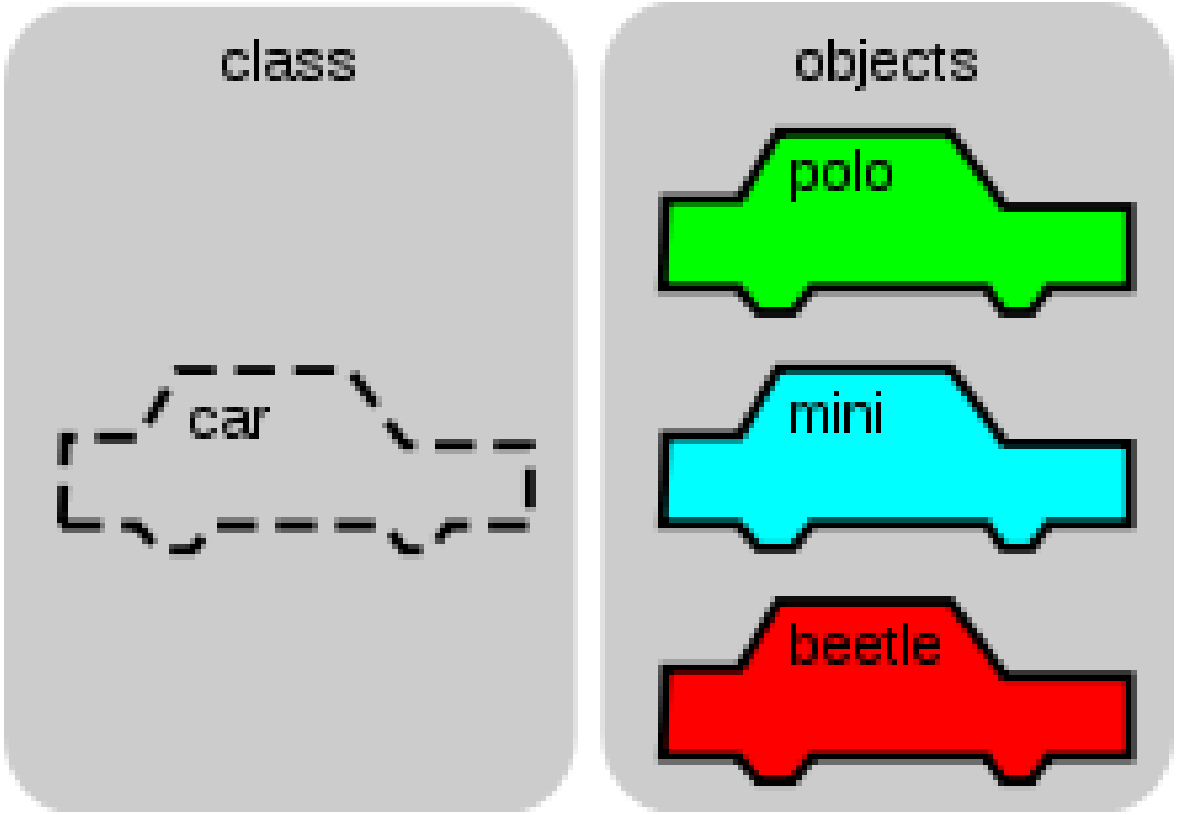
Object Oriented Concepts

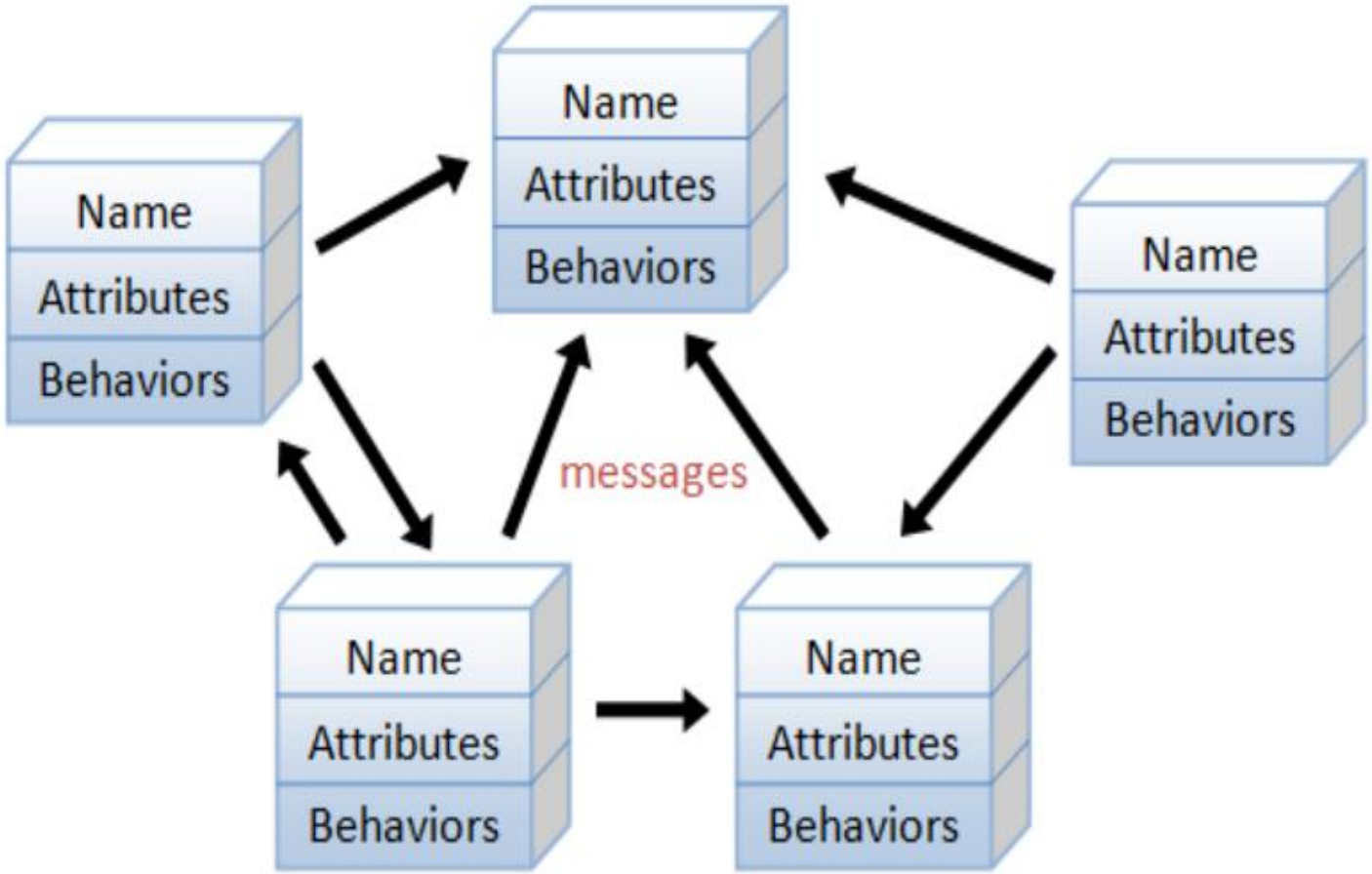
Object Basics



Properties
Make
Model
Color
Year
Price

Events
On_Start
On_Parked
On_Brake

Methods
Start
Drive
Park



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Object

- In **object**-oriented programming (OOP), **objects** are the units of code that are eventually derived from the process.
- Each **object** is an instance of a particular class.
- They have,
  - Properties or attributes
    - Describe the state of an object
  - Methods or procedures
    - Define its behavior

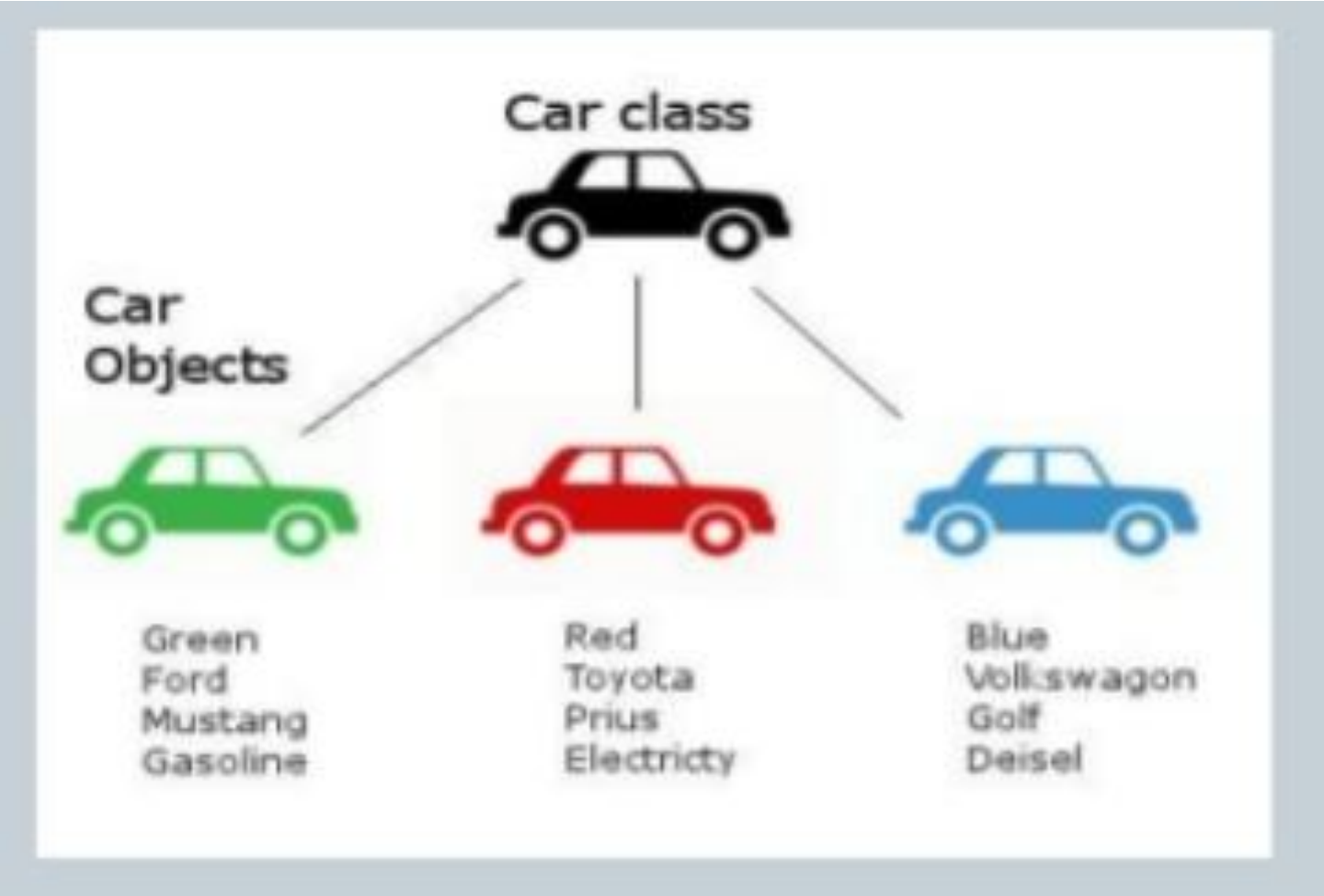
An Object-Oriented Philosophy

- Traditional development methodology
  - **Algorithm-centric methodology**
    - Think algorithm, then build data structures
  - **Data-centric methodology**
    - Think how to structure data, then build algorithm
- **Object-oriented programming**
  - Allows the basic concepts of the language to be extended to include ideas and terms closer to those of its applications.
  - The algorithm and data are packaged together as an object, which has a set of attributes or properties.

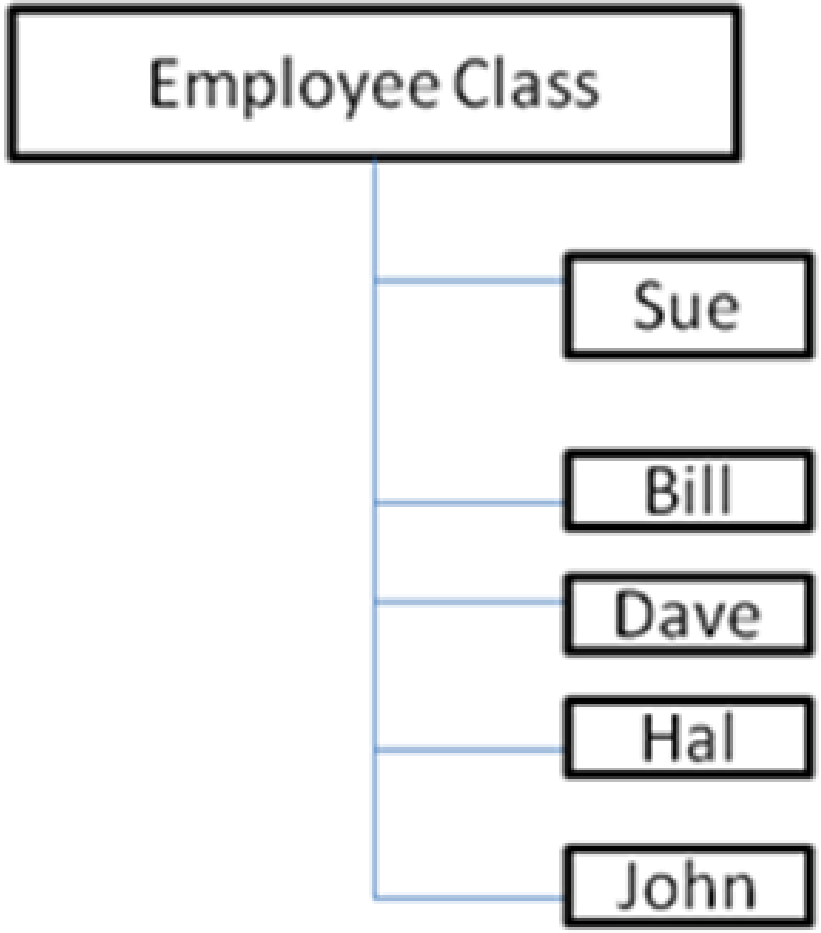
Objects

- The term Object means a combination of data and logic that represents some real world entity.
- In an object-oriented system, **everything is an object.**

Objects are grouped in Classes



- Classes are **used to distinguish one type of object from another.**
- A class is a **set of objects** that share a common structure and behavior.
- A single **object is simply an instance of a class**.
- A class is a specification of **structure** (instance variables), **behavior** (methods) and **inheritance** for objects.
- Classes are an important **mechanism for classifying objects.**
- A method or behavior of an object is defined by its class.
- Each object is an instance of a class.
- Eg) Objects of the class Employee



Attributes: Object state and Properties

- Properties represent the state of an object.

  Eg) the attributes of a car object

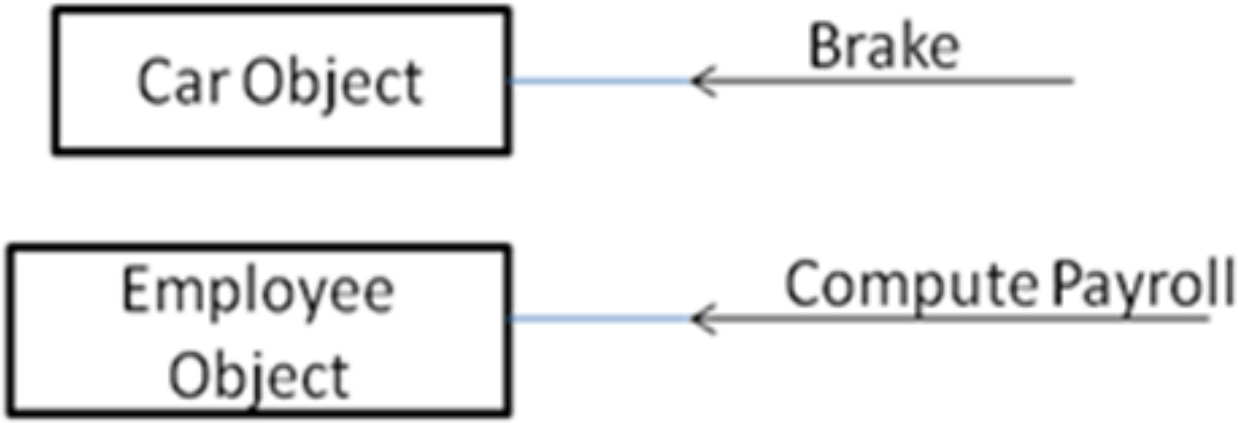
Object Behavior and Methods

- In the object model, object behavior is described in **methods or procedures.**
- A method implements the behavior of an object.
- **A method is a function or procedure that is defined for a class and typically can access the internal state of an object of that class to perform some operation.**
- Behavior denotes the collection of methods that abstractly describes what an object is capable of doing.
- The object is that on which the method operates.
- Methods encapsulate the behavior of the object, provide interfaces to the object, and hide any of the internal structures and states maintained by the object.

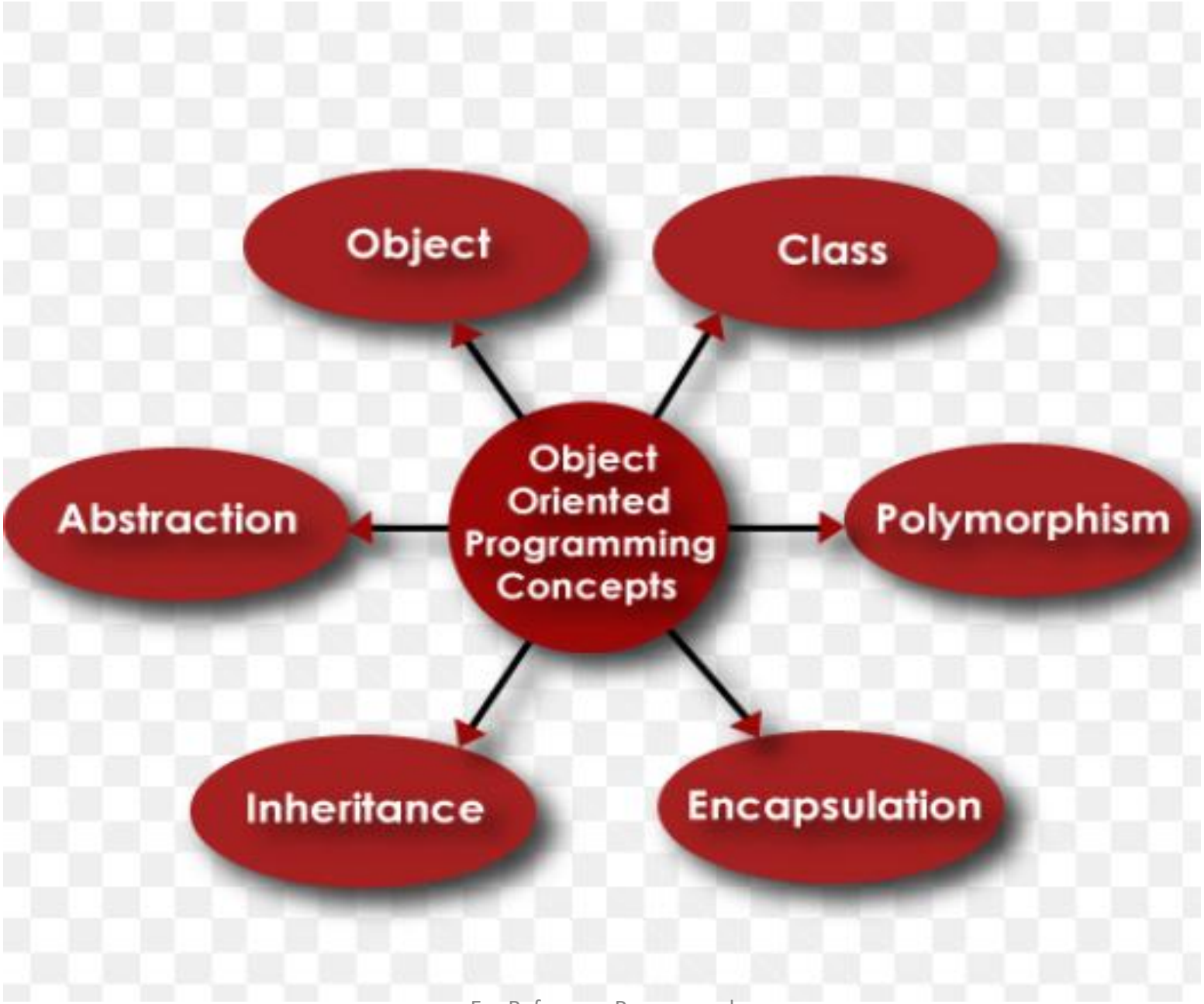
Objects Respond to Messages

- An object's capabilities are determined by the methods defined for it.
- **Objects perform operations in response to messages.**

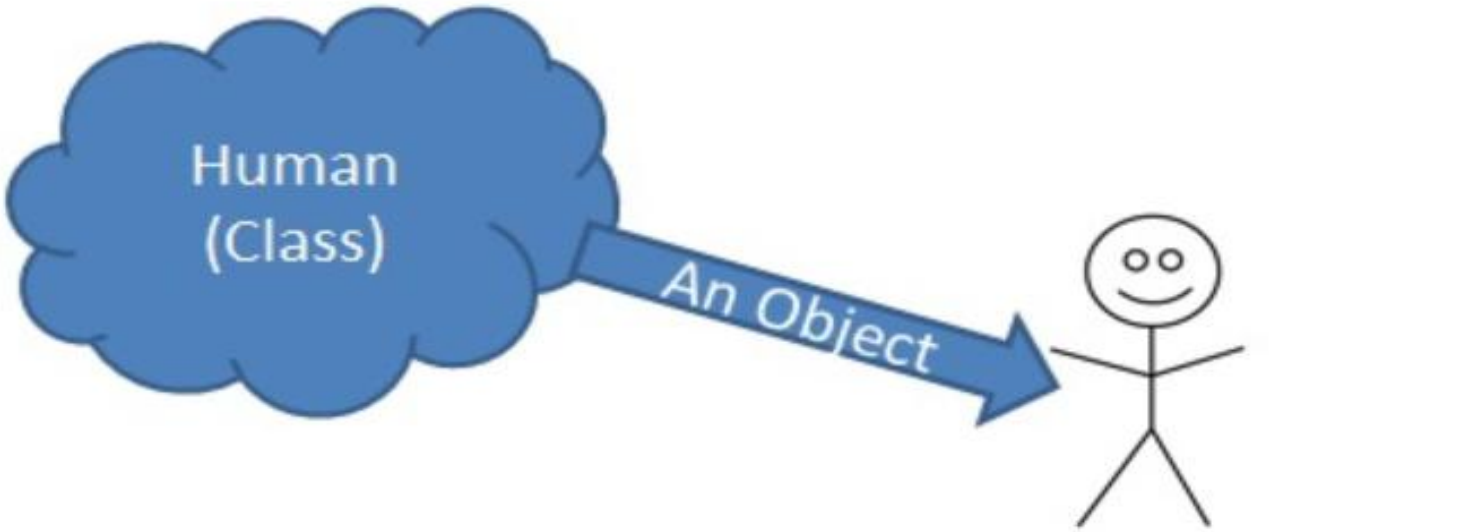
  Eg) **stop method -> car object**
- Messages especially are nonspecific function calls.
- Different objects can respond to the same message in different ways-polymorphism
- Message is the instruction and method is the implementation.
- **An object understands a message when it can match the message to a method that has the same name as the message.**



- The object first searches the methods defined by its class.
- If not found, it searches the superclass of its class.
- An error occurs if none of the superclass contains the method.

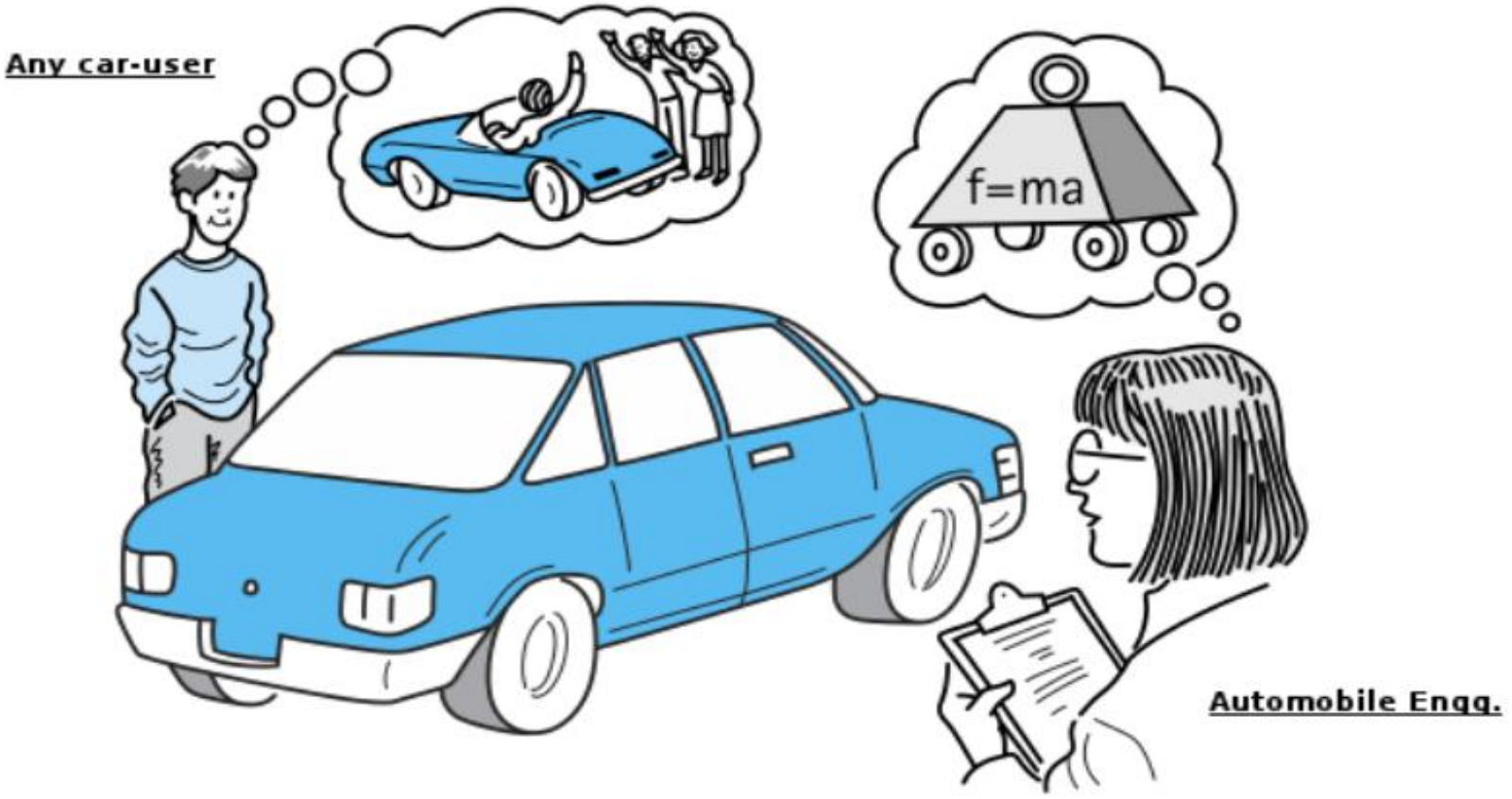


Class and Object



Human Class and Human Object

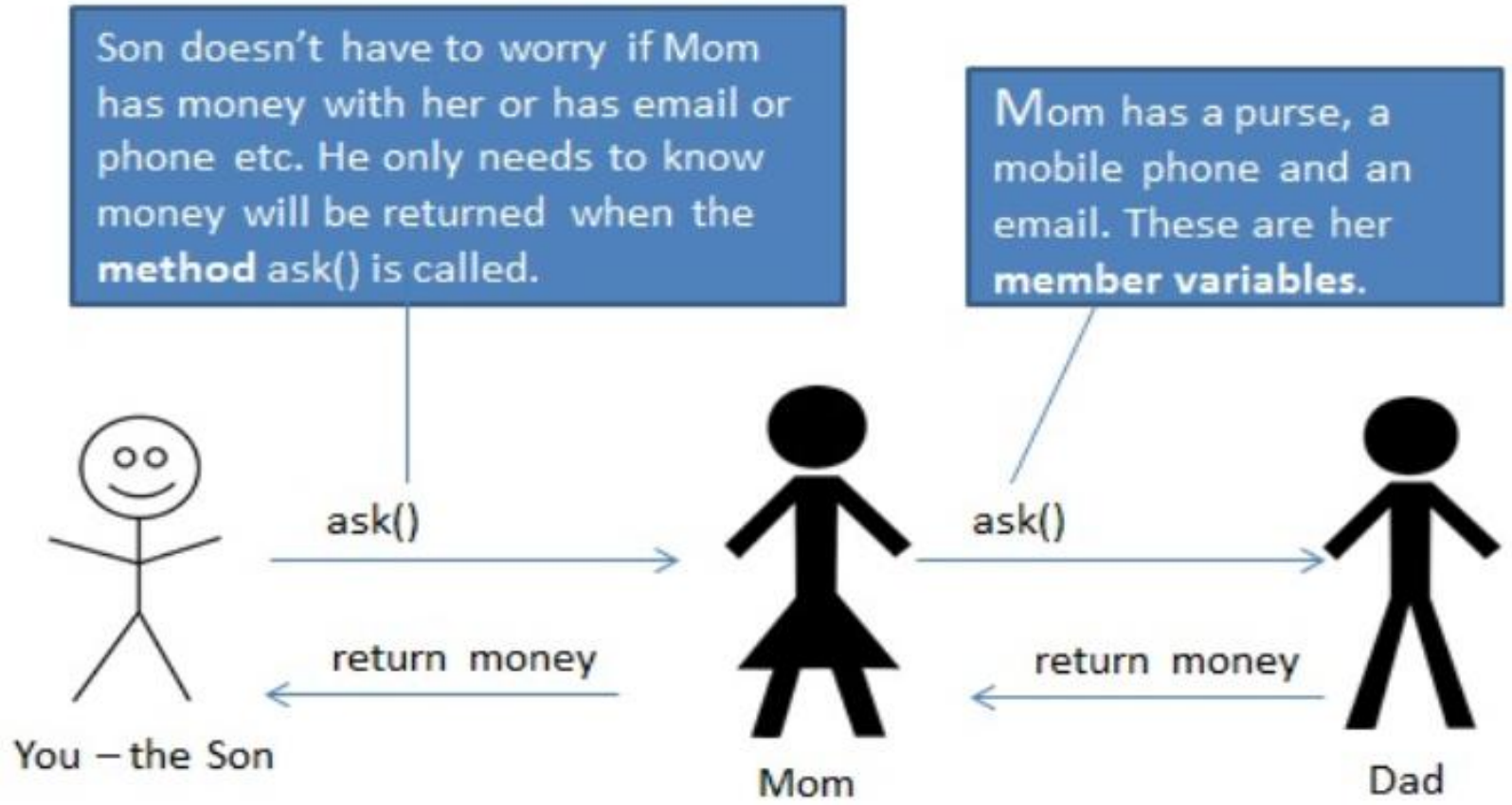
Abstraction



An abstraction includes the essential details relative to the perspective of the viewer

- **Abstraction** means displaying only essential information and hiding the details.
- Data **abstraction** refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- Consider a real life **example** of a man driving a car.

Encapsulation



Mom's details are hidden or encapsulated from Son.

Encapsulation and Information Hiding

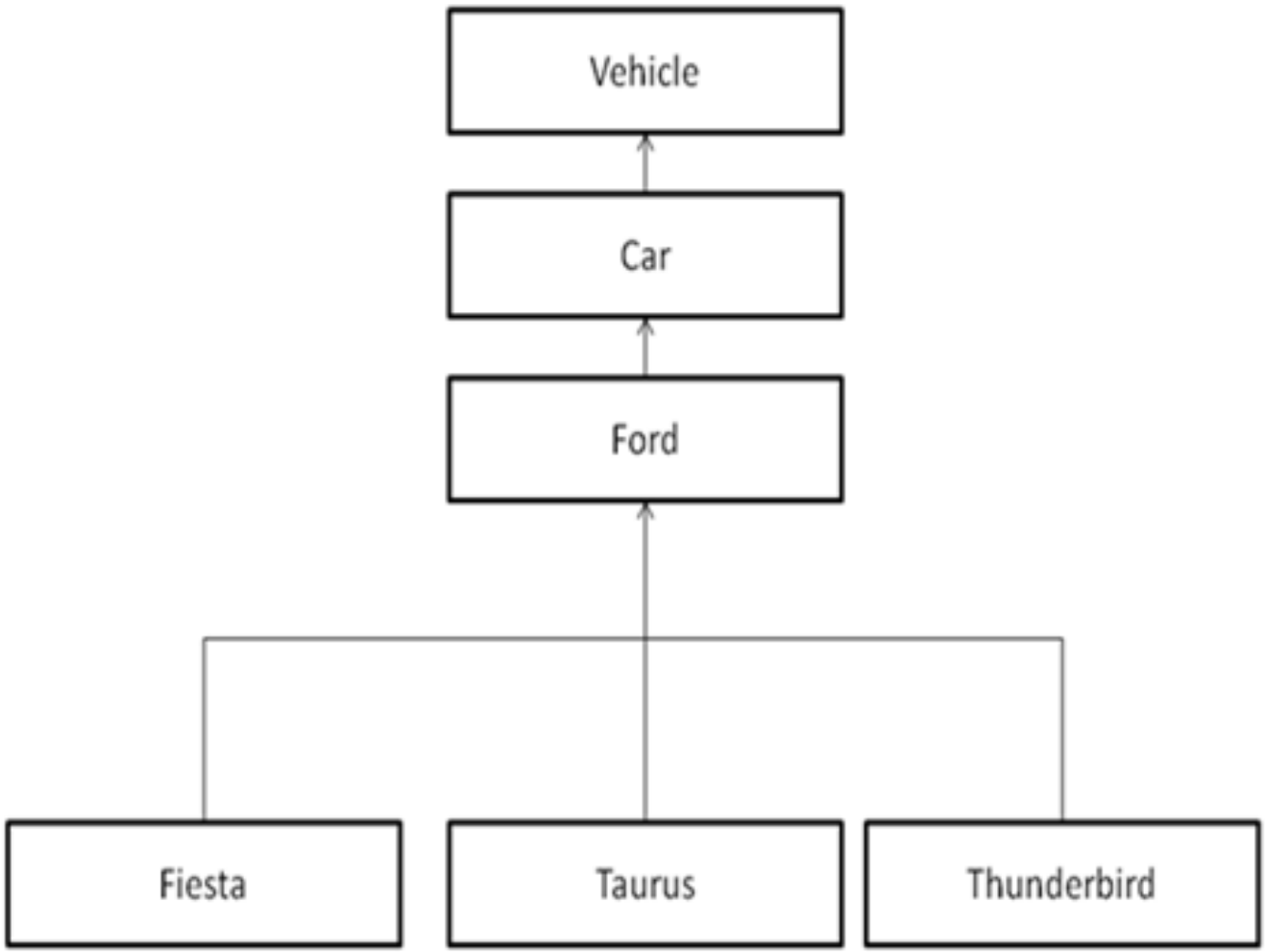
- Principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way as to **reveal as little as possible about its inner workings.**
- Eg) C++ has a very general encapsulation protection mechanism with public, private and protected members.
- Public members may be accessed from anywhere.
- Private members are accessible only from within a class.
- Protected members can be accessed only from subclasses.

- **Per-class protection**
  - Class methods can access any object of that class and not just the object or receiver.
- **Per-object protection**
  - Methods can access only the object or receiver.
- **Important factor in achieving encapsulation**
  - The design of different classes of objects that operate using a common protocol or object's user interface.
  - This means many objects will respond to the same message, but each will perform the message using operations tailored to its class.

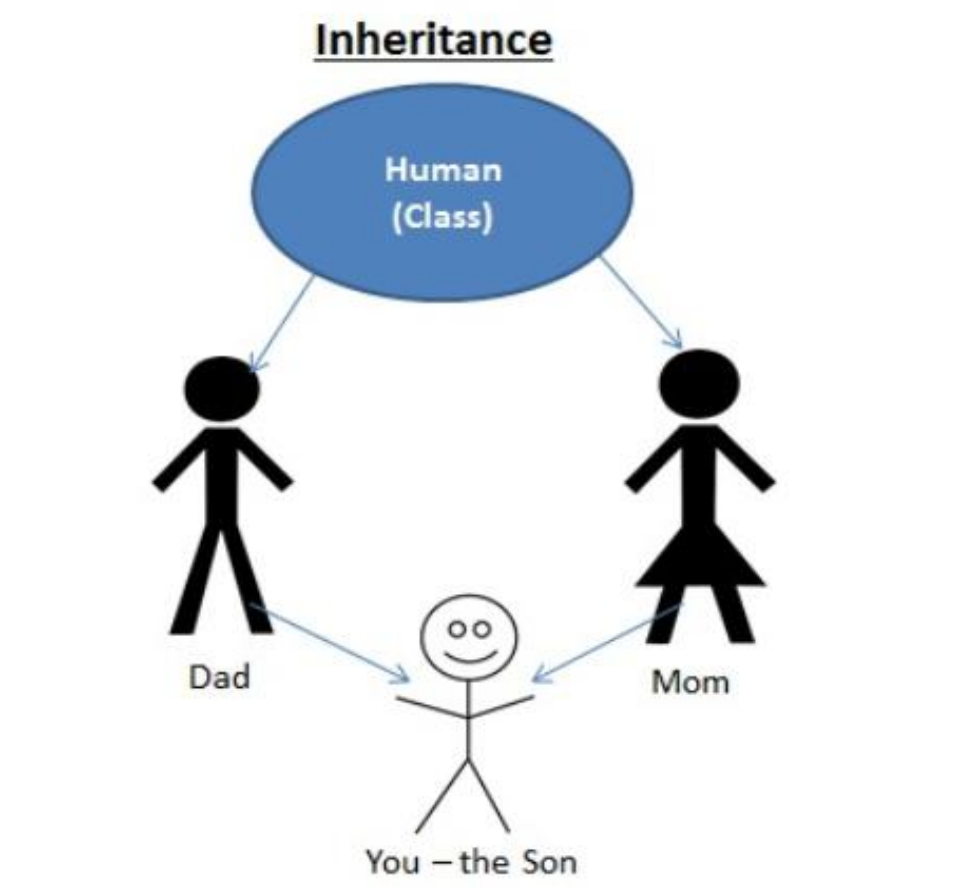
Abstraction	Encapsulation
Abstraction is a general concept formed by extracting common features from specific examples or The act of withdrawing or removing something **unnecessary**.	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interface** and **Abstract** Class	You can implement encapsulation using **Access Modifiers** (Public, Protected & Private)
Abstraction solves the problem in **Design** Level	Encapsulation solves the problem in **Implementation** Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

Class Hierarchy

- An object-oriented system organizes classes into **subclass-superclass hierarchy**.
- At the top of the class hierarchy are the most general classes and at the bottom are the most specific.
- A subclass inherits all of the properties and methods defined in its superclass.
- A class may simultaneously be the subclass to some class and a superclass to another class or classes.



Inheritance

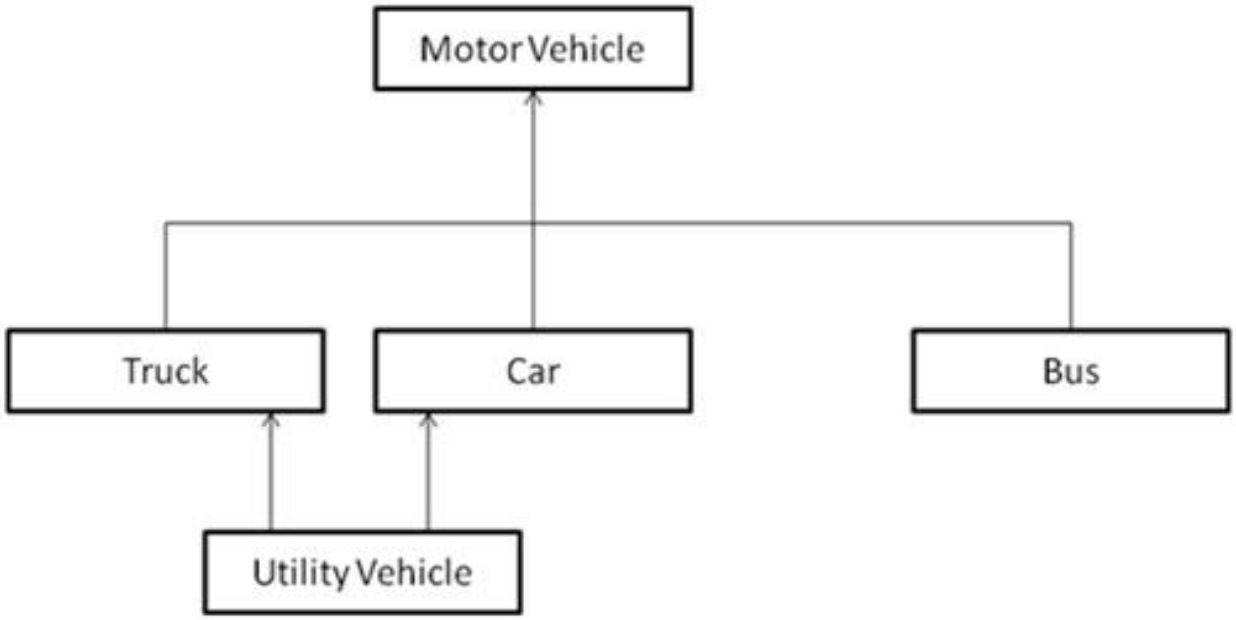


- It is the property of object-oriented systems that allows objects to be built from other objects.
- **Inheritance is a relationship between classes where one class is the parent class of another (derived) class.**
- The parent class is known as the base class or superclass.
- Inheritance allows reusability.

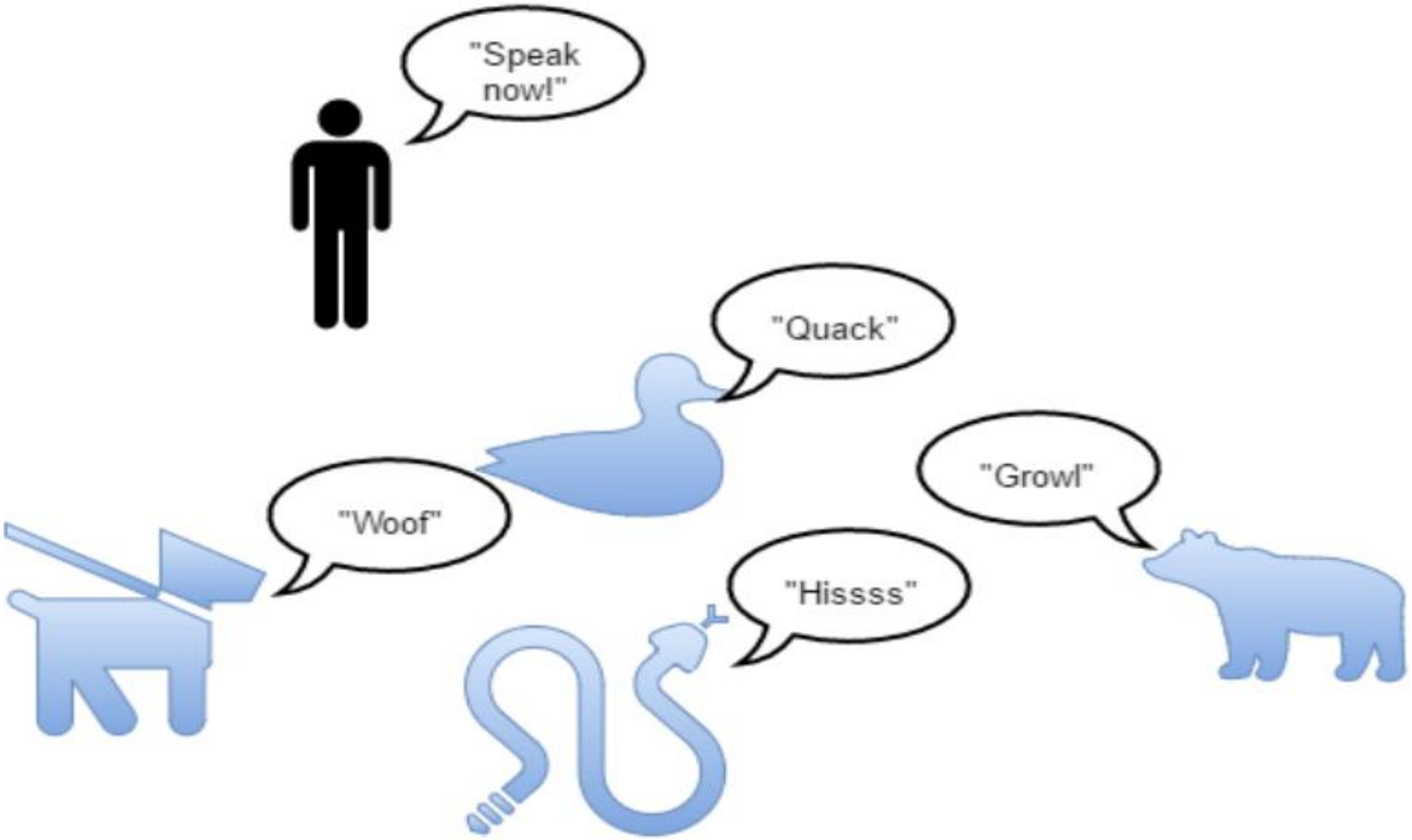
  Eg) the stop method may not be defined in Taurus class but it would be defined in Ford class. Then Taurus can reuse that method from Ford.
- **Dynamic inheritance** allows objects to change and evolve over time.
- It refers to the ability to add, delete, or change parents from objects or classes at run time.

Multiple Inheritance

- Some object-oriented systems permit a class to inherit its state and behaviors from more than one superclass.
- This kind of inheritance is referred to as multiple inheritance.



Polymorphism

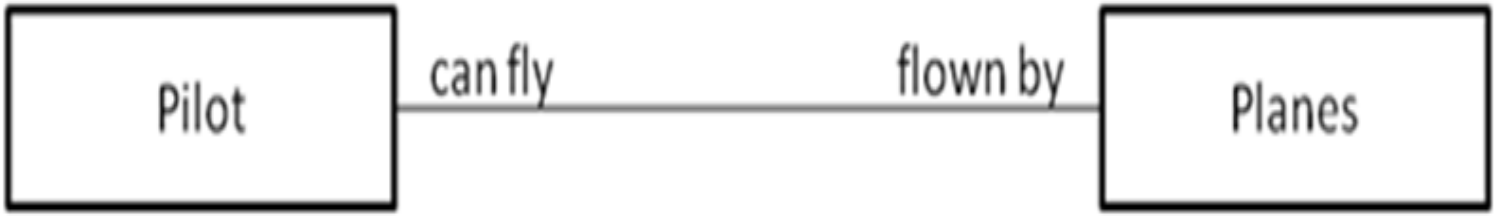


- It means objects that can take on or assume many different forms.
- The same operation may behave differently on different classes.
- Allows us to write generic, reusable code more easily.

Object Relationships and Associations

Association

- A reference from one class to another is an association.
- Association represents the **relationships between objects and classes.**
- **Basically a dependency between two or more classes is an association.**
- Associations are bidirectional.
- The directions implied by the name are the forward direction; the opposite direction is the inverse direction.



Cardinality:

- Specifies how many instances of one class may relate to a single instance of an associated class.

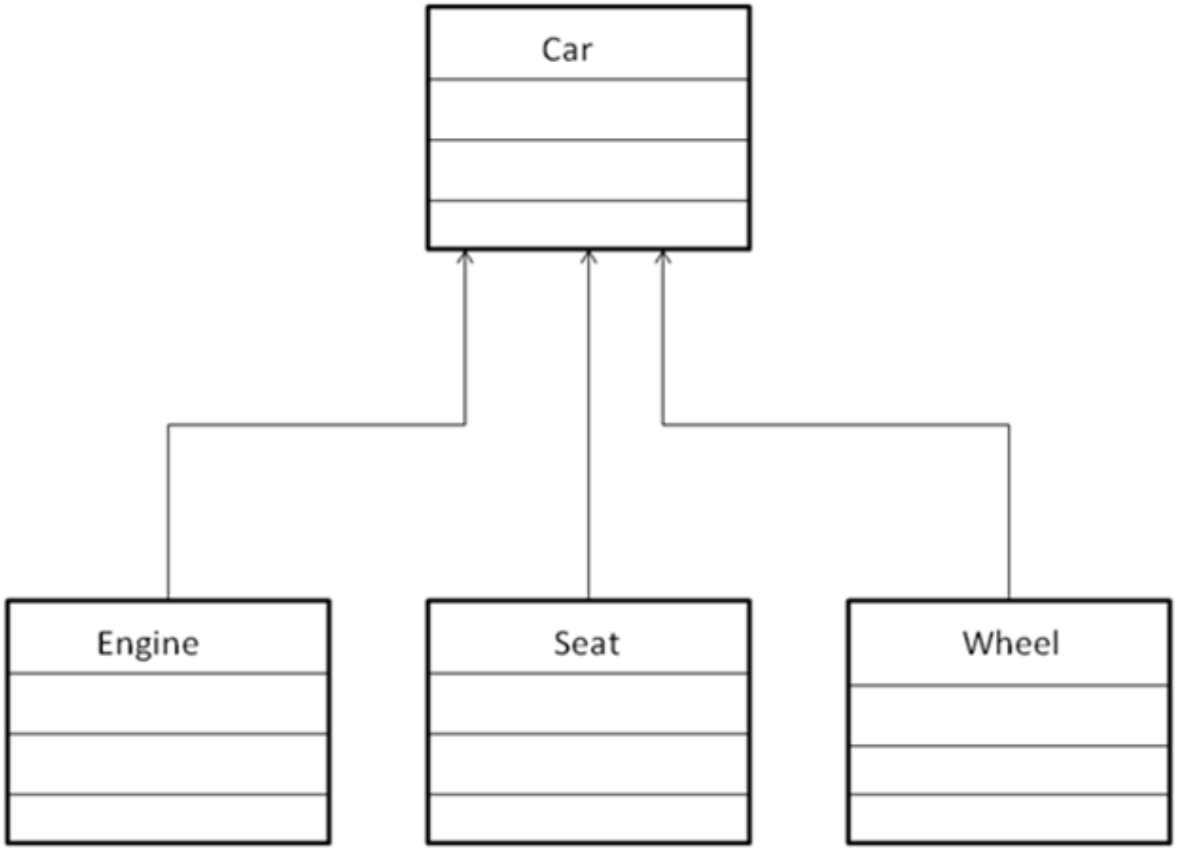
  Eg) client-account relationship – cardinality is many to many if one client can have one or more accounts and vice versa.

Consumer-Producer Association

- Also known as client-server association or a use relationship.
- Viewed as one-way interaction: one object requests the service of another object.
- Requesting object is the consumer and the object that is receiving and providing the service is the producer.

Aggregations and Object Containment

- All objects, except the most basic ones, are composed of and may contain other objects.
- Breaking down objects into the objects from which they are composed is decomposition.
- **Since each object has an identity, one object can refer to other objects. This is known as aggregation, where an attribute can be an object itself.**



- A car object is an aggregation of other objects such as engine, seat and wheel objects

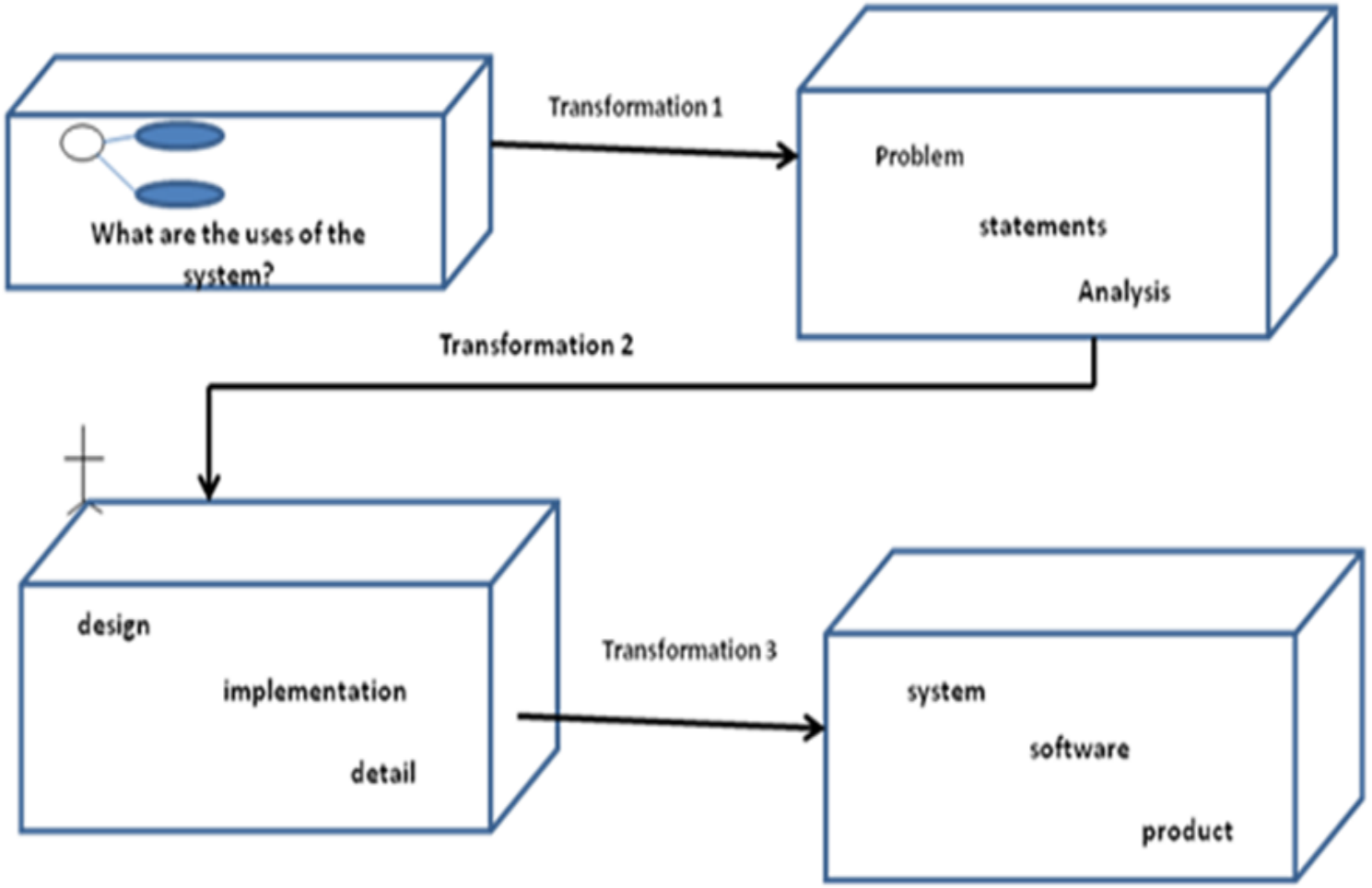
OBJECT ORIENTED SYSTEMS DEVELOPMENT LIFE CYCLE

- *Software development process* consists of **analysis, design, implementation, testing and refinement**
- Transforms users needs into software solution that satisfies those needs.
- The prototype give users a chance to comment on the usability and usefulness of the design

1. The software development process

- Software development can be viewed as a process.
- **Development is a process of change, refinement, transformation or addition to the existing product.**
- Within the process, it is possible to replace one sub process with a new one, as long as the new subprocess has the same interface as the old one, to allow it to fit into the process as a whole.
- With this method of change, it is possible to adapt the new process.

- The process can be divided into small, interacting phases- subprocess.
- Each subprocess must have the following:
  - A description in terms of how it works.
  - Specification of the input required for the process.
  - Specification of the output to be produced.
- The software development process
  - Can be divided into smaller, interacting sub processes.
  - **Can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation.**



Transformation I (analysis):

- Translates the user's needs into system requirements and responsibilities.

Transformation II (design):

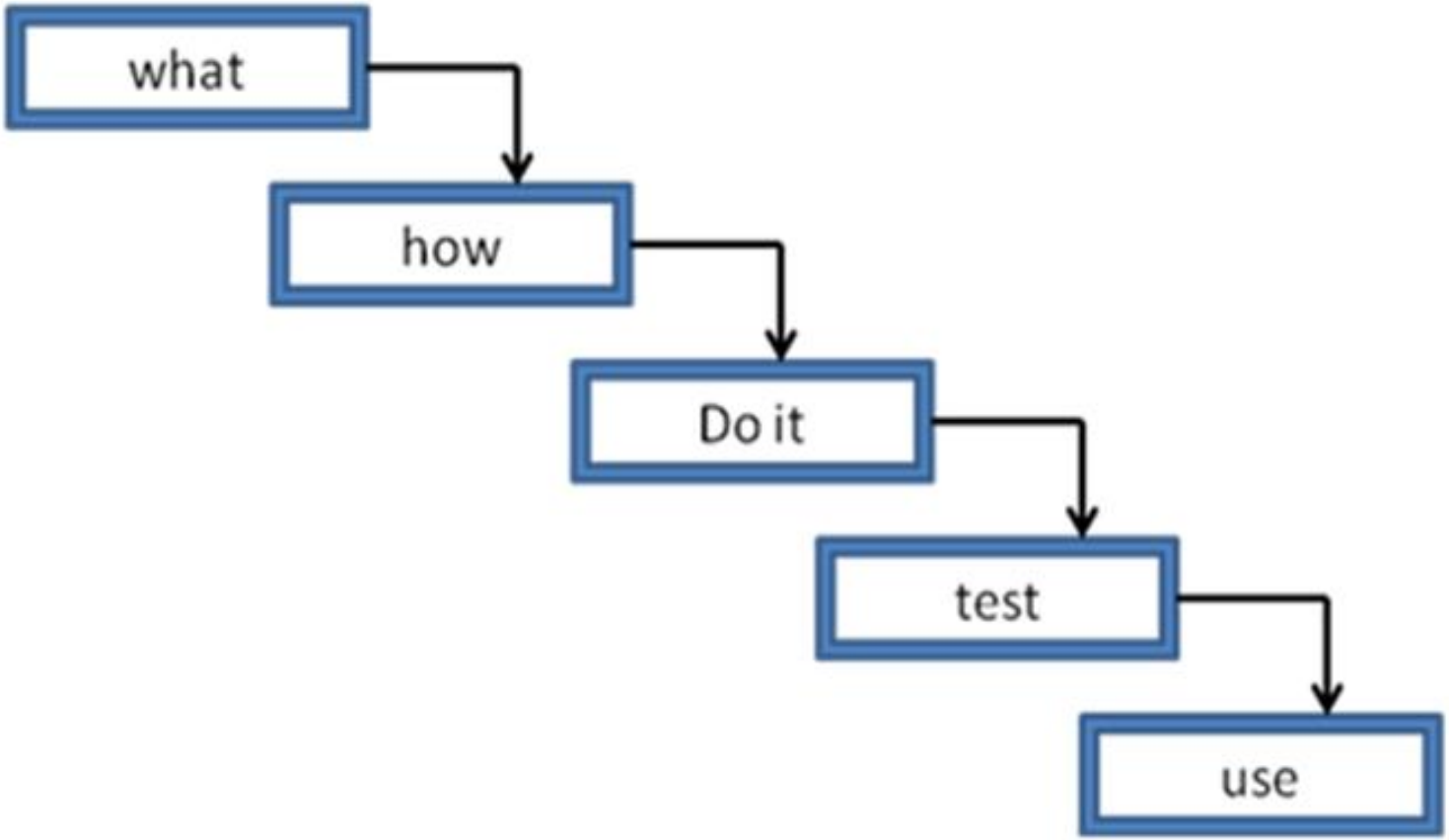
- Begins with a problem statement and ends with a detailed design that can be transformed into an operational system.
- It includes the design descriptions, the program and the testing material.

Transformation III (implementation):

- Refines the detailed design into the system deployment that will satisfy the user's needs.
- Represents embedding the software product within its operational environment.

2. Waterfall Approach

- Example of software development process.
  - Starts with deciding what to be done.
  - Next decide how to accomplish them.
  - Followed by a step in which we do it, whatever “it” has required us to do.
  - Then test the result to see if we have satisfied the user’s requirements.



3. Building High-Quality Software

- Once the system exists, we must test it to see if it is free of bugs.
- The ultimate goal of building high-quality software is **user satisfaction**.
- There are two basic approaches to systems testing.
  - according to how it has been built
  - or what it should do

Blum describes 4 quality measures

- **Correspondence**
  - measures how well the delivered system matches the needs of the operational environment
- **Validation**
  - task of predicting correspondence (Am I building the right product)
- **Correctness**
  - Measures the consistency of the product requirements with respect to the design specification.
- **Verification**
  - exercise of determining correctness (Am I building the product right)

UNIFIED MODELING LANGUAGE

- A model is an **abstract representation of a system**, constructed to understand the system prior to building or modifying it.
- Most modelling techniques used for analysis and design involve graphic languages. These graphic languages are **set of symbols**.
- The symbols are used according to certain rules of the methodology for communicating the complex relationships of information more clearly than descriptive text.

- Objectory is built around several different models
  - **Use-case model**
    - Defines the outside(actors) and inside(use case) of the system's behaviour
  - **Domain object model**
    - Objects of the "real world" are mapped into the domain object model.
  - **Analysis object model**
    - Presents how the source code should be carried out and written.
  - **Implementation model**
    - Represents the implementation of the system.
  - **Test model**
    - Constitutes the test plans, specifications and reports.

- **Static Model**
  - snapshot of a system's parameters at a specific point of time.
  - Static models assume stability and an absence of change in data over time.
  - **Class diagram** is an example of a static model
- **Dynamic Model**
  - collection of procedures or behaviors that reflect the behavior of a system over time.
  - show how the business objects interact to perform tasks.
  - most useful during the design and implementation phases of the system development.
  - The **UML interaction diagrams** and **activity models** are examples of UML dynamic models.

- Need for Modeling
  - Clarity
  - Familiarity
  - Maintenance
  - Simplification
- Advantages of Modeling (Turban)
  - easier to express complex ideas
  - reduction of complexity
  - enhance & reinforce learning and training
  - cost of the modeling analysis is much lower
  - Manipulation of the model is much easier

UML diagrams

1. Use case diagram
2. Class Diagram
3. Behavioral diagrams
    - State chart diagrams
    - Object diagram
    - Activity diagrams
    - Interaction diagrams
        - Sequence diagrams
        - Collaboration diagrams
4. Implementation diagrams
    - Component diagram
    - Deployment diagram

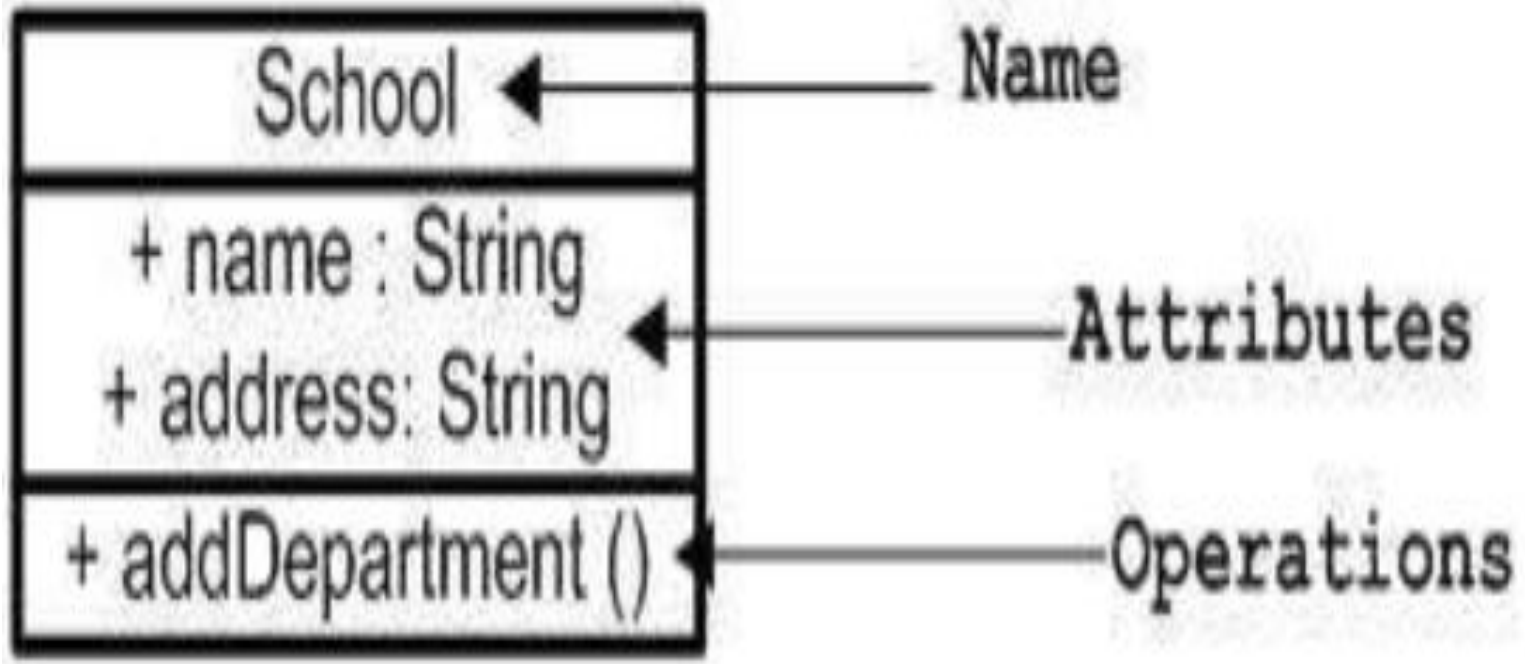
UML CLASS DIAGRAM

- also referred to as **object modeling**
- main static analysis diagram.
- Show the static structure of the model.
- collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents.

Class Notation: static structure

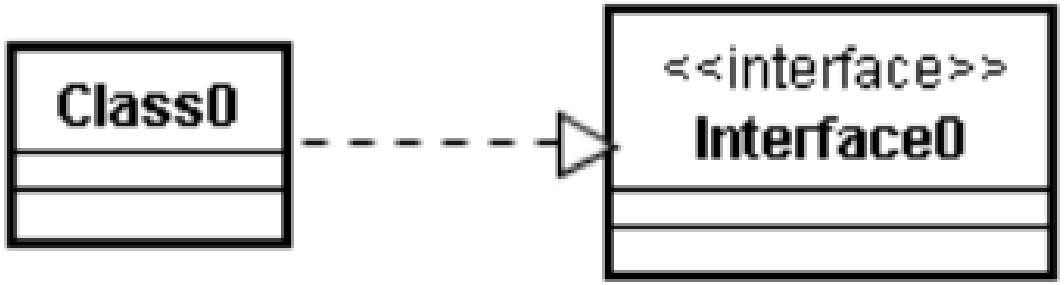
- A class is drawn as a rectangle with three components separated by horizontal lines.
  - **top compartment** -> holds class name
  - **middle compartment** -> general properties of the class, such as attributes
  - **bottom compartment** -> holds a list of operations

Active Class



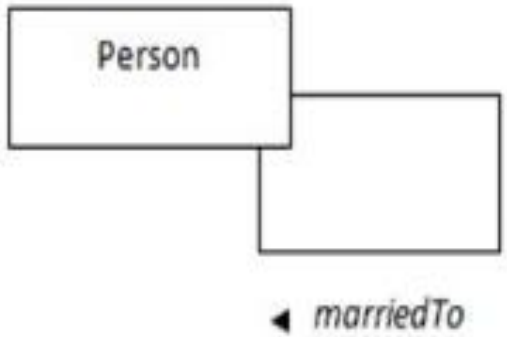
Class Interface Notation

- used to describe the **externally visible behavior of a class**
- The UML notation for an interface is a **small circle with the name of the interface connected to the class.**



Binary Association Notation

- A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class.
- The association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name.
- **The end of an association, where it connects to a class, is called the association role.**

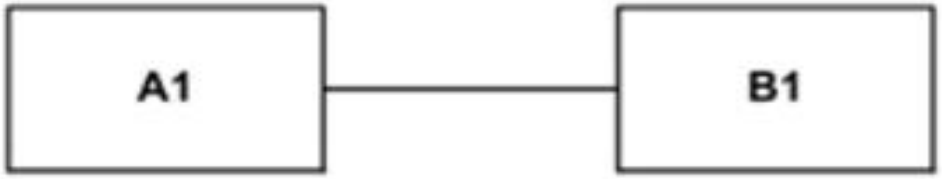


Association Role

- Each association has two or more roles to which it is connected.
- In the above fig, the association worksFor connects two roles, employee and employer.
- A Person is an employee of a Company and a Company is an employer of a Person.

Association Navigation or navigability

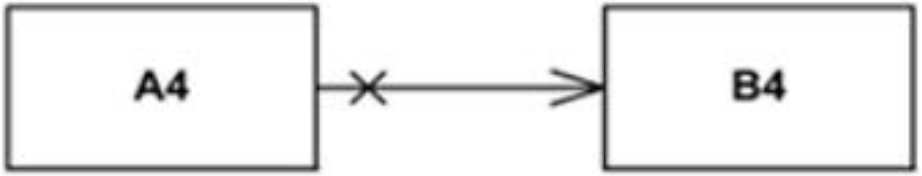
- is visually distinguished from interface, which is denoted by an unfilled arrowhead symbol near the superclass
- to indicate that navigation is supported in the direction of the class pointed to.
- The arrows describe navigability.
  - Navigable end is indicated by an open arrowhead on the end of an association
  - Not navigable end is indicated with a small x on the end of an association
  - No adornment on the end of an association means unspecified navigability



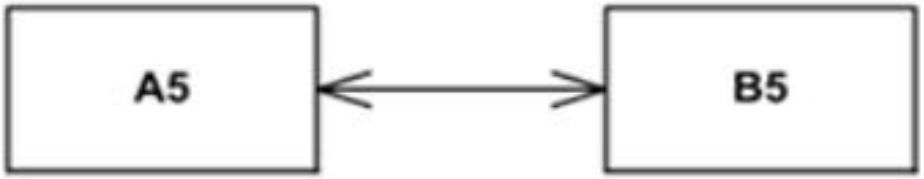
Both ends of association have ***unspecified navigability.***

A2 has ***unspecified navigability*** *while B2 is* ***navigable*** *from A2.*

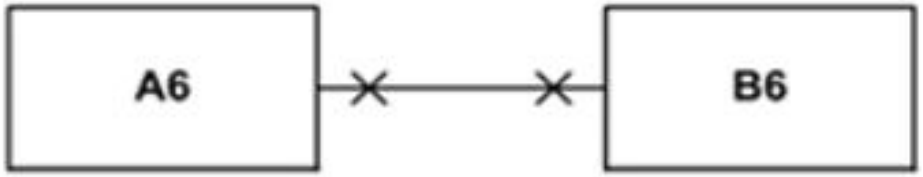
A3 is ***not navigable*** *from B3 while B3 has* ***unspecified navigability.***



A4 is ***not navigable*** *from B4 while B4 is* ***navigable*** *from A4.*



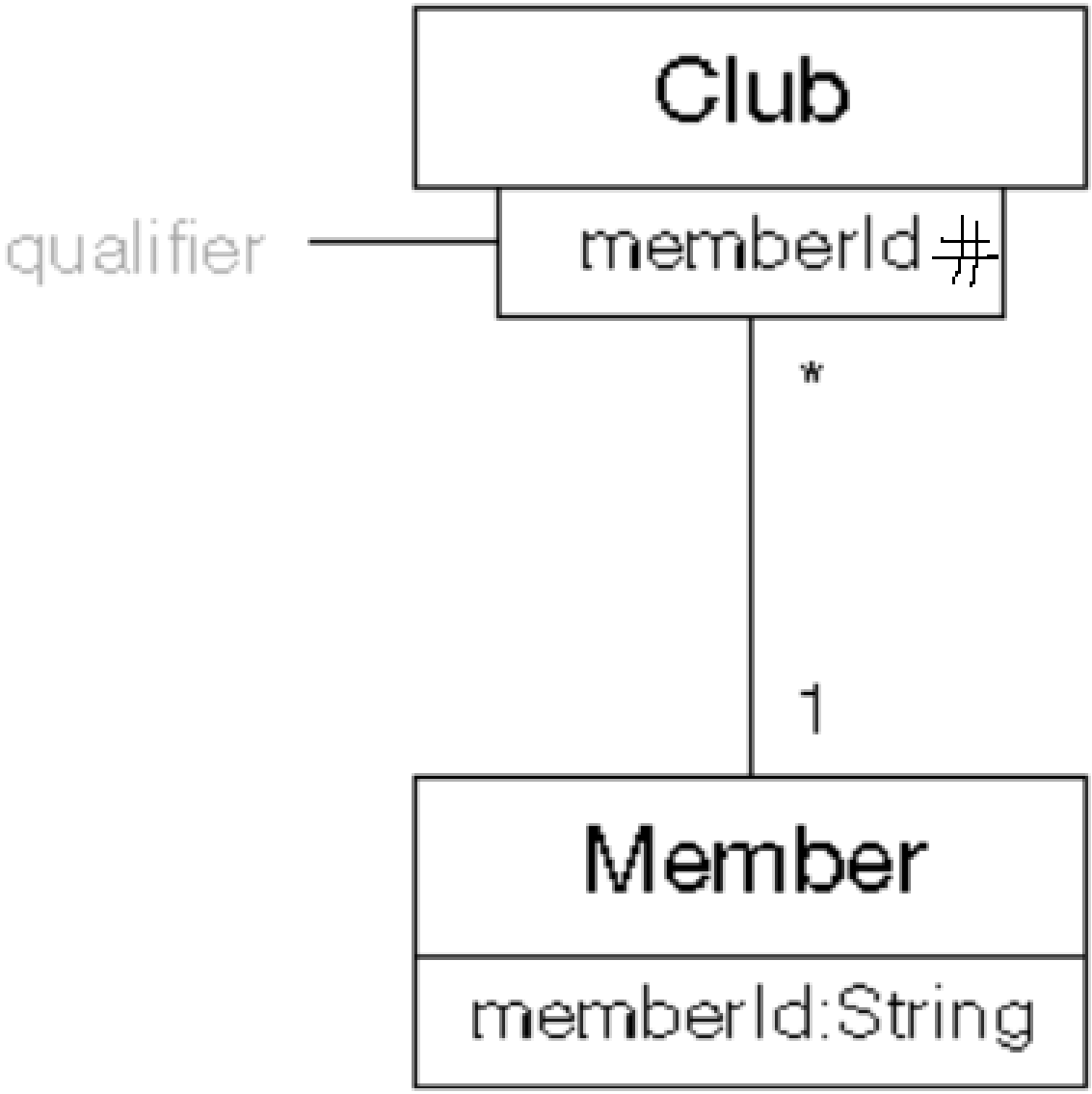
A5 is ***navigable*** *from B5 and B5 is* ***navigable*** *from A5.*



A6 is ***not navigable*** *from B6 and B6 is* ***not navigable*** *from A6.*

Qualifier

- A qualifier is an **association attribute.**
- For example, a Member may be associated to a Club object. An attribute of this association is the memberid#.
- The memberid# is the qualifier of this association.
- A qualifier is shown as a small rectangle attached to the end of an association path.
- Usually smaller than the attached class rectangle.



Multiplicity

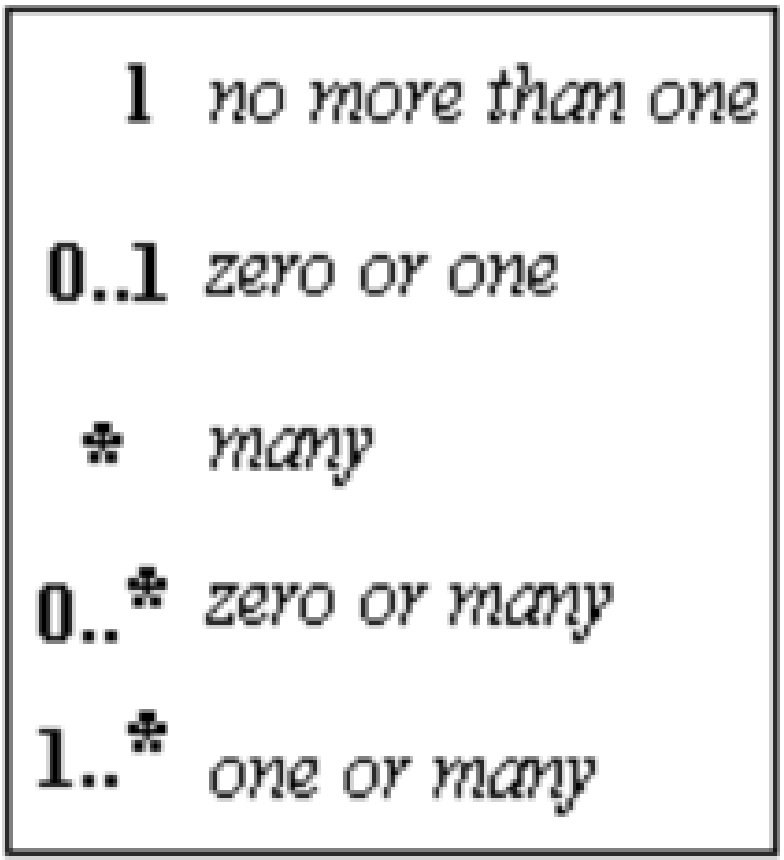
- Specifies the range of allowable classes.
- A multiplicity specification is shown as a text string comprised a period-separated sequence of integer intervals in this format
- lower bound ….. upper bound
- The star character (*) may be used for the upper bound, denoting an unlimited upper bound

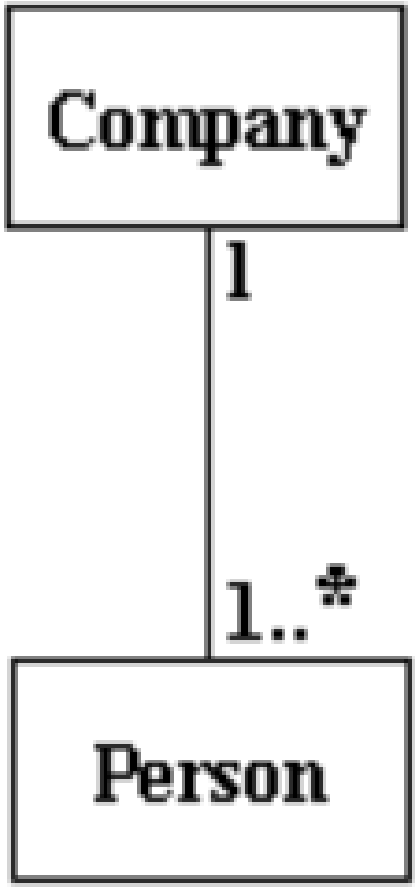
Eg,

0…1

0…*

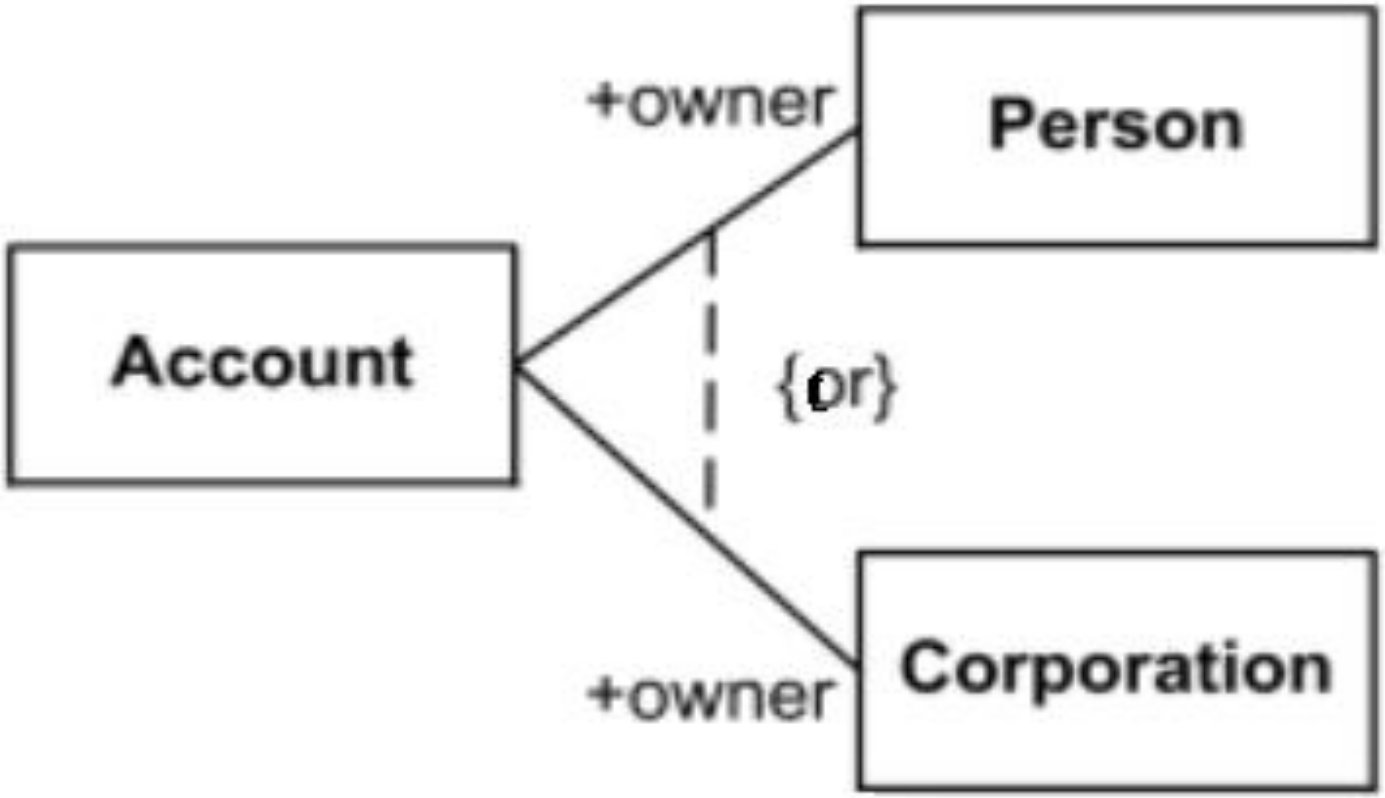
1…3, 7…10, 15, 19…*





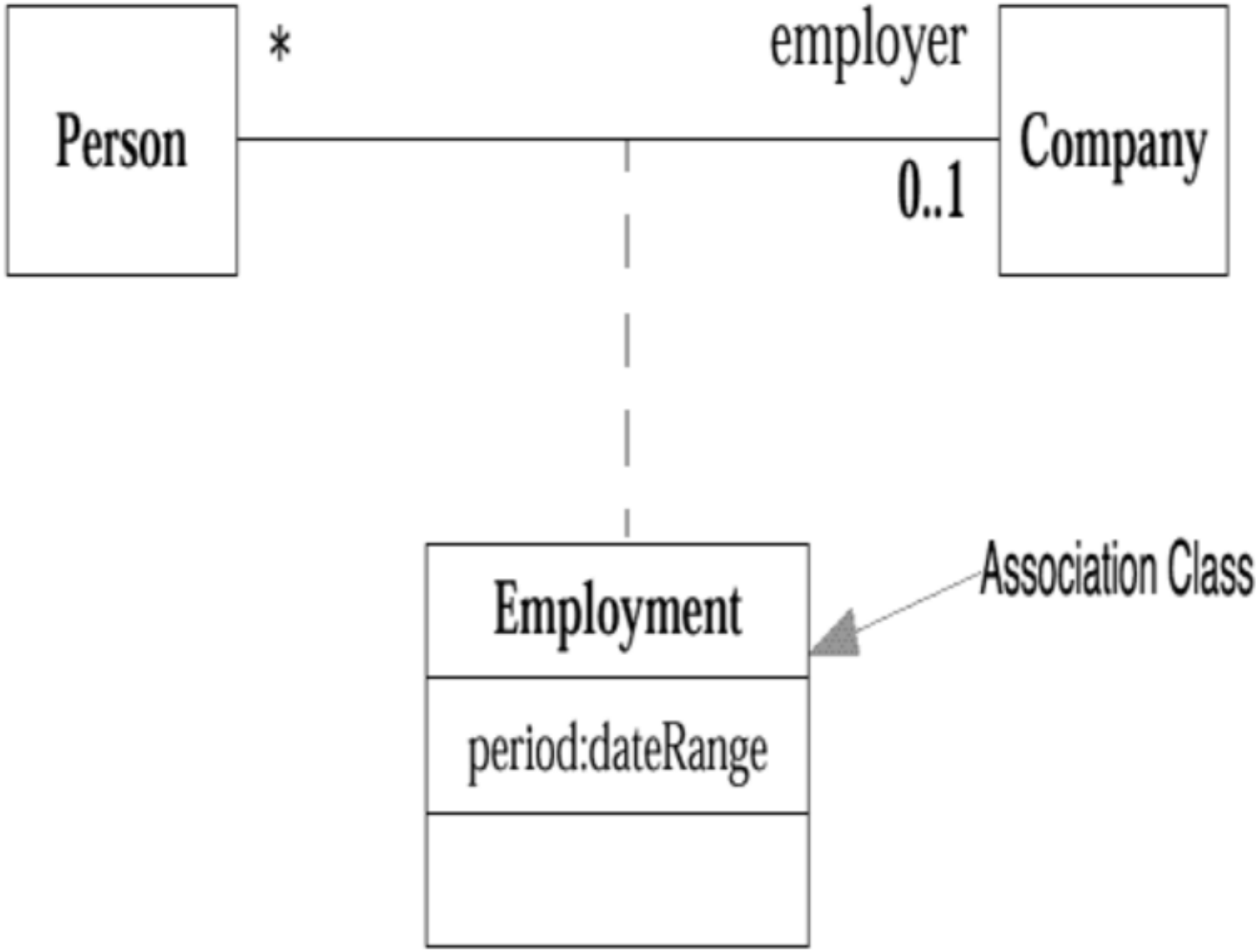
OR Association

- Indicates a situation in which only one of several potential associations may be instantiated at one time for any single object.
- This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string {or} labeling the dashed line.



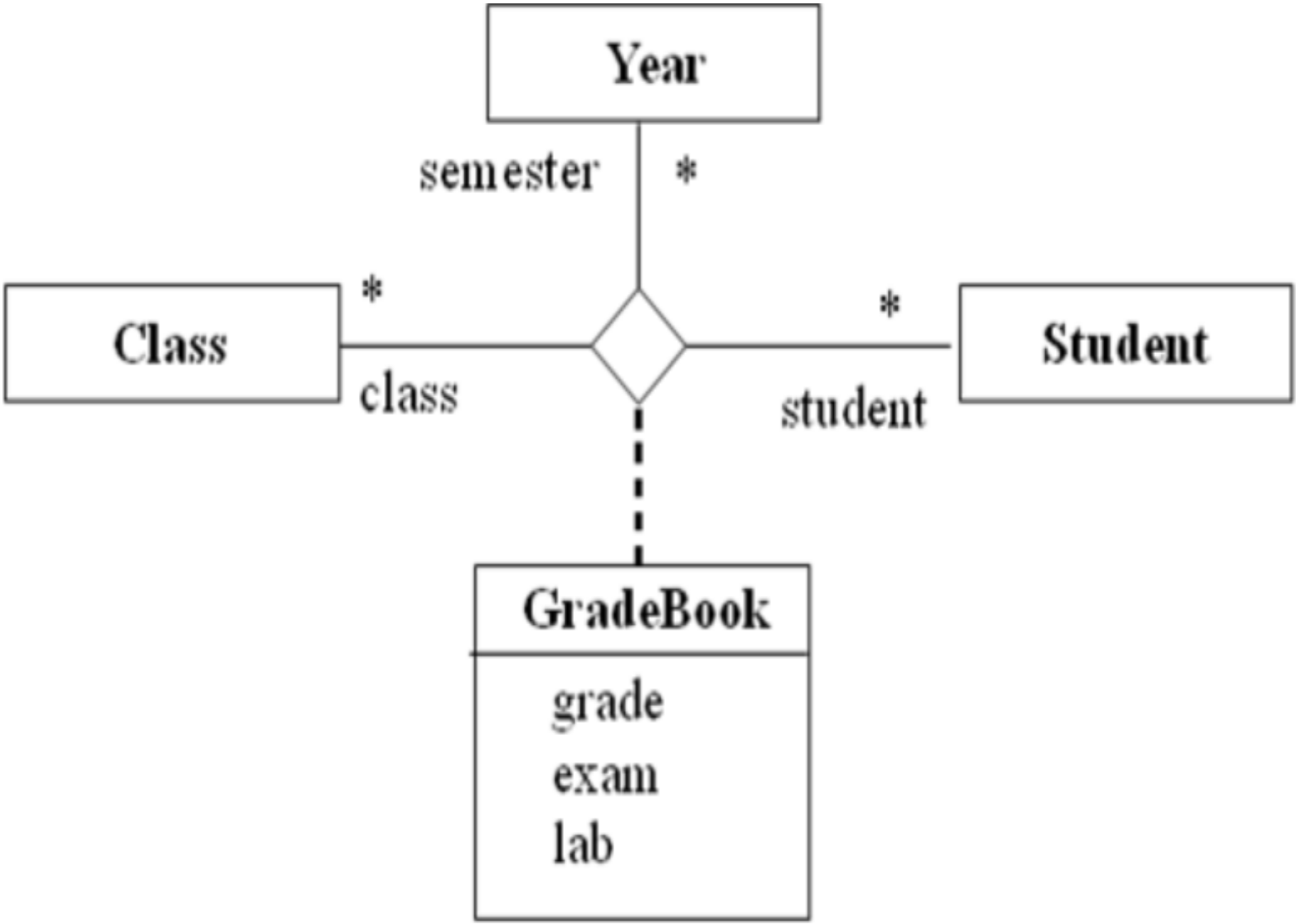
Association class

- An association class is an association that also has class properties.
- An association class is shown as a class symbol attached by a dashed line to an association path.



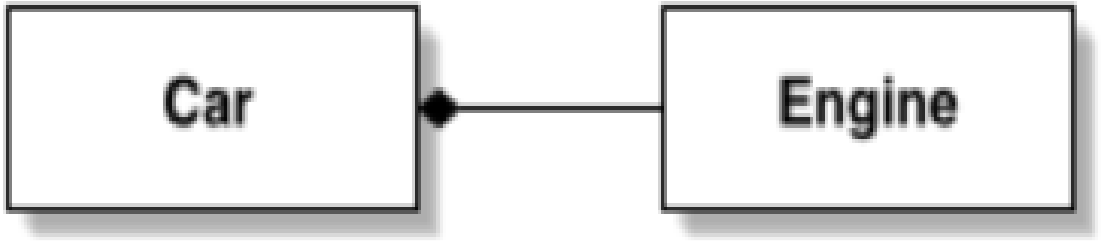
N-Ary Association

- An n-ary association is an **association among more than two classes.**
- It is shown as a large diamond with a path from the diamond to each participant class
- An association class symbol may be attached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation or associations.

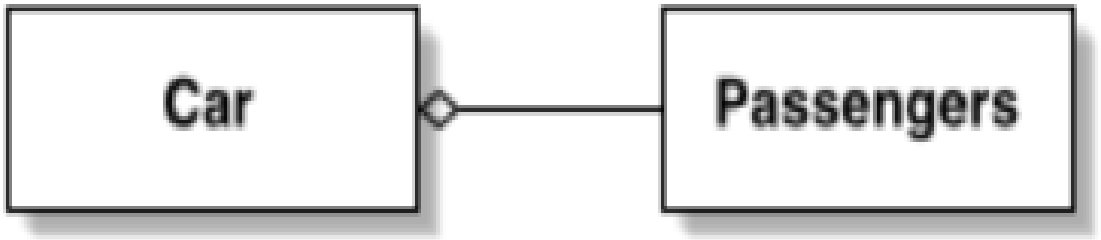


Aggregation

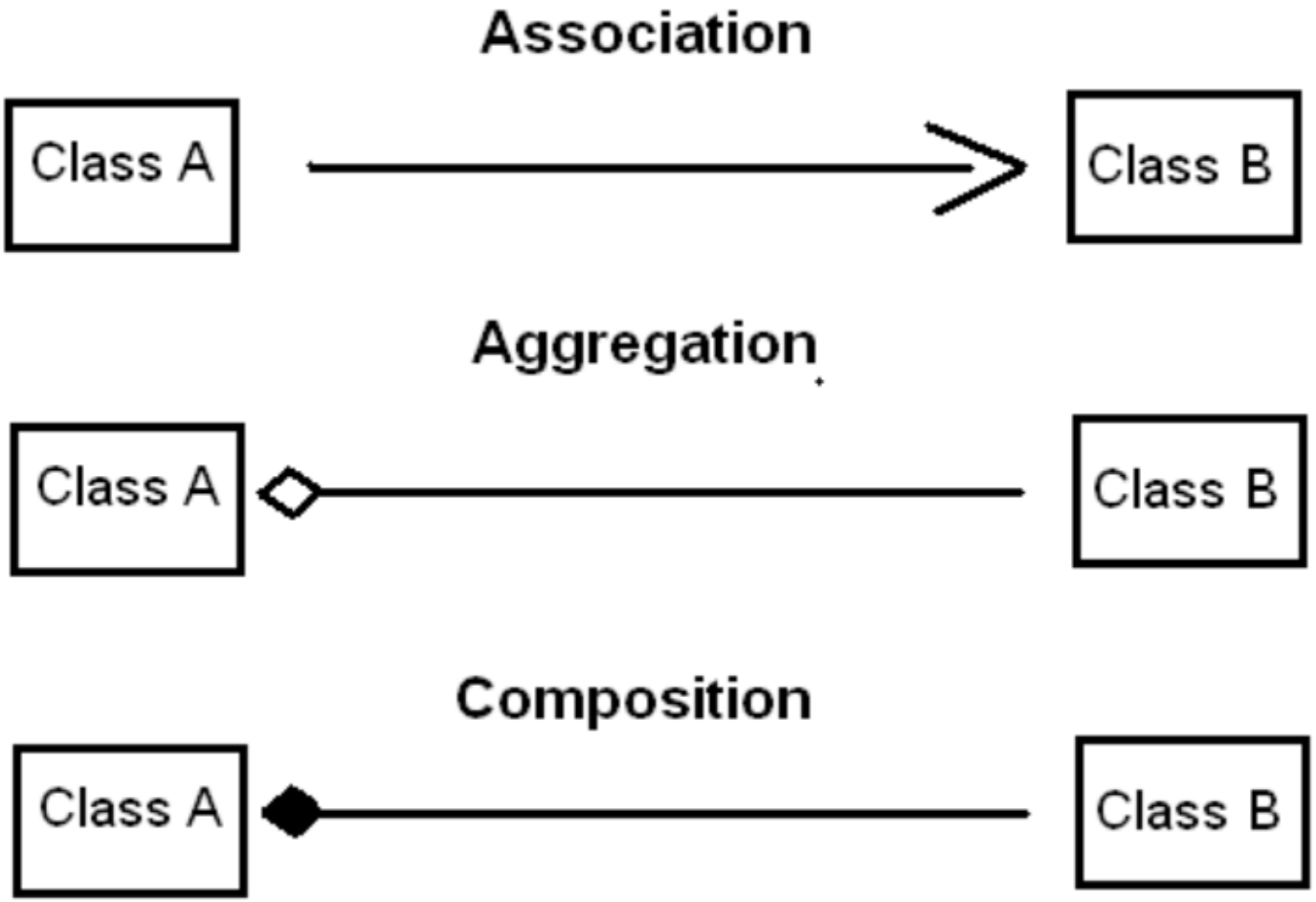
- Aggregation is a form of association.
- A **hollow diamond** is attached to the end of the path to indicate aggregation.
- However, the diamond may not be attached to both ends of a line.
- **Composition**
- Also known as the ***a-part-of***, is a form of aggregation with strong ownership to represent the component of a complex object.
- Also referred to as a ***part-whole*** relationship.
- UML notation for composition is a **solid diamond** at the end of the path.



Composition: every car has an engine.

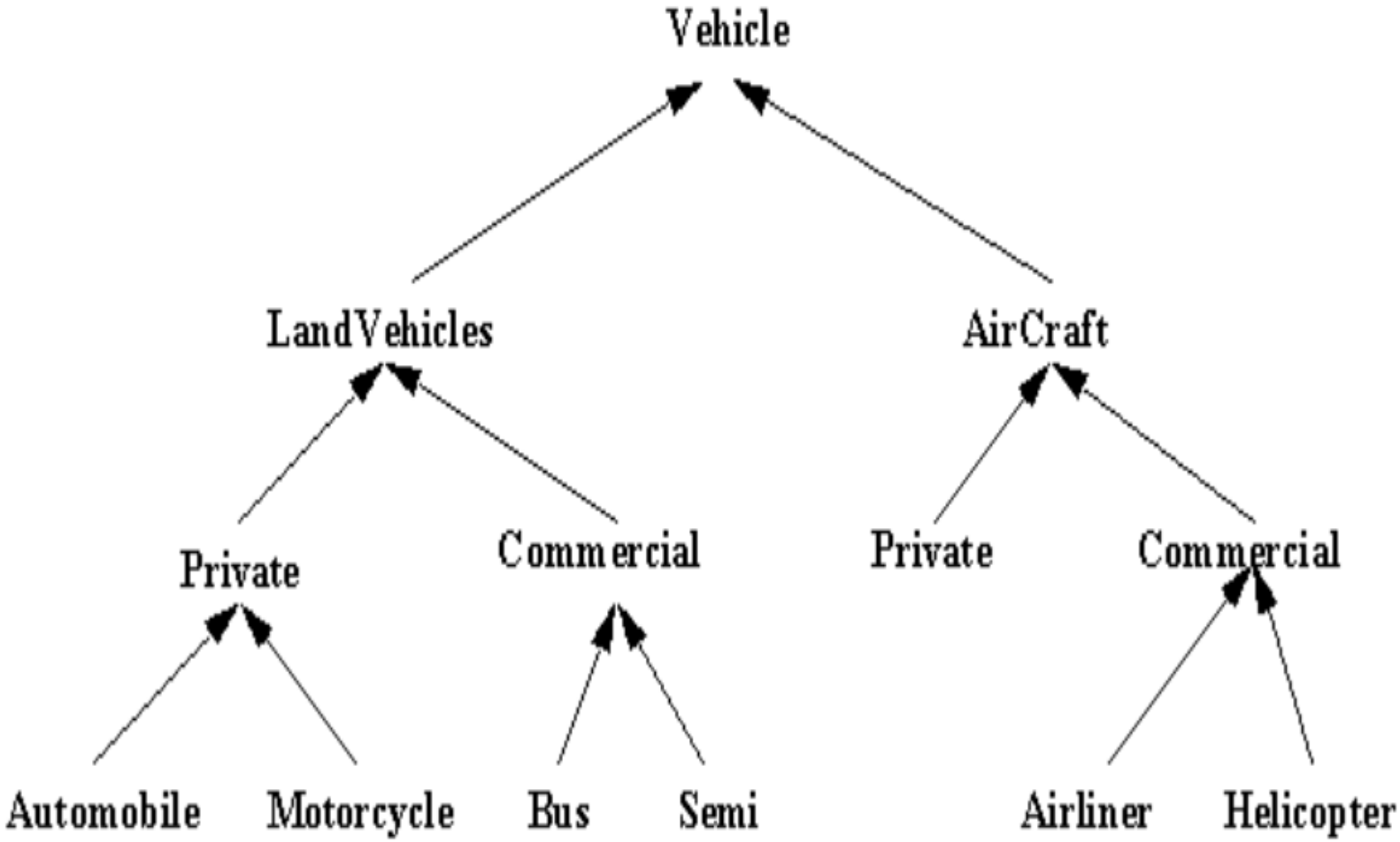


Aggregation: cars may have passengers, they come and go



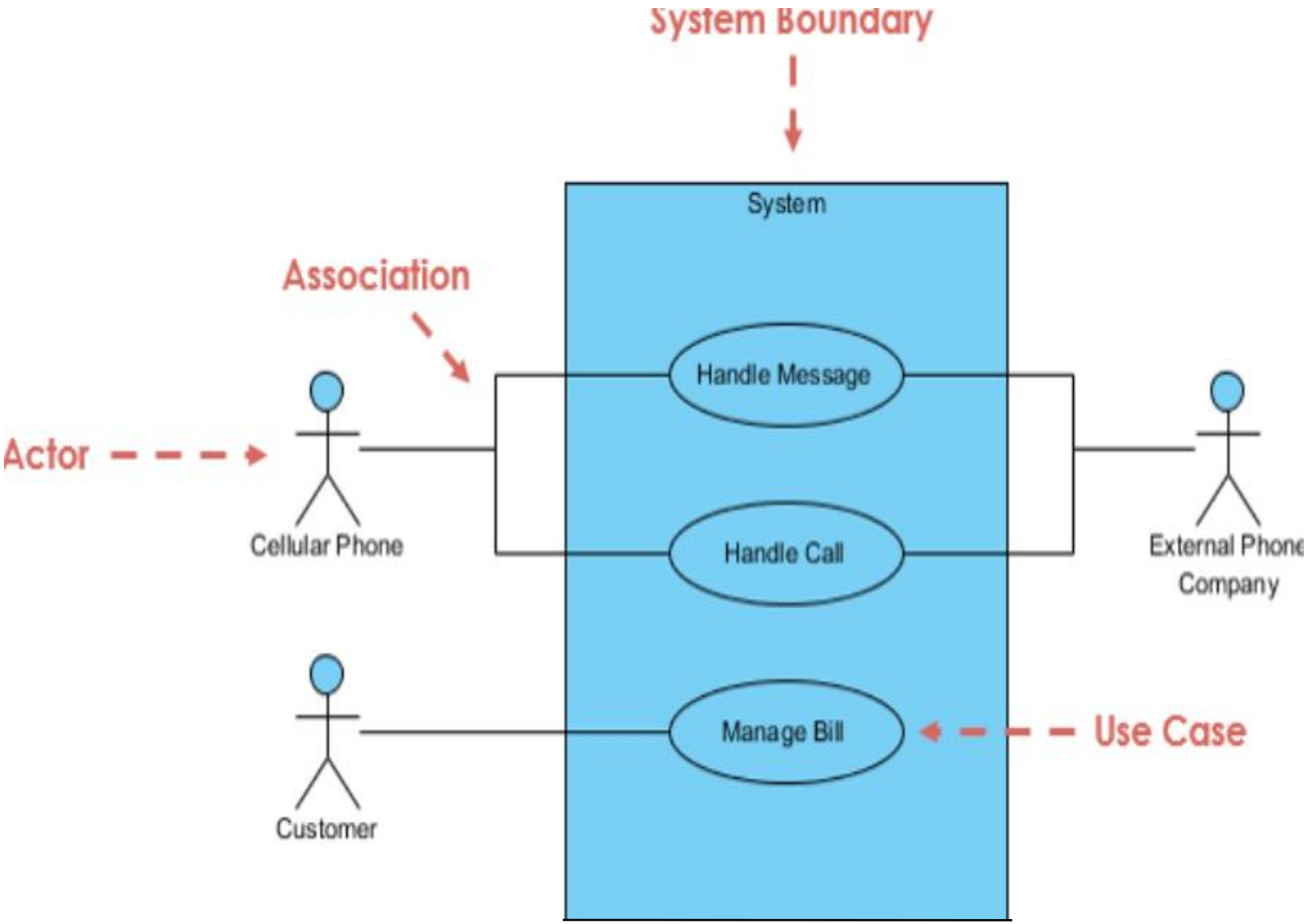
Generalization

- Generalization is the **relationship between a more general class and a more specific class.**
- Generalization is displayed as a directed line with a closed, hollow arrowhead at the superclass end.



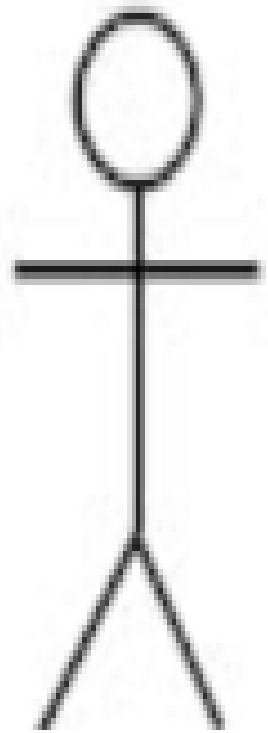
USE CASE DIAGRAM

- A use case diagram establishes the capability of the system as a whole.
- Components of use case diagram:
    - Actor
    - Use case
    - System boundary
    - Relationship
    - Actor relationship



Actor

- An actor is someone or something that must interact with the system under development
- Actors are not part of the system they represent anyone or anything that must interact with the system.
- Actors carry out use cases and a single actor may perform more than one use cases.
- An actor may
  - Input information to the system.
  - Receive information from the system.
  - Input to and out from the system.



Actor

4-Categories of an actor

- Principle : Who uses the main system functions.
- Secondary : Who takes care of administration & maintenance.
- External h/w : The h/w devices which are part of application domain a must be used.
- Other system: The other system with which the system must interact.

USE CASE

- A use case is a pattern of behavior, the system exhibits
- Each use case is a sequence of related transactions performed by an actor and the system in dialogue.
- USE CASE is dialogue between an actor and the system.

USE CASE documentation example

- The following use case describes the process of opening a new account in the bank.

Use case : Open new account

Actors : Customer, Cashier, Manager

Purpose : Like to have new saving account.

Description:

A **customer** arrives in the bank to open the new account. Customer requests for the new account form, fill the same and submits, along with the minimal deposit. At the end of complete successful process customer receives the passbook.

Type : Primary use case.

Relationship

- Relationship between use case and actor.
  - Communicates
- Relationship between two use cases
  - Extends – is used when you have one use case that is similar to another use case but does a bit more.
  - Uses
- Notation used to show the relationships:

<< >>