# UML - Class Diagram

2021-’22 Winter SWE B.Tech

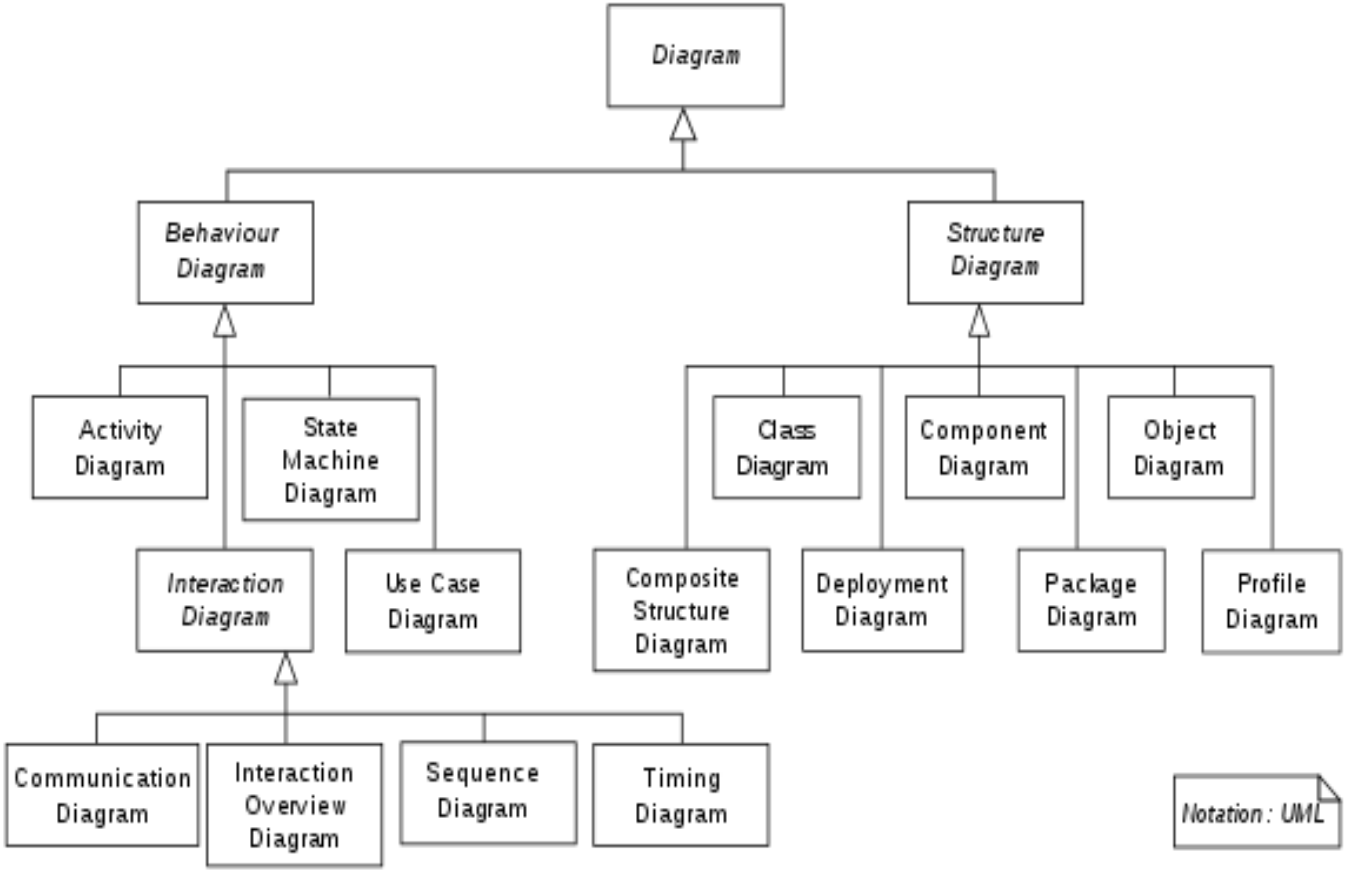


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Use Case Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Communication Diagram
Interaction Overview Diagram
Sequence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

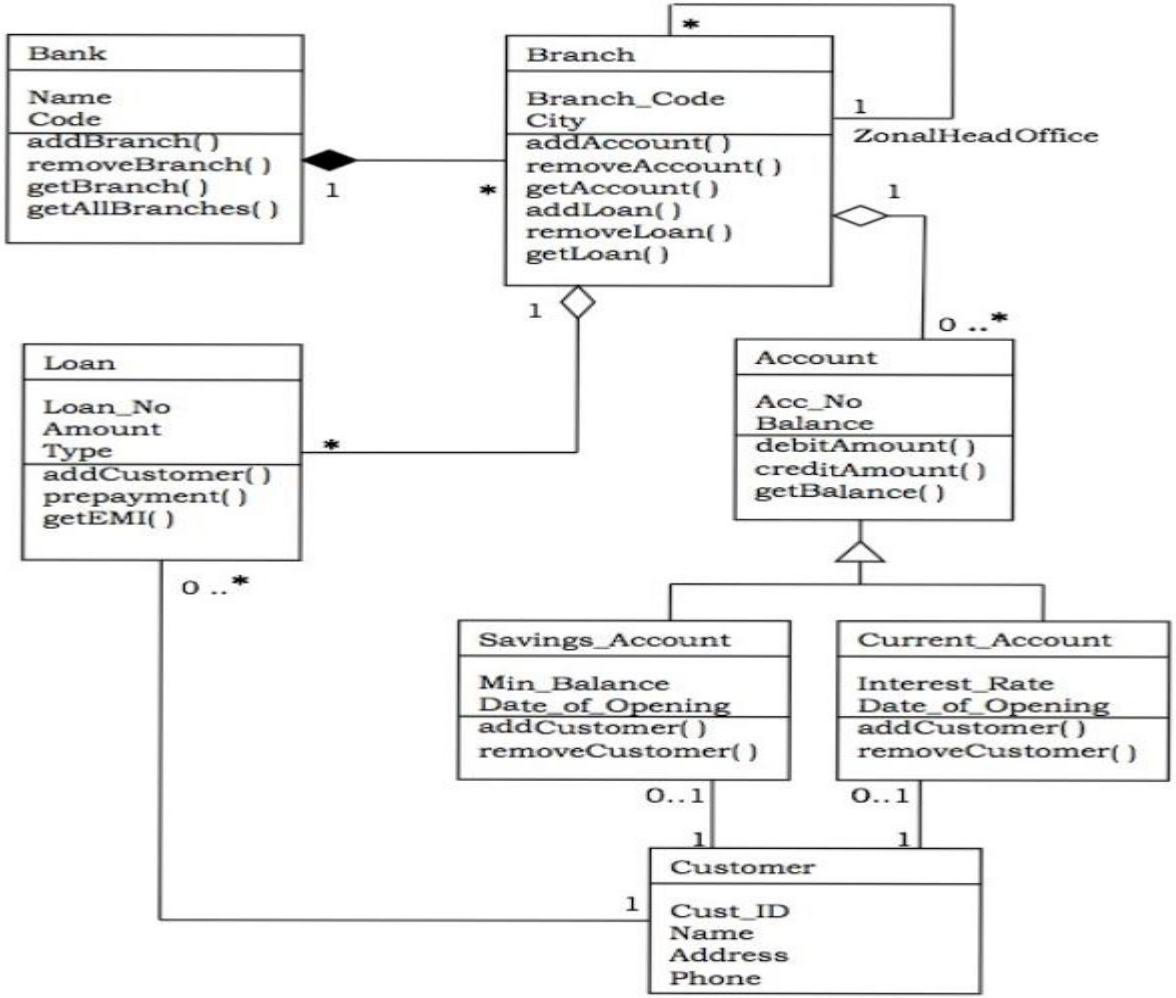
# UML Class Diagram

- A type **of static structure diagram**

- In the design of a system**, a number of classes are identified and grouped together that helps to determine the static relations** between them.

# Essentials of UML Class Diagrams

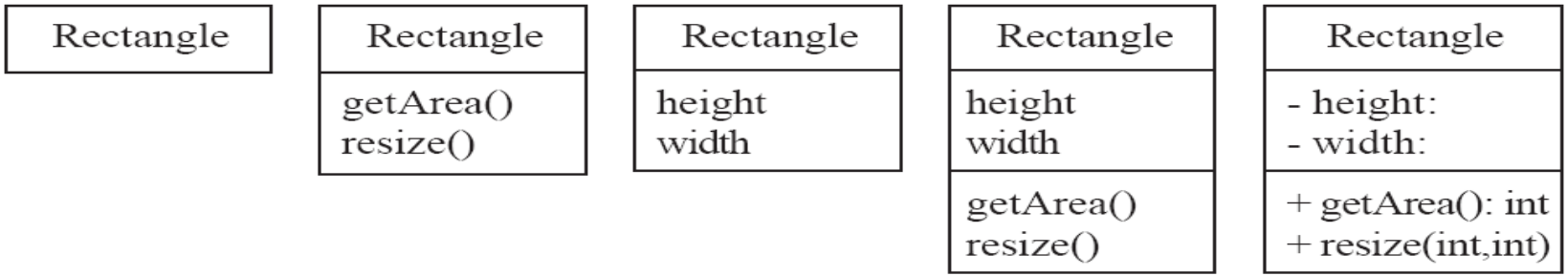
- *The main symbols shown on class diagrams are:*
  - ***Classes***
    - represent the types of data themselves
  - ***Attributes***
    - are simple data found in classes and their instances
  - ***Operations***
    - represent the functions performed by the classes and their instances
  - ***Associations***
    - represent linkages between instances of classes
  - ***Generalizations***
    - group classes into inheritance hierarchies


Bank
Name
Code
addBranch( )
removeBranch( )
getBranch( )
getAllBranches( )
Branch
Branch_Code
City
addAccount( )
removeAccount( )
getAccount( )
addLoan( )
removeLoan( )
getLoan( )
ZonalHeadOffice
Loan
Loan_No
Amount
Type
addCustomer( )
prepayment( )
getEMI( )
Account
Acc_No
Balance
debitAmount( )
creditAmount( )
getBalance( )
Savings_Account
Min_Balance
Date_of_Opening
addCustomer( )
removeCustomer( )
Current_Account
Interest_Rate
Date_of_Opening
addCustomer( )
removeCustomer( )
Customer
Cust_ID
Name
Address
Phone
1
*
0 ..*
0..1

# Classes

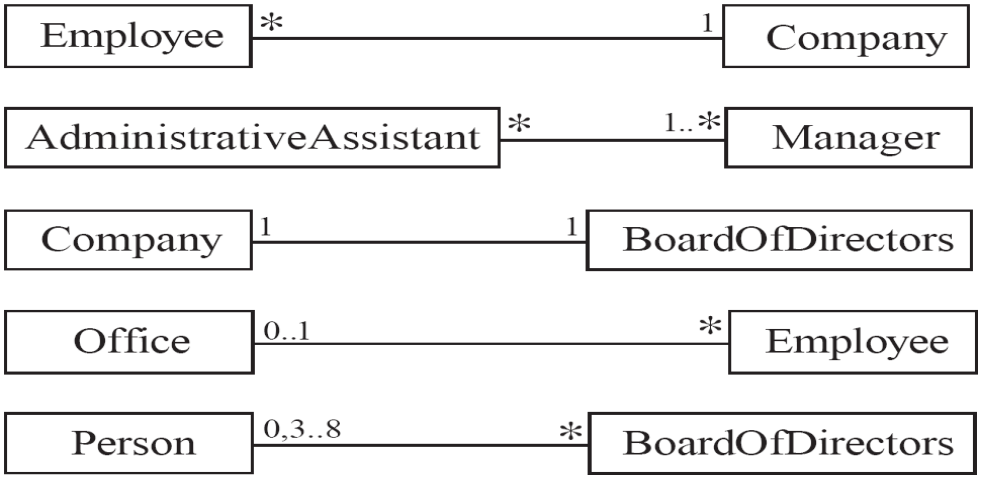
- A class is simply represented as a box with the name of the class
- The complete signature of an operation is:

  operationName(parameterName: parameterType …): returnType



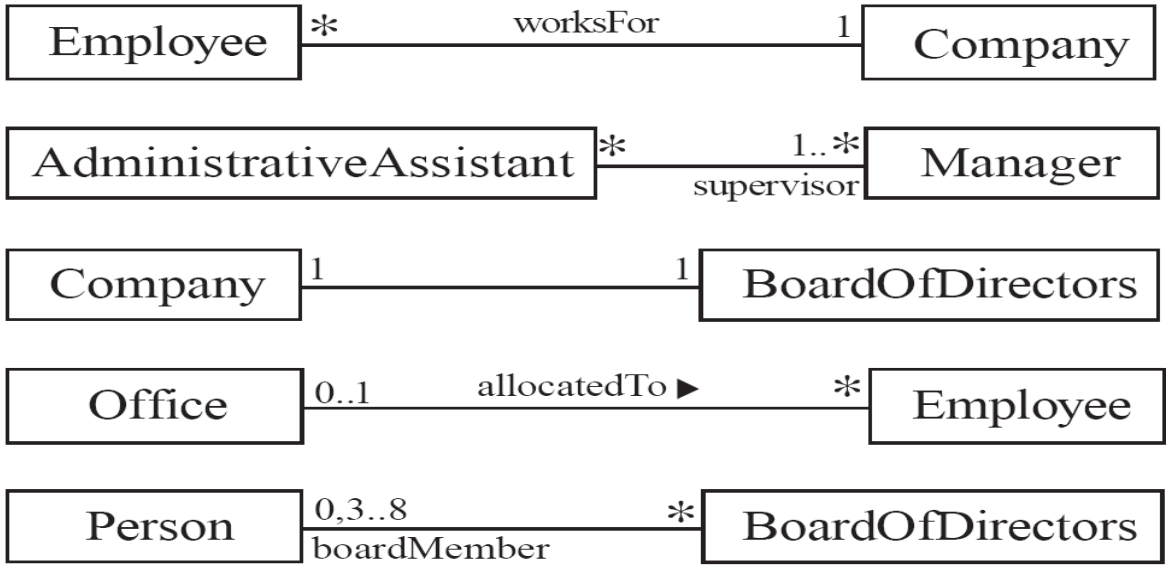
# Associations and Multiplicity

An ***association*** is used to show how two classes are related to each other. Symbols indicating ***multiplicity*** are shown at each end of the association



# Labelling associations

- Each association can be labelled, to make explicit the nature of the association



# Analyzing and validating associations

- **One to many**
  - A company has many employees,
  - An employee can only work for one company.
  - A company can have zero employees
  - It is not possible to be an employee unless you work for a company

# Analyzing and validating associations

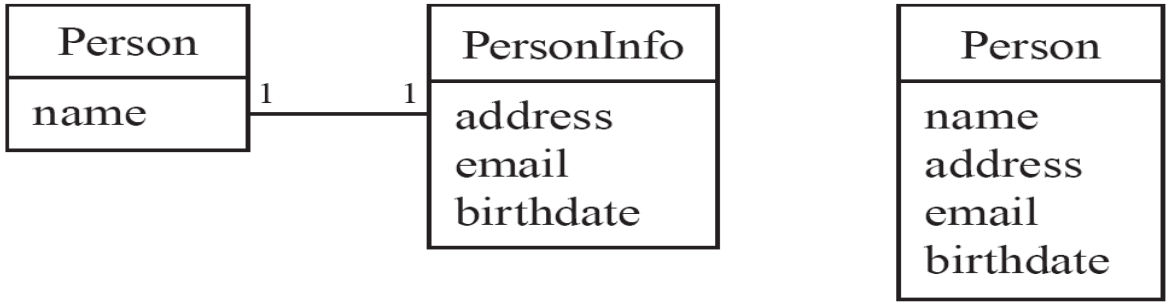
- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Managers can have a group of assistants
  - Some managers might have zero assistants.
  - Is it possible for an assistant to have, perhaps temporarily, zero managers?

# Analyzing and validating associations

- **One-to-one**
  - For each company, there is exactly one board of directors
  - A board is the board of only one company
  - A company must always have a board
  - A board must always be of some company

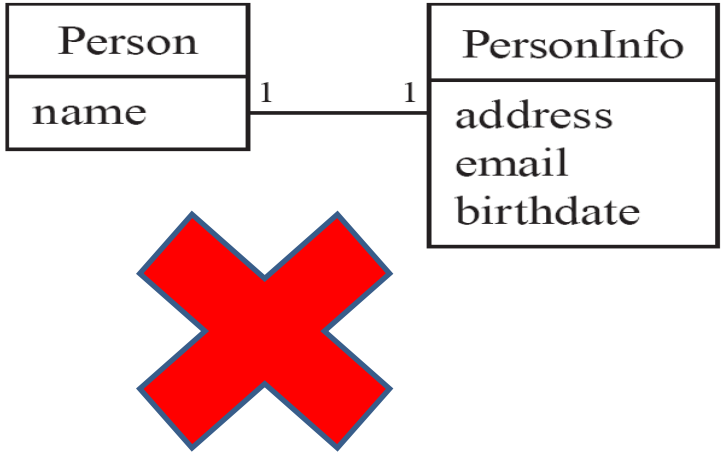
# Analyzing and validating associations

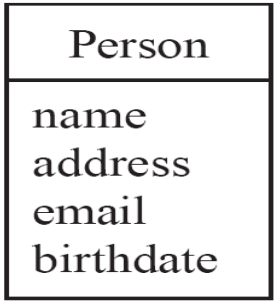
- Avoid unnecessary one-to-one associations



# Analyzing and validating associations

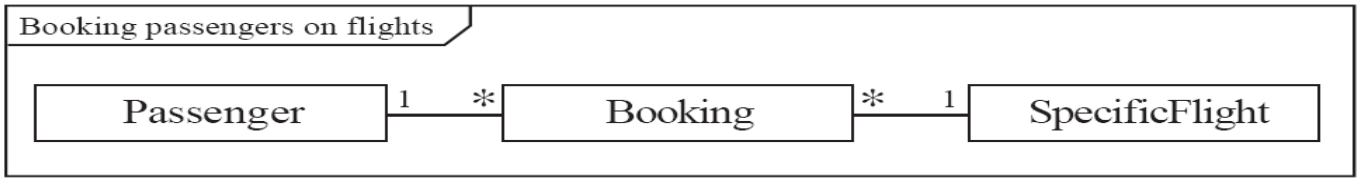
- Avoid unnecessary one-to-one associations



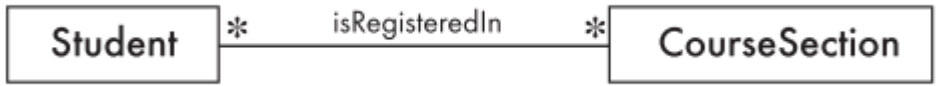



# A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



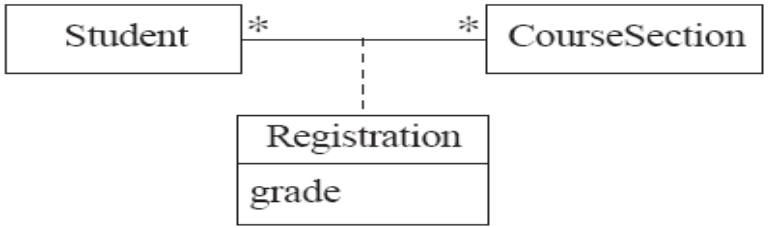
# Association classes




Grade???

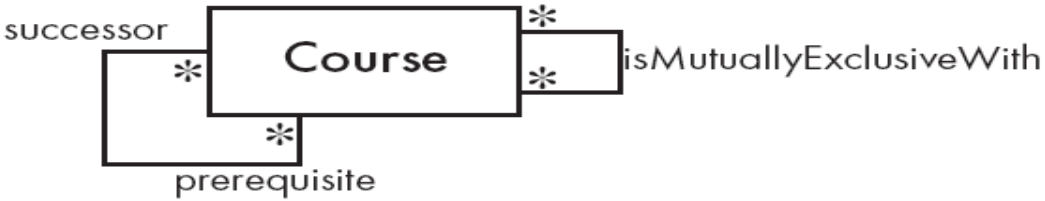
# Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent



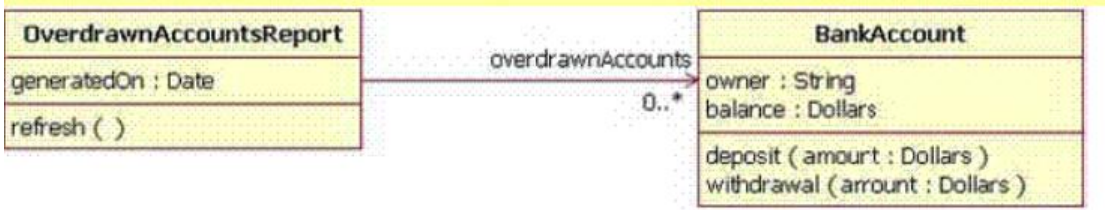
# Reflexive associations

- It is possible for an association to connect a class to itself



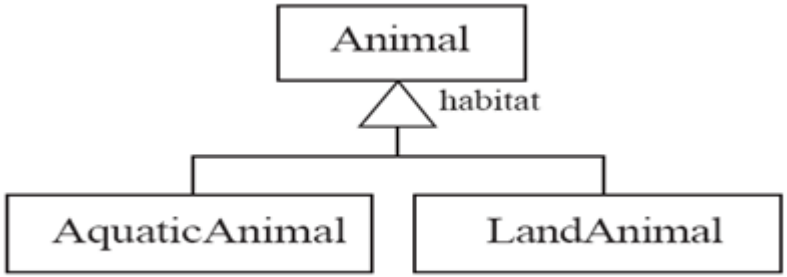
# Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end

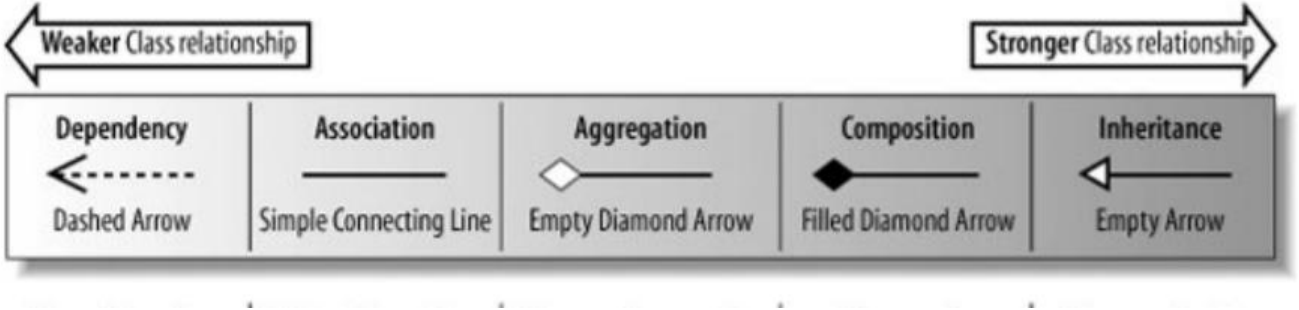


# Generalization

- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labeled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization



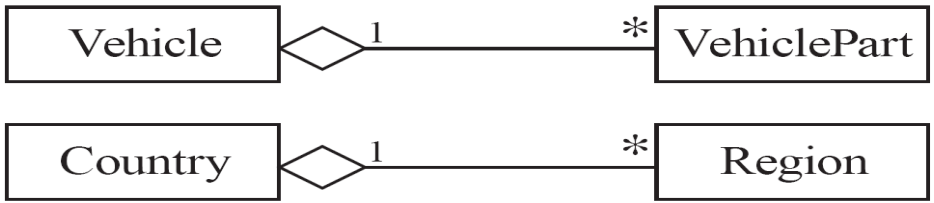
# Relationships



# Aggregation

Aggregations are special associations that represent ‘part-whole’ relationships.

- The ‘whole’ side is often called the *assembly* or the *aggregate*
- This symbol is a shorthand notation association named isPartOf



# When to use an aggregation

As a general rule, you can mark an association as an aggregation if the following are true:

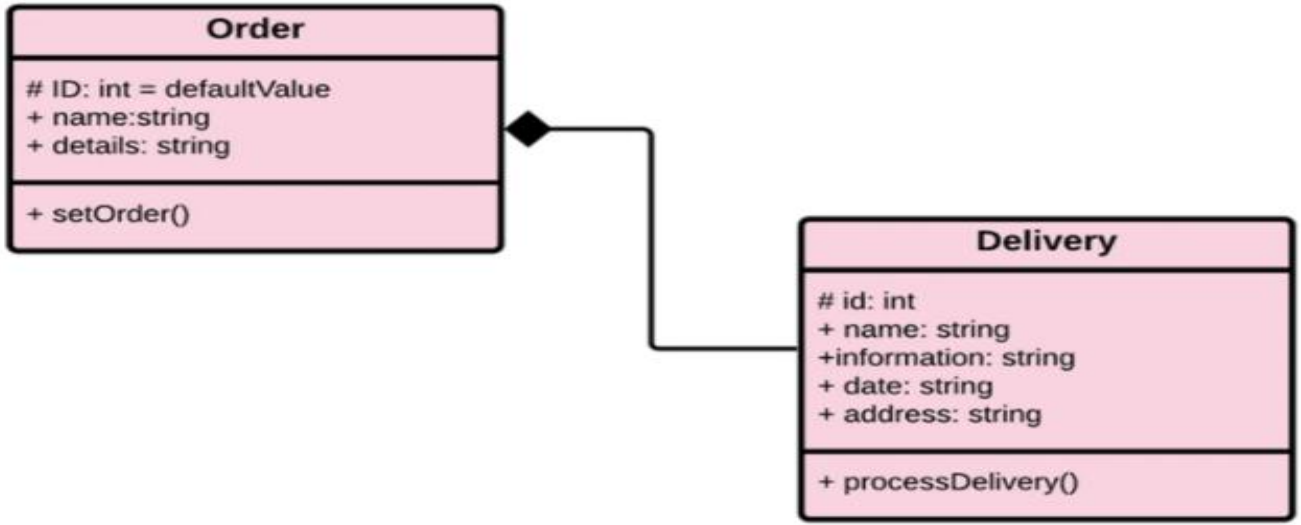
- *The parts **‘are part of’** the aggregate or the aggregate **‘is composed of’** the parts*
- *When something **owns or controls** the aggregate, then they also own or control the parts*

# Composition

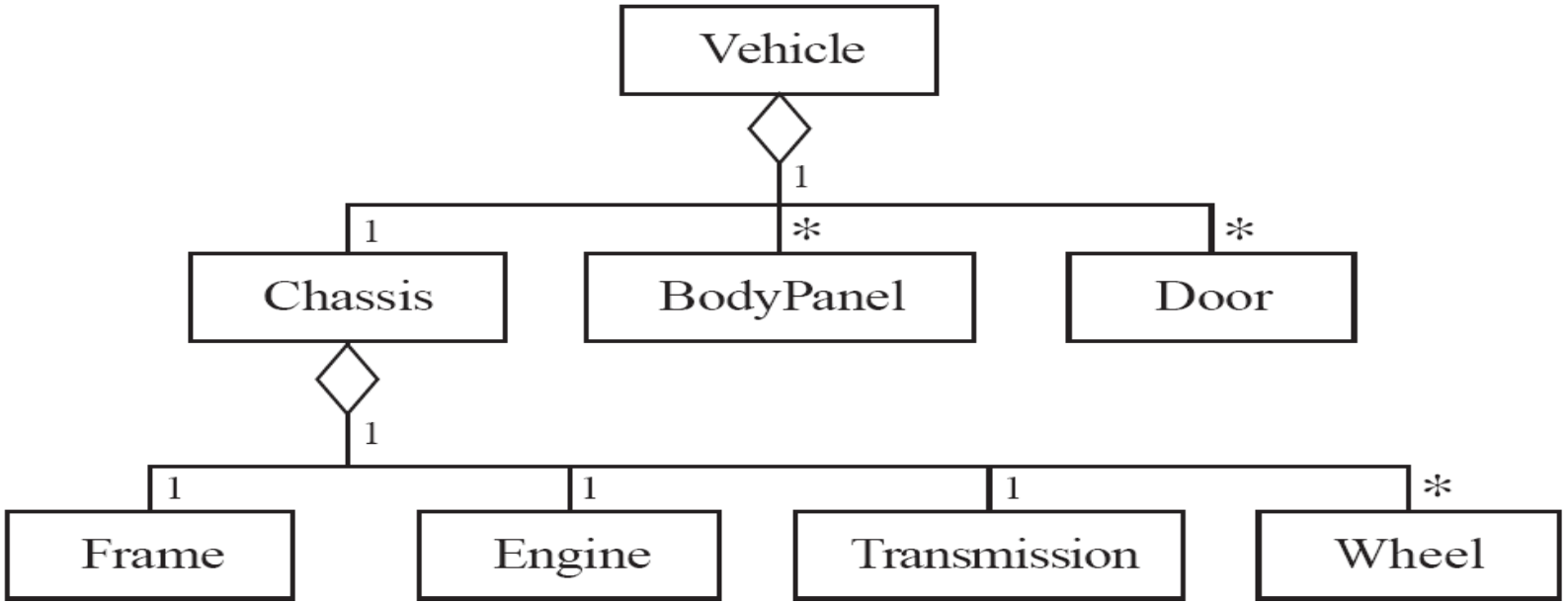
A *composition* is a strong kind of aggregation

- if the aggregate is destroyed, then the parts are destroyed as well

# Composition Example



# Aggregation hierarchy



# Propagation

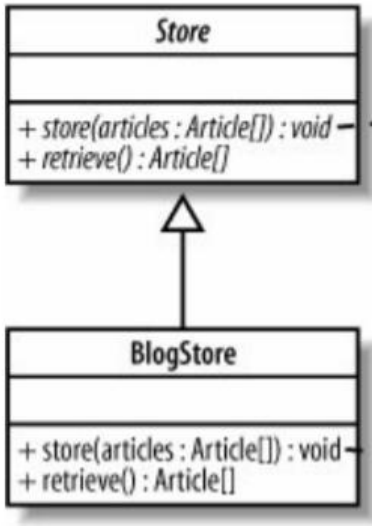
- A mechanism where an **operation in an aggregate** is implemented by having the aggregate perform that operation on its parts
- At the same time, **properties of the parts are often propagated back** to the aggregate

# Abstract class

- When the concrete implementation of methods are left for the subclasses.
- Can contain both abstract and non-abstract methods

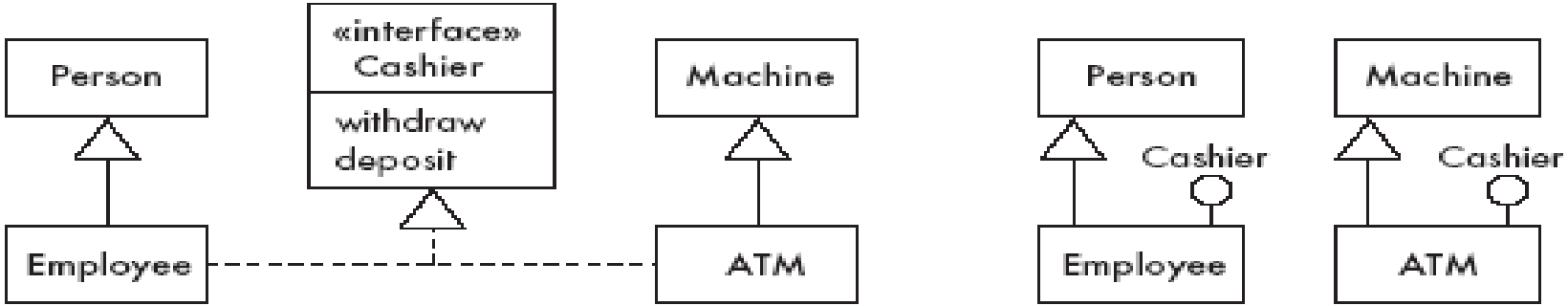

```
public abstract class Store {
   public abstract void store(Article[] articles);
   public abstract Article[] retrieve(   );
}
```

# Abstract class



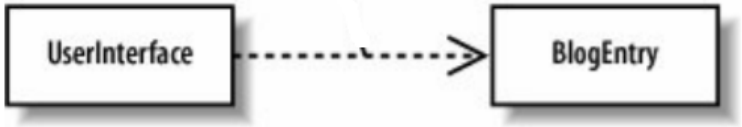
# Interfaces

- An *interface* is similar to a class, except it lacks instance variables and implemented methods
- An interface describes a *portion of the visible behaviour* of a set of objects.

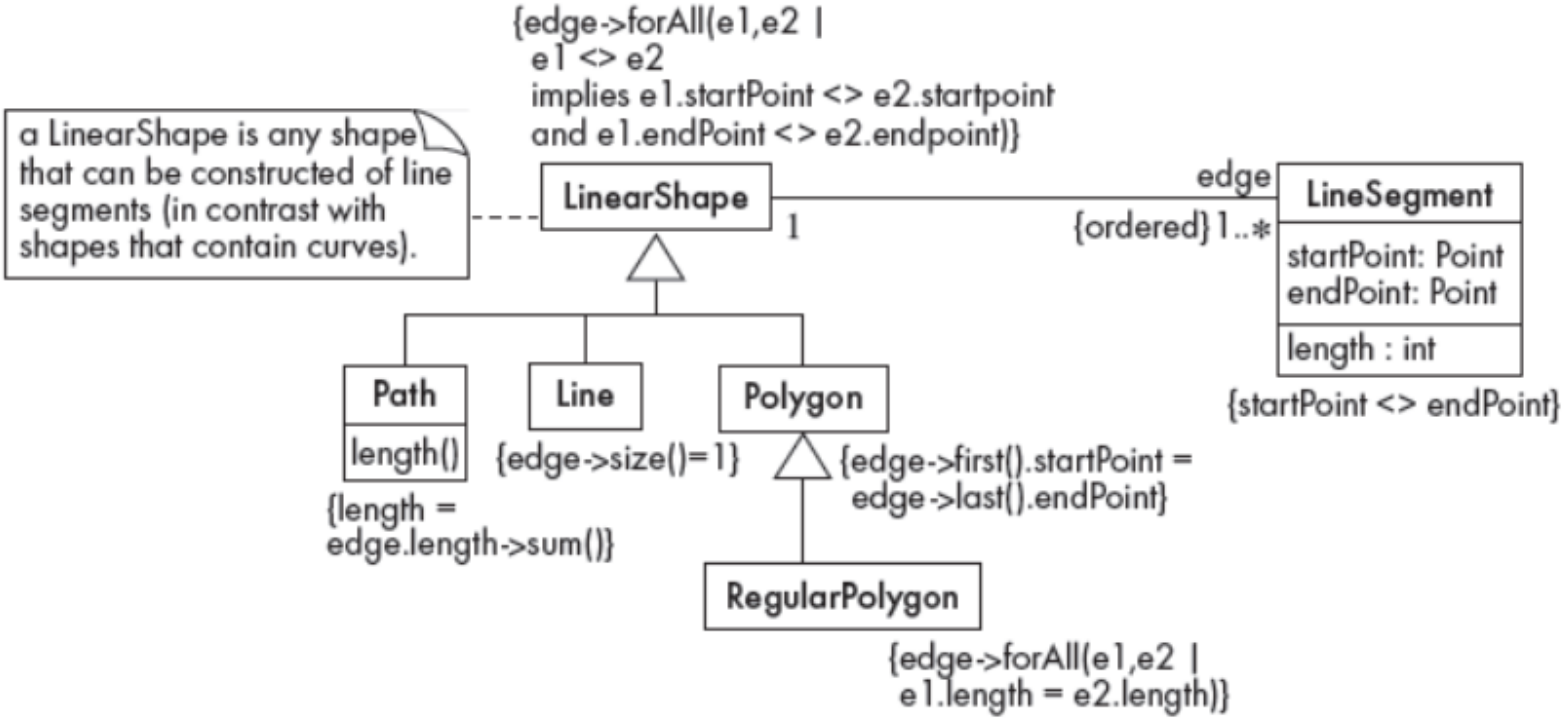


# Dependency

- A class needs to know about the other class in order use it’s objects
- When the UserInterface wants to display, it accesses BlogEntry




- Dependency implies only that the classes can work together, so is the weakest relationship


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered}1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

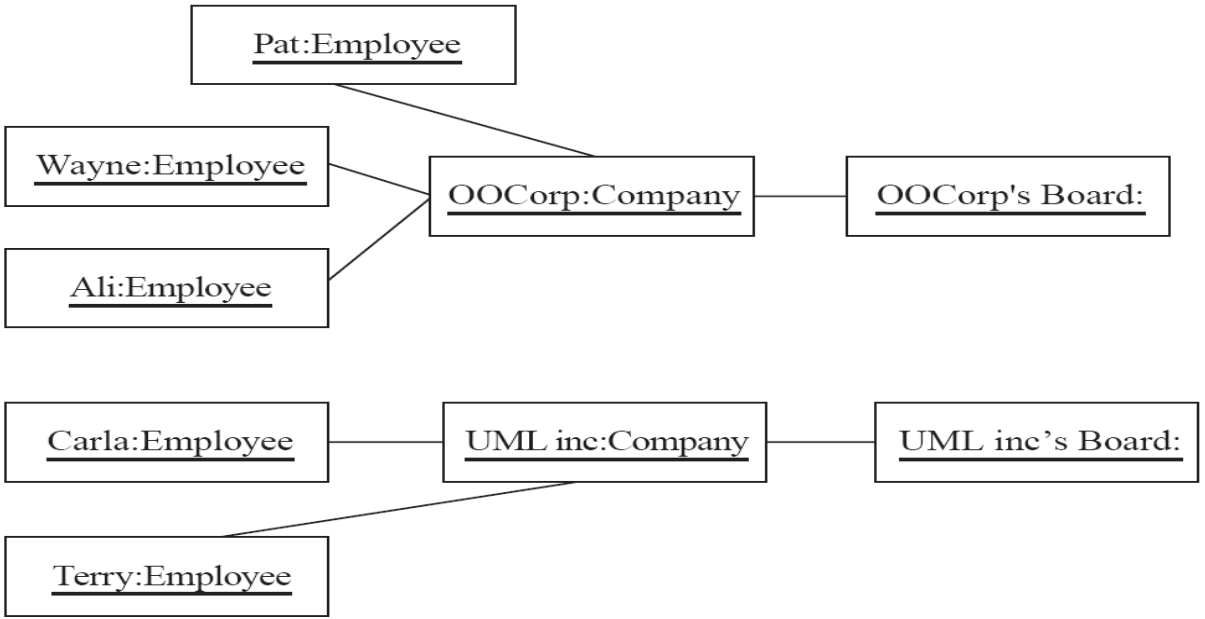
# Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces

*Don't be too disorganized. Don't be too rigid either.*

# Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



# Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses