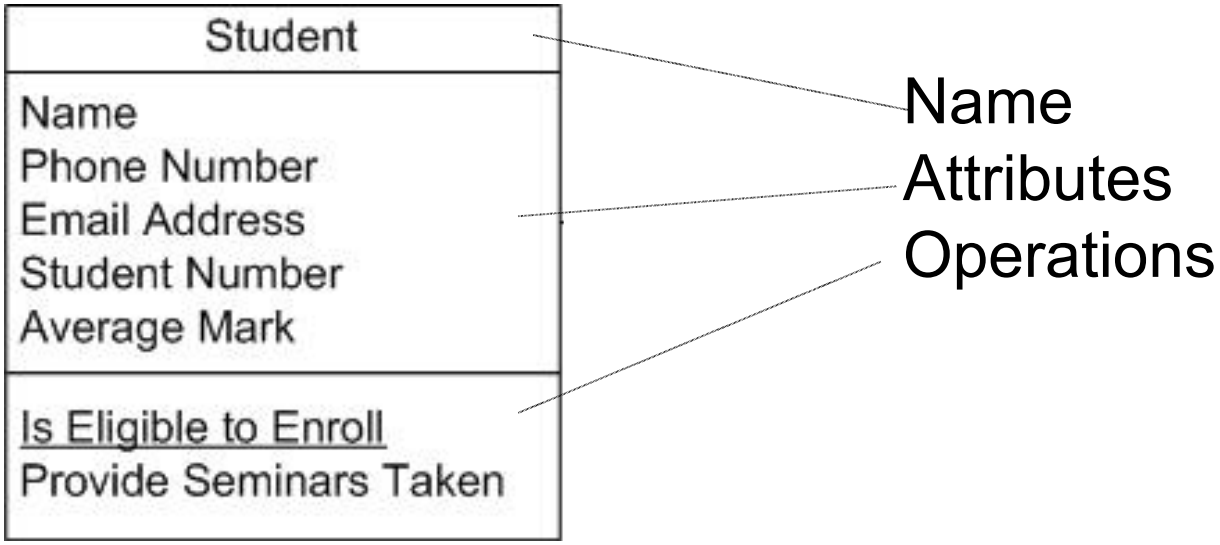
Métodos de Desenvolvimento de Software (MDS) 2014/2015

Diagrama de classes (Domínio)

Classes

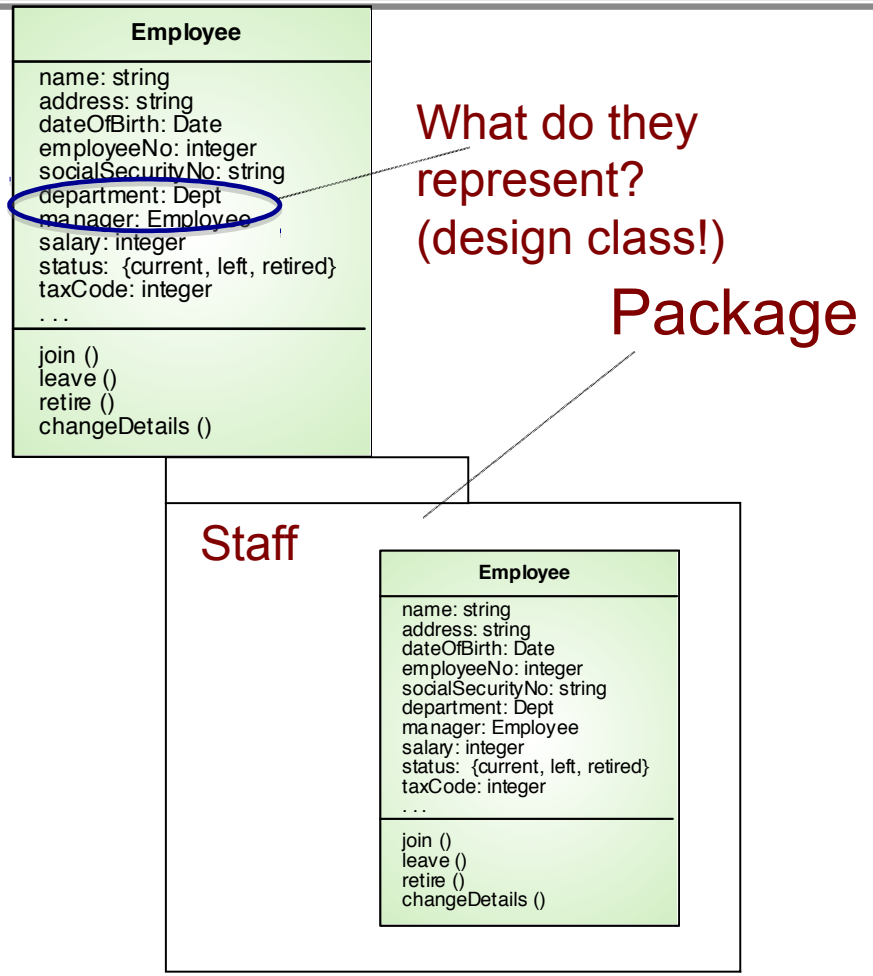
- Describes a set of objects that share the same attributes, operations, associations, and semantics.



Name of the class

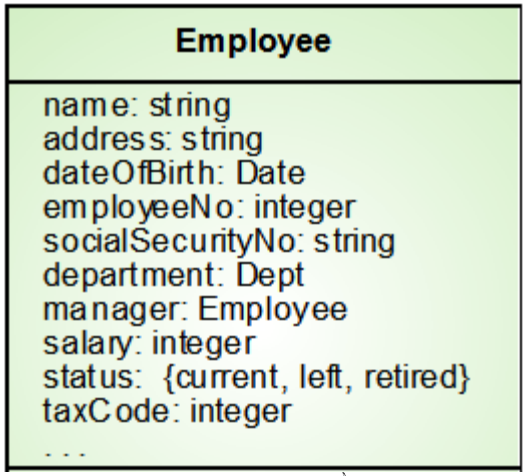
- Identity
  - Noun – Domain vocabulary

- Simple Name
  - Name of the package followed by the name of the class - pth (Staff::Employee);
  - Different Classes with the same names only possible if in different packages



Attributes

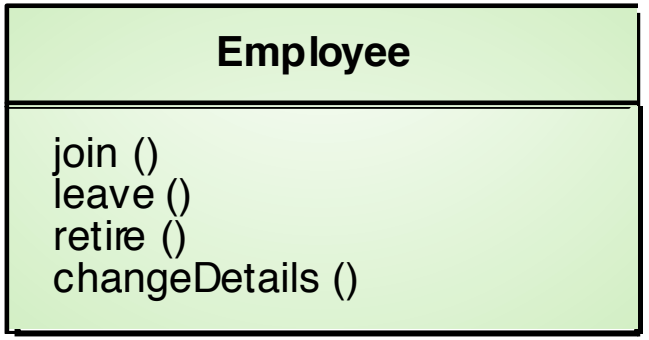
- Property of a class
  - noun
- Describes a set of values that is part of an instance



Hidden operations

Operations

- Abstract something the object can realize
  - Described with a verb to represent the class behaviour
- Shared by all objects from that class



Hidden attibutes

Taxonomy of the operations

- Constructors
  - Create a new object(allocate memory space for an object)
- Selectors
  - Return a value (contained or referred by an object)
- Modifiers
  - Provoke change
  - State change might refer to change one or more attributes
- Destructors
  - Destroy the object (free memory)

Well-Formed Class

- Represents an abstraction of part of the problem
- Encapsulates a set of well defined responsabilities
- Offers a clear separation between specification and implementation
  - And between what is visible and hidden

Domain Class Diagram

- Contains the set of base classes of the problem(classes of type entity)
- This diagram will be later extended to contemplate other kinds of classes and dependencies between them

Techniques for identifying classes

- Extract names (nouns)
  - Good for identifying classes
  - Bad for identifying their requirements (characteristics and relations)
- Identify responsabilities
  - Demands for good knowledge of the domain

Name extraction

- Example:
  - The clients of a given bank can retrieve and credit money amounts in their accounts, or ask for the current balance. These operations are completed at the atm or at the counter. The transactions can be done by cheque, direct payment, or by atm with a card. There are two types of accounts: checking account and savings account. The savings account gives interests and can not be accessed by atm.
- Class candidates: client, bank, amount, account, balance, atm, counter, cheque, direct payment, card, checking account, savings account, interests
  - Nouns… almost all, …
- What is kept in the final list?

Identifyng CRC – Class, Responsabilities and Collaborations

- A responsability is a contract of a given class
- Steps
  - Identify sets of classes that cooperate to achieve a specific behaviour
  - Identify the responsabilities and operations and attributes that each class shall contain to carry its own responsabilities
  - Divide classes with too many responsabilities or join classes that are too simple
- Sources
  - Problem statement, Use Case Diagrams and its Scenarios, Activities Diagrams

Identify CRC (cont.)

- Responsability: is a contract or a mission of the class.

Ex.: TemperatureSensor class is responsible for measuring the temperature and activate the alarm if it reaches a given threshold

Class Responsibility Card):

Class Name	Temperature Sensor
Responsability	Colaborators
readTemperature()	Alarm
Temperature	

Generalization and Inheritance

- Classes can be organized hierarchicaly where the superclass is the generalization of one or more classes (subclasses)
- A subclass Inherits the attributes and operations of the superclass and can add more properties. (and also inherits the dependencies!)
- Generalization in UML is implemented as inheritance in OOP

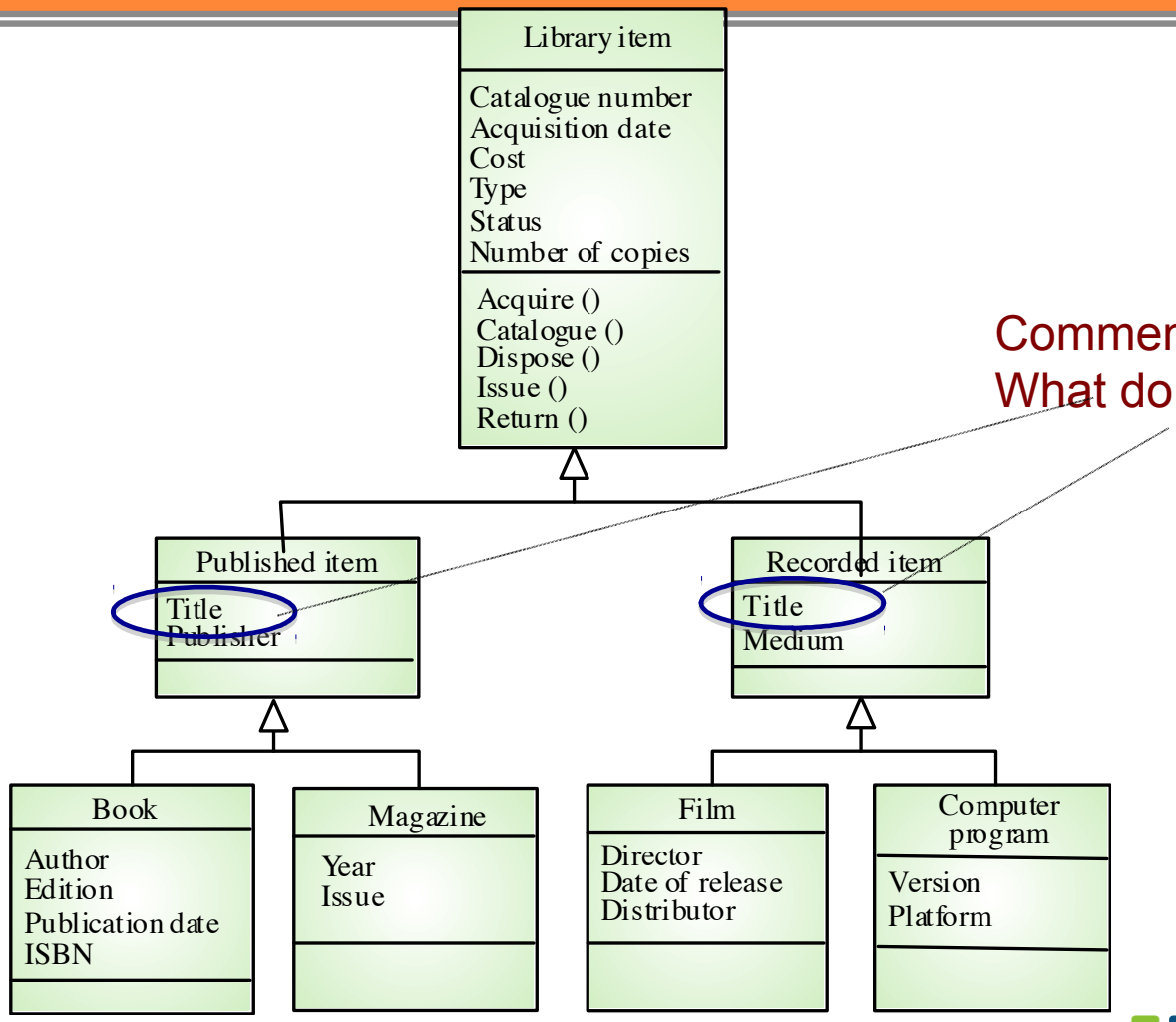
Inheritance: the substitution principle

- In object-oriented programming, the Liskov substitution principle is a particular definition of subtype that was introduced by Barbara Liskov and Jeannette Wing in a 1993 paper entitled Family Values: A Behavioral Notion of Subtyping [1]. (It is not the only definition; see datatype.)
- The principle was formulated succinctly [2] as follows:
  - Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.
- Thus, Liskov and Wing's notion of "subtype" is based on the notion of substitutability; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).

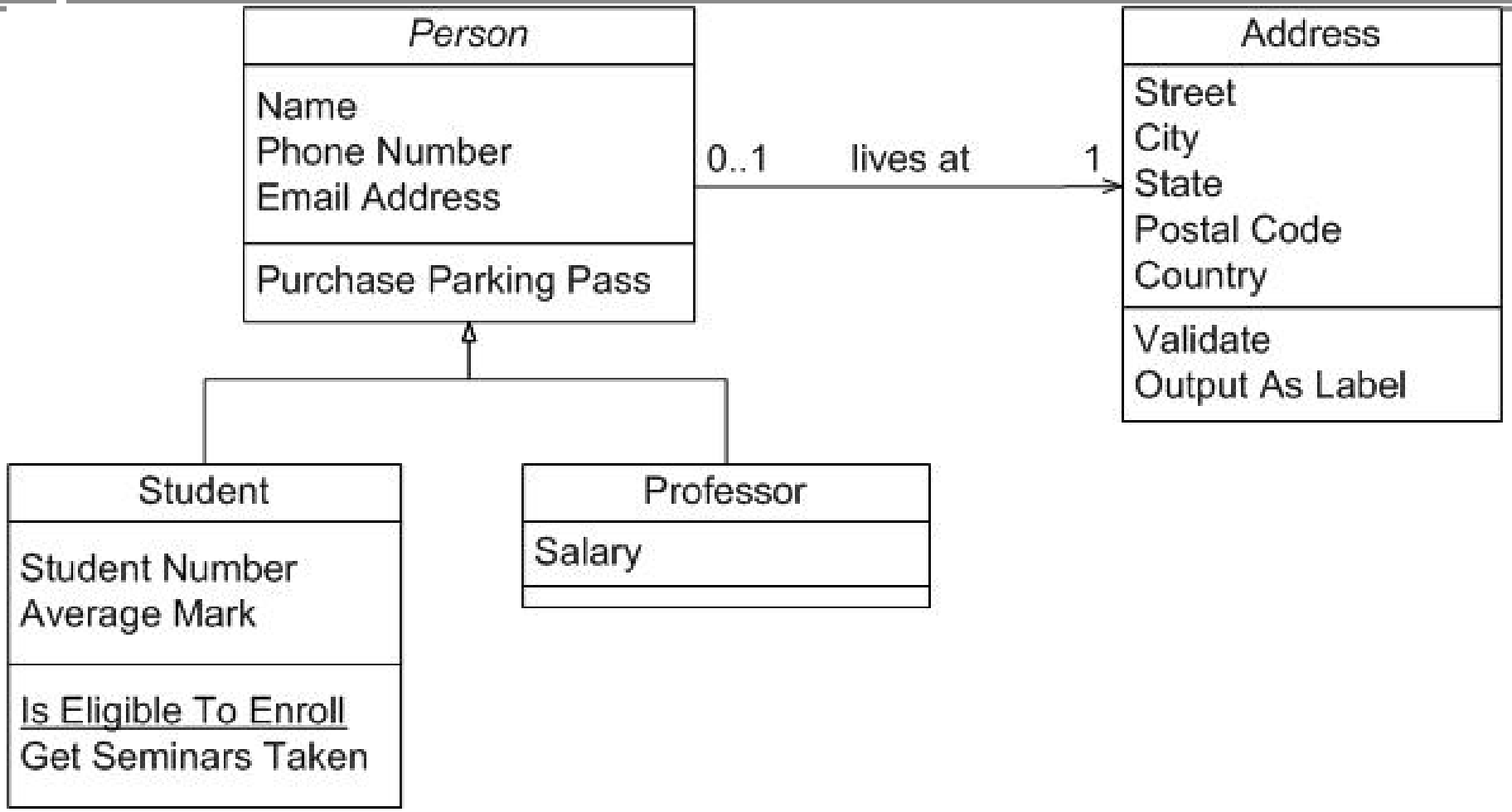
Generalization and Inheritance

- Advantages of Inheritance
  - Abstraction mechanism/ classify entities
  - Mechanism for reuse
- Problems
  - We can only understand classes if we know their superclasses
  - Sometimes the inheritance graph is not compatible with efficiency

Class hierarchy in a Library System



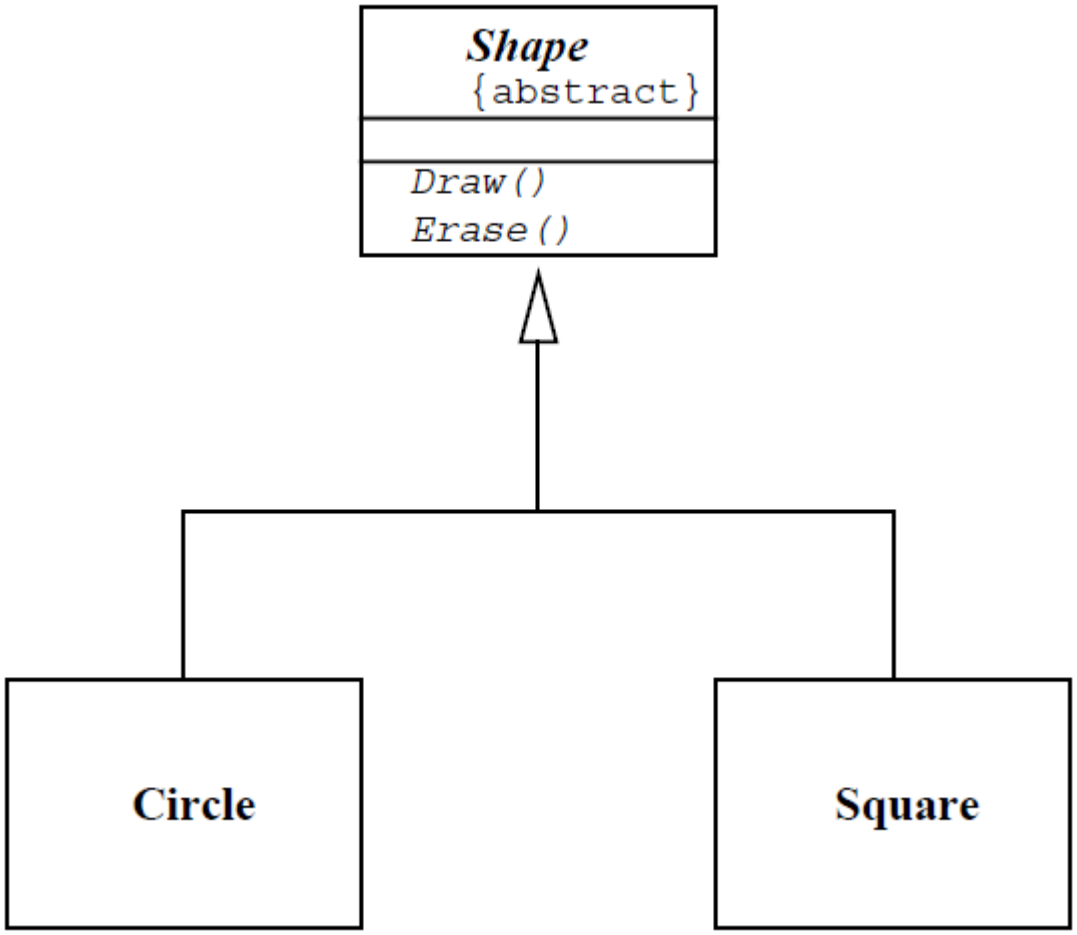
Example


Person
Name
Phone Number
Email Address
Purchase Parking Pass
0..1
lives at
1
Address
Street
City
State
Postal Code
Country
Validate
Output As Label
Student
Student Number
Average Mark
Is Eligible To Enroll
Get Seminars Taken
Professor
Salary

Abstract Classes

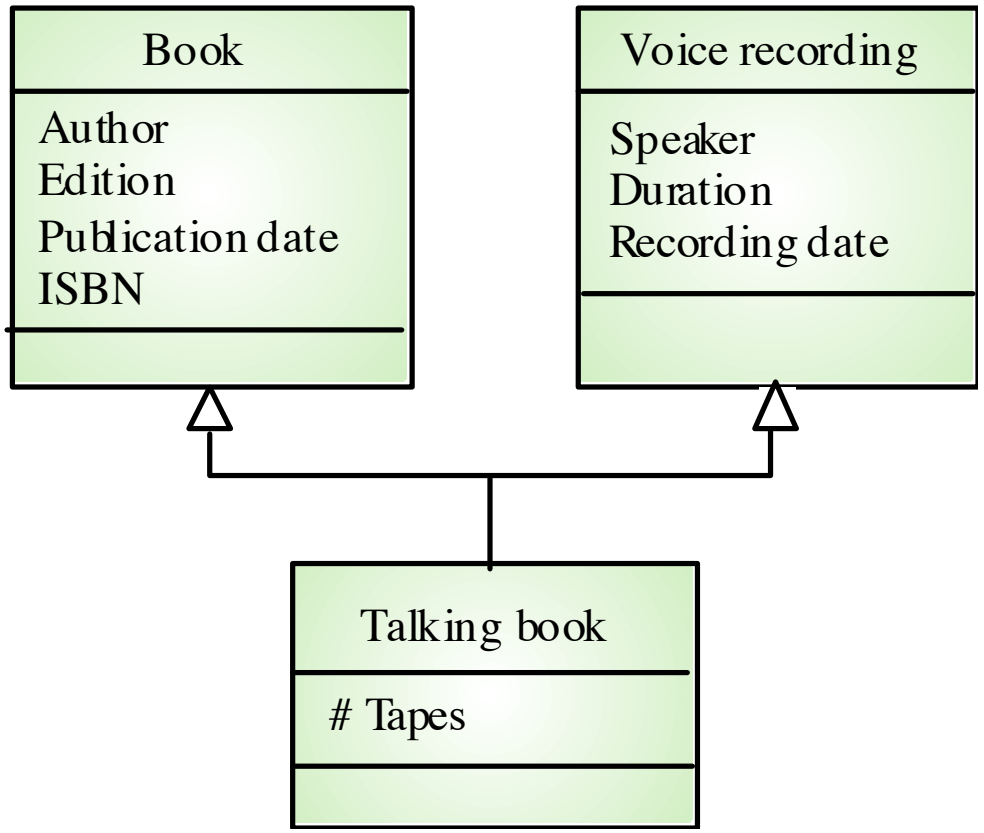
- When the superclass was created to factorize common properties to a set of classes of the same type, but contain one or more abstract operations (without implementation)
- The Abstract Classes are only used for class inheritance
- Represented by their names in italic, or with the stereotype <<abstract>>, or tagged value {abstract}
- Abstract classes can not be instantiated

Example of an abstract class



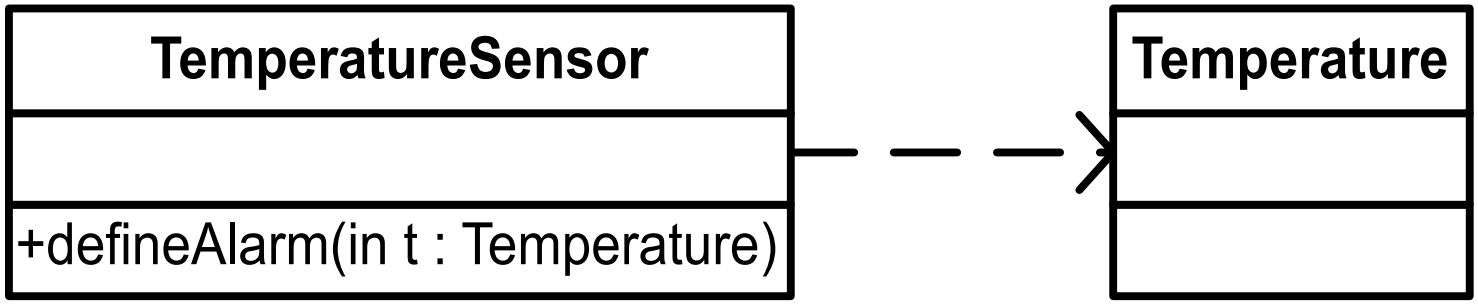
Multiple inheritance

- A class can inherit attributes and operations of several superclasses
- Can lead to conflicts when the operations of different superclasses have the same name (and do different things)
- The hierarchy is more complex to understand



Dependency

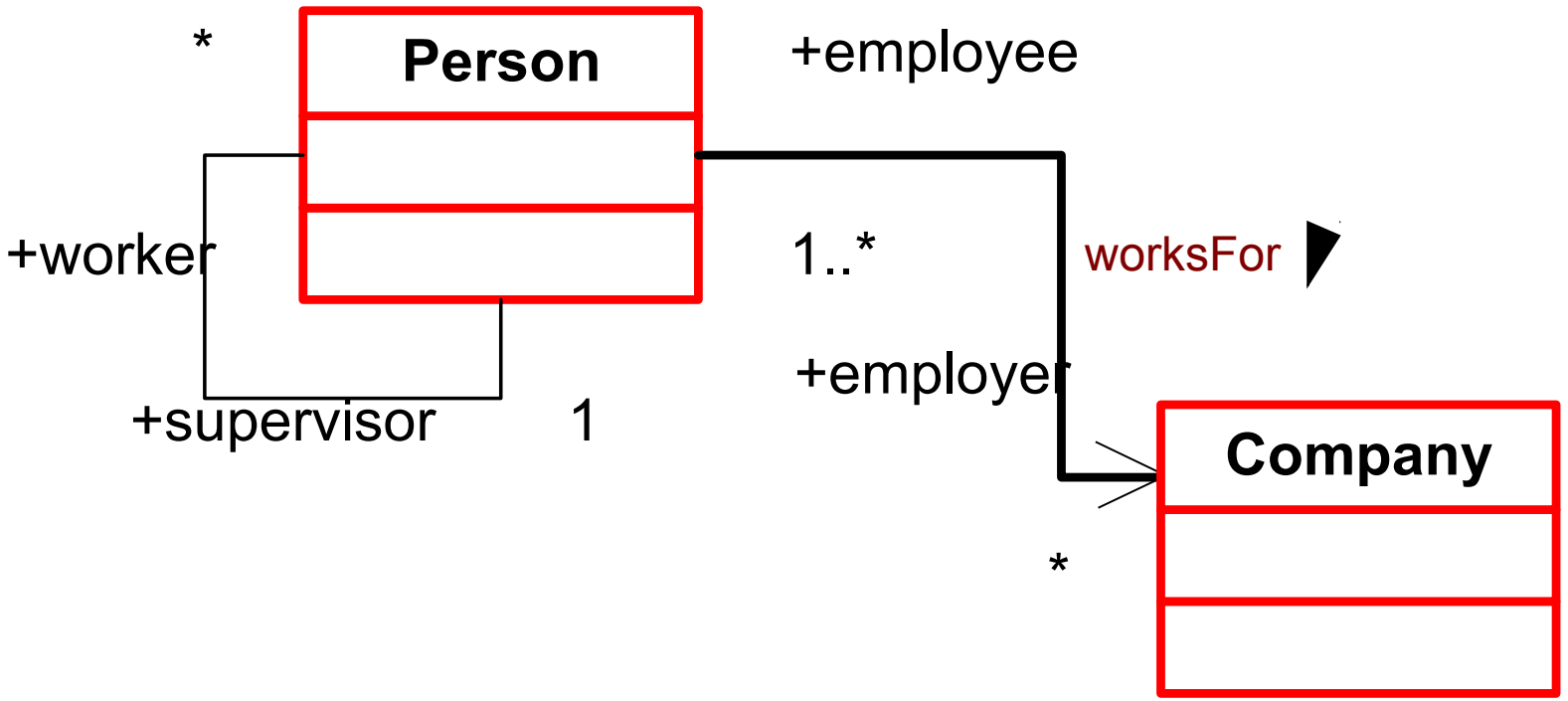
- It is a relationship that determines whether if a change in the specification of a class can affect another class, but not necessarily the opposite

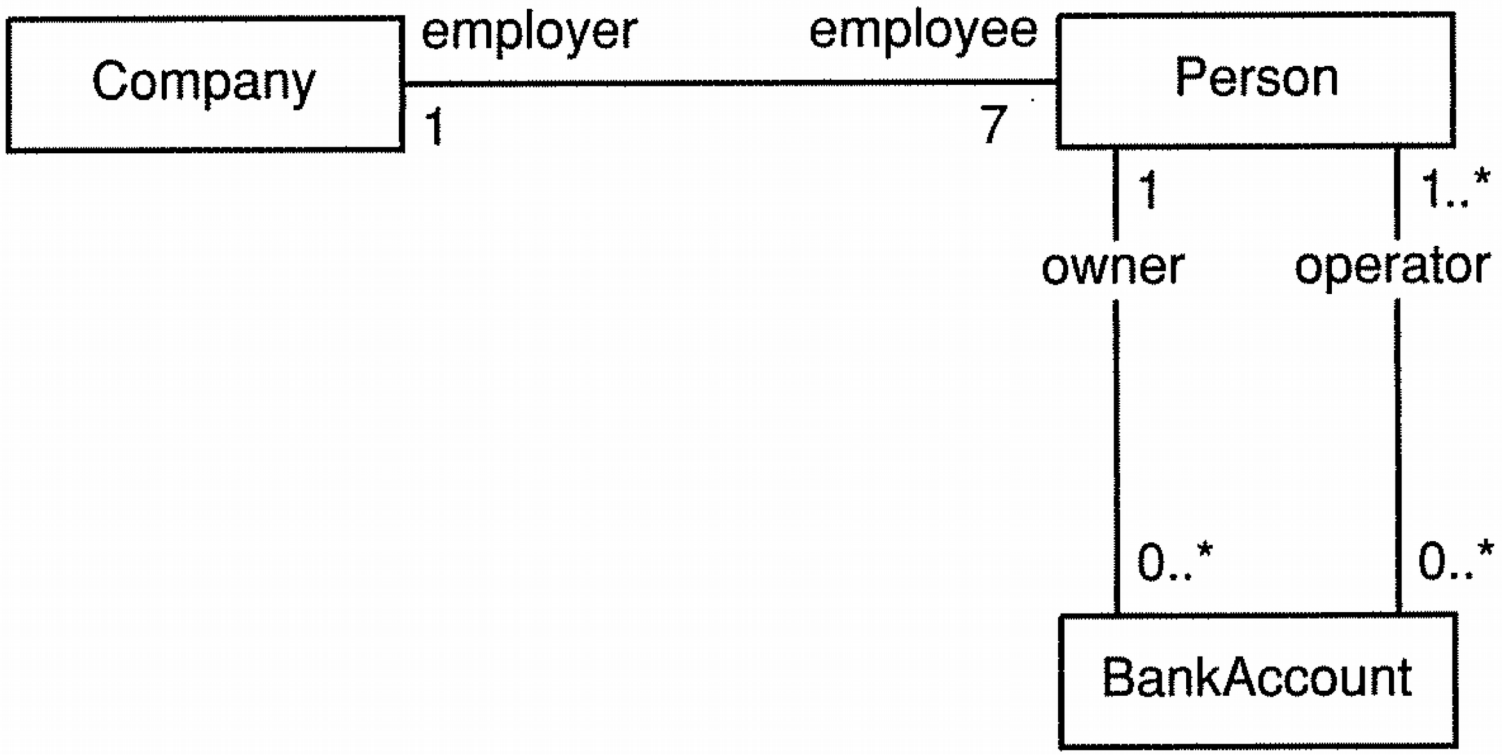


Associations in UML

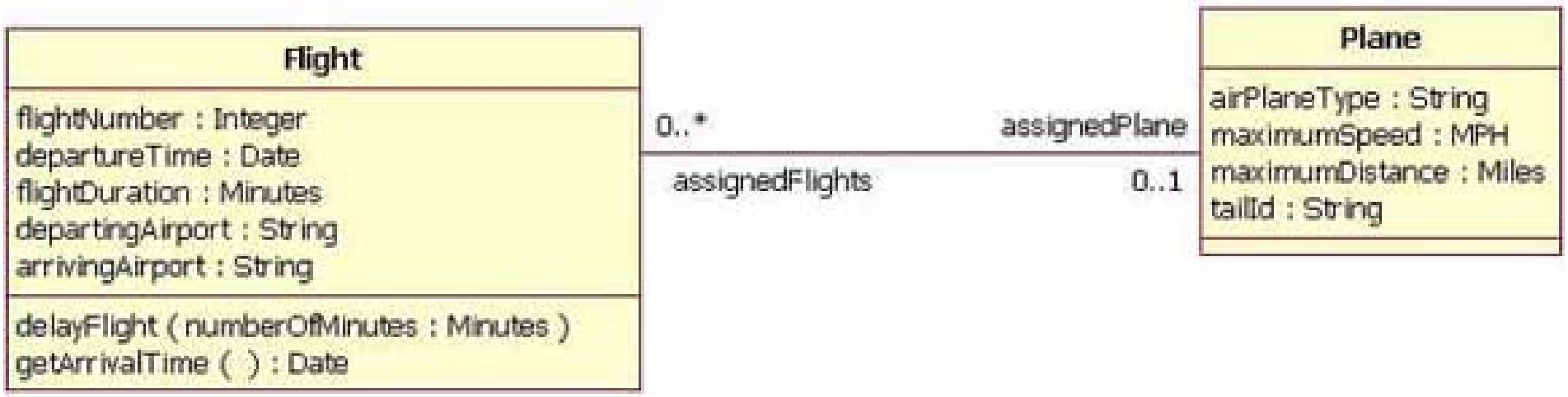
- Objects of a class are linked to objects from another class(es)
- Must have a name
- **Role**: When a class is in an association it must play a specific role
- **Multiplicity**: the specification of the nº of elements that a set have
  - Eg.: 1..*, *, 0..1, 3..9, 3..*, etc.
- **Navegation**: shows how from a class instance we can access to one or more instances of another class

Associations

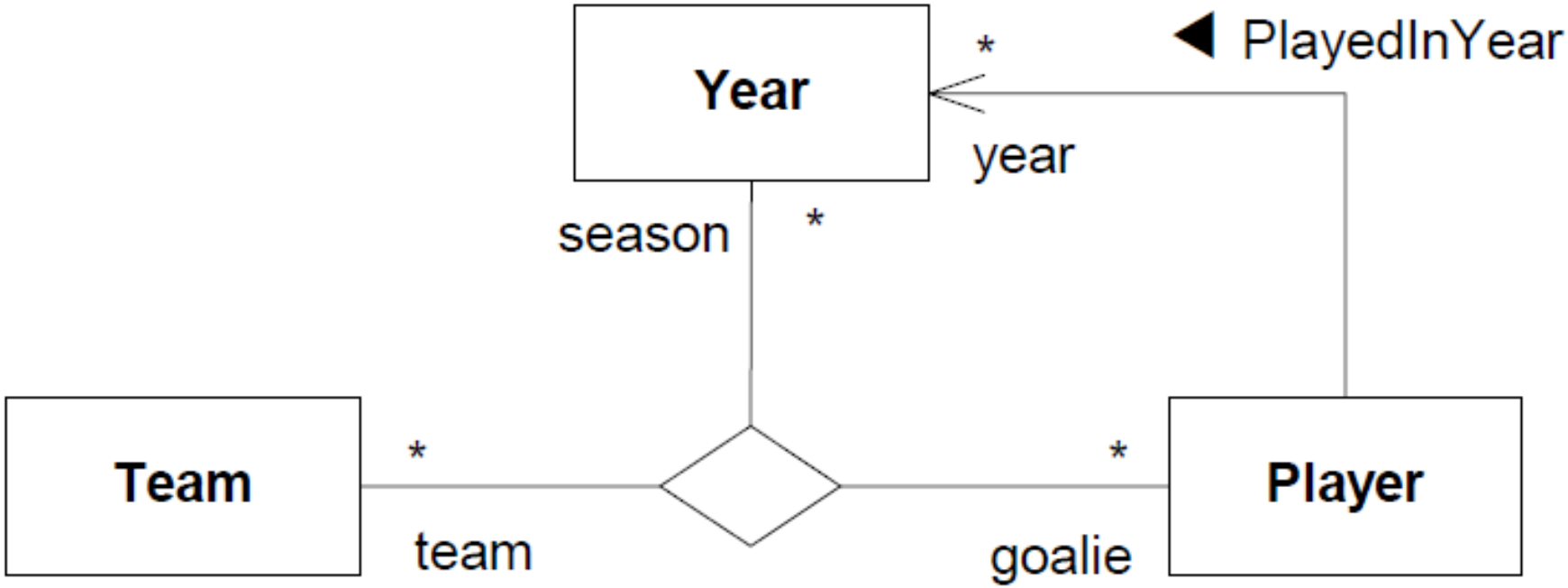



Company
employer
1
employee
7
Person
1
owner
1..*
operator
0..*
0..*
BankAccount

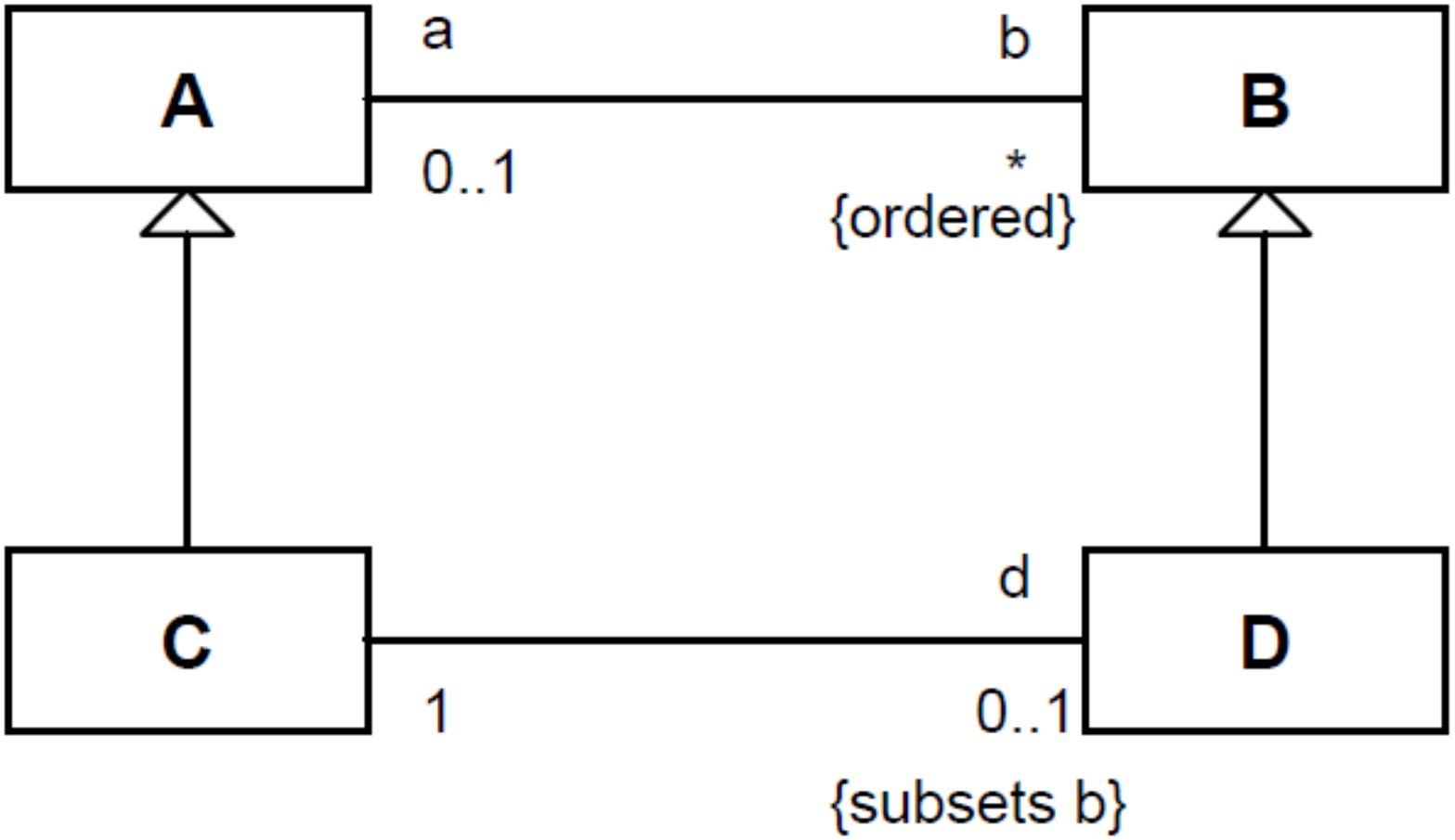
Roles


Flight
flightNumber : Integer
departureTime : Date
flightDuration : Minutes
departingAirport : String
arrivingAirport : String
delayFlight (numberOfMinutes : Minutes)
getArrivalTime ( ) : Date
0..*
assignedFlights
assignedPlane
0..1
Plane
airPlaneType : String
maximumSpeed : MPH
maximumDistance : Miles
tailId : String

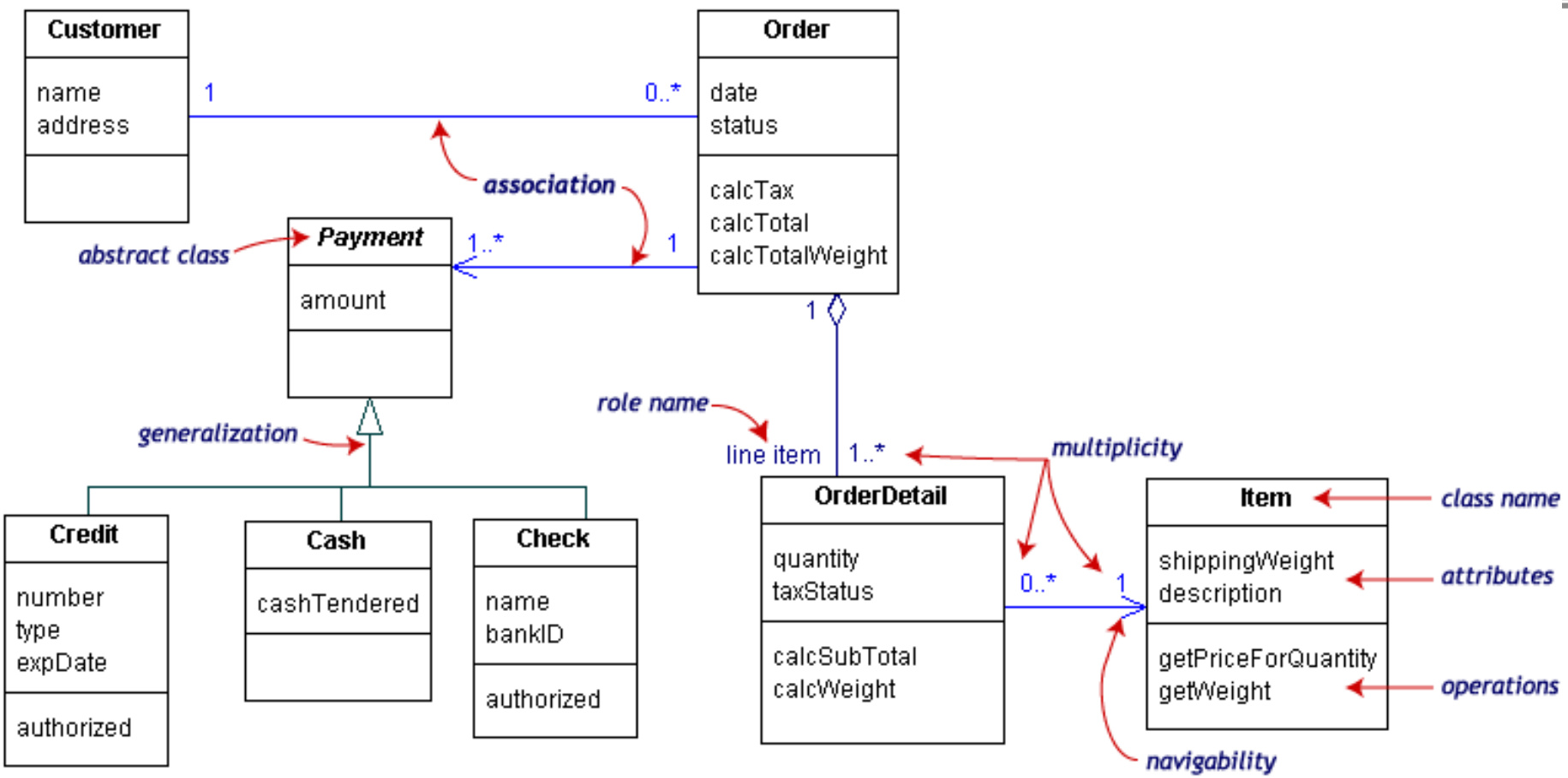
Ternary Association



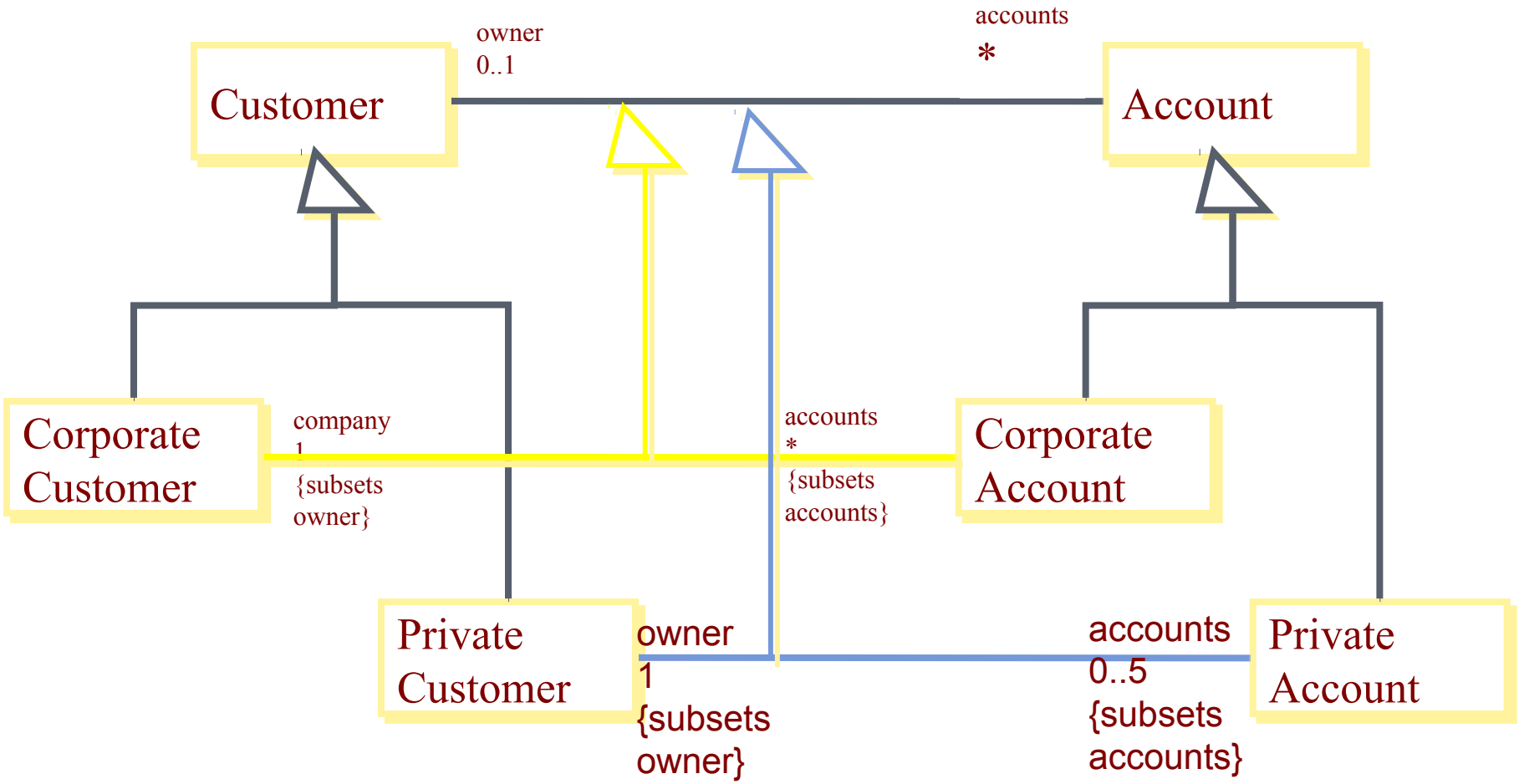
Associations with restrictions



Class Diagrams: example

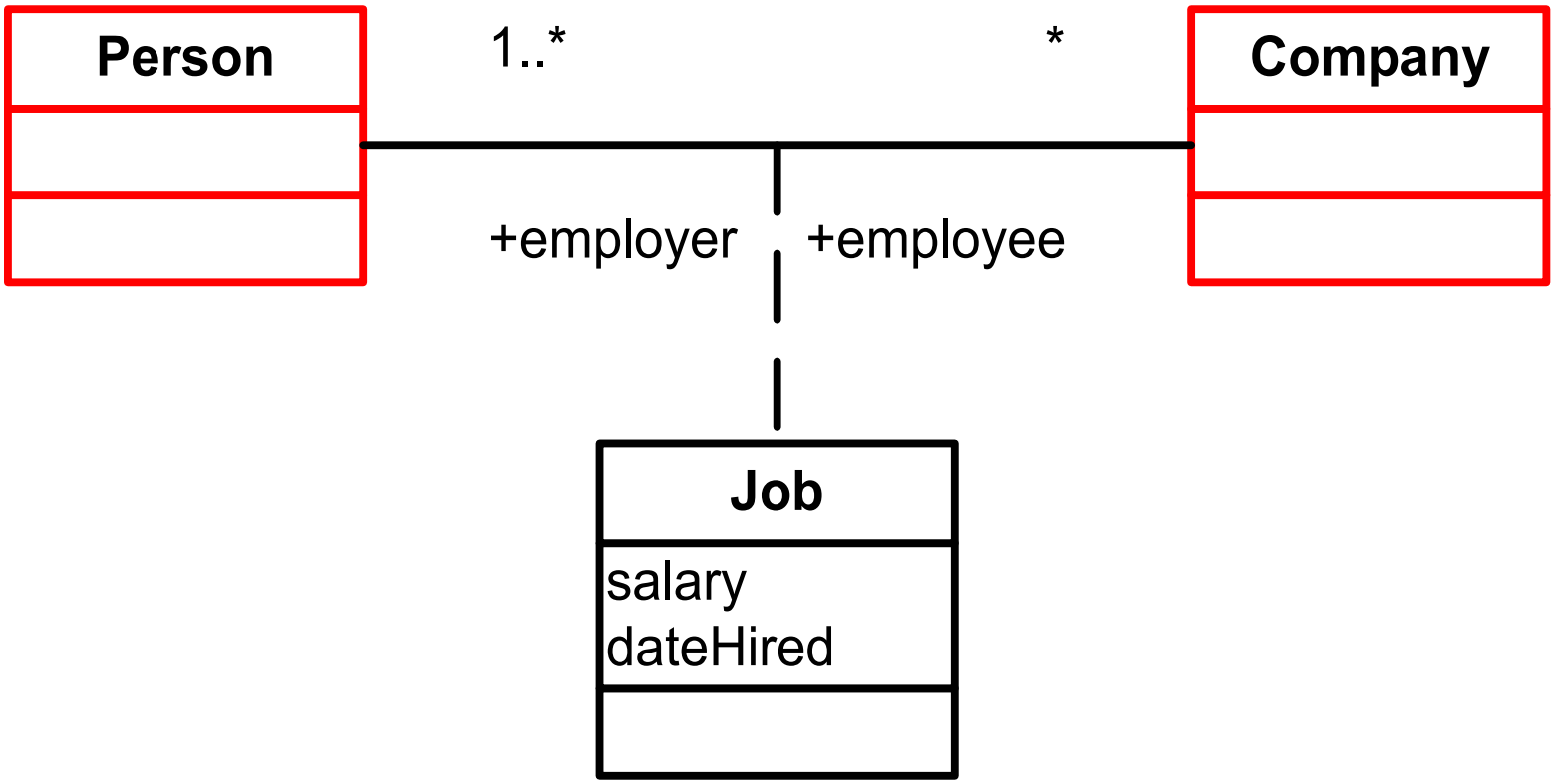


Specialization of associations

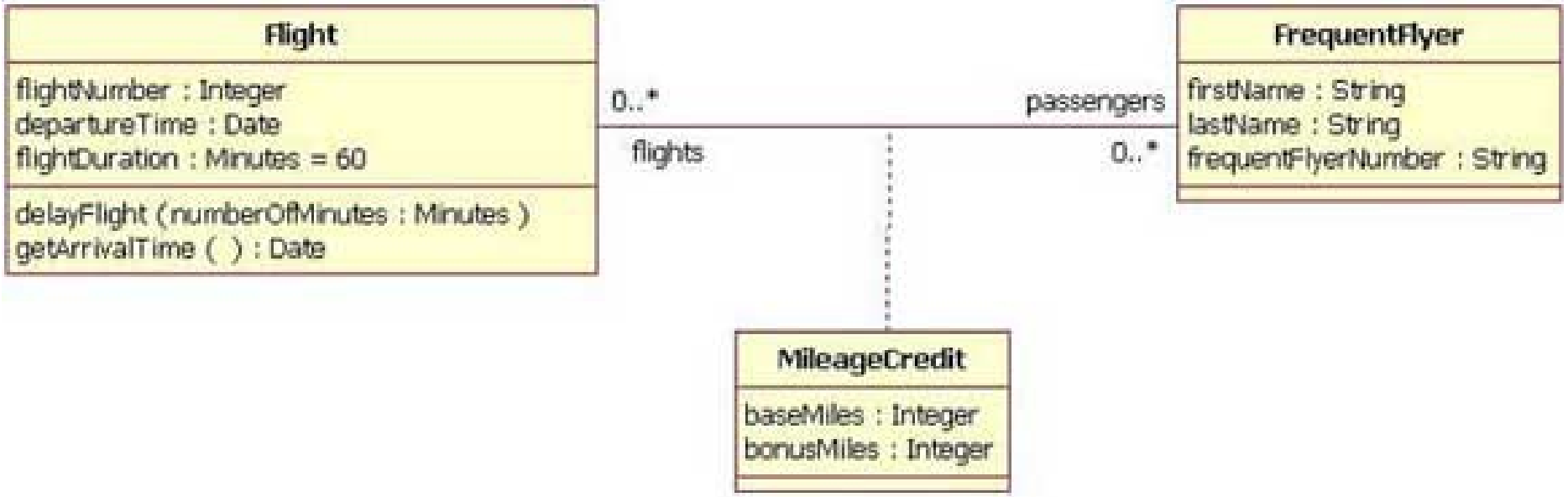


Association class

- Happens when the association has its own properties

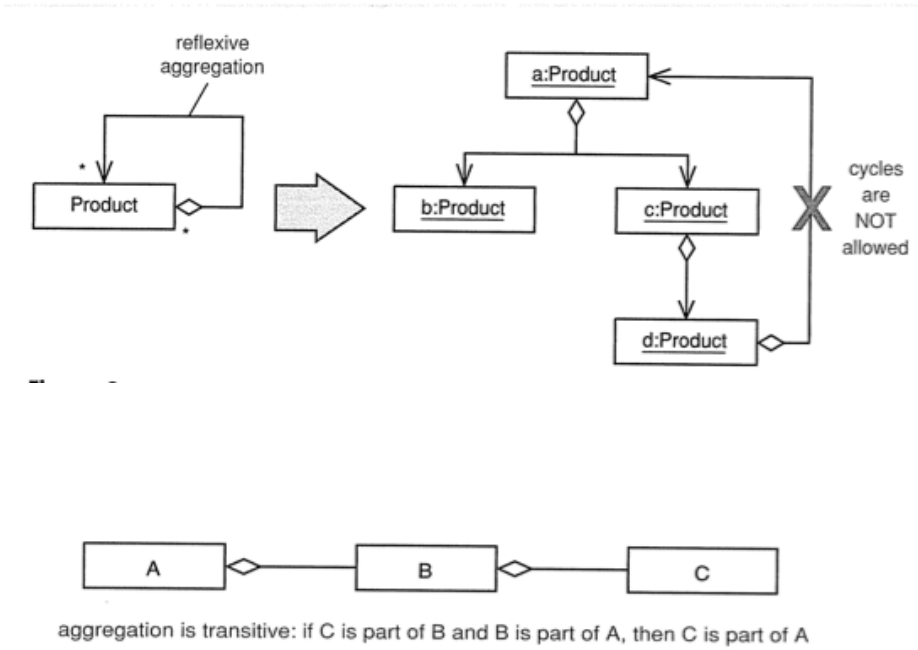


Association Class(2)

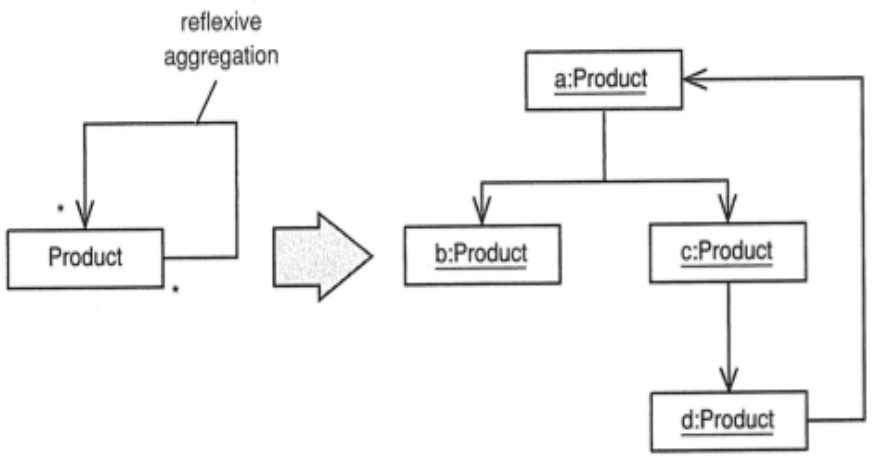


Aggregation

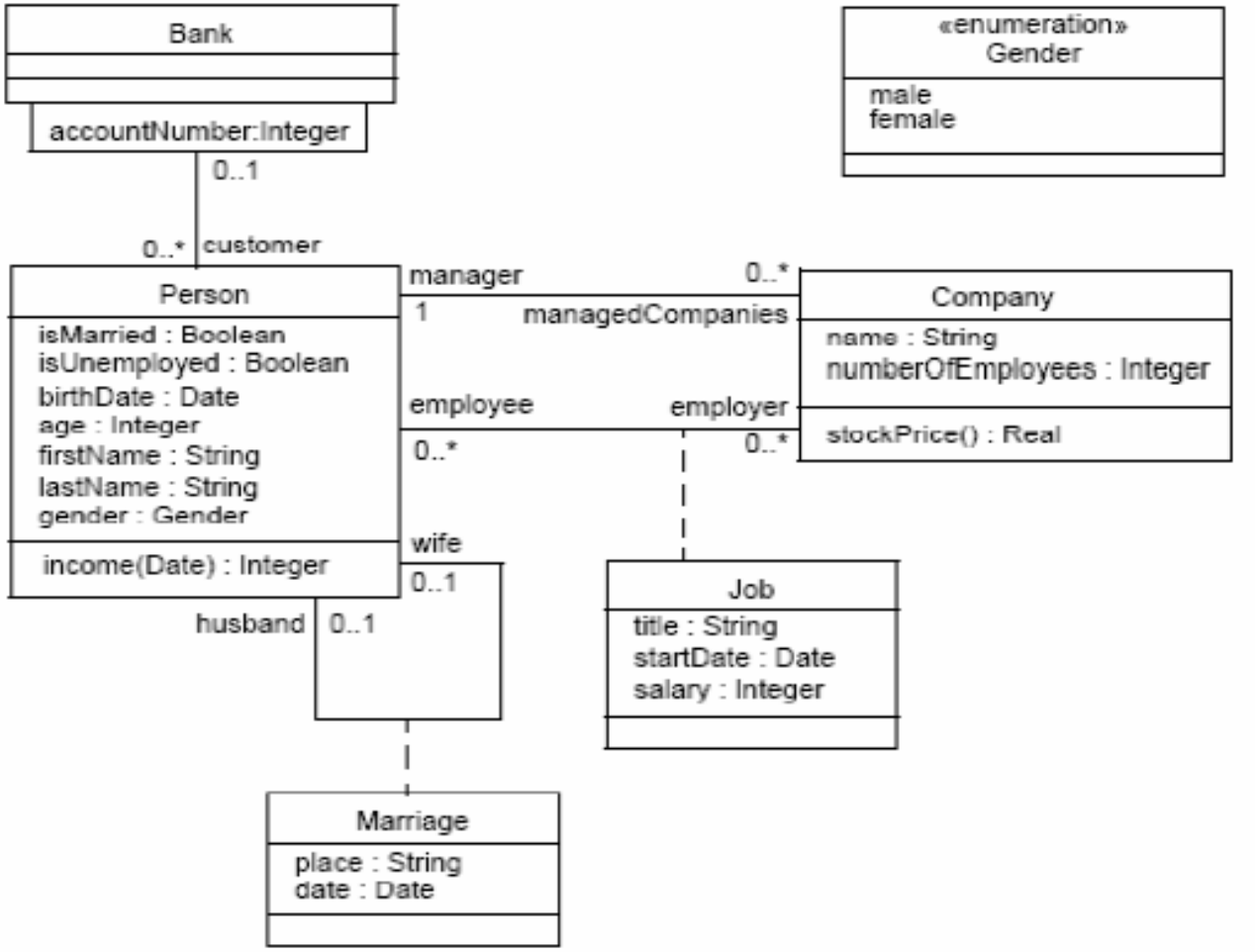
- Shows how the classes are composed by other classes
- “is part-of” Association
- Transitive
- Asssymetric (in reflexive aggregations)



In contrast with association

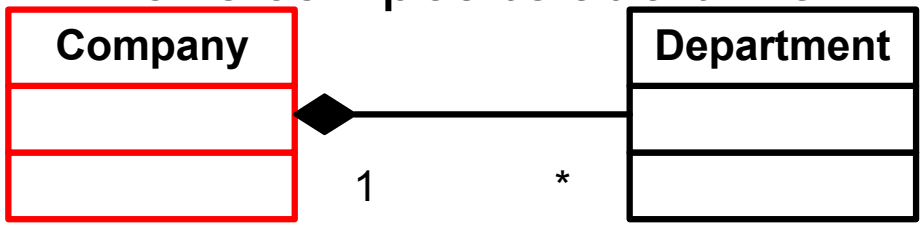


Association Class



Composition

- A strong type of aggregation:
  - Strong presence of the part with respect to the whole
  - The parts can not exist without the whole
  - Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time



Composition

Parts can only belong to a composite at a given time
The composite is responsible for the creation and deletion
The composite can release parts as long as the responsability is assumed another object

Aggregation and Composition