UML

Dr. Ashish Sai

BCS1430

EPD150 MSM Conference Hall

Week 3 Lecture 2

Introduction to UML

Unified Modeling Language

- The Unified Modeling Language (UML) is a standardized modeling language enabling developers to specify, visualize, construct, and document artifacts of software systems.

> "The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems." – OMG (Object Management Group), 2003

Applications of UML

- **Software Design**: Models software's architecture of all sizes.
- **Business Process Modeling**: Visualizes workflows and operations.
- **Database Design**: Represents data models and relationships. *(We will use this in our Databases course.)*

Analysis and Design in UML

UML’s Role in Analysis:

- **Understanding the Problem Domain**: Visualize system requirements and actors.
- **Specifying Requirements**: Detail system interactions and flows.

UML’s Role in Design:

- **Planning the Solution**: Class and component diagrams clarify structure.
- **Defining System Architecture**: Deployment/package diagrams illustrate hardware/software elements.
- **Detailed Design**: State and interaction diagrams detail behavior and interactions.

UML as a Language - Overview

- Beyond Notation: UML isn’t just a set of diagrams.
- Today’s Overview: We’ll cover many UML diagram types.
- Practice: You’ll create UML models in labs/tutorials.

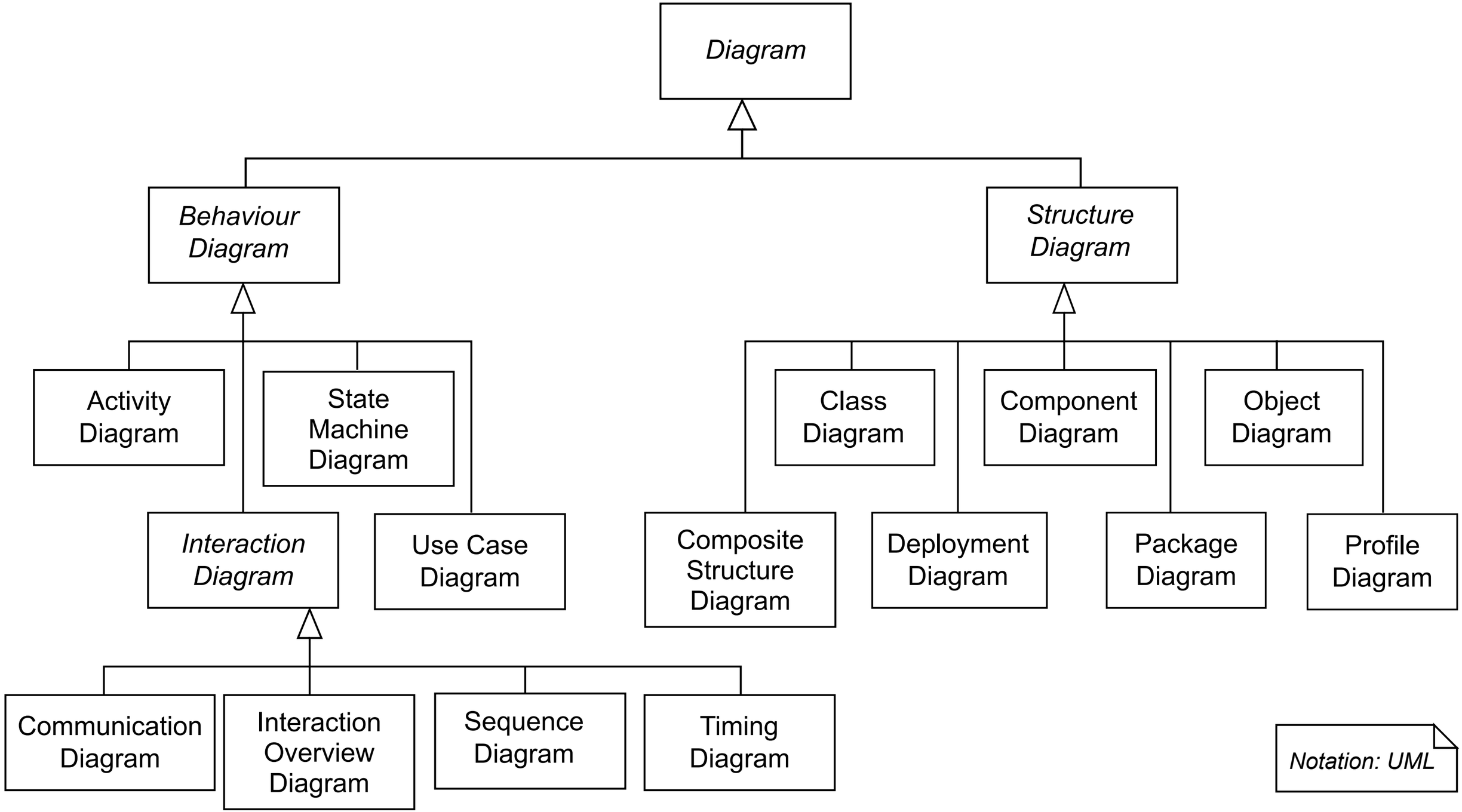


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Use Case Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Communication Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram
Notation: UML

Structural Diagrams - Overview

- Structural diagrams depict the static aspects of the system, showing how elements are organized and related.

Types of Structural Diagrams

1. **Class Diagram** – Classes, attributes, methods, relationships
2. **Object Diagram** – Class instances at a specific time
3. Component Diagram
4. Composite Structure Diagram
5. Deployment Diagram
6. **Package Diagram** – Element organization and dependencies
7. Profile Diagram

Behavioral Diagrams - Overview

- Behavioral diagrams represent dynamic aspects and system behavior over time.

Types of Behavioral Diagrams

1. **Use Case Diagram** – From an end-user perspective
2. **Activity Diagram** – Flow of control/activity over time
3. **State Machine Diagram** – States and transitions

Interaction Diagrams - Overview

- Subset of behavioral diagrams detailing how system elements interact.

Types of Interaction Diagrams

1. **Sequence Diagram** – Time-ordered message flow
2. Communication Diagram
3. Interaction Overview
4. Timing Diagram

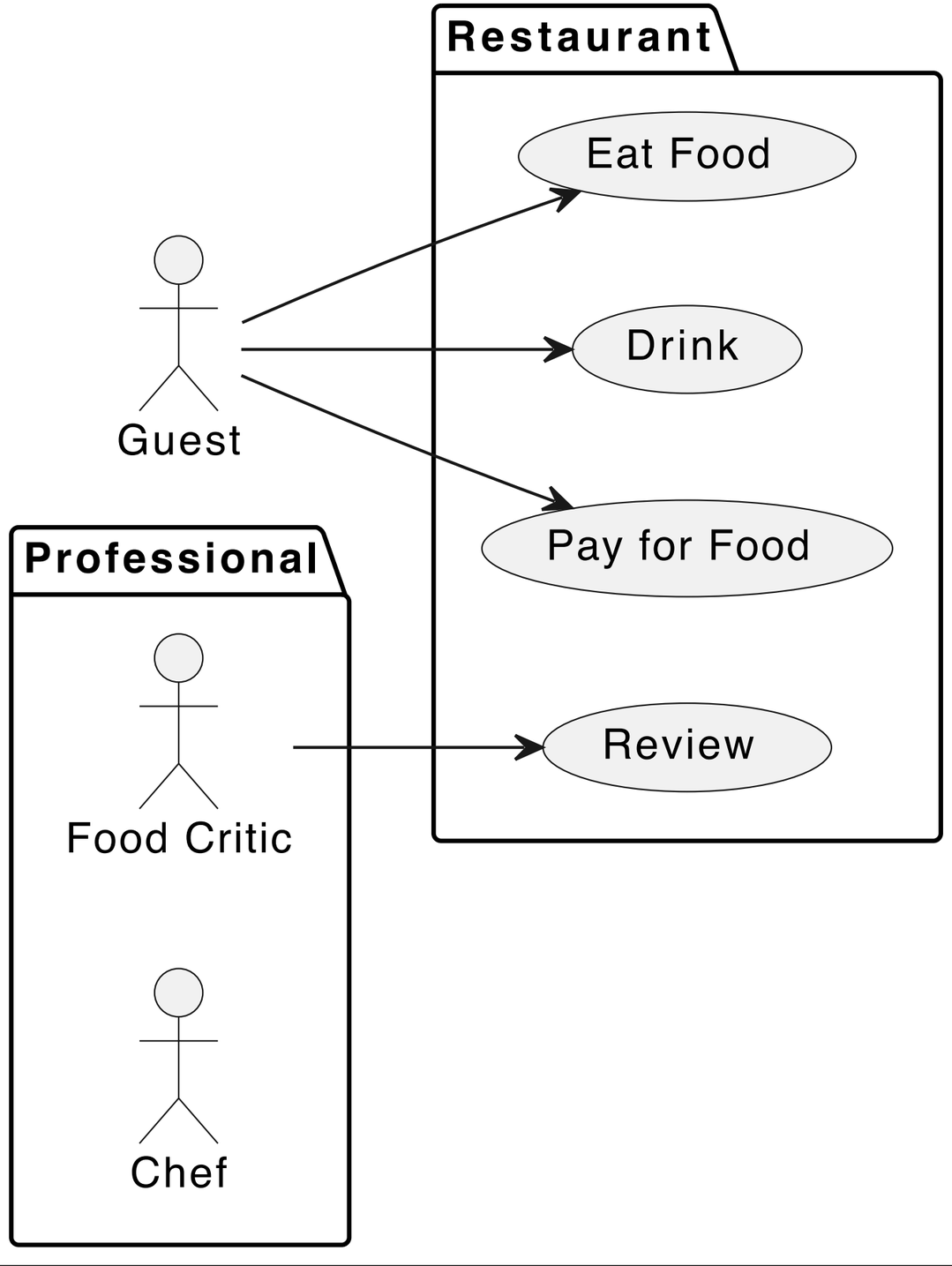
Use Case

Use Cases

- Narratives describing how actors (users/other systems) interact with a system to achieve a goal.
- **Why Use Cases?**
  - Understand/communicate functional requirements.

Components of a Use Case

- **Actors**: Entities (users, other systems, devices)
- **Scenarios**: Sequences of actions between actor + system
- **Goals**: End result the actor wants


Restaurant
Eat Food
Drink
Pay for Food
Review
Guest
Professional
Food Critic
Chef

Designing Use Cases

- **Clarity**: Use accessible, non-jargon language
- **Completeness**: Include normal + exceptional paths
- **Consistency**: Uniform detail/format across use cases

Importance of Clarity and Simplicity

Bad Use Case Example (lack of clarity):

- **Title**: System Authenticate User
- **Description**: User invokes system authentication subroutine, inputs credentials. System hashes input, compares against DB. On match, system initializes session.

Issues: Overly technical, unclear.

Improved Version:

- **Title**: User Logs In
- **Description**: The user enters username/password; system checks credentials and, if correct, grants access.

Ensuring Completeness

Bad Use Case Example (lack of completeness):

- **Title**: User Makes a Purchase
- **Description**: The user selects products and purchases them.

Issues: Overly simplistic, missing steps for payment, errors, etc.

Improved: Include detailed steps for product selection, payment method, error handling.

Maintaining Consistency

Bad Use Case Example (lack of consistency):

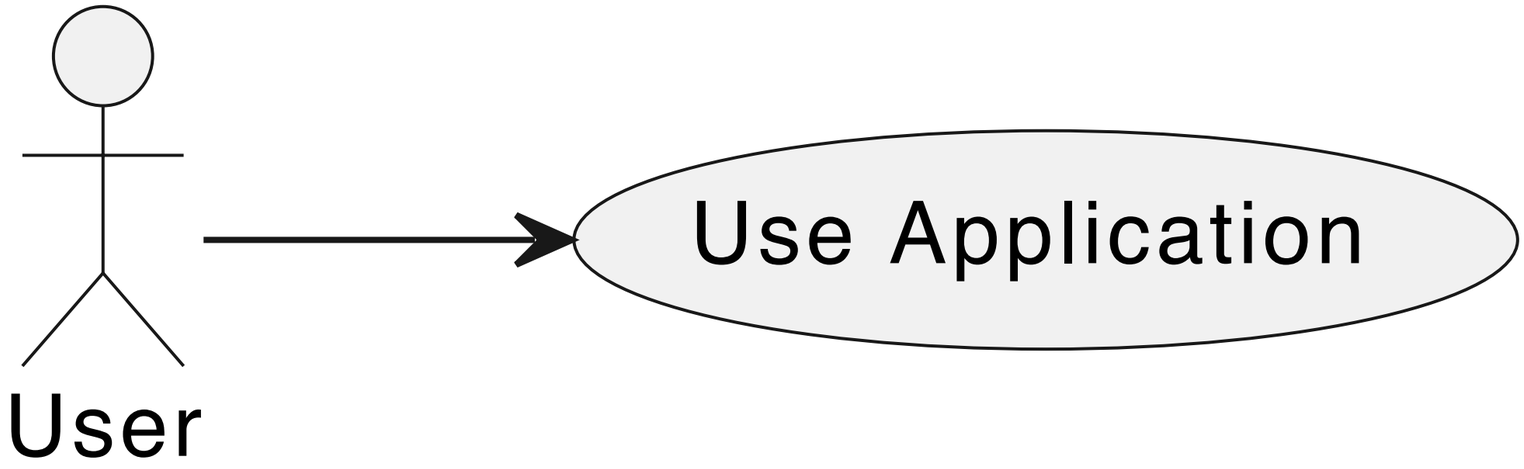
- **Title**: User Submits Feedback
- **Description**: A highly detailed, technical description with system-level operations + DB transactions.

Issues: Inconsistent format/detail with other use cases.

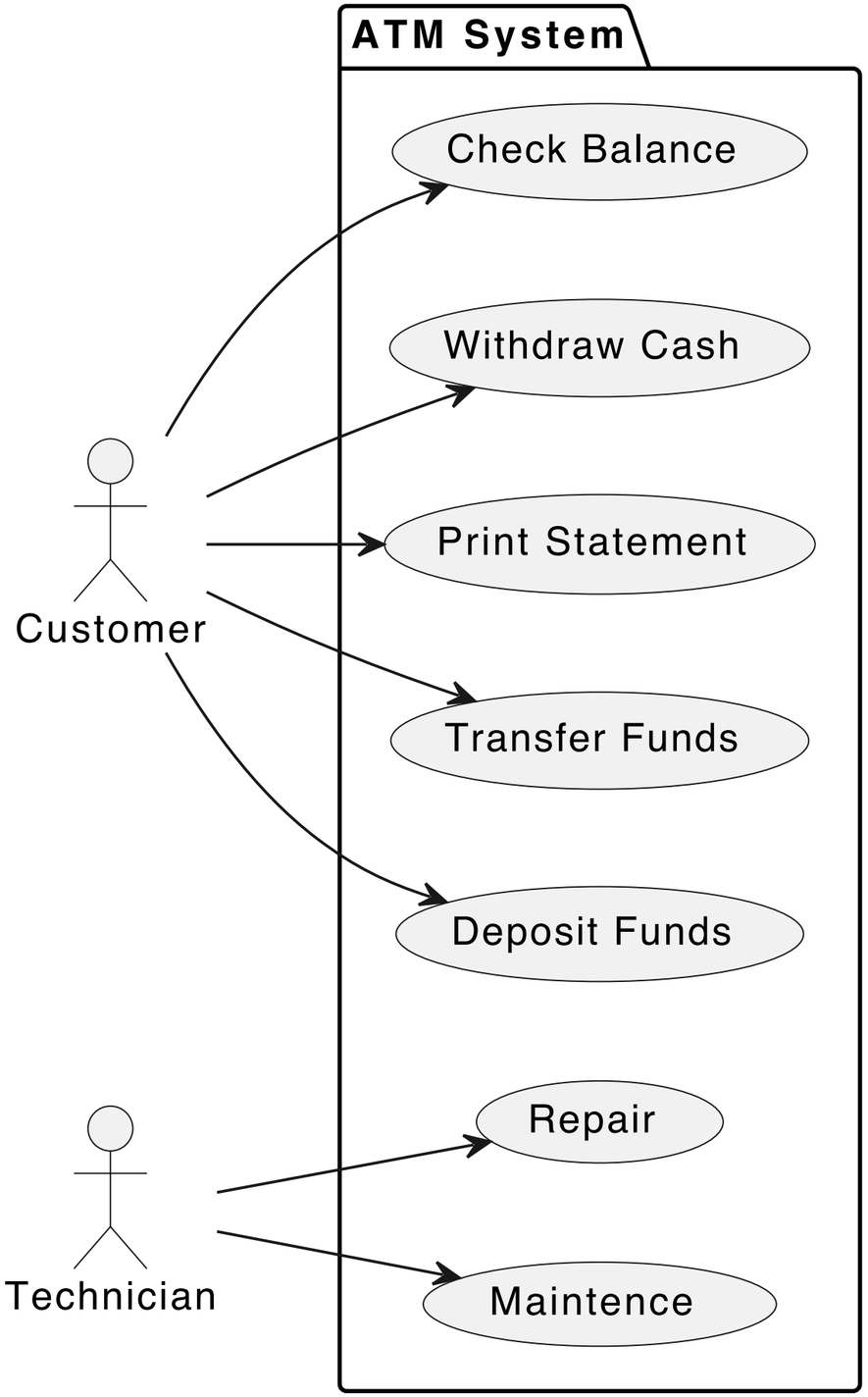
Improved: Align detail and language style with other use cases.

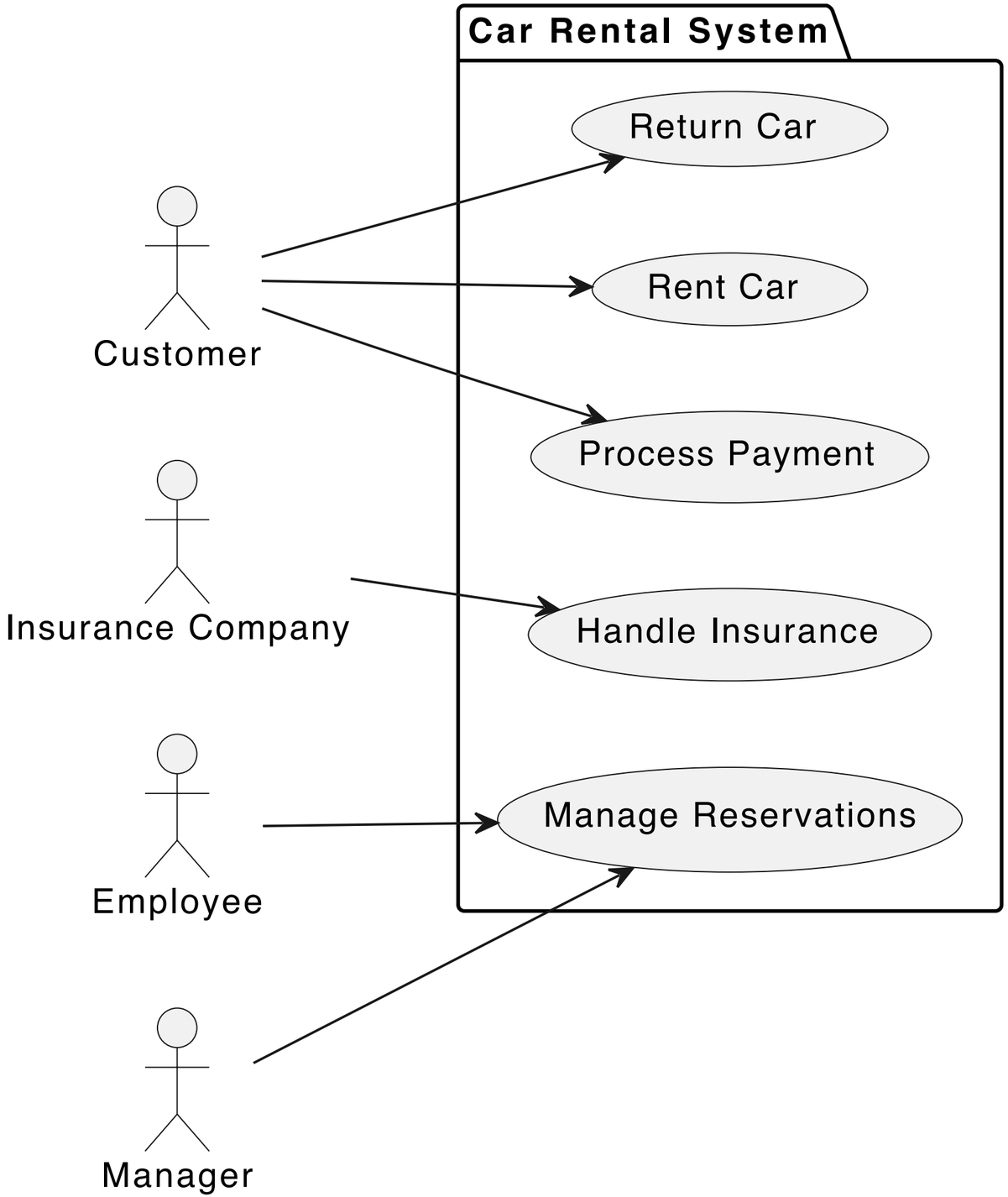
Use Case Diagrams

- **Visualizing Interactions**: Show relationships between actors + use cases
- **Identifying Relationships**: Use `include`, `extend`, generalization



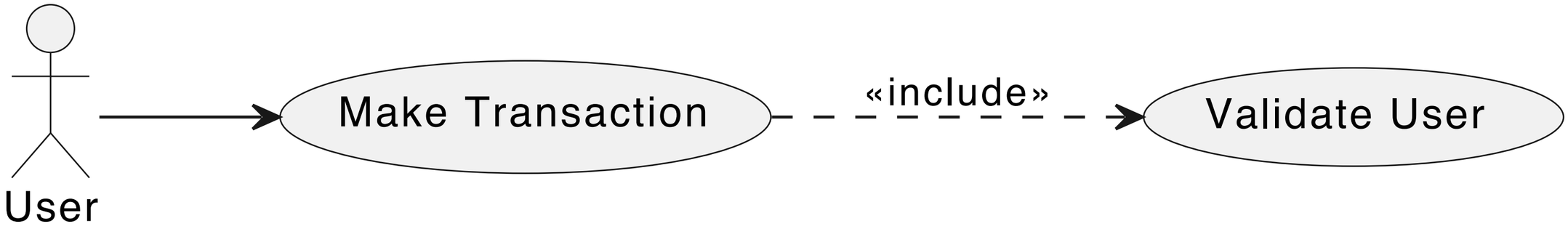
Example: ATM System Use Case Diagram



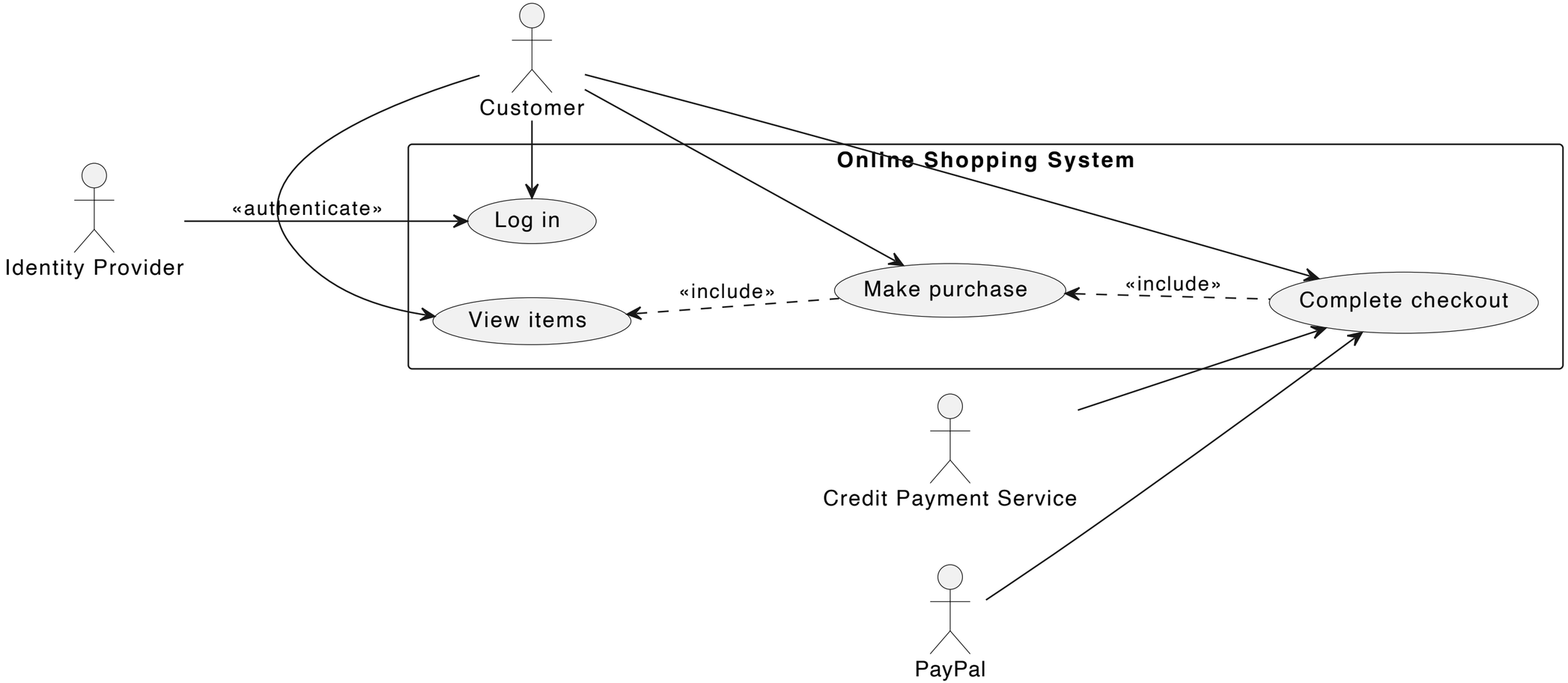


Example: Car Rental System

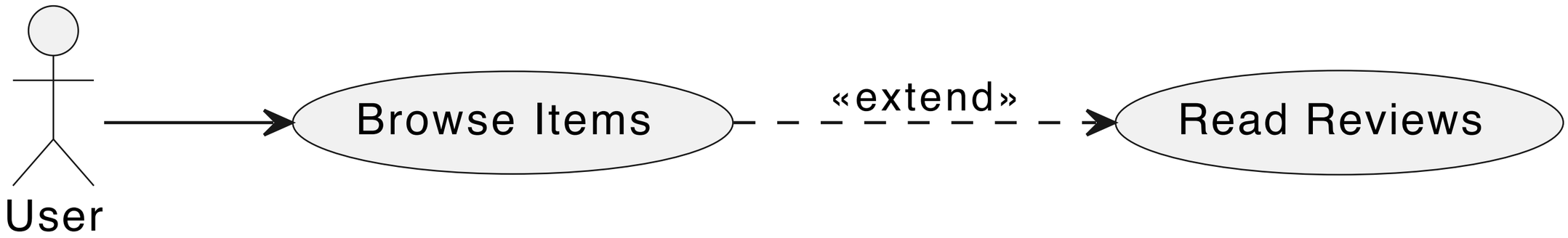
Use Case with 'Include' Relationship



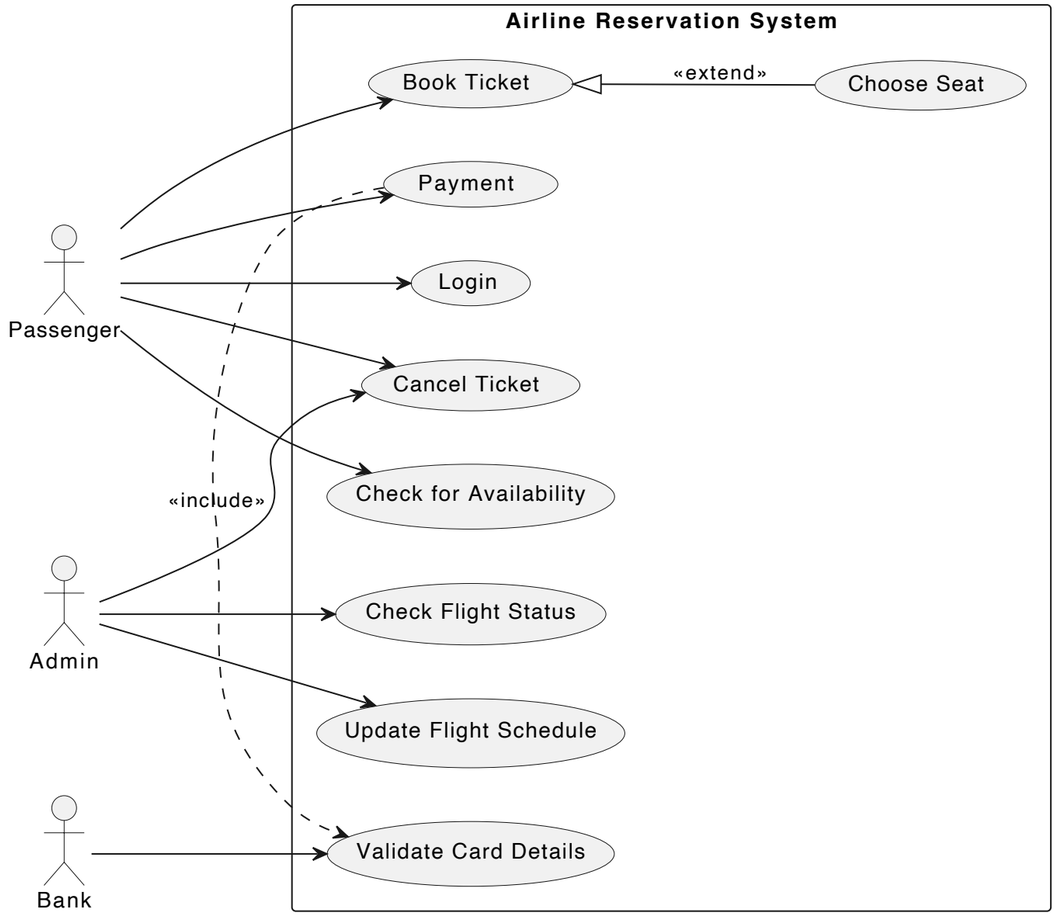
Example: Online Shopping



Use Case with 'Extend' Relationship



Example: Airline Reservation



Class Diagrams

Class Diagrams

- Depict classes, attributes, operations, and relationships among them (static view).

Utilizing Class Diagrams Effectively

- **Class**: Blueprint for objects, defining attributes + operations.
- **Attributes**: Characteristics of a class.
- **Operations**: Functions/methods belonging to a class.
- **Relationships**: How classes interact (associations, generalizations, dependencies).

Types of Relationships

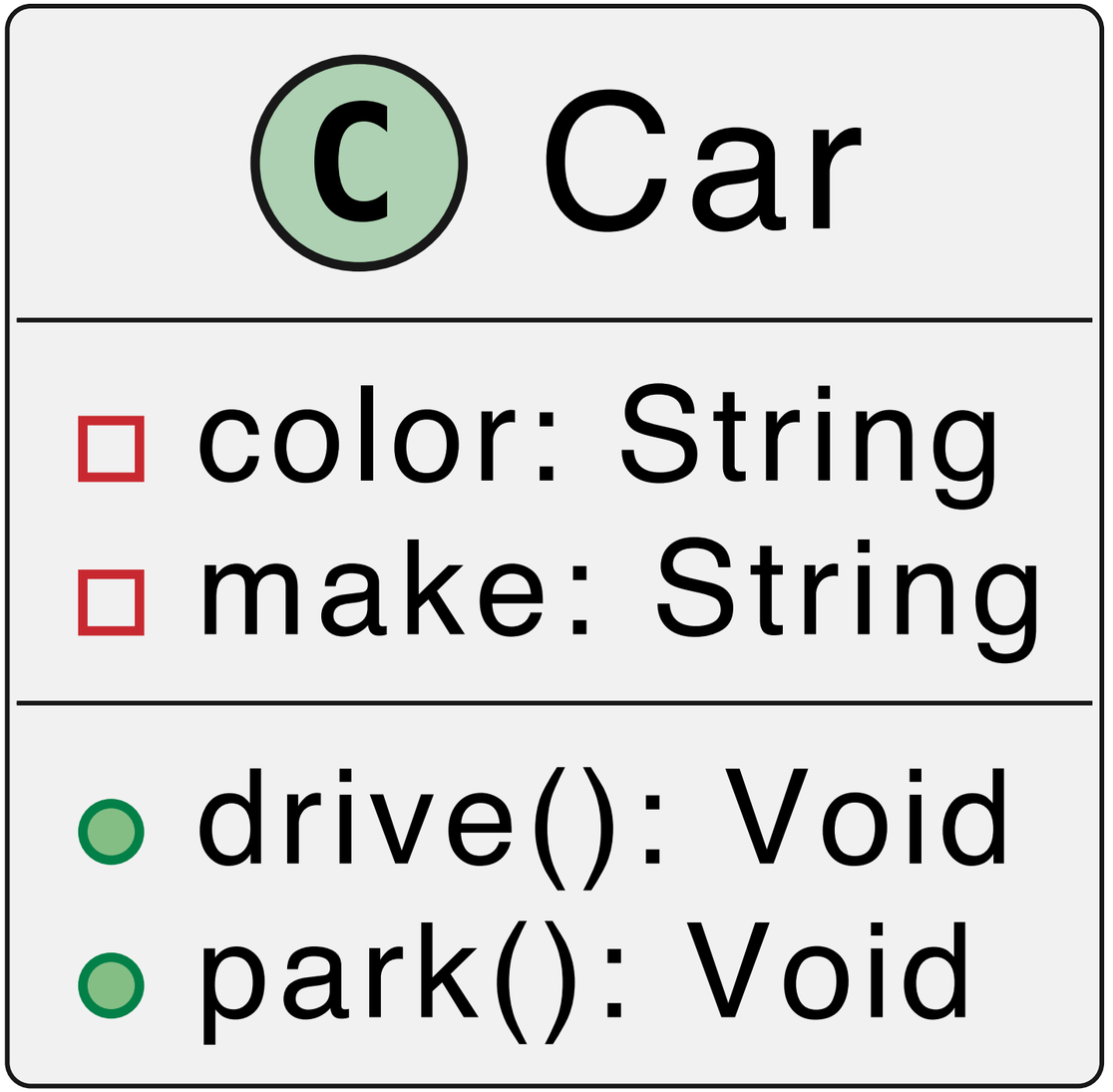
- Associations: Links between classes.
- Generalisations: Hierarchical parent-child class relationships.
- Dependencies: When one class uses another.

When to use class diagrams

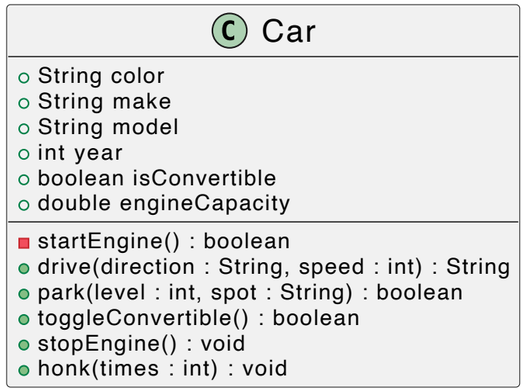
- **Early Development**: Outline system structure.
- **Documentation**: Provide a clear, maintainable system blueprint.

Class Example

```
public class Car {
    private String
color;
    private String
make;
    // ...
}
```



Classes

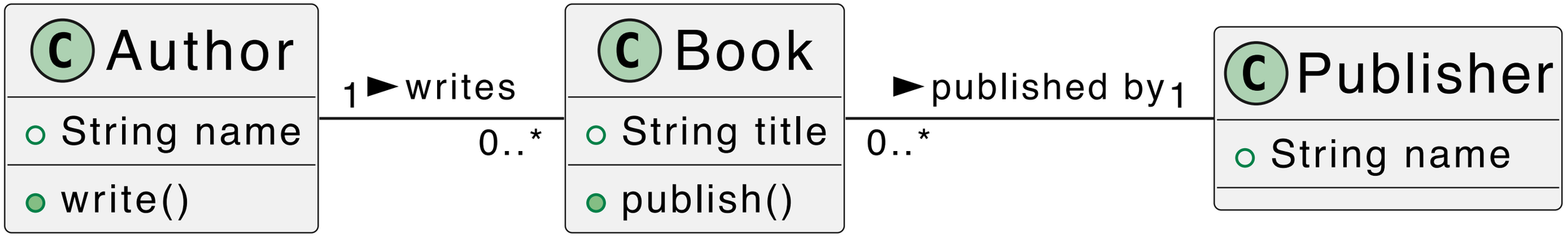



```
public class Car {
    private String color;
    private String make;
    // ...
    public Car(String color,
String make, ... ) {
        // Constructor
    }
    private boolean
startEngine() { ... }
    public String
drive(String direction, int
speed) { ... }
    // etc.
}
```

> This highlights the structure and capabilities of a `Car` class.

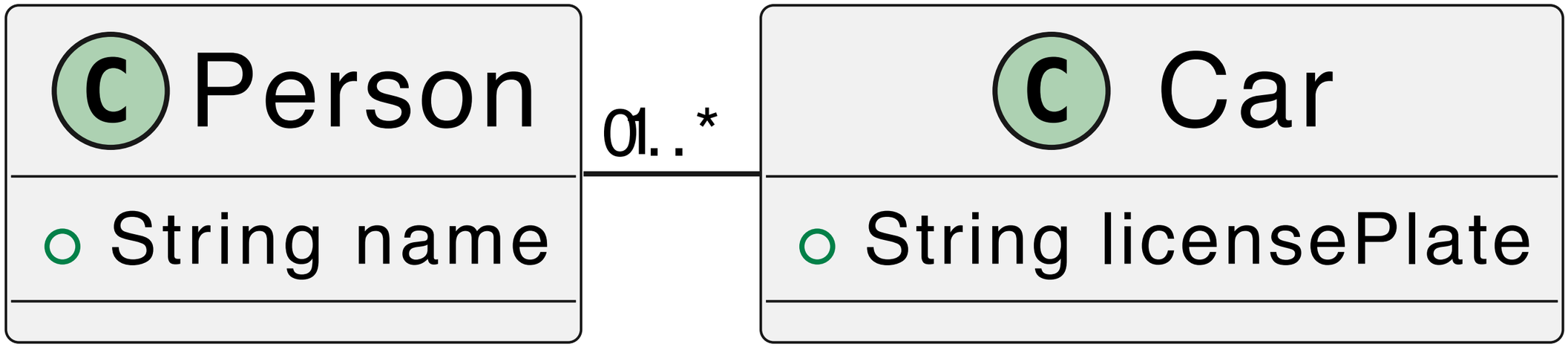
Associations: Basics of Associations

– Represent relationships between classes (1–1, 1–*, etc.)



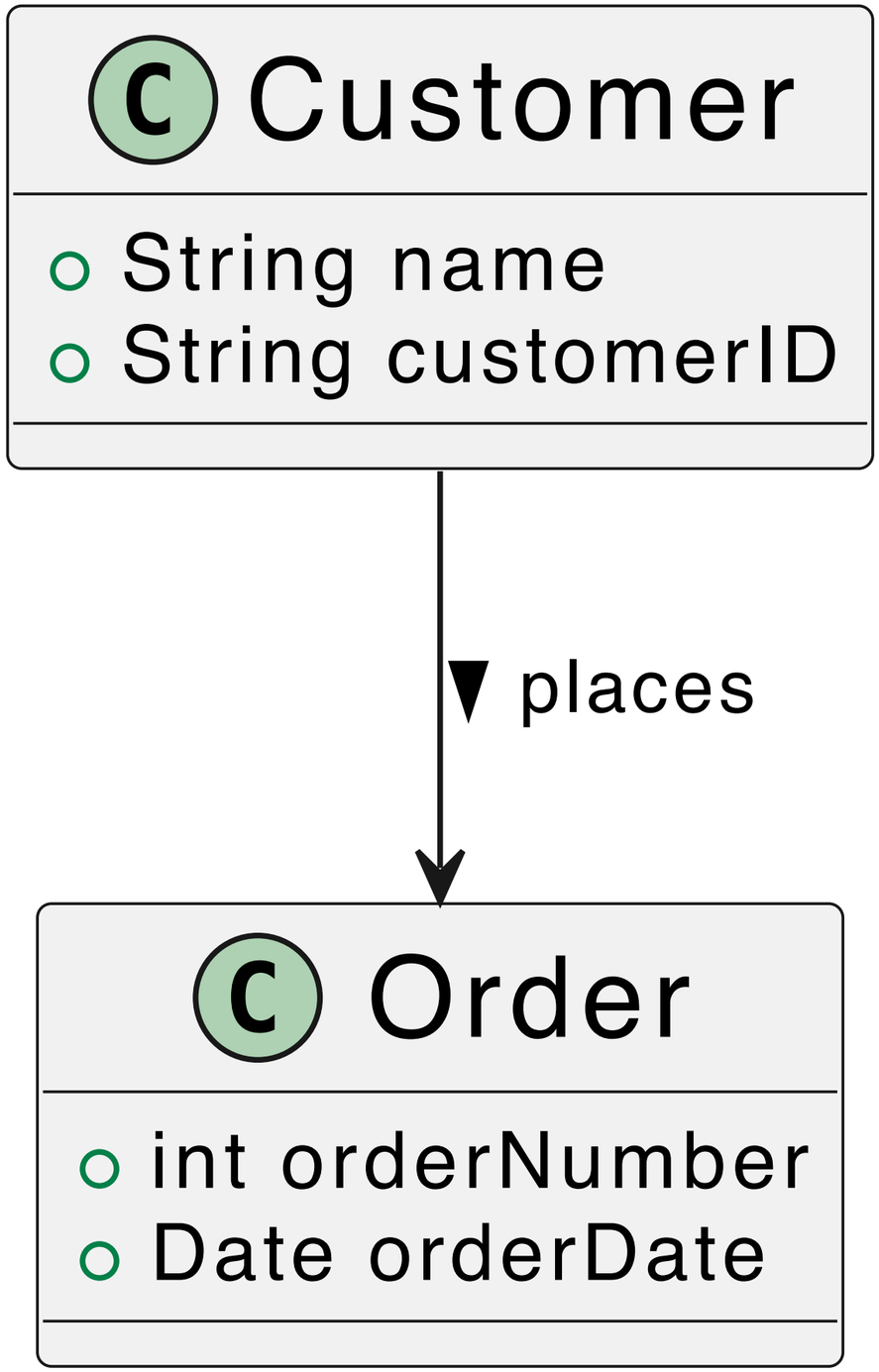
Associations: Multiplicity

- Multiplicity describes how many instances of one class can be associated with another.



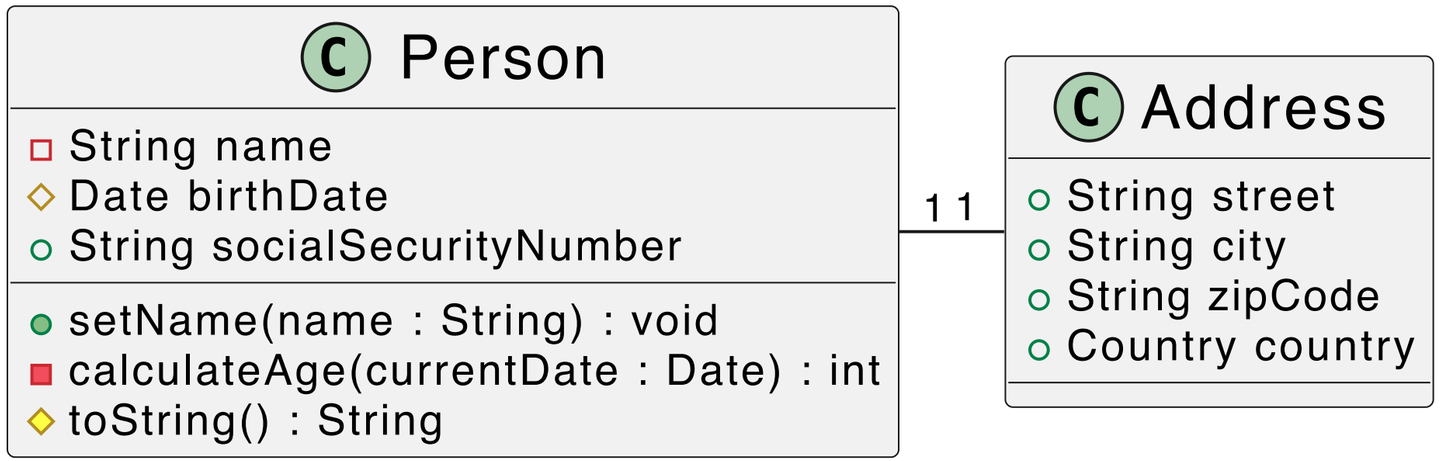
Associations: Navigability

Shows which class can “see” the other via an arrow.



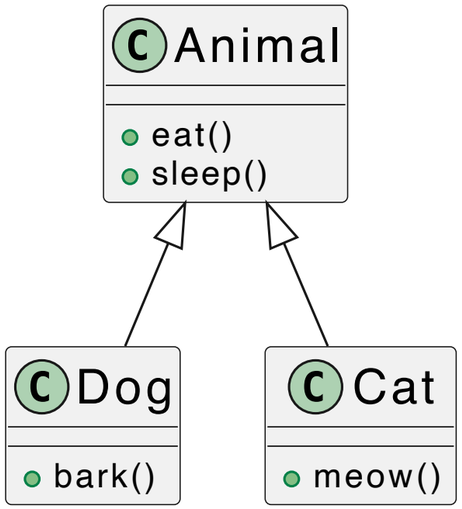
Attributes and Operations in Class Diagrams

- **Attributes**: May be simple or complex types, with visibility (+, -, #).
- **Operations**: Functions with parameters/return types, also with visibility.



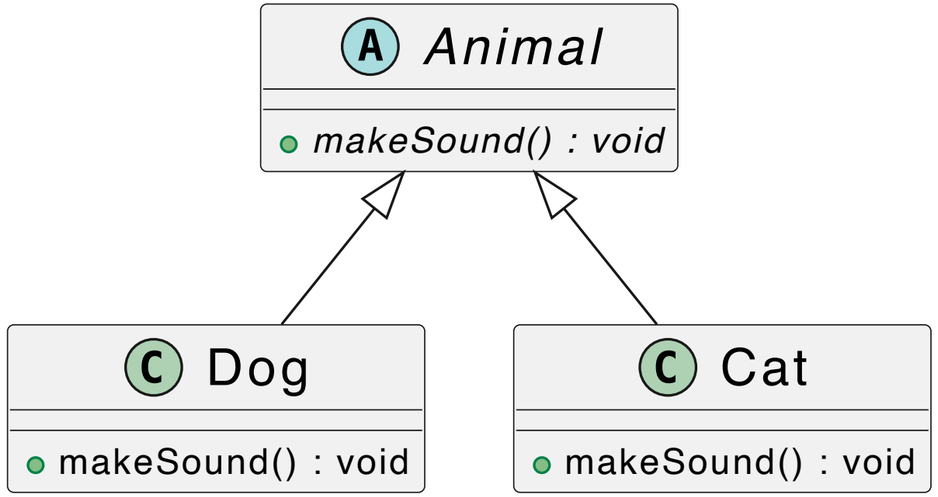
Inheritance in Generalisation

Generalization creates a parent class extended by child classes.



Polymorphism

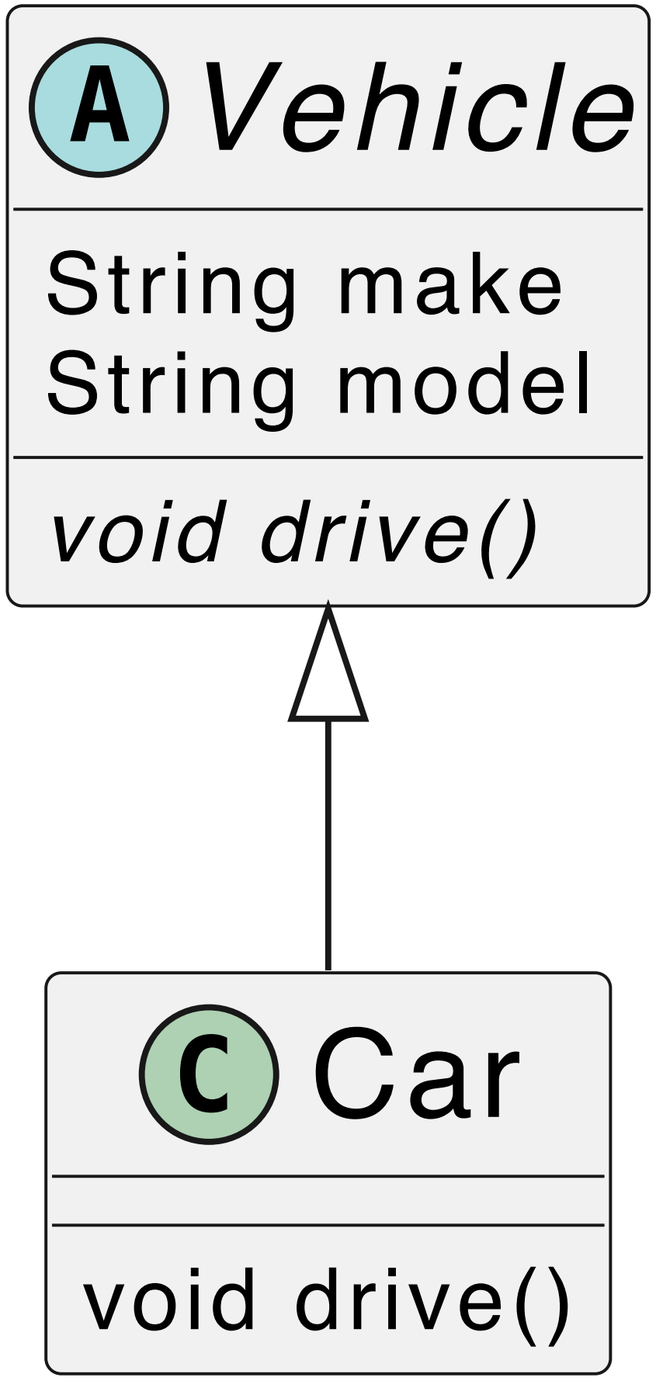
– Subclasses can be treated as instances of a superclass.

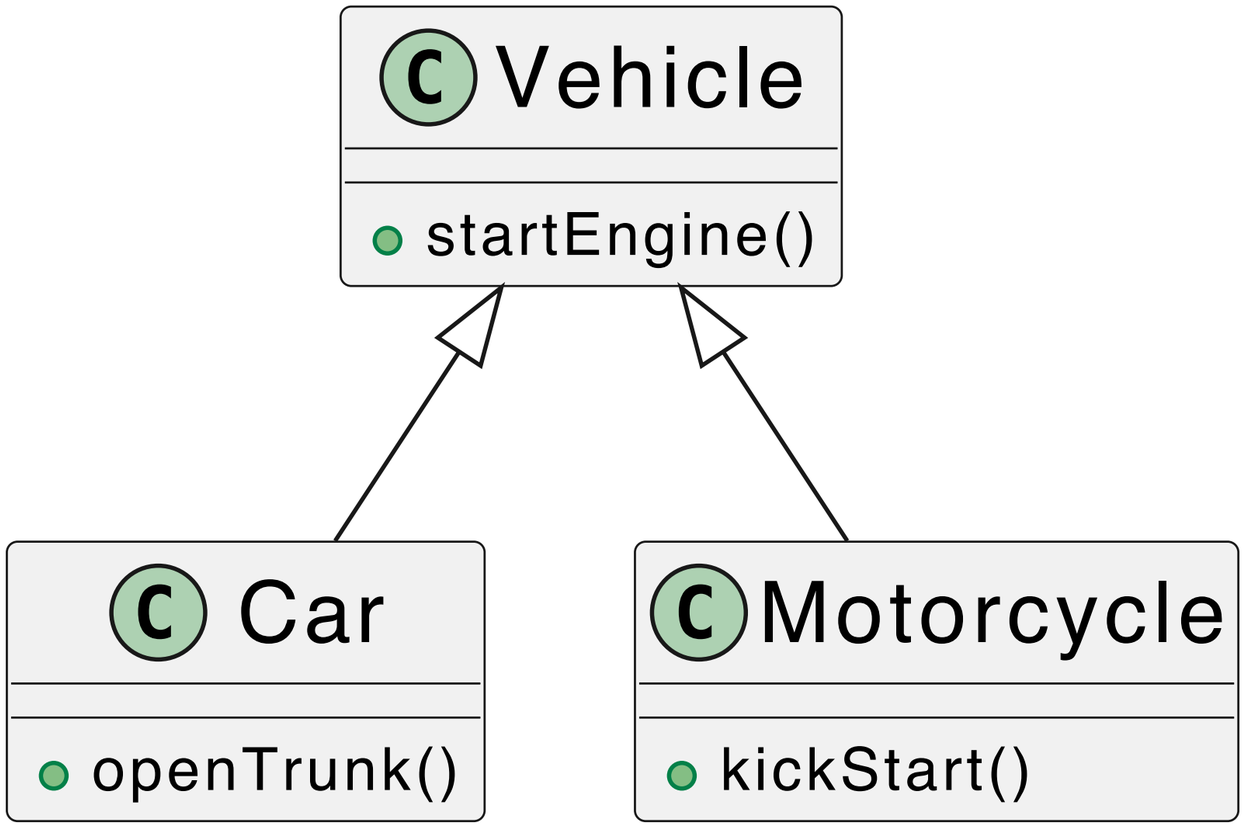


`Dog` and `Cat` override `makeSound()` from `Animal`.

Abstract Classes

- Cannot be instantiated, may have abstract methods.



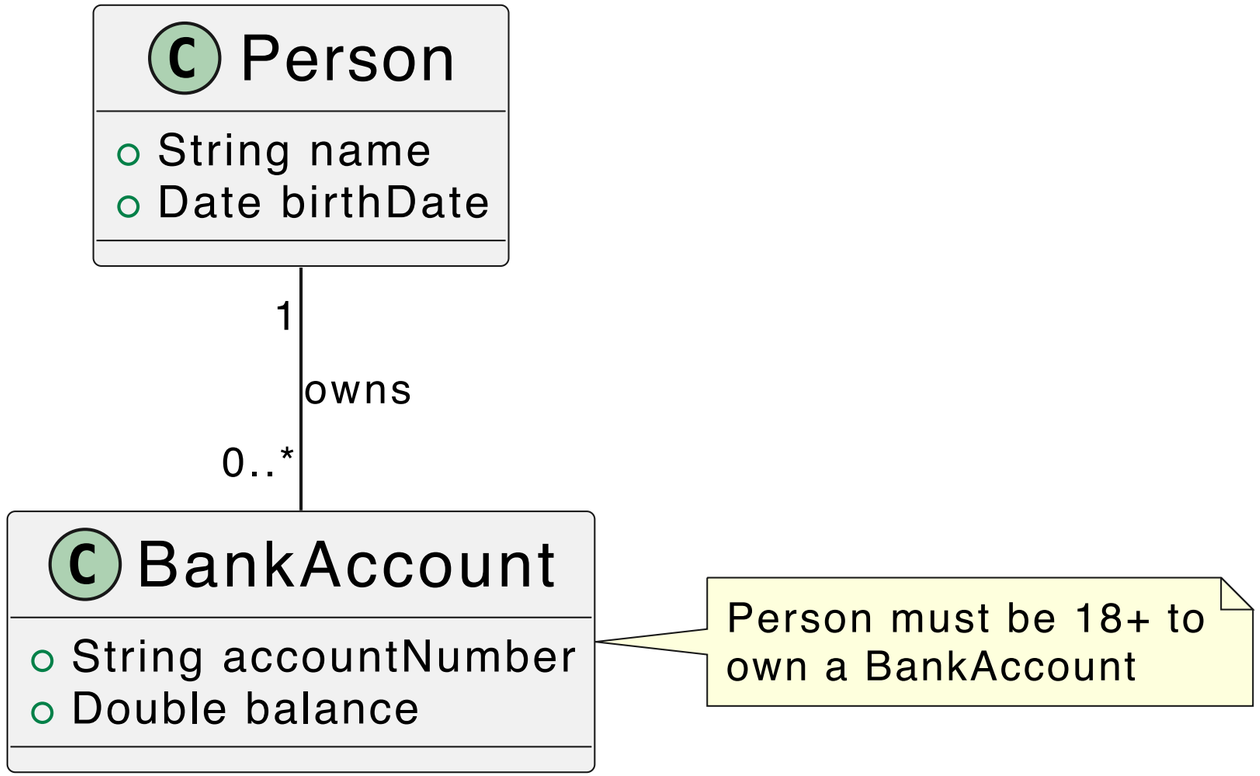


Benefits of Generalization

- Encourages reusability, easy maintenance (changes in superclass flow to subclasses).

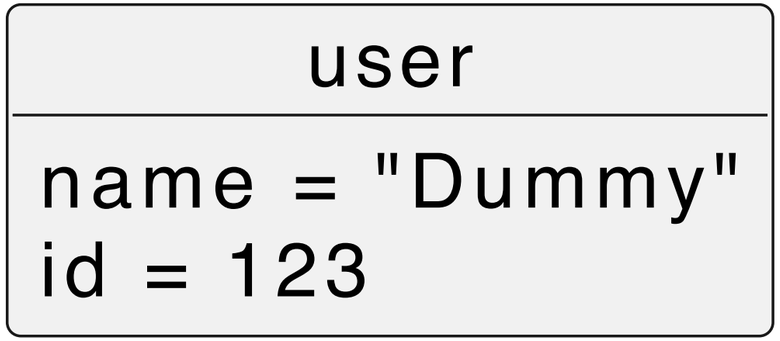
Using Constraints in Class Diagram

- Constraints ensure business rules or logic are met.



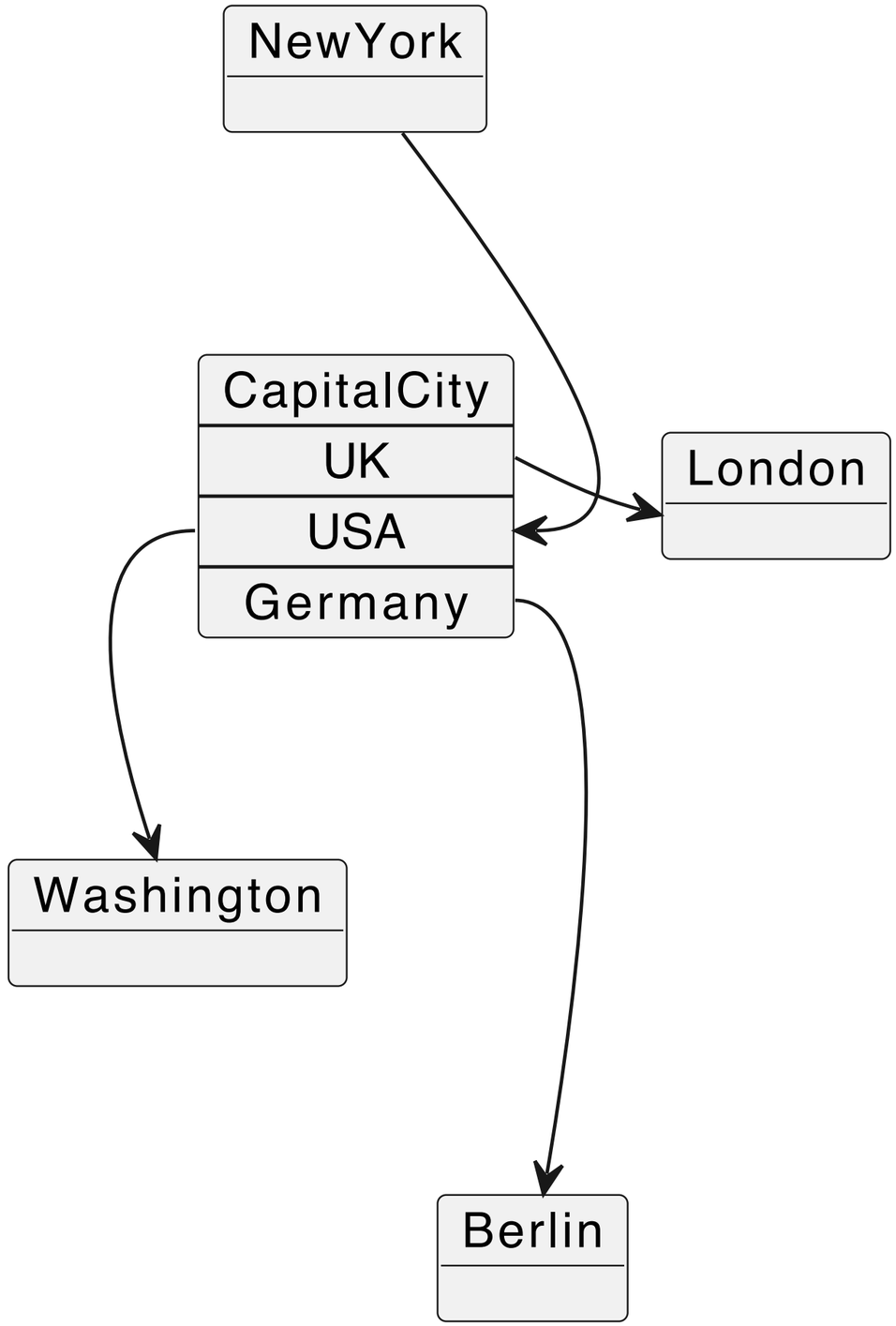
Object Diagrams

- Snapshots of system instances at a specific time.
- Similar to class diagrams but show objects (underlined names) and their links.



Object Diagrams Example

> Constraints + object diagrams together ensure design meets rules while showing a real-time snapshot of instances.



Interaction Diagrams

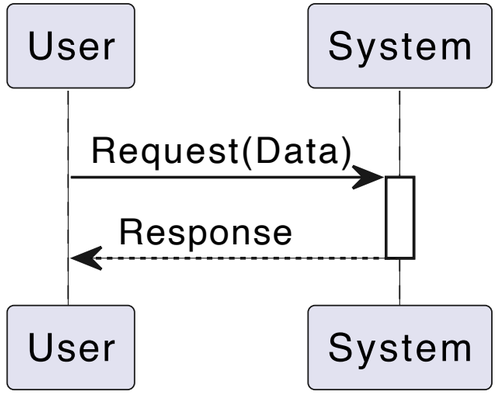
Interaction Diagrams

- **Purpose**: Show dynamic behavior via object interactions + message flows.
- **Types**:
  - **Sequence Diagrams**: Time-ordered messages
  - Collaboration Diagrams: Structural organization of object interactions

Sequence Diagrams Overview

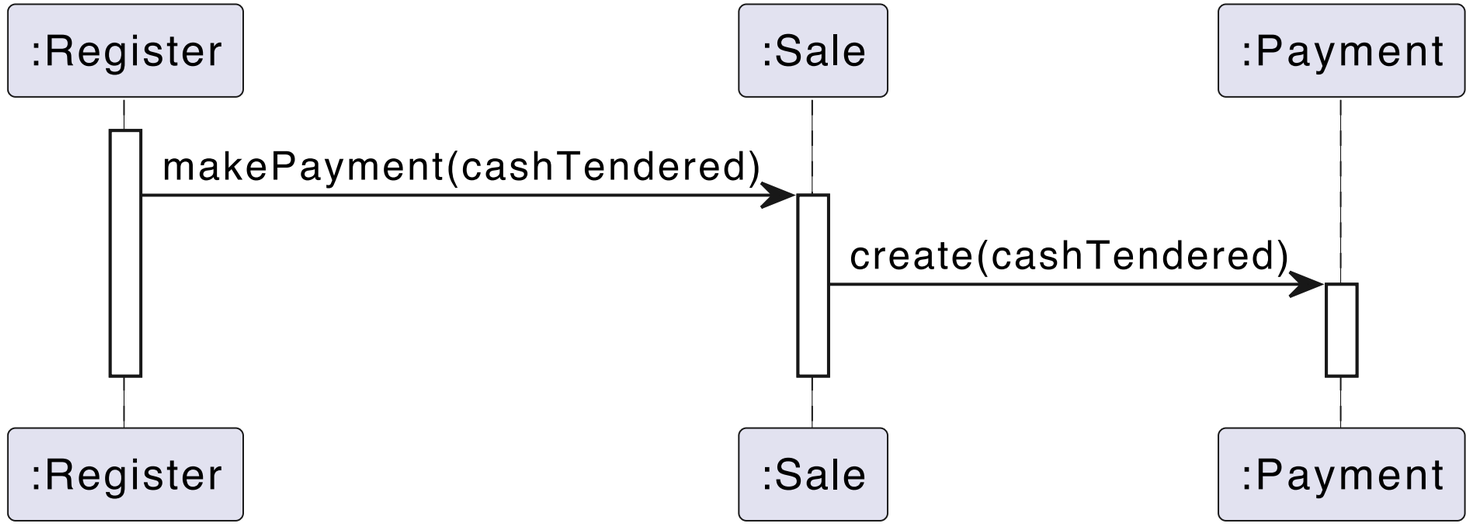
- Illustrate object interaction over time:
  - **Objects/Actors** (rectangles with lifelines)
  - **Messages** (arrows)
  - **Activation** (narrow rectangles on lifelines)

Sequence Diagrams



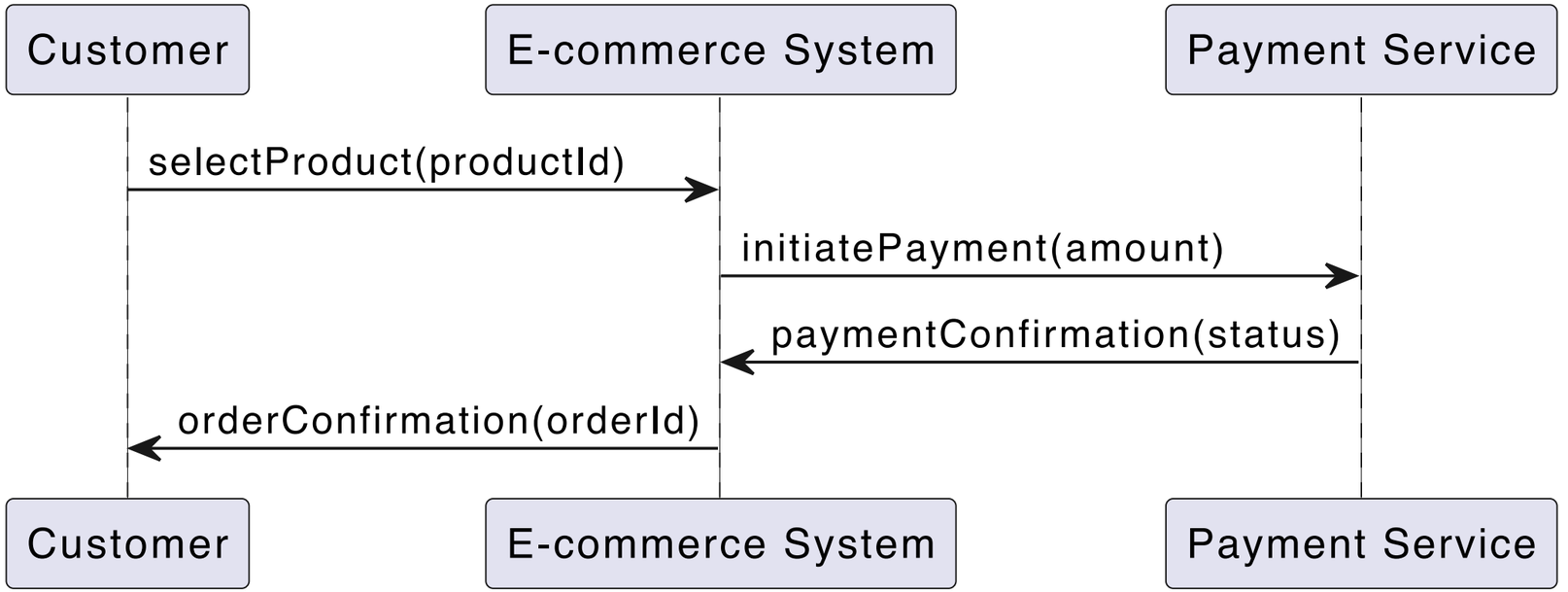

```
public class System {
    public String
processRequest(String data) {
return "Response"; }
}
public class User {
    public static void
main(String[] args) {
        System system = new
System();
        String response =
system.processRequest("Data");

System.out.println("Received
response: " + response);
    }
}
```

Reading A Sequence Diagram



> `Register` calls `Sale.makePayment()`, which then creates `Payment`.

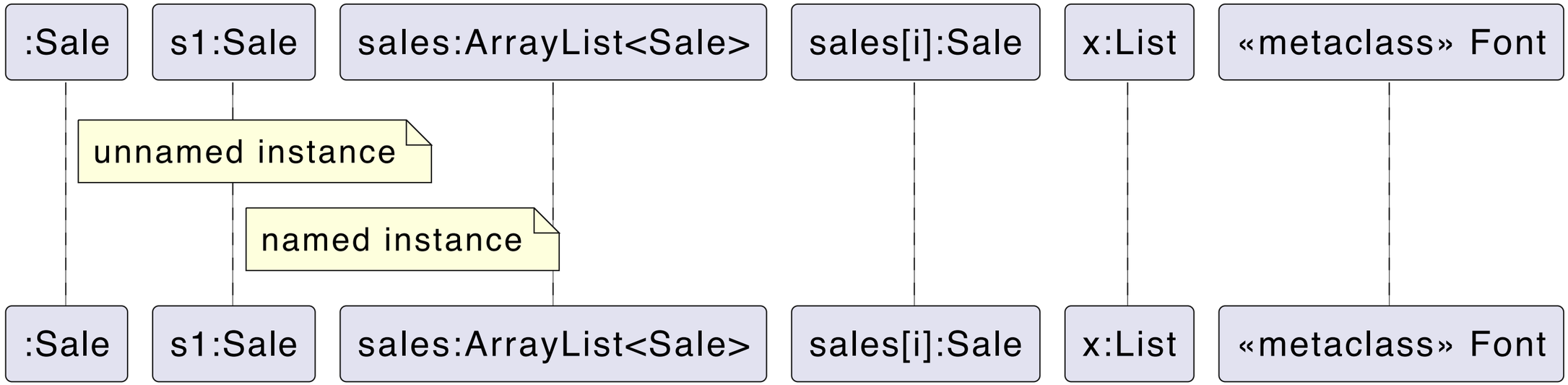
Sequence Diagrams: E-Commerce Example



> Demonstrates the flow of product selection, payment initiation, confirmation.

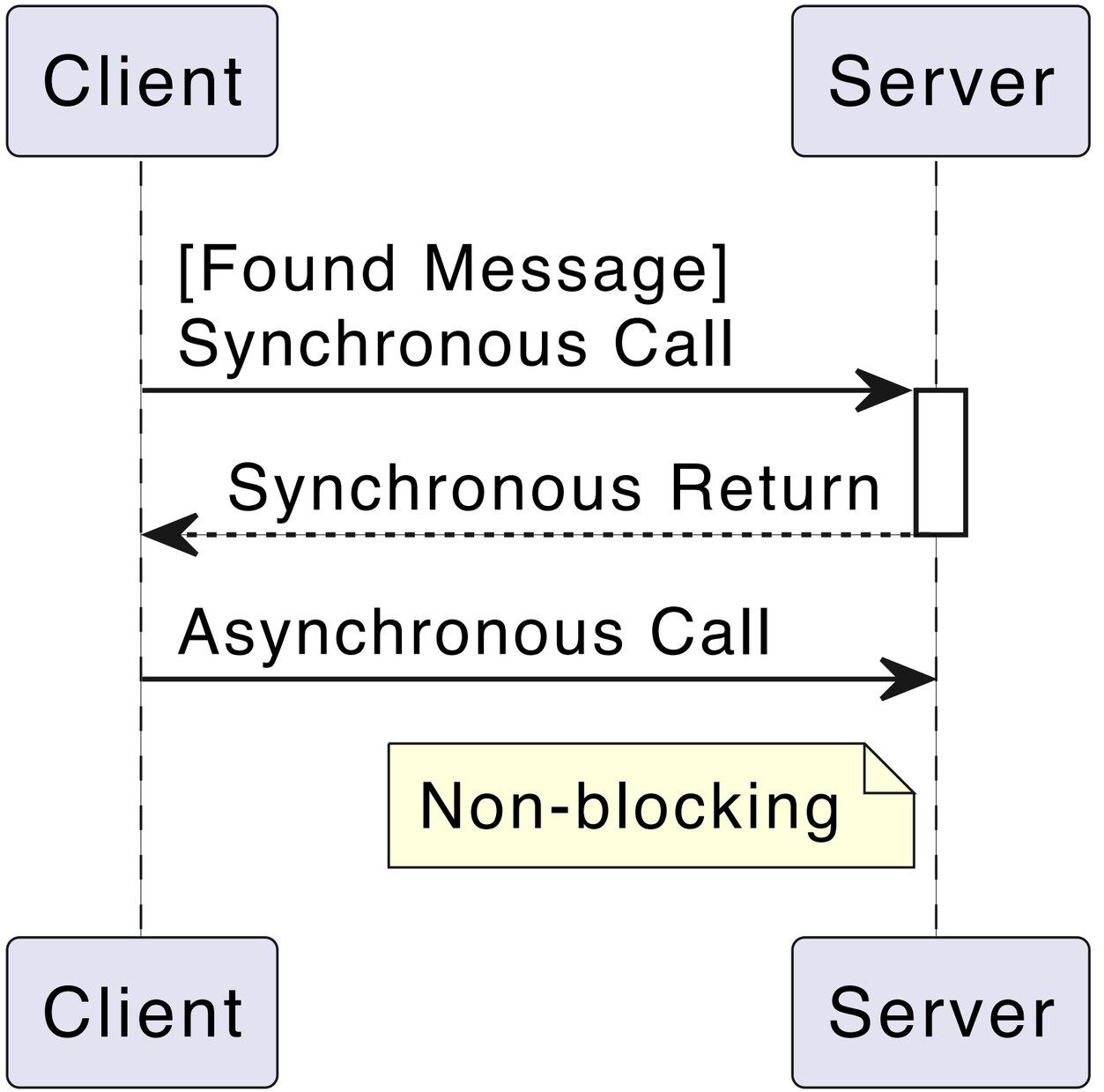
Sequence Diagrams: Lifeline Box Notation

- Each participant has a lifeline (dotted vertical line).
- Messages typically follow: `return = message(parameter) : returnType`.



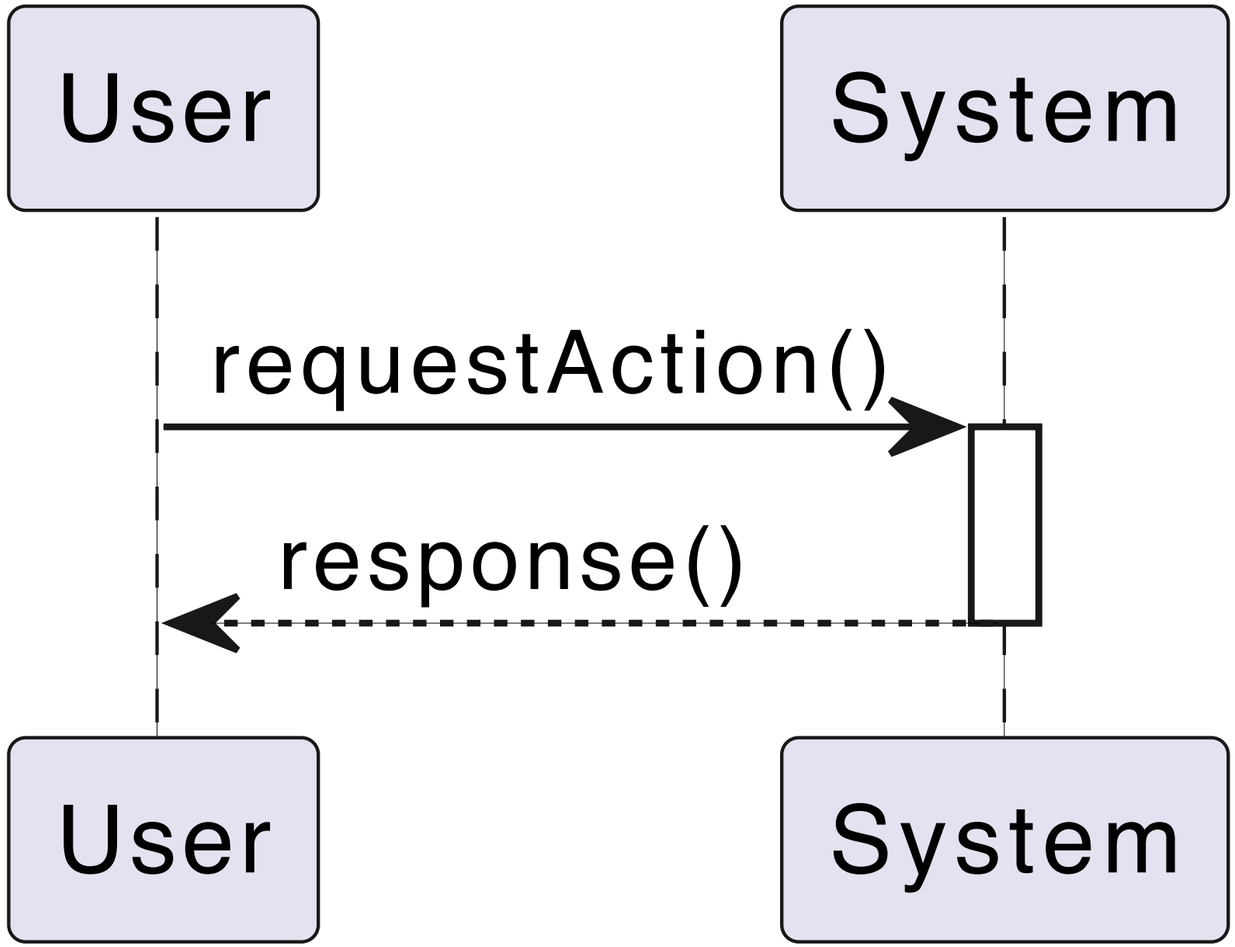
Sequence Diagrams: Messages

- Synchronous = blocking
- Asynchronous = non-blocking
- Return = dashed arrow



Sequence Diagrams: Specifics

- **Execution Specification Bar**: operation on the call stack
- **Replies**: dotted line, possibly with a return value
- **Message to Self**: `this` call
- **Instance Creation/Destruction**: can appear in the diagram



When to Use Interaction Diagrams

- **Modeling Scenarios**: Understand event flows
- **Designing Methods**: Craft complex logic
- **Performance Analysis**: Identify message bottlenecks