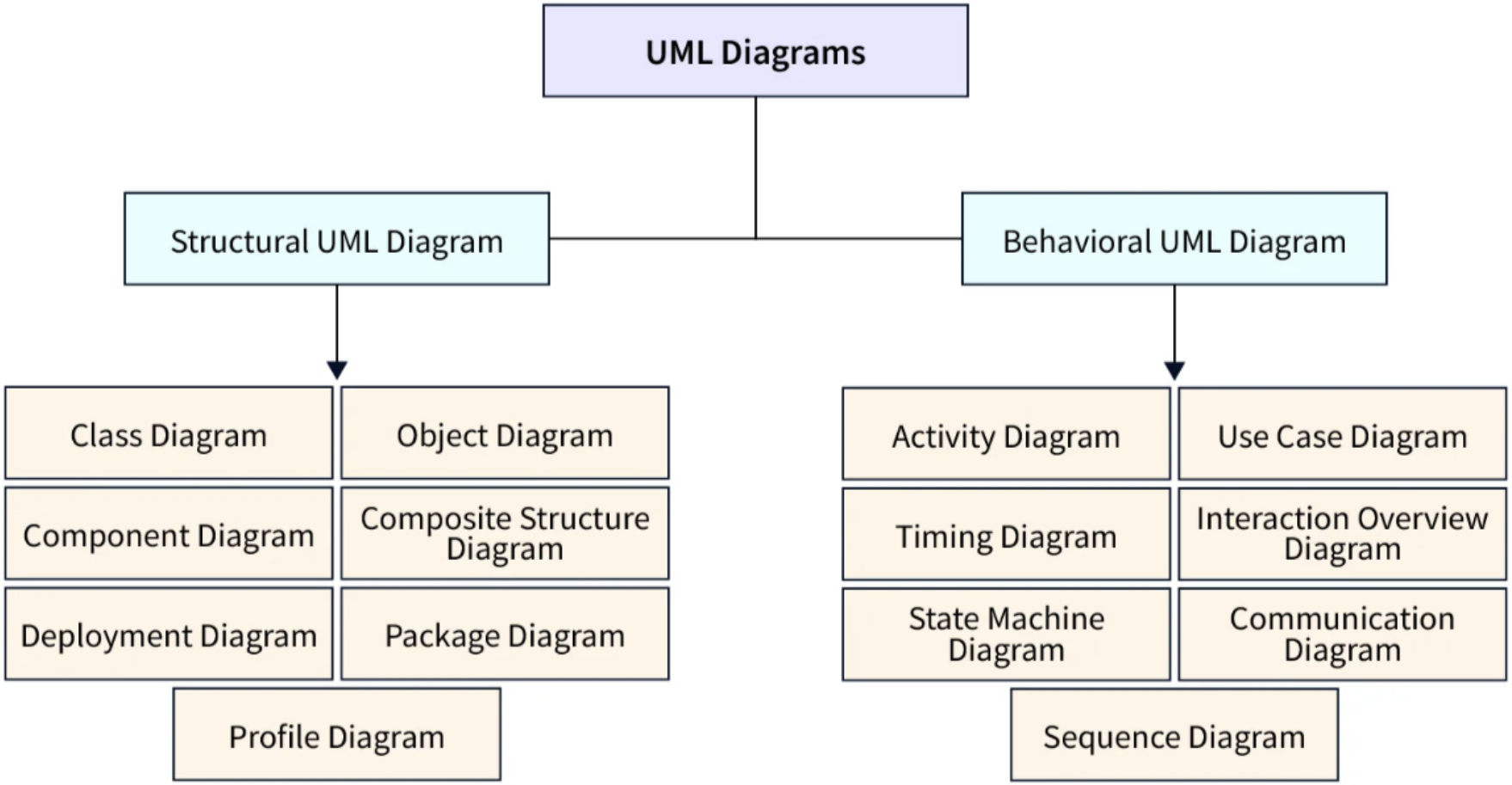


Figure: Sample image caption

UML diagrams are broadly classified into two categories:

Structural UML Diagrams
Structural diagrams represent the static aspects of a system. They focus on the organization of system components, such as classes, objects, and their relationships. These diagrams describe what a system is composed of.

Behavioral UML Diagrams
Behavioral diagrams represent the dynamic aspects of a system. They capture how the system behaves during execution, including interactions among objects, workflows, and state changes in response to events.

What is a Class Diagram in UML?

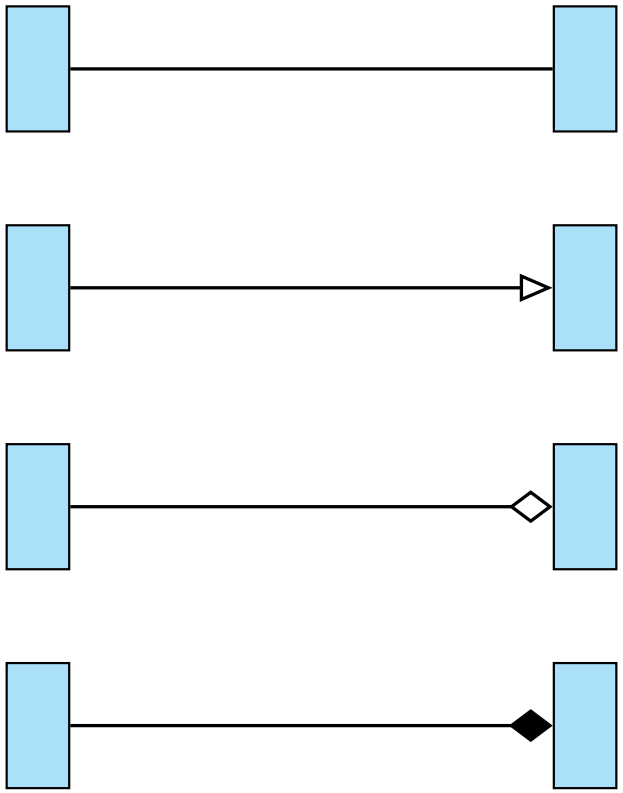
A **Class Diagram** in UML (Unified Modeling Language) is a **static structure diagram** that describes the **blueprint of a system** by showing:

- **Classes** in the system
- **Attributes** (data/properties) of those classes
- **Methods** (operations/functions) of those classes
- **Relationships** between classes (like inheritance, association, aggregation, composition)

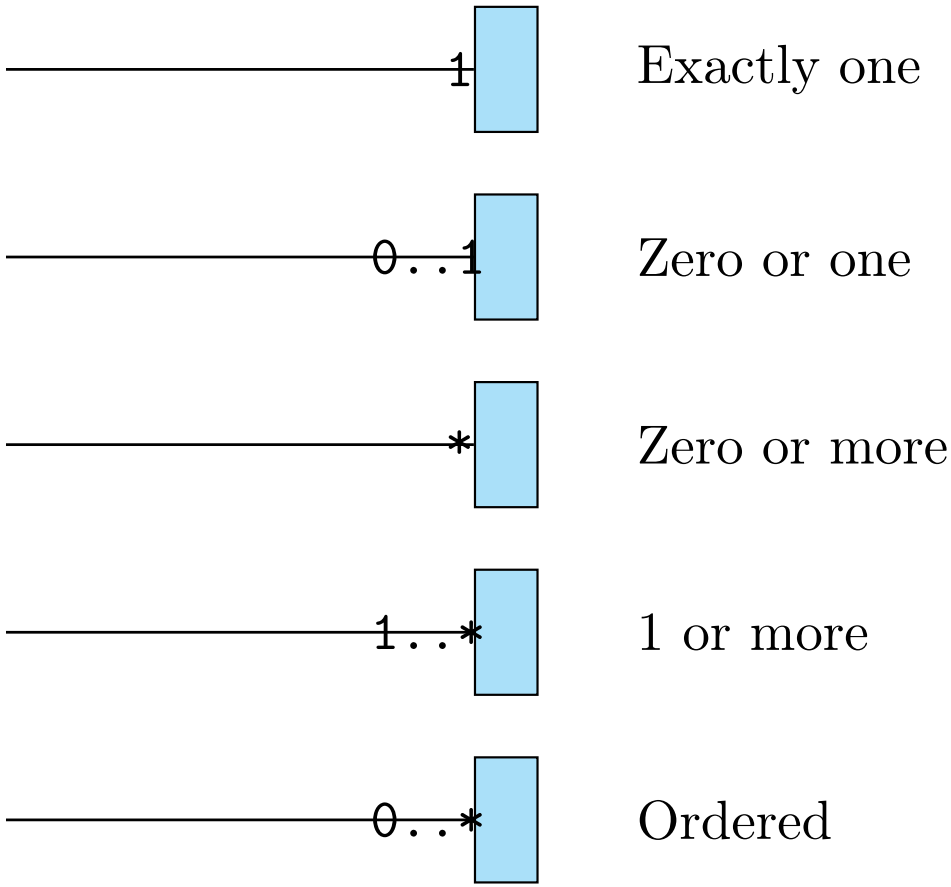
Key Components of a Class Diagram

Element	Description
Class	Represents an object type with attributes and methods
Attribute	Represents a property or characteristic of a class
Method	Represents an operation/function the class can perform
Association	A general link between two classes (e.g., a `Teacher` teaches a `Student`)
Aggregation	A "has-a" relationship with shared ownership (e.g., `Team` has `Players`)
Composition	A strong "has-a" relationship with full ownership (e.g., `House` has `Rooms`)
Inheritance	One class inherits from another (`Dog` is a subclass of `Animal`)
Multiplicity	Indicates how many instances of a class can be associated with another

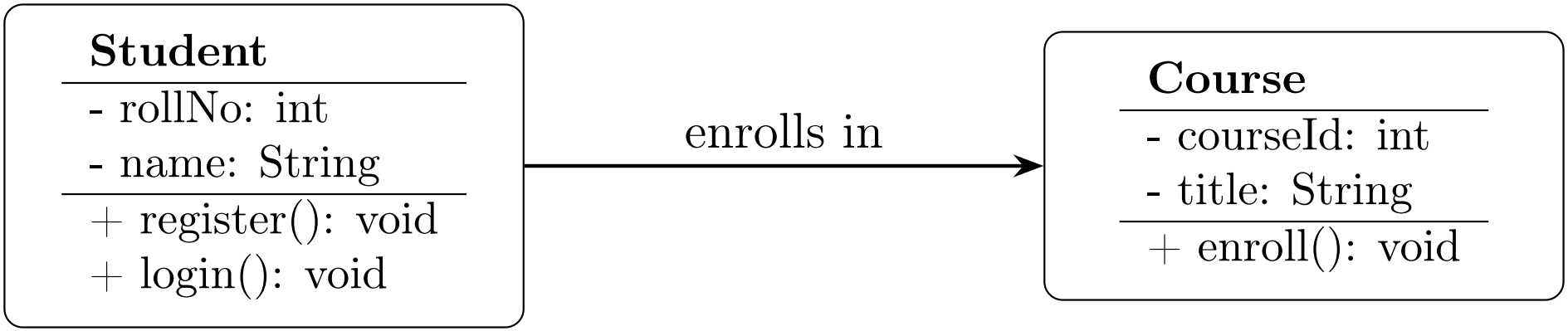
UML Relationships in Class Diagrams



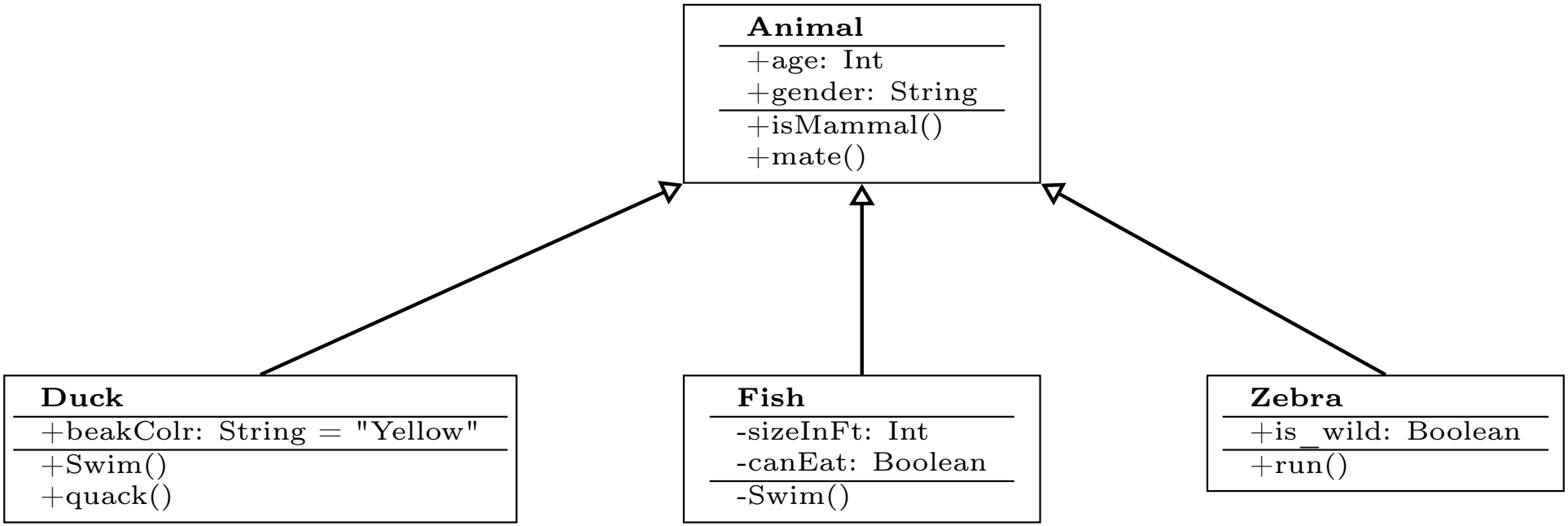
Multiplicity in UML Class Diagrams



Class Diagram



Animal Inheritance Class Diagram



Object Diagram

An **Object Diagram** is a type of structural diagram in UML that shows the **instances of classes (i.e., objects)** at a particular moment in time. It includes the objects, their current attribute values, and the links (relationships) between them.

It is essentially a **snapshot** of the system, derived from a Class Diagram, with actual data values instead of abstract definitions.

Key Characteristics

Feature	Description
Focus	Instance-level representation (not class-level)
Timing	Represents the system at a specific moment (static snapshot)
Derived from	Class Diagram
Purpose	To show real-time examples of data structures and object links
Usage	Used in debugging, reverse engineering, and test case modeling

Components of an Object Diagram

- **Objects:** Instances of classes, written as `objectName : ClassName`, with attribute values.
- **Links:** Connections between objects (instances of associations).
- **Attribute Values:** Show actual state of an object at a particular moment.

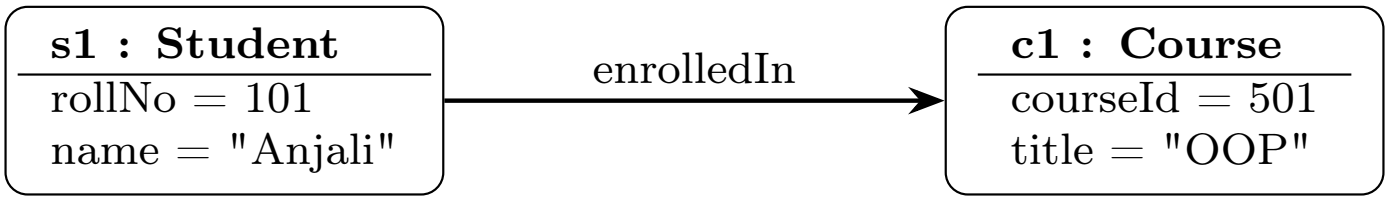
Class Diagram and Object Diagram Example

Class: Student

- Attributes: `rollNo: int`, `name: String`
- Methods: `register()`, `login()`

Class: Course

- Attributes: `courseId: int`, `title: String`
- Methods: `enroll()`



Comparison: Class Diagram vs Object Diagram

Class Diagram	Object Diagram
Defines structure using classes	Shows specific instances (objects)
Abstract — no real values	Concrete — contains real data values
Shows associations and operations	Shows object links at a point in time
Used for system design	Used for system analysis or examples

What is a Use Case Diagram in UML?

Definition:
A **Use Case Diagram** is a type of **Unified Modeling Language (UML)** diagram that represents the interaction between *actors* (users or external systems) and a *system* to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating various ways users can interact with it.

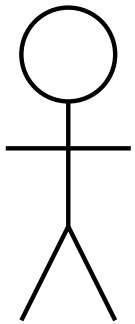
Use-Case Diagram Notations:

- **Actors:** Represented by stick figures, denote external users or systems.
- **Use Cases:** Represented by ovals, denote system functionalities.
- **System Boundary:** Rectangle enclosing use cases.

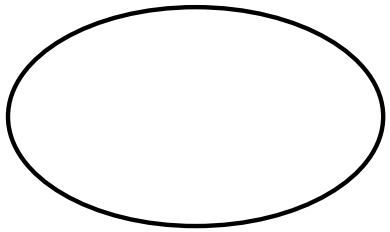
When to Apply Use Case Diagrams?

- To gather and clarify user requirements.
- To communicate with non-technical stakeholders.
- During system design to outline user interactions and plan features.
- To define system boundaries and identify external elements.

Use Case Diagram Notations



Actor



UseCase

System Boundary

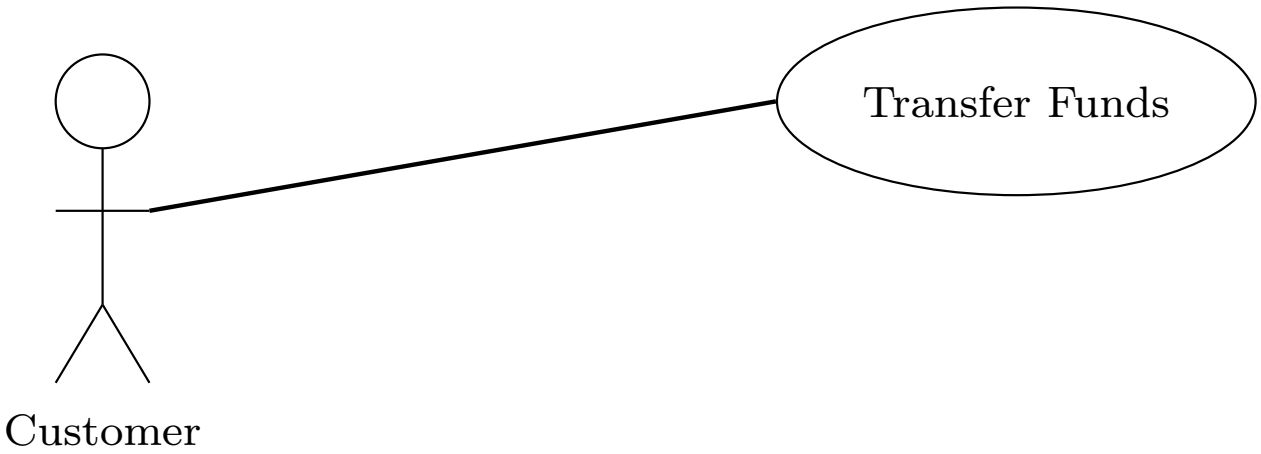
Use Case Diagram Relationships

1. Association Relationship

In a Use Case Diagram, relationships depict interactions between *actors* and *use cases*. The Association Relationship represents a communication between an actor and a use case using a solid line.

Example – Online Banking System:

- **Actor:** Customer
- **Use Case:** Transfer Funds
- **Association:** Customer interacts with the use case "Transfer Funds"



*Represents communication between an actor (**Customer**) and a use case (**Transfer Funds**).*

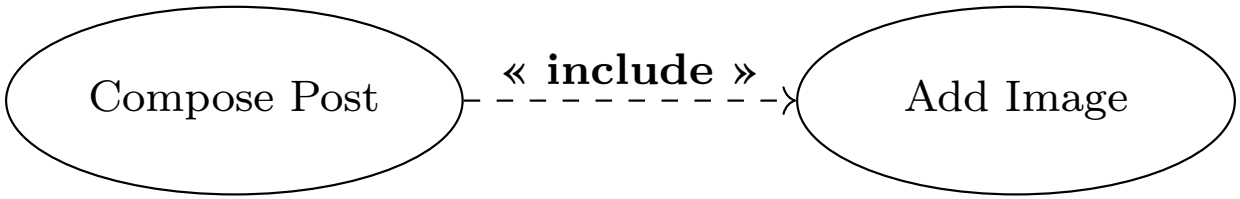
Use Case Diagram Relationships

2. Include Relationship

The **Include Relationship** indicates that a use case includes the functionality of another use case. It is denoted by a *dashed arrow* pointing from the including use case to the included one. This supports modular and reusable design.

Example – Social Media Posting:

- **Use Cases:** Compose Post, Add Image
- **Include:** "Compose Post" includes the functionality of "Add Image"



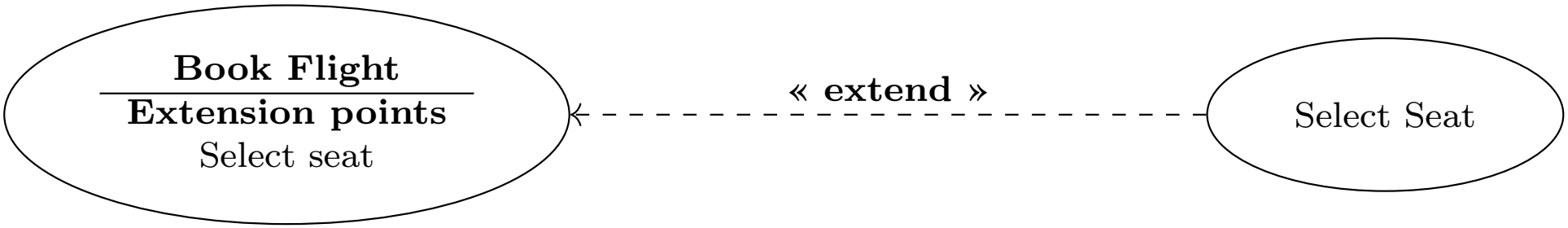
"Compose Post" includes the functionality of another use case "Add Image."

3. Extend Relationship

The **Extend Relationship** illustrates that a use case can be extended by another use case under specific conditions. It is represented by a *dashed arrow* with the keyword **«extend»**. This relationship is useful for optional or exceptional behavior.

Example – Flight Booking System:

- **Use Cases:** Book Flight, Select Seat
- **Extend:** "Select Seat" may extend "Book Flight" when the user opts to choose a seat



The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat.

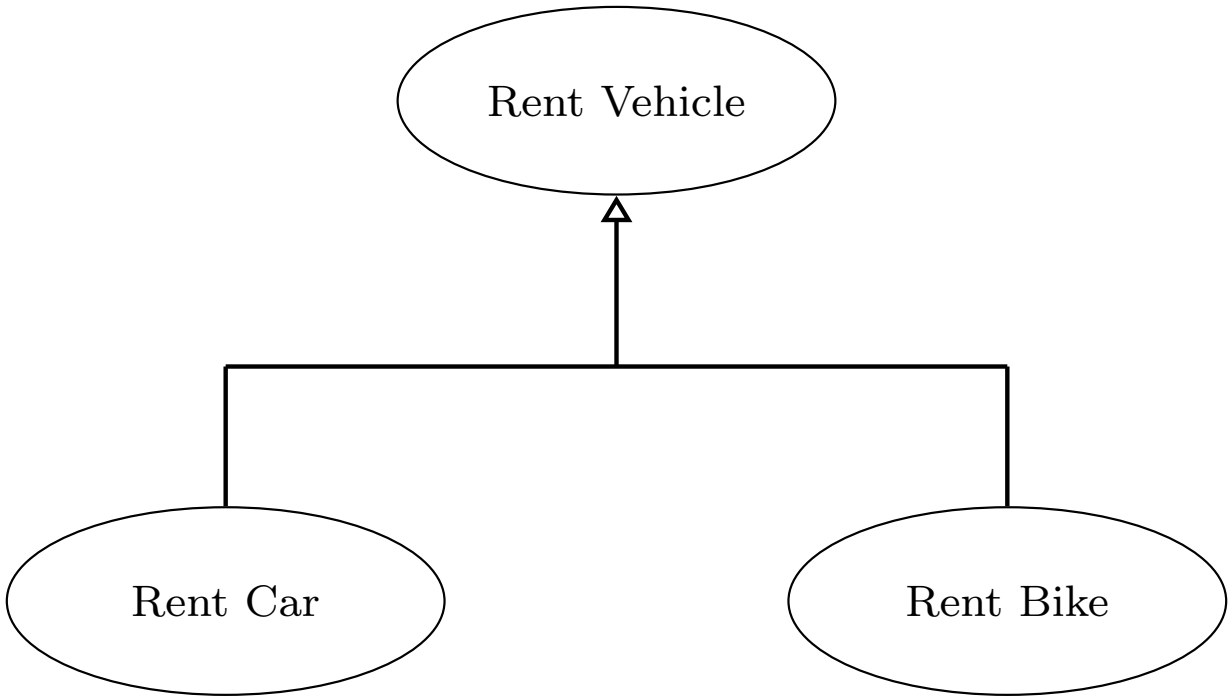
Use Case Diagram Relationships

4. Generalization Relationship

The **Generalization Relationship** establishes an *"is-a"* connection between use cases, indicating specialization. Represented by a solid line with a *hollow triangle* arrow pointing to the general use case.

Example – Vehicle Rental System:

- **Use Cases:** Rent Car, Rent Bike
- **Generalization:** Both are specializations of "Rent Vehicle"



Steps to Draw a Use Case Diagram

Below are the main steps to draw a use case diagram in UML:

1. **Identify Actors:** Determine who or what interacts with the system. These can be users, external systems, or devices.
2. **Identify Use Cases:** List the primary actions or services the system must support. Each use case represents a functionality.
3. **Connect Actors and Use Cases:** Draw *associations* (lines) between actors and their respective use cases to represent interactions.
4. **Add System Boundary:** Enclose the use cases within a rectangle to indicate what is part of the system and what is external.
5. **Define Relationships:** Use *«include»*, *«extend»*, and generalization arrows to show relationships among use cases.
6. **Review and Refine:** Examine the diagram to ensure accuracy and completeness. Remove redundancies or fix errors.
7. **Validate:** Share the diagram with stakeholders. Confirm it reflects their expectations and understanding of the system.

Understanding Use Case with a Practical Scenario

Let's understand how to draw a Use Case Diagram using the Online Shopping System:

Actors:

- Customer
- Admin

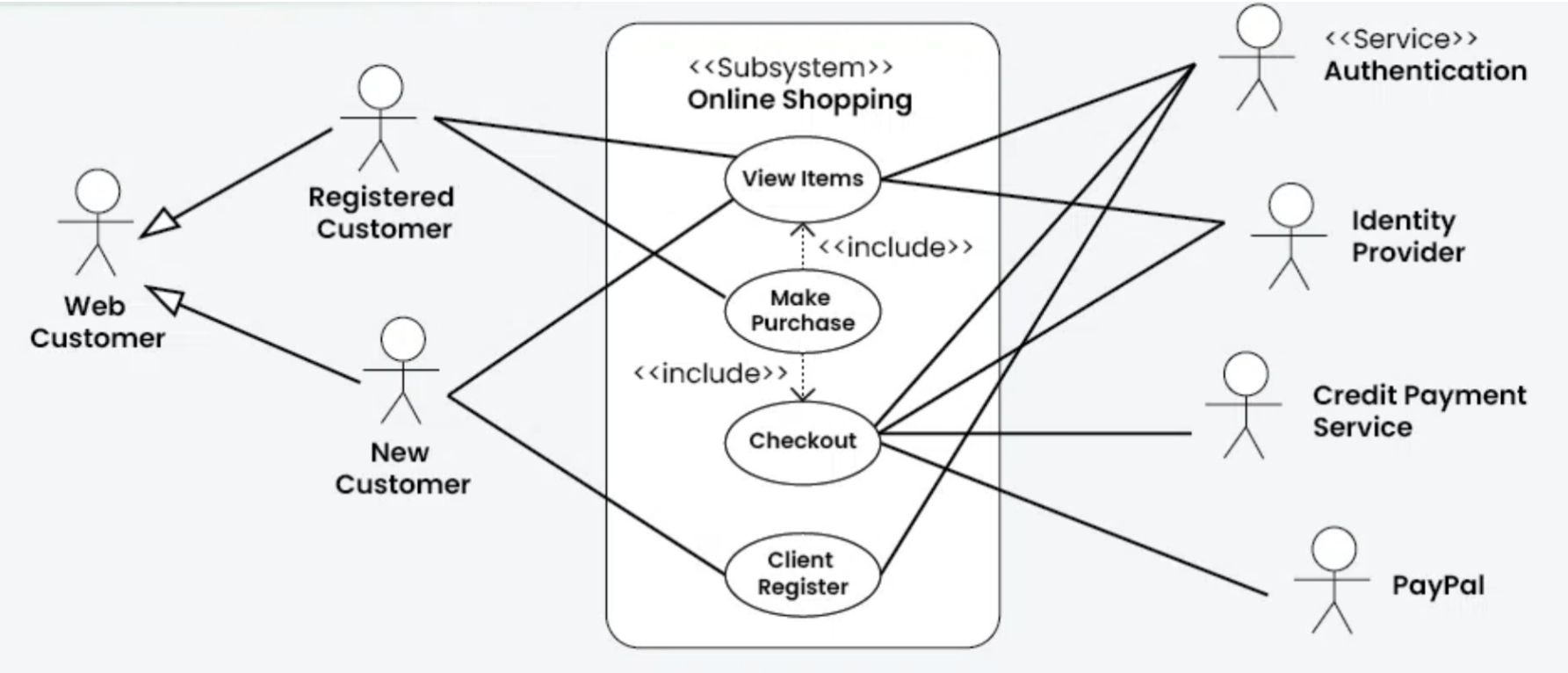
Use Cases:

- Browse Products
- Add to Cart
- Checkout
- Manage Inventory (Admin)

Relations:

- The **Customer** can:
  - Browse Products
  - Add to Cart
  - Checkout
- The **Admin** can:
  - Manage Inventory

UML Use Case Diagram Example



This diagram illustrates the interaction between different customers and services with the online shopping system. It includes relationships like «include» and subsystem boundaries.

Activity Diagrams

Activity diagrams show the steps involved in how a system works, helping us understand the flow of control.

Key Characteristics:

- Show the **order of activities** in a system.
- Represent whether actions happen **sequentially** or **concurrently**.
- Help explain **what triggers** certain events or decisions.

Structure of an Activity Diagram:

- Starts from an **initial node** (black circle).
- Ends at a **final node** (encircled black dot).
- May contain **decision nodes** (diamonds) to represent alternate paths.
- May show **forks and joins** for parallel execution.

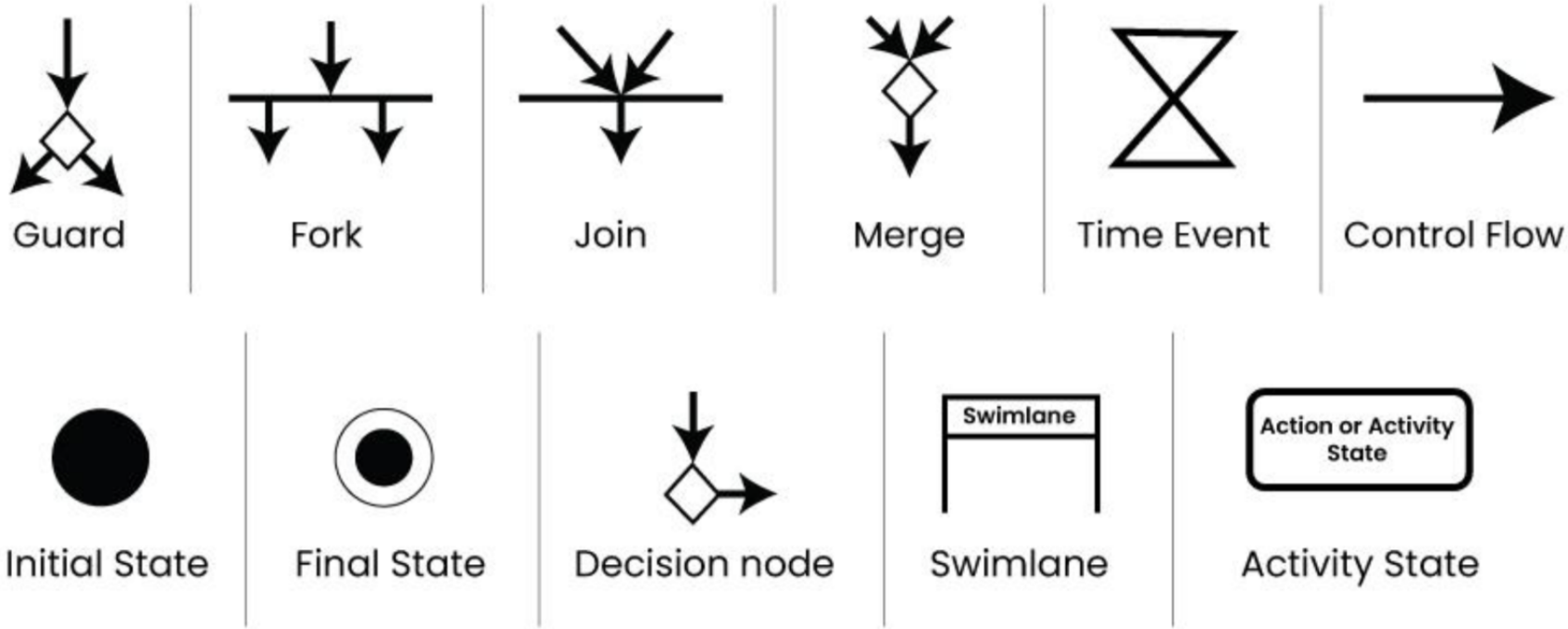
Scenarios for Using Activity Diagrams

Activity diagrams are useful when you need to visually represent the flow of processes or behaviors in a system.

Key Situations to Use an Activity Diagram:

- **Modeling Workflows or Processes:**
  - Map out a business process, workflow, or steps of a use case.
  - Visualize the sequence and flow of activities.
- **Concurrent or Parallel Processing:**
  - Represent activities that happen simultaneously.
  - Show parallel paths of execution.
- **Understanding Dynamic Behavior:**
  - Depict how the system transitions between states.
  - Show behavior changes based on events or conditions.
- **Clarifying Complex Logic:**
  - Simplify complex decision-making with branching paths.
  - Highlight different outcomes for decisions.
- **System Design and Analysis:**
  - Help developers and stakeholders understand system interactions.

Standard UML Notation Symbols



Understanding Initial Basic Notations

1. **Initial State**
   - The starting point of an activity diagram.
   - Represented as a filled black circle.
   - Only one initial state exists unless depicting nested activities.
   - Marks the entry point or instantiation of an object.
2. **Action or Activity State**
   - Represents execution of an action.
   - Shown using a rectangle with rounded corners.
   - Example: "Open Application".
3. **Action Flow / Control Flow**
   - Indicates transition from one activity state to another.
   - Represented as a line with an arrowhead.
   - May include constraints or conditions.
4. **Decision Node and Branching**
   - Used to make choices or decisions.
   - Represented as a diamond shape.
   - Must have two or more outgoing flows.
   - Can be labeled with guard conditions.

Fork, Join, Guard, Merge

5. **Guard**
   - A condition next to a decision node on an arrow.
   - Determines the path of execution based on its truth value.
   - Written in square brackets, e.g., [number $>$ 0]
6. **Fork**
   - Splits one activity into multiple concurrent activities.
   - Represented by a thick bar with one input and multiple outputs.
   - All resulting actions must execute.
7. **Join**
   - Combines multiple concurrent activities into one.
   - Represented by a thick bar with multiple inputs and one output.
   - All incoming actions must complete before proceeding.
8. **Merge**
   - Merges non-concurrent alternate flows into one.
   - Represented as a diamond shape.
   - Useful when different conditions lead to the same next step.

Swimlanes, Time Event, Final State

9. Swimlanes

- Group related activities into vertical or horizontal sections.
- Improves readability and modularity.
- Optional but recommended for complex diagrams.

10. Time Event

- Represents delay or time-based trigger.
- Shown as an hourglass shape.
- Useful to depict delays or wait times in processes.

11. Final State / End State

- Indicates the end of the process.
- Represented by a filled circle within another circle.
- Multiple final states are possible.

Activity Diagram vs Flowchart

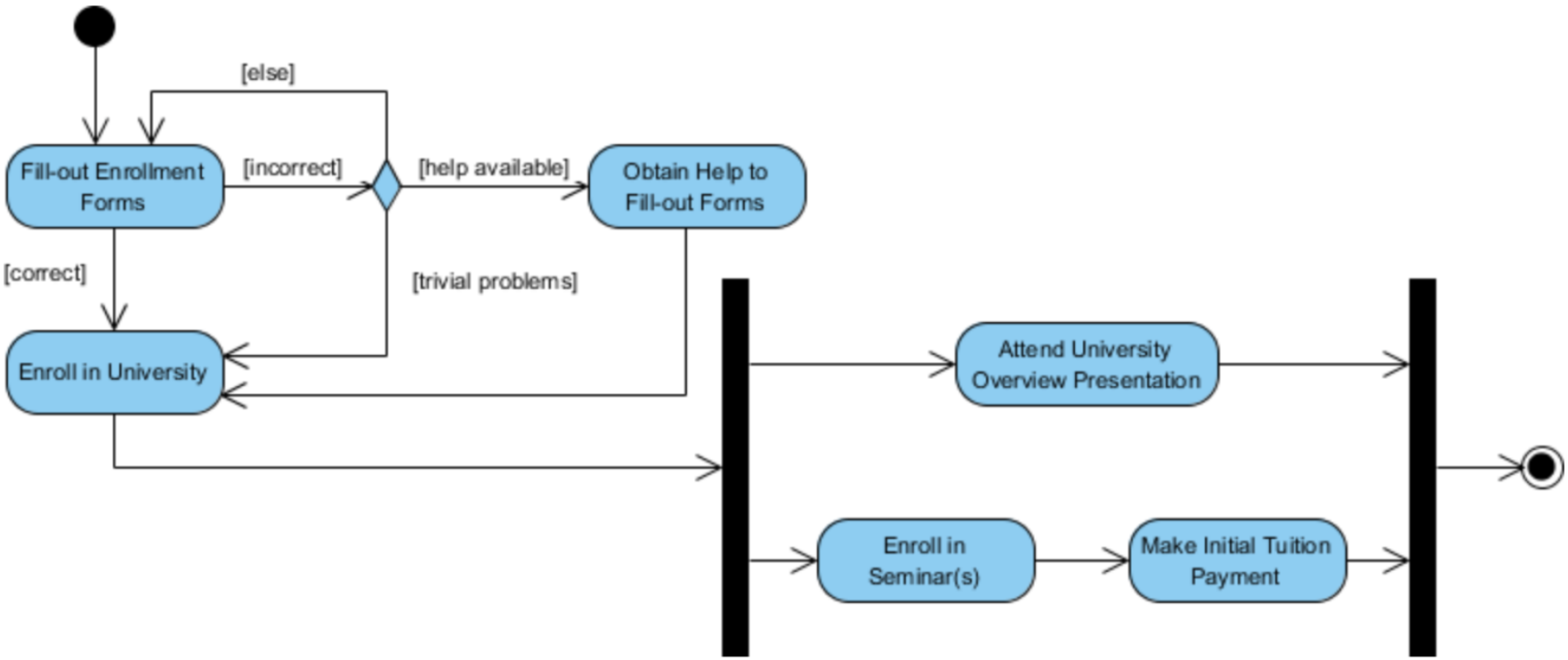
Aspect	Activity Diagram	Flowchart
Purpose	Represents the flow of control in a system or process, focusing on activities and conditions.	Depicts the step-by-step flow of a process or algorithm, focusing on operations.
Used In	Primarily used in software and systems modeling, especially in UML.	Commonly used in programming, business, and process mapping.
Complexity	Can represent both simple and complex workflows, including concurrent processing.	Typically used for simpler, linear processes, though can handle limited complexity.
Symbols	Uses UML symbols like initial nodes, activity nodes, decisions, swimlanes, etc.	Uses basic symbols like rectangles (process), diamonds (decisions), and arrows (flow).
Concurrency Representation	Supports concurrent activities and parallel flows.	Does not support parallel processes explicitly; focuses on sequential flow.

Understanding Student Enrollment Process

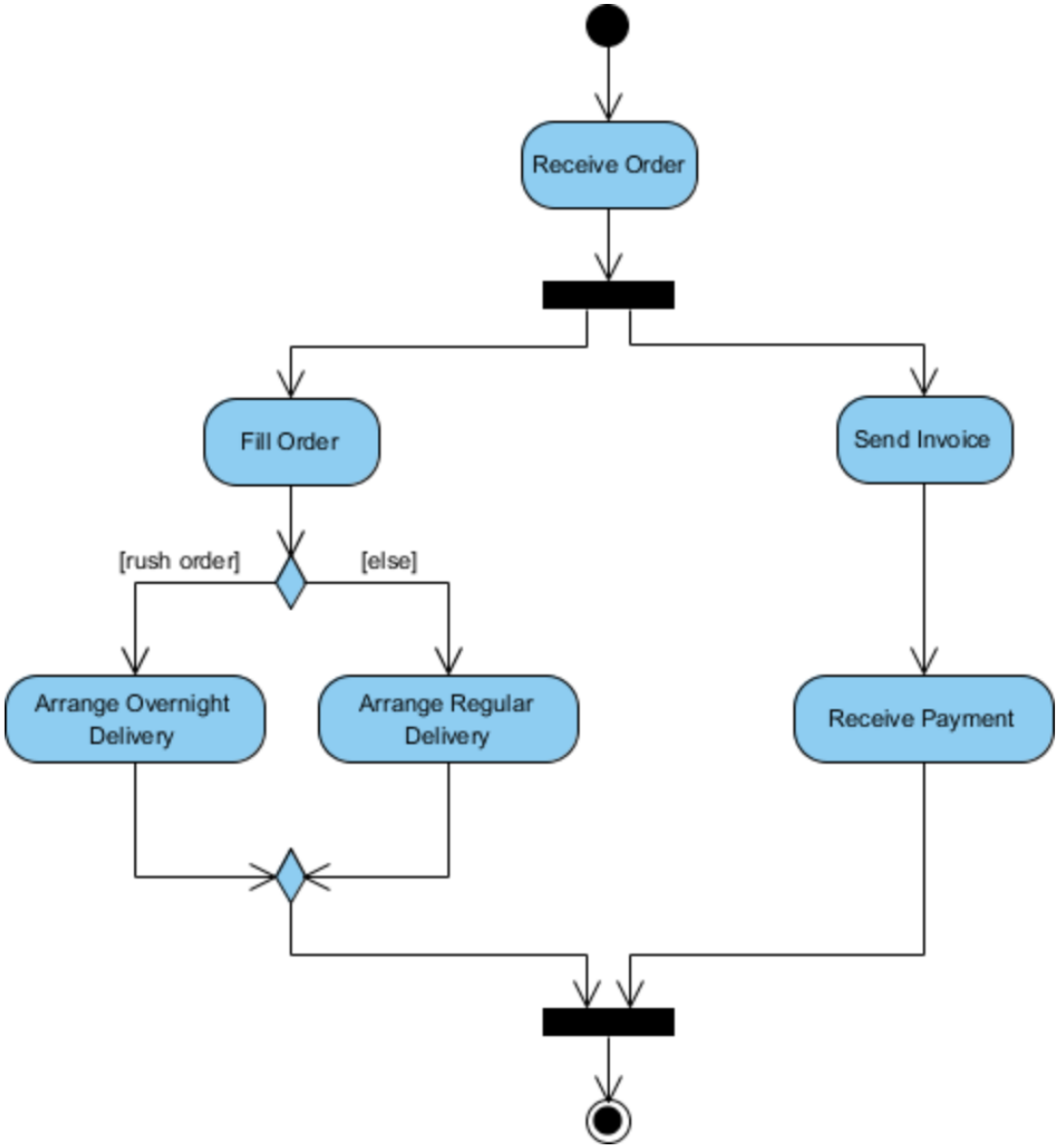
Scenario: Student Enrollment activity diagram describes the process for student enrollment in a university:

- An applicant wants to enroll in the university.
- The applicant submits a completed Enrollment Form.
- The registrar reviews and inspects the form.
- If the form is filled out correctly:
  - The registrar informs the student to attend a university overview presentation.
  - The registrar helps the student enroll in seminars.
  - The registrar asks the student to pay for the initial tuition.

Activity Diagram - Student Enrollment



Activity Diagram – Processing an Order



State Diagram

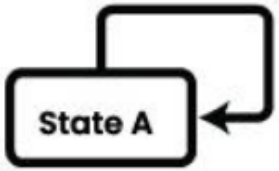
- A **State Machine Diagram** (also called **State Diagram** or **State-Chart Diagram**) is a UML behavioral diagram.
- It represents the condition or **state of a system or object** at finite points in time.
- It focuses on the **transitions between different states**, triggered by external events or stimuli.
- These diagrams are used to **model dynamic behavior of a class** based on time and changing conditions.
- While every class may have states, **not every class requires a state diagram**.

Common UML State Diagram Symbols

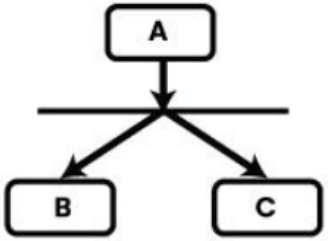
Initial State

State

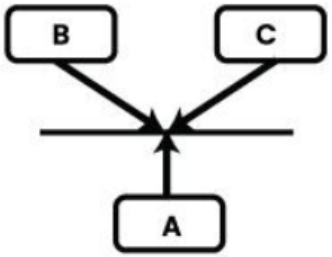
Transition



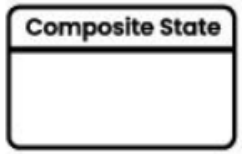
Self Transition



Fork



Join



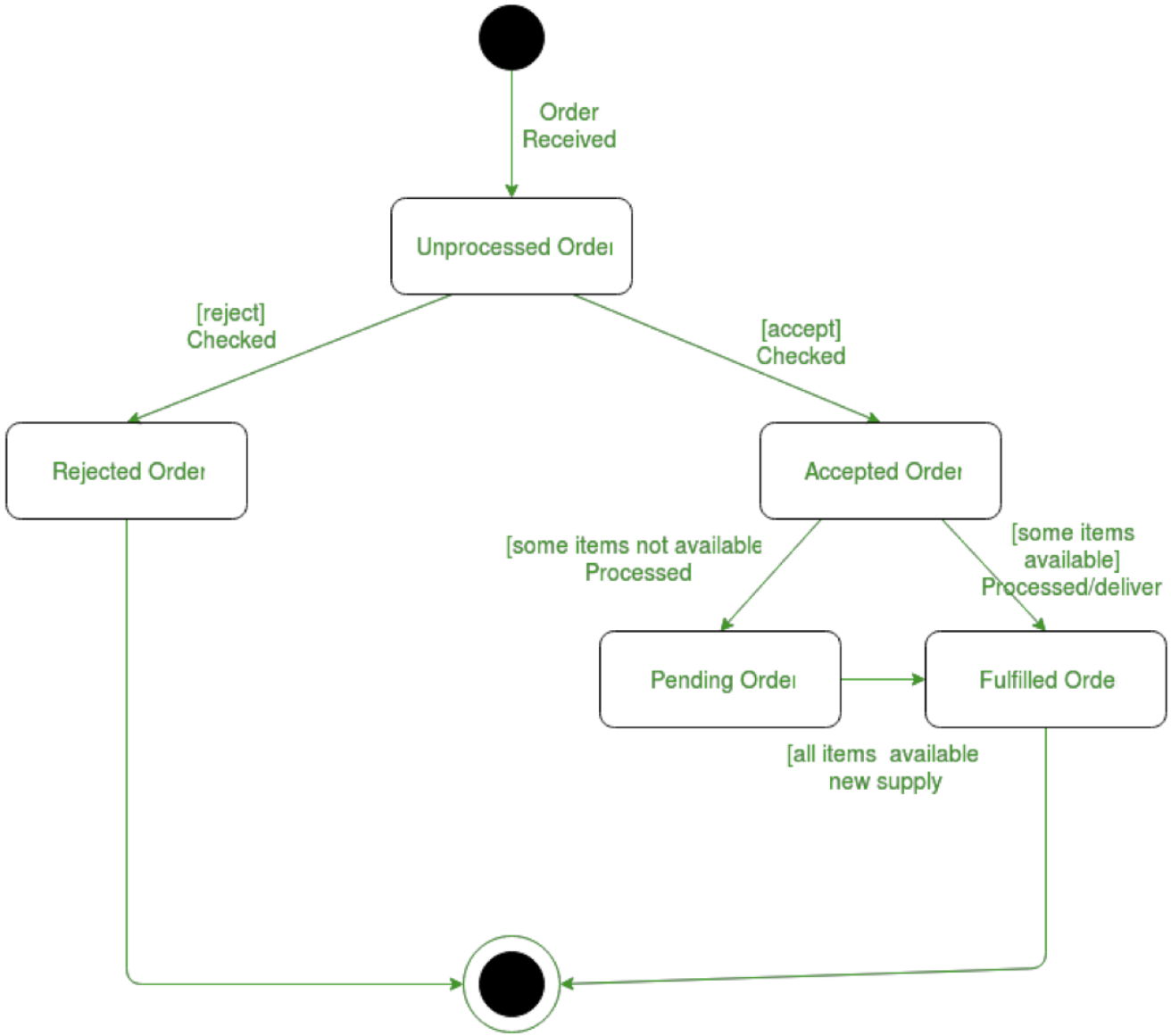
Composite State

Final State

Common UML State Diagram Symbols

1. **Initial State** – Represented by a black filled circle. It denotes the starting point of a system or class.
2. **Transition** – Represented by a solid arrow from one state to another, labeled with the event triggering the change.
3. **State** – A rounded rectangle symbol that represents the condition or situation of an object at an instant of time.
4. **Fork** – A thick horizontal bar with one incoming and multiple outgoing arrows. It shows a single state splitting into concurrent states.
5. **Join** – A thick horizontal bar with multiple incoming and one outgoing arrow. It represents the convergence of concurrent states.
6. **Self Transition** – An arrow that loops back to the same state. Used when an event does not cause a state change.
7. **Composite State** – A rounded rectangle divided into compartments, representing a state with internal activities.
8. **Final State** – A filled black circle inside an unfilled circle. It denotes the termination of the state machine.

State Diagram Example – Online Order



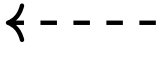
Sequence Diagrams

Sequence diagrams are UML diagrams that visually represent the interactions between objects or components in a system over time. They emphasize the order and timing of messages or events exchanged between system elements.

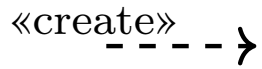
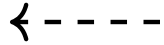
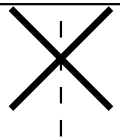
Why Use Sequence Diagrams?

- **Visualizing Dynamic Behavior:** Show sequential object interactions, clarifying dynamic workflows.
- **Clear Communication:** Intuitively convey behavior without needing to look at code.
- **Use Case Analysis:** Represent and analyze how a process unfolds within a use case.
- **Designing System Architecture:** Help define communication between distributed system components.
- **Documenting System Behavior:** Useful for developers and maintenance teams to understand system flow.
- **Debugging and Troubleshooting:** Identify potential errors, inefficiencies, or bottlenecks.

Sequence Diagram Notations

Symbol	Name	Description
	Synchronous message symbol	Represented by a solid line with a solid arrowhead. Used when a sender must wait for a response before continuing. The diagram should show both the call and the reply.
	Asynchronous message symbol	Solid line with lined arrowhead. Asynchronous messages don't require a response before the sender continues. Only the call is shown.
	Asynchronous return message symbol	Dashed line with lined arrowhead, representing return messages.

Sequence Diagram Notations

Symbol	Name	Description
	Asynchronous create message symbol	Dashed line with arrowhead; this message creates a new object.
	Reply message symbol	Dashed line with arrowhead, used to show replies to calls.
	Delete message symbol	Solid line followed by an X; this message destroys an object.

Sequence Diagram Notations

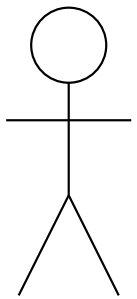
1. Actor

An **actor** in a UML diagram represents a role that interacts with the system from outside. It can be a human, another system, or a device.

Purpose:

- Shows external entities interacting with the system.
- Helps define system boundaries.

Notation:

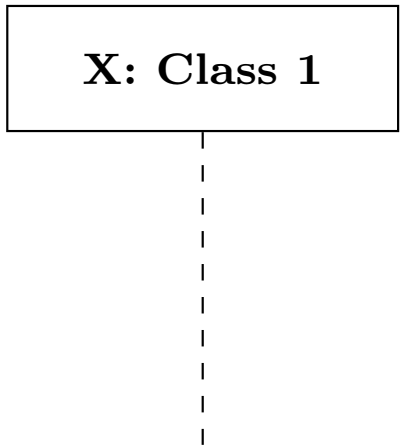


Actor

Sequence Diagram Notations – Lifelines

2. Lifeline

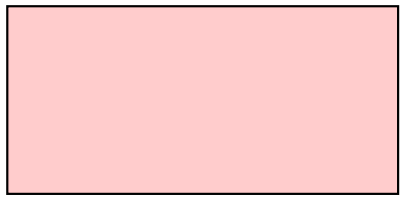
- A **lifeline** is a named element which depicts an individual participant in a sequence diagram.
- Each instance in a sequence diagram is represented by a lifeline.
- Lifeline elements are located at the **top** in a sequence diagram.
- Standard UML format: `<object name>: <class name>`



Sequence Diagram Notations – Object Symbol

3. Object Symbol

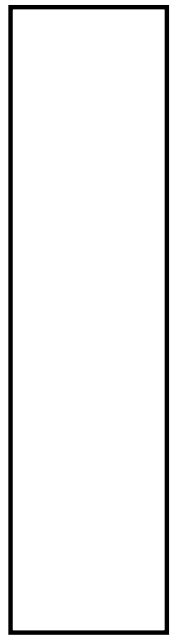
- Represents a **class or object** in UML.
- Demonstrates how an **object behaves** in the context of the system.
- **Class attributes** should not be listed in this shape.



Sequence Diagram Notations – Activation Box

4. Activation Box

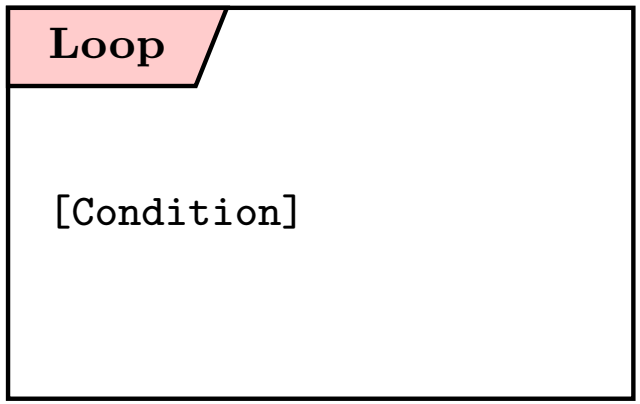
- Represents the **time needed** for an object to complete a task.
- The longer the task takes, the **longer** the activation box becomes.
- It shows the **active period** of the object in the interaction.



Sequence Diagram Notations – Option Loop Symbol

6. Option Loop Symbol

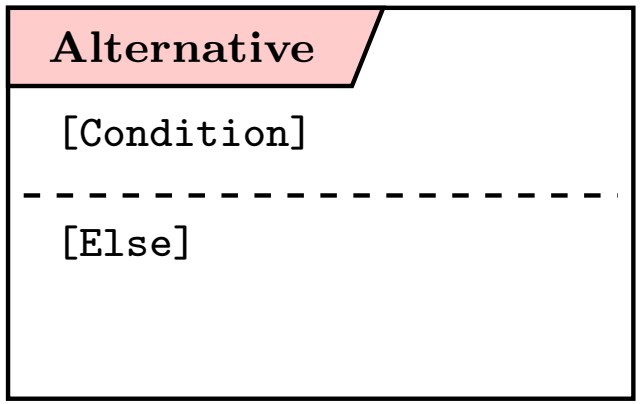
- Models **if/then** conditions within sequence diagrams.
- Represents a fragment that executes **only under specific conditions**.
- The **Loop** label and condition are placed within the boundary box.



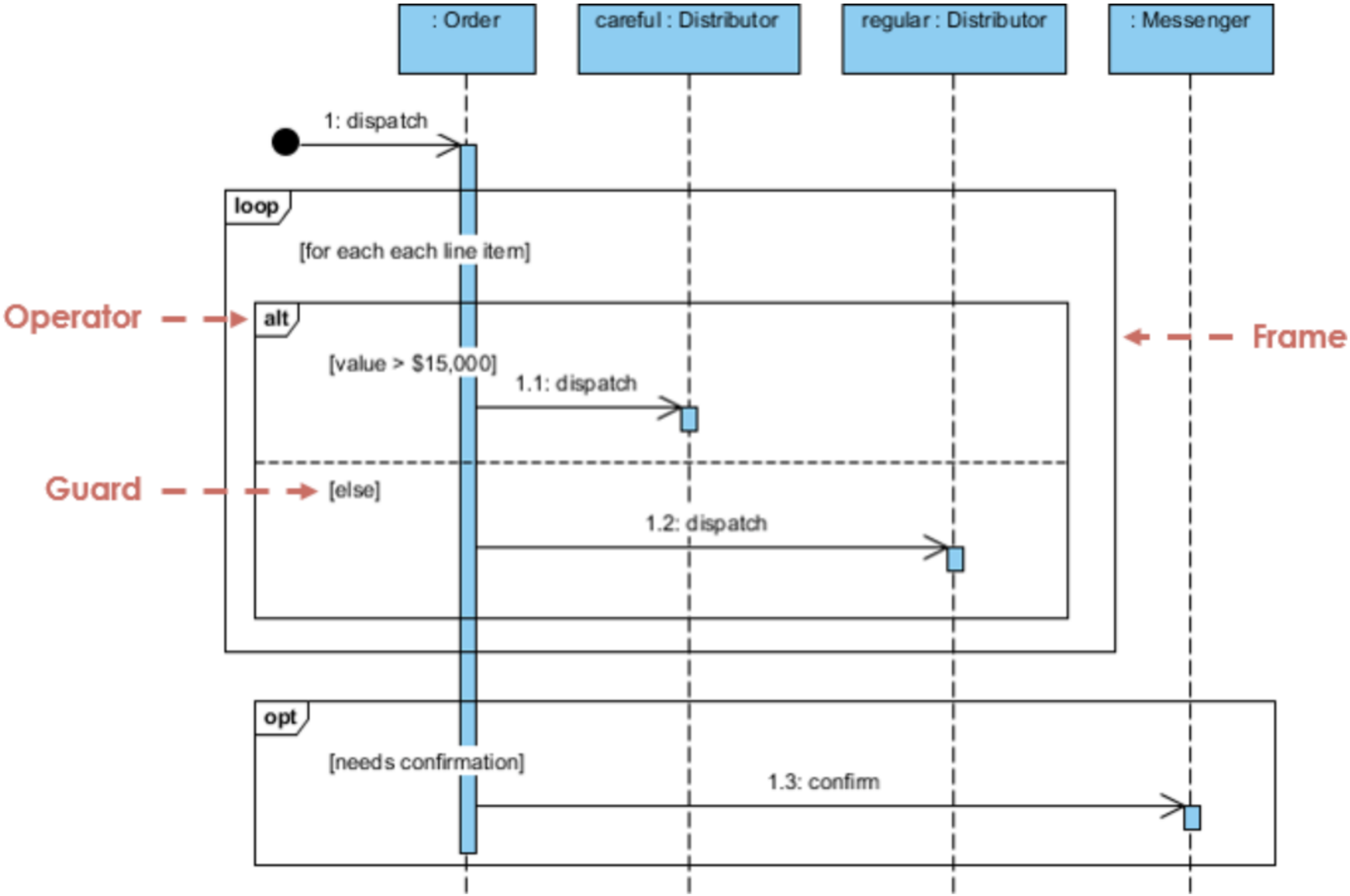
Sequence Diagram Notations – Alternative Symbol

Alternative Symbol

- Symbolizes a **mutually exclusive choice** between two or more message sequences.
- Labeled rectangle with a dashed line separating **condition** and **else**.
- Commonly used to model **if-else** logic in sequence diagrams.



Dispatch Order Sequence Diagram



Book Search Sequence Diagram

Search Book : Use Case

- Main scenario -

The Customer specifies an author on the Search Page and then presses the Search button.

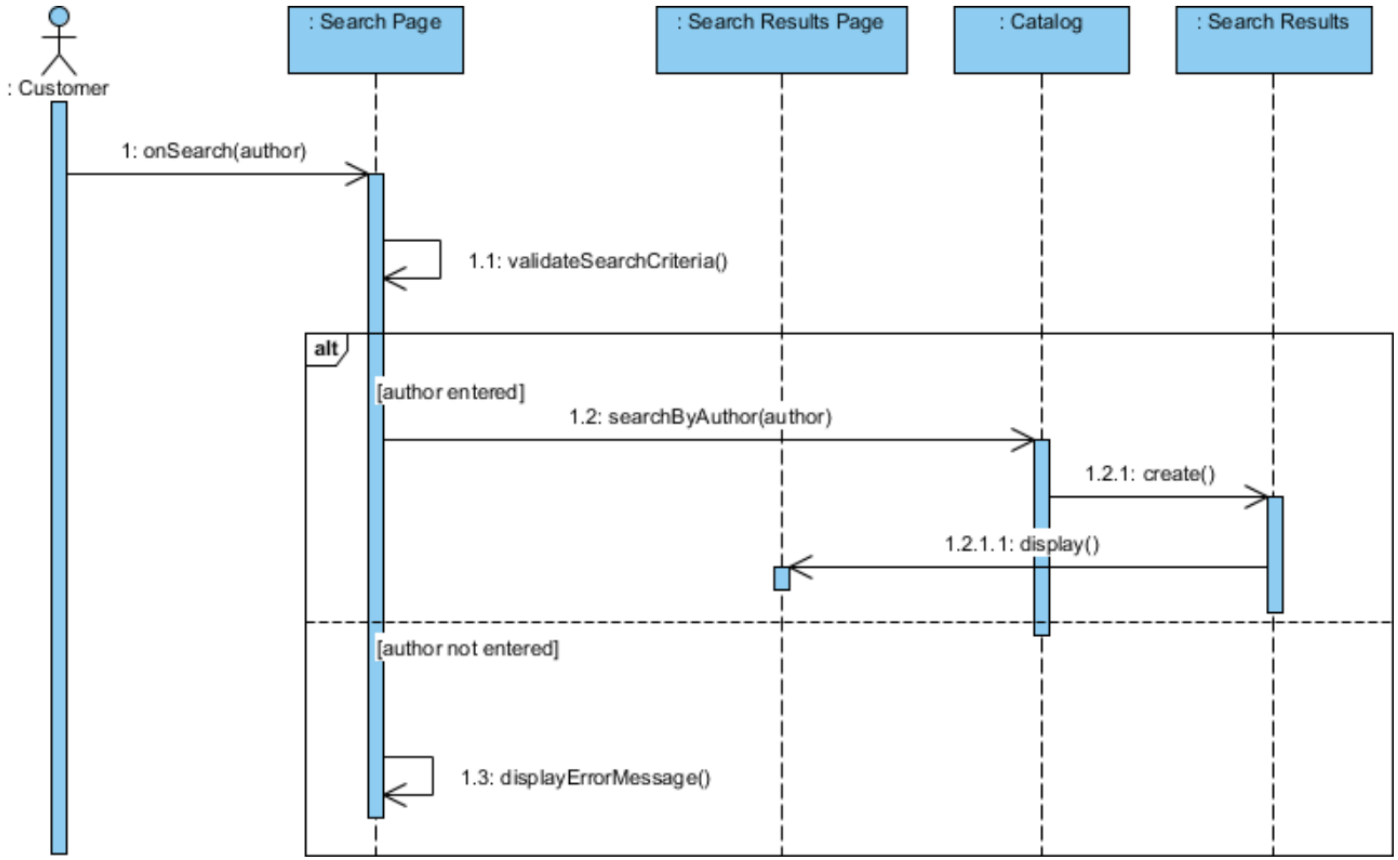
The system validates the Customer's search criteria.

If author is entered, the System searches the Catalog for books associated with the specified author.

When the search is complete, the system displays the search results on the Search Results page.

- Alternate path -

If the Customer did not enter the name of an author before pressing the Search button, the System displays an error message



The Rational Unified Process (RUP)

- The Rational Unified Process (RUP) is an **agile software development and project management framework**.
- It is used to build **complex systems** and supports the entire **Software Development Life Cycle (SDLC)**.
- Developed by **Rational Software**, which is now a part of **IBM**.
- RUP follows an **iterative and incremental** approach to development.
- It is the most widely-used specific variant of the generic **Unified Process (UP)**.

Phases of RUP

There are a total of five phases in the life cycle of RUP:

- Inception
- Elaboration
- Construction
- Transition
- Production

RUP Phase: Inception

- Focus: **Communication and Planning**.
- Identifies the scope of the project using a **use-case model**, enabling estimation of cost and time.
- Captures customers' requirements to aid planning.
- Key outputs:
  - Vision & Scope
- Project is reviewed against milestone criteria:
  - If criteria are not met, project may be canceled or redesigned.

RUP Phase: Elaboration

- Focus: **Planning and Modeling**.
- Detailed evaluation and risk mitigation.
- Key outputs:
  - Architecture & risk handling
- Establish an **Executable Architecture Baseline**.
- Again, checked against milestone criteria:
  - Failure results in redesign or cancellation.

RUP Phase: Construction

- Focus: **Development**.
- Source code is written and system functionality is implemented.
- **Unit, integration, and system testing** are performed.
- Main deliverable: **Working software system**.
- Focus on meeting quality standards and preparing for transition.
- Key outputs:
  - Working system build

RUP Phase: Transition

- Focus: **Deployment**.
- Final version of the product is released to users.
- Activities:
  - Beta testing
  - User training
  - Final documentation updates
- Bugs and defects are fixed based on user feedback.
- Goal: Ensure software is ready for production use.
- Key outputs:
  - Delivered system in use

Advantages of RUP

- RUP provides good **documentation** and completes the process thoroughly.
- Offers strong **risk-management support**.
- Encourages **component reuse**, which reduces overall development time.
- Availability of good **online support** such as tutorials and training.

Disadvantages of RUP

- Requires a **team of expert professionals** due to its complexity.
- Considered a **complex and not properly organized** process.
- High **dependency on risk management**.
- **Reintegration** during iterations can be difficult and time-consuming.

UML in RUP – Role of Diagrams

RUP Phase	UML Diagram(s) Used	Purpose
Inception	1. Use Case Diagram	Identify actors and system functionality; define project scope and goals.
Elaboration	1. Class Diagram 2. Activity Diagram 3. Sequence Diagram 4. Statechart Diagram	Define architecture; model workflows and logic; validate object behavior and interactions.
Construction	1. Class Diagram 2. Sequence/Communication Diagram 3. Component Diagram	Guide implementation; define object interactions and modular structure.
Transition	1. Deployment Diagram 2. Component Diagram	Describe physical deployment; finalize configuration and system rollout.

UML in RUP — Why Diagrams Repeat

Core idea: RUP is *iterative* ⇒ diagrams start rough, then are refined.

Phases (one-liners)

- **Inception**: scope — *what & who.*
- **Elaboration**: architecture/behavior — *how it works.*
- **Construction**: implementation — *how we build.*
- **Transition**: deployment — *where it runs.*

Same diagram may appear in multiple phases, but with ***different level of detail***

Inception Phase — UML Diagrams

Main diagram: Use Case

- Identify actors and system goals.
- Define scope of the system.
- *Example:* Student interacts with Admission System.

Why here?

- Early stage: only need to know *who* uses the system and *what* they need.

Elaboration Phase — UML Diagrams

Diagrams used:

- **Use Case (refined)** — add flows, alternate scenarios.
- **Class Diagram** — sketch main classes (Student, Course, Teacher).
- **Activity Diagram** — model workflows (e.g., Register for Course).
- **Sequence Diagram** — show interactions (Student → Controller → DB).
- **State Diagram** — lifecycle (Course: Planned → Open → Closed).

Purpose: capture architecture and behavior, remove risks before coding.

Construction Phase — UML Diagrams (Part 1)

Class Diagram (again)

- Now detailed for coding: attributes, methods, visibility.
- *Example:* `Student{name:String, rollNo:int}` with `registerForCourse()`, `payFee()`.

Sequence/Communication Diagram (again)

- Now refined with error handling, alt flows, retries.
- *Example:* Fee payment → if payment fails, system retries or shows error.

Construction Phase — UML Diagrams (Part 2)

Component Diagram (Construction)

- Define code modules and their interfaces.
- *Example:* Admission UI, StudentService, ResultService, Repository, Database.

Why not Activity/State here?

- They are analysis diagrams finalized in Elaboration.
- In Construction, developers implement them — no need to redraw unless behavior changes.

Transition Phase — UML Diagrams

Component Diagram (again)

- Now represents release packaging.
- *Example:* StudentService v1.0.jar, ResultService v1.0.jar, Web UI bundle.

Deployment Diagram

- Shows runtime environment.
- *Example:* Nginx (Web) $\longleftrightarrow$ App Node (StudentService, ResultService) $\longleftrightarrow$ DB Node (MySQL).

Quick Recap — UML in RUP

- **Inception** → Use Case (scope).
- **Elaboration** → Class/Activity/Sequence/State (behavior & architecture).
- **Construction** → Detailed Class, refined Sequence, Component (code-ready).
- **Transition** → Final Component (release) & Deployment (runtime).

Same diagram in different phases = ***different level of detail and purpose.***

RUP vs SDLC: Key Differences

Aspect	**SDLC**	**RUP**
Nature	General framework (concept)	Specific methodology (implementation of SDLC)
Phases	Requirement, Design, Coding, Testing, Deployment	Inception, Elaboration, Construction, Transition
Approach	Linear (Waterfall) or Iterative (Agile/Spiral)	Always iterative & incremental
Flexibility	Depends on chosen model	Built-in adaptability through iterations
Focus	Defines *what* to do	Defines *how* to do it (use-cases, risk-driven)

Case Study: College Management System

Scenario: A university wants to develop a system to handle:

- Student admissions
- Attendance tracking
- Examination results
- Fee payment

SDLC Perspective:

1. Requirement Analysis – talk to faculty, admin staff, students
2. Design – prepare database schemas, UI mockups
3. Implementation – developers write code for modules
4. Testing – test admission module, result processing
5. Deployment – install system for real use
6. Maintenance – fix bugs, update features

SDLC tells us ***what steps must be followed*** *in any software project.*

RUP Perspective:

1. **Inception** – Define project scope: "Admissions + results in phase 1"
2. **Elaboration** – Analyze risks: "Scalability for 10,000 students, data security"
3. **Construction** – Build admission + result modules first, test with small student group
4. **Transition** – Deploy incrementally: admissions this semester, attendance next semester

RUP shows ***how to carry out SDLC steps*** *iteratively and risk-driven.*

Procedural vs Object-Oriented Programming

Procedural Oriented Programming	**Object-Oriented Programming**
Program is divided into small parts called **functions**.	Program is divided into small parts called **objects**.
Follows a **top-down** approach.	Follows a **bottom-up** approach.
No access specifiers.	Has access specifiers like `private`, `public`, `protected`, etc.
Adding new data and functions is **not easy**.	Adding new data and functions is **easy**.
Does not support data hiding; hence, **less secure**.	Supports data hiding; hence, **more secure**.
Overloading is **not possible**.	Overloading is **possible**.
No concept of **data hiding and inheritance**.	Uses the concept of **data hiding and inheritance**.
Function is more important than data.	Data is more important than function.
Based on the **unreal world**.	Based on the **real world**.
Used for designing **medium-sized** programs.	Used for designing **large and complex** programs.

Procedural vs Object-Oriented Programming

Procedural Oriented Programming	**Object-Oriented Programming**
Uses the concept of **procedure abstraction**.	Uses the concept of **data abstraction**.
Code reusability is **absent**.	Code reusability is **present**.
Examples: C, FORTRAN, Pascal, BASIC, etc.	Examples: C++, Java, Python, C#, etc.

Client-Server Communication

- **Client-Server Communication** refers to the exchange of data and services among multiple machines or processes.
- One process acts as a **client**, requesting data or services, while another acts as a **server**, providing them.
- This model is widely used in **Distributed Systems**, **Internet Applications**, and **Networking Communication**.
- Communication occurs via various **protocols and mechanisms**.

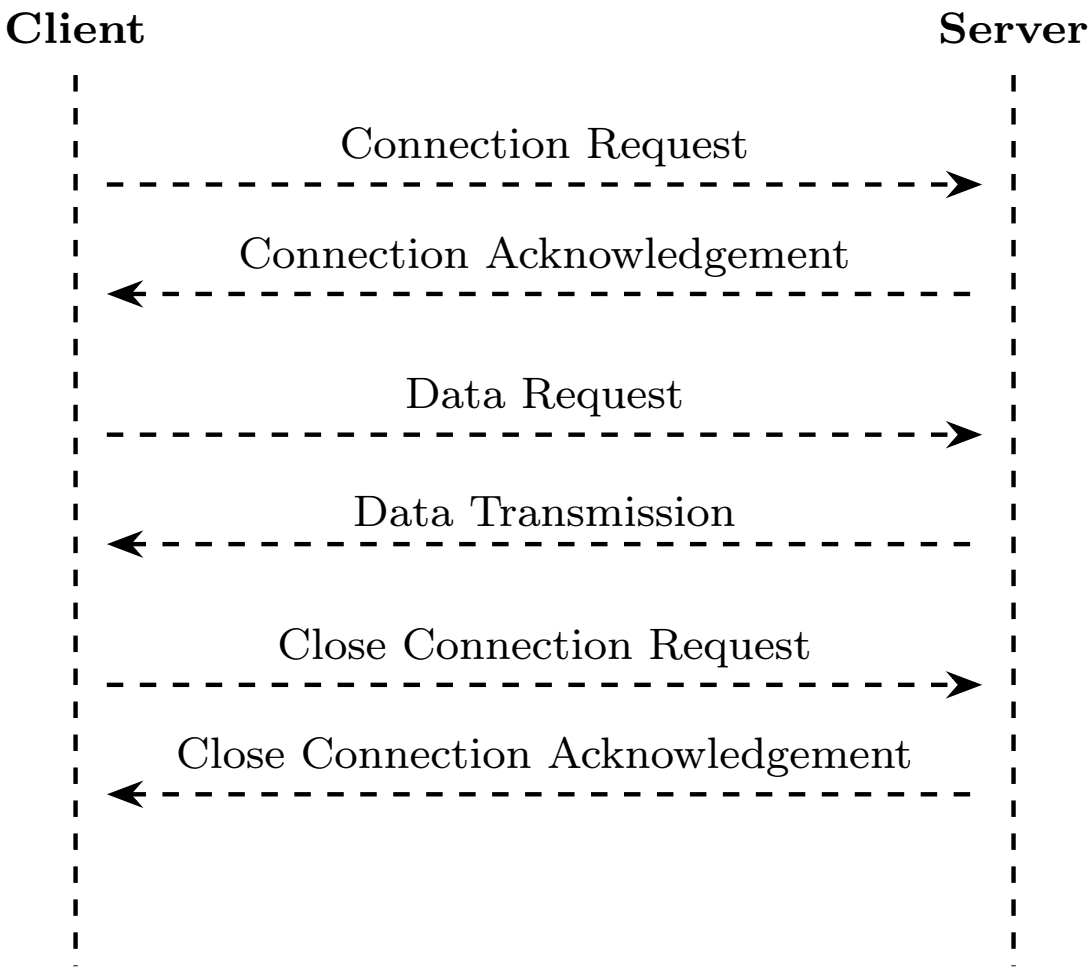
Different Ways of Client-Server Communication:

- Sockets Mechanism
- Remote Procedure Call (RPC)
- Message Passing
- Inter-process Communication (IPC)
- Distributed File Systems

Client-Server Communication Using Sockets

- Sockets are the **endpoints of communication** between two machines.
- They allow processes to communicate either on the **same machine** or over the **Internet**.
- Sockets enable a **bidirectional communication** channel between the client and server for data exchange.

Communication Using Sockets

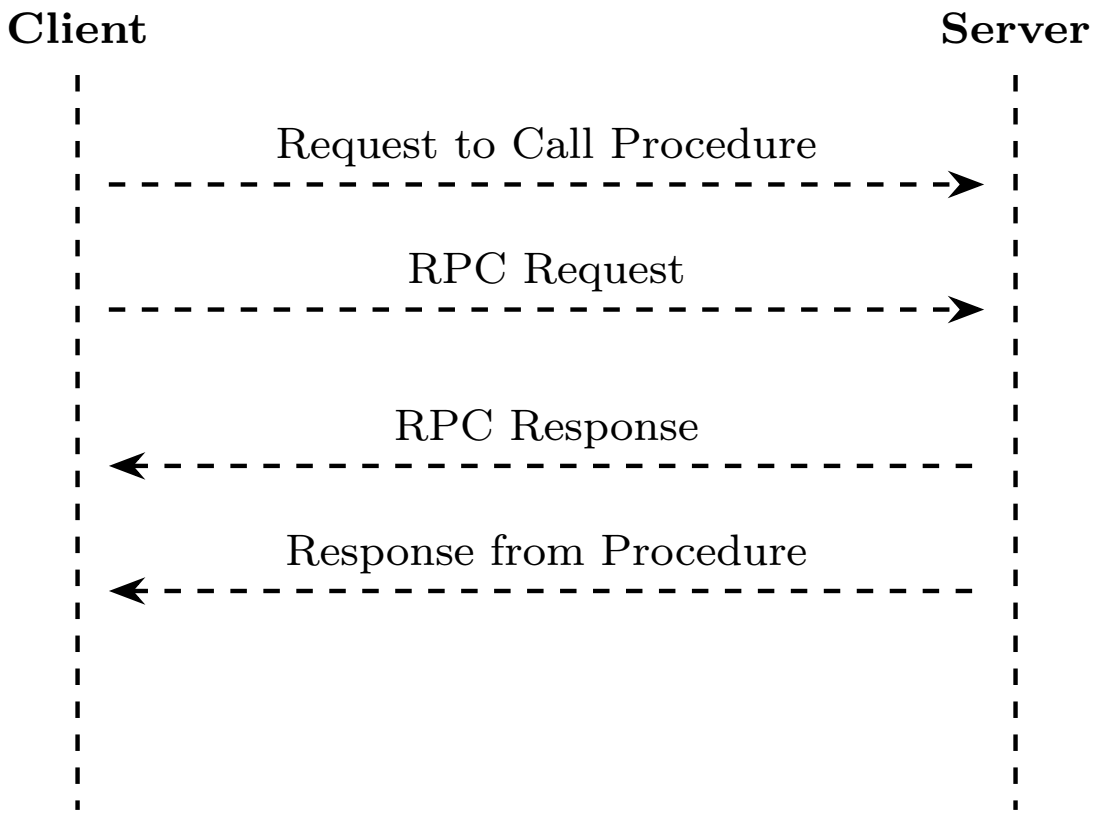


Client-Server Communication using Sockets

Client-Server Communication Using RPC

- Remote Procedure Call (RPC) is a **protocol** that allows a client to execute a procedure on a remote server as if it were a local call.
- RPC is commonly used in **Client-Server communication architecture**.
- It provides a **high level of abstraction** to the programmer.
- The client program issues a procedure call, which is translated into a message sent over the network to the server.
- The server executes the procedure and sends the response back to the client.

Remote Procedure Call Process



Remote Procedure Call Process

Client-Server Communication Using Message Passing

- Message Passing is a **communication method** where machines communicate by sending and receiving messages.
- It is commonly used in **parallel and distributed systems**.
- This method enables **data exchange among systems** without shared memory.
- It simplifies inter-process communication across machines.

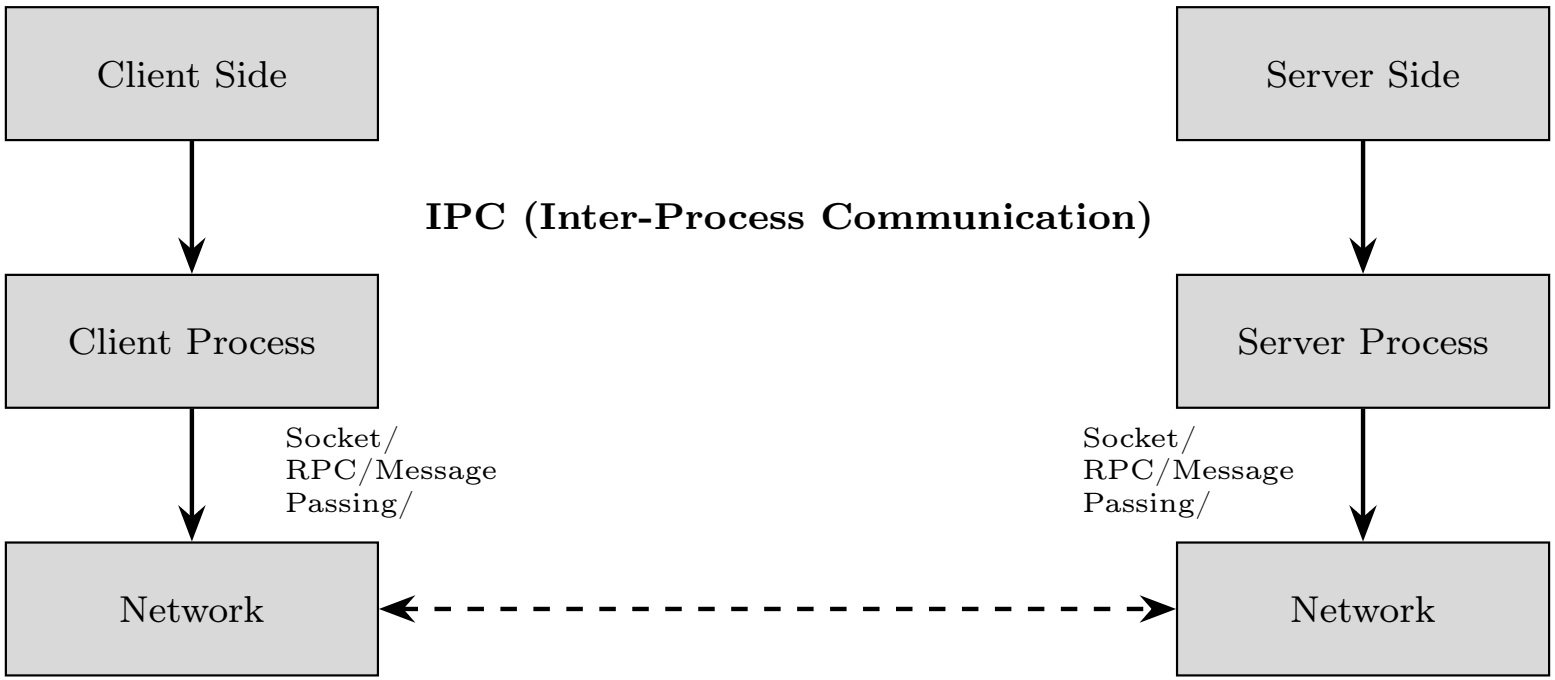
Message Passing Diagram

Client-Server Message Exchange via Message Passing

Client-Server Communication Using IPC

- Inter-Process Communication (IPC) allows **communication between processes** within the same machine.
- It enables **data sharing and synchronization** among different processes running concurrently on an operating system.
- IPC includes mechanisms like **shared memory**, **message queues**, **semaphores**, and **pipes**.
- It is fundamental to designing **cooperative multitasking** and **modular applications**.

IPC Diagram



Inter-Process Communication Process