

# *Object-Oriented Analysis and Design*

Session 3: Structure Modeling

# Outline

---

1.	Objects and Classes.....	4
2.	Basic Association Concepts.....	17
3.	Class Diagrams and Object (instance) Diagrams....	32
4.	Multiplicity constraints on associations.....	51
5.	Associations as objects.....	65
6.	Class hierarchies.....	77
7.	Constraints on associations.....	103
8.	Aggregations .....	110
9.	Relationship overview.....	118
10.	Attribute characterization.....	125

# *What is structural modeling?*

---

A structural model is a momentary snapshot (view) of a system showing the structure of the objects, their classifiers, relationships, attributes and operation signatures.

- It shows:
  - The identified abstract entities:
    - **Classes, interfaces.**
  - The abstract entities internal structure:
    - **Attributes.**
  - Relationships between the abstract entities:
    - **Associations, dependencies, inheritance.**
- It does not show:
  - temporal information
    - **Method definitions, processes, threads.**

# *1. Objects and Classes*

# Object - 1

---

- **Object** – A thing that is meaningful for an application:
  - Proper nouns: Ben-Gurion-University, Moshe.
  - Physical objects: A table, a contract.
  - Abstract objects: An agreement, a sale, an algorithm, a formula, an access permission.
  - A composite object: A list, a table.
- An object has an *identity* that distinguishes it from all other objects.
- An object belongs to a Class, based on:
  - Similarity of structure (*attributes*).
  - Similarity of behavior (*methods*).

# Object - 2

---

- Visual notation:

Possibly named instance (object) figures.

<b>&lt;object name&gt;: &lt;class name&gt;</b>
-attribute 1 = Value 1
-attribute 2 = Value 2

- *Examples:*

<b><u>Bank-Hapoalim:Bank</u></b>
-number of accounts = 4000
-address = Hashalom street

<b><u>Moshe-Account:Bank-Account</u></b>
-balance = 1000

# Object - 3

---

*Examples:*

<u>aBinaryTree:BinaryTree</u>

<u>Huston:City</u>
-cityName = Huston
-population = 30000

<u>LHR:Airport</u>
-airportCode = LHR
-airportName = Heathrow

<u>TLV:Airport</u>
-airportCode = TLV
-airportName = Ben-Gurion

- Object names are optional!
- Constraint : Object name **can not** be repeated for a class (in a specific context)

# Class - 1

---

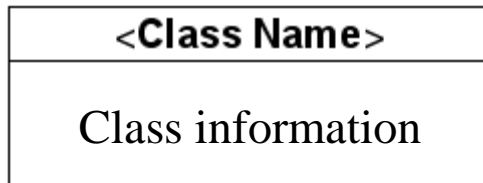
- **Class** – A group of objects (*instances*) that have common:
  - Properties.
  - Behavior.
  - Relationships to other objects .
  - Semantics.
- Finding classes in text: Common nouns, noun phrases.
- **Constraint:** In the context of a single diagram, a class has a **unique name** that identifies it!



# Class - 2

---

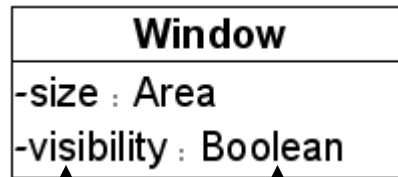
- Visual notation:



- Example:

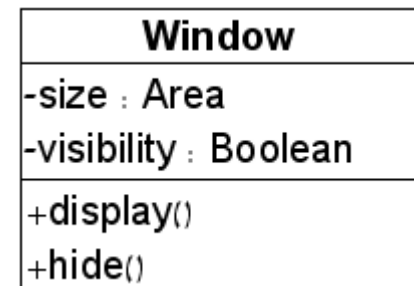


Class Name



Attributes

Data type



Operations

# Class - 3

- More class information visualized

<b>Window</b> {abstract, author=Joe, status=tested}
+size:Area = (100,100) #visibility:Boolean = invisible +default-size:Rectangle #maximum-size:Rectangle -xptr: XWindow*
+display () +hide () +create () -attachXWindow(xwin:Xwindow*)

<b>Reservation</b>
<b>operations</b> guarantee() cancel () change (newDate: Date)
<b>responsibilities</b> bill no-shows match to available rooms
<b>exceptions</b> invalid credit card

# *Class - 4*

---

- Class and object visualization is by default **incomplete**:
  - Having no attributes does not mean that there are no attributes.
  - Having no attribute types does not mean that there are no attribute types.
  - Having no operations.....
- The **only mandatory** part is the class name: unique.
- **Responsibility**: A specification of the class intention.
  - Always recommended (recall the high cohesion principle!)
  - Understood as a **contract** or an obligation of a class.
  - As the model is refined the responsibilities are transformed into attributes and operations.

# Attributes

---

- **Attribute:** A named property within in a class

An attribute is a mapping:

*Class*  $\rightarrow$  *Value-domain (Data type)*

- **Visual notation:** Names in the lower part of a class rectangle.
- **Finding attributes in text:** adjectives, value abstractions.
- **Constraint:** Unique name per class!

# *Data types and Values*

---

- Value: A piece of data. No identity.
- Data type: like an *Abstract data type (interface)*:  
A set of values + operations.  
Values might have their own attributes.
- **Finding *values in text*:**
  - Enumerations
  - Examples in problem documentation.
- ***Visualization*:**

<<data type>> <b>Date</b>	
Number-in-week : [1..7]	
Month()	

<<data type>> [1..7]	
size : integer	
first() : integer	

# *Objects vs. Values*

---

- **Objects** have identities, can have attributes, operations, and participate in relationships.
  - Two forms of equality:
    - **Identity equality.**
    - **Content equality.**
- **Values** – do not have identities
  - A single form of equality: **Value equality**
  - Value “duplications” are indistinguishable.
- **Values** - can only be **used** to form other values, or as the values of attributes and parameters for operations.

# *Objects vs. Values*

---

- An object attribute value is a *VALUE* not an object
  - An element of a data type.
- **To be an object or to be a value**, that is the question
  - Depends on the application (one application values may become an object in another application):
    - In a library system *Date* is probably a data type. Its elements are values.
    - In a calendars translation system *Date* is probably a class. Its elements are objects.
- Object-Value heuristic rule:  
**If it's important it's an object!**

# Terminology

---

A *Class* is a set of *objects* (instances)

A *Data type* is a set of *data values*

An *Attribute* is a mapping from a class into a data type.

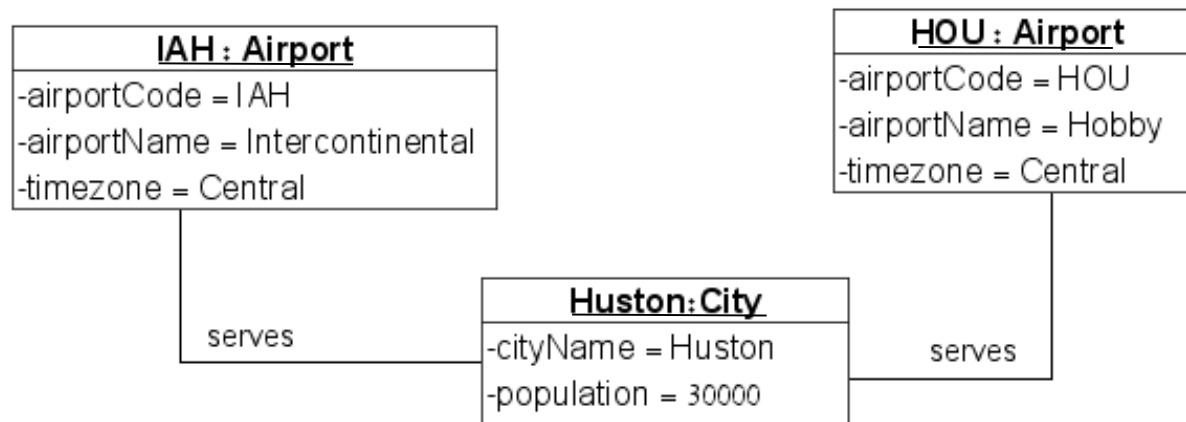


## *2. Basic Association Concepts*

# *Link and Association concepts - 1*

---

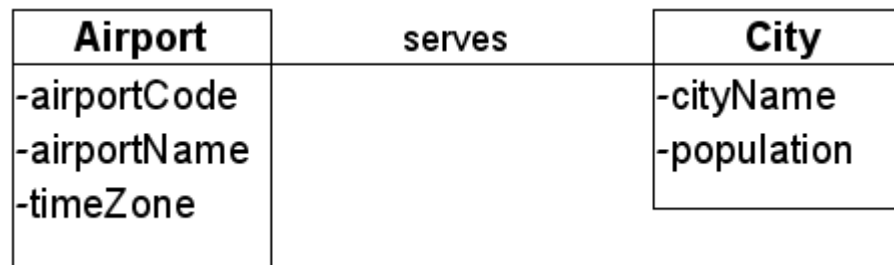
- **Link** – conceptual or physical connection between objects
- *A link is (denotes) a tuple*
- A binary link is a pair. A ternary link is a triplet, etc.
- ***Visual notation:*** Possibly labeled line among objects. The label is the name of the association to which the link belongs.



# *Link and Association concepts - 2*

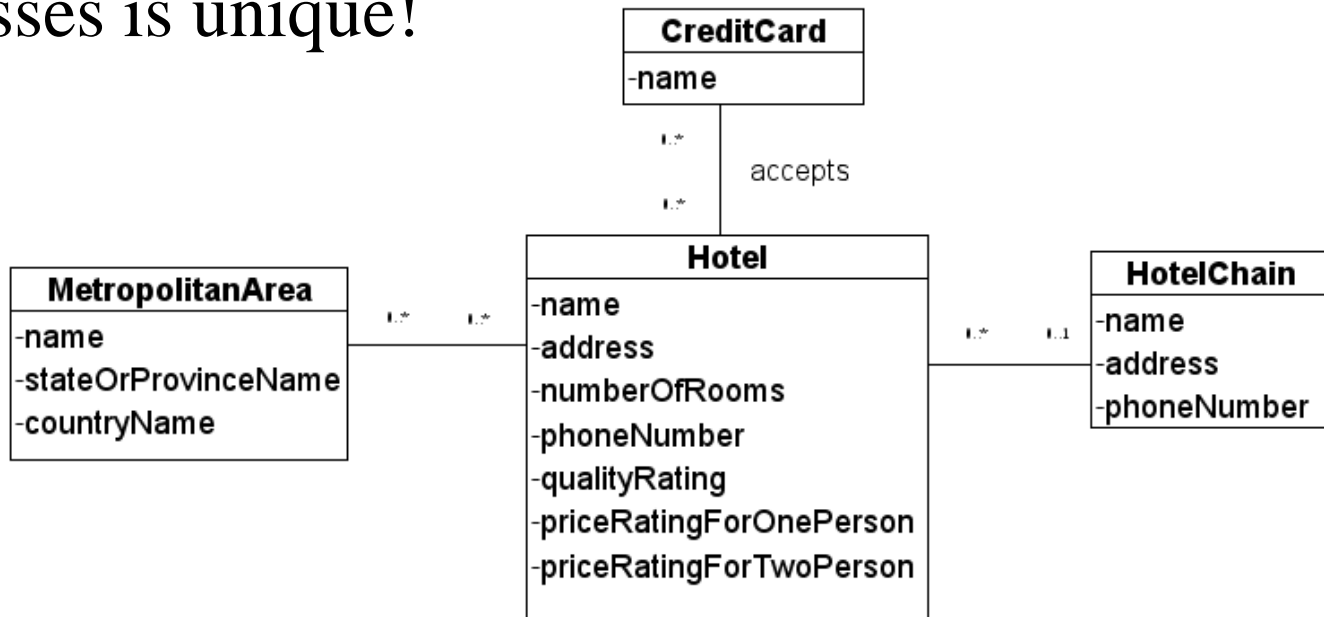
---

- **Association** – A group of links with common structure and common semantics :
  - Links among objects from the same classes and with the same structure (*similar properties*)
- ***Visual notation*** : Possibly labeled line among classes.



# *Link and Association concepts - 3*

- The line label is the **name** of the association
- It is optional (but mandatory for multiple associations among classes)
- Associations are not directed!
- *Finding Associations in text: verbs.*
- **Constraint:** The name of an association between a set of classes is unique!



# *Classes and Associations*

---

**A link is an instance of an association.**

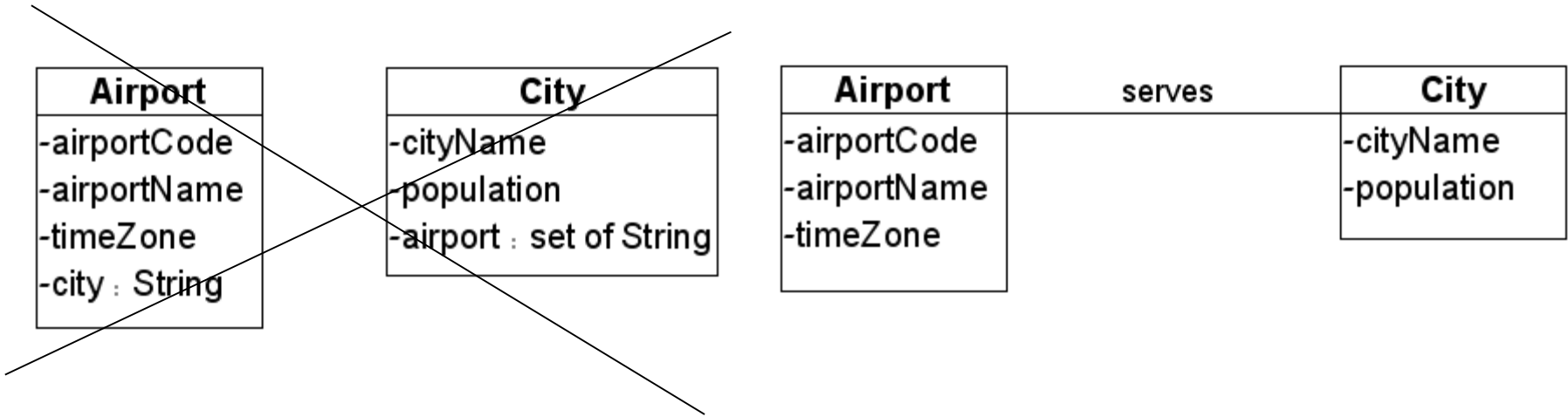
**A link is not a value.**

**A link is not an object:** No identity.

<b>Analogy:</b>	Association	—	link
	Class	—	object

# Associations vs. Attributes - 1

- Novice modelers (especially with DB experience) tend to overuse attributes instead of associations.



- Associations are the essence of object-oriented modeling. Connections between objects are always modeled as links.
- Attributes provide additional (secondary) information about identified objects.
- Attribute heuristic rule:

**If it's important it's not an attribute!**

# Associations vs. Attributes - 1

## Example:

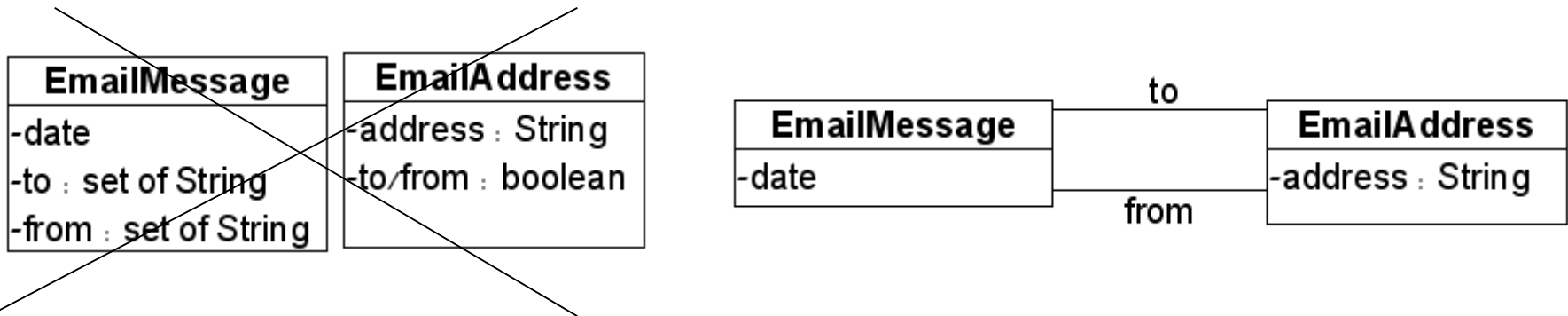
*Consider: An Email Message has To Addresses and From Addresses.*

*Question: Is the To or From an **attribute** of EmailAddress? Or of EmailMessage?*

*Answer: NO NO NO!*

*The To and From describe relationships between a message and its email addresses.*

*→ To and From belong to two different **associations** between EmailMessage and EmailAddress:*



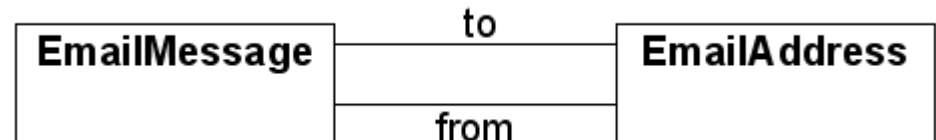
# White Class Diagrams

---

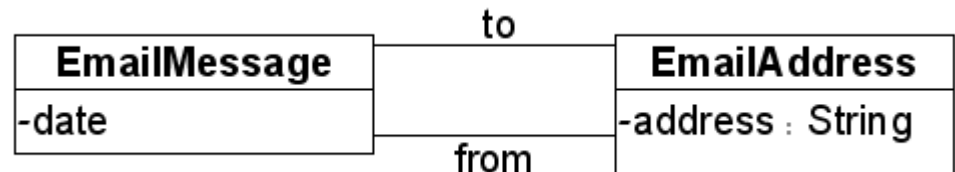
- **White class diagrams** are diagrams with no attribute information. They are used in the first stage of a system modeling in order to enforce concentration on classes and associations.

## *Example:*

- *A white diagram:*



- *A non-white diagram:*





# *Link and Association concepts: Roles - 1*

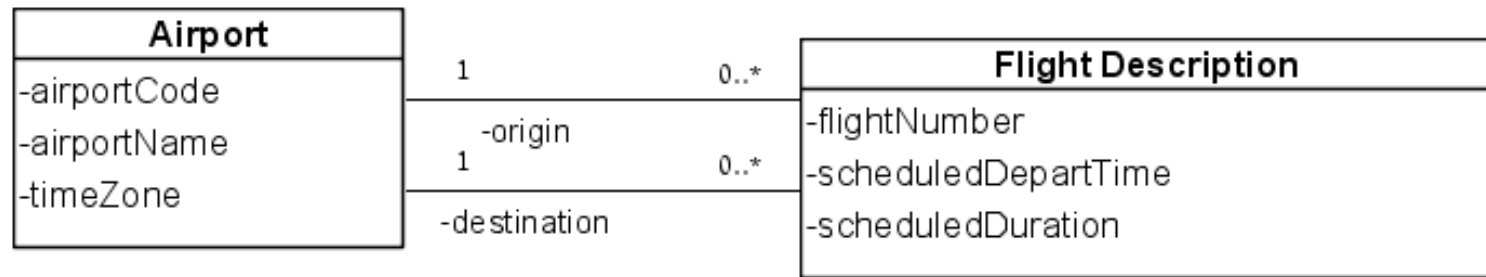
---

- A **role** is a *relation between*
  - a class in an association
  - the association
- A role denotes the “role” of the class in the association.
- It provides a (conceptual) direction on the association.
- A (binary) association has exactly two roles.
- A role has a *name*.

# Link and Association concepts: Roles - 2

---

**Visual notation:** The *role name* is marked as a label on the class edge in the association line.



Role names are used in the **Object Constraint Language (OCL)** for traversing associations:

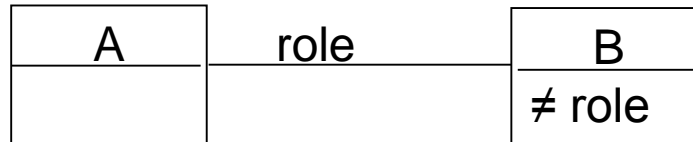
*aFlightDescription.origin* – accesses the origin airport.

*aFlightDescription.destination* – accesses the destination airport.

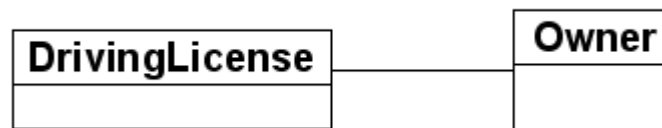
# *Link and Association concepts: Roles - 3*

---

- Role names function as pseudo-attributes –
  - Should be distinguished from the attributes of the origin class:



- If the name of a class adequately describes its role, you may **omit role names**.



DriverLicense.Owner

- The class name is used as the role name, if needed.

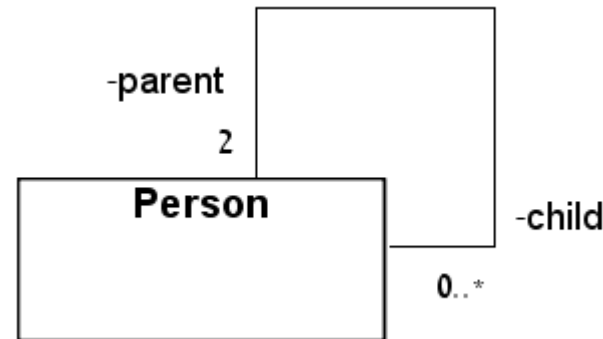
# Link and Association concepts: Roles - 4

---

**Constraint:** *Role names* are mandatory

in ambiguous situations:

1. Multiple nameless associations between classes.
2. Multiple roles for a single class in an association.



aPerson.parent – accesses the parents of a person.

aPerson.child – accesses the children of a person.

The parent-child association is cyclic (recursive).

Describes hierarchical instance structures (directed graphs).

# Roles vs. Objects - 1

**Roles** should not be confused with *objects*.

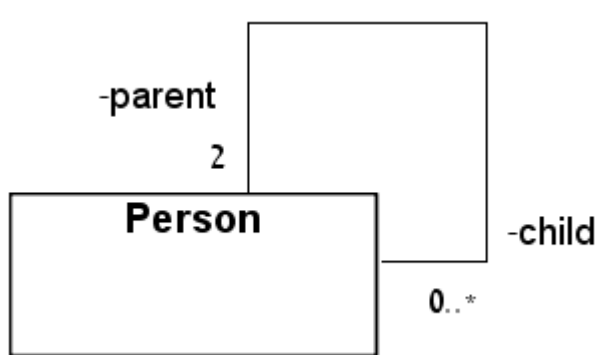
## Example:

**Consider:** A person has exactly two parents and any number of children.

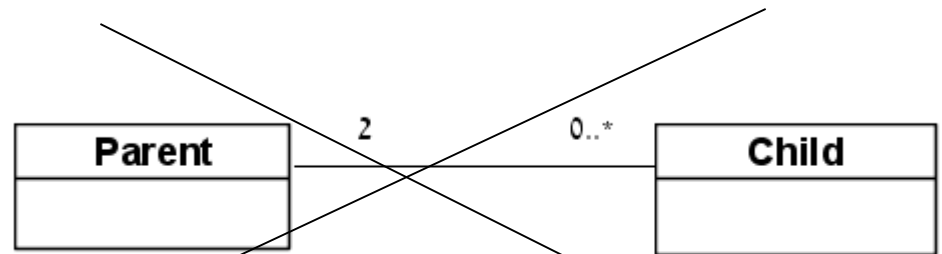
**Question:** Are Person, Parent, Child **roles** or *objects*?

**Answer:** Person is a class;

parent, child are roles in a parent-child recursive association on Person.



Correct Model



Wrong Model

# *Roles vs. Objects - 2*

---

- **To be a role or to be an object, that is the question**
  - depends on the application (one application roles may become another application objects).
- **Object-Role heuristic rule:**
  - It's an object only if it is an entity characterization!
  - Think if you wish to characterize it with attributes and associations.
  - Otherwise, it's a role in an association!
- **In most applications:**
  - Person is a characterization of people.
  - Parent, child are not characterizations of entities :
    - a child can be a parent.
    - a parent can be a child.
- **But in a school application Parent and Child might be appropriate classes.**

# *Roles vs. Objects - 3*

---

## *Examples:*

*Consider: A Flight Description has an Origin Airport and Destination Airport.*

*Question: Are Flight Description, Airport, Origin Airport, Destination Airport roles or objects?*

*Answer: In standard airports:*

*Flightdescription, Airport are classes;  
destination, origin are roles in flightDescription-  
airport associations between flightDescription and Airport.*

*Consider: A Sale Transaction has a Seller and a Buyer.*

*Question: Are Sale Transaction, Seller, Buyer roles or objects?*

*Answer: ???*

### *3. Class Diagrams and Object (instance) Diagrams*



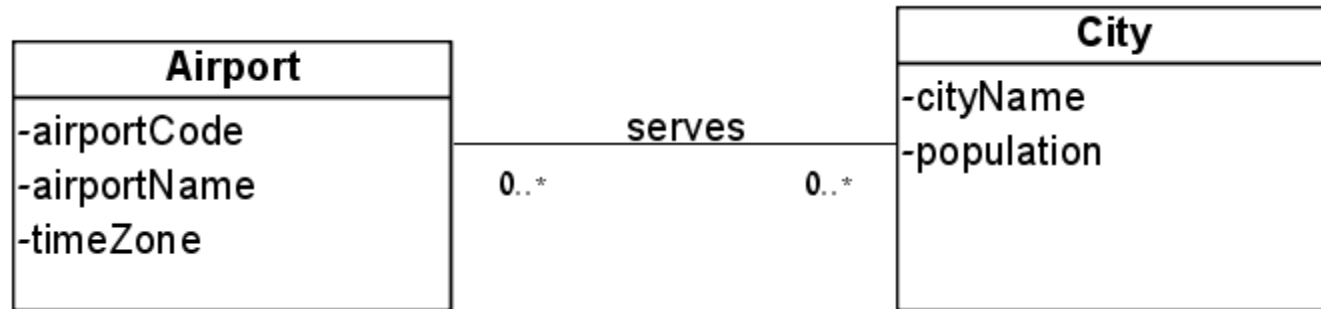
# *Class diagrams: Syntax - 1*

---

A **class diagram** is a visual specification of UML *class level* concepts.

It includes: *Classes, associations, attributes, roles, multiplicity constraints, and more.*

Alternative names: *Object model, class model.*



# *Class diagrams: Syntax - 2*

---

- Syntactically correct class diagram :
  - **An association** *line connects only class figures, not associations.*
  - **No class name repetitions**
  - **No association name repetitions among same classes.**
  - **Mandatory role-name constraints.**
  - **No attribute name repetitions within a class.**
  - **And more...**

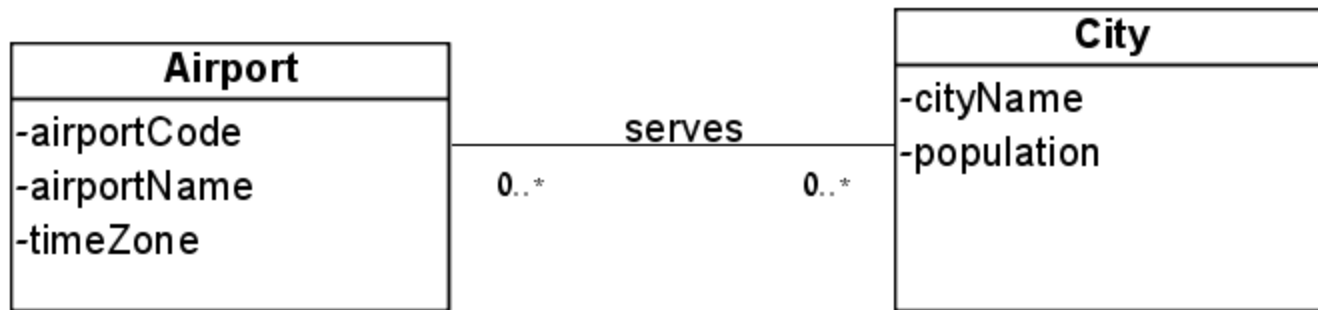
# Class diagrams: Syntax - 3

- A *class diagram* describes (constrains) *possible realities* (worlds).
- Such a world consists of:
  - *classes* which are sets of *objects*.
  - *associations* which are (are) sets of *links* (object tuples).
  - *attributes* which are mappings.
  - *data types* which are sets of *data values*.
  - *roles* which are relations between *associations* and *classes*.
  - and more...

*class level*

*data level*

# Class diagrams: Syntax - 4



Possible worlds:

- W1: **Classes:** *Airport* = {AOU, JAH}; *City* = {Houston};  
**Associations:** *Serves* = {(AOU, Houston), (JAH, Houston)};  
**Attributes:** *city name*(Houston) = "Houston";...
- W2: **Classes:** *Airport* = { }; *City* = {c1, c2};  
**Associations:** *Serves* = { };  
**Attributes:** *city name*(c1) = "Tel-Aviv";...
- W3: **Classes:** *Airport* = *City* = { };  
**Associations:** *Serves* = { };
- W4: .....

**Question:** How many worlds are there?

# *Class diagrams: Syntax - 5*

---

- A possible world of a class diagram consists of:
  - An instantiation :
    - **For all classes, associations, and attributes in the diagram**
    - **Only for classes, associations, and attributes in the diagram**
  - **It should satisfy all constraints imposed in the diagram.**
- *Every symbol/figure has an instantiation!*
- *Only symbols/figures in the diagram have instantiations!*
- *All constraints hold!*

# *Class diagrams: Semantics*

---

The **semantics** (*meaning, denotation*) of a **class diagram** is the set of worlds that it describes.

A world that is described by a **class diagram** can be

empty – all classes are empty;

non-empty – some classes are non-empty;

finite – all classes are finite;

infinite – some classes are infinite.

**Intended worlds: *Finite non-empty*.**

- The UML specification: <http://www.uml.org/> does not provide formal semantics.
- Enormous research efforts (academic and industry).

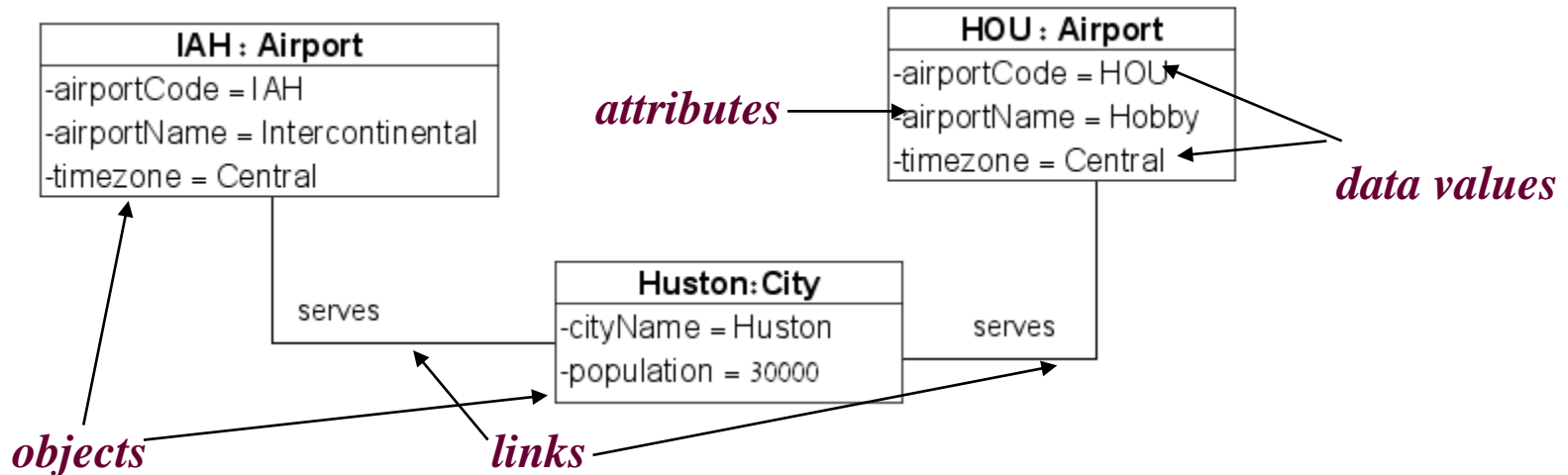
# *Instance (Object) diagrams*

---

- **Symbolic description of possible worlds is annoying and not readable.**
  - Visualization can help in this case
  - Possible worlds are visualized by instance diagrams.
- **An instance diagram describes part of a world denoted by a class diagram:**
  - The all requirement in a possible world is waived
  - The only and all constraints requirements must be satisfied

# Instance (Object) diagrams: Syntax - 1

- An **instance diagram** is a visual specification of UML instance (data) level concepts.
- It includes : *Instances, links, attributes values, roles.*



- *An instance diagram visualizes a non-empty and finite world.*
- *Empty means that it includes no objects → no diagram!*

Infinite *means that it includes an infinite number of objects:*

*Diagrams cannot be visualized!*



# *Instance (Object) diagrams: Syntax - 2*

---

- **Syntactically correct instance diagram:**
  - *An* object (instance) cannot be repeated – each instance figure stands for a different object.  
Instance names cannot be repeated within a class.
  - A link line connects instances.
  - A link cannot be repeated. Different links between same objects must belong to different associations.
  - Attribute values *are* data values *not* objects.

# *Instance (Object) diagrams: Usage*

---

- **Instance diagrams** are always used relatively to a class diagram.
- They are used for:
  - **Analyzing** the intended application by demonstrating the worlds that can be described by the class diagram.
  - **Validating** a class diagram by testing whether the worlds described by the class diagram meet the intentions of the designer.

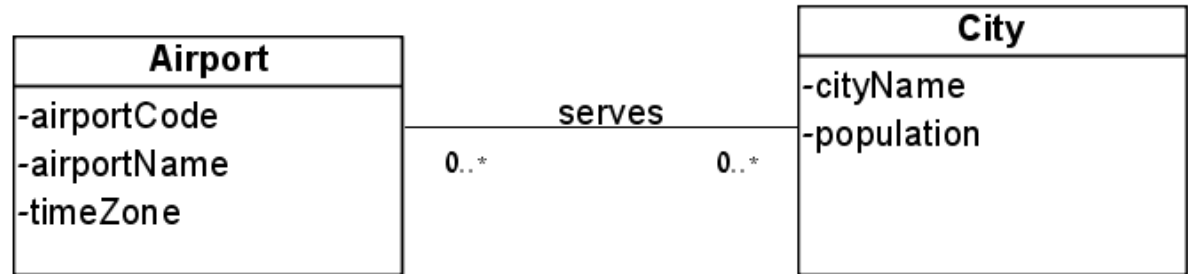
# *Instance (Object) diagrams: Terminology*

---

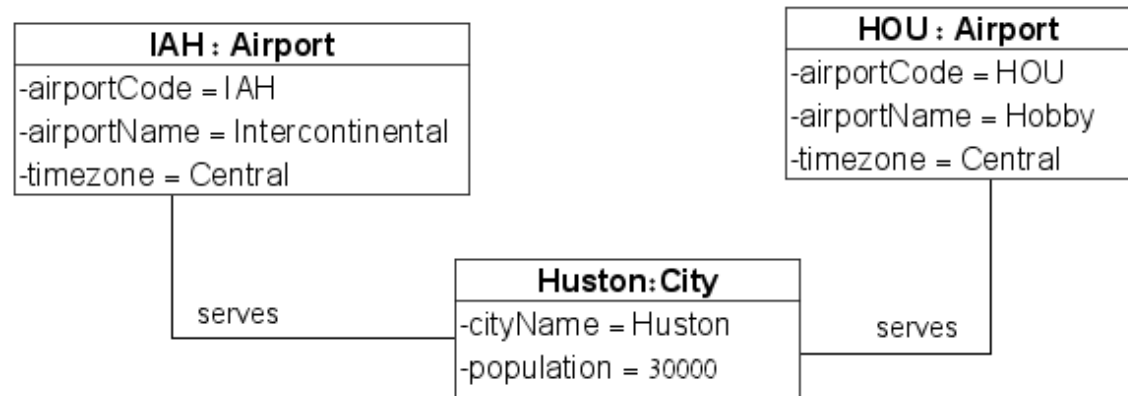
- An instance diagram *instantiates/populates* a class diagram
- An instance diagram is an *instance* of a class diagram if it instantiates **only** elements from the class diagram: Classes, associations, attributes
- An instance diagram is a *legal instance* of a class diagram if it is an instance and it satisfies **all constraints** imposed by the class diagram

# Instance (Object) diagrams: Examples - 1

*For the class diagram:*



*The instance diagram:  
is a legal instance*

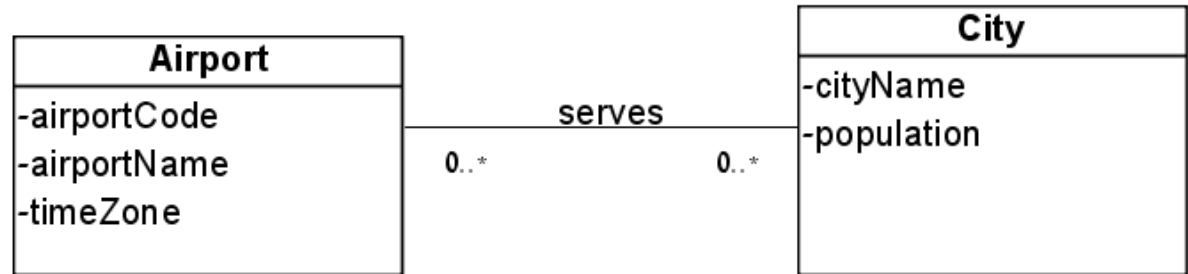


*It visualizes the world:*

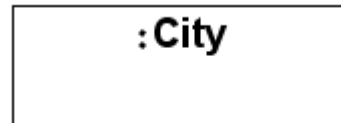
$W_I$ : **Classes:** **Airport** = {AOU, JAH}; **City** = {Houston};  
**Associations:** **Serves** = {(AOU, Houston), (JAH, Houston)};  
**Attributes:** **cityName**(Houston) = "Houston";...

# Instance (Object) diagrams: Examples - 2

*For the class diagram:*



*The instance diagram:*



*is a legal instance*

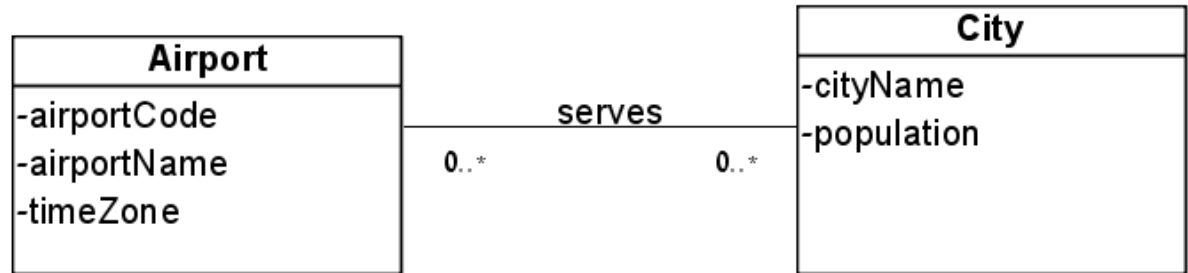
*It visualizes a portion of the  $W_1$  world:*

- *No Airport objects are visualized.*
- *No attributes of the City object are visualized.*

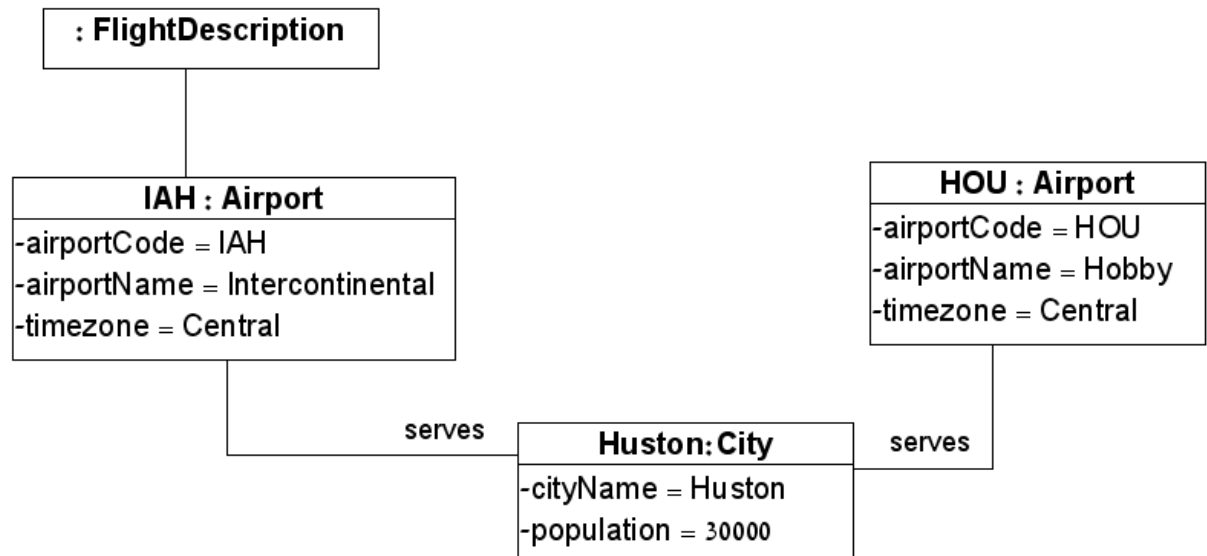
*Note: The object name is not part of the described world – objects are nameless. They exist by virtue of their identities. In the instance diagram, every object figure stands for an object identity. Names are added just for convenience of user references.*

# Instance (Object) diagrams: Examples - 3

*For the class diagram:*

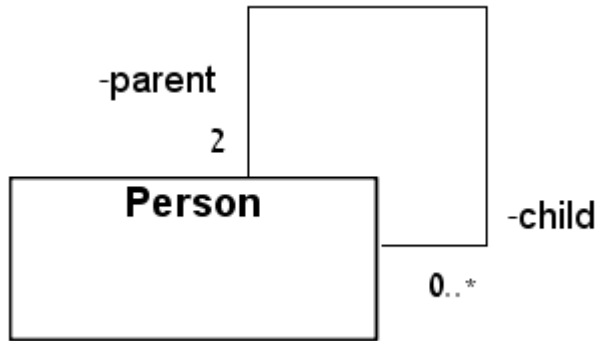


*The instance diagram:  
is **not** a valid instance  
diagram*

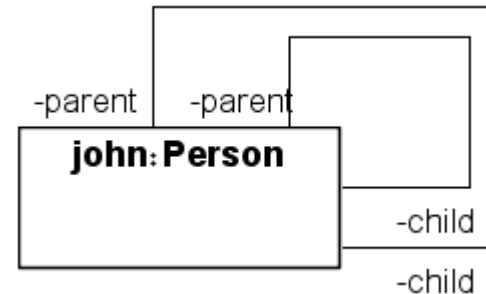


# Instance (Object) diagrams: Examples - 4

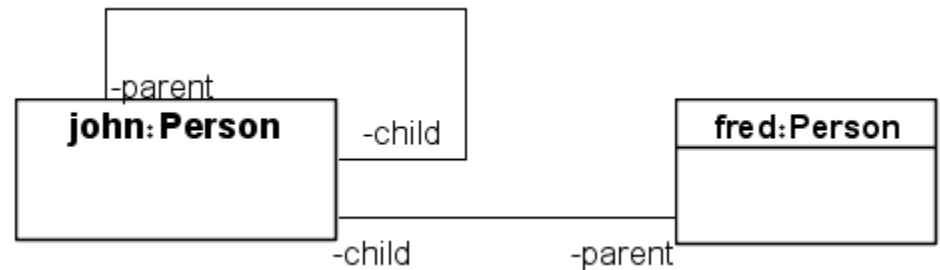
For the class diagram:



A syntactically incorrect instance:



A legal instance:



*It visualizes the world: Classes: **Person** = {john, fred};*

*Associations: **parent-child** = {(john, john), (john, fred)};*

*This is a legal not-intended instance!*

→ *The class diagram is too weak to enforce the designer intentions!*

# *Instance (Object) diagrams: Consistency*

---

- A class diagram is **satisfiable (consistent)** if it has a non-empty legal instance.
- A class diagram is **strongly satisfiable** if it has a non-empty legal instance that populates all classes (no empty classes).
- Later on we will see examples of unsatisfiable class diagrams. Ideally, CASE tools should:
  - either reject such diagrams as semantically incorrect ones,
  - or correct (provide advice) such diagrams.
  - or reject and advice as to how to correct such diagrams.



# *Class & instance diagrams : the logic analogy*

---

UML static **diagrams** are “sentences” in a visual language.

**symbols:**

**syntax:**

**UML diagram**

visual figures

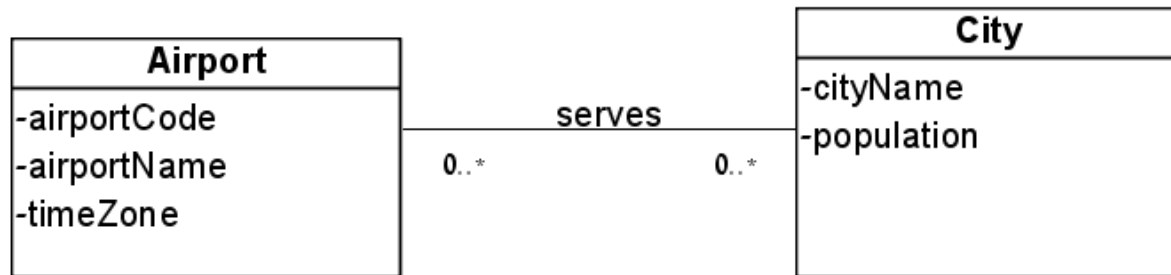
class diagram

**logic**

language symbols

universally quantified

equality/implication formulae

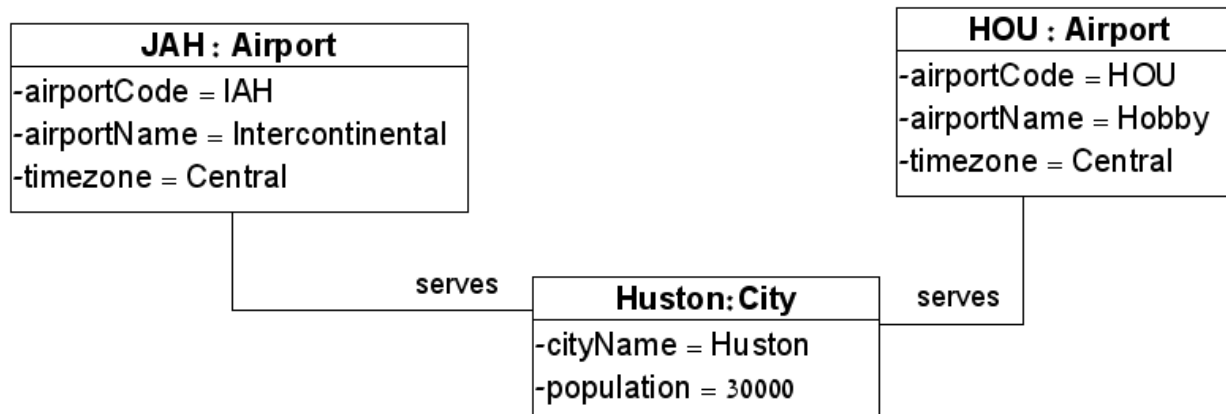

$$\forall x, y(\text{serves}(x, y) \rightarrow \text{Airport}(x) \wedge \text{City}(y))$$
$$\forall x \exists z(\text{City}(x) \rightarrow \text{cityName}(x, z))$$

...

# *Class & instance diagrams : the logic analogy*

---

	<u>UML diagram</u>	<u>logic</u>
<u>syntax:</u> ...	instance diagram	data assertions
<u>semantics:</u>	instantiation	structure (interpretation)
<u>consistency:</u>	possible world	model/world
	legal instance diagrams	



$$\text{Airport}(\text{HOU}) \wedge \text{Airport}(\text{JAH}) \wedge \text{City}(\text{Houston}) \wedge$$
$$\text{serves}(\text{HOU}, \text{Houston}) \wedge \text{serves}(\text{JAH}, \text{Houston}) \wedge$$
$$\text{cityName}(\text{Houston}, "Houston") \wedge \dots$$

## *4. Multiplicity constraints on associations*

# *Multiplicity constraints on associations - 1*

---

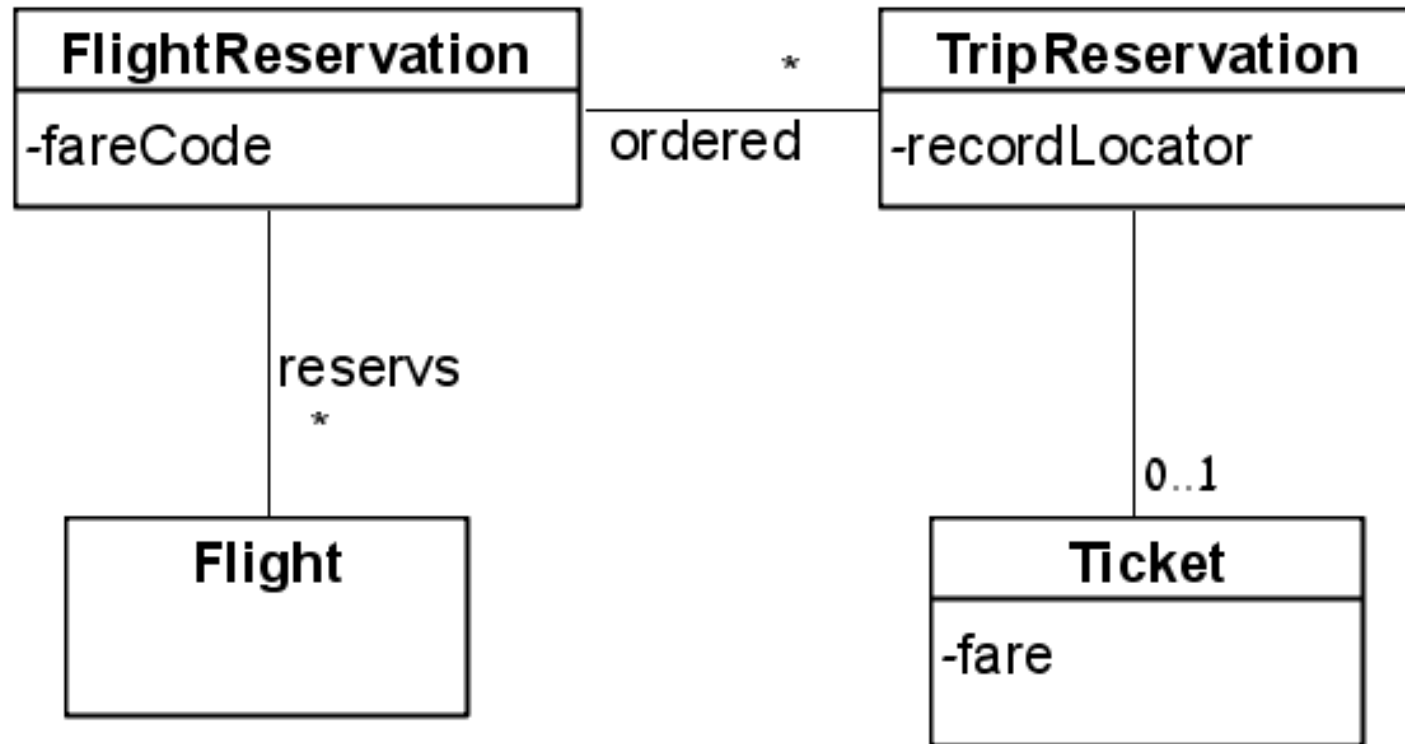
A multiplicity constraint specifies the number of instances that can be related through links in the association, to a single instance of an associated class.

## ***Notations:***

*	– 0 or more:	min=0, max= infinity.
1	– exactly 1:	Min=max=1
1..*	– 1 or more:	Min=1, max=infinity.
0..1	– 0 or 1:	Min=0, max=1.
2..4	– between 2 to 4:	Min=2, max=4.
5	– exactly 5:	Min=max=5.
2,4	– 2 or 4.	

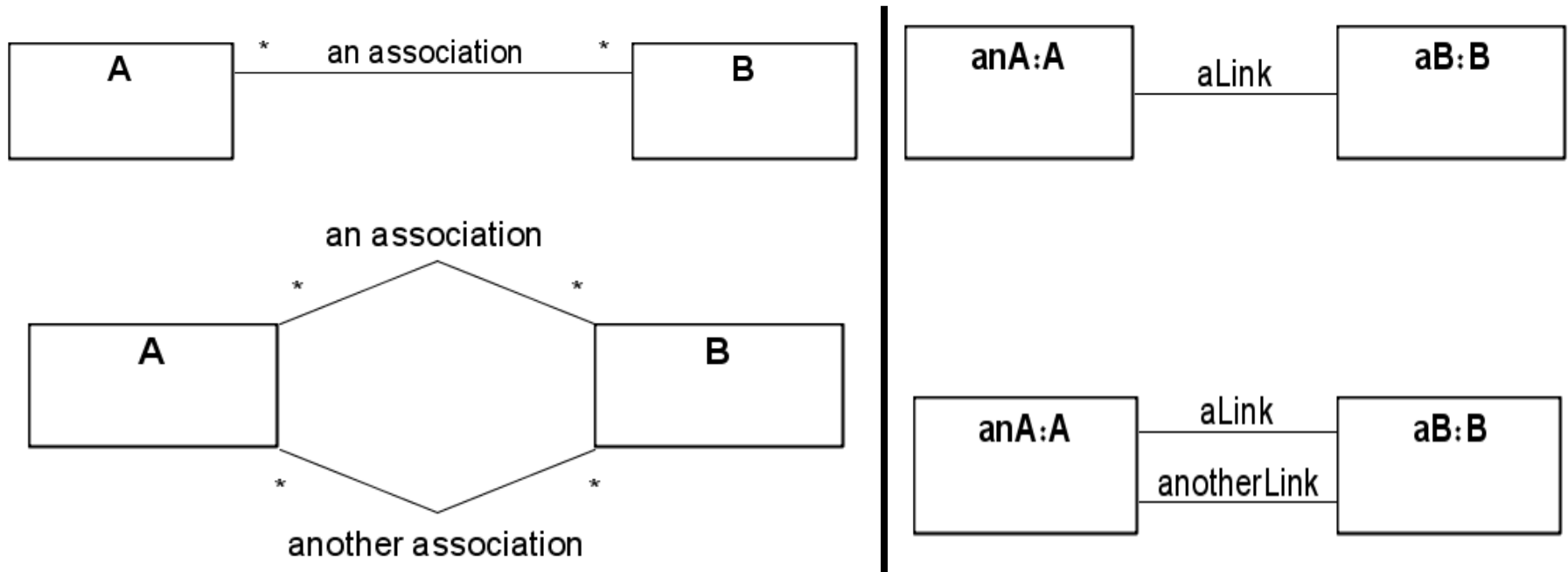
# *Multiplicity constraints on associations - 2*

---



# *Multiplicity constraints on associations - 3*

Multiplicity constraints refer to class diagrams.  
They are meaningless for instance diagrams:

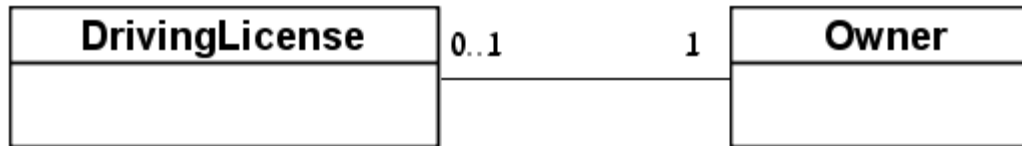


The 2 links **must** instantiate 2 different associations.

# *Multiplicity constraints on associations - 4*

---

**Existence dependency:** An exactly 1 multiplicity implies existential dependency between objects:

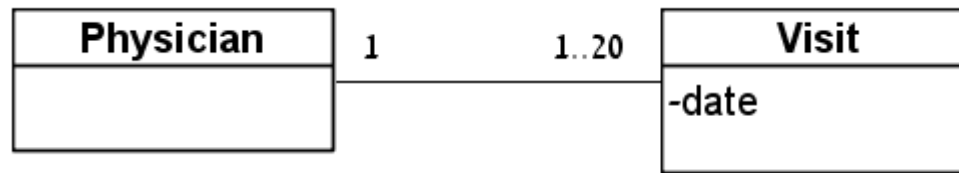


Deletion of an owner implies deletion of its driver license.

# *Multiplicity constraints on associations - 5*

---

- Multiplicity constraints *do not* involve attribute values:



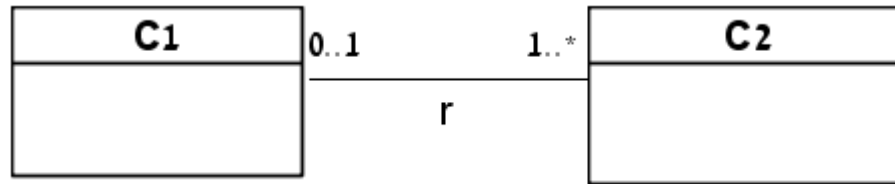
- What is the meaning of this diagram?
  - Draw a legal but unintended instance diagram.
  - Draw an illegal but intended instance diagram.



# *Multiplicity constraints on associations - 6*

---

Multiplicity constraints can be expressed in logic:



$$\forall x(C_1(x) \rightarrow \exists y(r(x, y))) \wedge$$

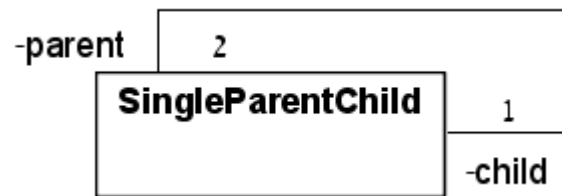
$$\forall x, y, z((C_2(y) \wedge r(x, y) \wedge r(z, y)) \rightarrow x = z)$$

- A minimum constraint requires *existential quantification*.
- A maximum constraint requires *universal quantification*.

# *Multiplicity constraints on associations - 7*

---

- An **unsatisfiable** class diagram – has either an empty or an infinite legal instance diagrams:

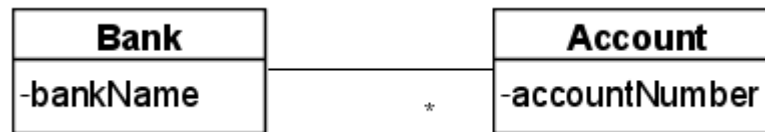


- A legal instance has the structure of a directed binary tree, where every node has:
  - exactly one incoming edge (the child).
  - exactly two outgoing edges (the parents).
  - **an infinite structure!**

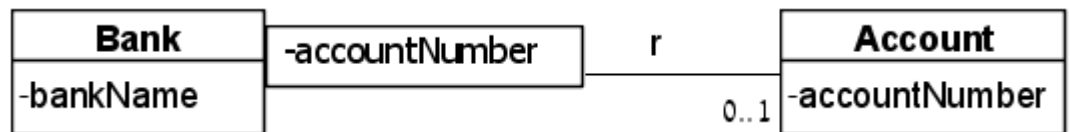
# Qualified Associations -1

- A **qualifier** is a means for disambiguating the objects in a “many” role:
  - A qualifier can be an attribute or a role
  - A qualifier *selects* among the target objects.

Not qualified:



Qualified:


$$\forall b, a1, a2, n \ ( (Bank(b) \ \& \ Account(a1) \ \& \ accountNumber(a1, n) \ \& \ r(b, a1) \ \& \ Account(a2) \ \& \ accountNumber(a2, n) \ \& \ r(b, a2) \ ) \rightarrow a1 = a2)$$

# *Qualified Associations -2*

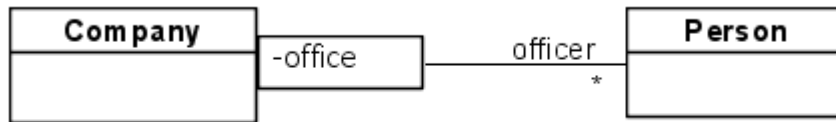
---

- A qualified association is more informative:
  - Reduces multiplicity.
  - Implies direction for traversing the association.
  - Usually affects only the maximum bound.

# *Qualified Associations - 3*

---

- Partial disambiguation:



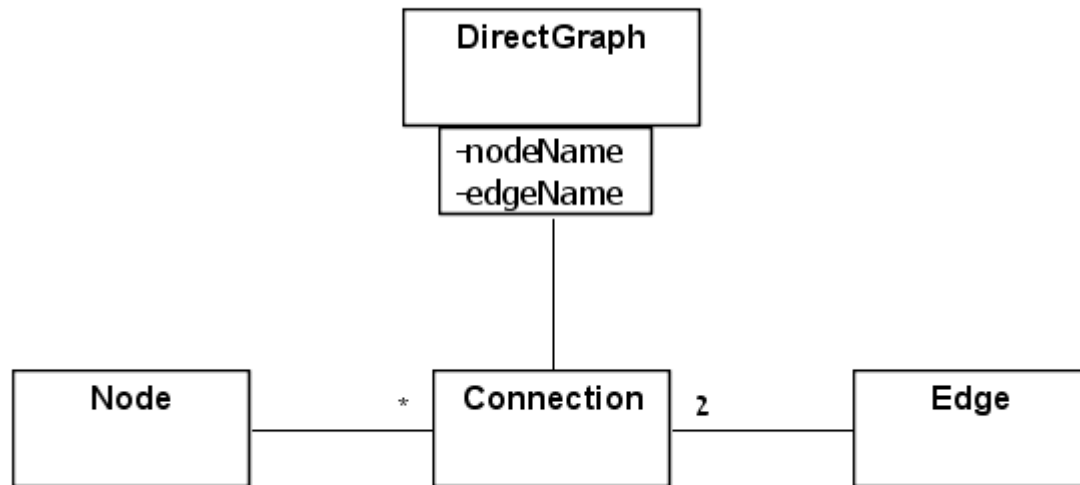
- Qualification cascade:



# *Qualified Associations - 4*

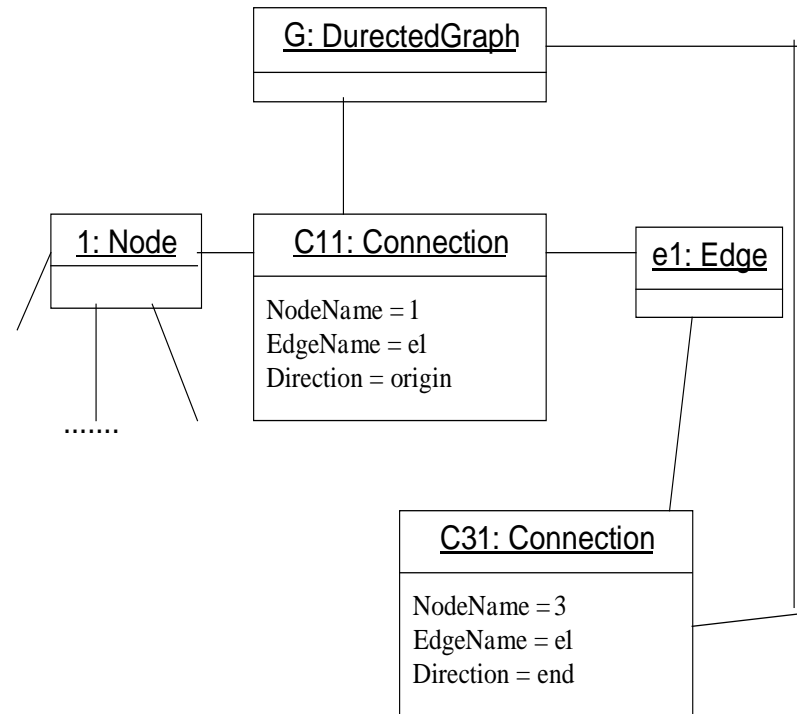
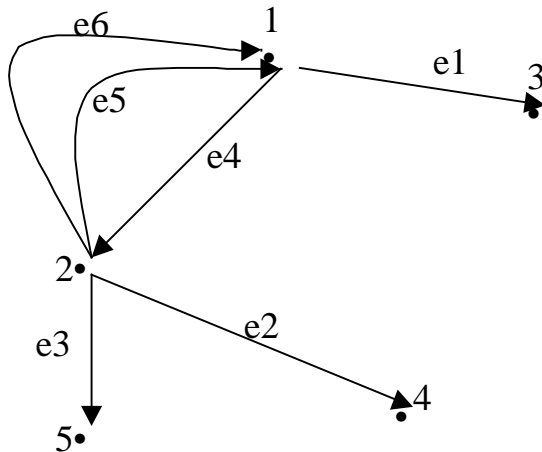
---

Compound Qualifier:



# Qualified Associations - 5

This class diagram can describe the graph and the legal instance diagram:

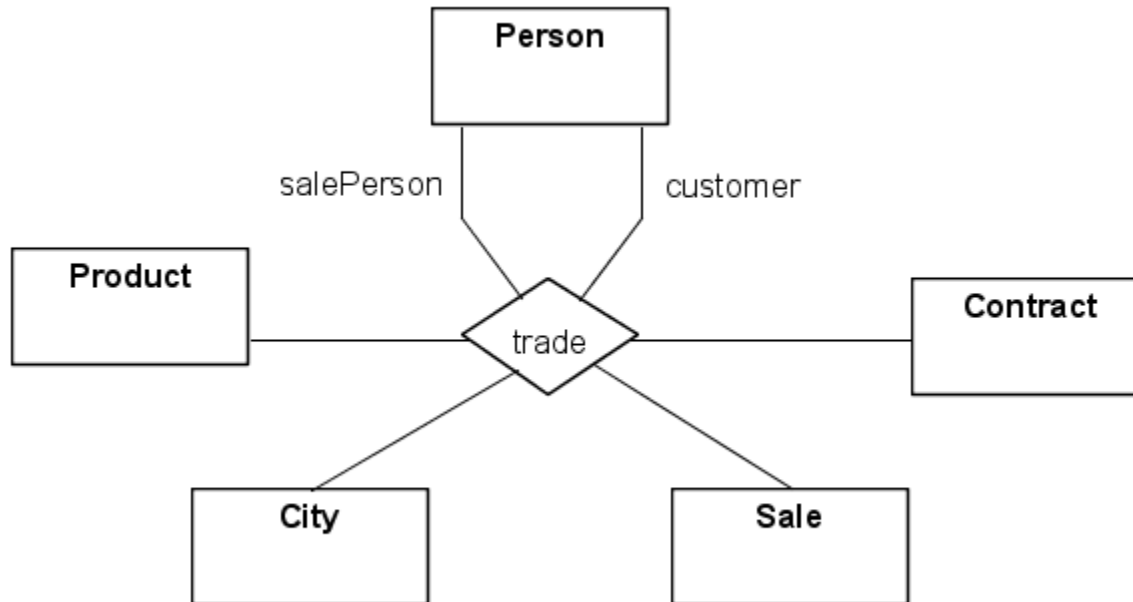


***Note that it has also non-intended but legal instance diagrams!***

# *N-ary Associations*

---

- An n-ary association is an association among n classes.
- Its links are n-tuples.
- No convention for multiplicity constraints.



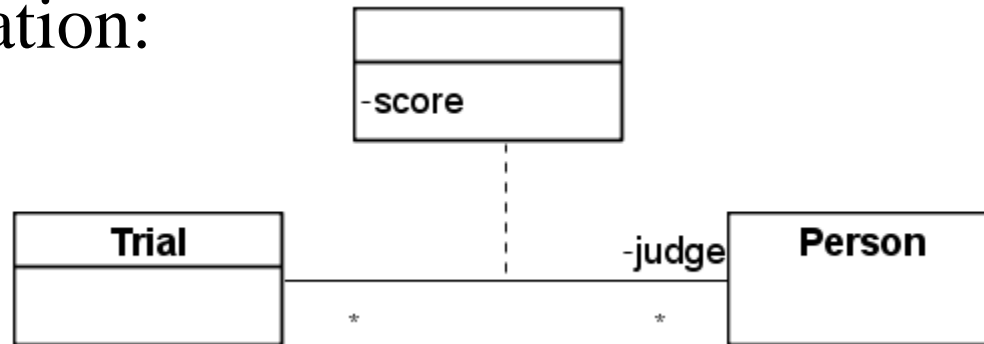


## *5. Associations as objects*

# Link Attributes - 1

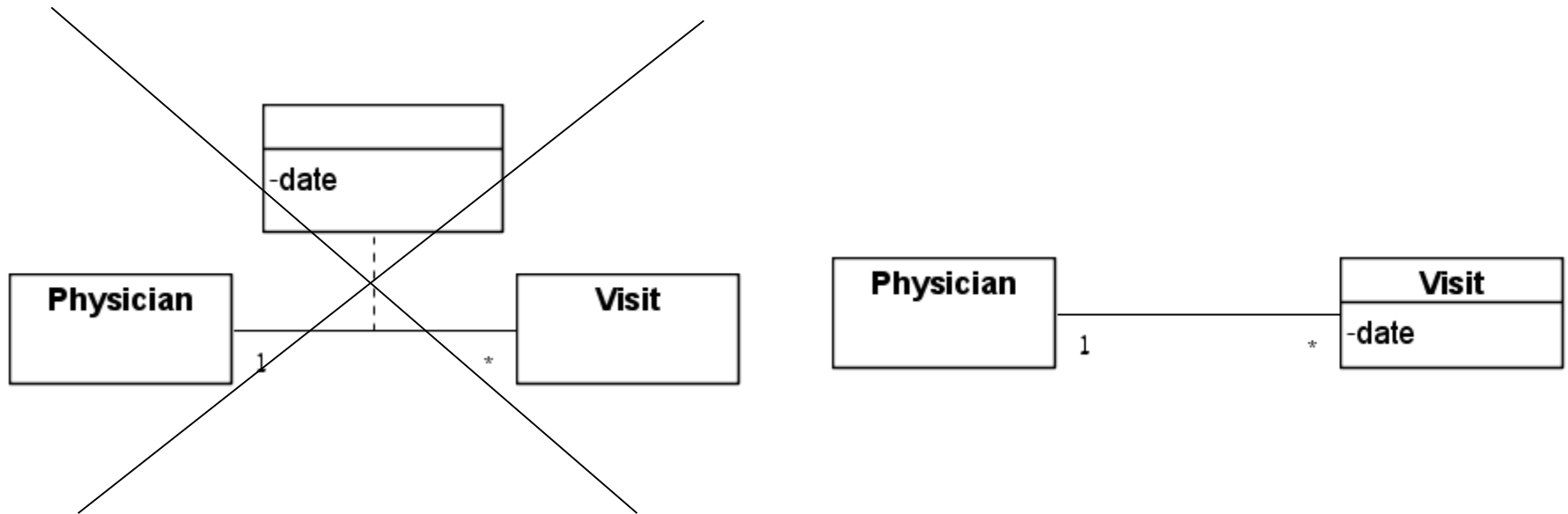
---

- A **link attribute** is a named property of an association that describes a value held by each link in the association.
- Identified – by following the *adverbs*, or by abstracting known values.
- Visual notation:



## Link Attributes - 2

- Link attributes are intuitively understandable for associations with *many-many* multiplicity – as in the *Trial-Person* association.
- But – can be confused with object attributes in *1:1* or *1:many* associations



# *Association classes - 1*

---

- An **association class** is an association whose links have identities.
- They can participate in other associations
- An association class is both:
  - A Class.
  - An Association.

## *Association classes - 2*

---

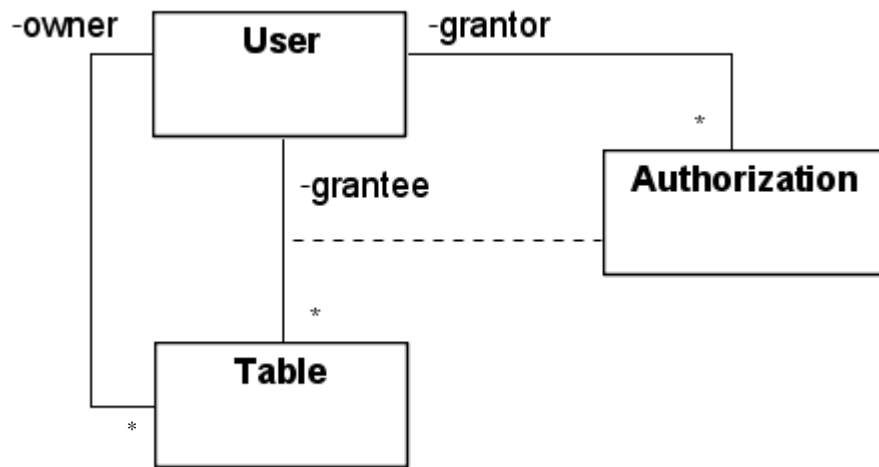
- The link identities derive from the identities of the related instances.
- Association classes add an extra constraint:
  - *A single instance of the association* between any 2 instances of the associated class - the regular association constraint!
- An association class can have *attributes*.

# *Association classes : Examples - 1*

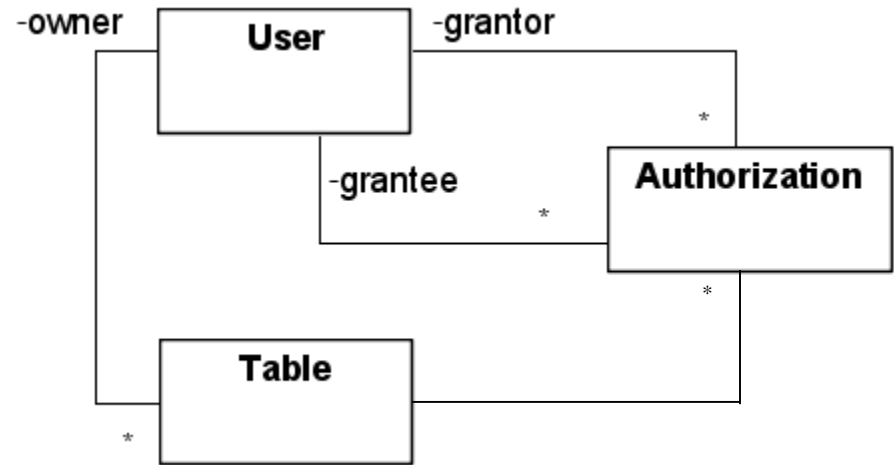
---

- *Authorization in a relational DBMS:*
  - *A user may own multiple tables.*
  - *The owner of a table may authorize 1 or more other users access to the table.*
  - *An authorized user may grant more permissions.*
- *Example:* A owner of T. B, C are grantees for T by A, the grantor. B can authorize D, E, etc.

# Association classes : Examples - 1



*Preferred model with  
association class*



*Degraded model with  
ordinary class*

*Drawbacks of the regular class model:*

- 1. Symmetric dependencies between Authorization and the User and Table classes.*
- 2. grantee-Table association and its unique multiplicity dependency on grantor is lost.*
- 3. No traversal path for reading the diagram.*

# *Association classes : Examples - 1*

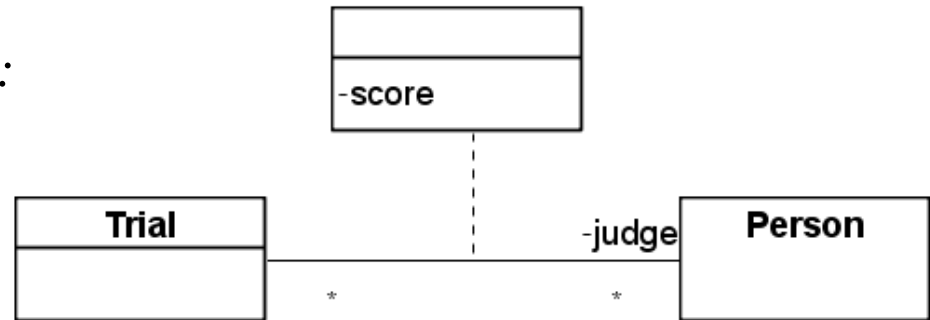
---

- *Does any class model (of the two given) capture all the requirements? Here are the requirements again:*
  1. *The user may own multiple tables.*
  2. *The owner of a table may authorize 1 or more other users access to the table.*
  3. *An authorized user can grant more permissions.*
- *Requirements 2 and 3 are not fully enforced! Why?*
  - *To see that – provide a **legal instance diagram** that does not meet the requirements.*
  - ***or** – rewrite the class diagrams in logic and provide a model that does satisfy the requirements.*



# Association classes : Examples - 2

- Consider the former class diagram:



- **New information:**

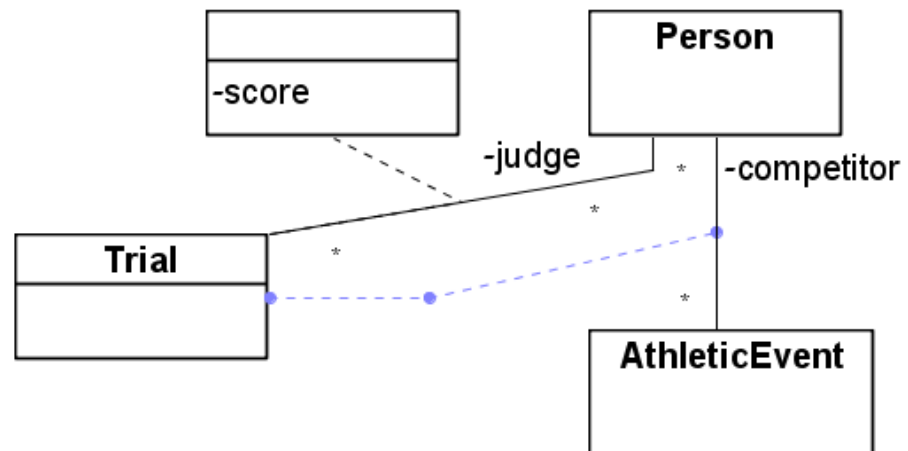
- A participation of a competitor in an athletic event is termed a trial.
- There are several competitors within an athletic event.
- In every trial there are several judges. Each judge assigns a score.
- A judge might serve in many trials.

- **Trial uniquely characterizes a competitor-AthleticEvent pair**

- **Trial is an association**

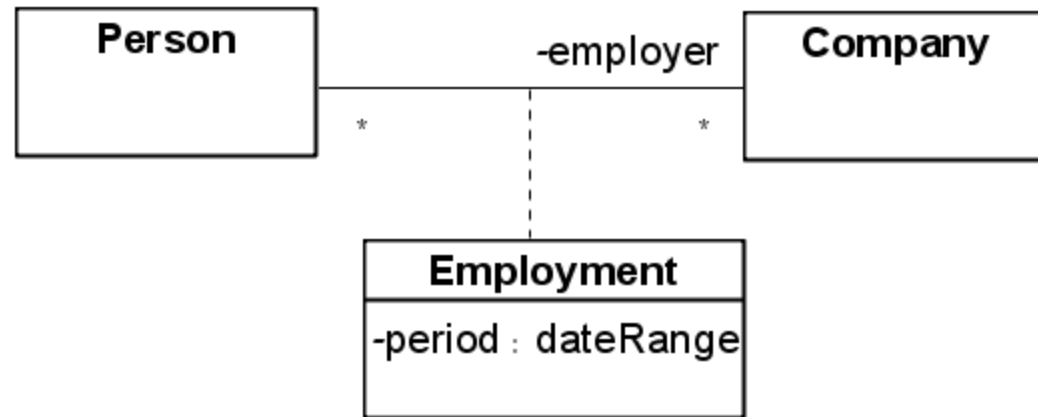
- A trial is related to judges

- **Trial must be a class!**



# Association classes : Examples - 3

*Employment  
association (a):*



*Employment  
association (b):*



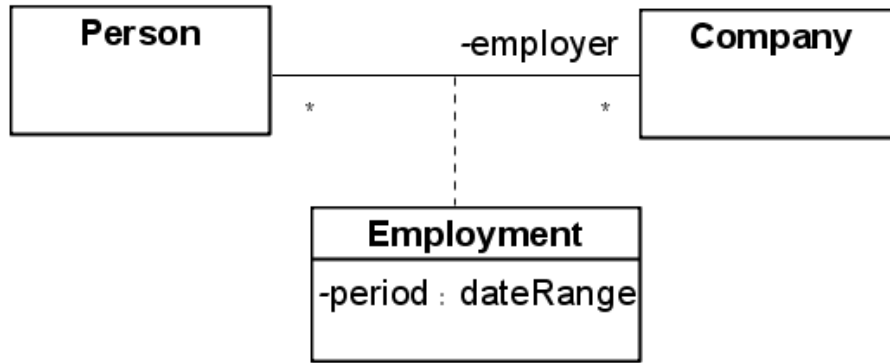
# *Association classes : Examples - 3*

---

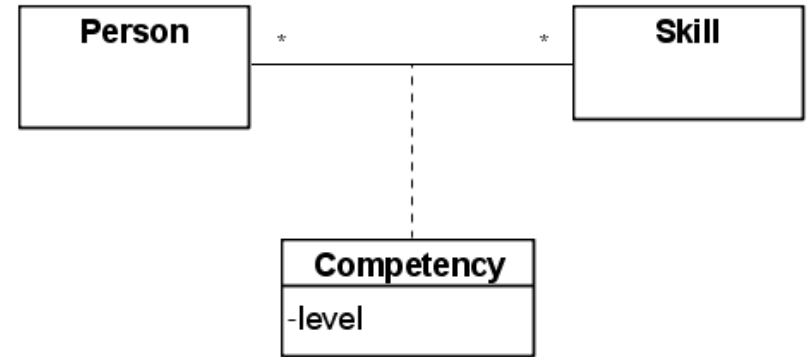
- *In (a), every person can be employed only once in a single company! (why?)*
- *In (b), a person can work for the same company several times.*
- *Note the changes in the multiplicity constraints.*

# Association classes : Examples - 4

*Compare the following :*



*(a):*



*(b):*

*(a) is not appropriate if people can work for the same company in several different periods.*

*(b) states the reasonable situation: A single person has a single level for a given skill.*

## *6. Class Hierarchies*

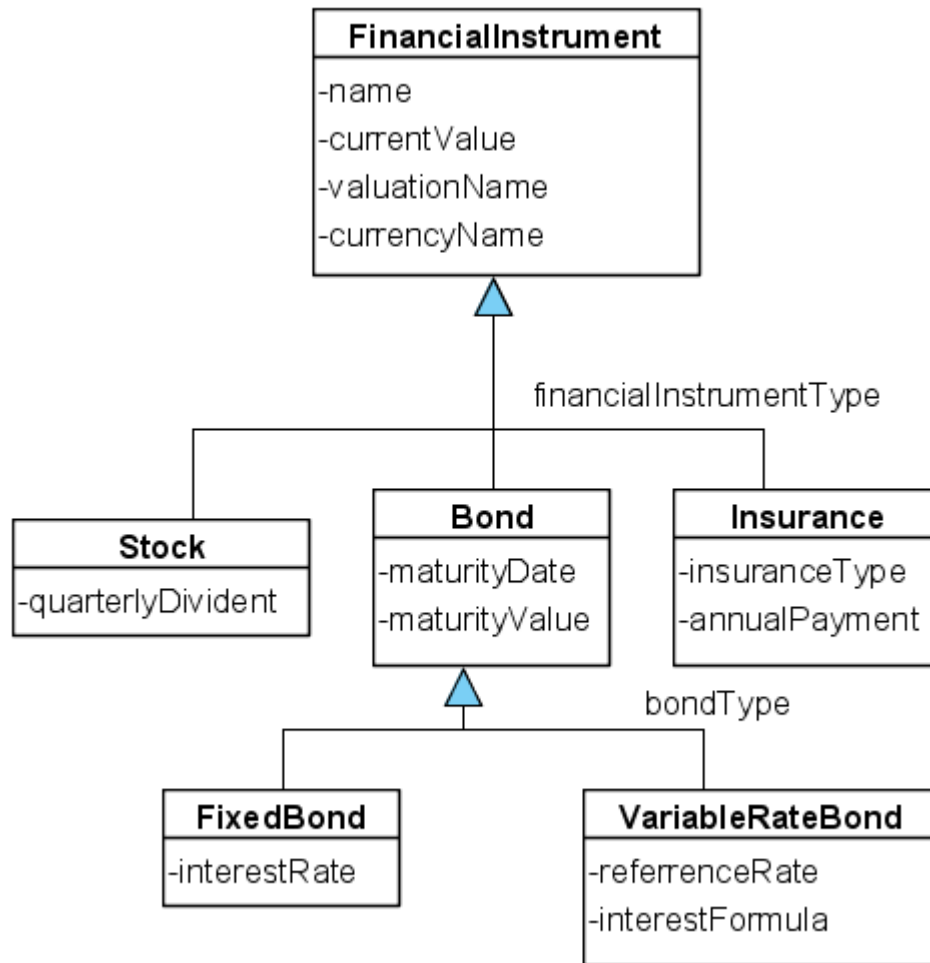
# *Class Hierarchies : Classification*

---

- **Class Hierarchy** is a new kind of relationship between classes: It relates a class (the *super-class*) with some of its *sub-classes*.
- Class Hierarchy organizes classes by:
  - **Similarity.**
  - **Differences.**
- Class Hierarchy arises either from **generalization** or from **specialization (sub-typing)**:
  - Generalization -- a bottom-up notion.
  - Specialization -- a top-down notion.

# *Class Hierarchies : Classification*

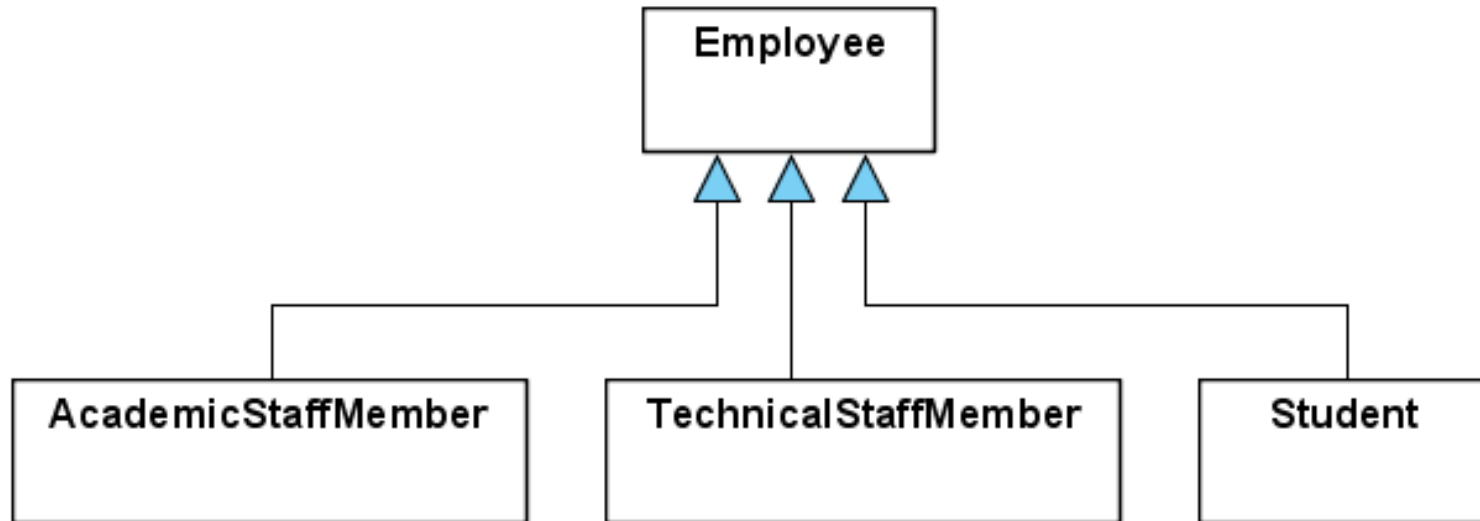
---



Size of a sub-classes group is greater or equal to 1.

# *A Class Hierarchy Without Grouping*

---



**Size of each “sub-classes group” is equal to 1.**



# *Class Hierarchy : Semantics - 1*

---

- A class hierarchy denotes the *subset* relation between the involved classes:
  - *Stock, Bond, insurance* are subsets of *FinancialInstrument*
  - *FixedRateBond, VariableRateBond* are subsets of *Bond*

*In logic:*

$$\forall b \ (Bond(b) \rightarrow Financialinstrument(b))$$

# *Class Hierarchy : Semantics - 2*

---

- The **subset semantics** implies :
  - **Inheritance** – A subclass inherits from its super-classes their attributes, associations, operations, state-charts
  - **Class hierarchy is transitive** - A class hierarchy specification associates a single *direct super-class* with each subclass, but possibly multiple *non-direct super-classes*.

# *Class Hierarchy Group Annotations - 1*

---

- **Kinds of group annotations:**
  - **Discriminator.**
  - **Generalization Constraints:**
    - Incomplete
    - Complete
    - Disjoint
    - Overlapping
  - **Dynamic.**

# *Class Hierarchy Group Annotations - 2*

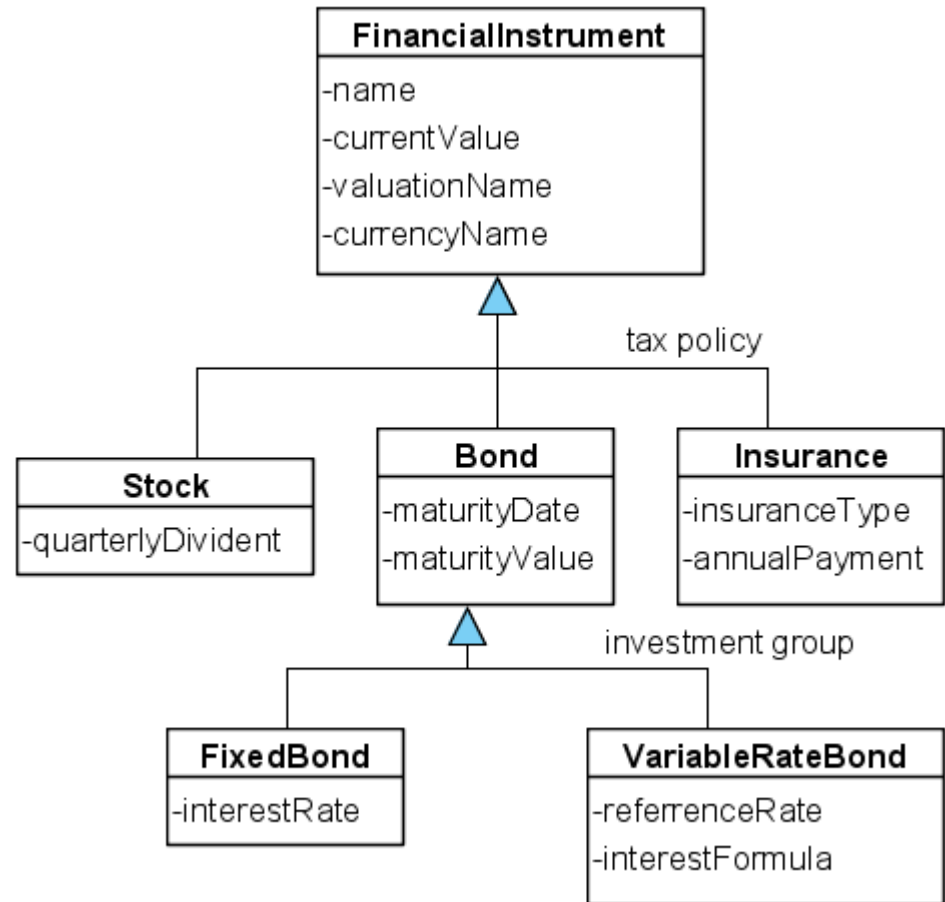
---

- In UML2:
  - The **discriminator** annotation is removed.
  - Class hierarchy groupings are termed **generalization sets**.

# Group Annotations – 3 : Discriminator

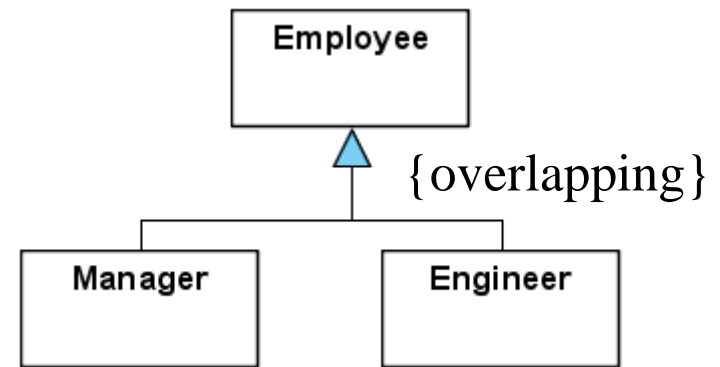
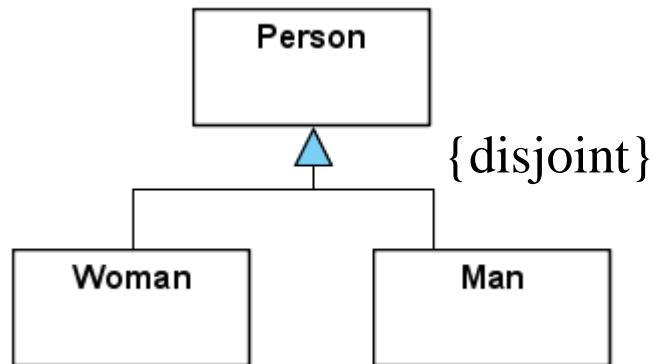
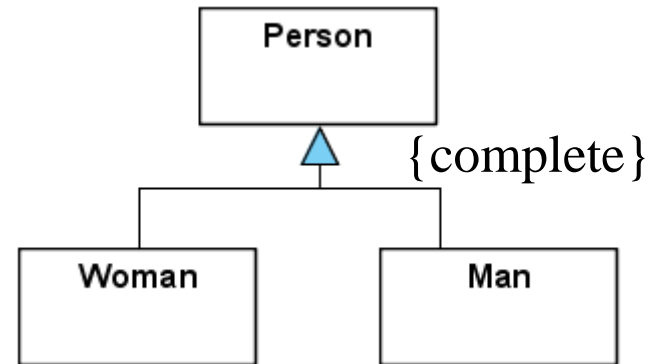
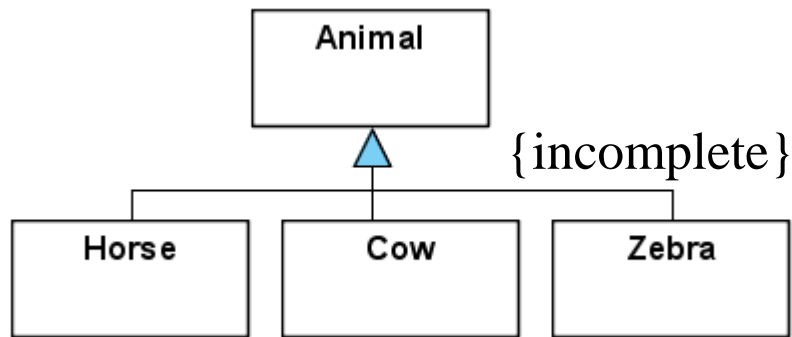
A **discriminator** to a class hierarchy group specification is an attribute of the super-class whose values distinguish between objects of the subclasses. For each subclass it assigns a common value to all of its objects.

*investment group  
is an attribute of Bond. All  
FixedRateBond instances  
have a common investment  
group value which is  
different from the common  
investment group value of  
VariableRateBond  
instances.*



# Group Annotations – 4

- Generalization Constraints.
- Built-in generalization constraints:



# Group Annotations – 5

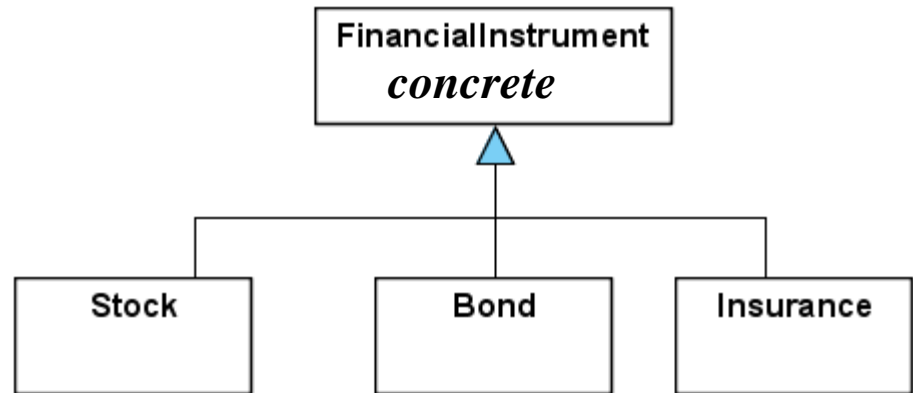
---

- Generalization constraints in logic :
  - Complete :  $\forall x(Person(x) \rightarrow Man(x) \vee Woman(x))$
  - Incomplete :  $\exists x(Person(x) \wedge \neg Man(x) \wedge \neg Woman(x))$
  - Disjoint :  $\forall x(Man(x) \leftrightarrow \neg Woman(x))$
  - Overlapping :  $\exists x(Man(x) \wedge Woman(x))$
- Overlapping and disjoint semantics is ambiguous. can refer to:
  - the intersection of *all* subclasses in the group – *non-liberal*.
  - the intersection of *every* two subclasses in the group – *mildly-liberal*.
  - the intersection of *some* two subclasses in the group – *liberal*.

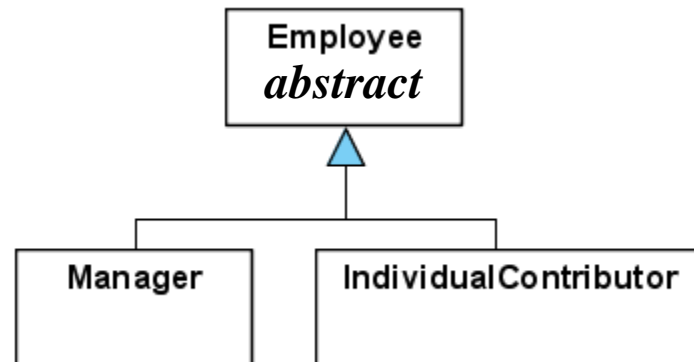
# Group Annotations – 6

- **Abstract and Concrete Classes.**

*A **concrete class** – can have direct instances.*



*An **abstract class** – no direct instances.*





# *Group Annotations – 7: Inter-Relationships*

---

- A discriminator implies:
    - a *disjoint* class annotation.
  - *Abstract* super-class annotation is equivalent to a *complete* group annotation.
  - *Concrete* super-class annotation is equivalent to a *incomplete* group annotation.
- Complete* and *incomplete* annotations are contradictory.
- *Disjoint* and *overlapping* annotations are contradictory.

# *Group Annotations – 8: Dynamic*

---

- **Dynamic annotation** – marks that objects can change types within a class hierarchy structure, during their life time.
- **Class Person:**
  - Subclasses Female, Male – with discriminator sex;
  - Usually will not be marked as `<<dynamic>>`.
- **But: Class Person:**
  - Subclasses Manager, Engineer, Salesperson
    - with Discriminator job;
  - can have the annotation `<<dynamic>>`

# *Group Annotations – 9: Multiple sub-groups*

---

- A class can have **multiple sub-groupings** – each distinguished by a discriminator. All subclasses with the same discriminator are disjoint:
- *Class Person :*
  - *Subclasses Female, Male – with discriminator sex;*
  - *Subclasses Doctor, Nurse, PhysioTherapist - with discriminator role;*
  - *Subclass Patient.*
- *The super-class is **abstract**.*
- CASE tools do not enforce group annotation consistency.

# *Group Annotations – 10*

---

- Support by implementation languages
- Programming languages do not support:
  - **Class hierarchy grouping.**
    - **Discriminator** annotation.
  - **Overlapping** annotation: An object cannot be an instance of multiple classes that are not related by sub-typing relationships. Only class hierarchy polymorphism is supported
  - **Complete/incomplete** is supported via the Abstract class annotation.
  - **Dynamic** annotation.

# *Kinds of Class Hierarchies - 1*

---

- **Simple class hierarchy:** A class is a sub-class of at most a single class hierarchy construct.
  - Implies **single inheritance**.
  - A **tree structure** class diagram.
  - **Entity-relationship** modeling supports only simple class hierarchies.
- **Complex class hierarchies:** A class might have multiple direct super-classes.
  - Implies **multiple inheritance**.
  - A **Directed Acyclic Graph (DAG)** class diagram.
  - Multiple inheritance might lead to contradictions.
    - Requires policies for resolving inheritance contradictions.

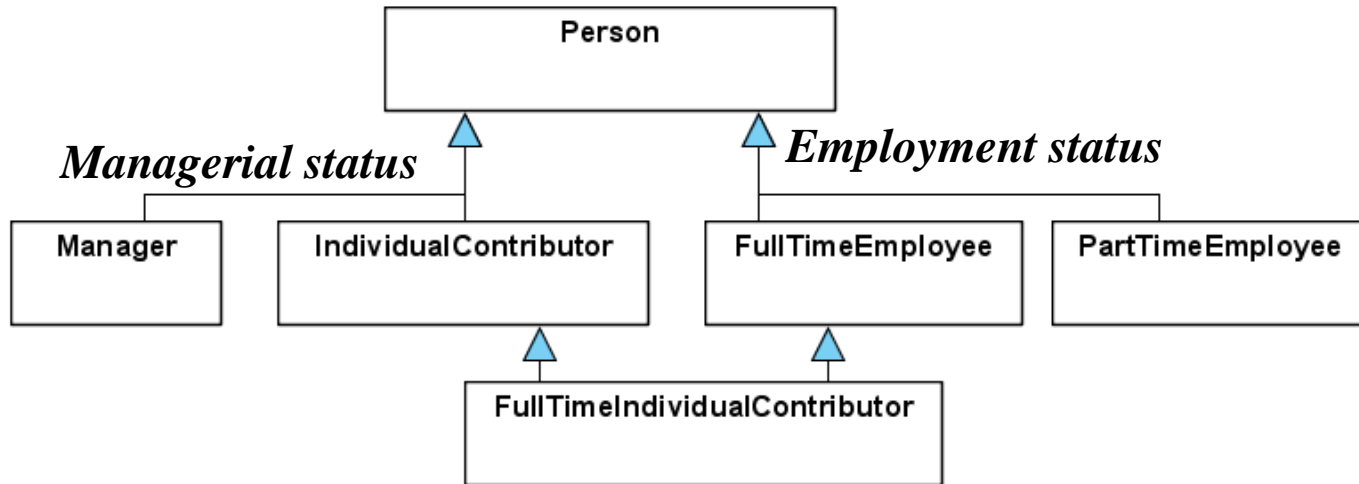
# *Kinds of Class Hierarchies - 2*

---

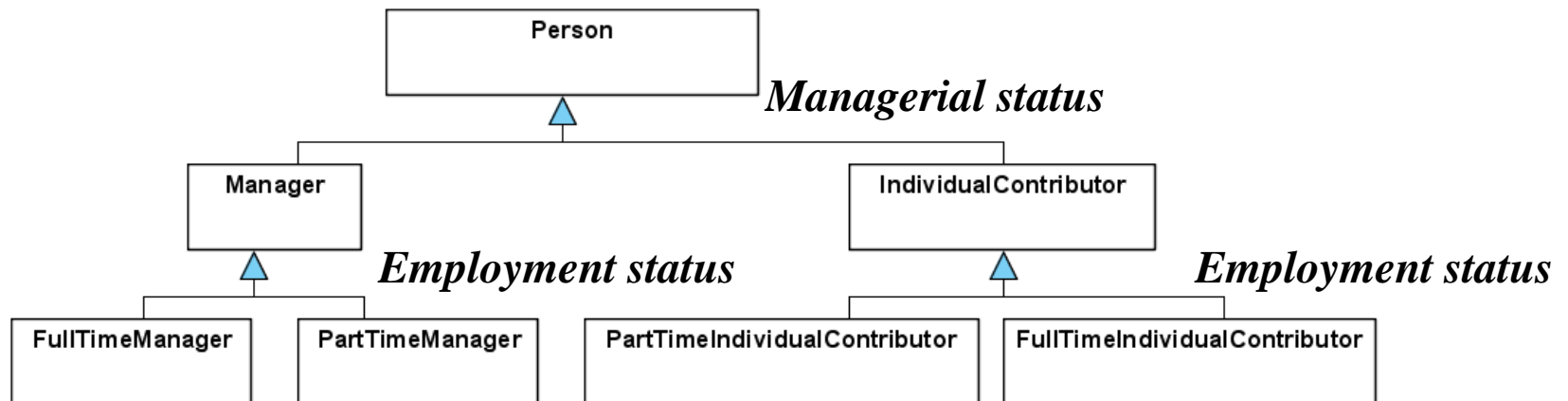
- Simple class hierarchy:
  - **Restricted expressivity:** In reality objects might have multiple categorization and multiple responsibilities:
    - e.g., FullTime-Consultant, Local-Car, Student-Employee, ...
  - **Supported** by all Object-Oriented programming languages.
- Complex class hierarchies:
  - **Real world compatible expressivity.**
  - **Not supported** by some Object-Oriented programming languages.
  - Requires methods for transforming complex hierarchies into simple ones.

# Removing complex class hierarchies - 1

Multiple inheritance from a single ancestor with different discriminators:

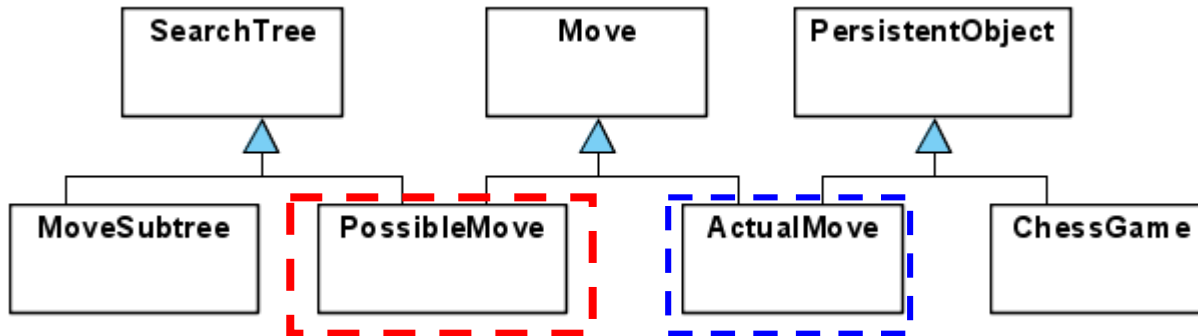


Removing multiple inheritance by **factoring**:

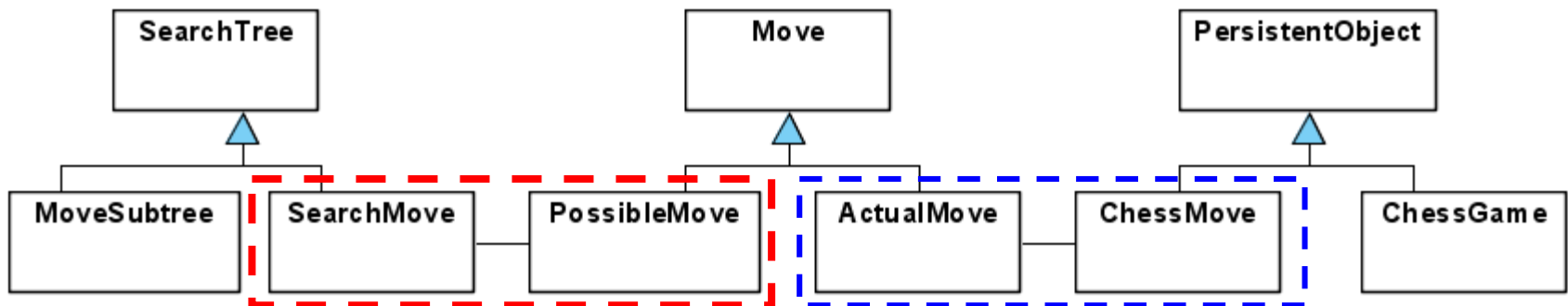


# Removing complex class hierarchies - 2

Multiple inheritance without a common ancestor:



Removing multiple inheritance by **subclass fragmenting**:

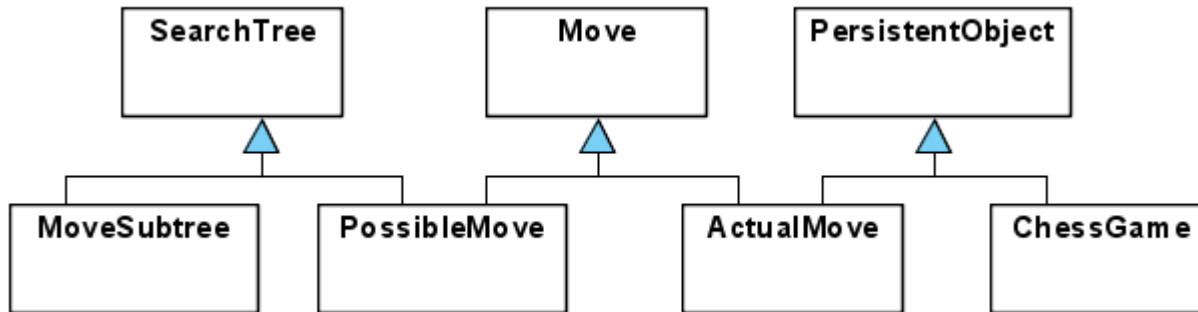




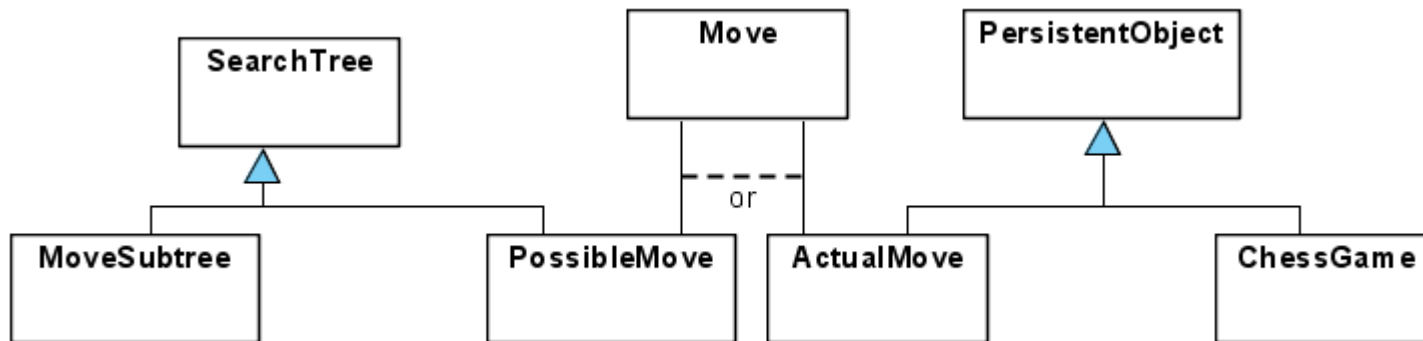
# *Removing complex class hierarchies - 3*

---

Multiple inheritance without a common ancestor:



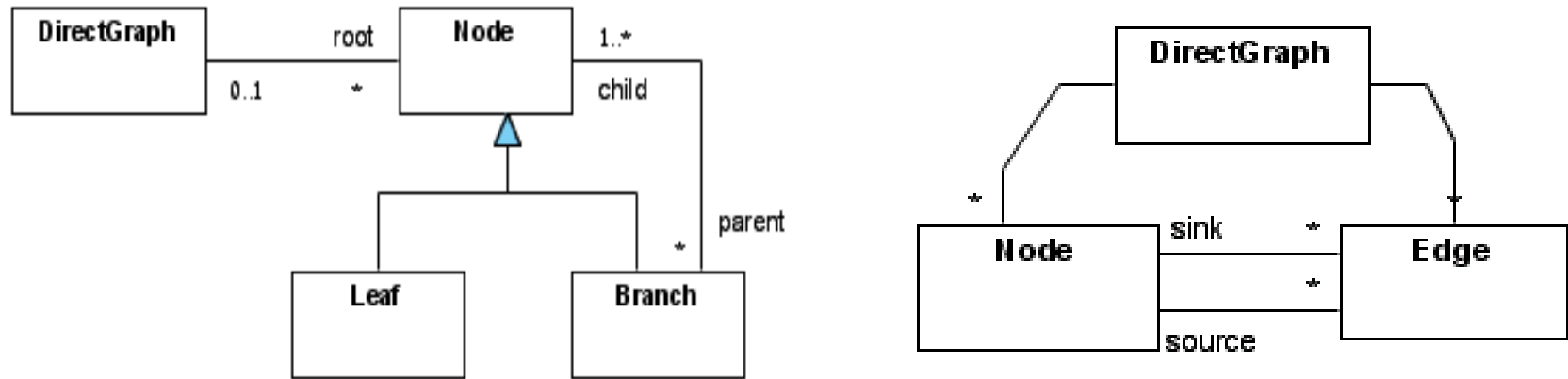
Removing multiple inheritance by replacing generalization with **exclusive-or associations**:



# Association cycles within class hierarchies

## *Example: Directed graph.*

*Consists of nodes and edges. Each edge connects 2 nodes. An edge has a direction.*



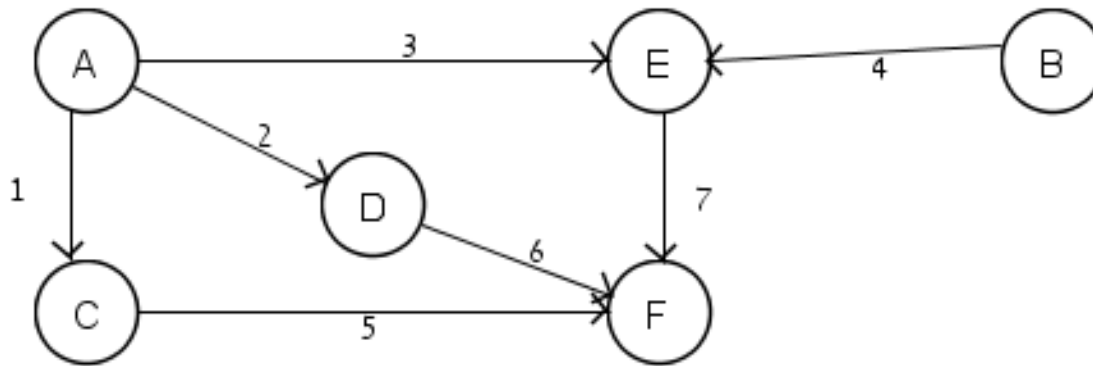
**Left :** Directed graphs with at most a single edge between 2 nodes – edges are captured by the Branch – Node association.

**Right :** Any directed graph.

# *Association cycles within class hierarchies*

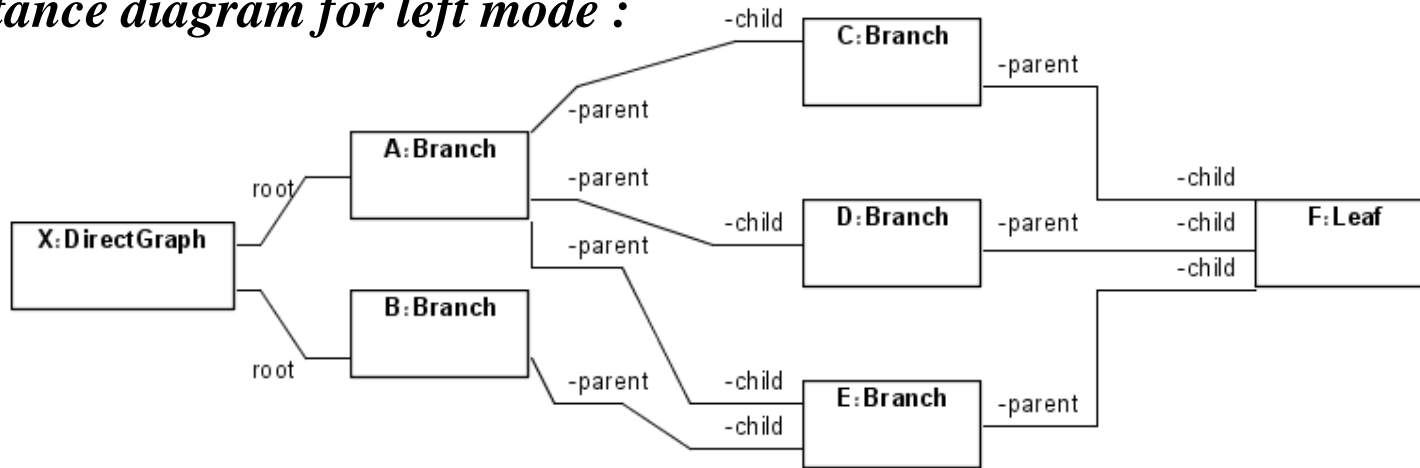
---

Observing an instance diagram can help selecting a desired class diagram.

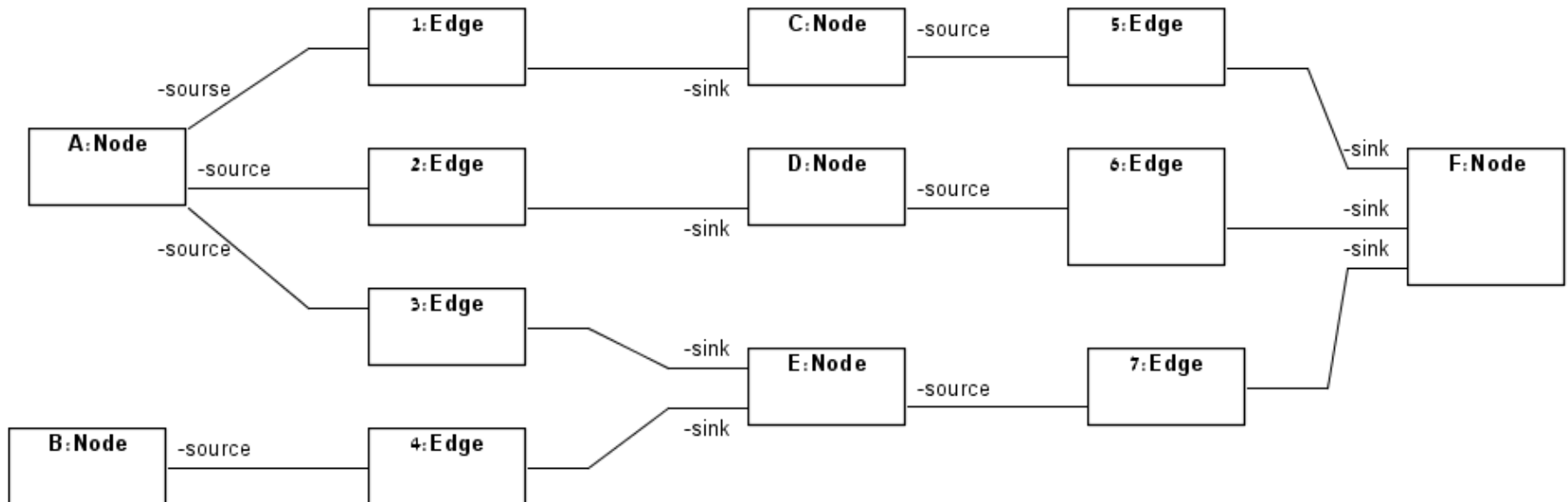


# Association cycles within class hierarchies

*Instance diagram for left mode :*



*Instance diagram for right mode :*



# *Association cycles within class hierarchies*

---

- *Evaluation:*

- *Right diagram:*

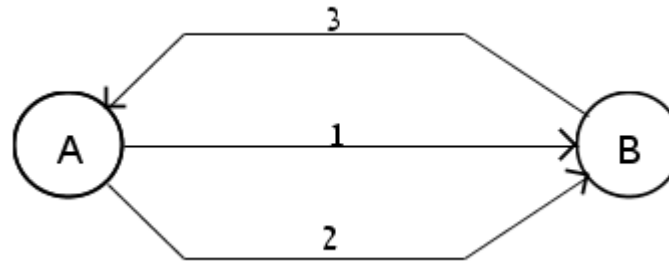
- *More complex.*
    - *Both Nodes and Edges are conceived as objects.*

- *Left diagram:*

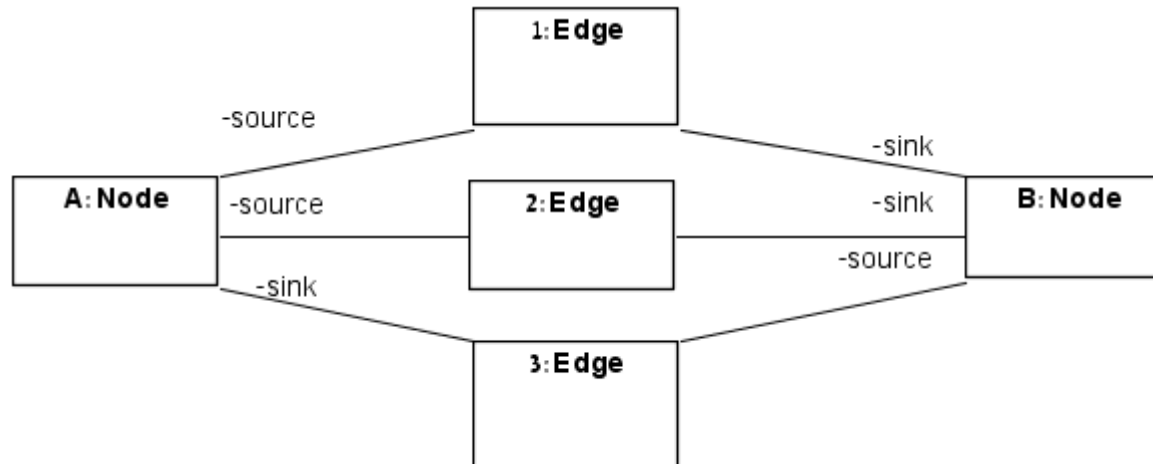
- *Simpler.*
    - *Direct correspondence to the graph model.*
    - *The left diagram provides a structural enforcement of the constraint:*
      - *At most a single edge between any two nodes.*

# Association cycles within class hierarchies

Another directed graph. This graph **cannot** be represented by the left model.



*Instance diagram for right mode :*



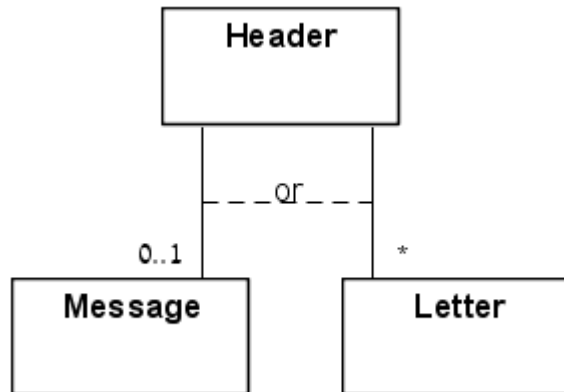
## *7. Constraints on Associations*

# *Constraints on associations: exclusive-or:1*

---

Exclusive-or is a grouping of associations that involves a **single source class**.

Visual notation:



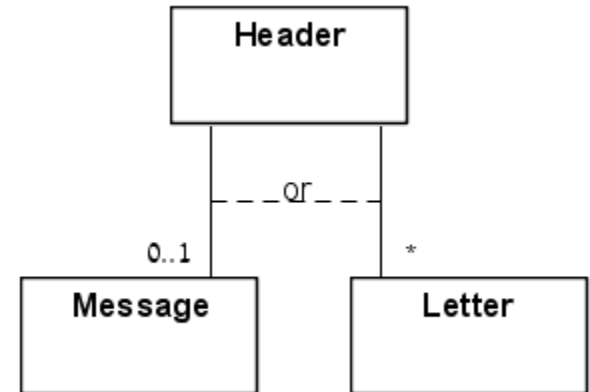


# *Constraints on associations: exclusive-or:2*

---

- The semantics is that an object of the source class can participate in at most single link, out of all associations in the grouping.

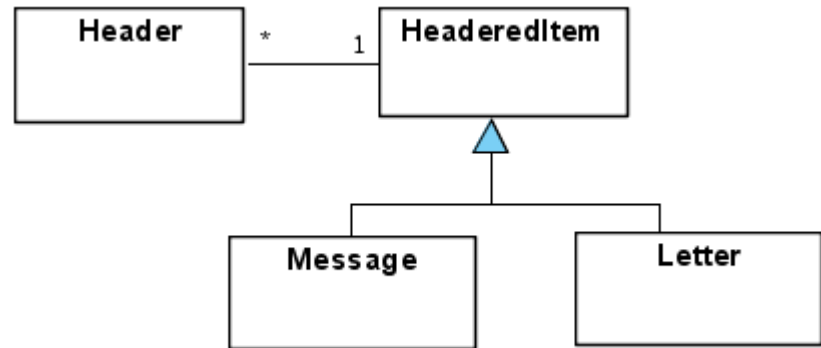
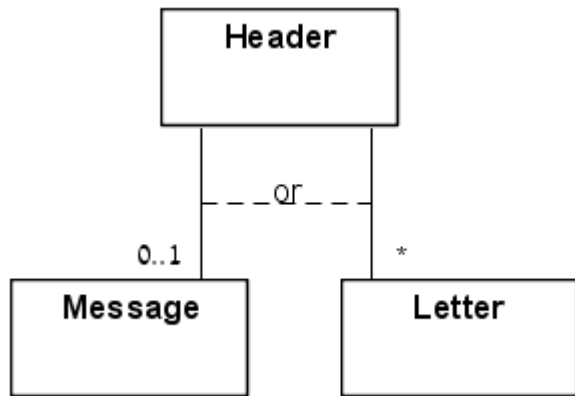
$$\begin{aligned} \forall x, y, z (Header(x) \wedge headerMessage(x, y) \rightarrow \\ \neg headerLetter(x, z)) \wedge \\ \forall x, y, z (Header(x) \wedge headerLetter(x, y) \rightarrow \\ \neg headerMessage(x, z)) \end{aligned}$$



- No exclusive-or grouping overlap.
- The semantics of exclusive-or constraint dominates the regular semantics of multiplicity constraints.

# *Constraints on associations: exclusive-or:3*

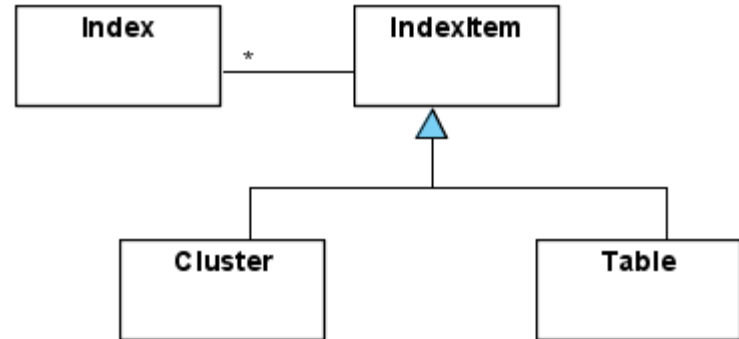
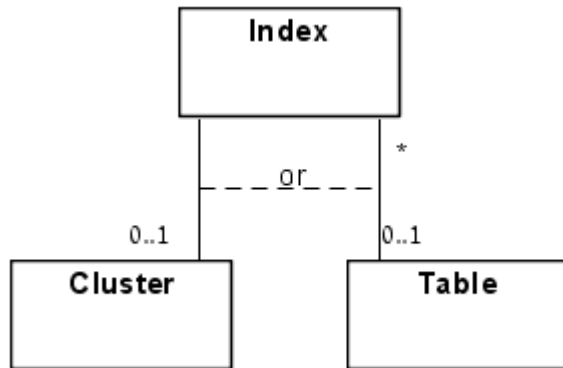
- Modeling with exclusive-or is more accurate:



- The left model is preferable since:
  - It enables a single link for every header instance (source class object).
  - The right model loses the multiplicity differences between the two associations.

# *Constraints on associations: exclusive-or:4*

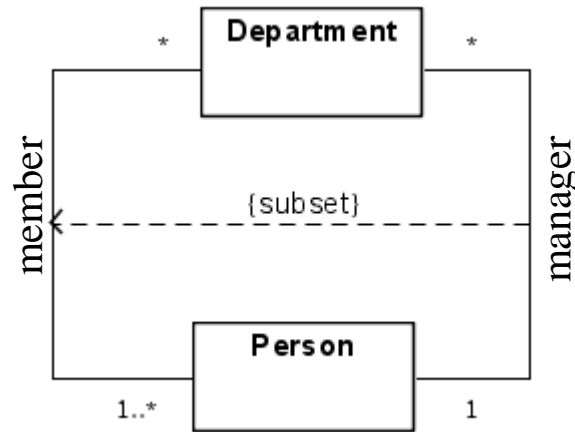
---



*What is the preferred model?*

# Constraints on associations: inclusion-1

Association inclusion means inclusion of the link sets.



$$\forall x, y (manager(x, y) \rightarrow member(x, y))$$

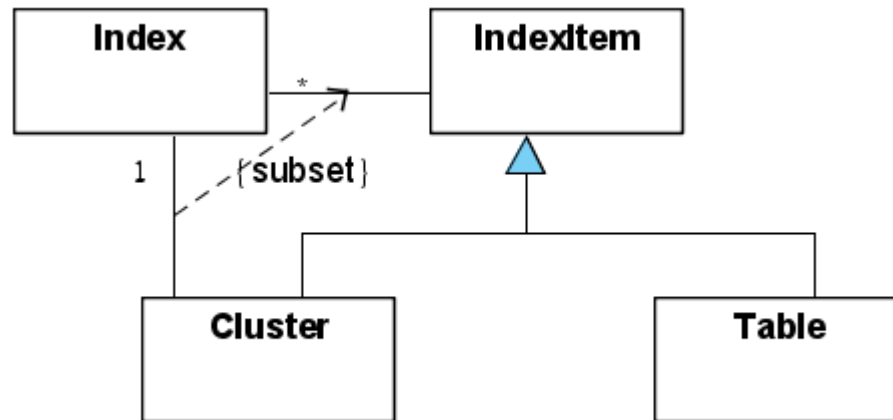
Does association inclusion extend UML expressivity?

Answer: NO (why?)

## *Constraints on associations: inclusion-2*

---

Association inclusion constraints are popular especially in the presence of class hierarchies, where the association of a sub-class provides more specific information.



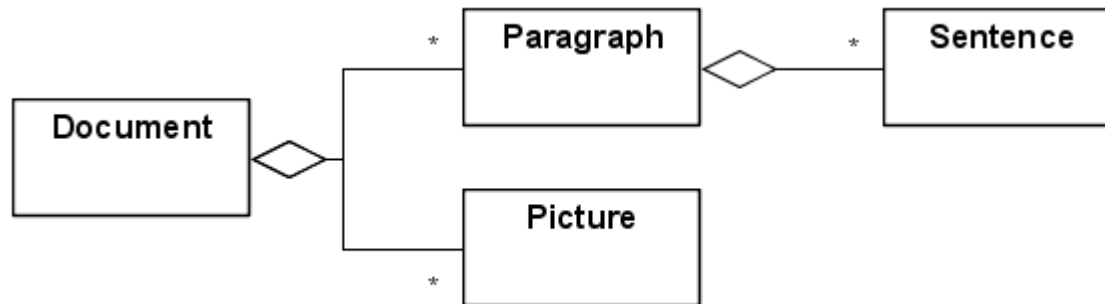
## *8. Aggregations*

# Aggregation - 1

---

**Aggregation** : is an association between a class – the assembly – to its parts – the component classes.

Visual Notation:



# *Aggregation - 2*

---

- The **aggregation** relation is singled out in UML as a special kind of association.
- **Aggregation is transitive.**
  - **The transitive closure** of an assembly is the full set of its parts.
- **Aggregation is anti-symmetric.**
- **Aggregation is not ordered.**

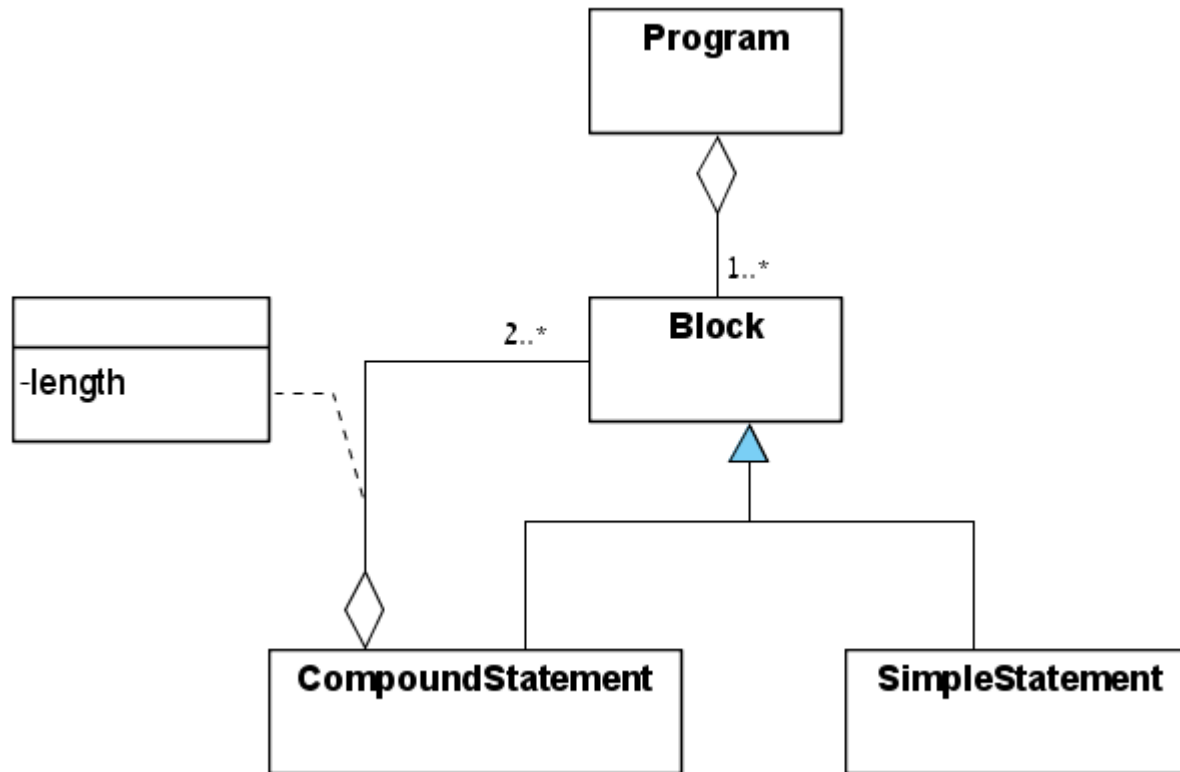
Different aggregation assemblies are not visualized. May be marked/commented



# *Recursive Aggregation*



---

Creates Tree or DAG structures of unbounded depth.



# *Physical and Logical Aggregations - 1*

---

- **Logical (catalog) Aggregation** – a component can be used in multiple assemblies.
- **Physical (composition) Aggregation** – a component belongs to at most a single assembly.
- **UML offers two kinds of aggregation notations:**
  - Logical (catalog) aggregation – denoted: 
  - Physical (composition) aggregation – denoted: 
- ***Examples:***
  - *A concrete car vs. a car model.*
  - *The Program aggregation – physical or logical?*

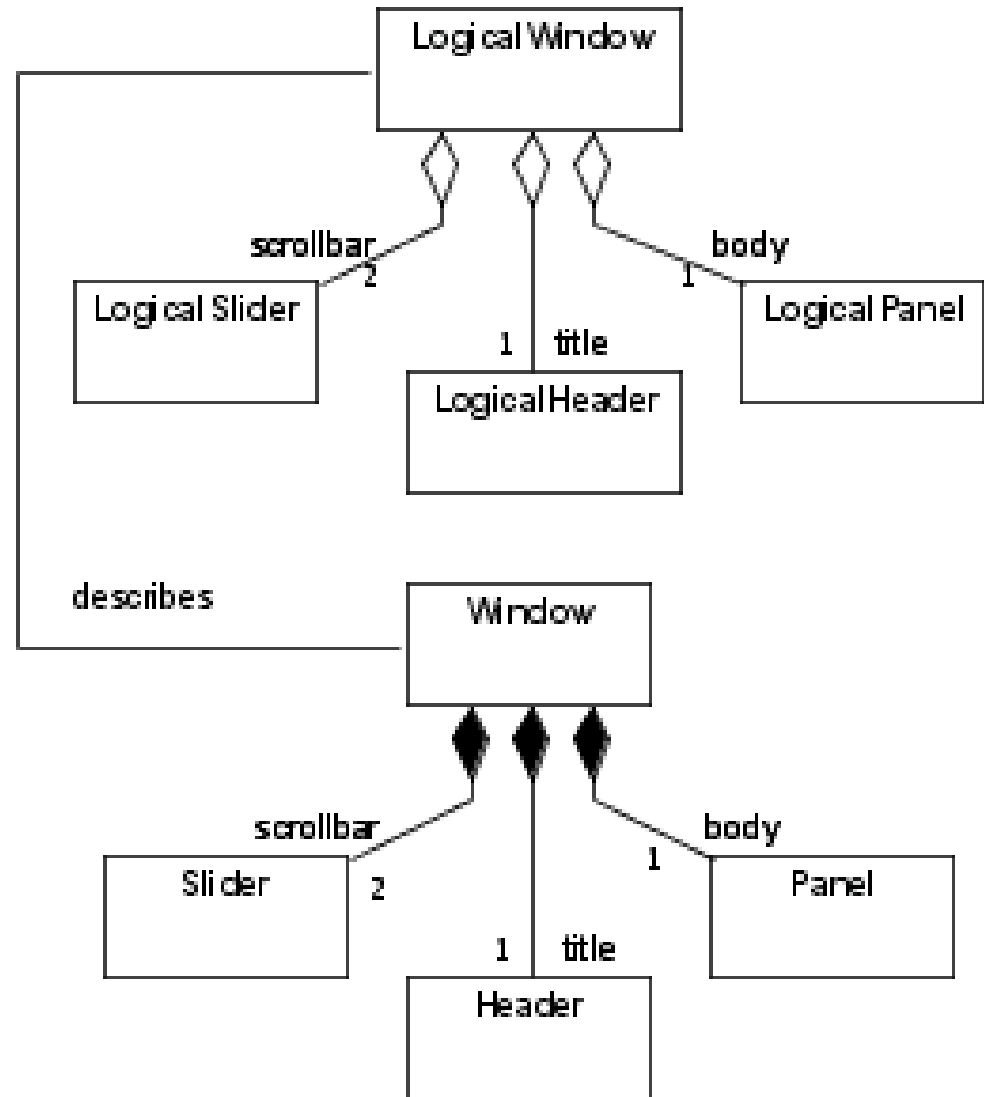
# *Physical and Logical Aggregations - 2*

---

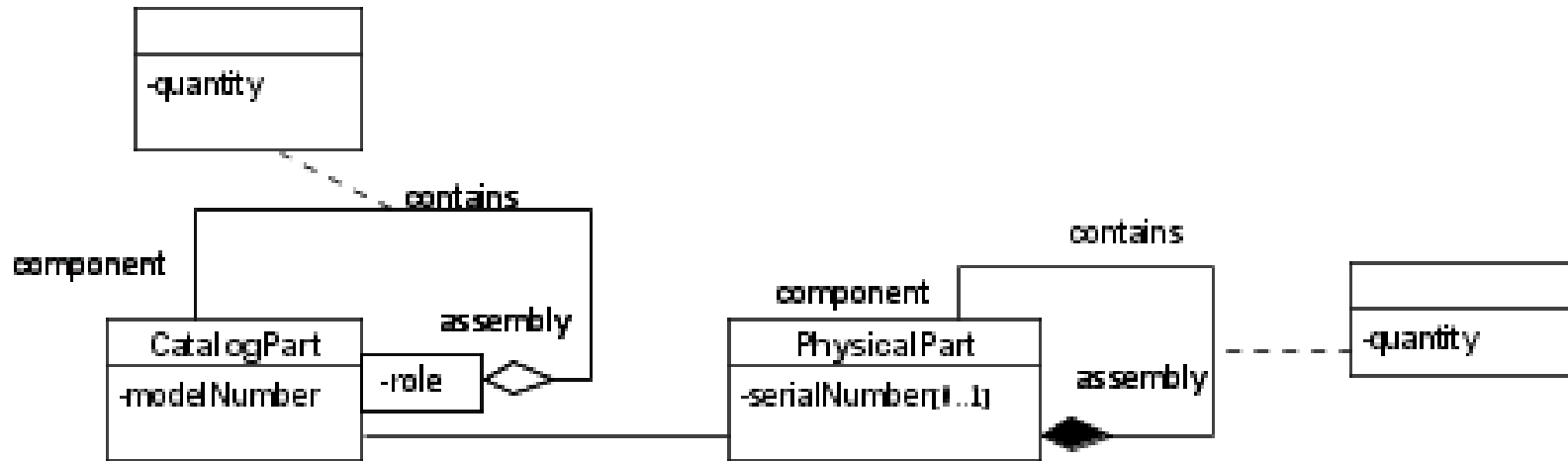
- The instances of a physical aggregation are **trees**.
- The instances of a logical aggregation are **Directed Acyclic Graphs (DAGs)**.
- The instances of a recursive aggregation have **unbounded depth**.
- Tree aggregation can support:
  - Propagation of properties.
  - Default values.
- DAG aggregation requires resolution of clashes.

# Physical and Logical Aggregations - 3

Logical elements exist independently of physical ones.



# Physical and Logical Aggregations - 4



- A **catalog part** may belong to multiple assemblies and contain multiple parts.
- A **catalog part** is identified by its **role in the assembly**.
- A **physical part** may belong to at most a single assembly and contain multiple parts.
- There might be indistinguishable **catalog parts** (like nail kinds). The **Contains** associations has a **quantity** attribute for quantifying these parts.

## *9. Relationships Overview*

# *Relationships reviewed in UML*

---

- All relationships are visualized by lines between classes.
- **Sub-typing / generalization** (in class hierarchies). Denoted by a solid line and an empty arrow.
- **Instantiation**. Denoted by a dashed line labeled <<instance>>.
- **Association**. Denoted by a solid line.
- **Aggregation**. Denoted by a solid line ended with a solid or empty diamond.

# *Sub-typing\Generalization*

---

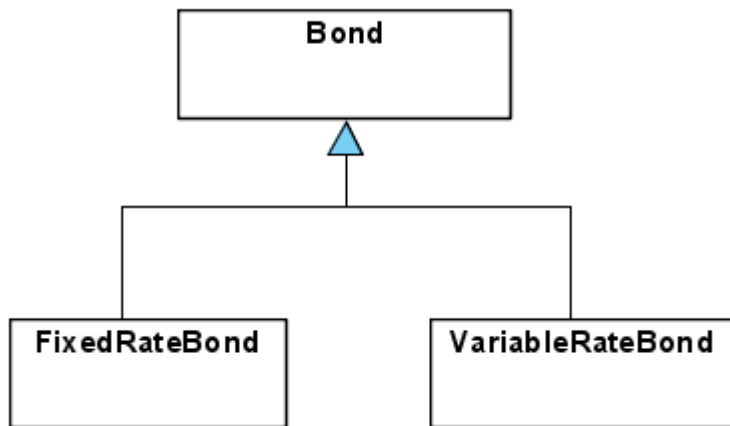
- *Not inter-object relationships!*
- **They distinguish:**
  - **Similarities and differences.**
  - Emphasize different aspects of instances.
  - An **OR** grouping.
  - Involve classes alone.
  - Important during design, for **reuse**.
  - Eliminate **duplications**.



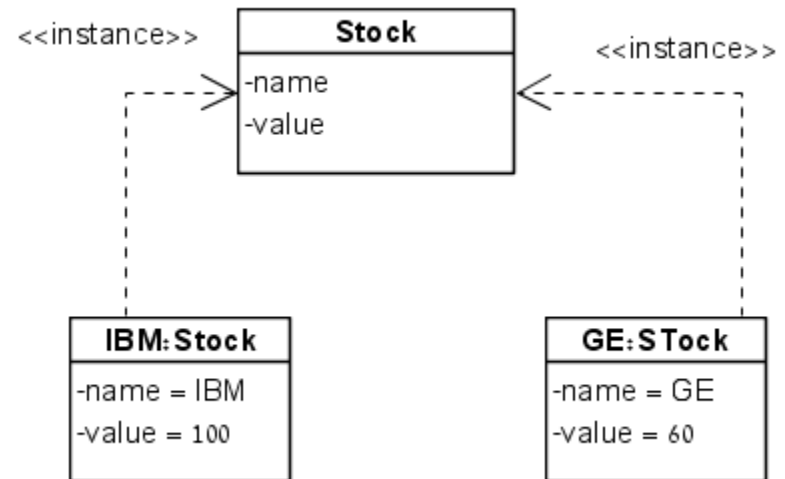
# *Instantiation*

- **Instantiation:** Involves an object and a class. Mixture of class diagram and instance diagram elements.

*Generalization*



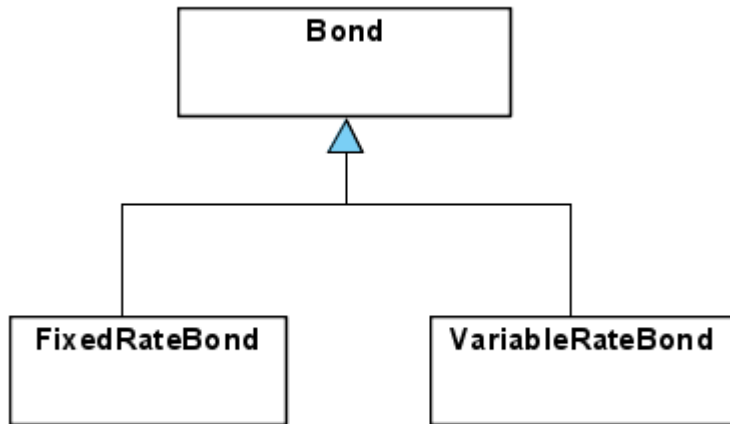
*Instantiation*



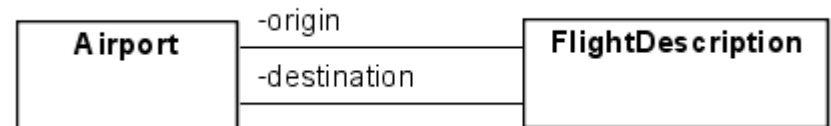
# Association

- **Association:** Describes **relationships (links)** among instances. Adds information **beyond a class boundary**. Important during analysis.

*Generalization*

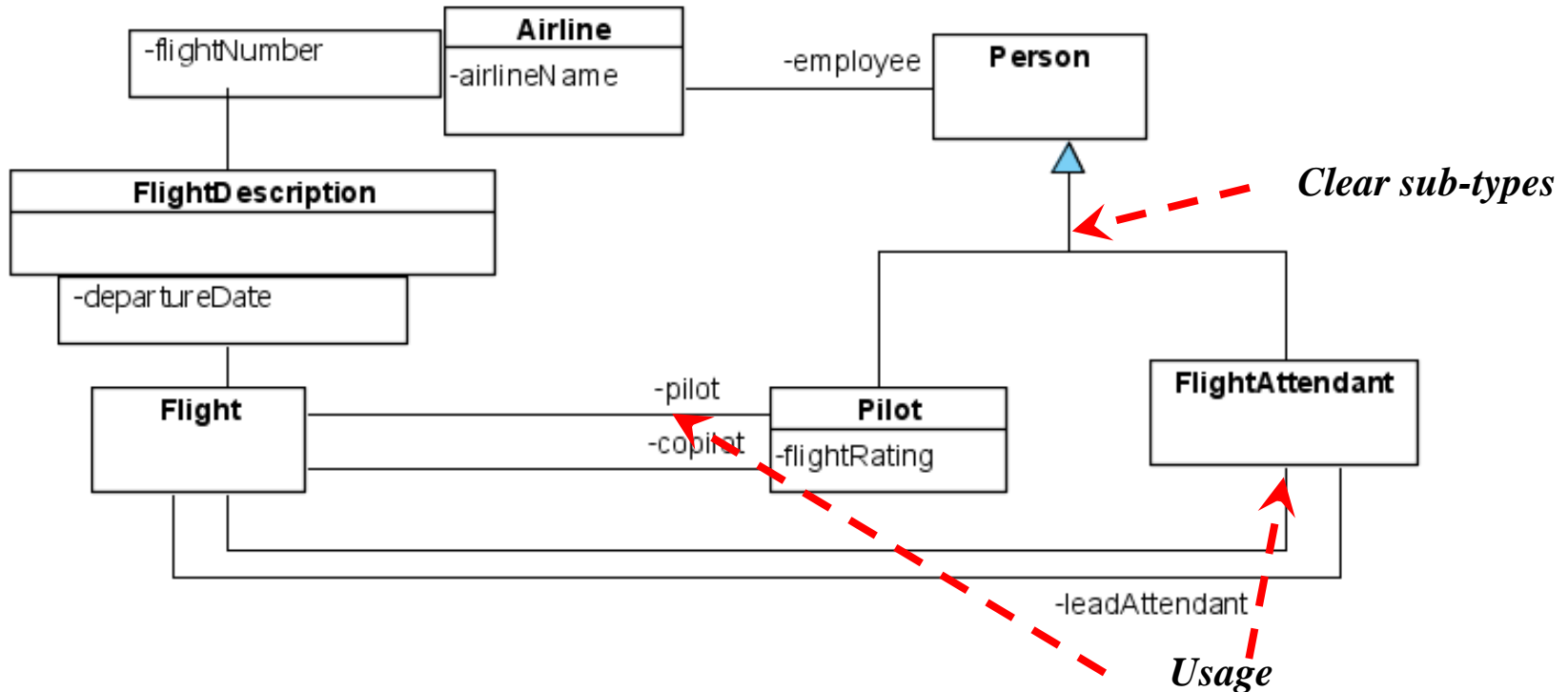


*Association*



# Association vs. Sub-typing

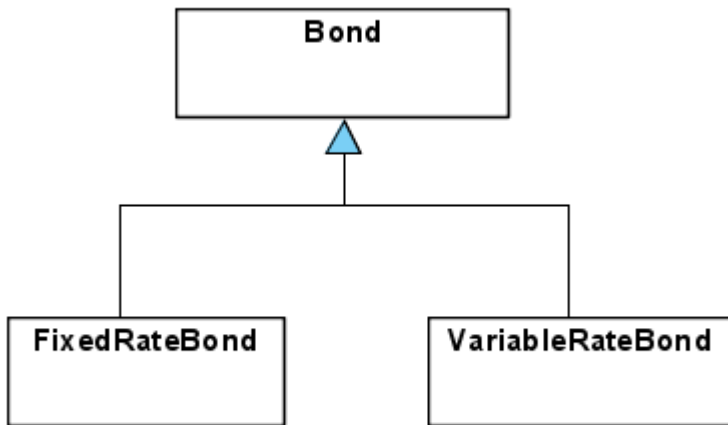
- **Association:** Do not confuse subclasses with roles:  
*Subclasses – Specialization.*  
*Roles – Usage.*
- Use **generalization** only when there are attributes or associations to distinguish: **there are clear sub-types!**



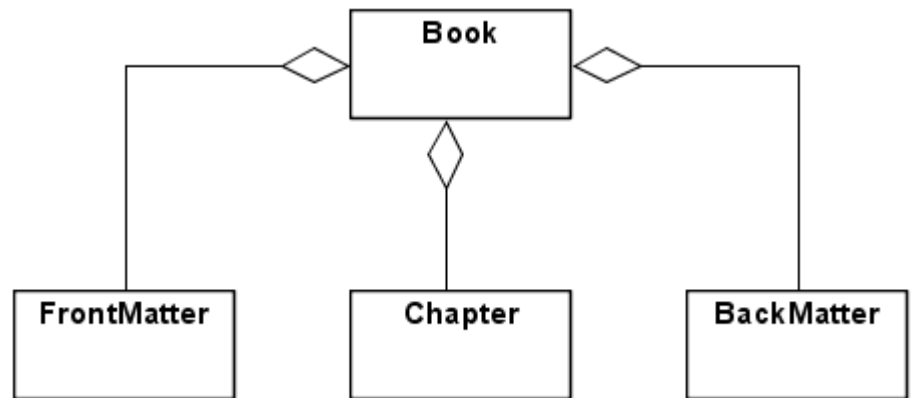
# Aggregation

- **Aggregation:** A special kind of association between a **whole** and a **part**.
- An **AND** grouping – relates different objects that together compose an **assembly**.

*Generalization*



*Aggregation*



# *10. Attribute Characterization*

# *Class level attributes and operations - 1*

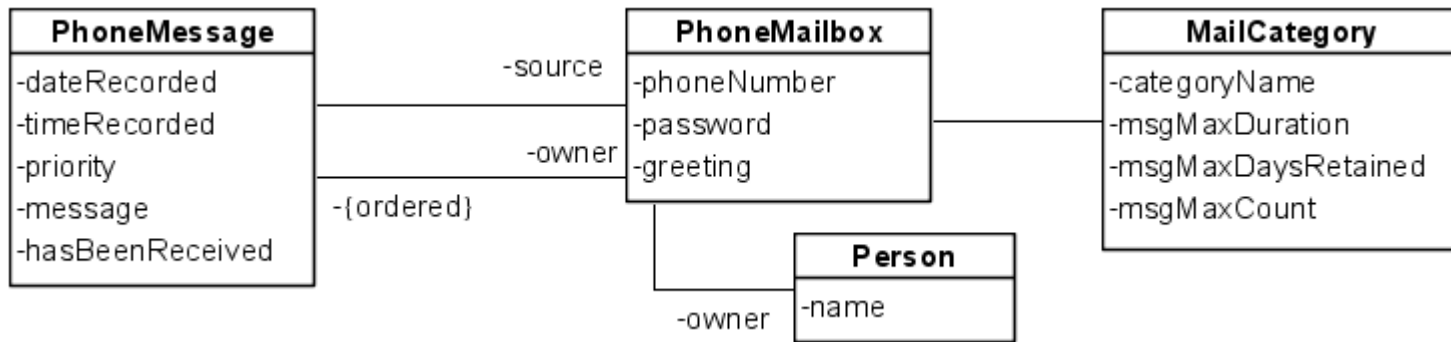
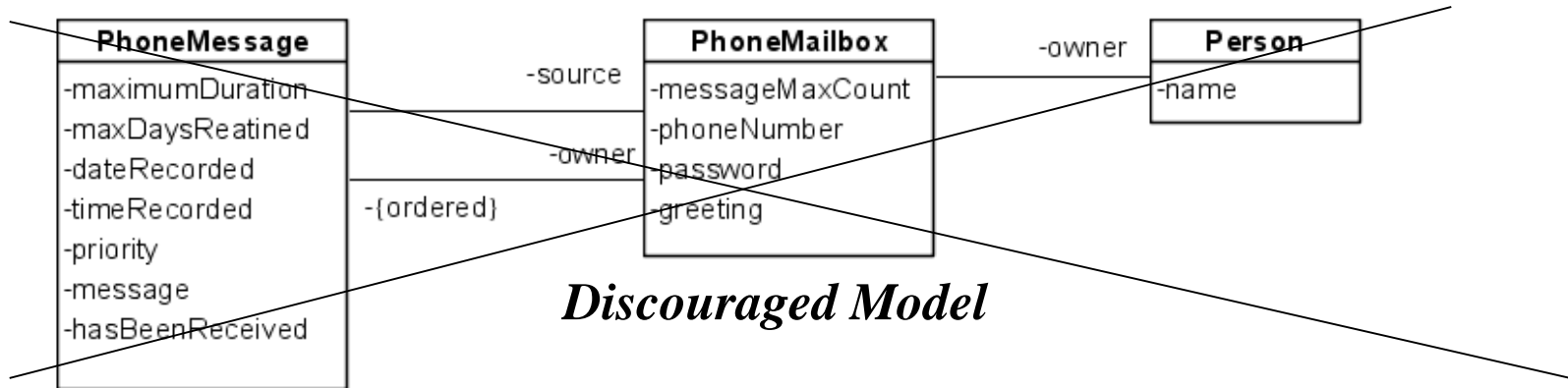
---

- A **class attribute/operation** refers to the class itself as an object.
- **It can take one of two roles:**
  1. Characterize the class as an object:
    - Average age of employees.
    - Update/management operations.
  2. Can stand for a property whose value is common to all instances of the class:
    - Interest rate of all accounts of a common type.
    - The location of BGU buildings.
    - The origin country of all French cars.

# *Class level attributes and operations - 2*

- Class attributes/operations are not recommended!

A better modeling: Add a new class of which the older class is an instance.

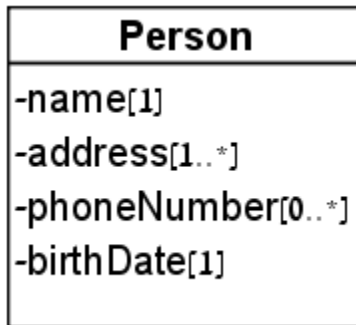


*Preferred Model*

# *Attribute Values – 1: Multiplicity*

---

- **Attribute multiplicity** specifies the possible number of values for an attribute of an object. This specification is relevant mainly for database purposes (persistent classes).





## *Attribute Values – 2: Key Constraints*

---

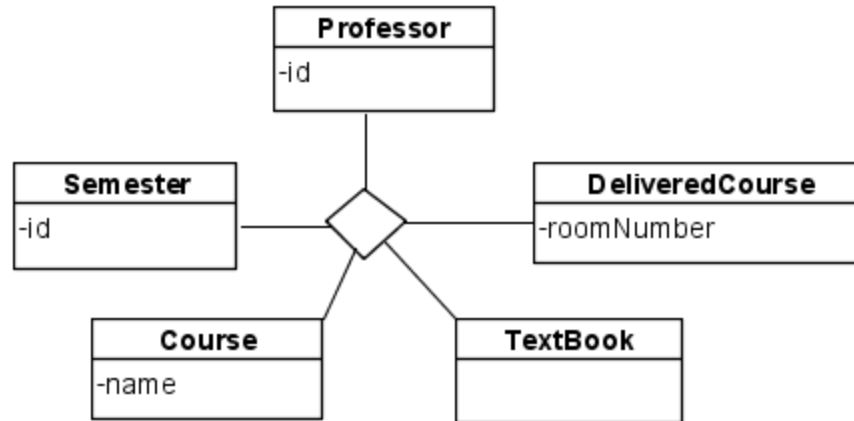
- A **Key Constraint** on a class (**candidate key**) is a **minimal combination** of attributes whose values uniquely identify objects of a class.  
This specification is relevant mainly for database purposes (persistent classes).

Airport
-airportCode {CK1}
-airportName {CK2}

# *Attribute Values – 3: Key Constraints*

---

A **Key Constraint on an association** is a minimal combination of roles and qualifiers that uniquely identify links of the association.

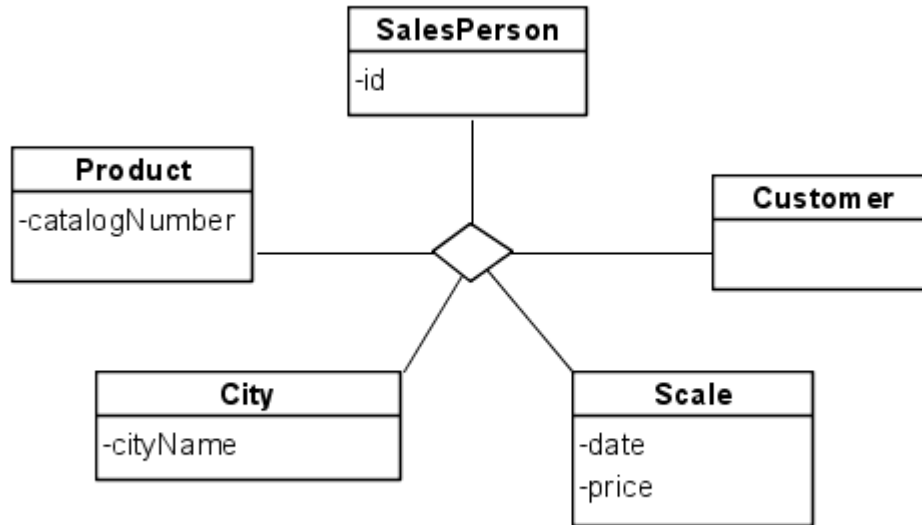


{ CK =(Semester.id, Professor.id, Course.name) }

“A course with a given name is given by at most a single professor, in a given semester”.

# *Attribute Values – 4: Key Constraints*

---



CK = (Product.catalogNo, City.name) }

“A product is sold by at most a single sales person, in a given city”.

Key constraints take the role of multiplicity constraints for non-binary associations.

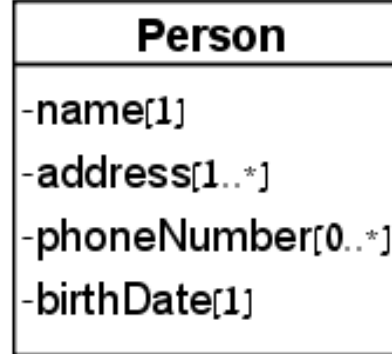
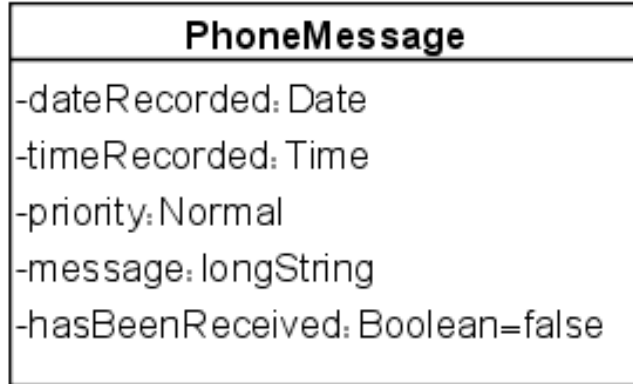
# *Attribute Values – 5: Domains*

---

- A **domain** (*Type*) is a **named set** of possible values for an attribute.
  - It has associated operations.
  - Domain values do not participate in associations.
- A domain can be described in different ways:
  - **Intentionally:**  
 $ODD = \{ n / n \text{ in } NAT, n/2 \text{ not in } NAT \}$
  - **Extensionally (enumeration domain):**  
 $PriorityType = \{ NORMAL, URGENT, INFORMATIONAL \}$
  - **Structured:**  
 $\{ PhoneNumber \}$  – a set of phone numbers.  
 $Date = \langle year: YEAR, month: MONTH, day: DAY \rangle$

# *Attribute Values – 6: Domains*

---



- **An attribute can be followed by:**
  - Key indication.
  - multiplicity indication.
  - domain indication.
  - default value.

# Derived Attributes

---

- **Derived data** is information that can be calculated from other elements in a diagram:
- For a *flightDescription*:  
$$\text{scheduledArrivalTime} = \text{scheduledDepartureTime} + \text{scheduledDuration}.$$
- **Visual notation:** a preceding slash:  
*/scheduledArrivalTime*

FlightDescription
-scheduledDepartureTime
-scheduledDuration / scheduledArrival
-frequency
-startEffectiveDate
-stopEffectiveDate

# Derived Attributes

Derived associations are also denoted using a preceding slash:

