CHRISTIAN HUEMER
MARION SCHOLZ

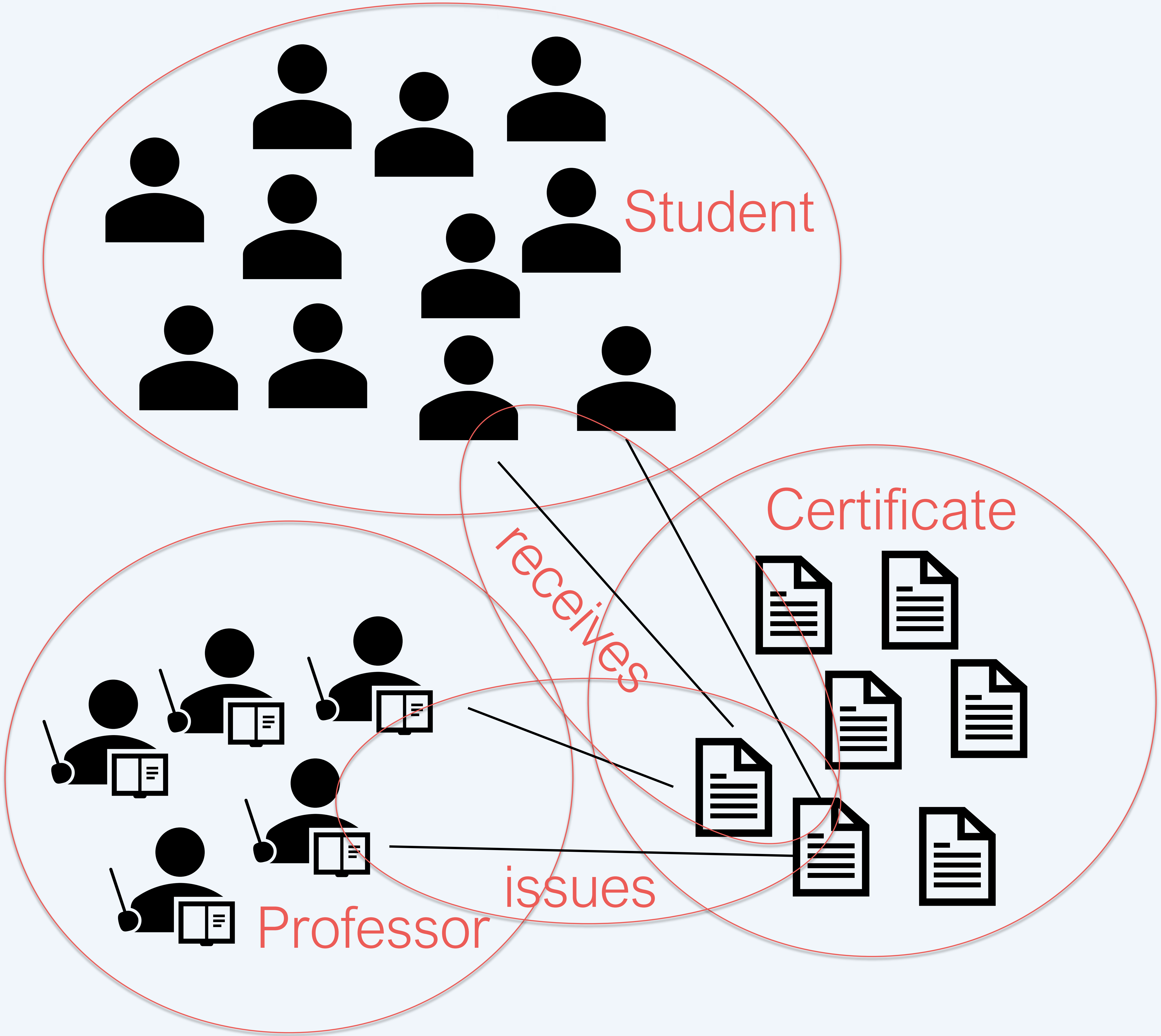
Object-Oriented Modeling with UML

Structural Modeling
From Object to Class

Christian Huemer und Marion Scholz

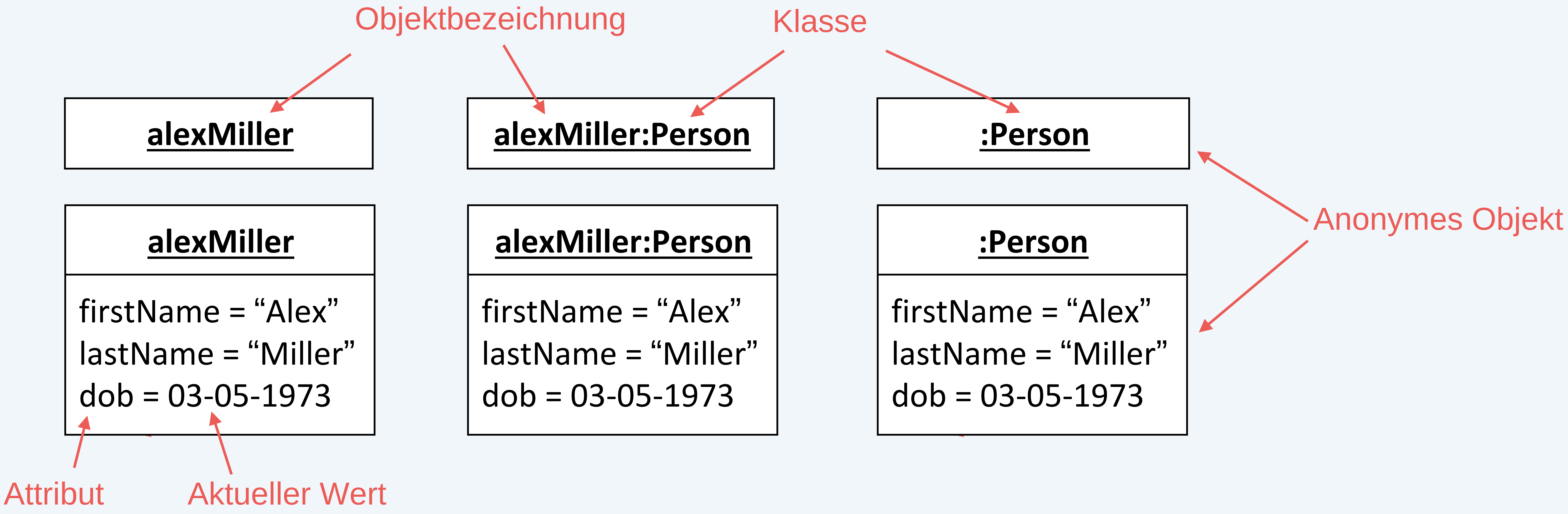
Presented by Nicholas Bzowski

Objects/Links and Classes/Associations



Objects

- Instances of a system
- Notation variants:



Objektbezeichnung

Klasse

alexMiller

alexMiller:Person

:Person

Anonymes Objekt

alexMiller

firstName = “Alex”
lastName = “Miller”
dob = 03-05-1973

alexMiller:Person

firstName = “Alex”
lastName = “Miller”
dob = 03-05-1973

:Person

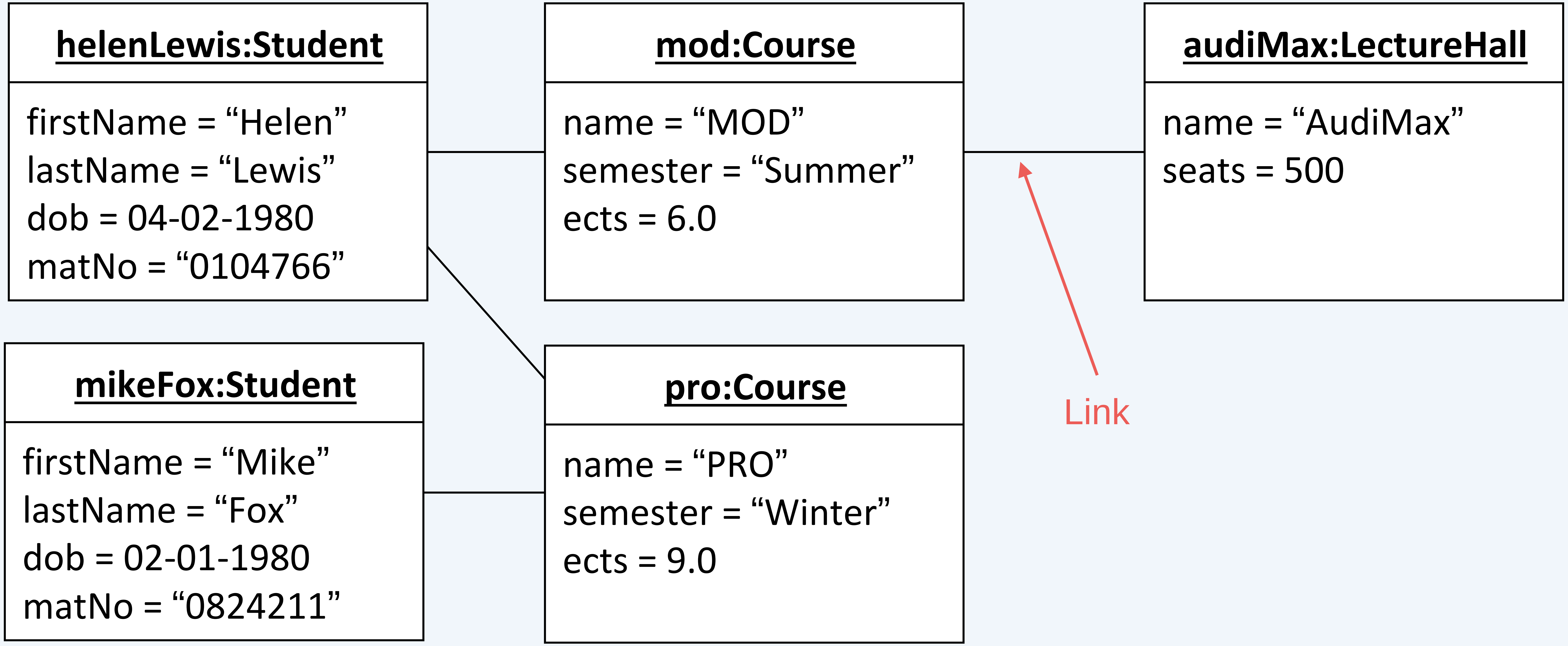
firstName = “Alex”
lastName = “Miller”
dob = 03-05-1973

Attribut

Aktueller Wert

The Object Diagram

- Describes the structural aspect of a system at the instance level
- Snapshot of the system
- Does not have to be complete



helenLewis:Student

firstName = “Helen”
lastName = “Lewis”
dob = 04-02-1980
matNo = “0104766”

mod:Course

name = “MOD”
semester = “Summer”
ects = 6.0

audiMax:LectureHall

name = “AudiMax”
seats = 500

mikeFox:Student

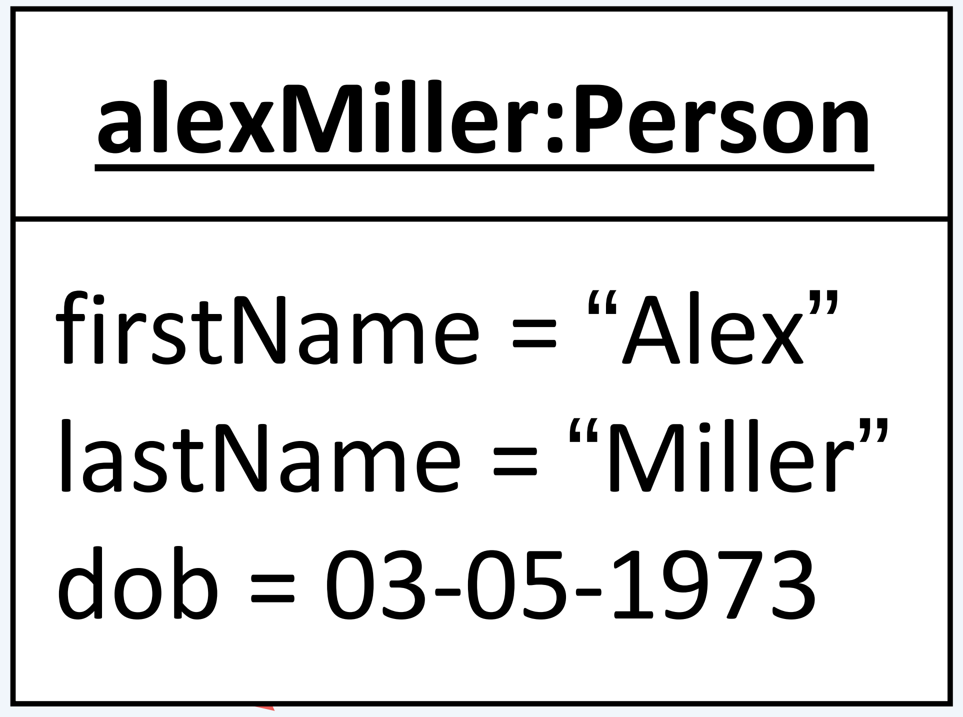
firstName = “Mike”
lastName = “Fox”
dob = 02-01-1980
matNo = “0824211”

pro:Course

name = “PRO”
semester = “Winter”
ects = 9.0

Link

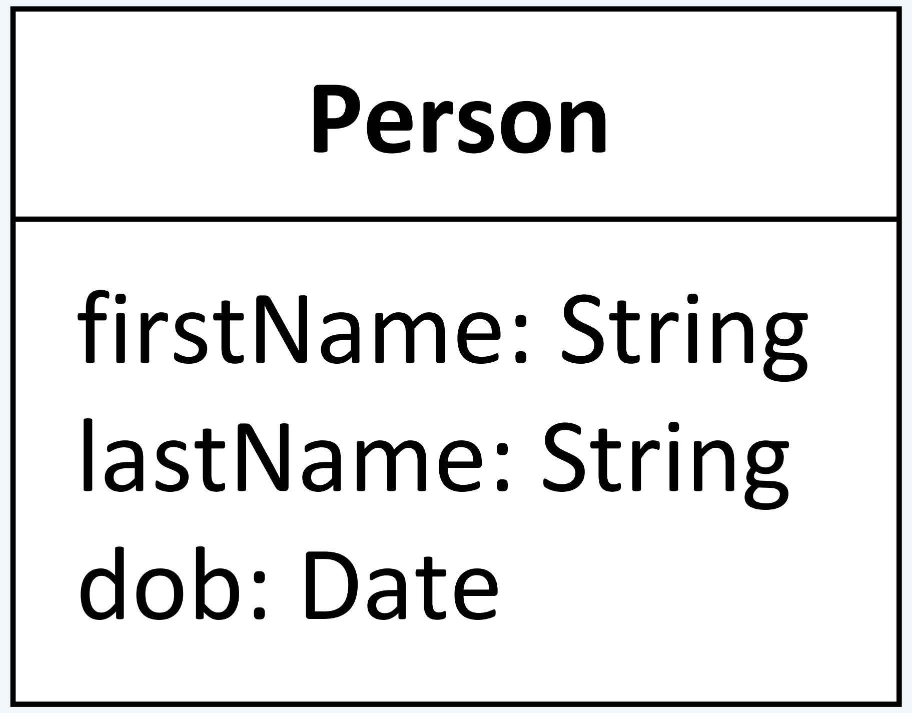
From Object to Class



alexMiller:Person

firstName = “Alex”
lastName = “Miller”
dob = 03-05-1973

Instance of class



Person

firstName: String
lastName: String
dob: Date

Class

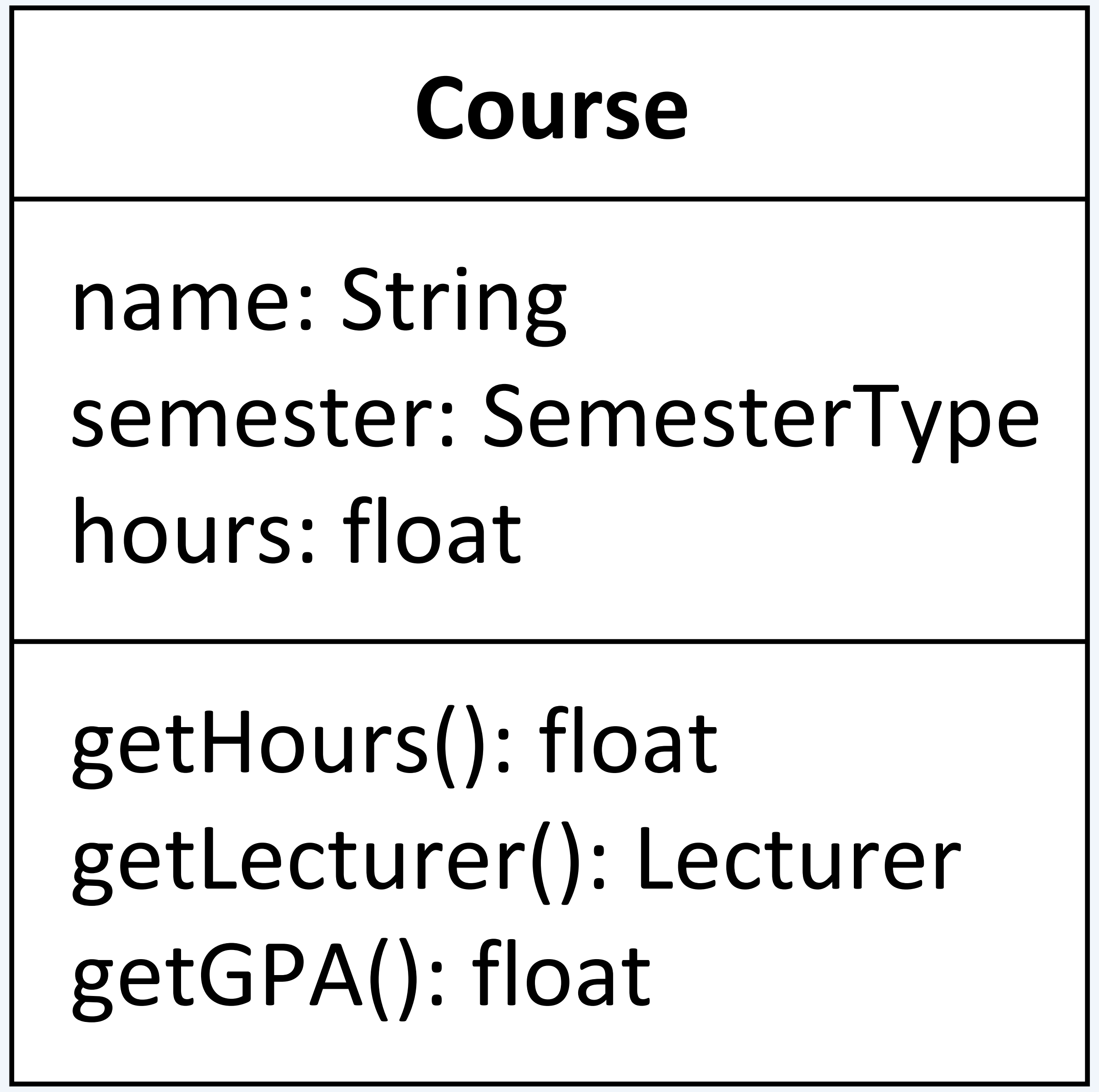
- Instances of a system often have the same characteristic features and the same behavior
- Class: blueprint for a set of similar objects of a system
- Objects: Instances of classes
- Attributes: structural characteristics of a class
  - Different value for each instance (= object)
- Operations: Behavior of a class
  - Identical for all objects of a classs
    ⇒ Not shown in the object diagram

Structural Modeling
The Class

Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Notation for Classes



Course

name: String
semester: SemesterType
hours: float

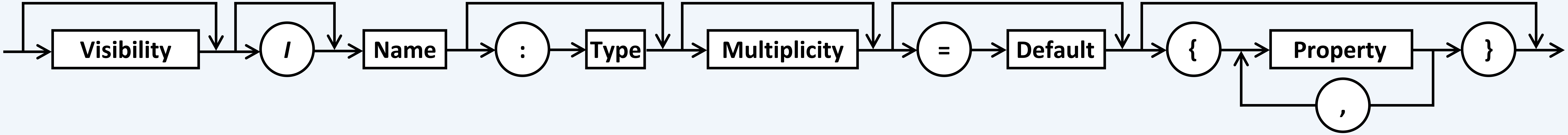
getHours(): float
getLecturer(): Lecturer
getGPA(): float

Class name

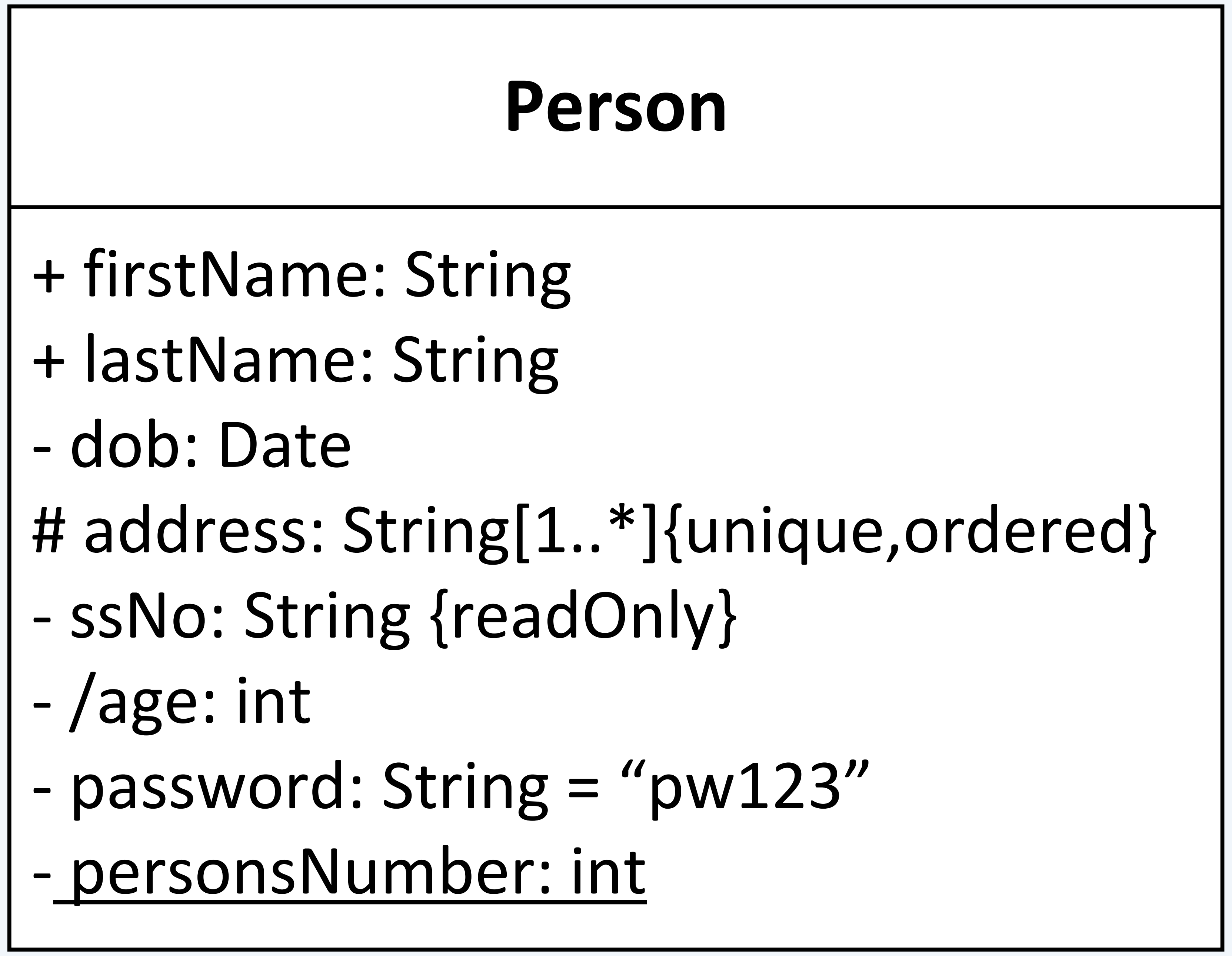
Attributes

Operations

Syntax of the attribute specification



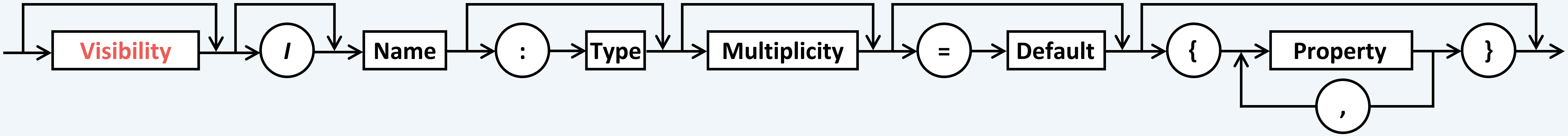
Visibility / Name : Type Multiplicity = Default { Property , }



Person

+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

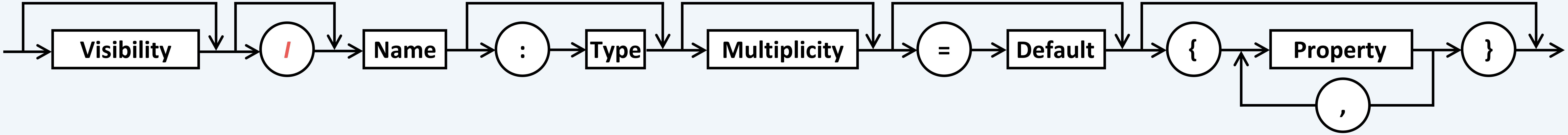
Attribute Syntax - Visibility

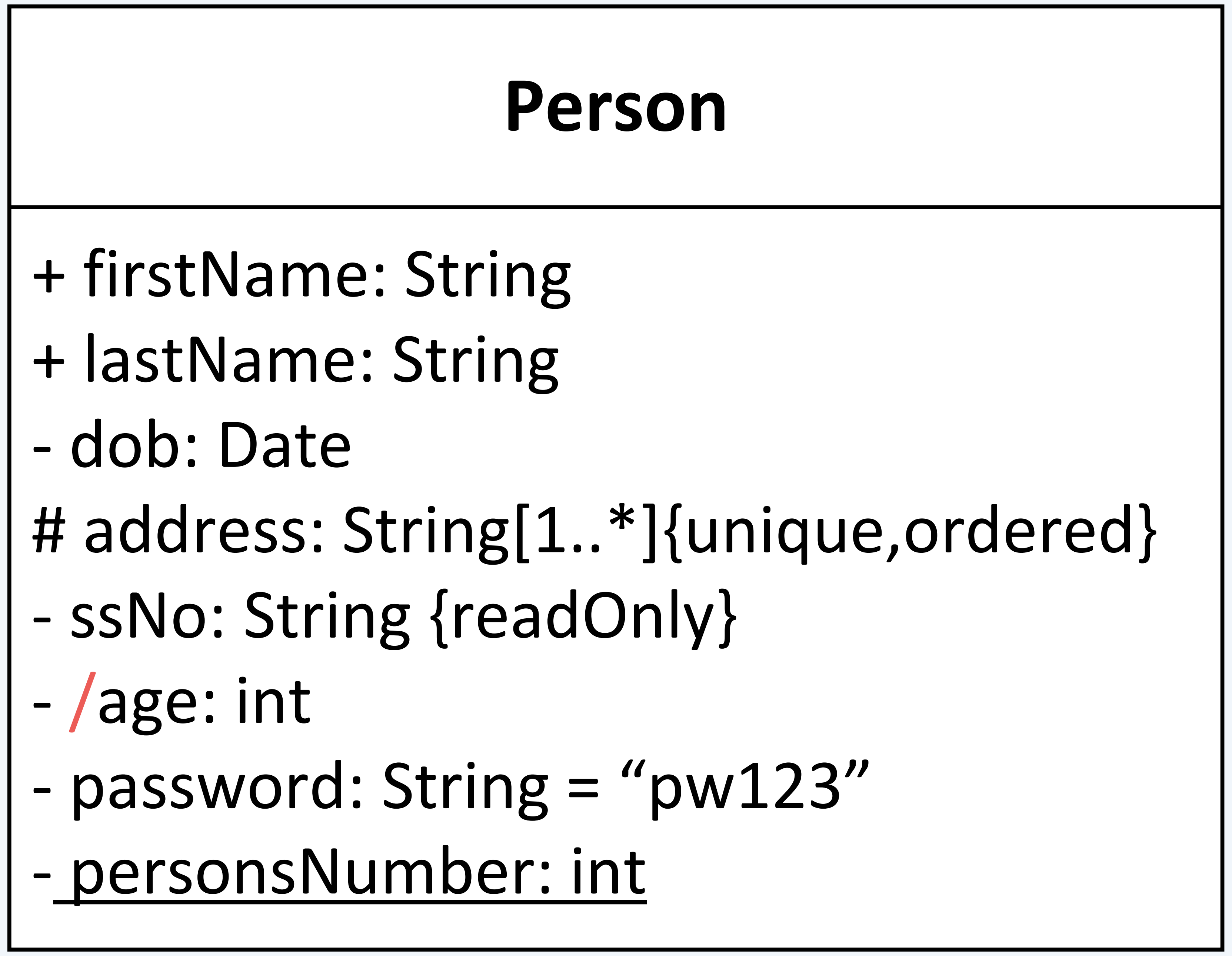


Person
+ firstName: String
+ lastName: String
- dob: Date
address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Who is allowed to access the attribute
  - + ... public
  - - ... private
  - # ... protected
  - ~ ... package

Attribute syntax – Derived attribute



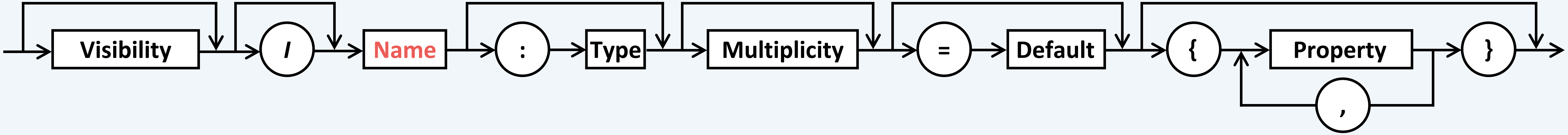


Person

+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Value of the attribute is derived from other attributes
  - **age**: derived from birthdate (**dob**)

Attribute syntax - Name

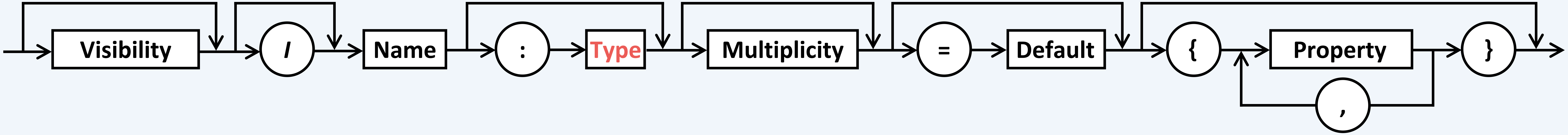


Person

+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Name of the attribute

Attribute syntax - Type



Visibility

/

Name

:

Type

Multiplicity

=

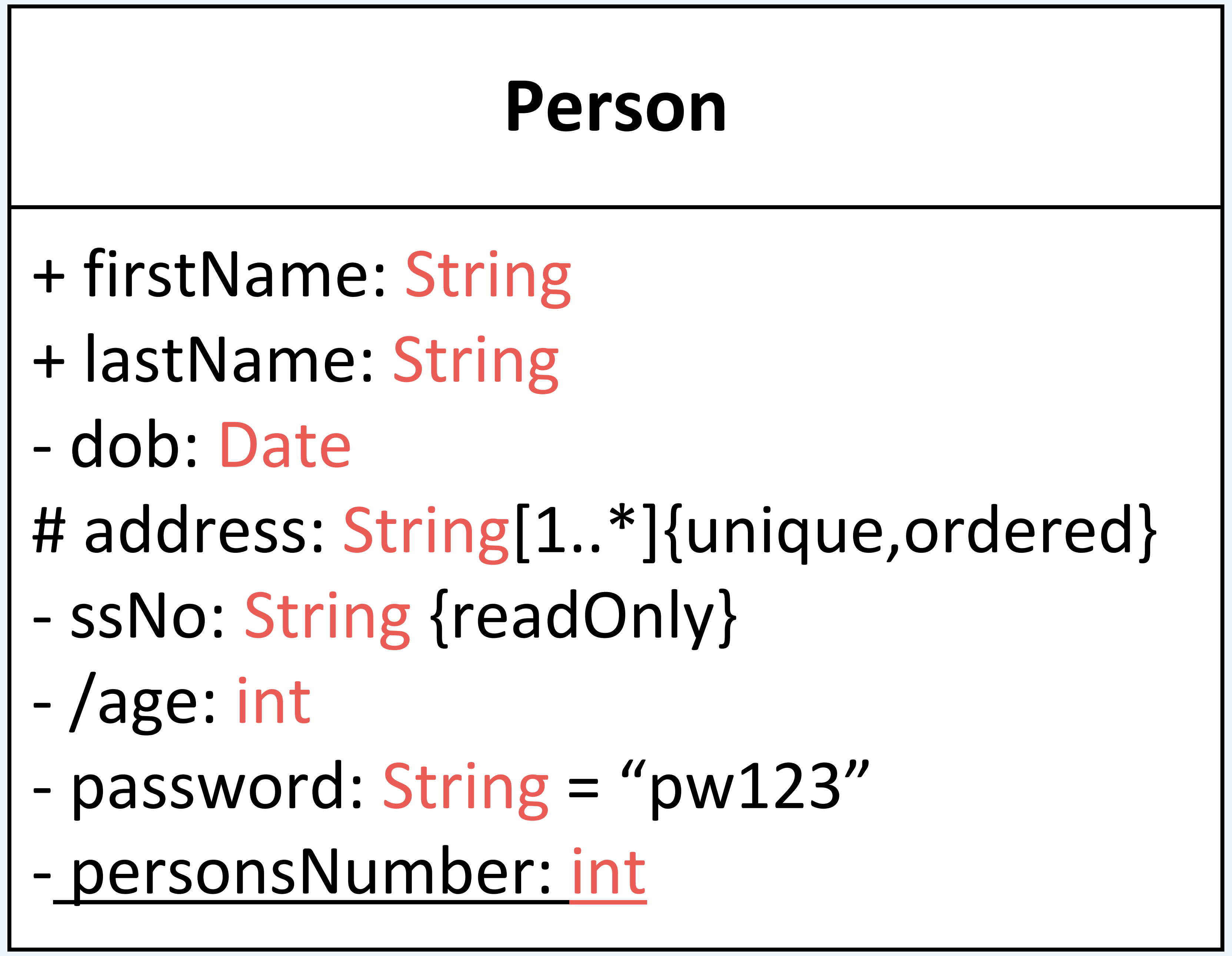
Default

{

Property

,

}



Person

+ firstName: String

+ lastName: String

- dob: Date

# address: String[1..*]{unique,ordered}

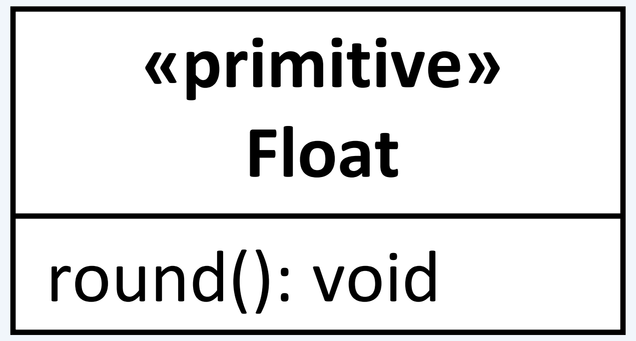
- ssNo: String {readOnly}

- /age: int

- password: String = “pw123”

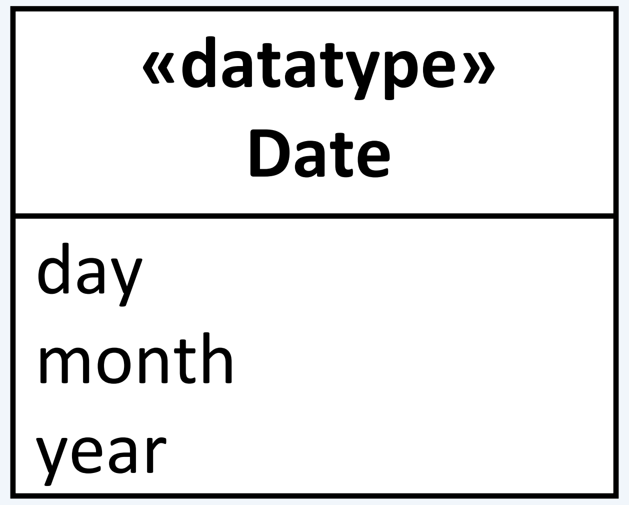
- personsNumber: int

- Class
- Datatype
  - Primitive datatype
    - Predefined : **`Boolean`**, **`Integer`**, **`UnlimitedNatural`**, **`String`**
    - User-defined: **`«primitive»`**
    - Composite datatypes: **`«datatype»`**
  - Enumerations: **`«enumeration»`**



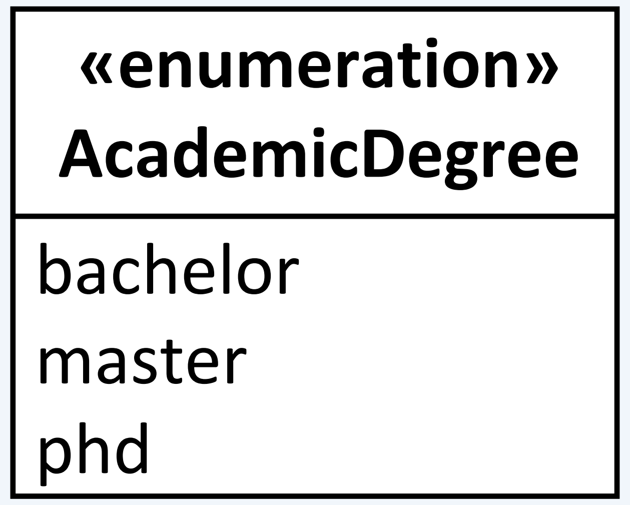
«primitive»
Float

round(): void



«datatype»
Date

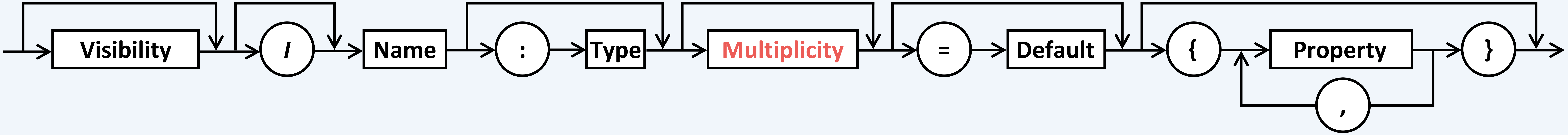
day
month
year

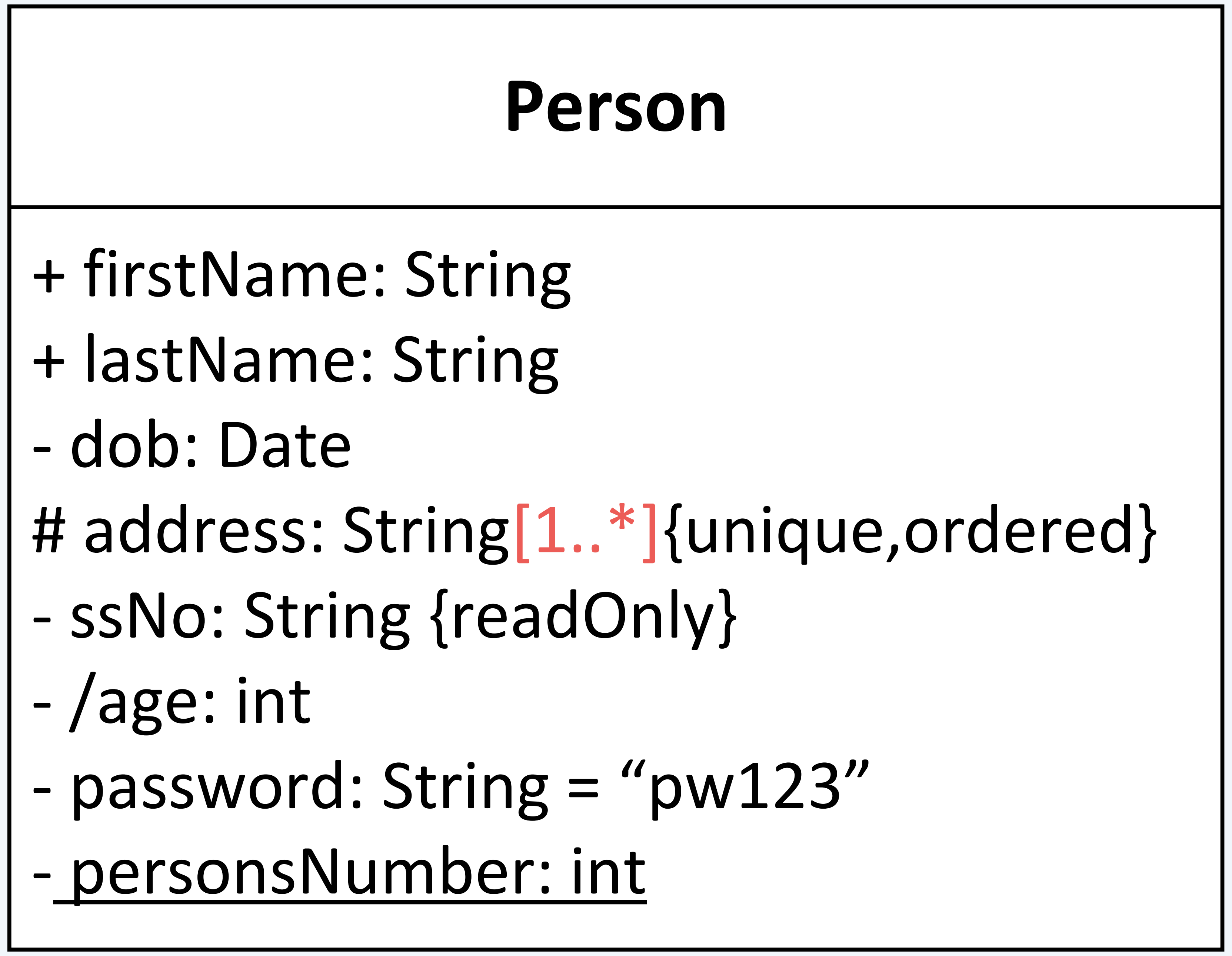


«enumeration»
AcademicDegree

bachelor
master
phd

Attribute syntax - Multiplicity

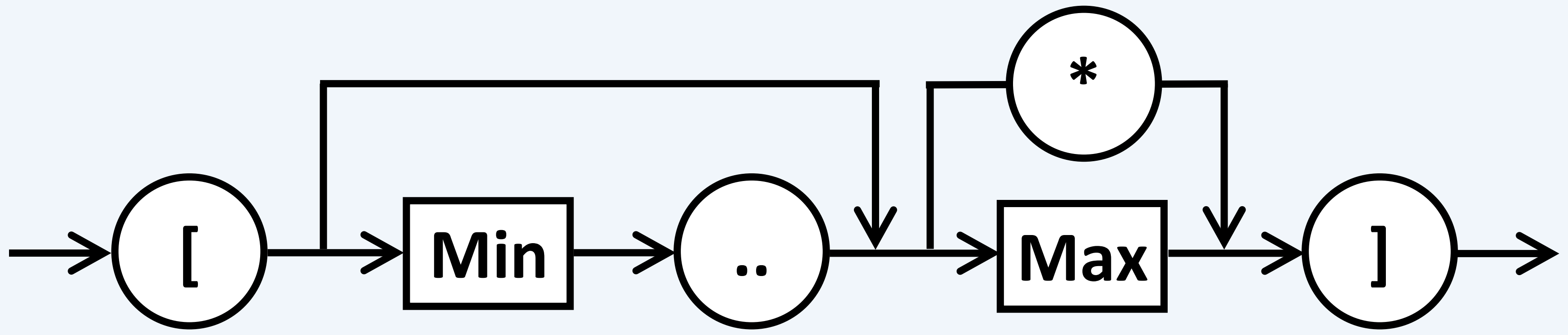




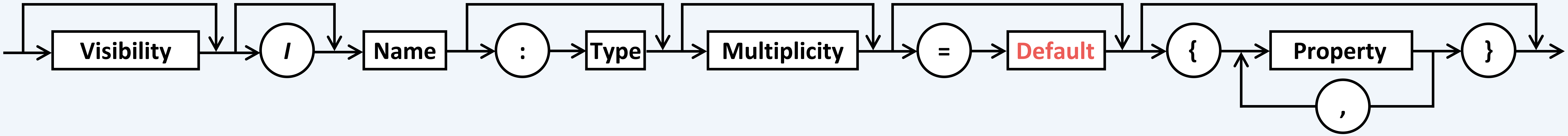
Person

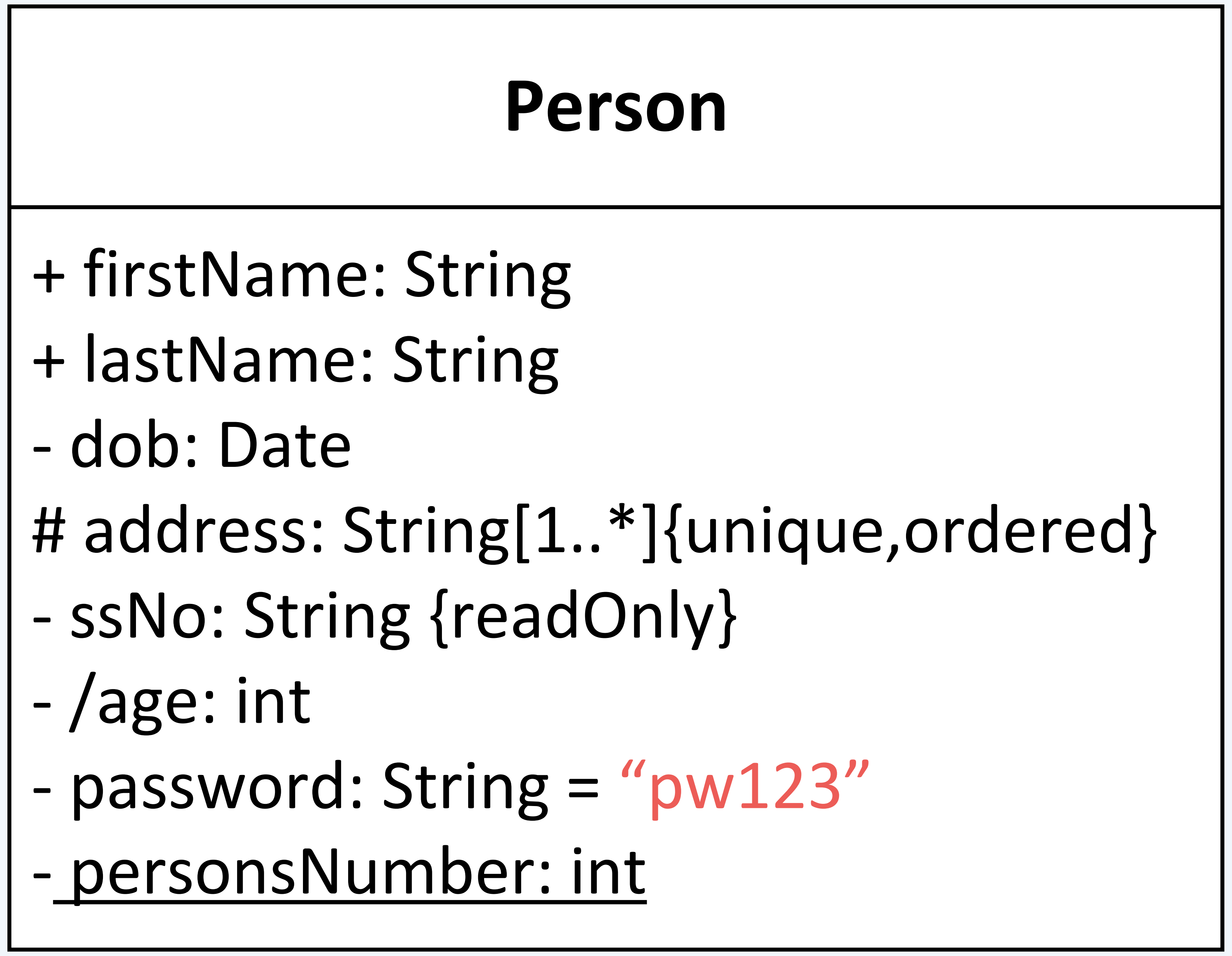
+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Number of values an attribute can contain
- Default: 1
- Notation: **[min..max]**
  - No upper limit: `[*]` oder `[0..*]`



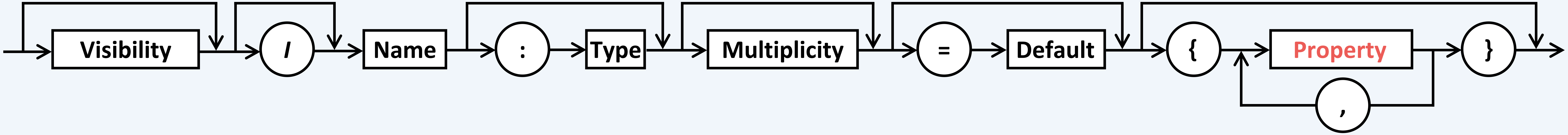
Attribute syntax – Default value



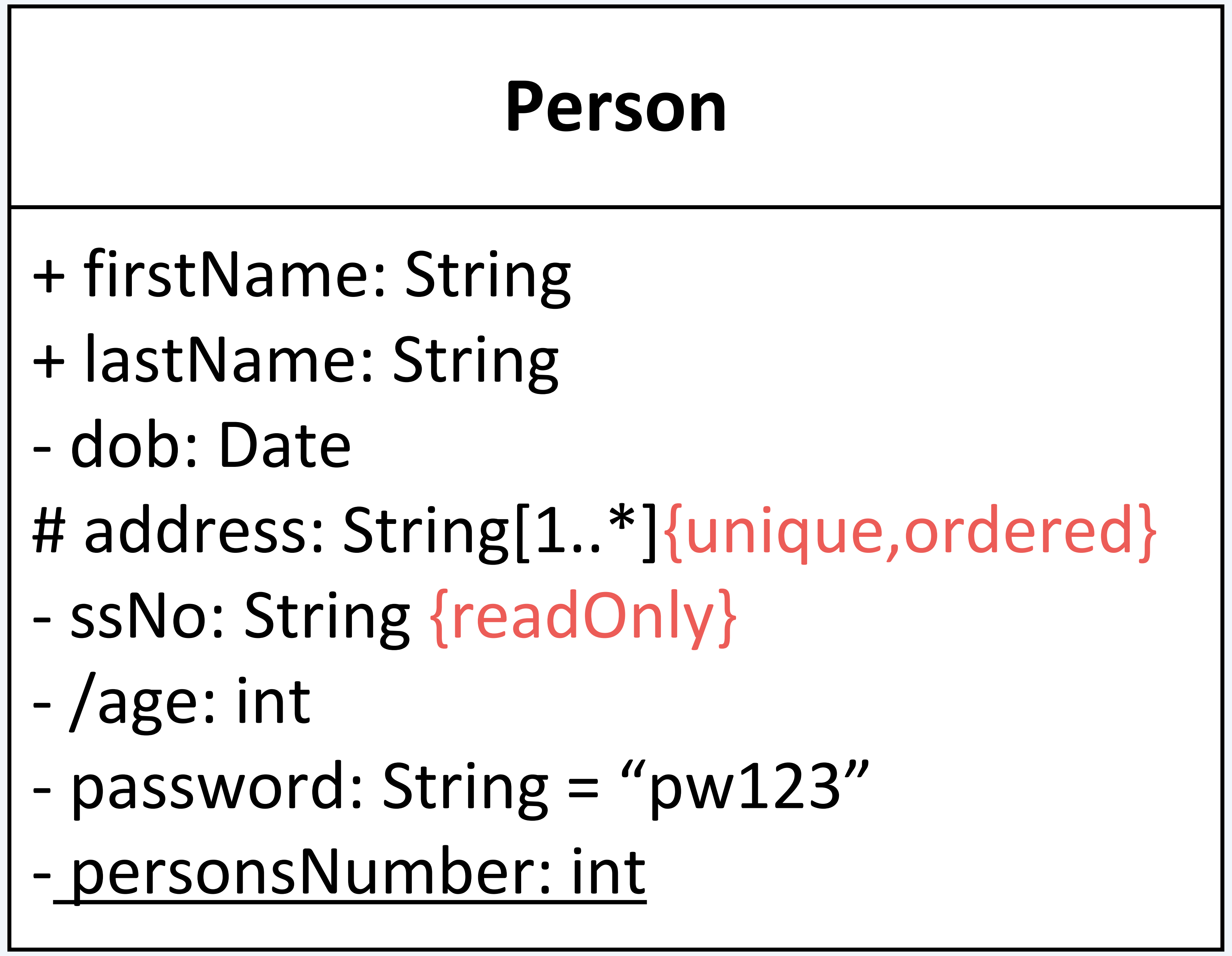

Person
+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Default value
- Used if the value of the attribute is not explicitly set

Attribute syntax - Properties



Visibility / Name : Type Multiplicity = Default { Property , }

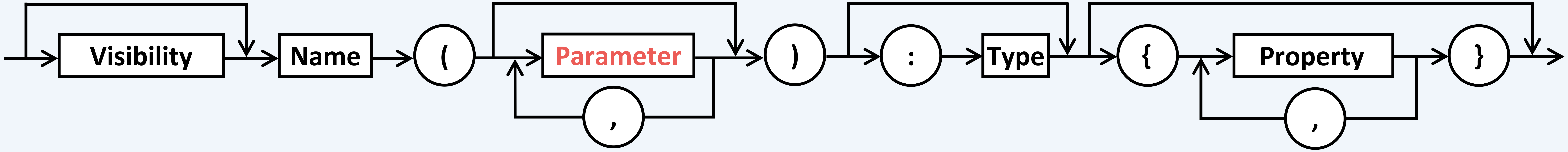


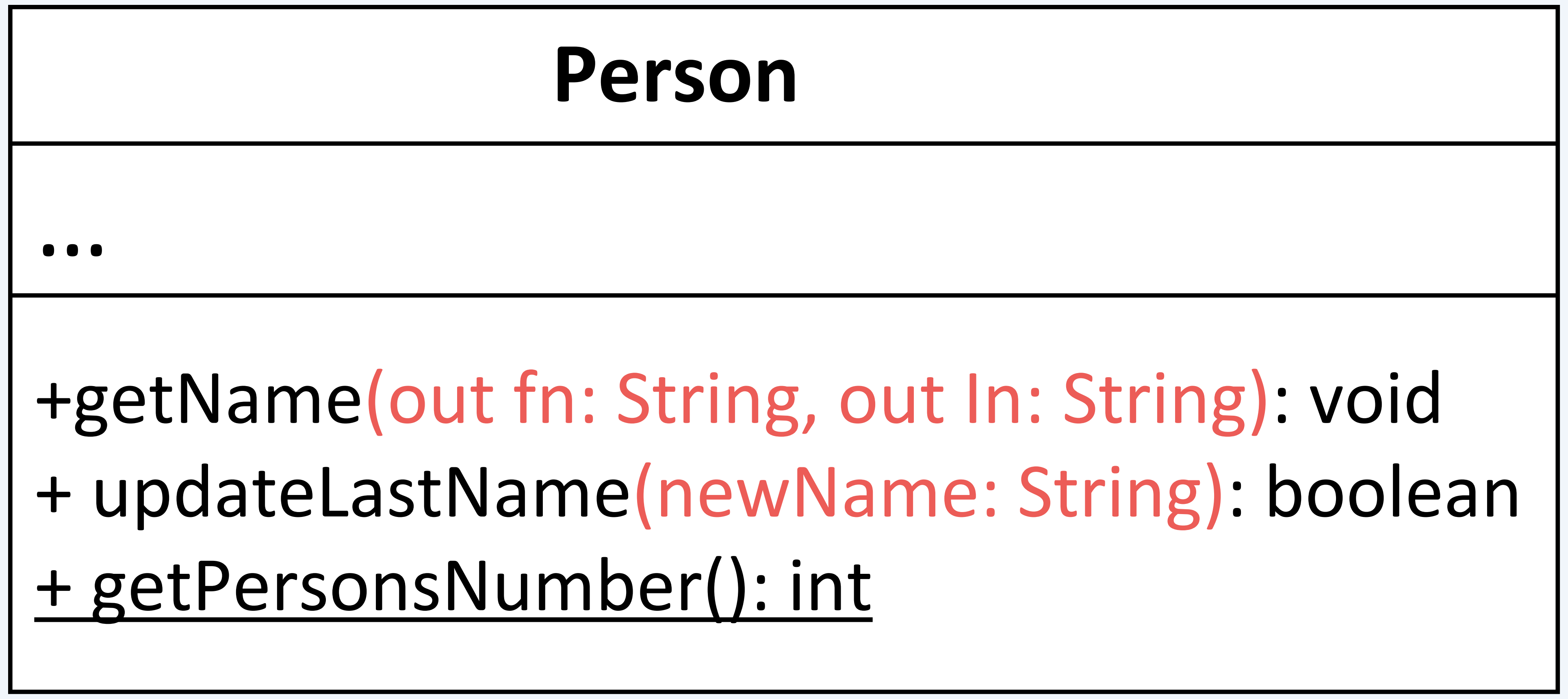
Person

+ firstName: String
+ lastName: String
- dob: Date
# address: String[1..*]{unique,ordered}
- ssNo: String {readOnly}
- /age: int
- password: String = “pw123”
- personsNumber: int

- Predefined properties
  - **`{readOnly}`**
  - **`{unique}`**, **`{non-unique}`**
  - **`{ordered}`**, **`{unordered}`**
- Possible Combinations
  - Set: **`{unordered, unique}`**
  - Multi-set: **`{unordered, non-unique}`**
  - Ordered set: **`{ordered, unique}`**
  - List: **`{ordered, non-unique}`**

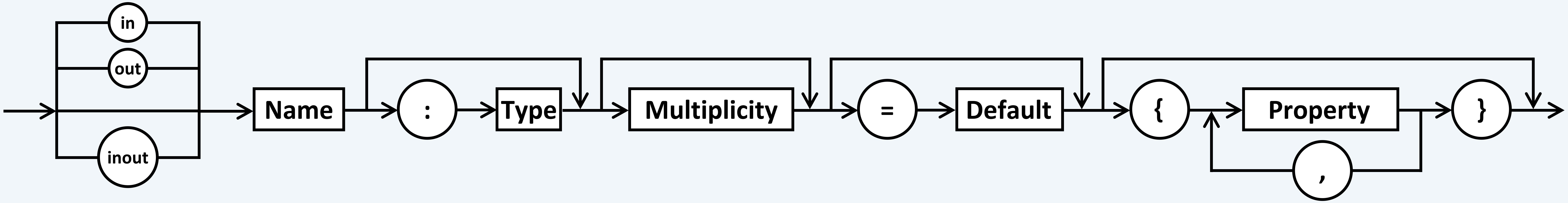
Operation syntax - Parameter



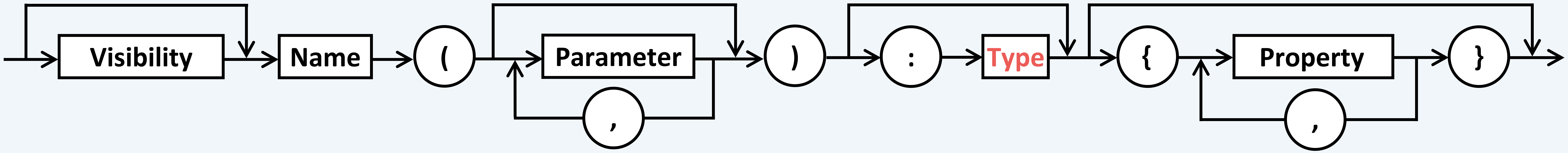


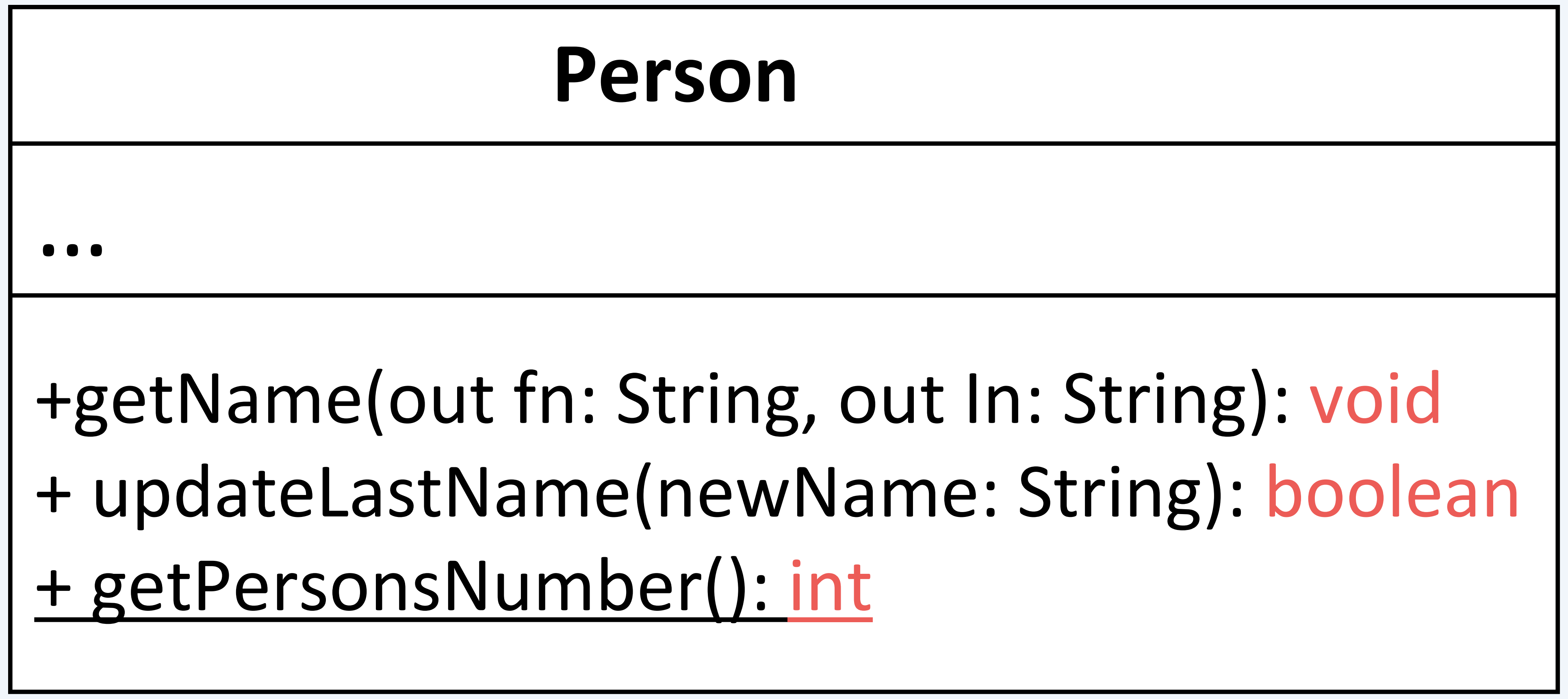
+getName(out fn: String, out In: String): void
+ updateLastName(newName: String): boolean
+ getPersonsNumber(): int

- Notation similar to attribute
- Direction of the parameter
  - **in** … Input parameter
  - **out** … Output parameter
  - **inout** … Combined input/output parameter



Operation syntax - Type





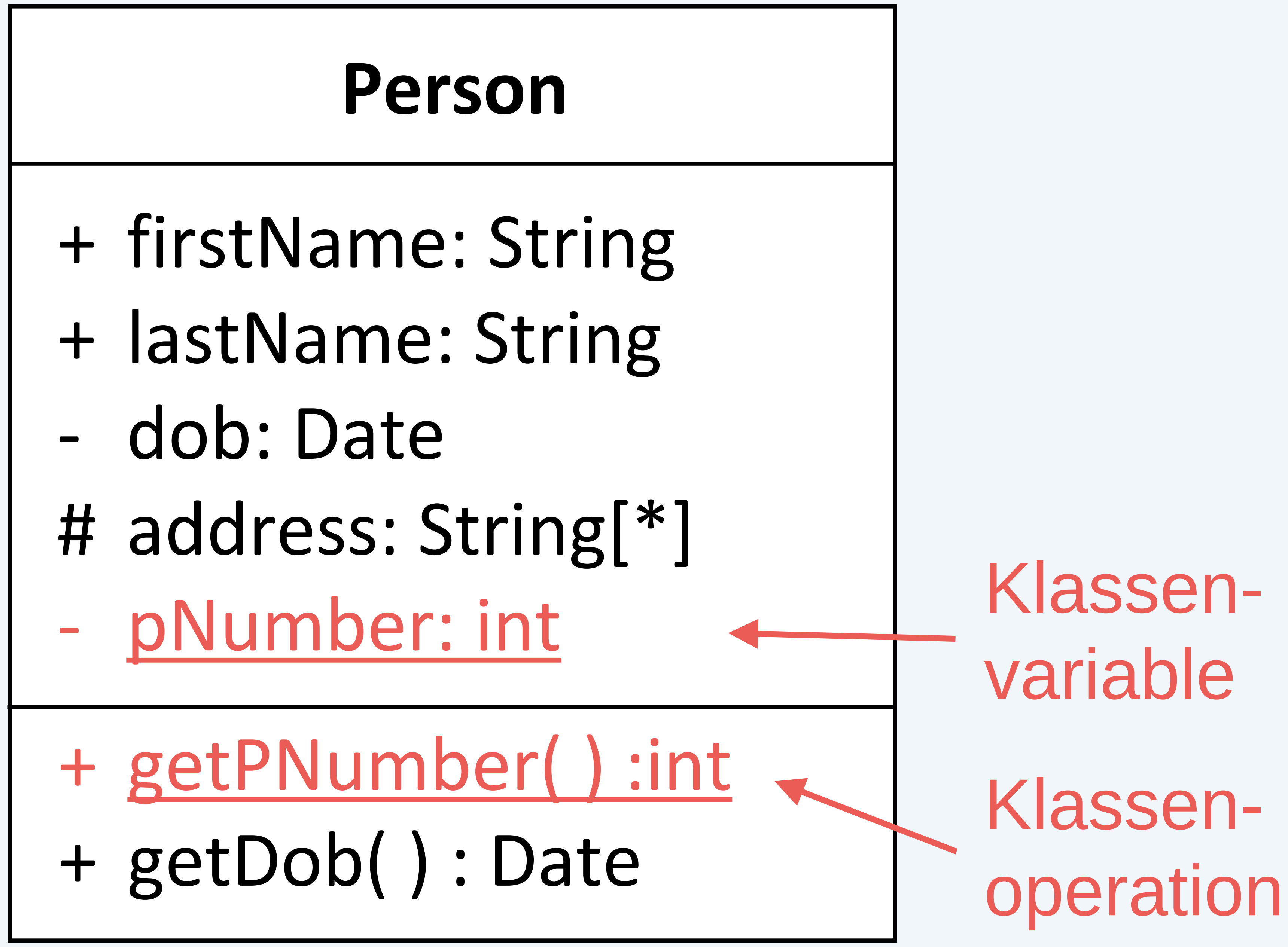
...

+getName(out fn: String, out In: String): void
+ updateLastName(newName: String): boolean
+ getPersonsNumber(): int

- Return value type

Class variables and class operations

- Instance variable (= instance attribute)
- Class variable (= class attribute, static attribute)
  - Only set up once per class
  - E.g. counters for the instances of a class, constants, etc.
- Class operation (= static operation)
  - Can be used even if no instance of the class containing them has been created
  - E.g. constructors, mathematical functions, etc.
  - Notation: Underlined



Person

+ firstName: String
+ lastName: String
- dob: Date
# address: String[*]
- pNumber: int

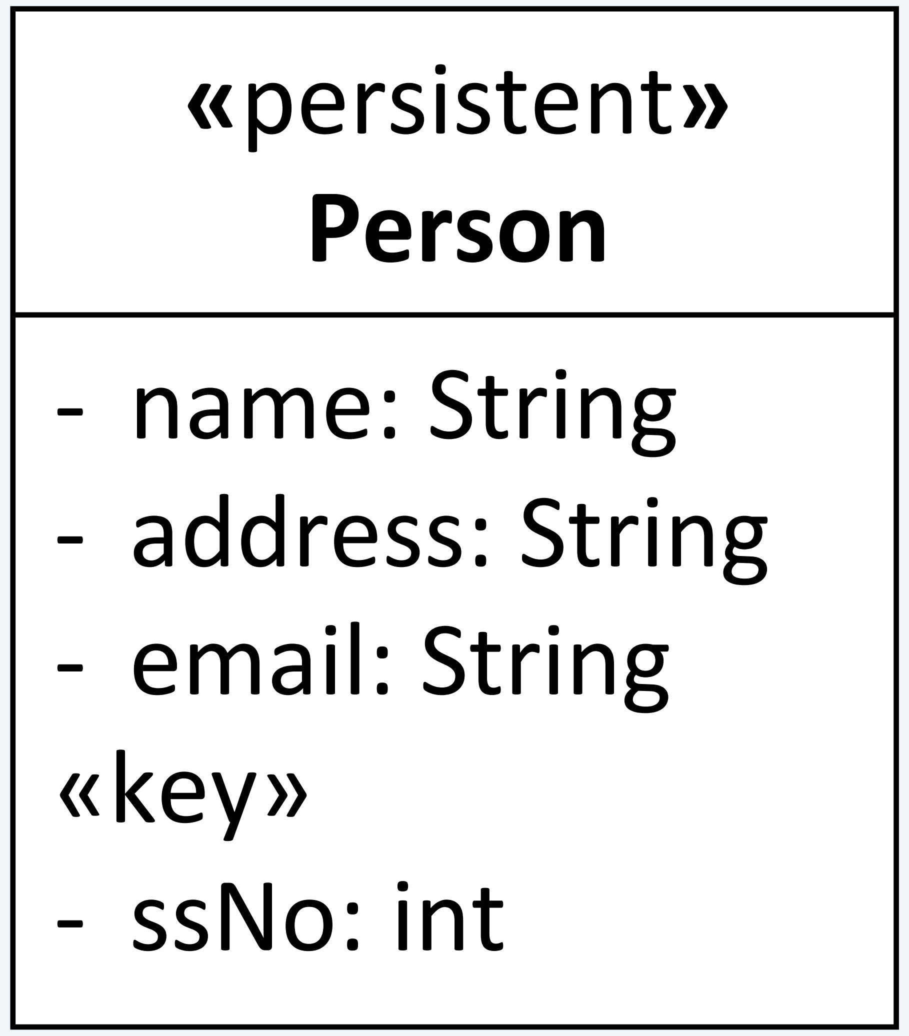
+ getPNumber( ) :int
+ getDob( ) : Date

Klassen-variable

Klassen-operation

```
class Person  {
  public String firstName;
  public String lastName;
  private Date dob;
  protected String[] address;
  private static int pNumber;
  public static int getPNumber() {…}
  public Date getDob() {…}
}
```

Extension of UML for data modeling



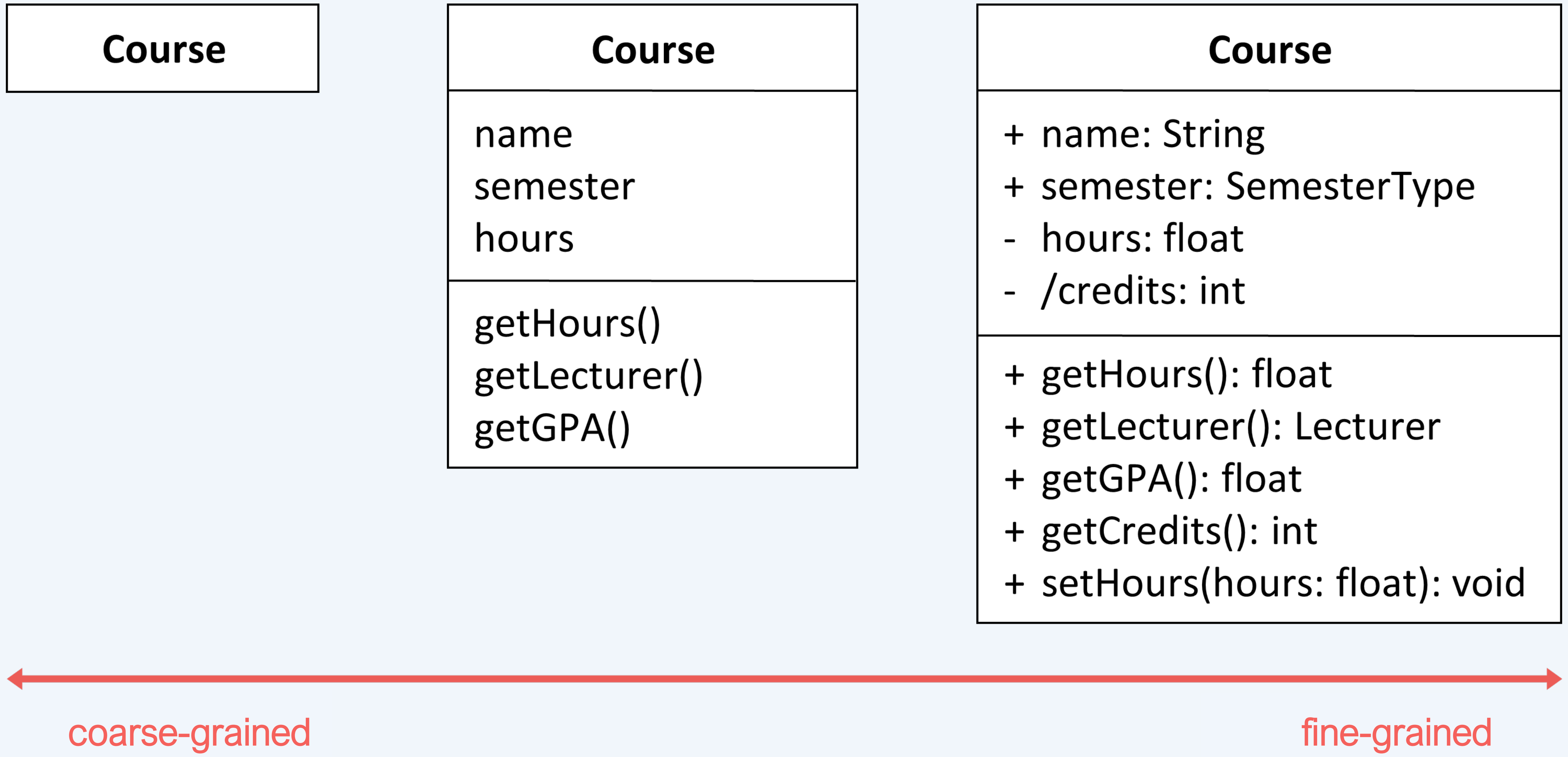
«persistent»
Person

- name: String
- address: String
- email: String

«key»

- ssNo: int

Specification of a class: Different level of detail



Course

Course

name
semester
hours

getHours()
getLecturer()
getGPA()

Course

+ name: String
+ semester: SemesterType
- hours: float
- /credits: int

+ getHours(): float
+ getLecturer(): Lecturer
+ getGPA(): float
+ getCredits(): int
+ setHours(hours: float): void

Structural Modeling

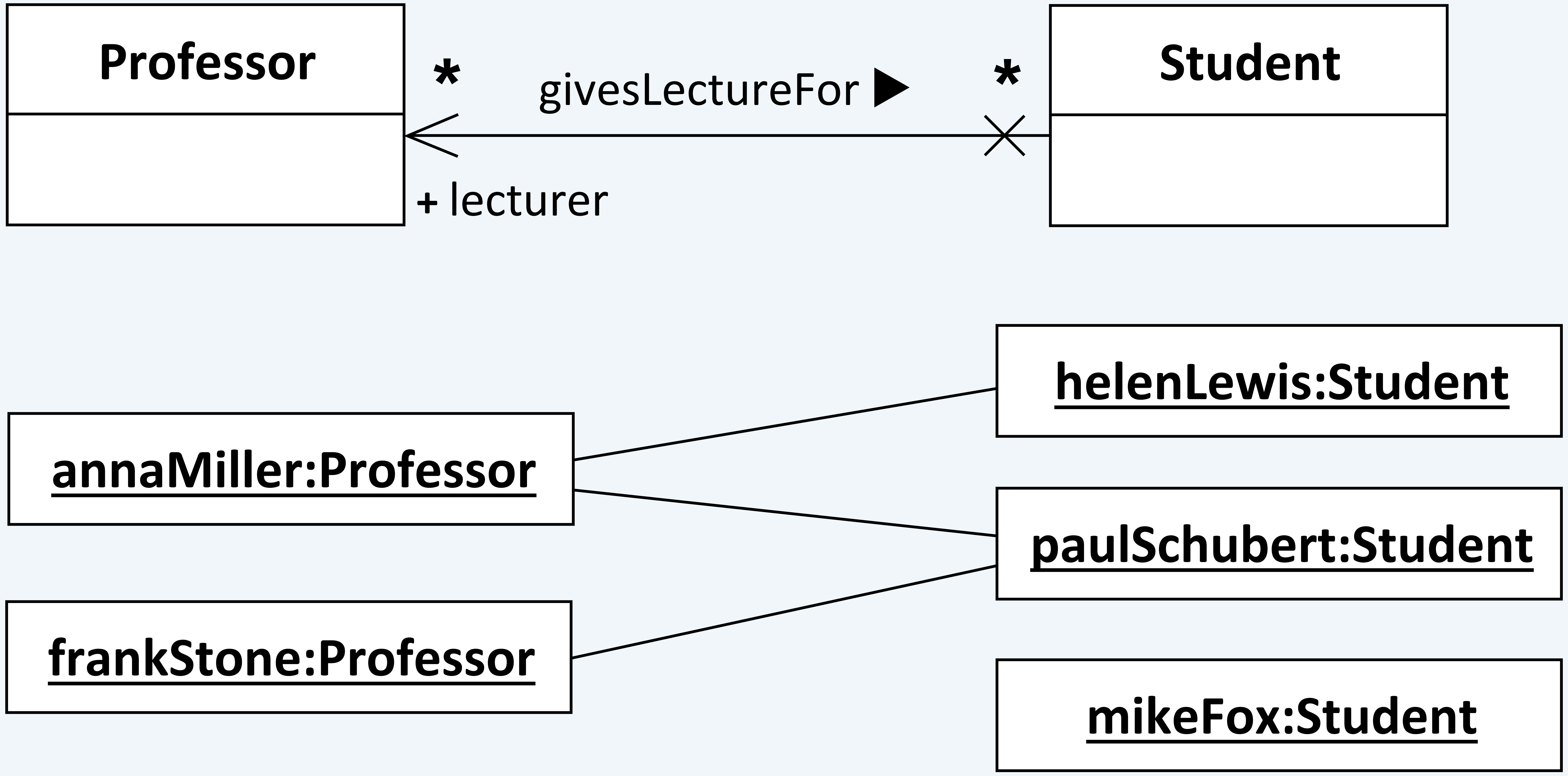
The Association

Christian Huemer und Marion Scholz

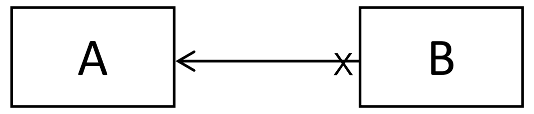
Presented by Nicholas Bzowski

Association

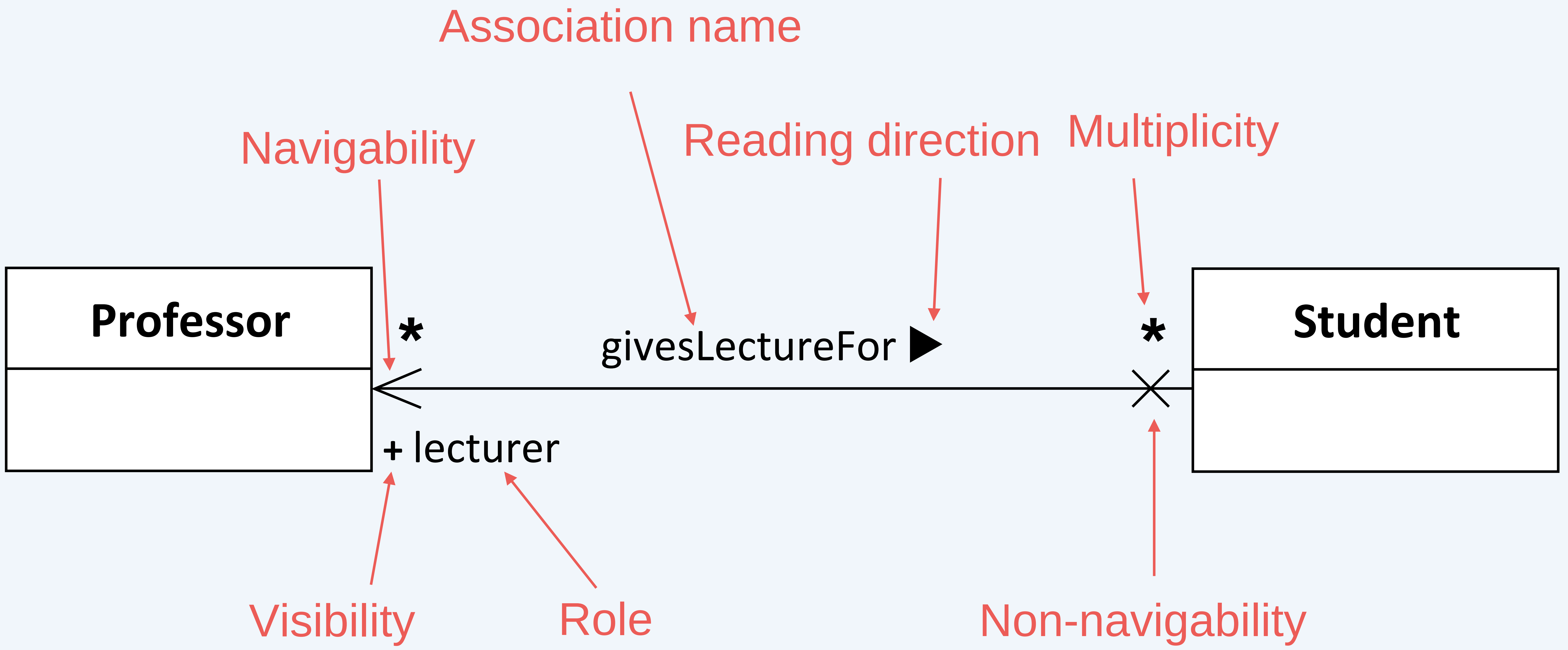
- Associations between classes model possible **links** between the **instances of the classes**



Binary Association

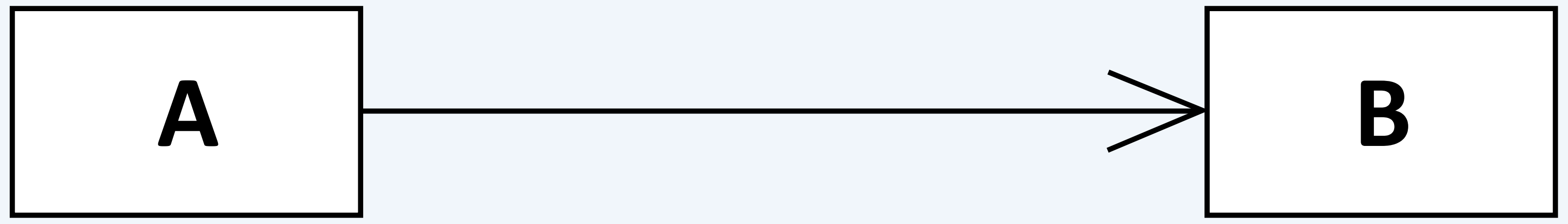


- Connects the instances of two classes with each other

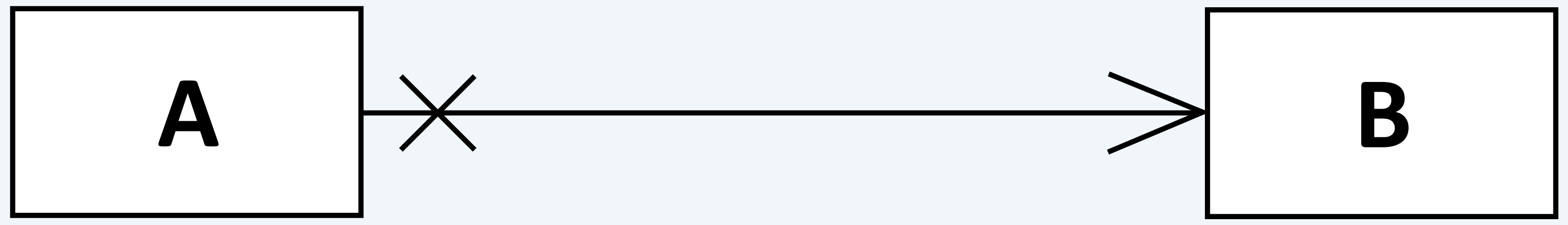


Binary Association: Navigability

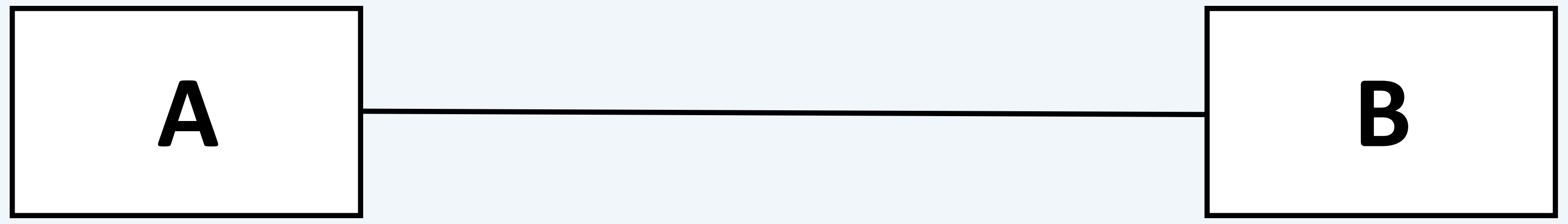
- Navigation directions are essential for later development
- **Navigable association end**: Arrow



- **Non-navigable association end**: Cross
  - **A** can access the visible attributes and operations of **B**
  - **B** cannot access any attributes and operations of **A**



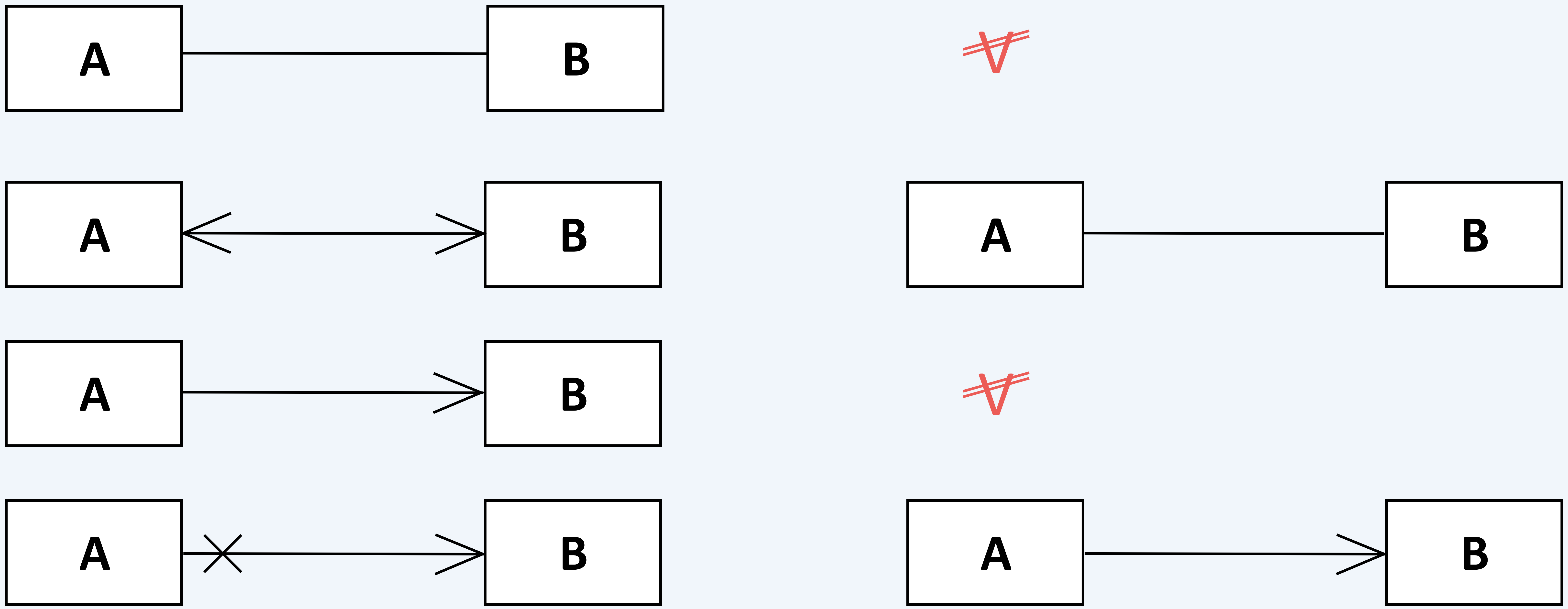
- **Neither arrow nor cross**: "undefined"



Navigability – UML Standard vs. Best Practice

UML Standard

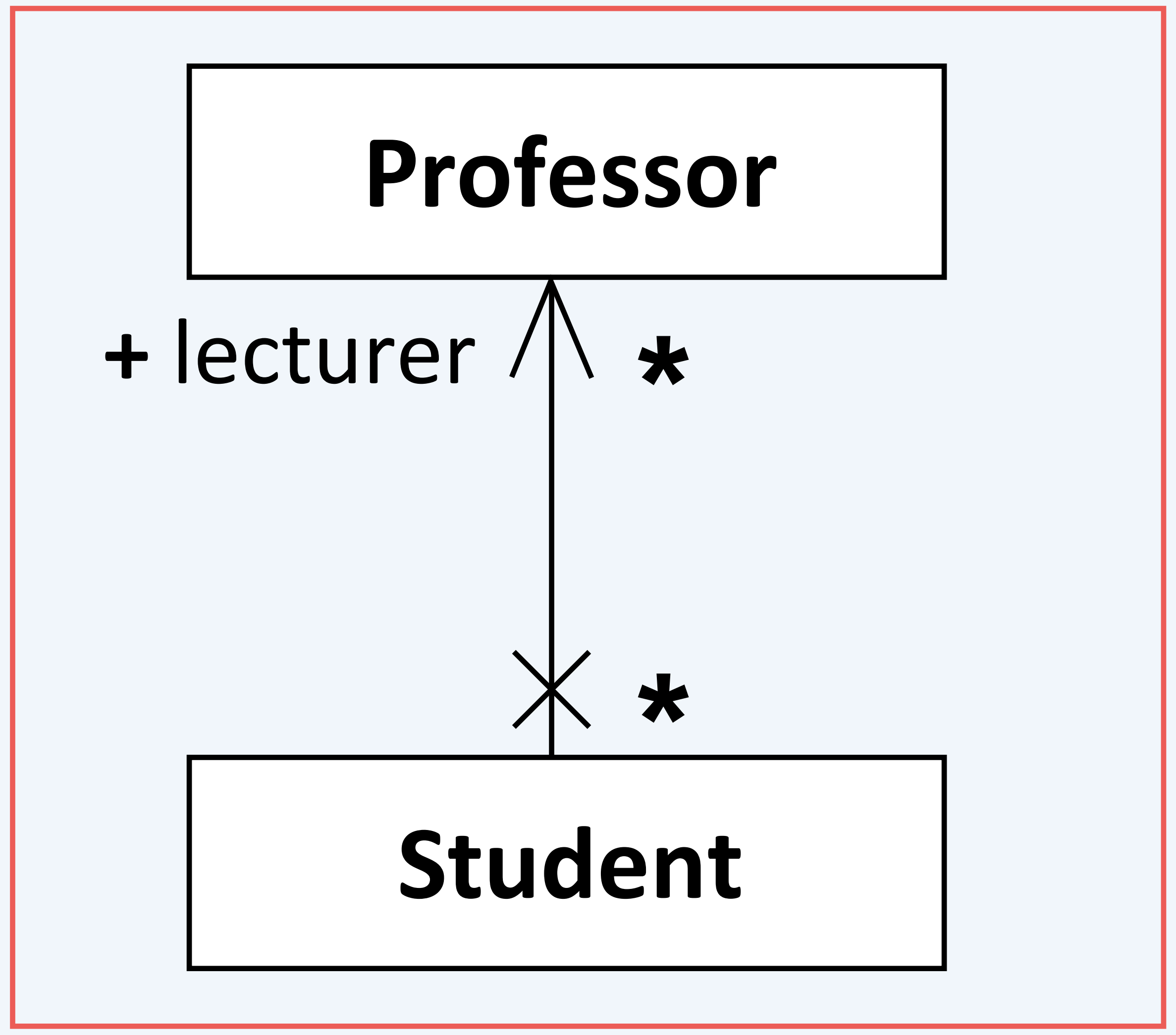
Best Practice



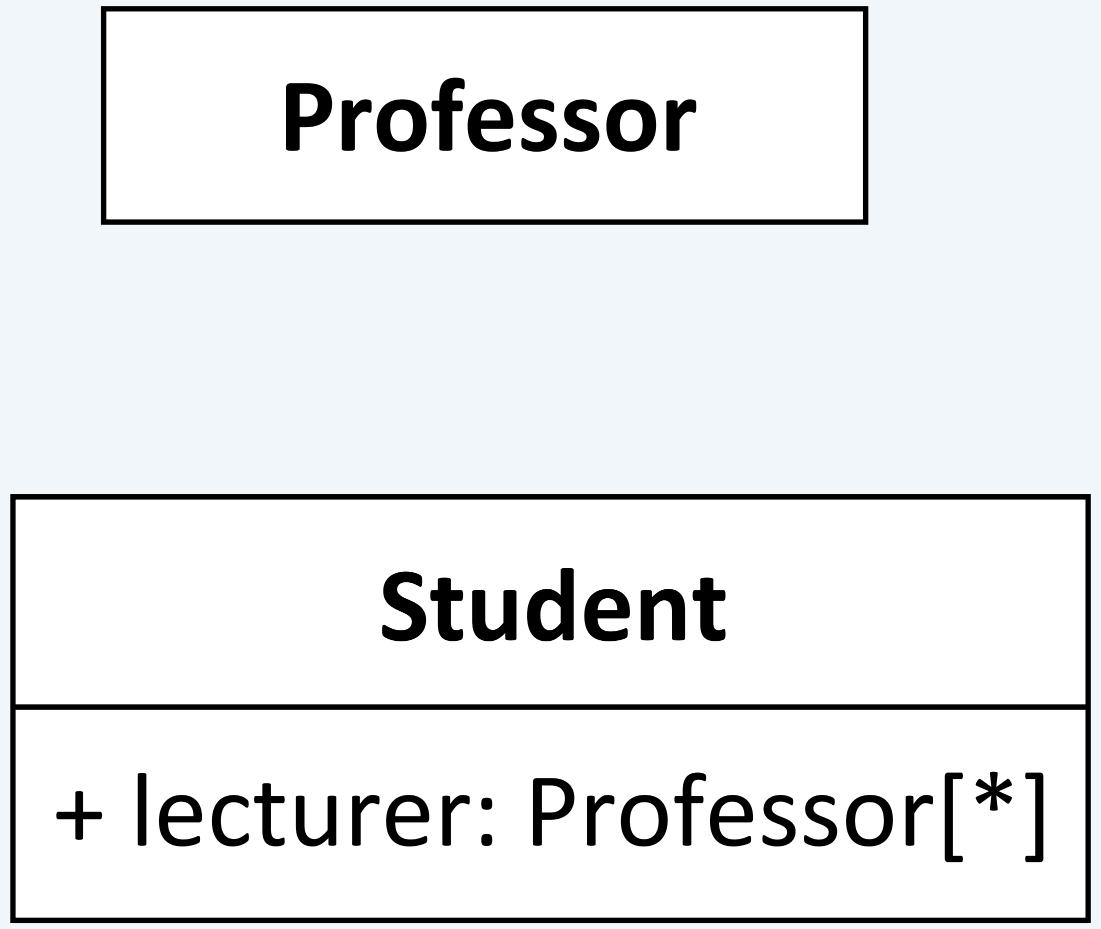
Binary association as an attribute

- A **navigable assocaition end**
  - has the same semantics as an attribute of the class at the opposite end of the association
  - can therefore also be modelled as an **attribute** instead of **a directed edge**
    - The class associated with the association end must correspond to the type of the attribute
    - The **multiplicity** must be the same
- All properties and notations of attributes can therefore be used for a navigable association end

```
class Professor{…}

 class Student{
  public Professor[] lecturer;
  …
}
```



better



Structural Modeling

Multiplicity and Roles

Christian Huemer und Marion Scholz

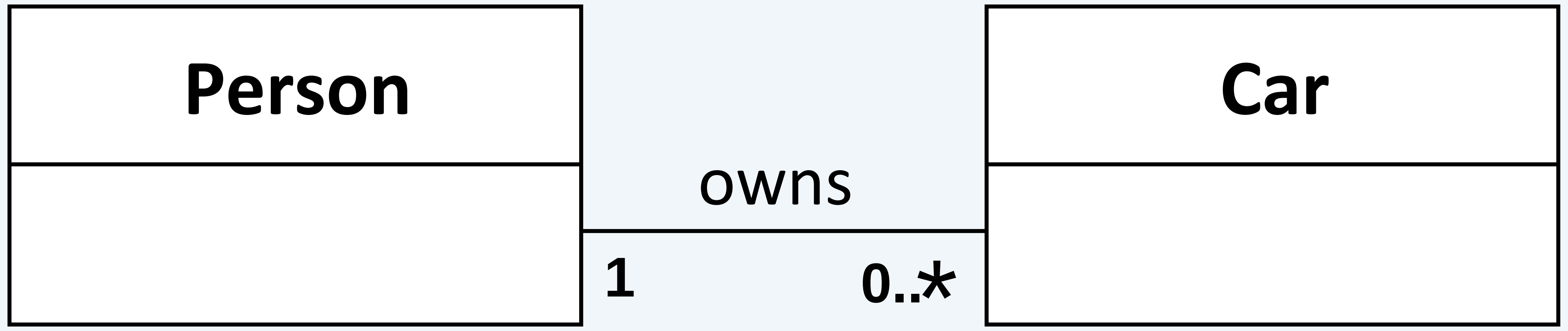
Presented by Nicholas Bzowski

Association: Multiplicity

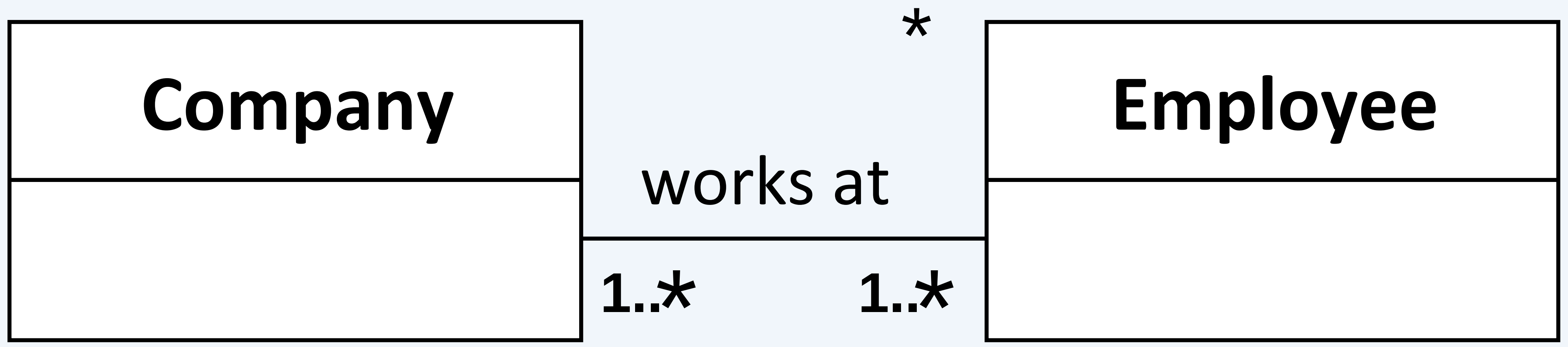
- Range: "min .. max"
- Any number: "*"
- Enumerate possible cardinalities (x, y, z)

Exactly 1:	1
Unrestricted:	* or 0..*
0 or 1:	0..1 or 0, 1
Fixed number (e.g. 3):	3
Range (e.g. >= 3):	3..*
Range (e.g. 3-6):	3..6
Enumeration:	3, 6, 7, 8, 9 or 3, 6..9

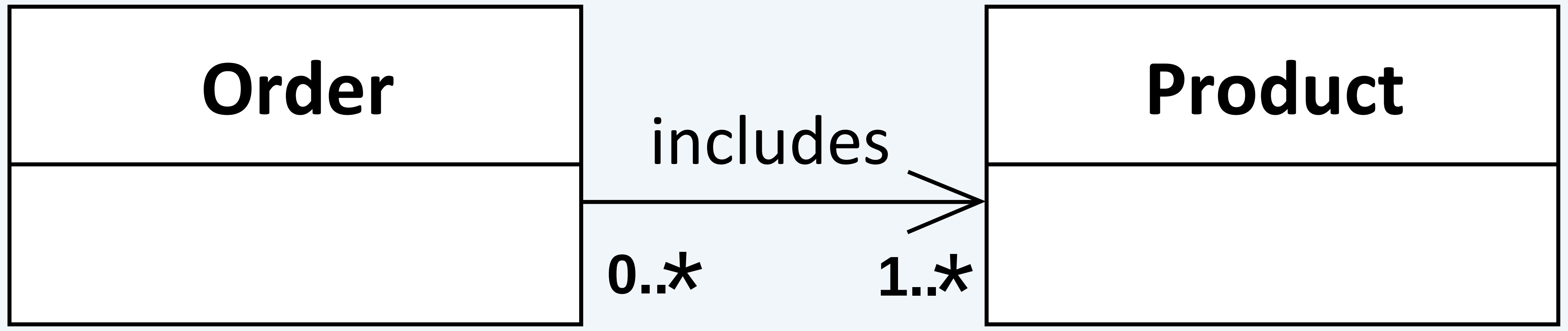
Association: Example



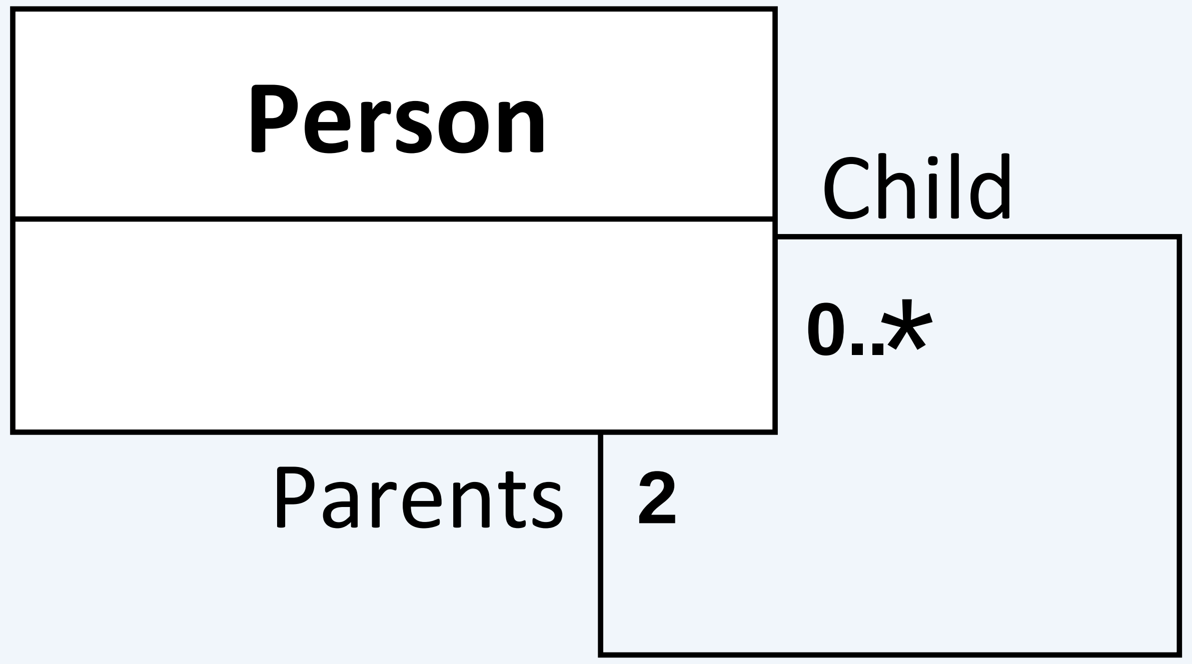
A **Car** has exactly one owner, but a **Person** can own several **Car**s (or none).



A **Company** has at least one **Employee**, an **Employee** works in at least one **Company**



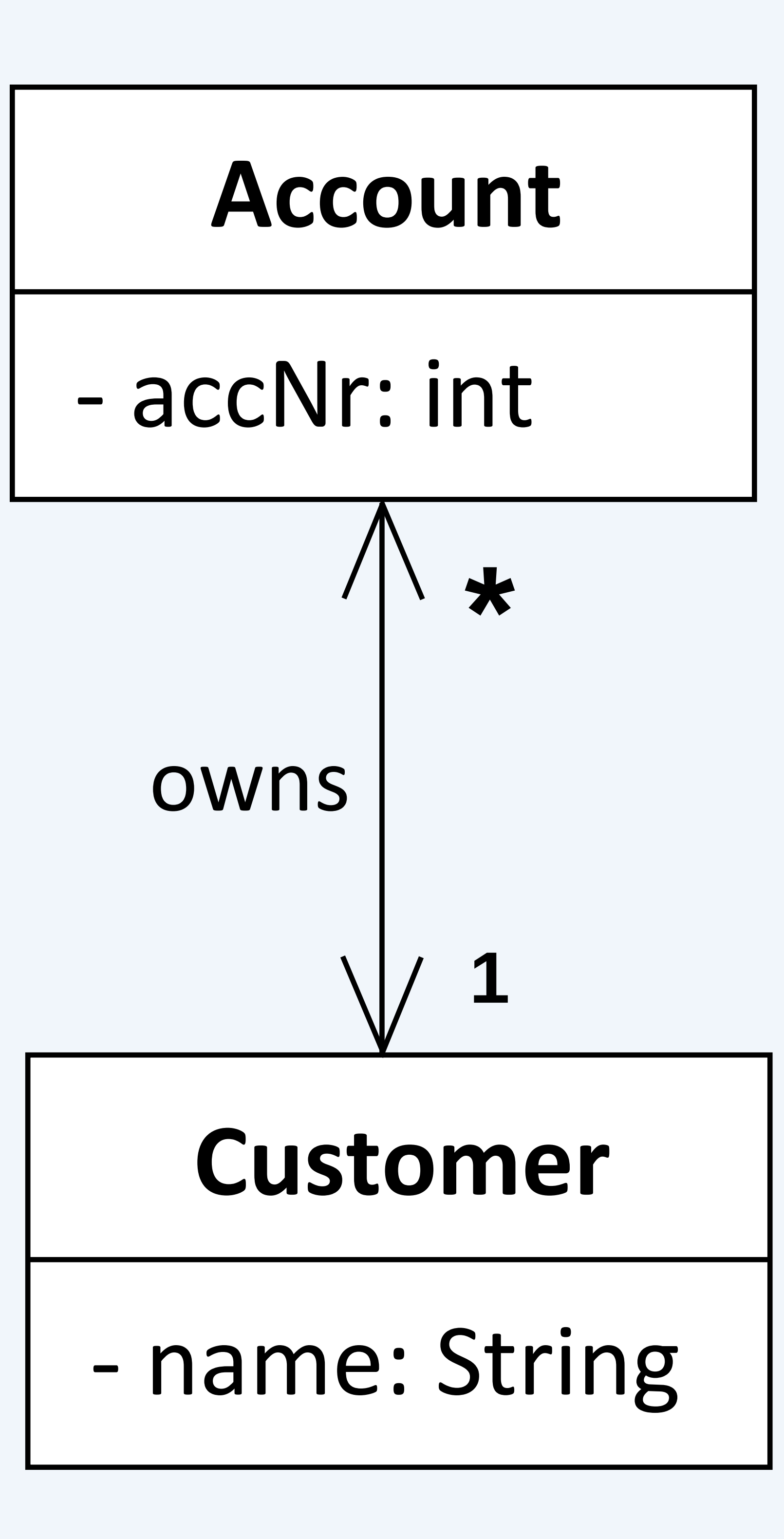
An **Order** consists of 1-n **Product**s; **Product**s can be ordered as often as required. An **Order** can be used to determine which **Product**s it contains.



A **Person** has 2 biological **Parents**, who are **Person**s, and 0 to any number of **Child**ren.
Does this model rule out the possibility of a ***Person*** *being a* ***Child*** *of themselves?*

Object diagram: Example

Class diagram



Account

- accNr: int

owns

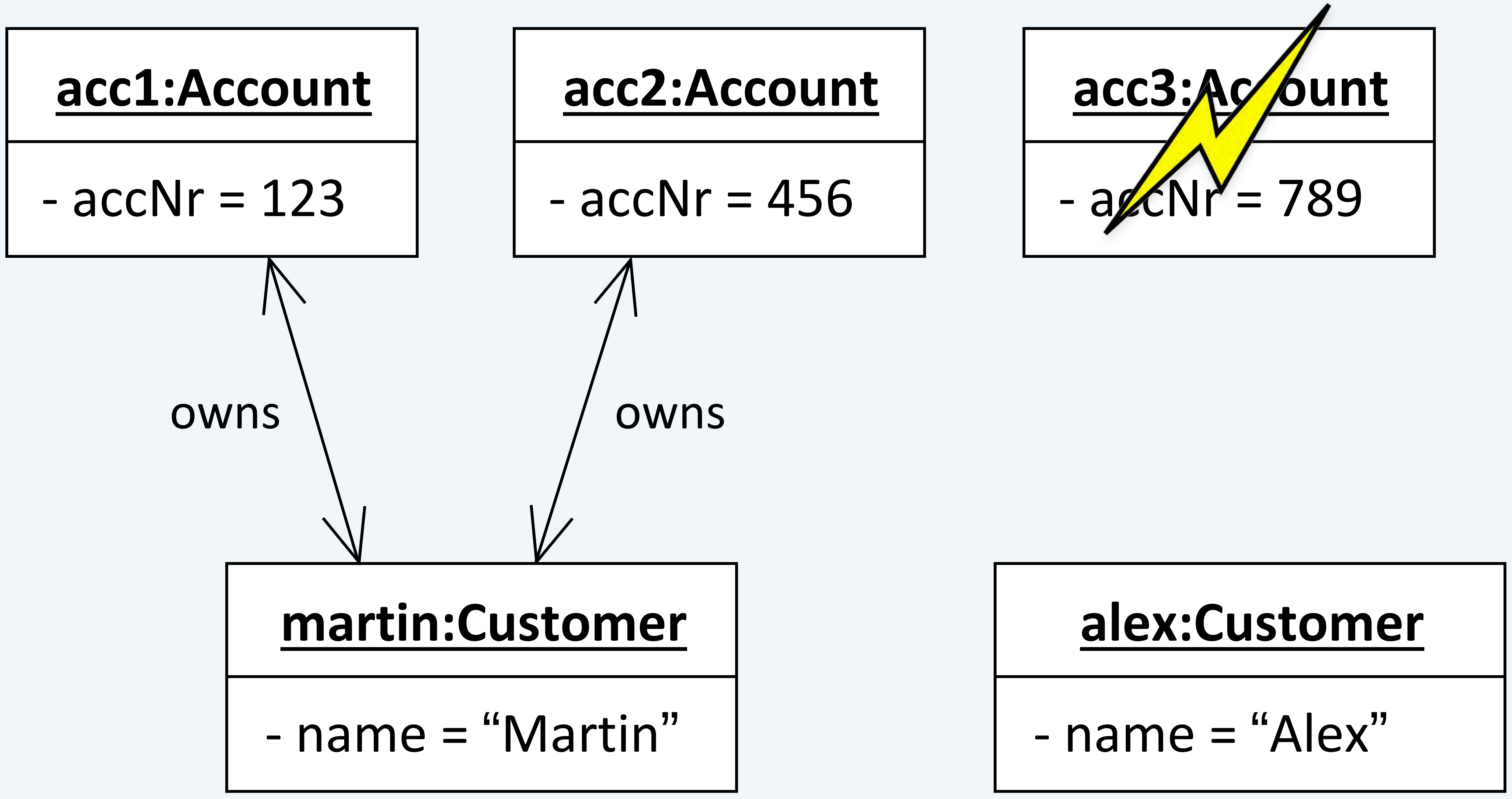
*

1

Customer

- name: String

Object diagram



acc1:Account

- accNr = 123

acc2:Account

- accNr = 456

acc3:Account

- accNr = 789

owns

owns

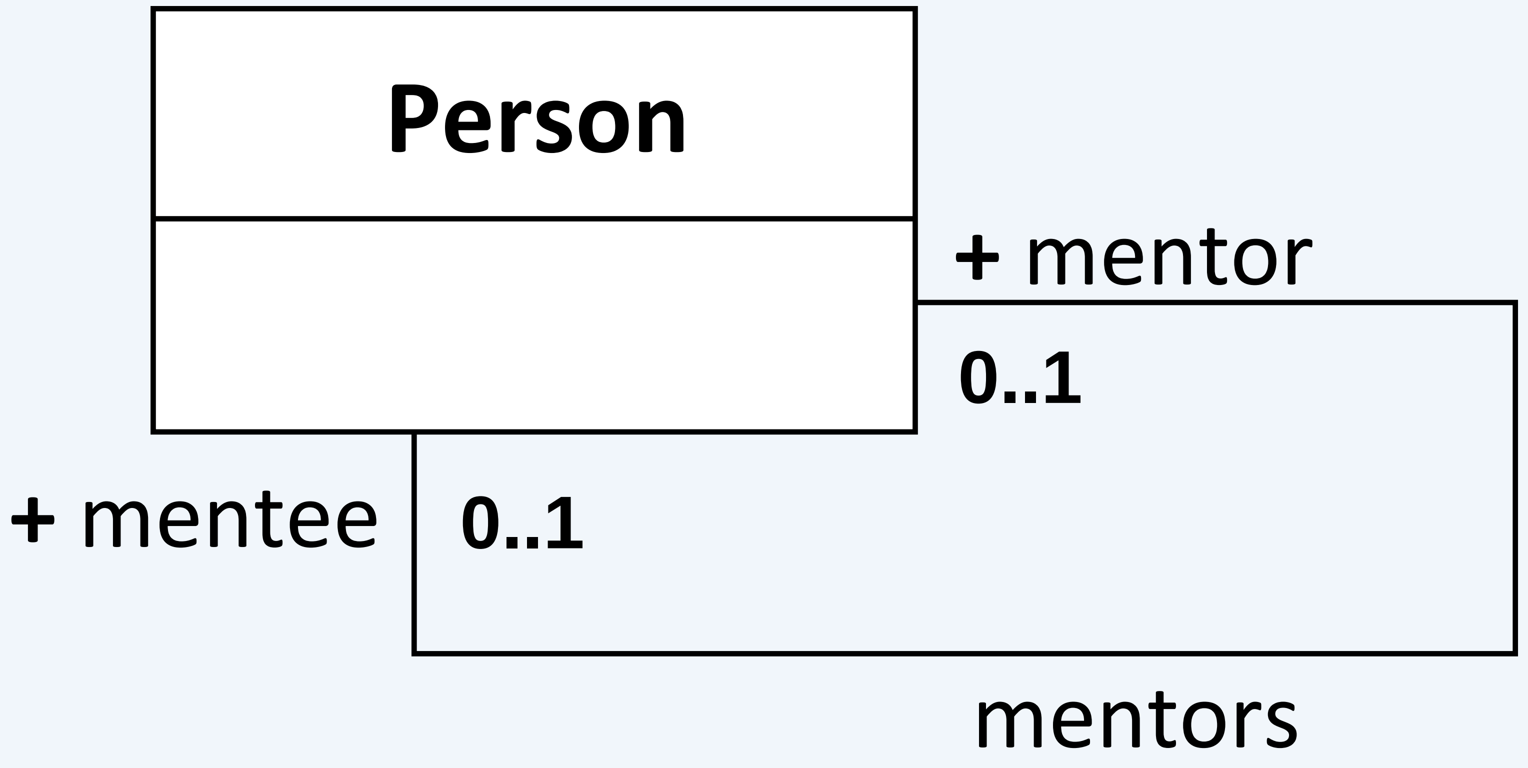
martin:Customer

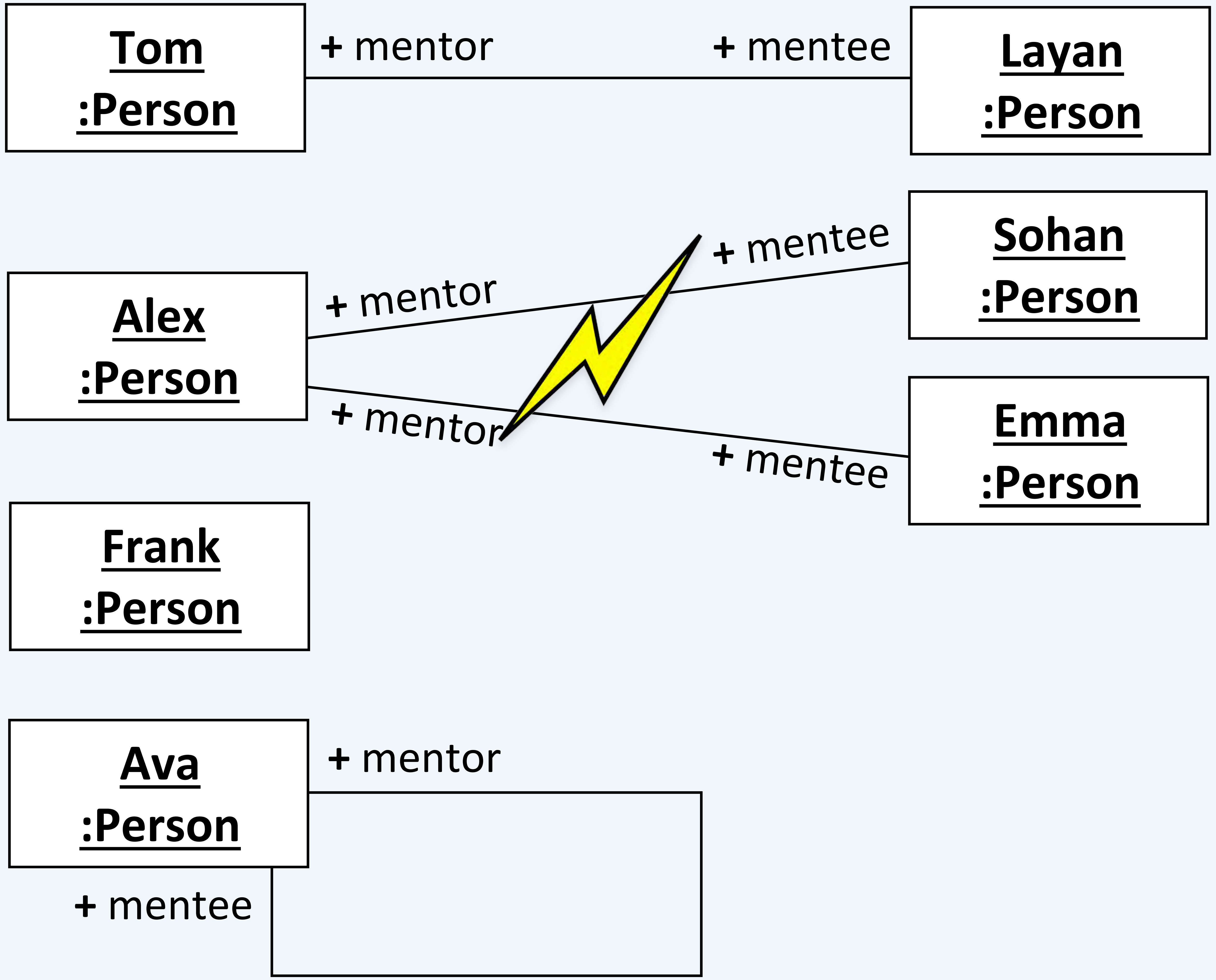
- name = “Martin”

alex:Customer

- name = “Alex”

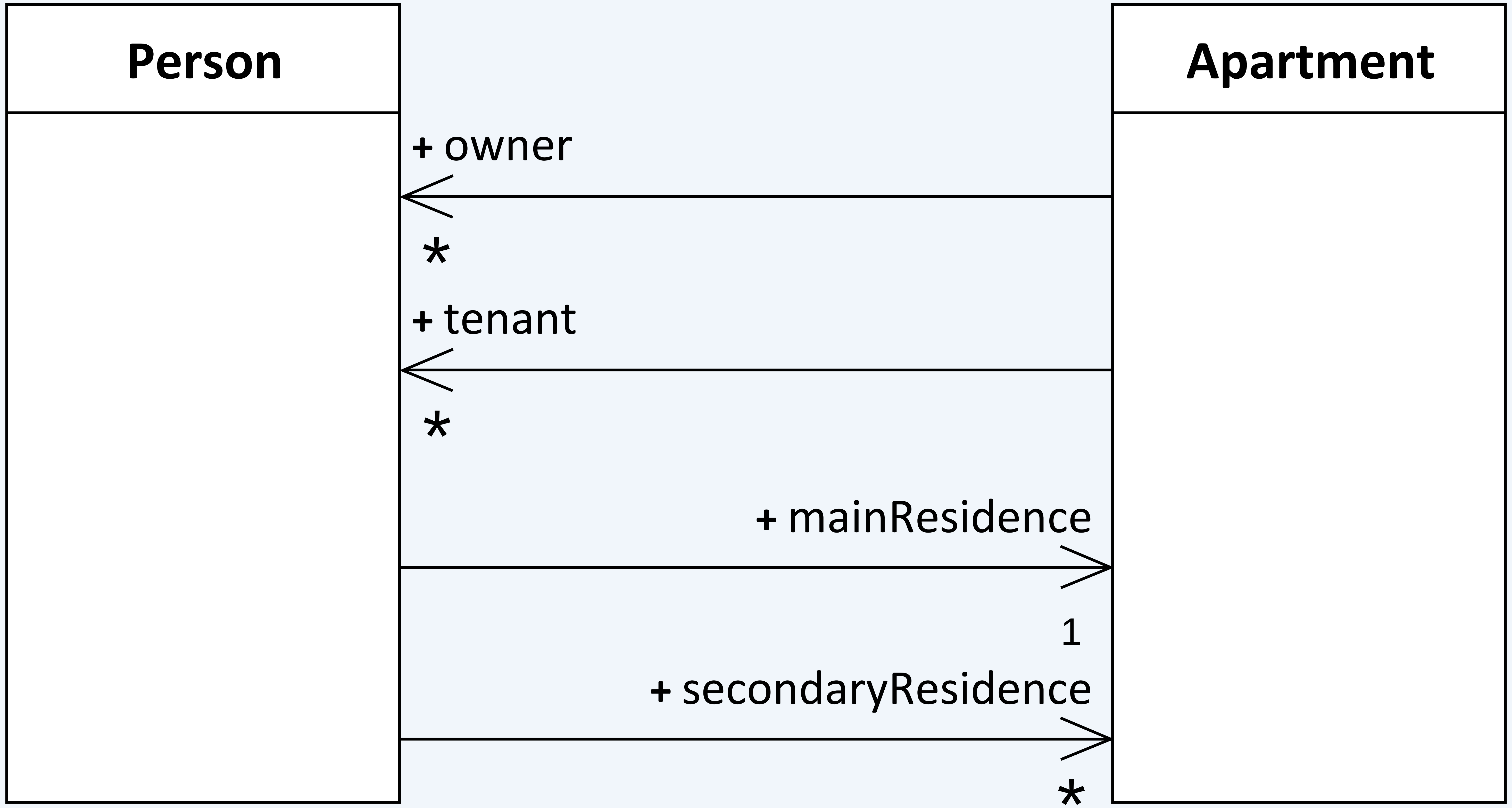
Object diagram: Example with unary association





Association: Roles

- The roles assigned to the individual objects in the object relationships can be defined



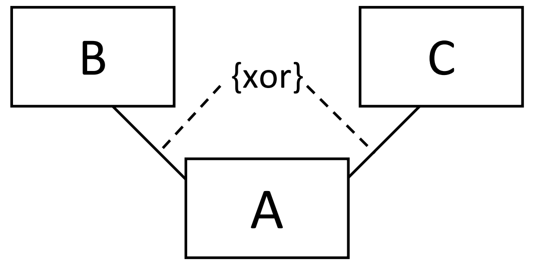
Structural Modeling

The Exclusive Association and The Association Class

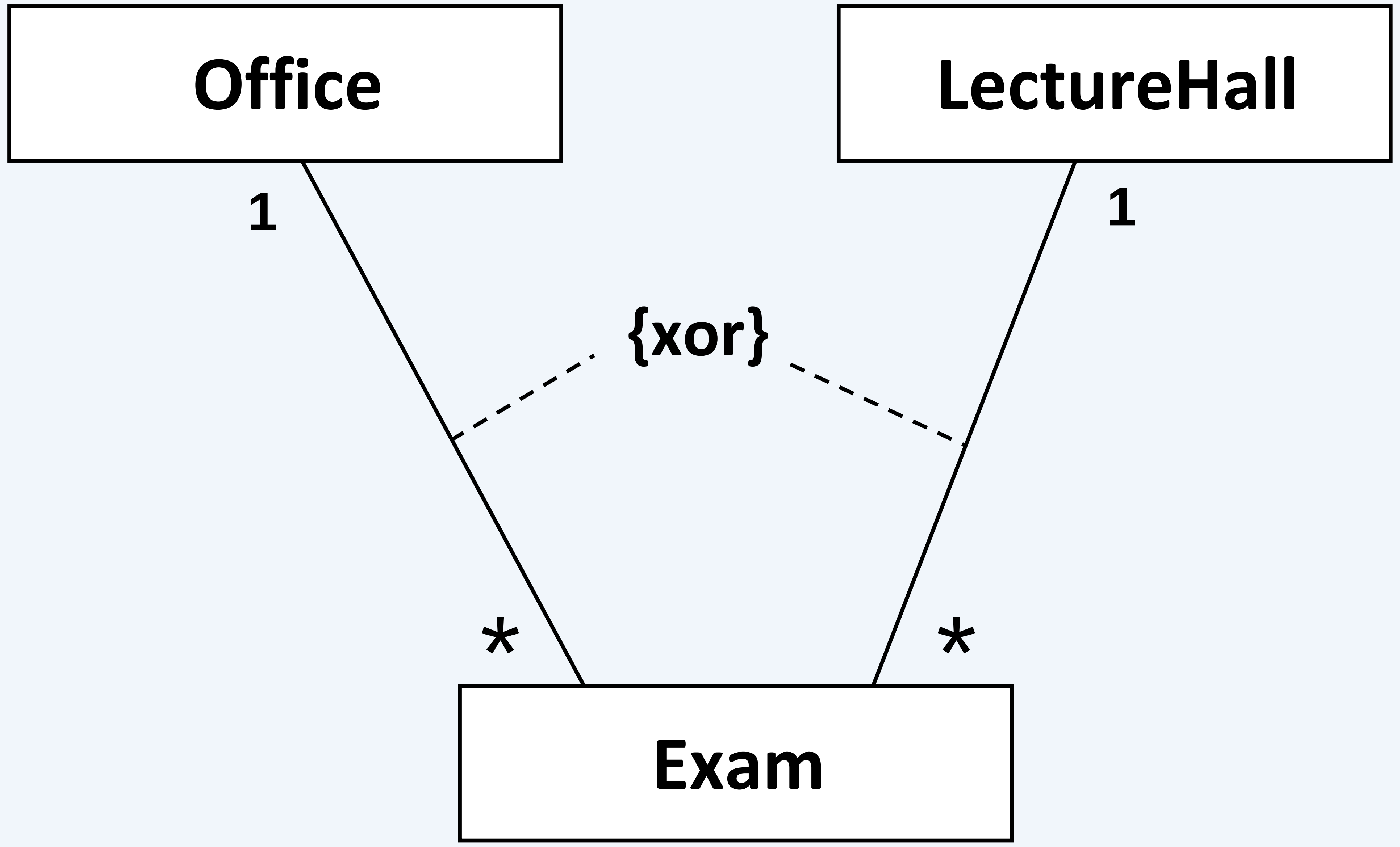
Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Exclusive Association

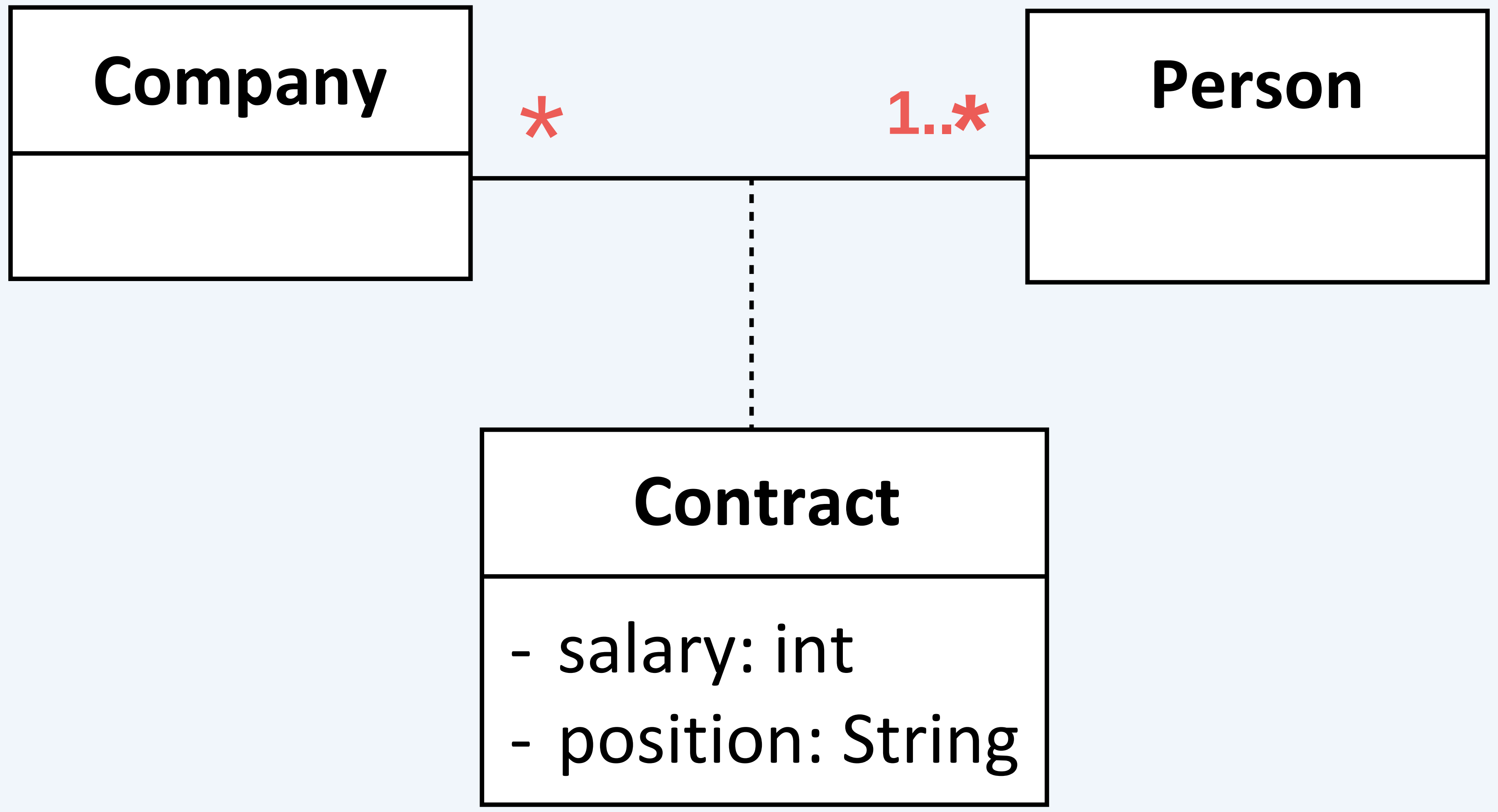


- **Only one of the possible associations** can be instantiated for a particular object at any given time: **`{xor}`**



Association Classes (1/2)

- May contain attributes of the association
  - Necessary for m:n associations with attributes
  - Useful for 1:1 and 1:n associations to add flexibility



Company

*

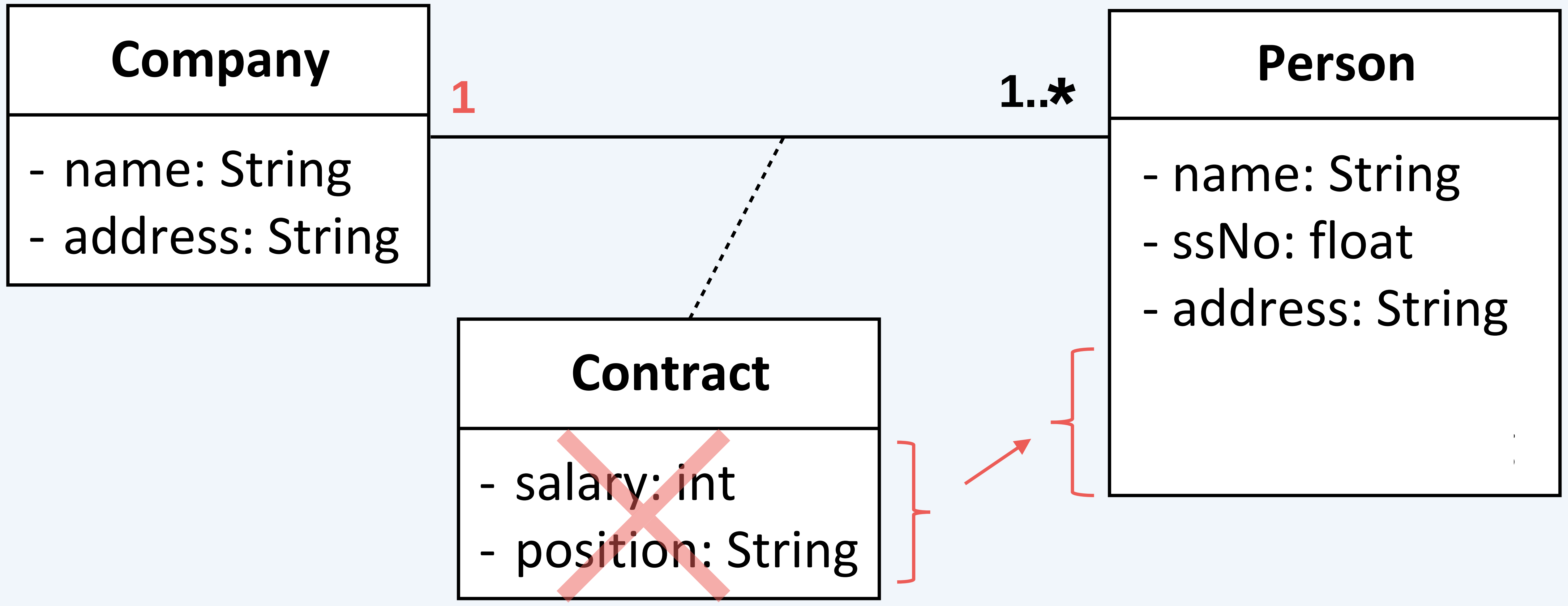
1..*

Person

Contract

- salary: int
- position: String

Association class



Company

- name: String
- address: String

1

1..*

Person

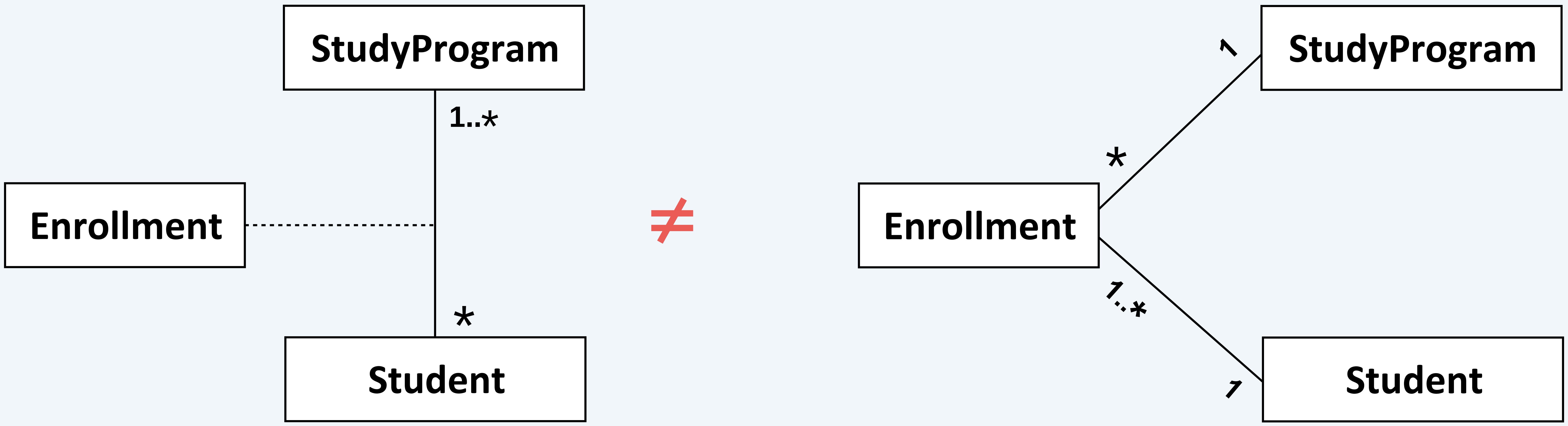
- name: String
- ssNo: float
- address: String

Contract

- ~~salary: int~~
- ~~position: String~~

Association Classes (2/2)

- Normal class not equivalent to association class



A **`Student`** can only register once for a particular ***`StudyProgram`***

A **`Student`** can have several ***`Enrollments`*** of the same ***`StudyProgram`***

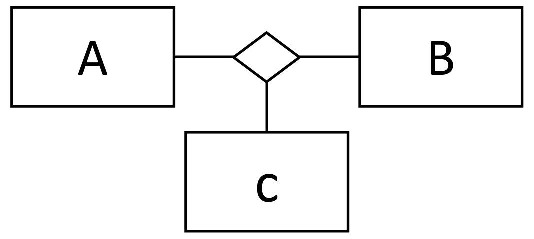
Structural Modeling

The n-ary Association

Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

n-ary Association

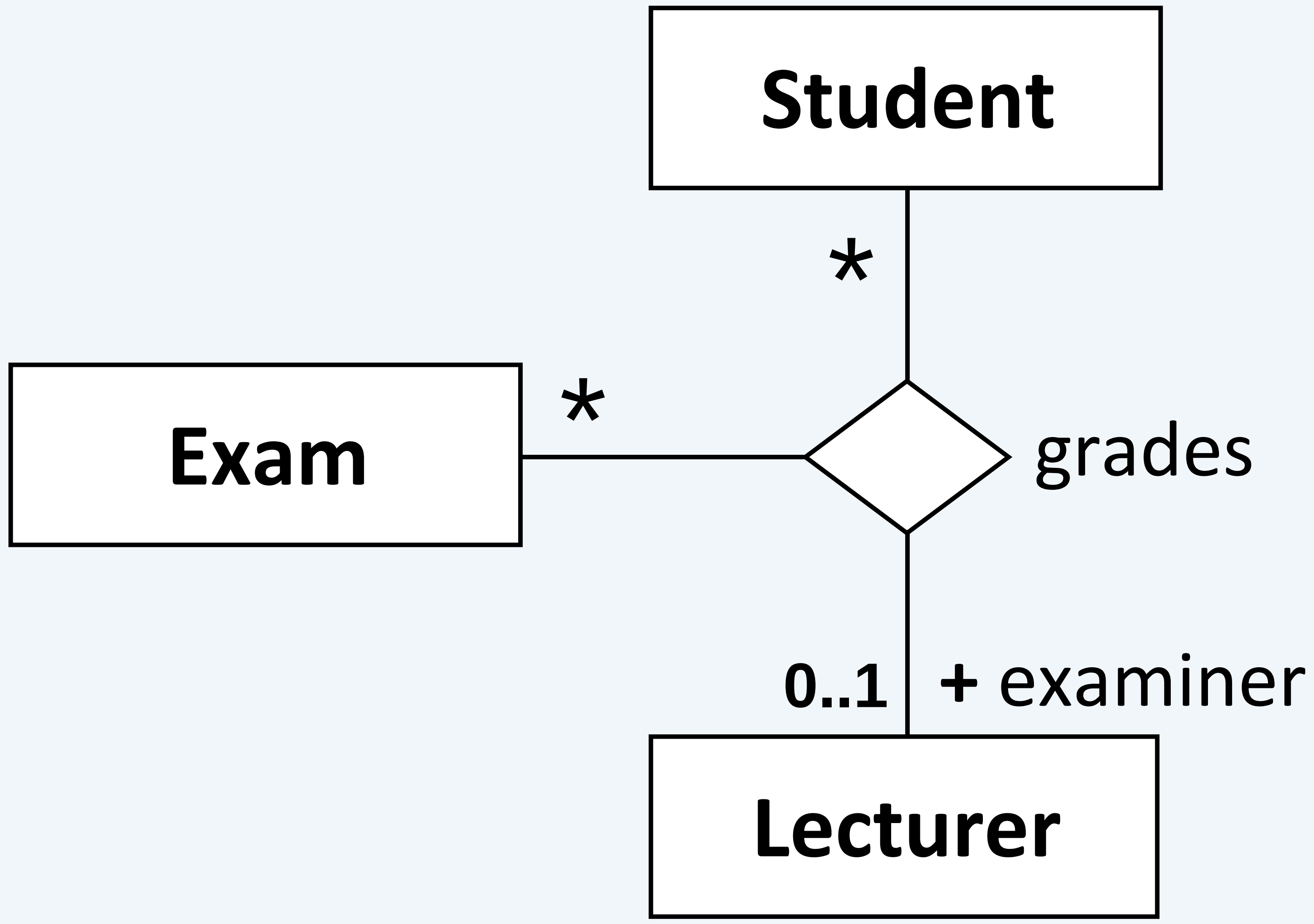


- **Relationship between more than two classes**
  - Navigation direction cannot be specified
  - N Lines for defining the multiplicities
  - A certain combination of objects of all other classes are related to a certain number of objects of this class.
- **Multiplicities imply restrictions,**

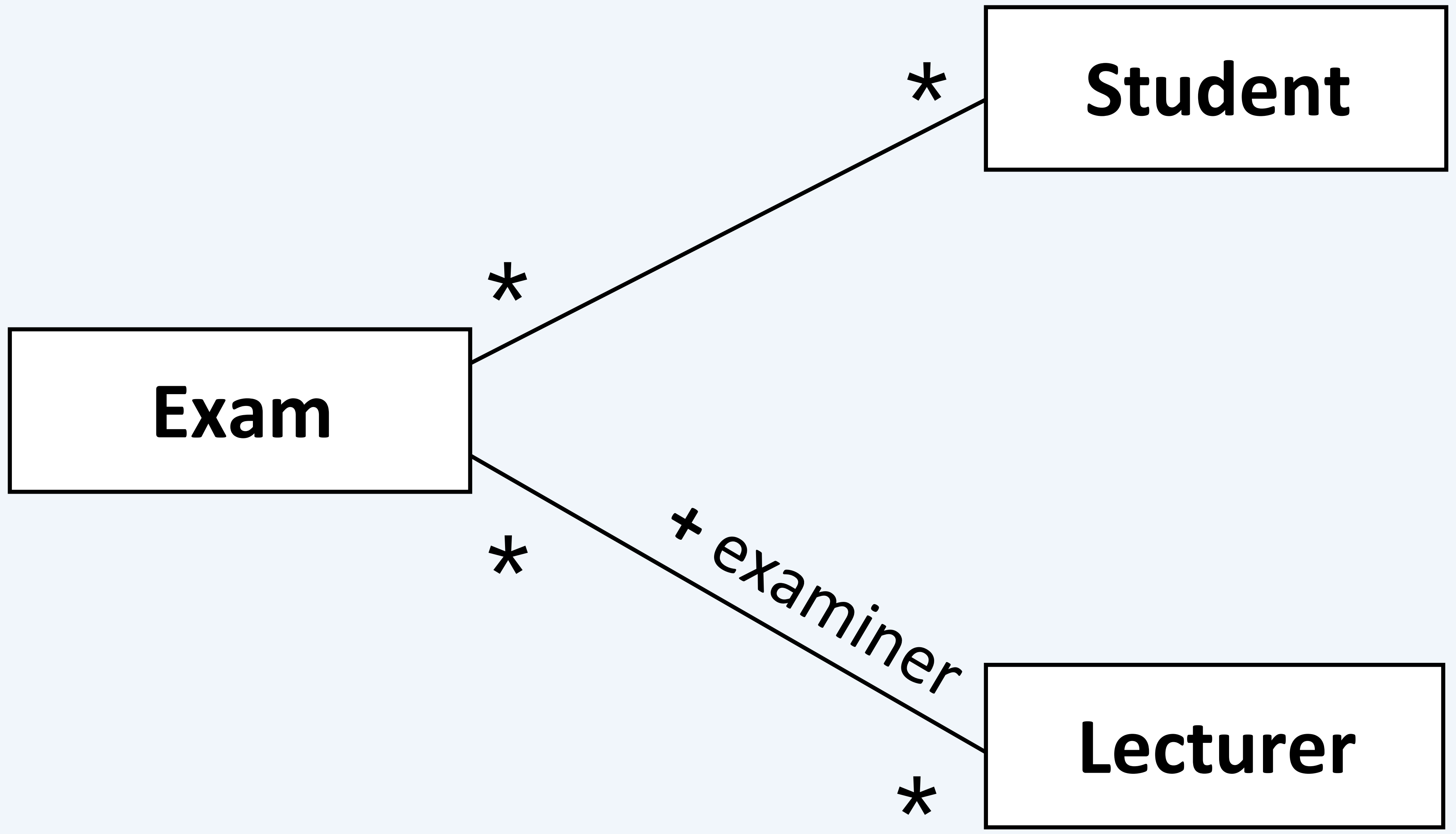
  in a certain case functional dependencies
- If the multiplicity of a ternary association is specified as 1 for class C, there is a functional dependency (A, B)→(C)

n-ary Association: Example

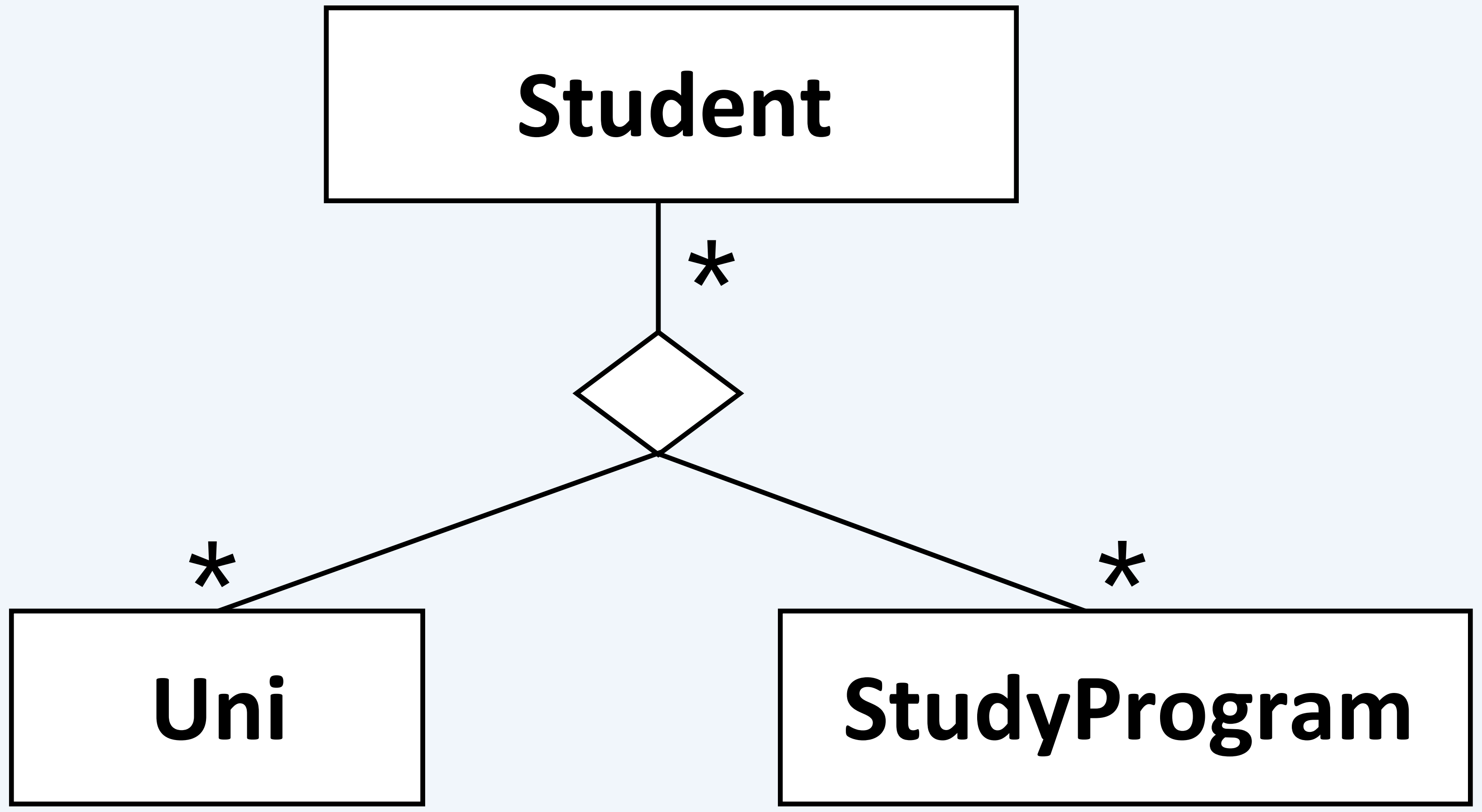
- (**Student**, **Exam**) → (**Lecturer**)
  - A **Student** takes an **Exam** with one or no **Lecturer**
- (**Exam**, **Lecturer**) → (**Student**)
  - An **Exam** can be taken by several **Student**s with one **Lecturer**
- (**Student**, **Lecturer**) → (**Exam**)
  - A **Student** can be assessed by one **Lecturer** for several **Exam**s

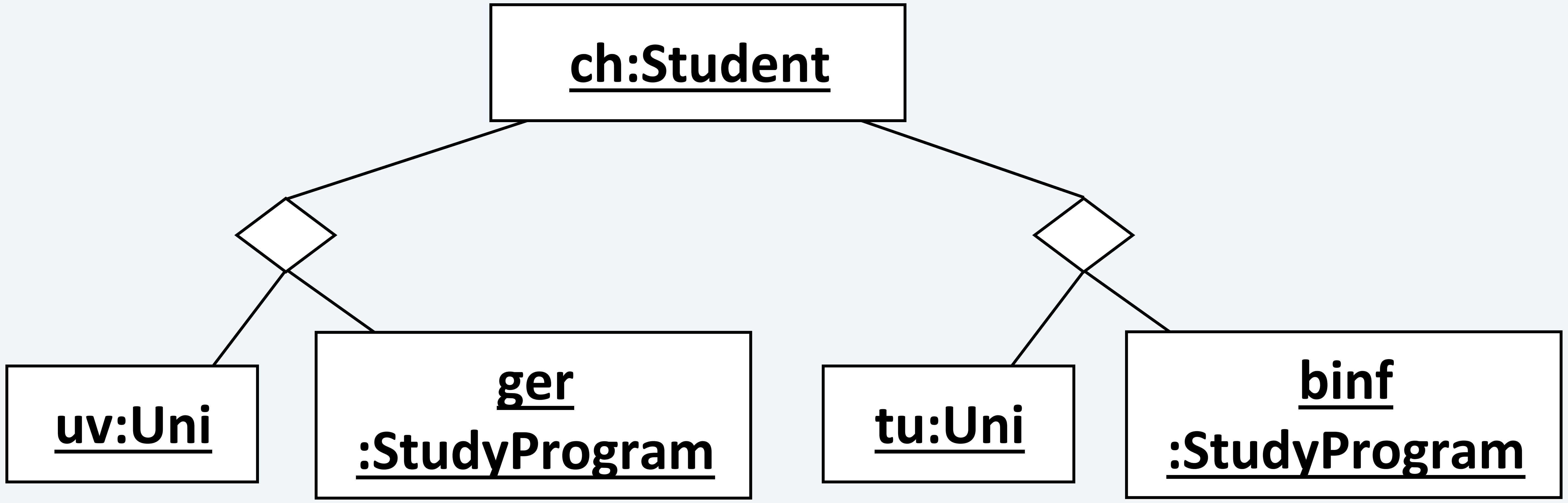


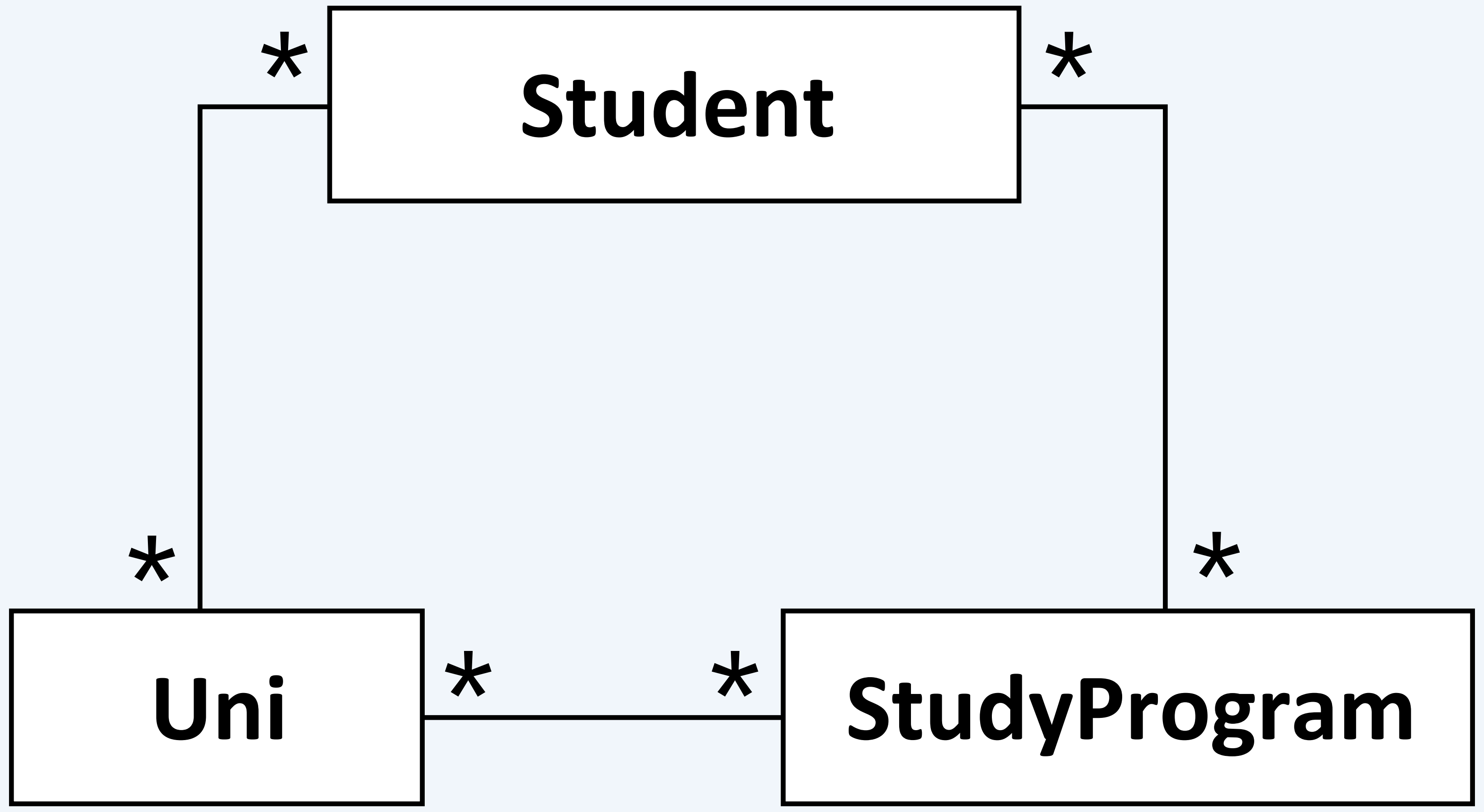
≠

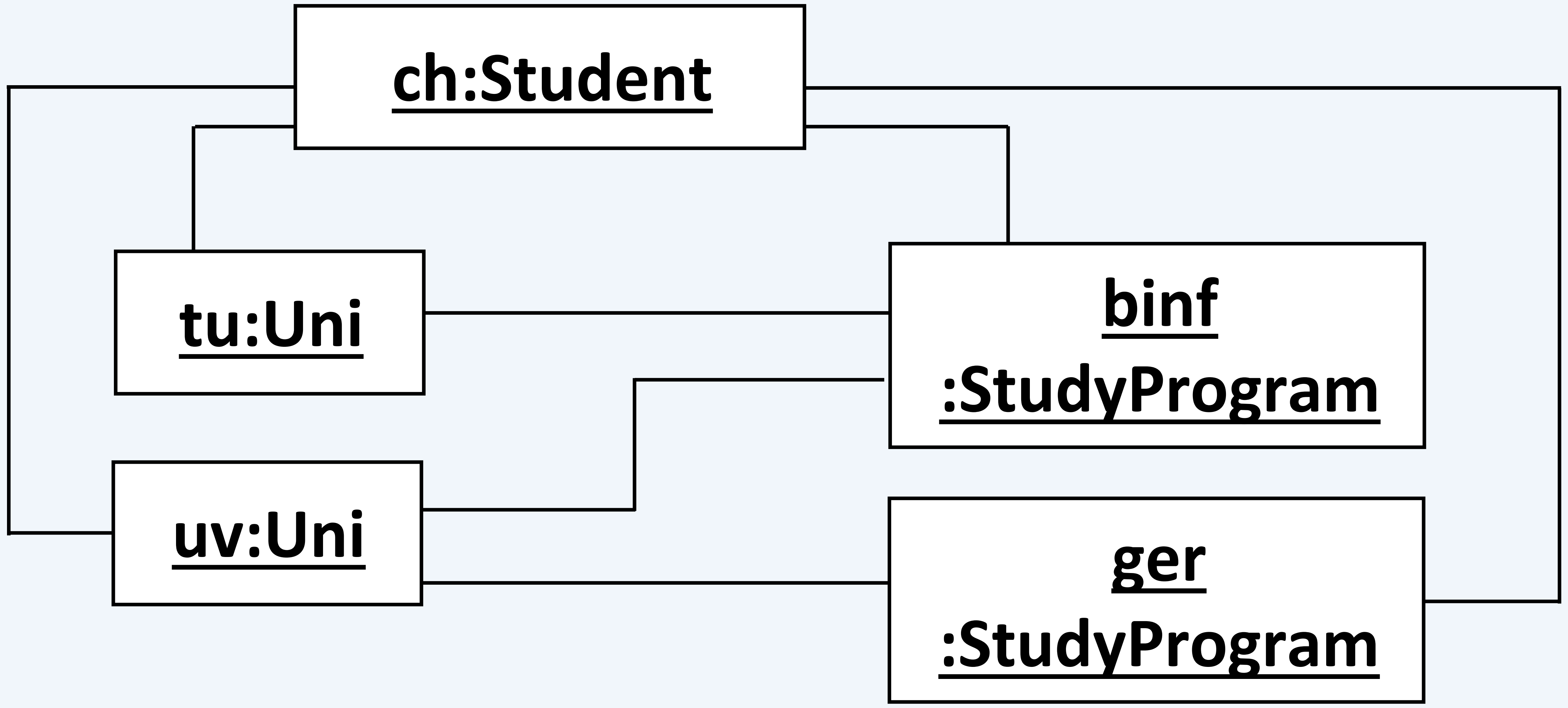


Ternary Association versus Binary Association









Structural Modeling

The Aggregation

Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Aggregation

- Aggregation is a special form of association for parts-whole relationships

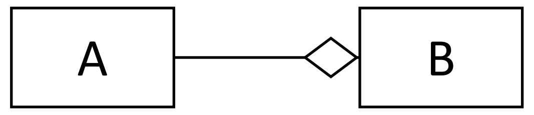
- Two types of aggregations:
  - Shared aggregation
  - Composition

- The following properties apply to both:
  - Transitivity:

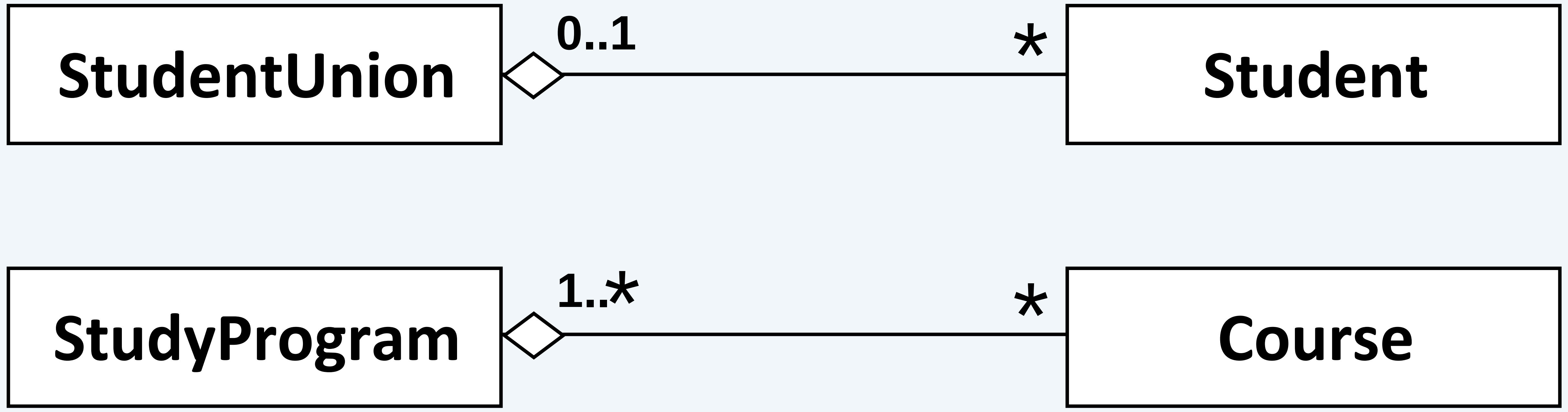
    C is part of B and B is part of A ⇒ C is part of A
  - Asymmetry:

    B is part of A ⇒ A is not part of B

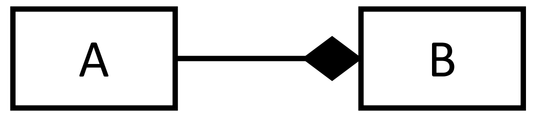
Shared Aggregation



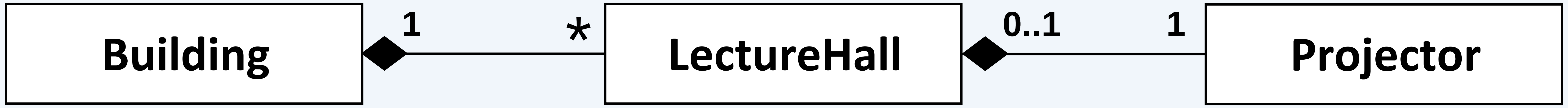
- Denoted as a white diamond
- Weak affiliation of the parts,
  i.e. the parts are **independent** of their whole
- The multiplicity on the side of the whole can be > 1
- Only restricted propagation semantics apply
- The composite objects form **a directed, acyclic graph**



Starke Aggregation (= Komposition)



- A specific part may only be contained in a **maximum of one composite object** at any given time
- The **multiplicity** of the aggregation end of the association can be (at most) 1
- The parts are **dependent** on the composite Object
- **Propagation semantics** apply
- The composite objects form a **tree**
- A hierarchy of „part-of“ relationships can be represented (transitivity!)

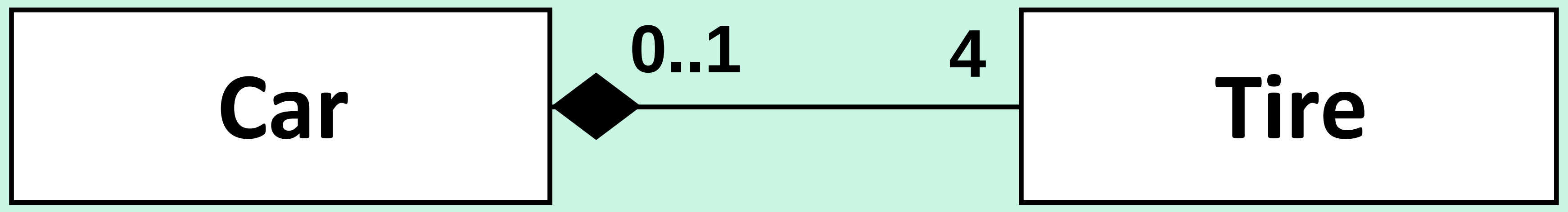


If the `Building` is deleted, the `LectureHall` is also deleted.

The `Projector` can exist without the `LectureHall`, but if it is included in the `LectureHall` when it is deleted, the `Projector` will also be deleted

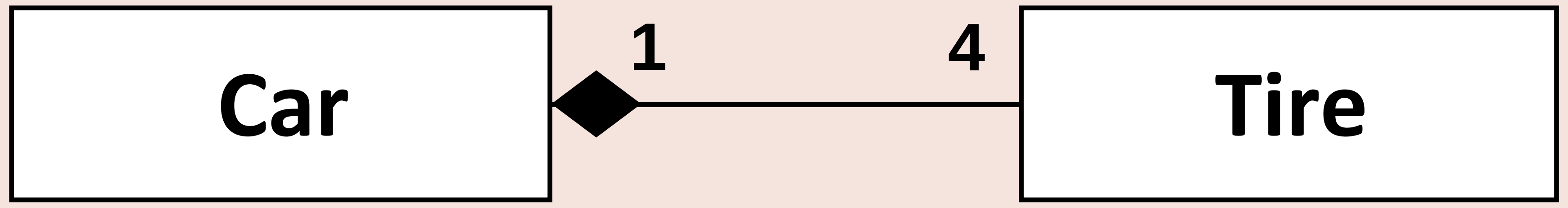
Composition and Aggregation

- Which of the following relationships is valid?



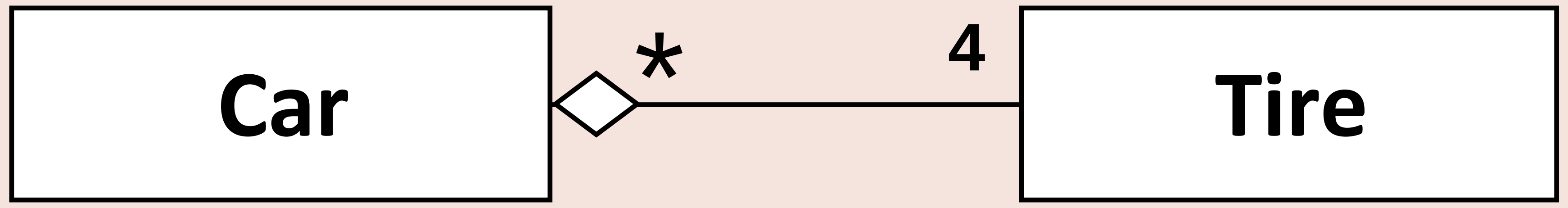
A tire can also exist without a car.
A tire belongs to at most one car.

YES



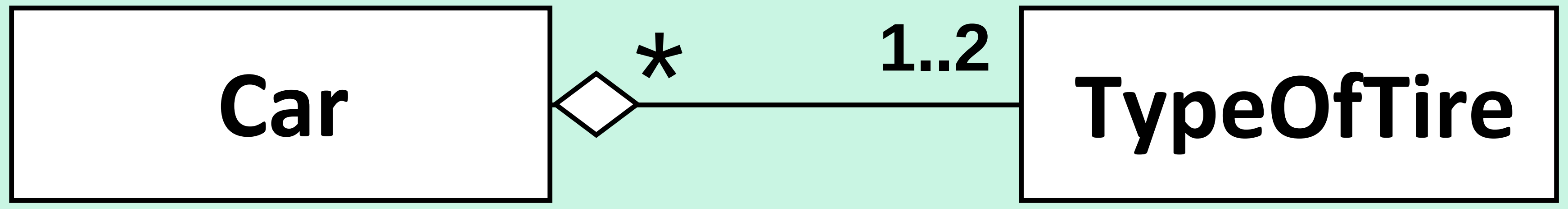
A tire cannot exist without a car.

NO



A tire can be part of several cars.

NO



A car has 1 or 2 types of tires.
Several cars can have the same type of tire.

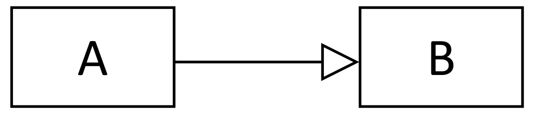
YES

Structural Modeling
The Generalization

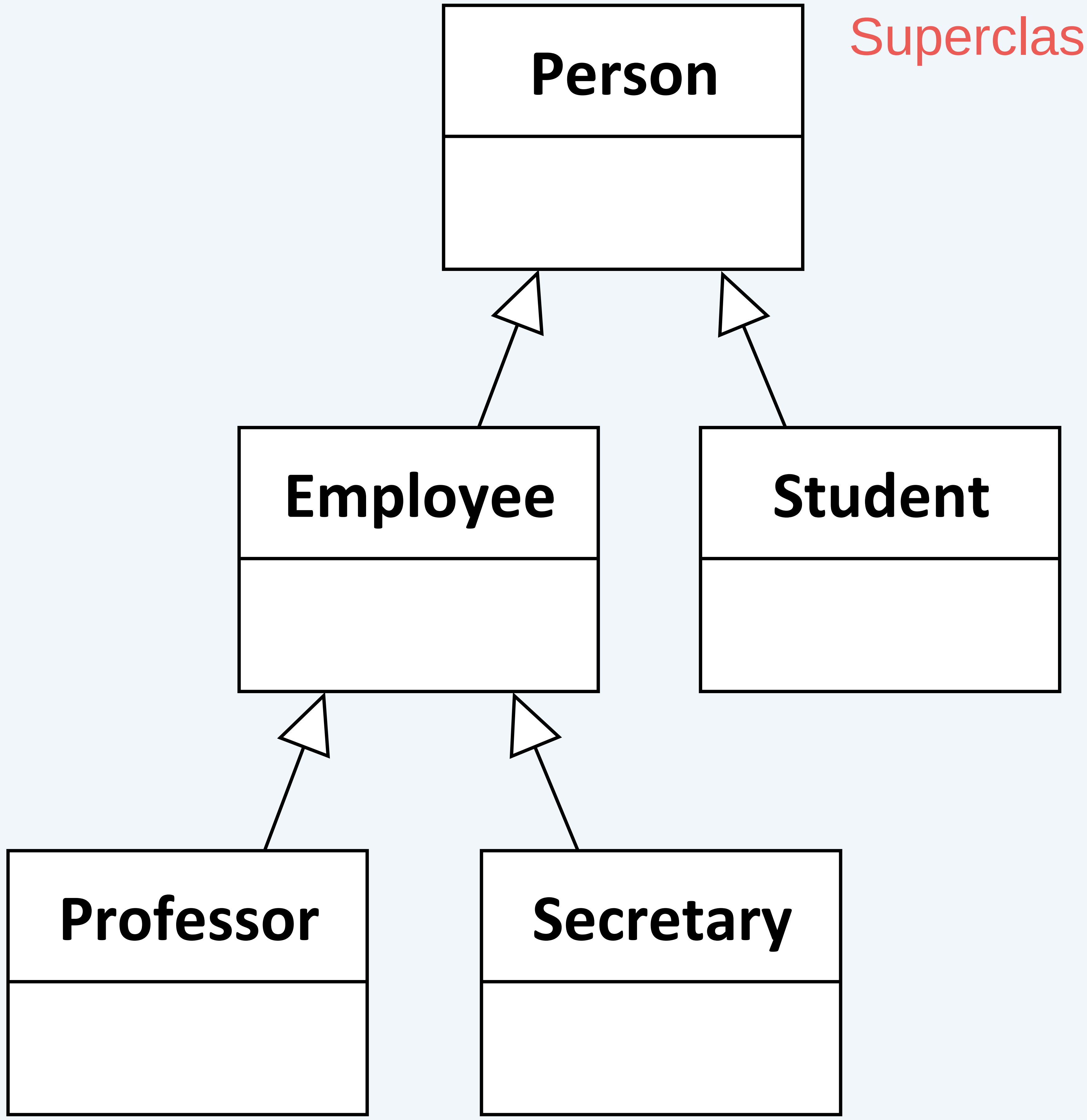
Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Generalization



- **Taxonomic relationship** between a more general class and a more specialized class
  - The subclass **inherits** the properties of the superclass
  - **Additional properties** can be added
  - An instance of the subclass can be used wherever an instance of the superclass is permitted (at least syntactically)
- Generalization is used to represent a hierarchy of "is-a" relationships (transitivity!)



Superclass

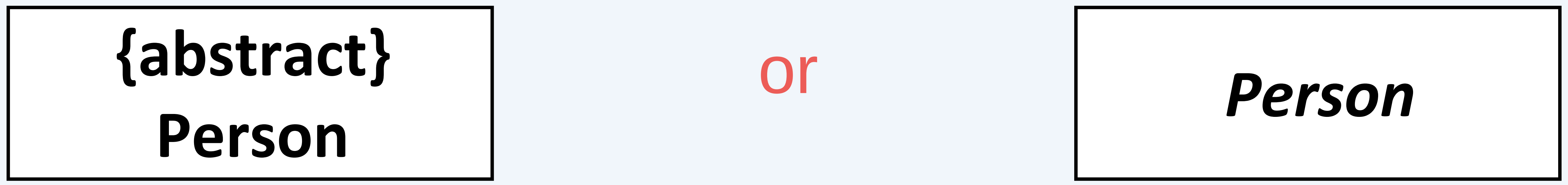
Subclasses

`Secretary` is an **`Employee`** and a **`Person`**

Abstract Classes (1/2)

- Classes that **cannot** be **instantiated**
- Only useful in **generalization hierarchies**
- Used to **“highlight“ common features** of a number of subclasses

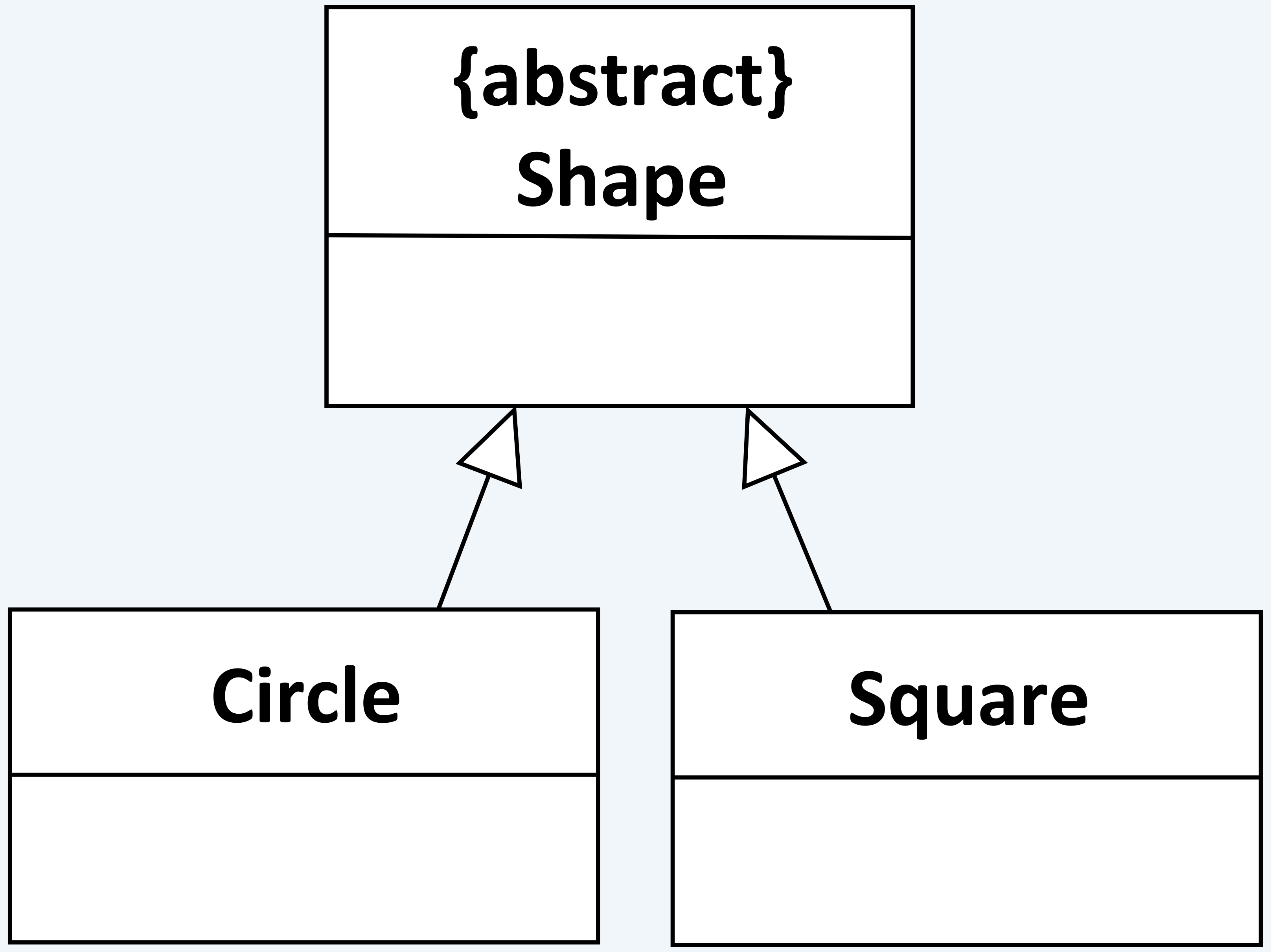
- Notation: Keyword **`{abstract}`** or class name in italic font

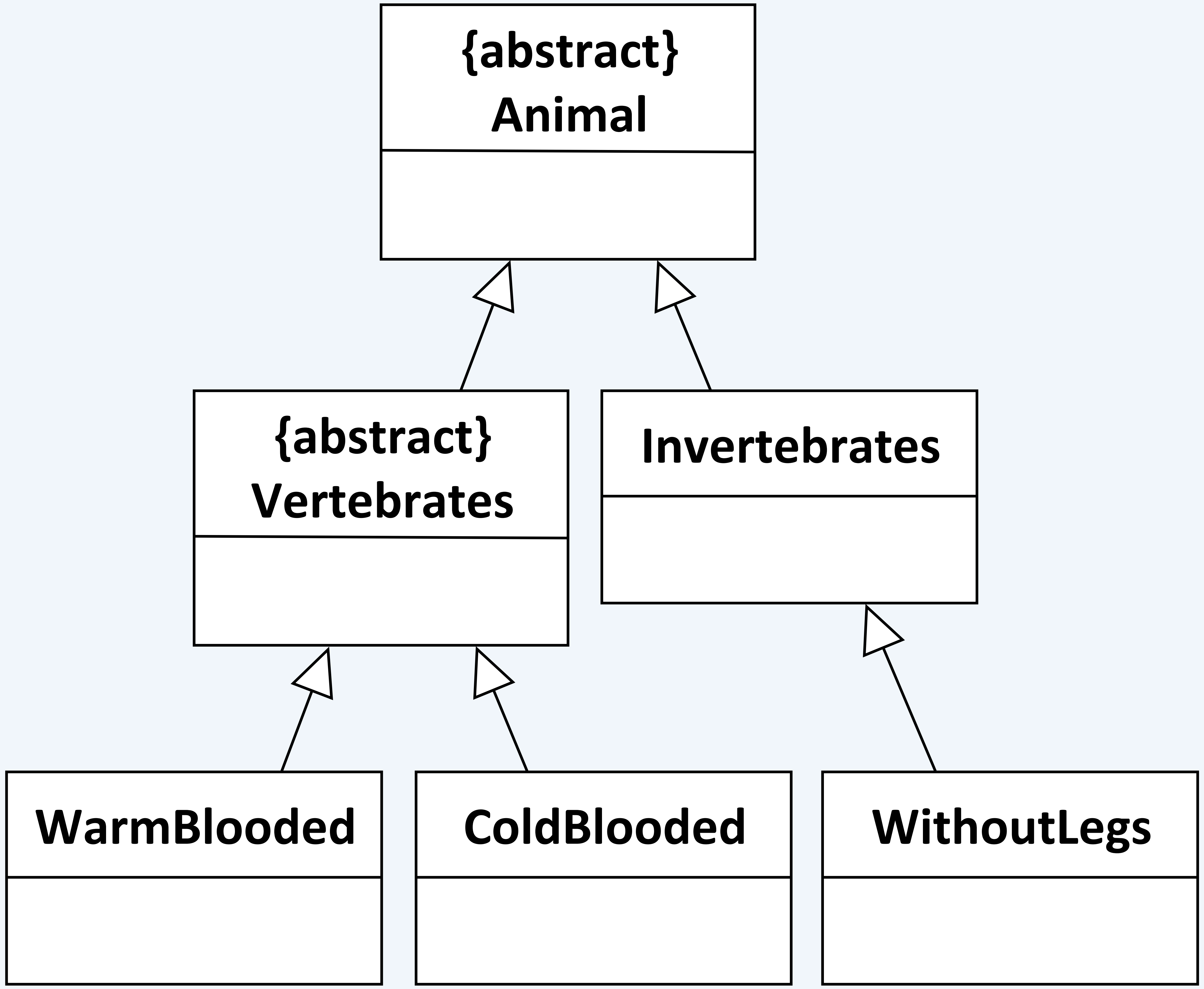


- The distinction between concrete and abstract operations of a class works in the same way

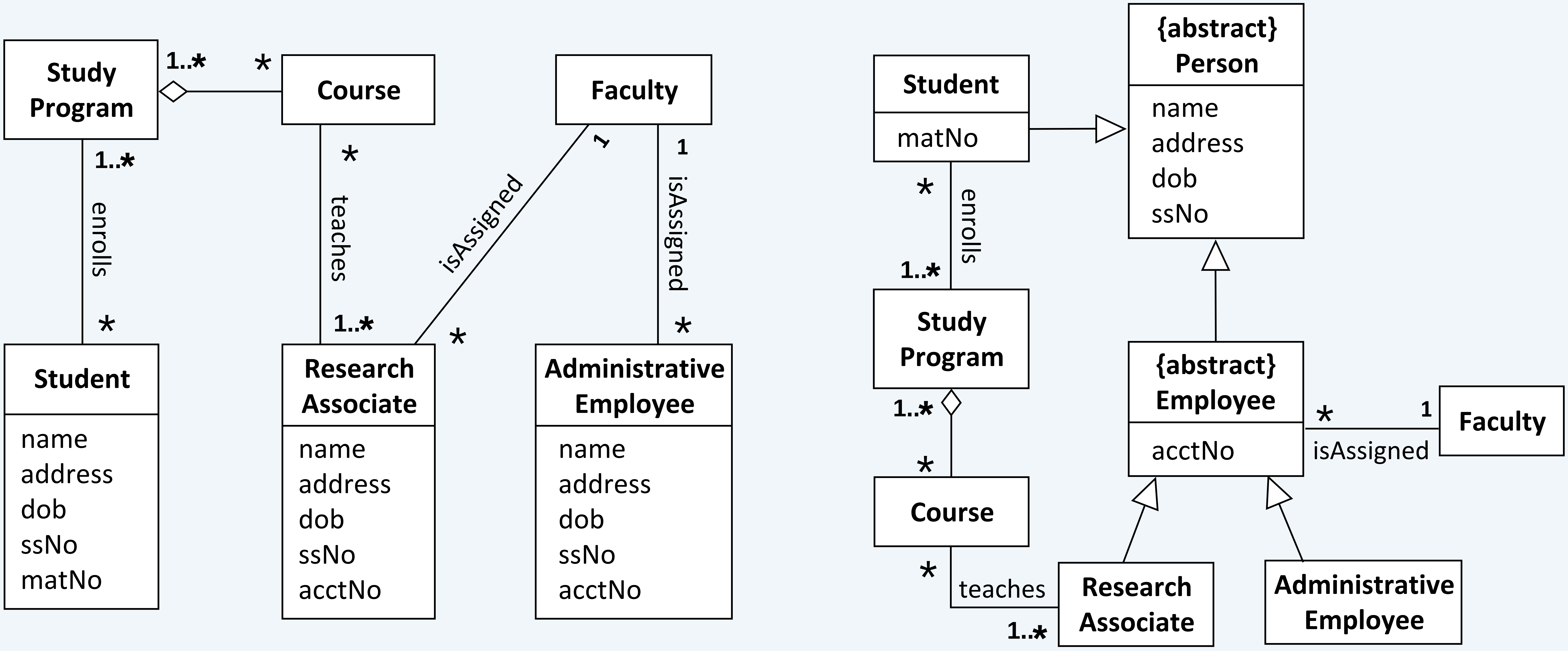
Abstract Classes (2/2)

- Examples:





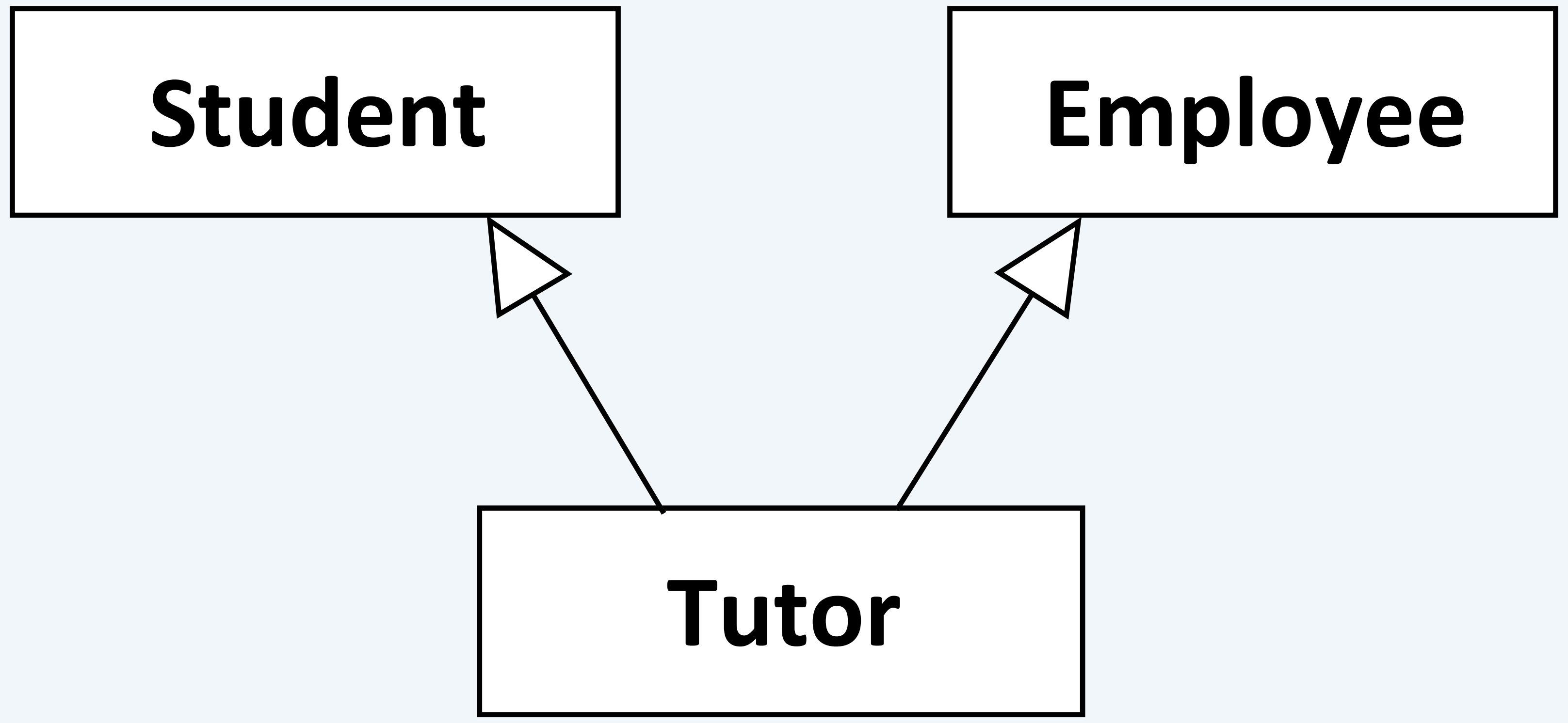
With and Without Generalization



Multiple inheritance

- Classes can also inherit from several classes

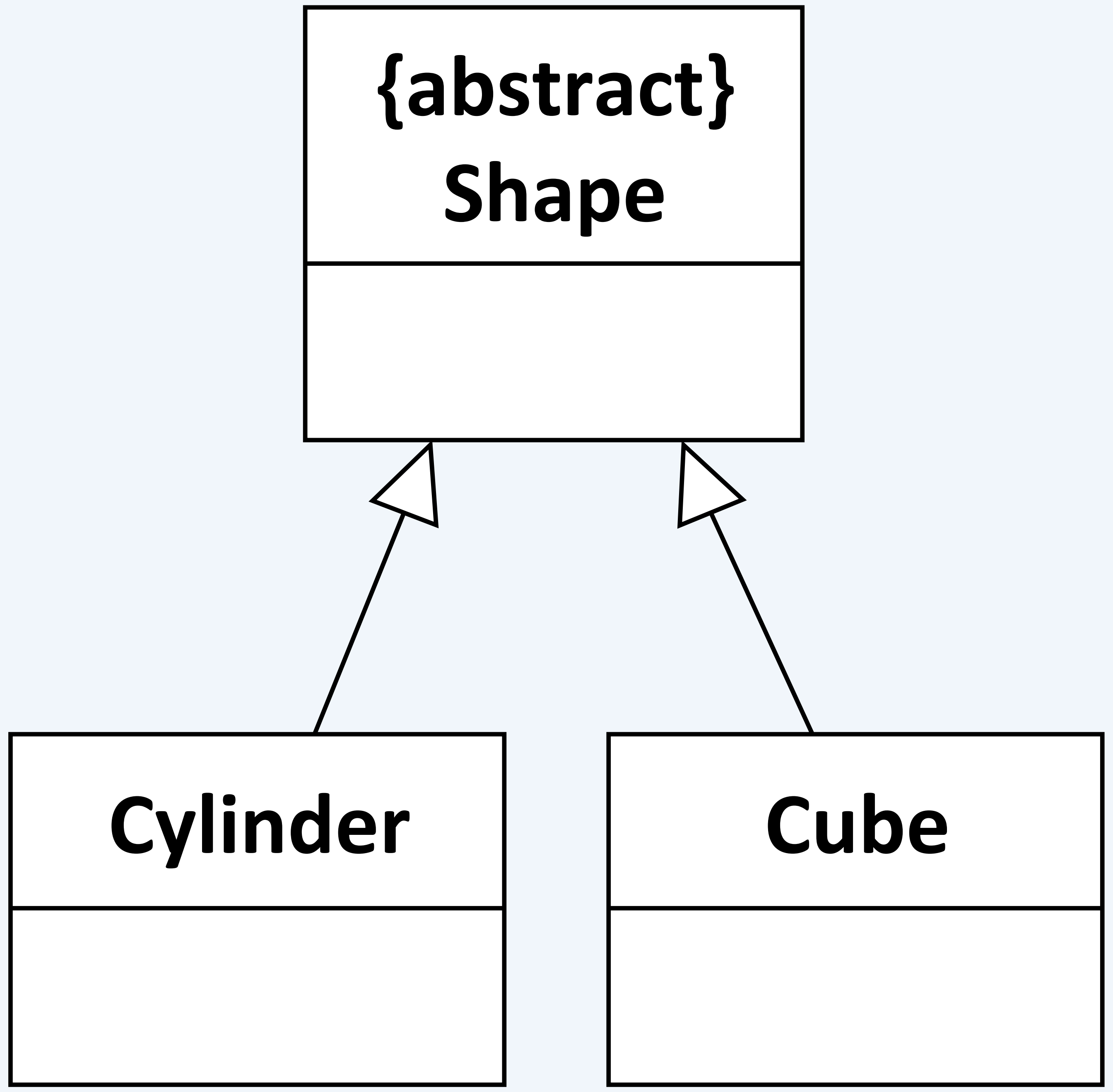
- Example:

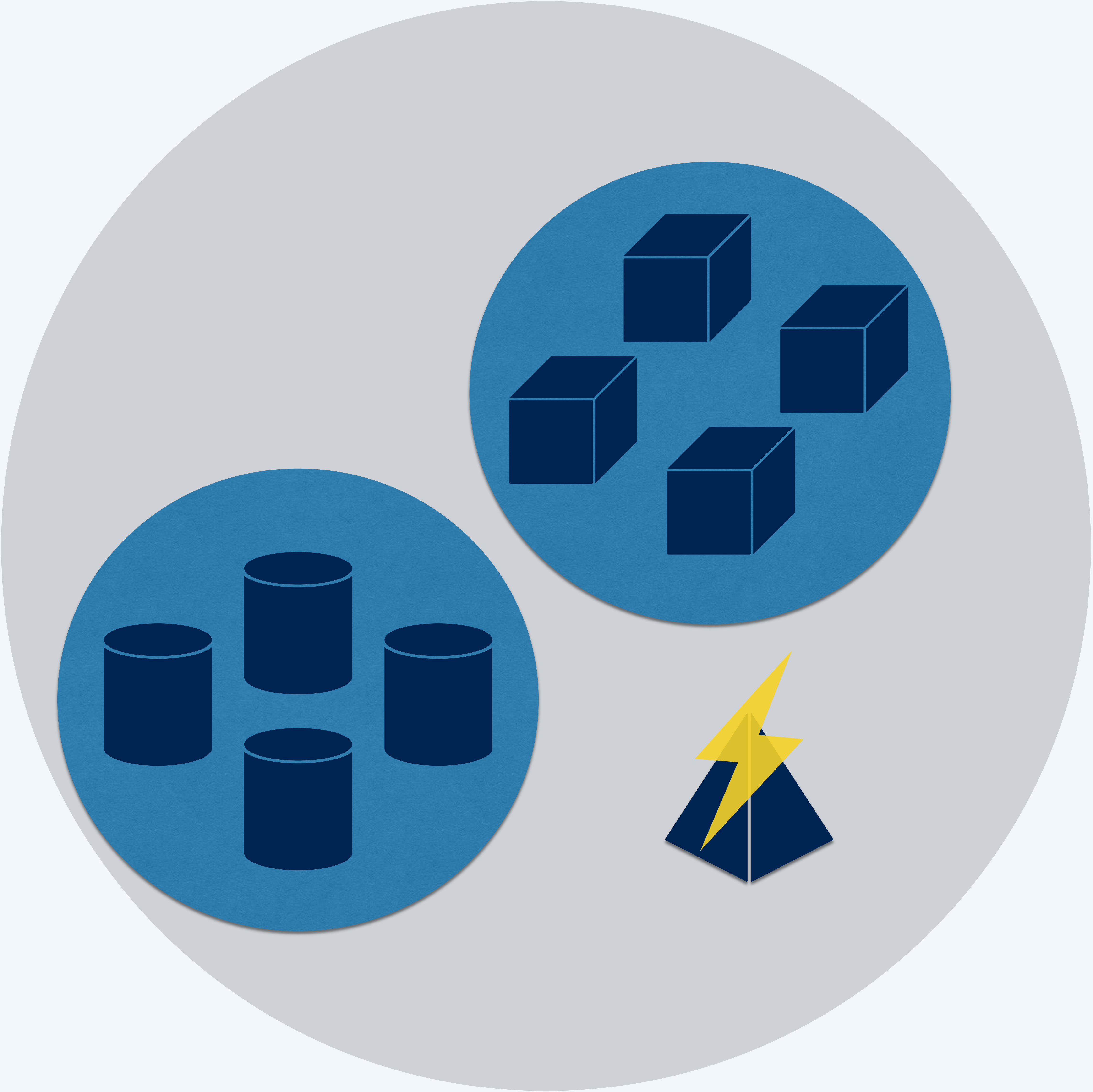


`Tutor` is an **`Employee`** and a **`Student`**

Generalization: Properties (1/2)

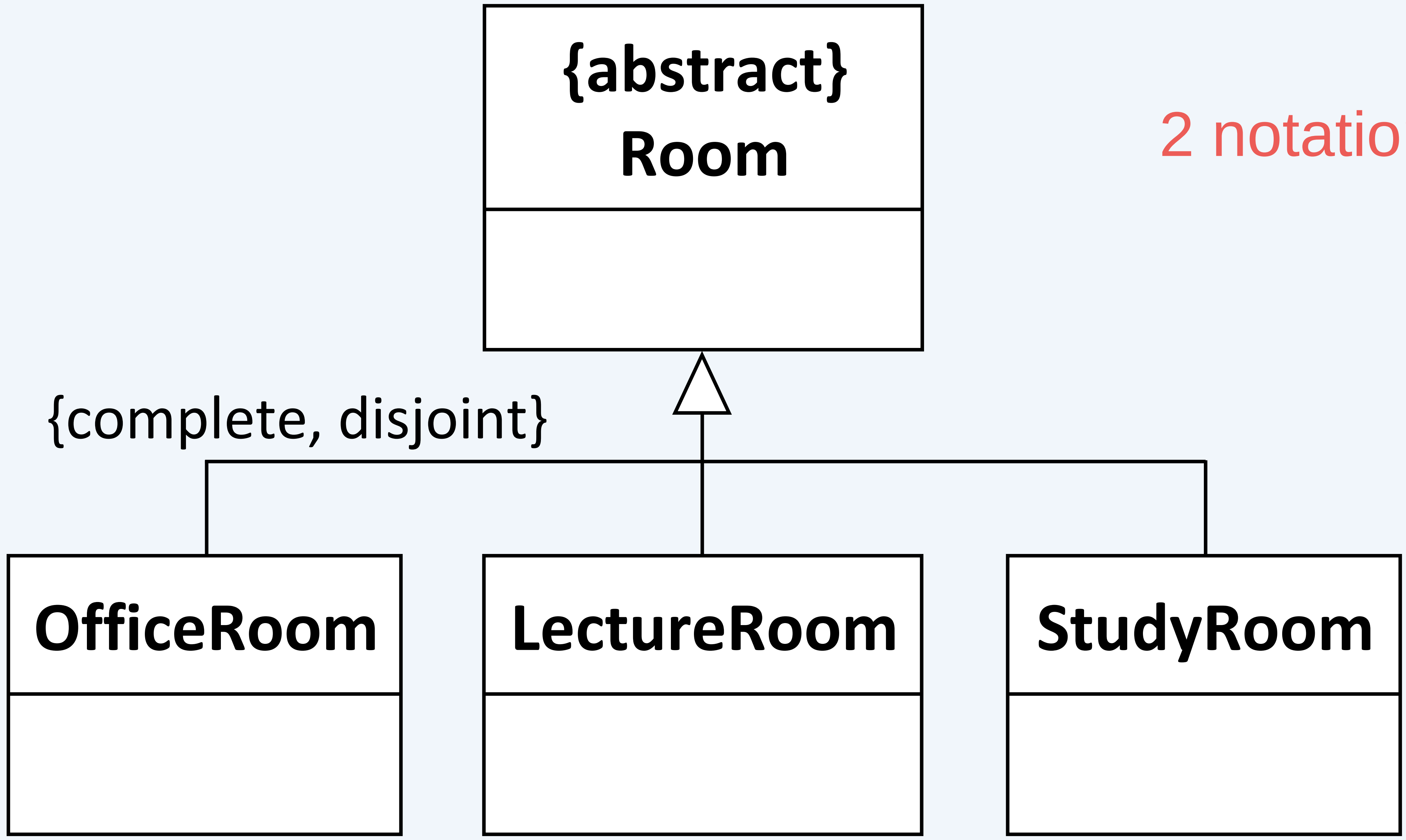
- Complete / incomplete division



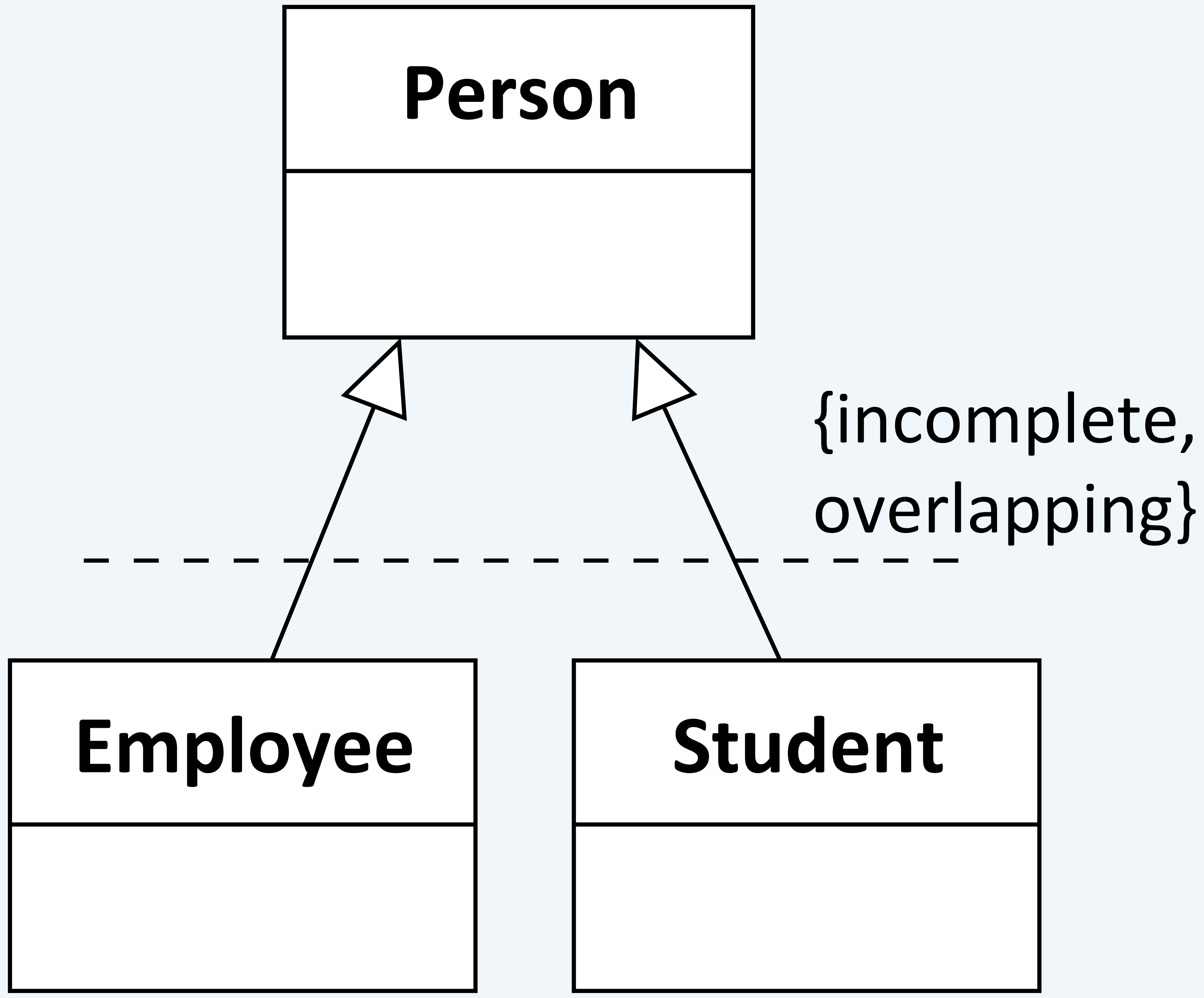


Generalization: Properties (2/2)

- A distinction can be made between
  - Incomplete / complete
  - Overlapping / disjoint



2 notations for >1 arrow



Structural Modeling

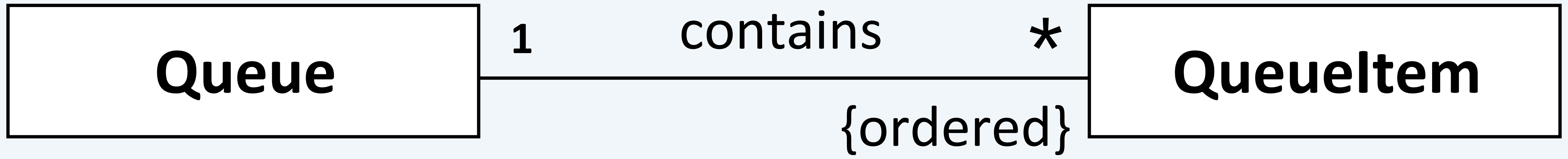
The Order and Uniqueness of Associations

Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Order and Uniqueness of Associations

- Order **`{ordered}`** is independent of attributes

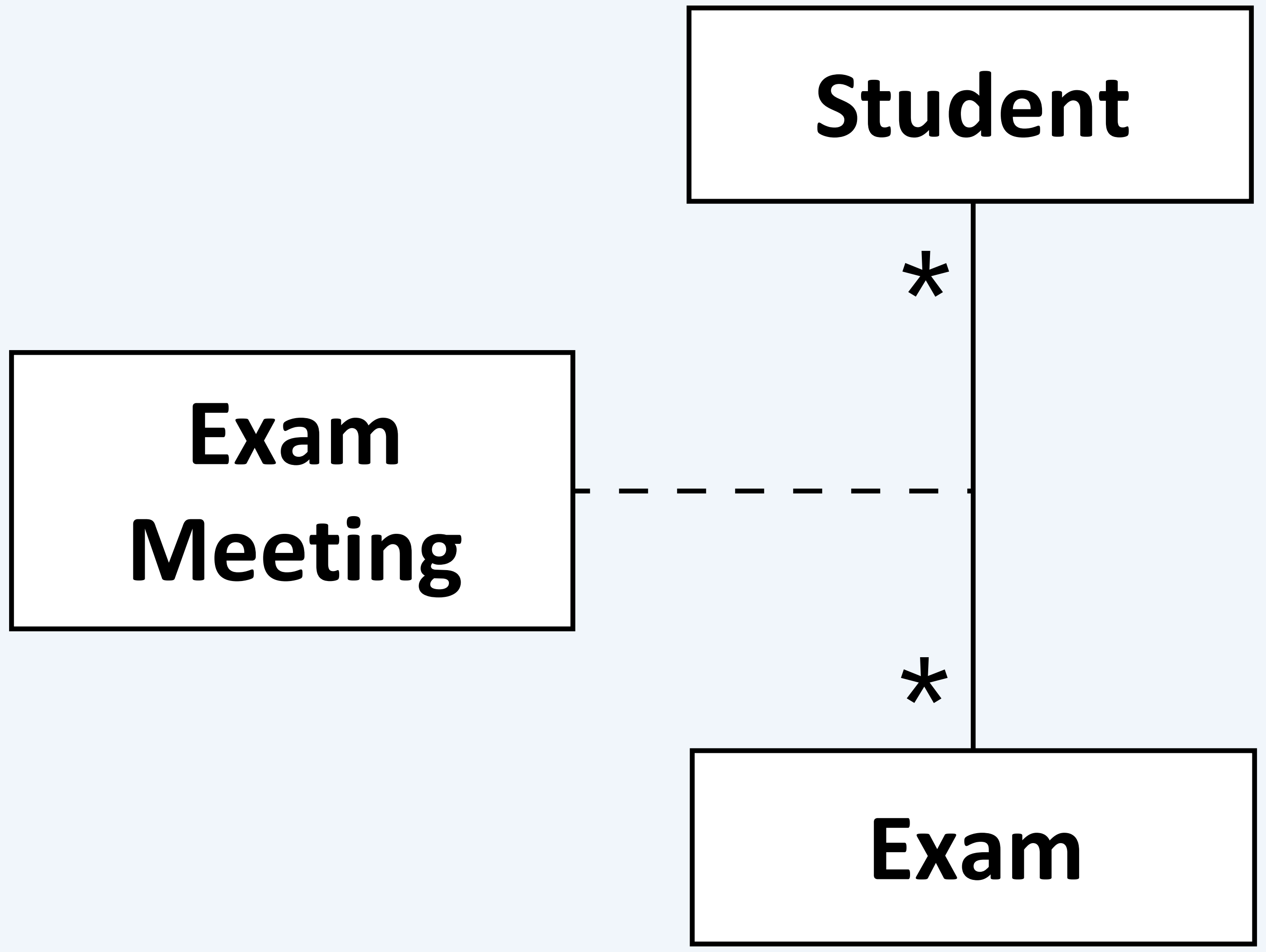


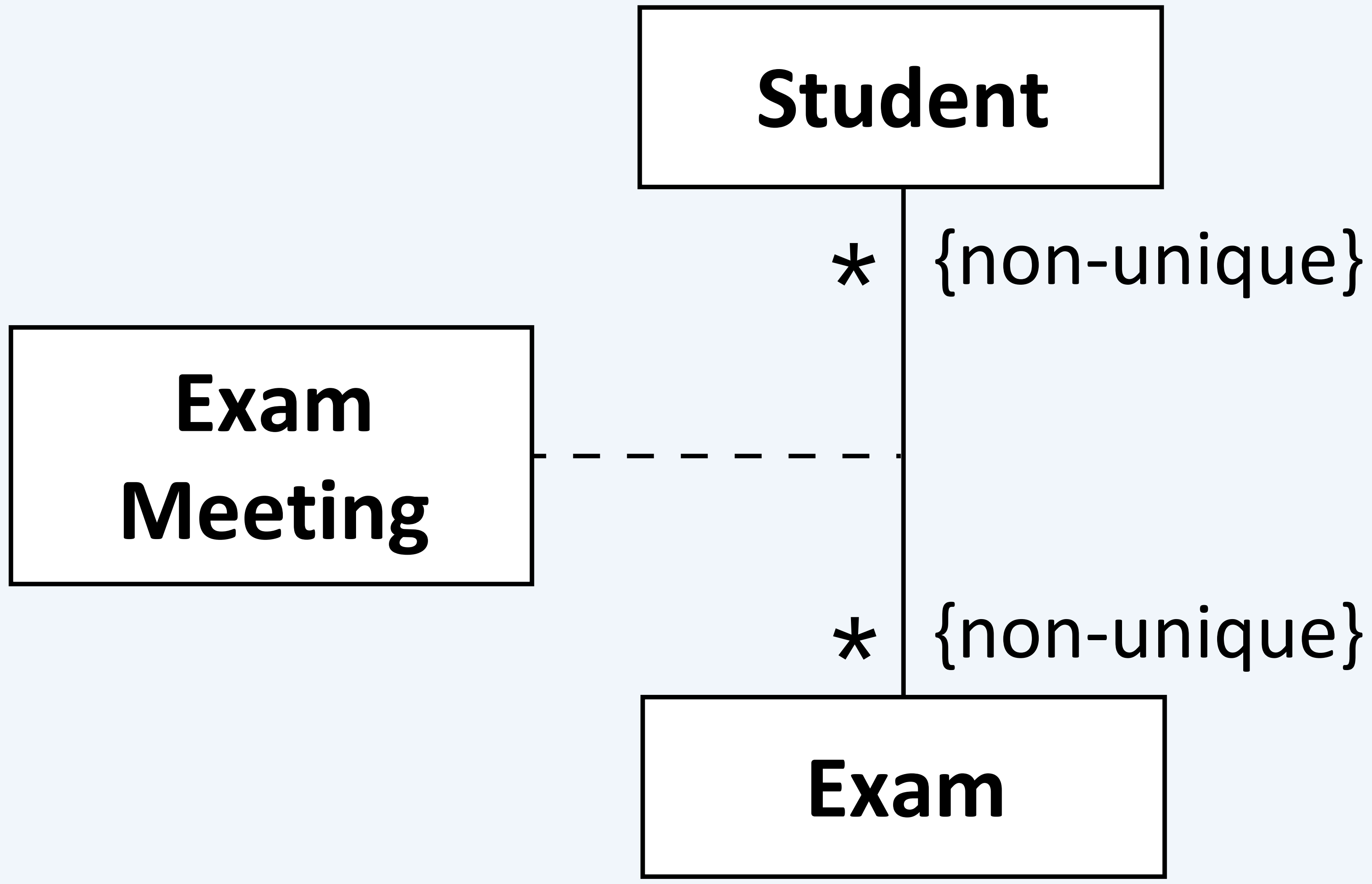
- Uniqueness
  - As in attributes by **`{unique}`** and **`{non-unique}`**
  - Combination with order **`{set}`**, **`{bag}`** and **`{sequence}`** or **`{seq}`**

Uniqueness	Order	Combination	Description
unique	unordered	set	Set (default value)
unique	ordered	orderedSet	Ordered set
nonunique	unordered	bag	Multi-set (= set with duplicates)
nonunique	ordered	sequence	Ordered set with duplicates (list)

Unique / Non-Unique (1/3)

- Default: no duplicates
- **`{non-unique}`**: duplicates allowed



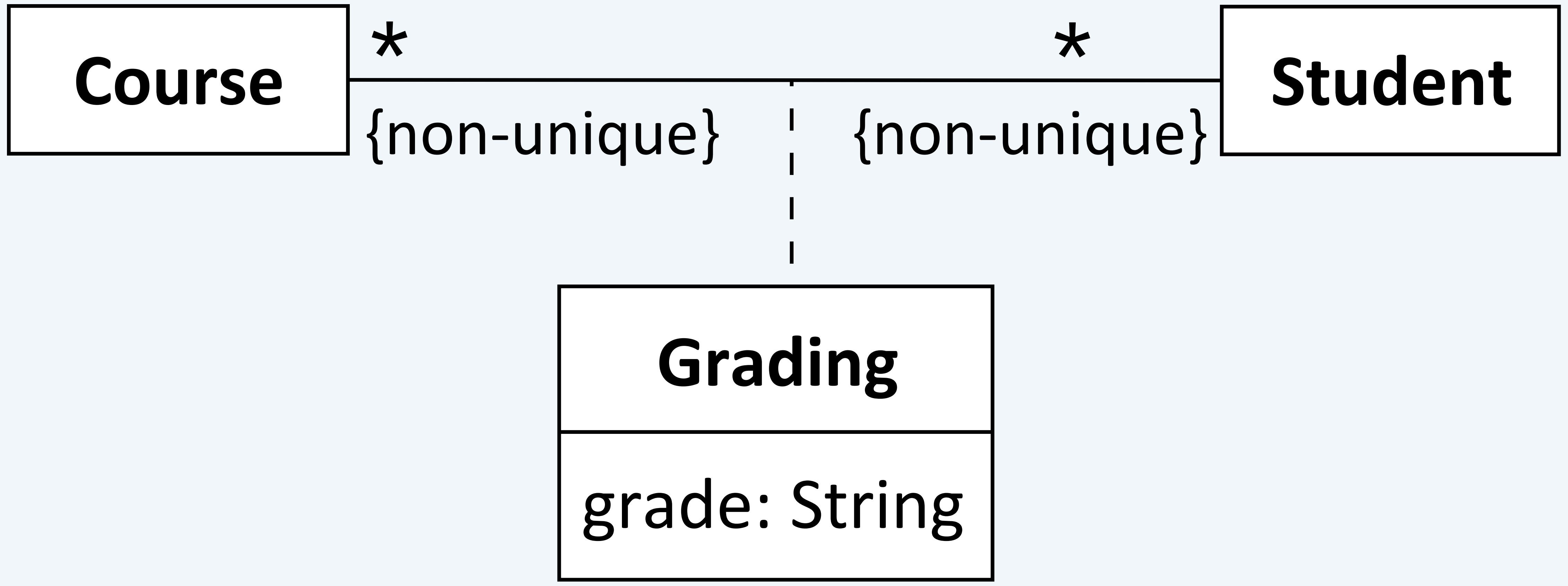


A **`Student`** can only have exactly one **`ExamMeeting`** for an **`Exam`**

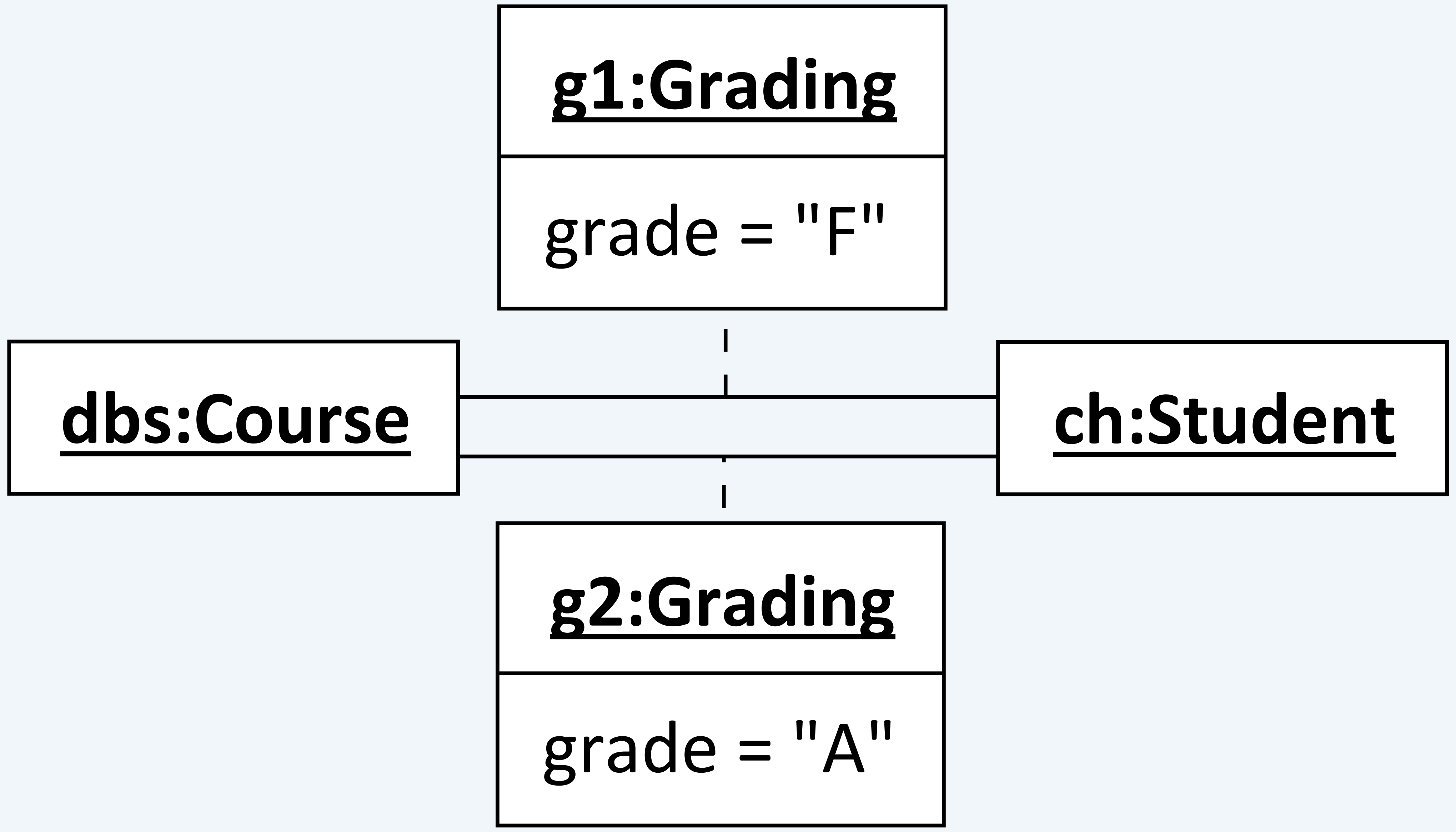
A **`Student`** can have more than one **`ExamMeeting`** for an **`Exam`**

Unique / Non-Unique (2/3)

Class diagram



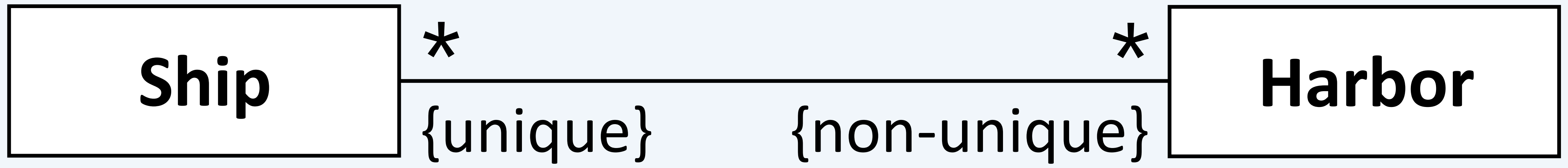
Object diagram



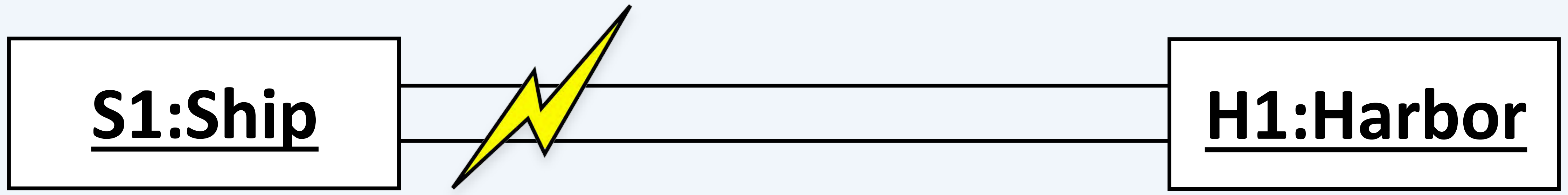
Unique / Non-Unique (3/3)

- It should be stored how often a certain **`Ship`** was in which **`Harbor`**

- It is not relevant that the **`Harbor`** knows how often a particular **`Ship`** has been there



- Mapping using links in the object diagram NOT possible



Structural Modeling
Summary Class Diagram Example

Christian Huemer und Marion Scholz

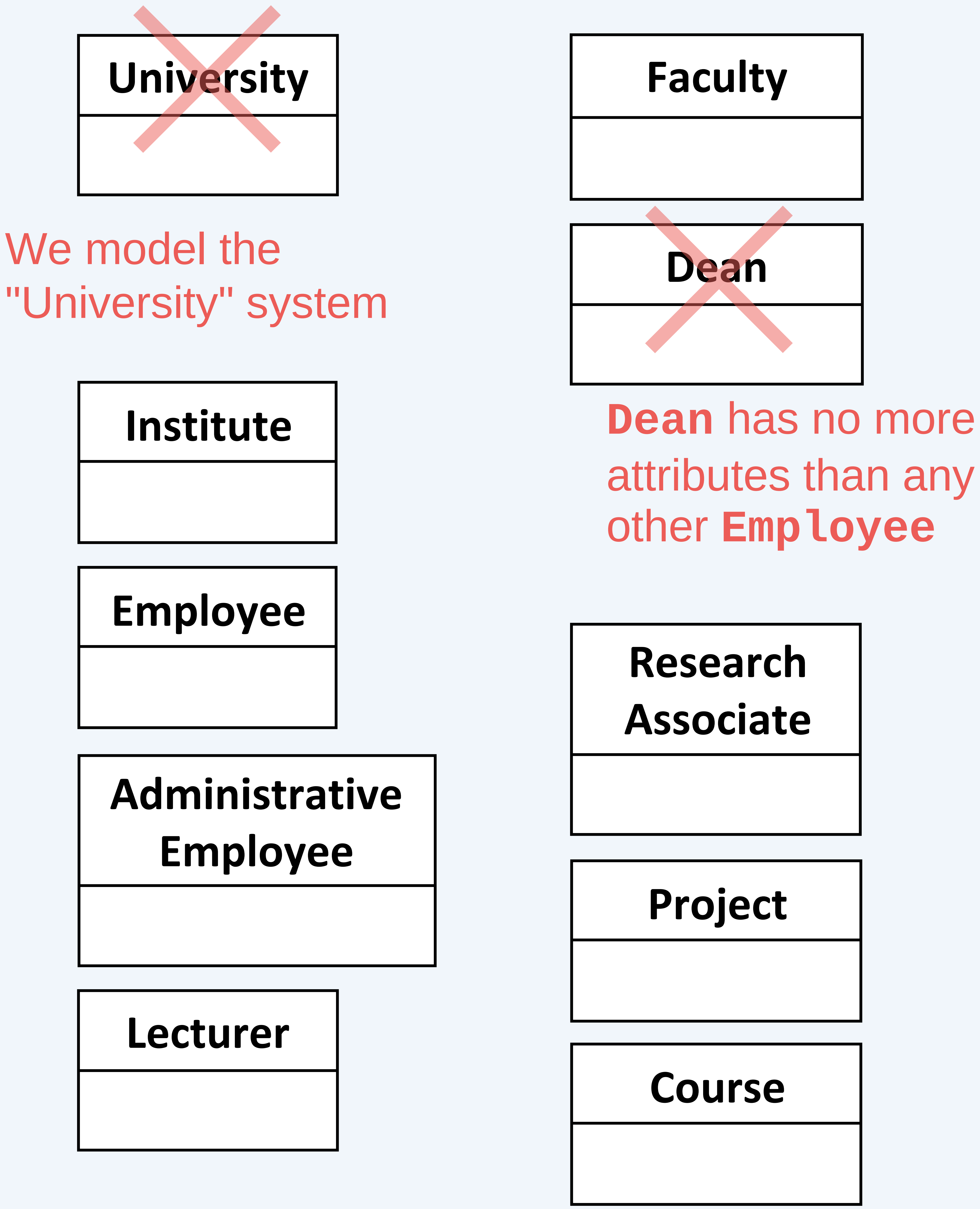
Presented by Nicholas Bzowski

Example - University Information System

- A university consists of several faculties, which are made up of different institutes. Each faculty and each institute has a name. An address is assigned to each institute.
- Each faculty is led by a dean who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name and an e-mail address. A distinction is made between research and administrative employees.
- Research associates (RA) are assigned to at least one institute. The field of study of each RA is given. In addition, RAs can be involved in projects for a certain number of hours, whereby the name, start and end dates of the projects are specified. Some RA hold courses. They are then called lecturers.
- Courses have a unique identification number (ID), a name and a weekly duration in hours.

Step 1: Identify classes

- A university consists of several faculties, which are made up of different institutes. Each faculty and each institute has a name. An address is assigned to each institute.
- Each faculty is led by a dean who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name and an e-mail address. A distinction is made between research and administrative employees.
- Research associates (RA) are assigned to at least one institute. The field of study of each RA is given. In addition, RAs can be involved in projects for a certain number of hours, whereby the name, start and end dates of the projects are specified. Some RA hold courses. They are then called lecturers.
- Courses have a unique identification number (ID), a name and a weekly duration in hours.

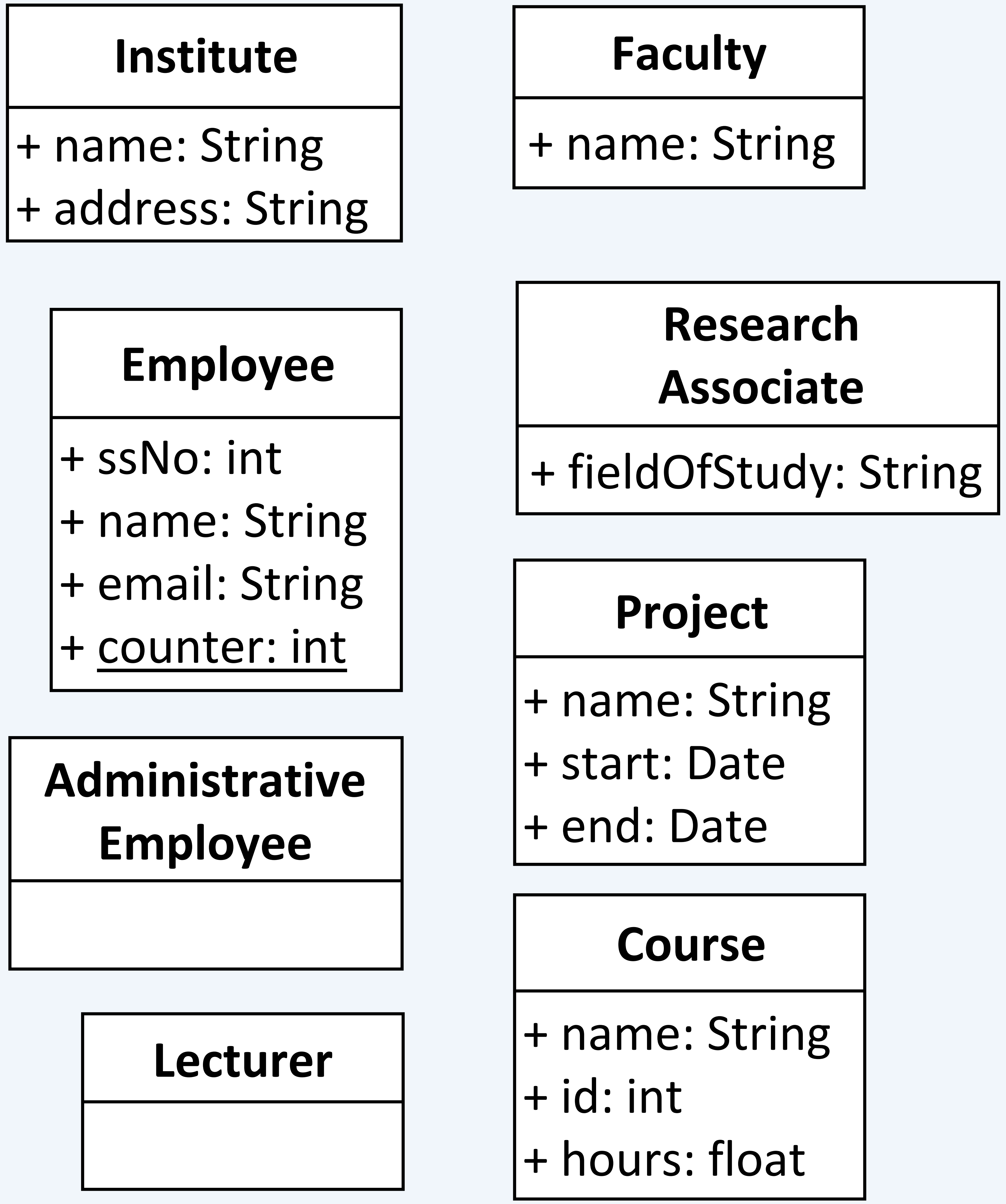


We model the "University" system

Dean has no more attributes than any other **Employee**

Step 2: Identify attributes

- A university consists of several faculties, which are made up of different institutes. Each faculty and each institute has a name. An address is assigned to each institute.
- Each faculty is led by a dean who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name and an e-mail address. A distinction is made between research and administrative employees.
- Research associates (RA) are assigned to at least one institute. The field of study of each RA is given. In addition, RAs can be involved in projects for a certain number of hours, whereby the name, start and end dates of the projects are specified. Some RA hold courses. They are then called lecturers.
- Courses have a unique identification number (ID), a name and a weekly duration in hours.



Institute
+ name: String
+ address: String

Faculty
+ name: String

Employee
+ ssNo: int
+ name: String
+ email: String
+ counter: int

Research Associate
+ fieldOfStudy: String

Administrative Employee

Project
+ name: String
+ start: Date
+ end: Date

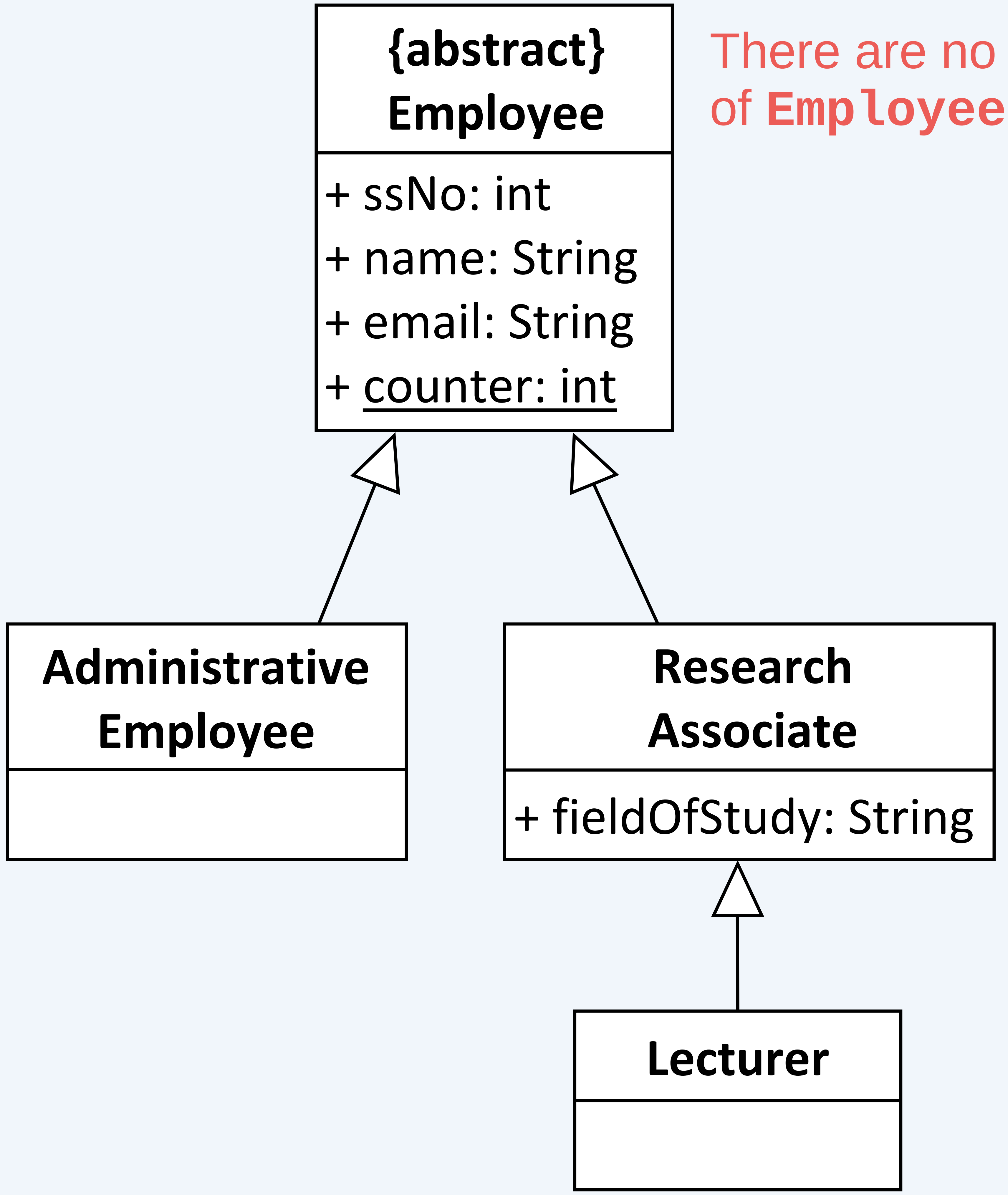
Lecturer

Course
+ name: String
+ id: int
+ hours: float

Step 3: Identify relationships (1/6)

- Three types of relationships:
  - Generalization
  - Association
  - Aggregation

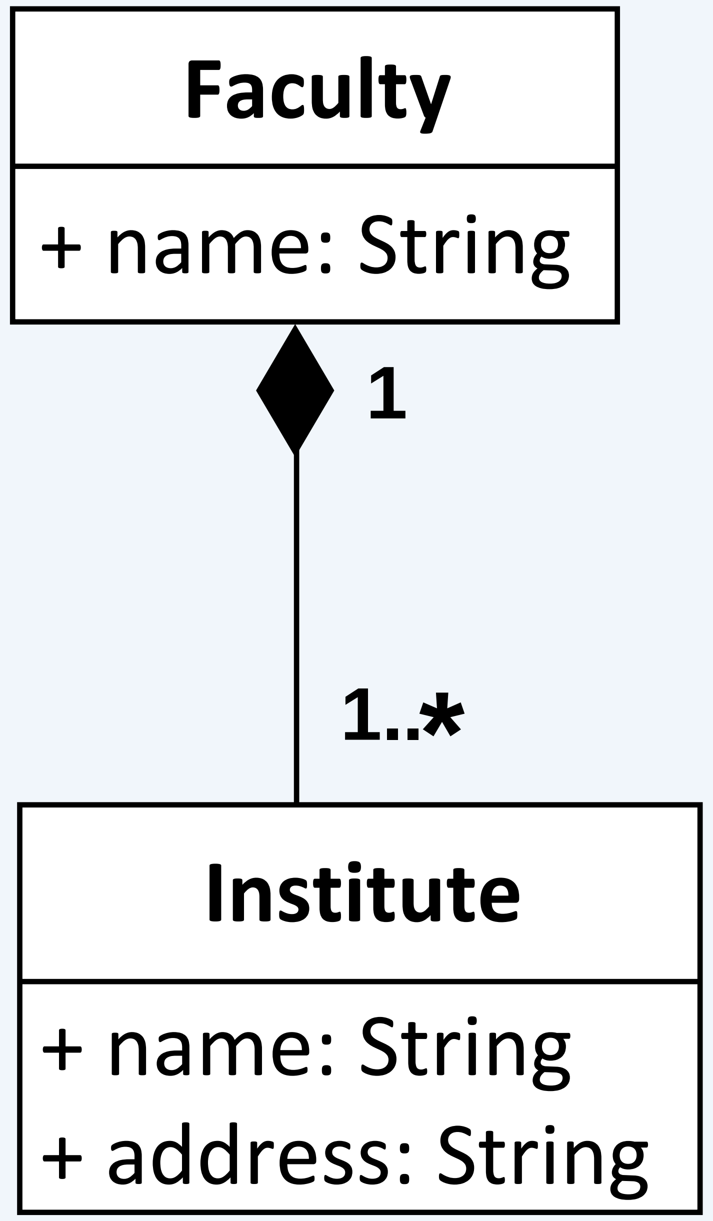
- Evidence of generalization
  - "A distinction is made between research and administrative employees."
  - "Some research associates hold courses. They are then called lecturers."



There are no other types of **`Employee`**s

Step 3: Identify relationships (2/6)

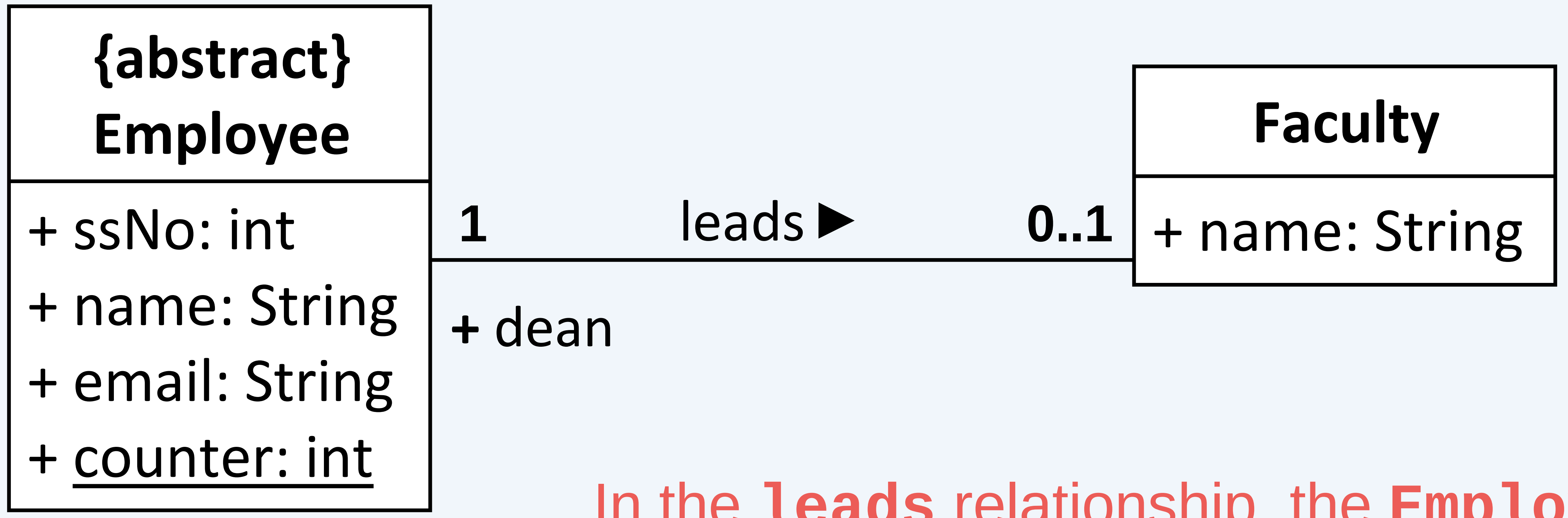
- "A university consists of several faculties, which are made up of different institutes."



Composition to reflect the existential dependency

Step 3: Identify relationships (3/6)

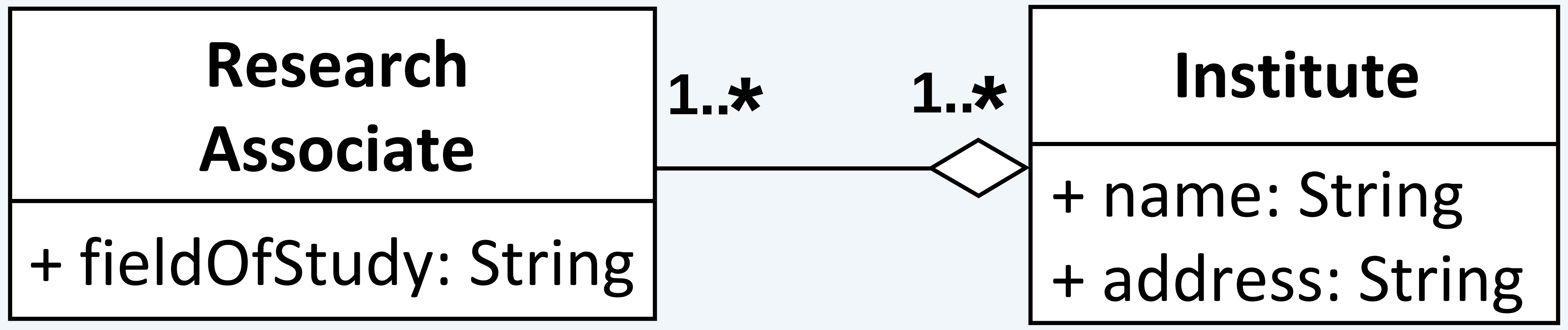
- "Each faculty is led by a dean who is an employee of the university."



In the **`leads`** relationship, the **`Employee`** takes on the role of the **`dean`**.

Step 3: Identify relationships (4/6)

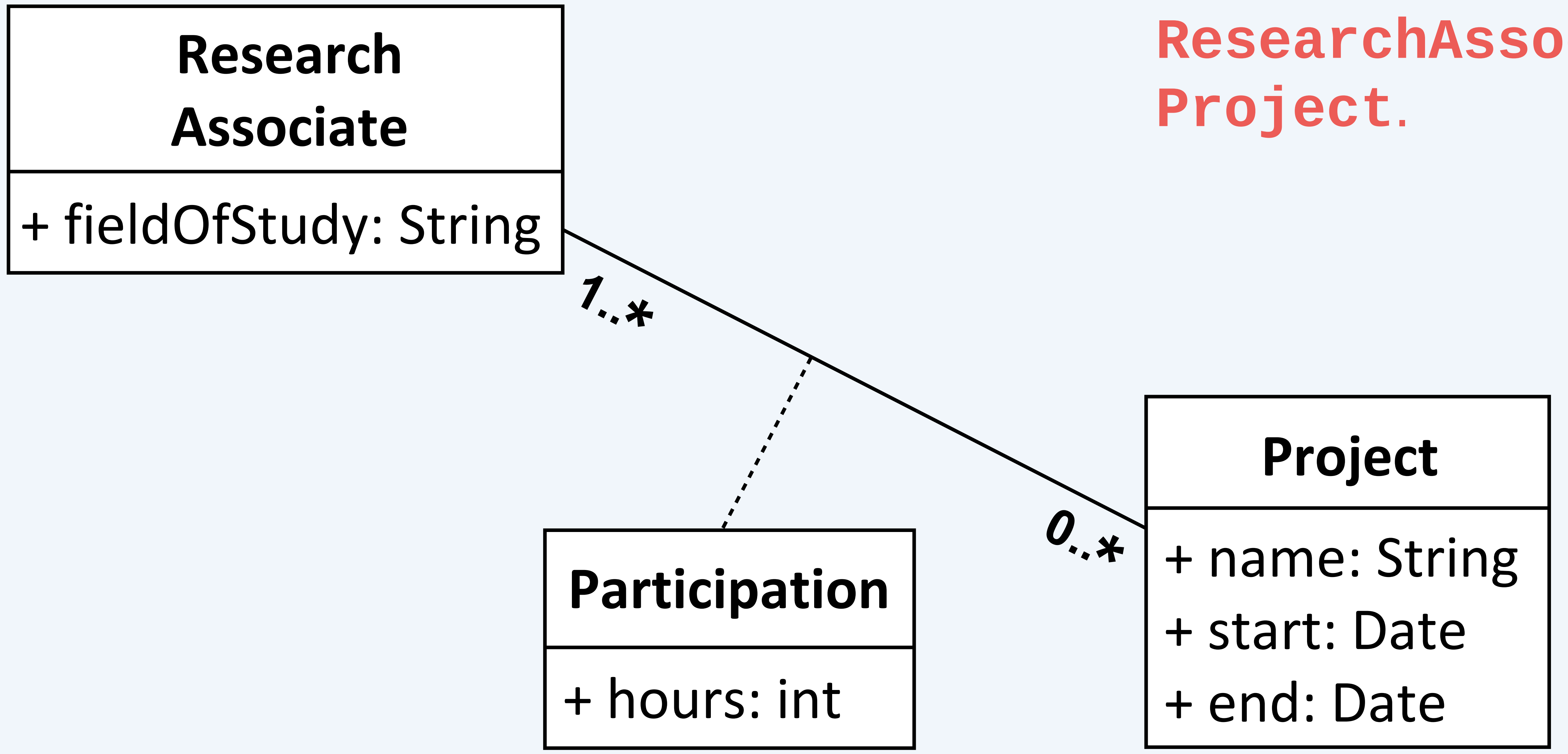
- "Research associates (RA) are assigned to at least one institute."



Weak aggregation to show that **`ResearchAssociates`** are part of an **`Institute`** but there is no dependency.

Step 3: Identify relationships (5/6)

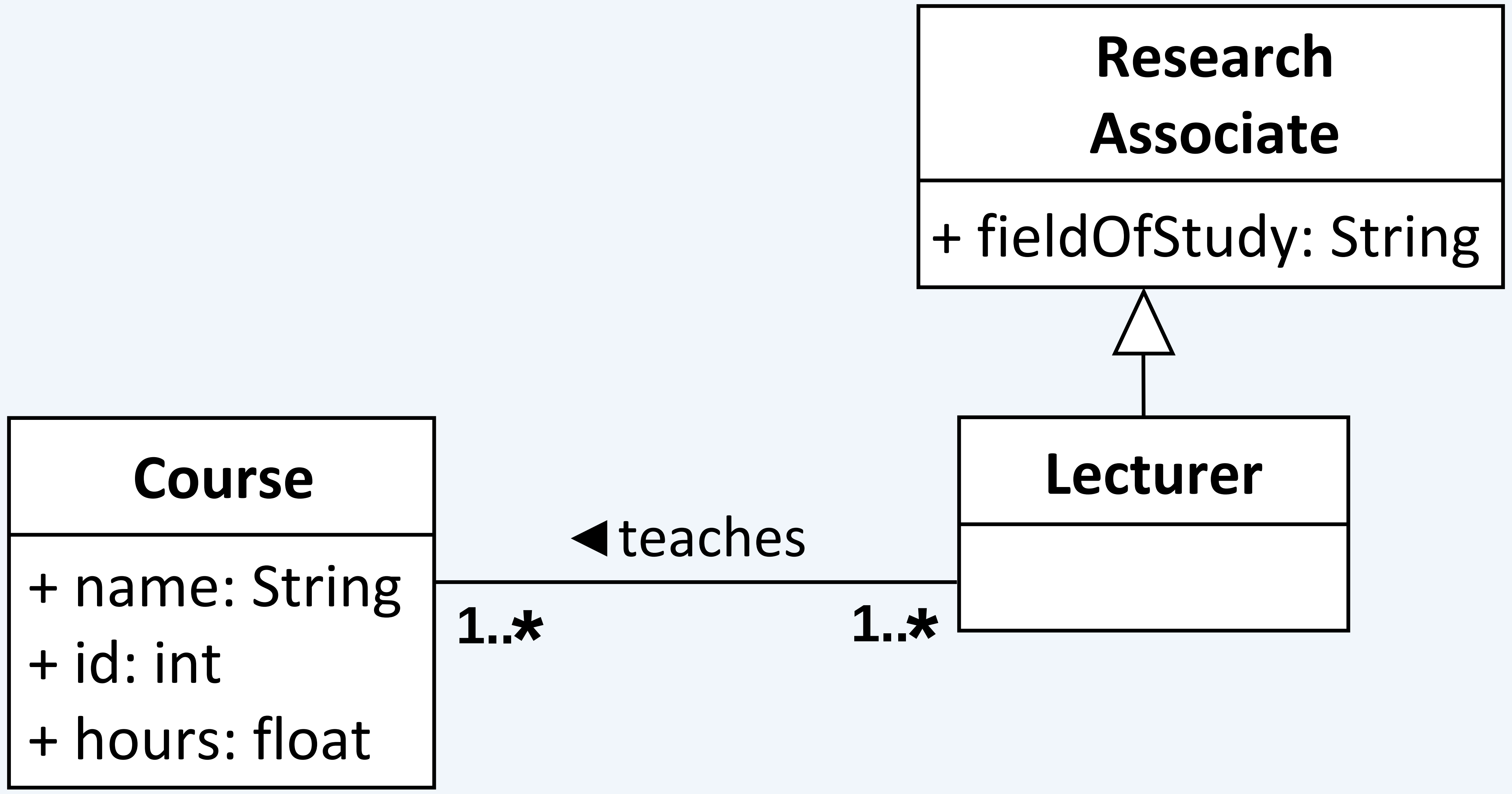
- "In addition, research associates can be involved in projects for a certain number of hours."



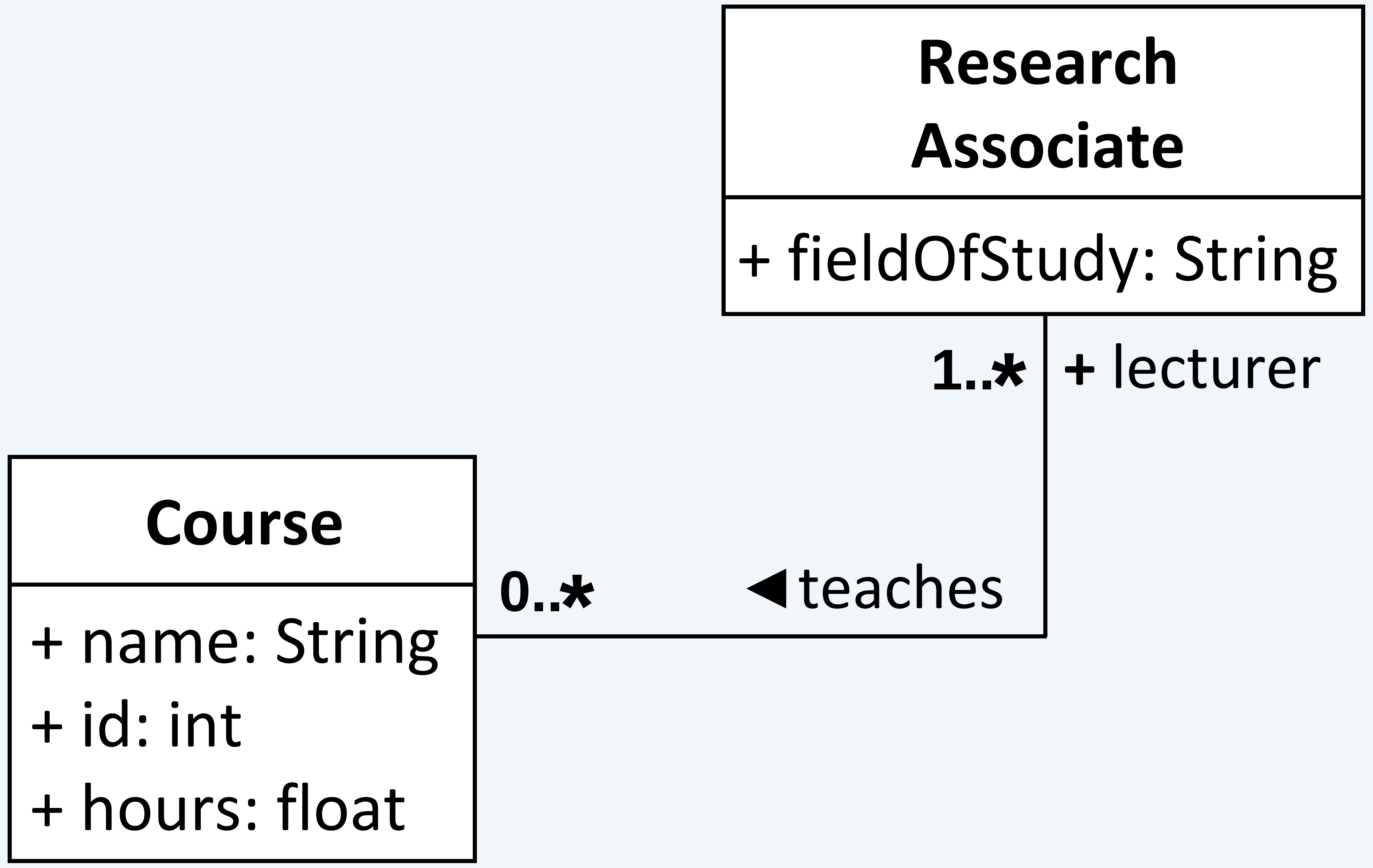
The association class makes it possible to store the number of hours that each individual **`ResearchAssociate`** is involved in a single **`Project`**.

Step 3: Identify relationships (6/6)

- "Some research associates hold courses. They are then called lecturers."



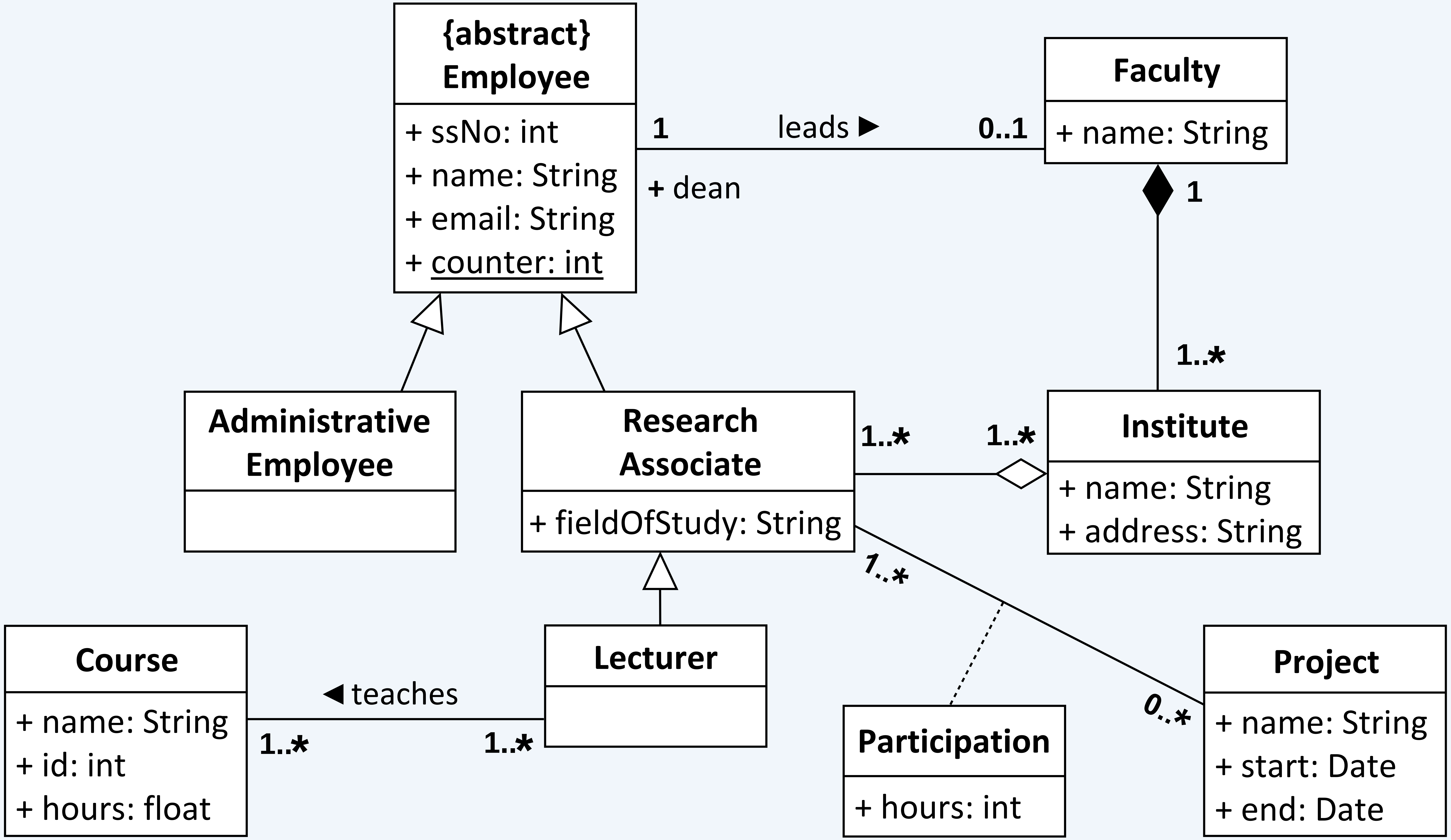
or



`Lecturer` inherits all the properties and relationships of **`ResearchAssociate`**.
A **`Lecturer`** also has a relationship **`teaches`** to **`Course`**.

`ResearchAssociate` has a relationship **`teaches`** to **`Course`**, in the role **`lecturer`**.

...and now everything together



Structural Modeling

Data Types

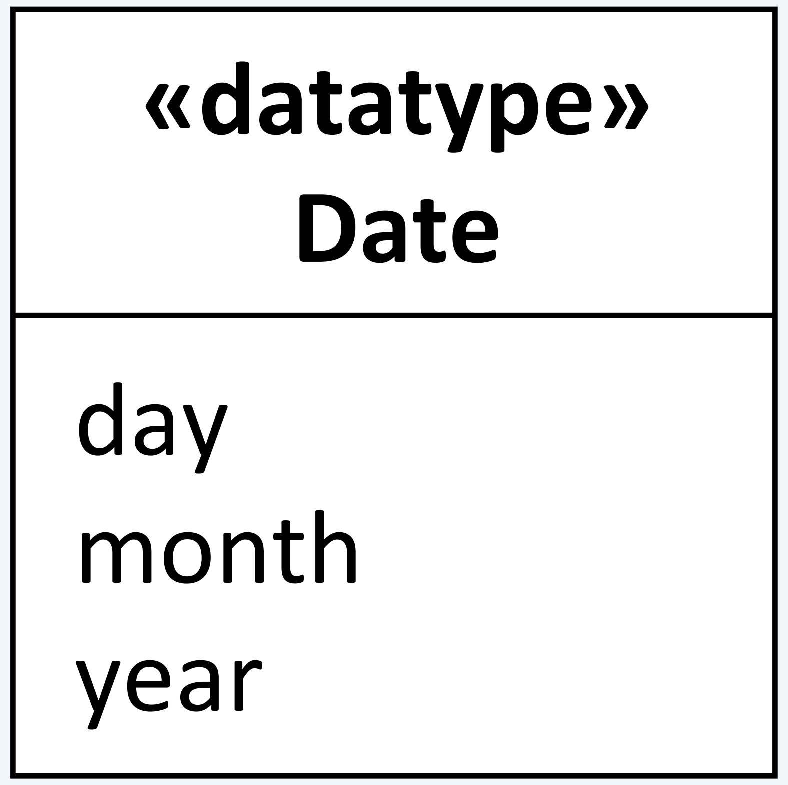
Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

Data Types in UML

- Instances of a data type have no identity
  - Objects: Instances of a class
  - Values: Instances of a data type (e.g. the number 2)
- Notation: Rectangle with keyword **`«datatype»`** in the first compartment

- Forms of data types:
  - Primitive data types
  - Data types with attributes (and operations)
  - Enumeration types



«datatype»
Date

day
month
year

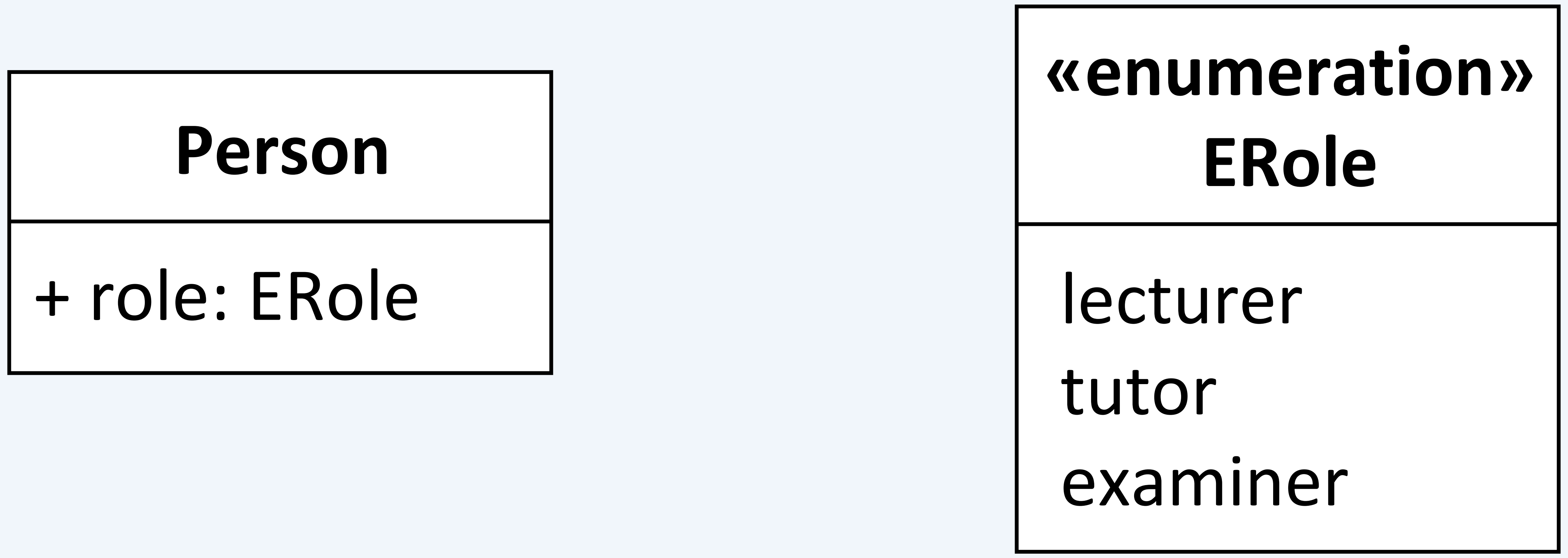
Forms of Data Types: Primitive Data Types

- Primitive data types: Data types without internal structure
- Primitive data types pre-defined by UML:
  - Boolean
  - Integer
  - UnlimitedNatural
  - String

- Primitive data types can also be defined:
  - Keyword **`«primitive»`**

Forms of Data Types: Enumeration Types

- Definition of the value range by enumerating the possible values
- Notation: Class symbol with keyword **`«enumeration»`**
- Possible characteristics are specified by user-defined identifiers (literals)



Structural Modeling

The Translation to Java

Christian Huemer und Marion Scholz

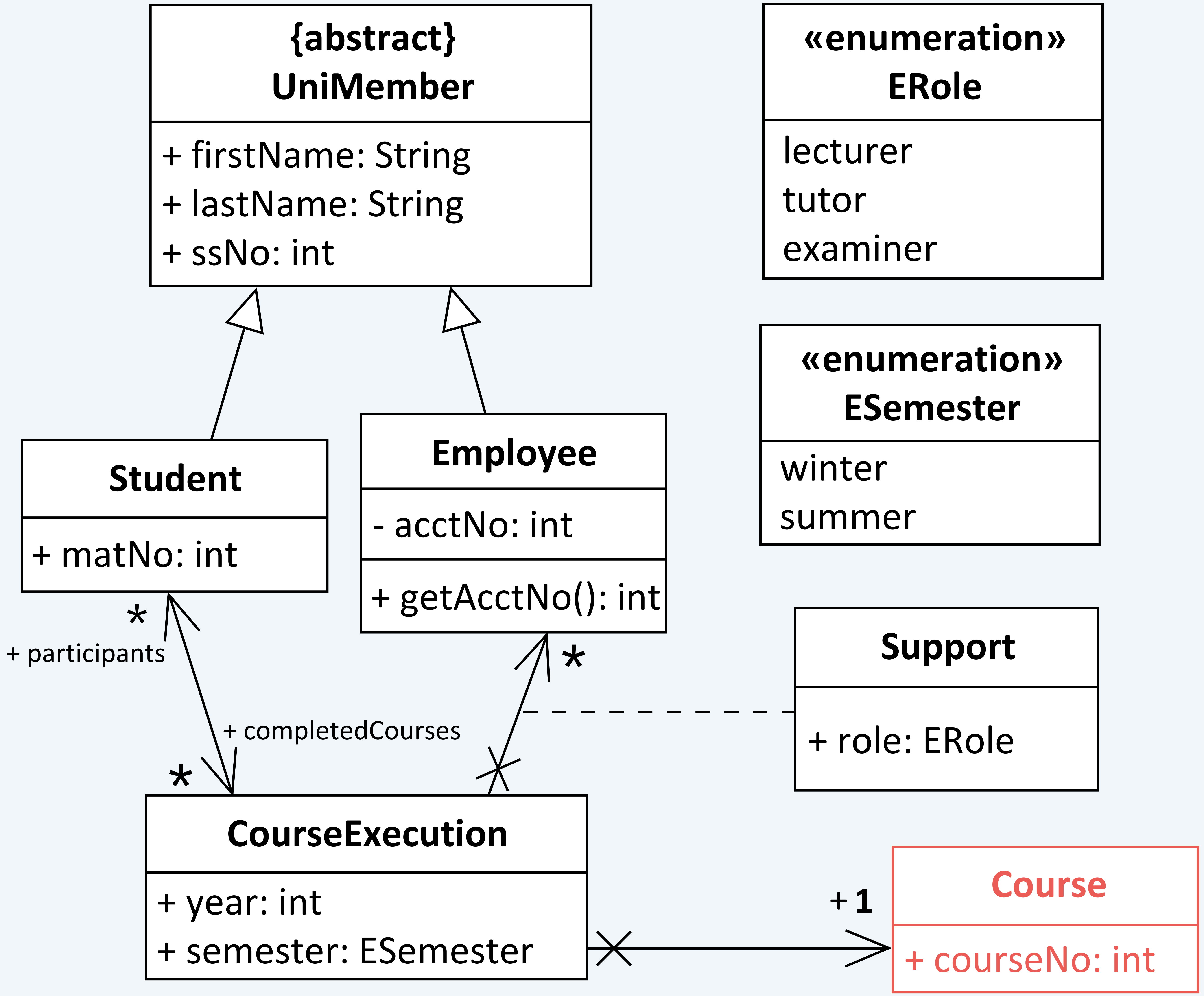
Presented by Nicholas Bzowski

Code Generation

- Class diagrams are often created with the intention of implementing the modeled elements in an object-oriented programming language

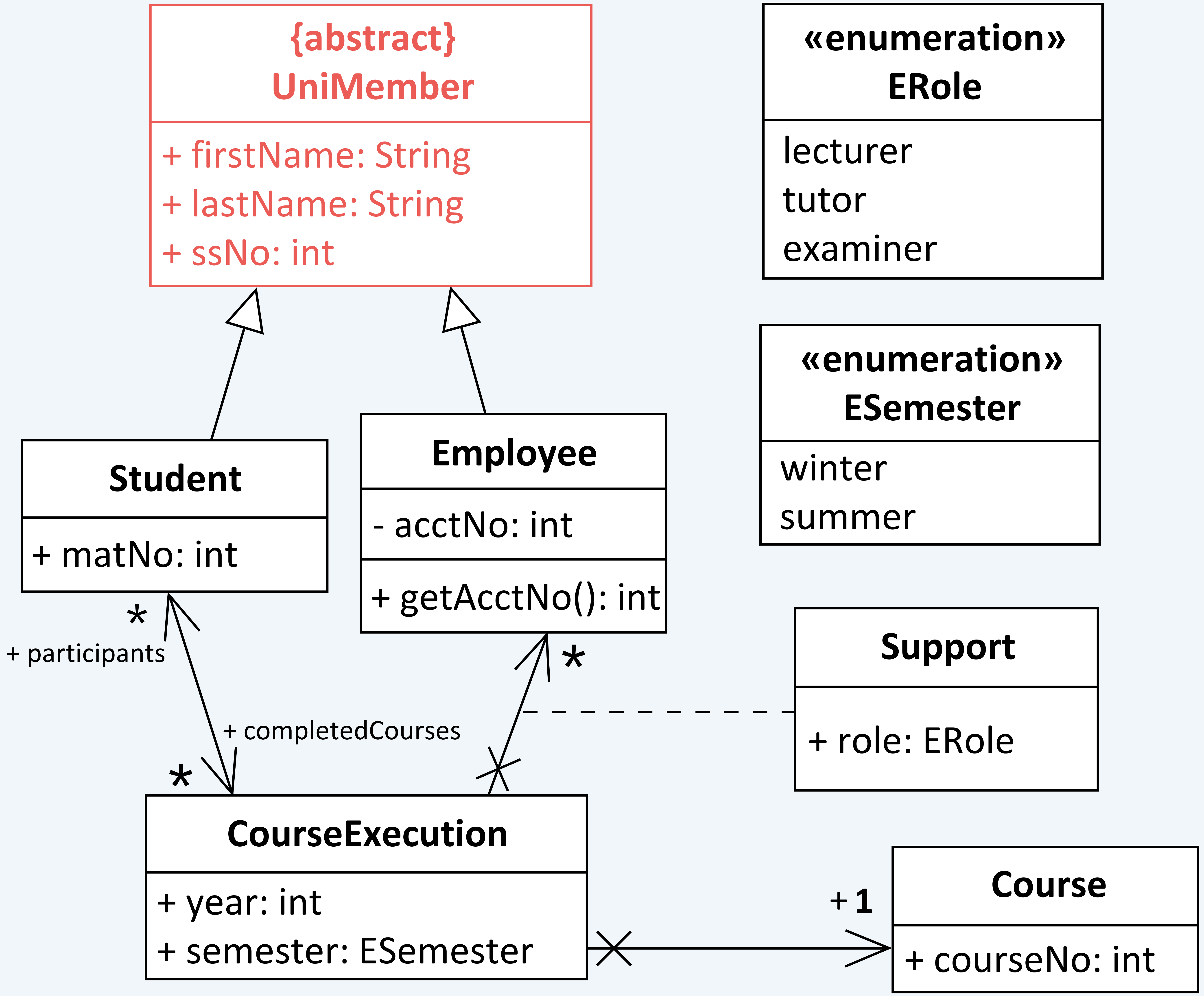
- In many cases, the translation can be carried out automatically and requires little manual intervention

Translation to Java – Example (1/6)



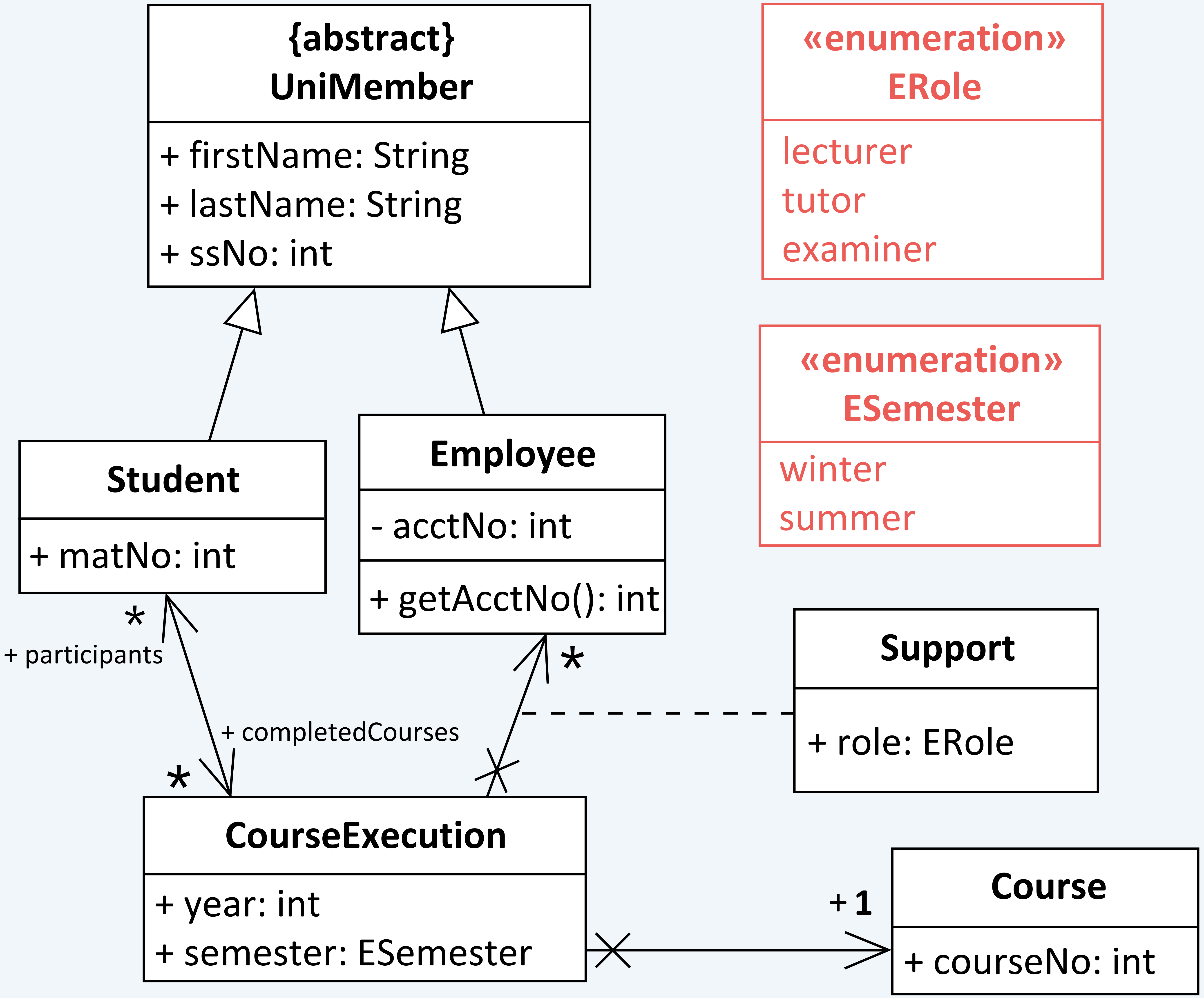
```
class Course {
  public int courseNo;
}
```

Translation to Java – Example (2/6)



```
abstract class UniMember {
  public String firstName;
  public String lastName;
  public int ssNo;
}
```

Translation to Java – Example (3/6)



{abstract}
UniMember

+ firstName: String
+ lastName: String
+ ssNo: int

«enumeration»
ERole

lecturer
tutor
examiner

«enumeration»
ESemester

winter
summer

Student

+ matNo: int

Employee

- acctNo: int

+ getAcctNo(): int

*
+ participants
+ completedCourses
*
*

Support

+ role: ERole

CourseExecution

+ year: int
+ semester: ESemester

+ 1

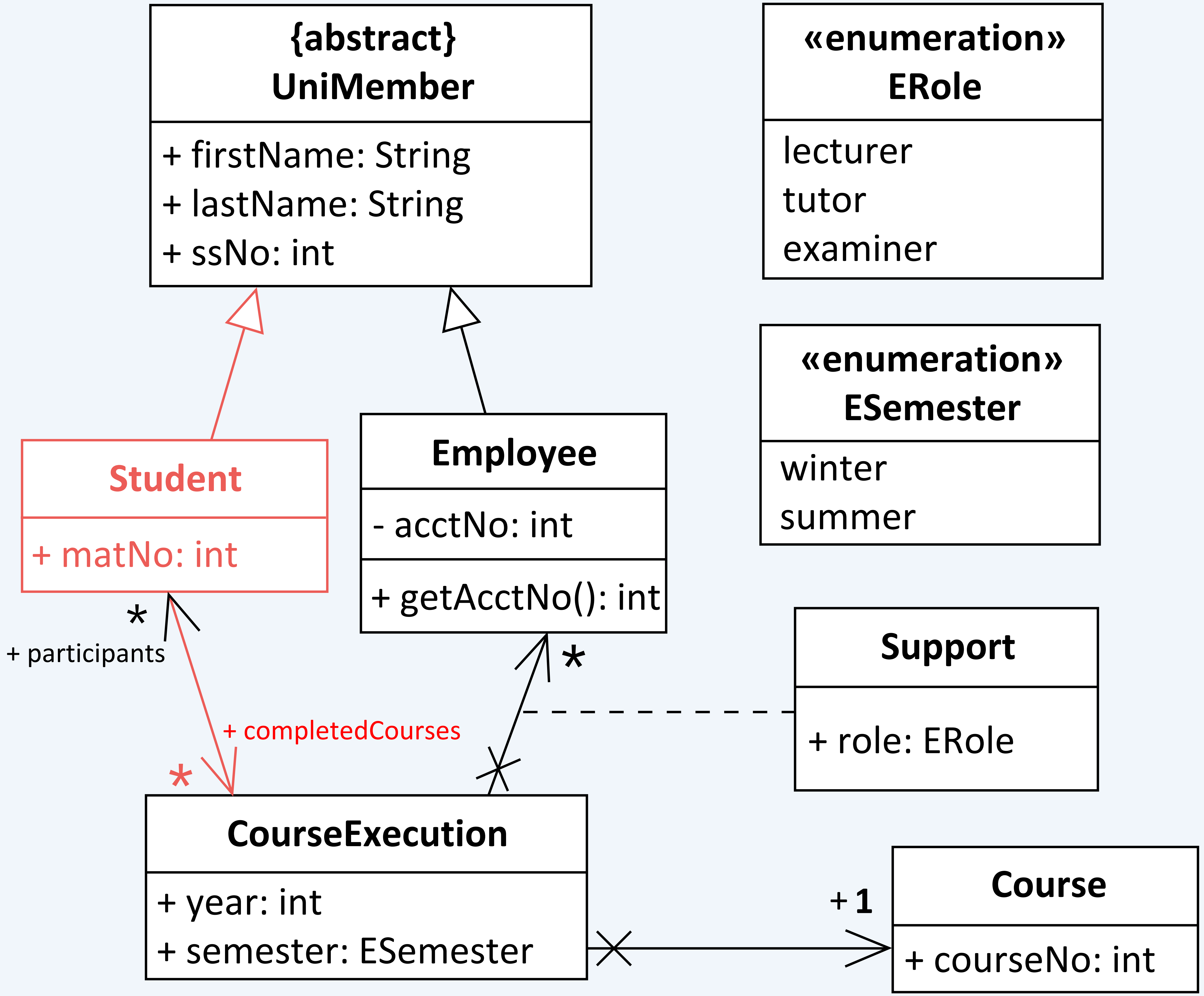
Course

+ courseNo: int

```
Enumeration ERole {
  lecturer,
  tutor,
  examiner
}
```

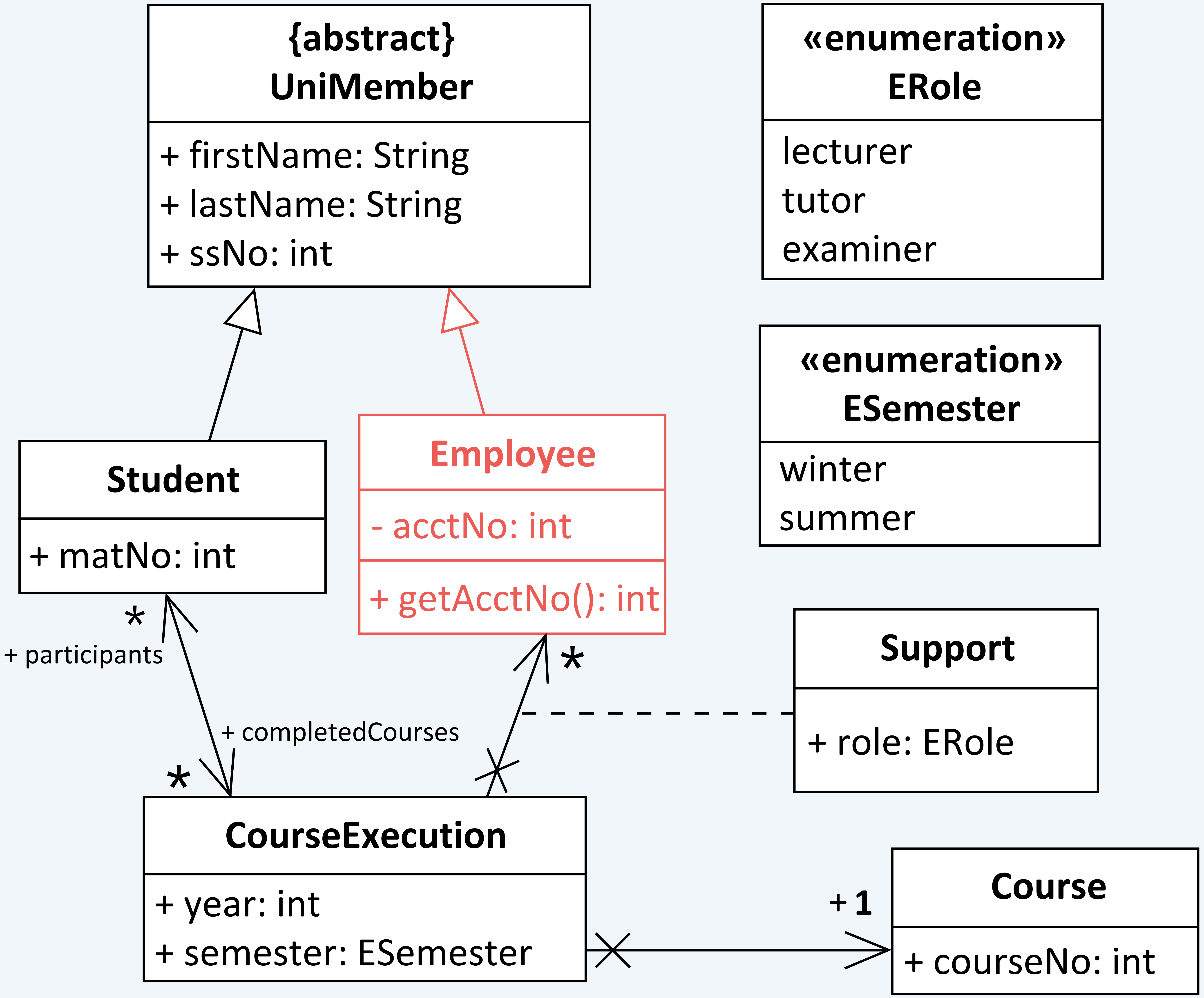
```
Enumeration ESemester {
  winter,
  summer
}
```

Translation to Java – Example (4/6)



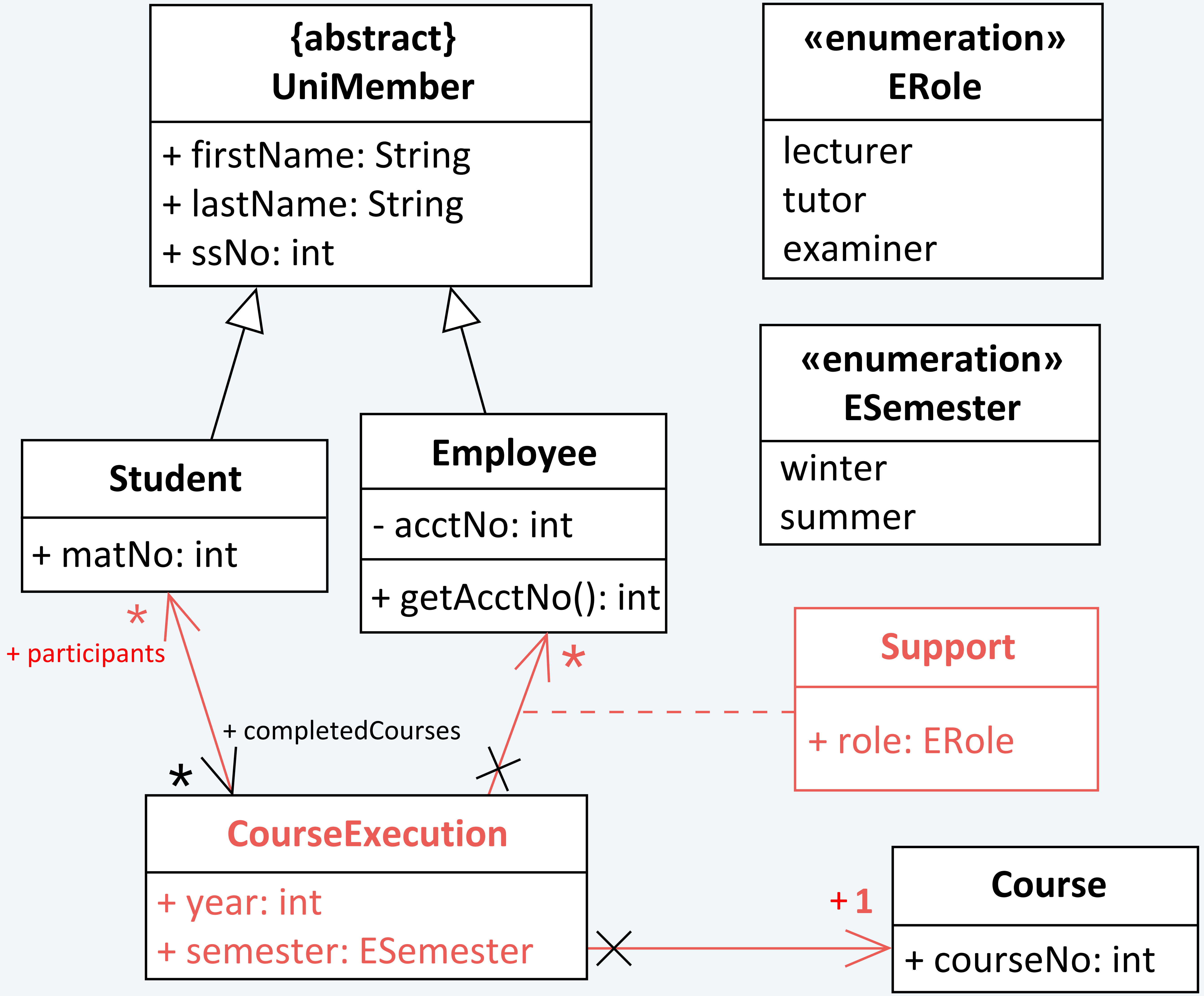
```
class Student extends UniMember {
  public int matNo;
  public CourseExecution []
                completedCourses;
}
```

Translation to Java – Example (5/6)



```
class Employee extends UniMember {
  private int acctNo;

  public int getAcctNo () {
    return acctNo;
  }
}
```

Translation to Java – Example (6/6)



```
class CourseExecution {
  public int year;
  public ESemester semester;
  public Student [] participants;
  public Course the_course;
  public Hashtable support;
    // Key: Employee
    // Value: ERole
}
```

Translation to Java: Summary

- Classes → Java Classes
- Attributes → Instance variables
- Operations → Methods
- Class variables and operations: **`static`**
- Associations
  - Inclusion of attribute only if navigation
  - Multiplicity = 1: Variable
  - Multiplicity >= 2 : Arrays, ArrayLists,..
- Single inheritance: **`extends`**
- Association classes: Hashtables

Structural Modeling

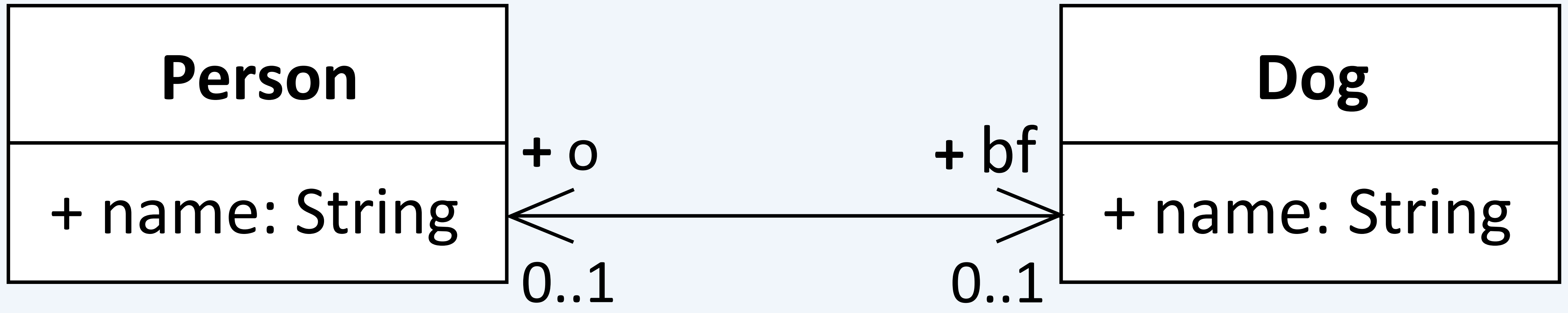
Reverse Engineering

Christian Huemer und Marion Scholz

Presented by Nicholas Bzowski

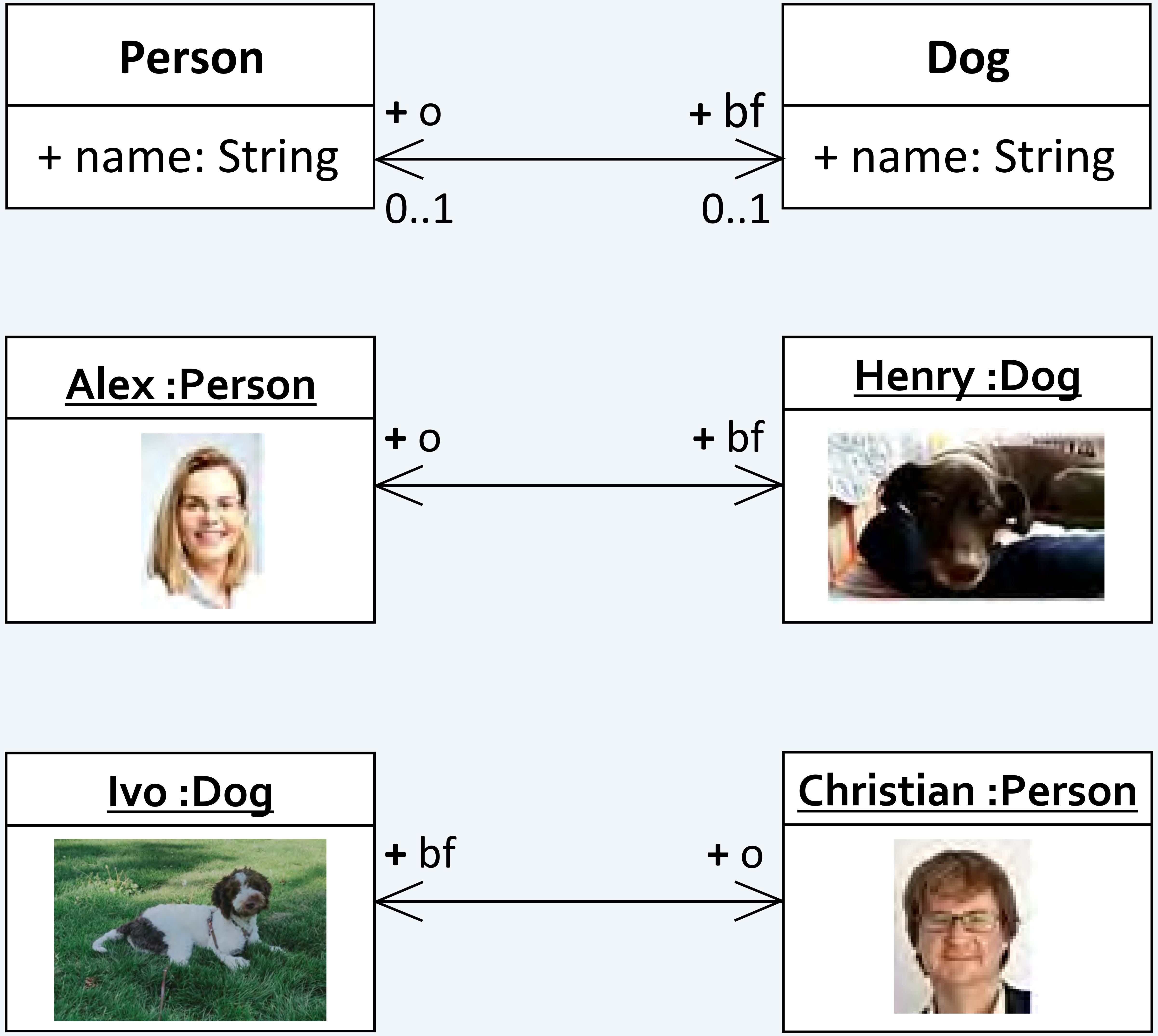
Reverse Engineering

```
class Person {
  public String name;
  public Dog bf;
}

class Dog {
  public String name;
  public Person o;
}
```



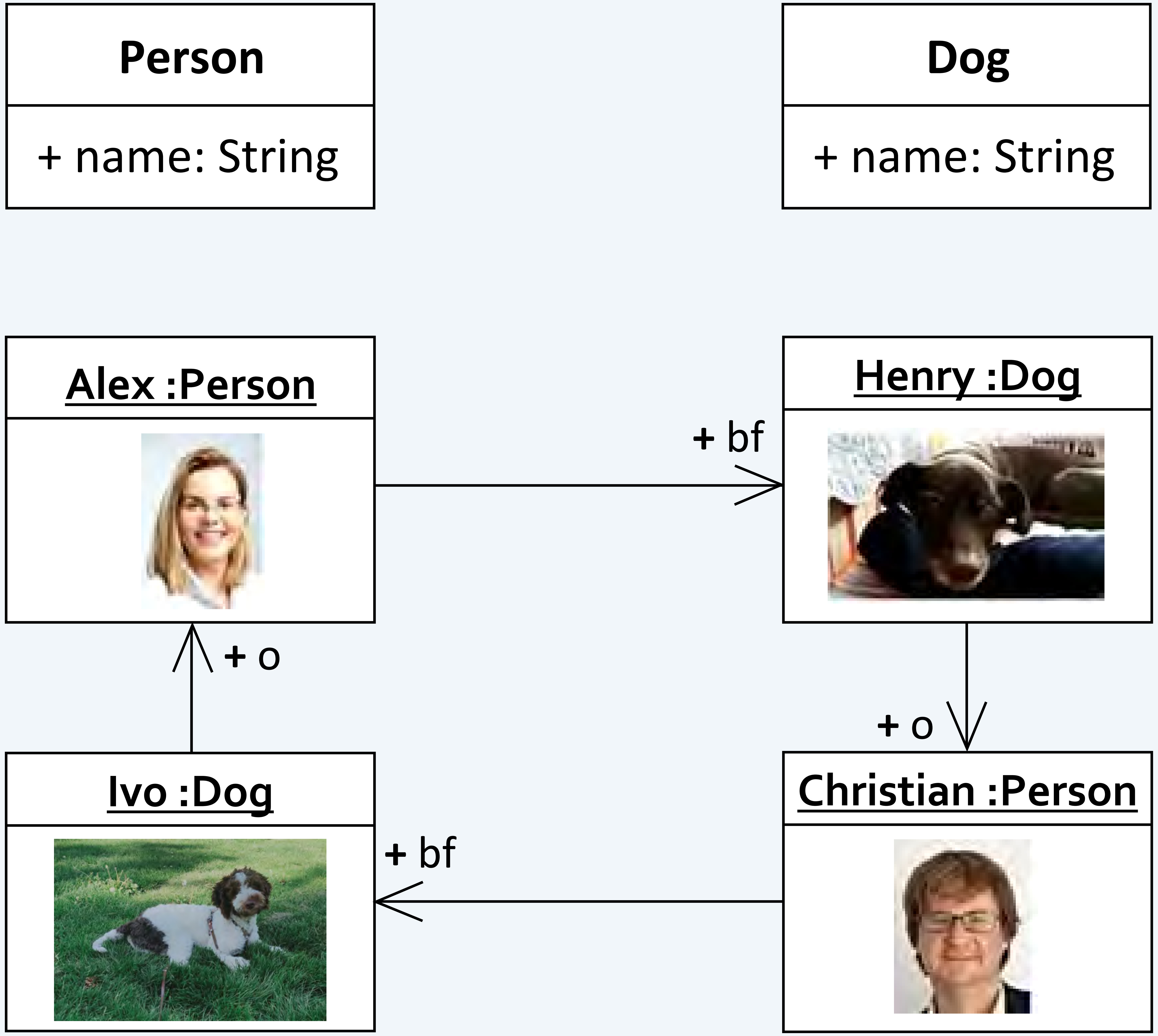
Reverse Engineering

```
class Person {
  public String name;
  public Dog bf;
}

class Dog {
  public String name;
  public Person o;
}
```



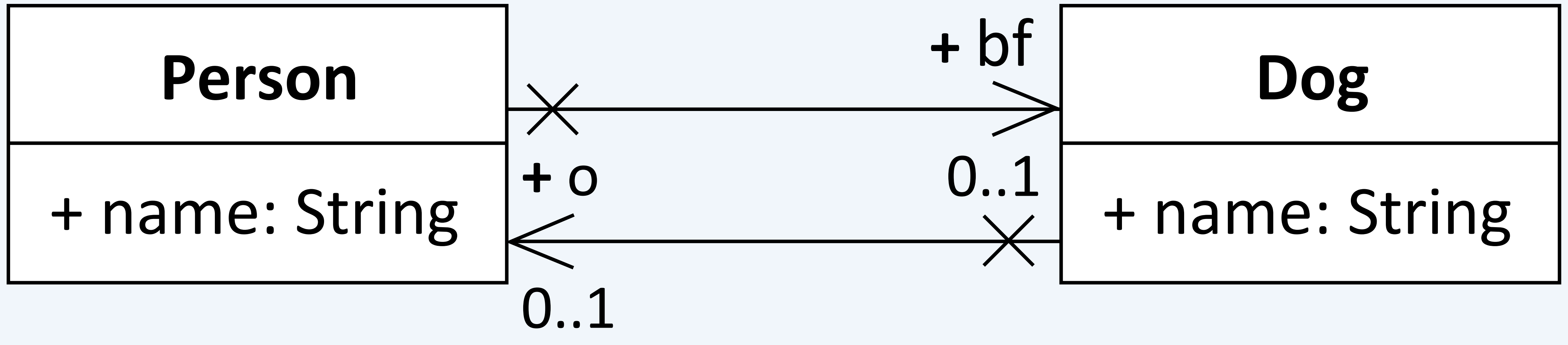
Reverse Engineering

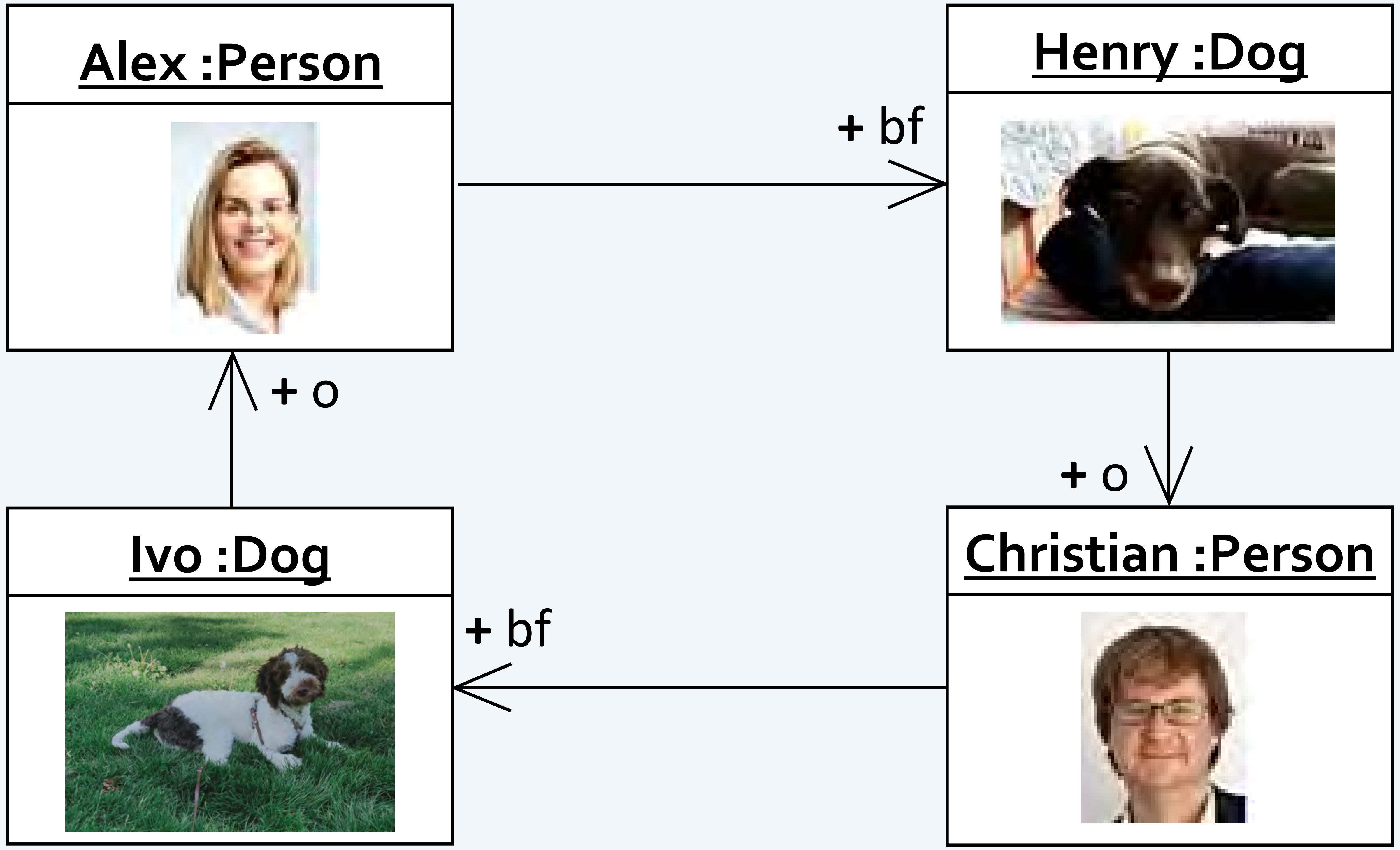
```
class Person {
  public String name;
  public Dog bf;
}

class Dog {
  public String name;
  public Person o;
}
```



Reverse Engineering

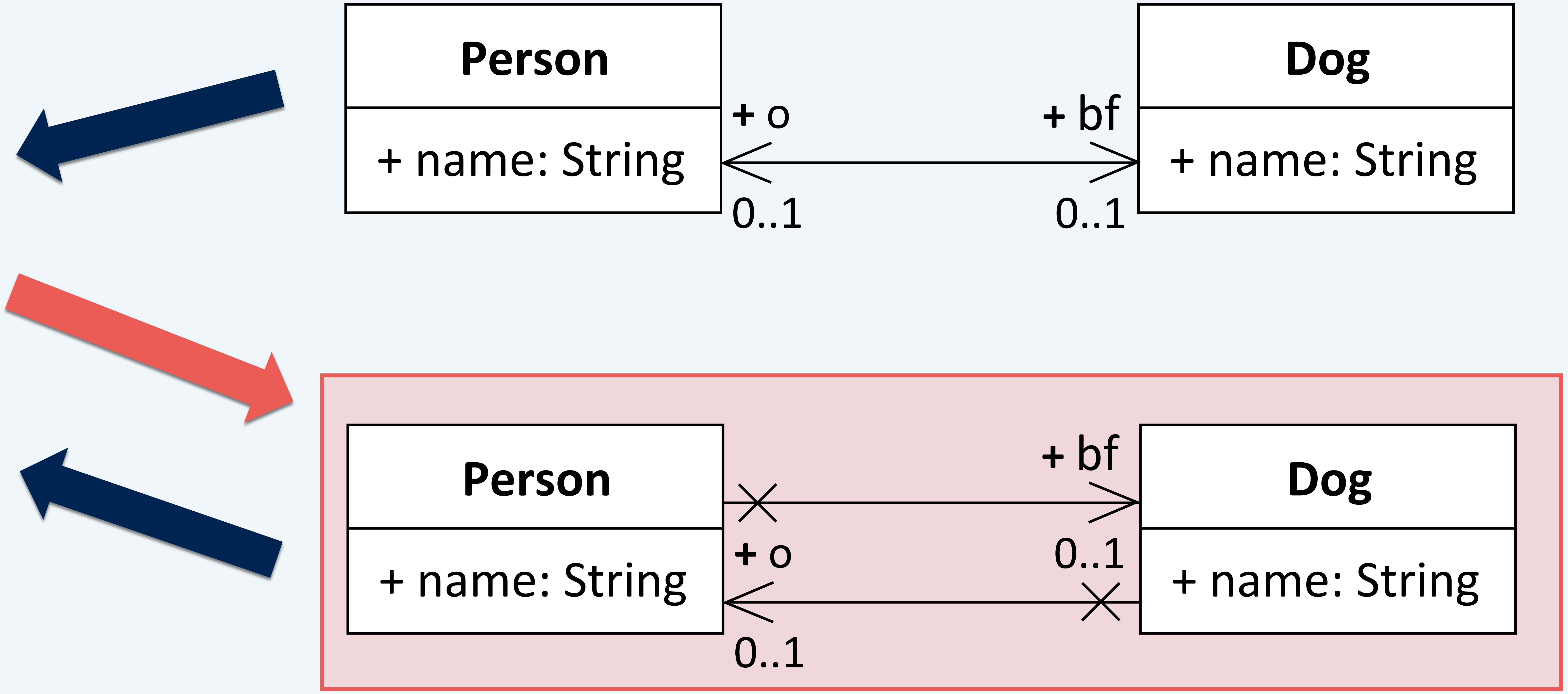
```
class Person {
  public String name;
  public Dog bf;
}

class Dog {
  public String name;
  public Person o;
}
```





Forward vs. Reverse Engineering

```
class Person {
  public String name;
  public Dog bf;
}

class Dog {
  public String name;
  public Person o;
}
```



Structural Modeling

The Package Diagram

Christian Huemer und Marion Scholz

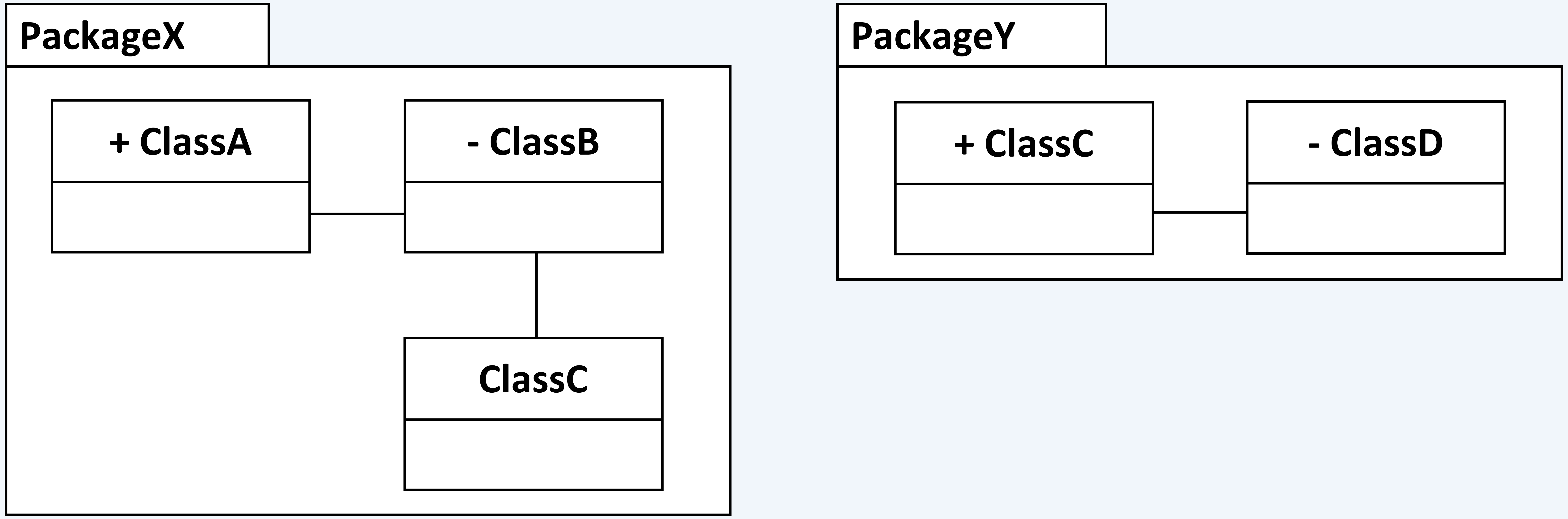
Presented by Nicholas Bzowski

The Package Diagram

- UML abstraction mechanism: Package
- Model elements can be assigned to a maximum of **one** package
- Partitioning criteria:
  - Functional cohesion
  - Information cohesion
  - Access control
  - Distribution structure
  - ....
- Packages form their own namespace
- Visibility of the elements can be defined as "+" or "-"

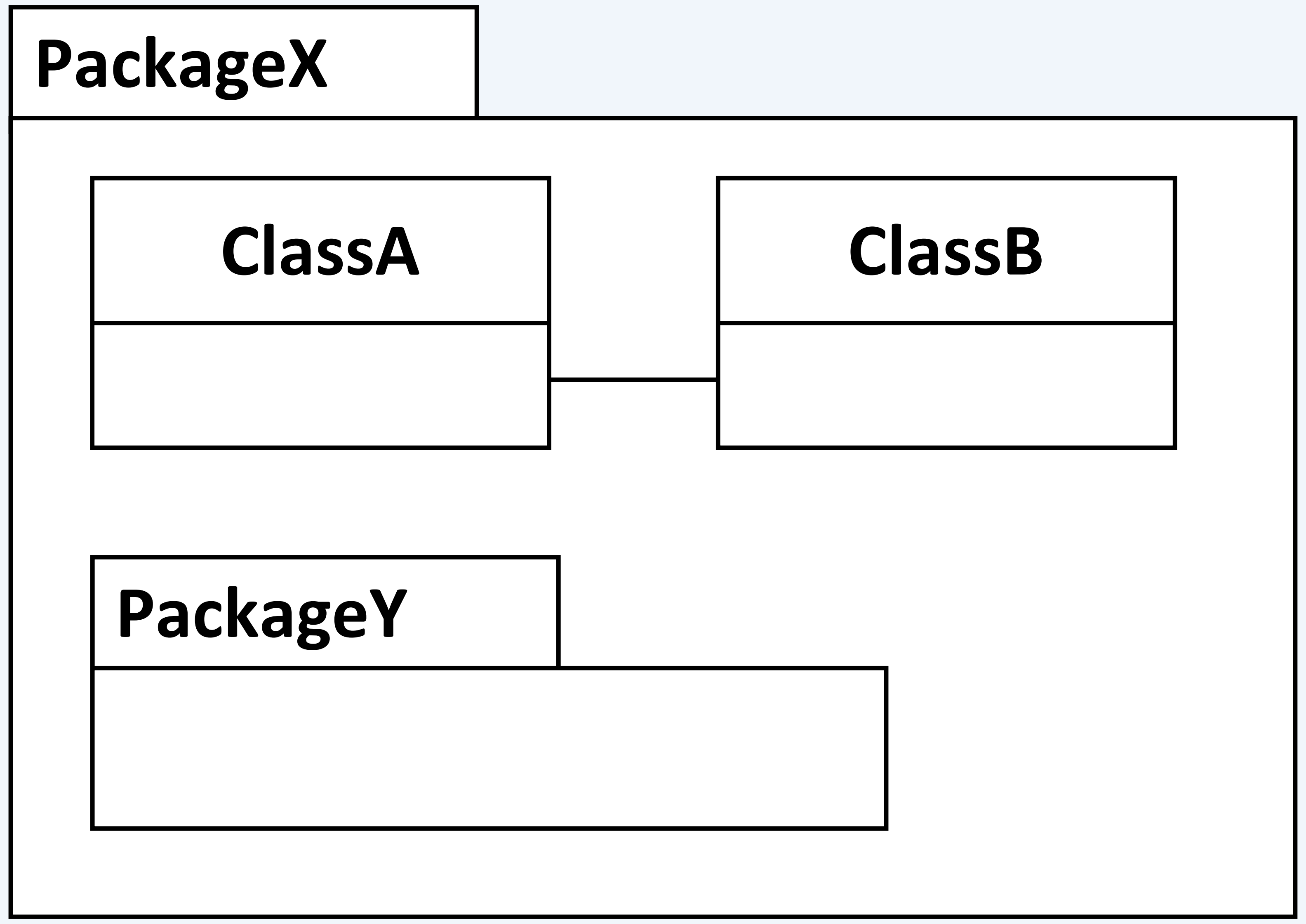
Use of elements from other packages

- Elements of one package require elements of another
- Qualification of these "external" elements
  - Access via qualified names
  - Only to public elements of a package

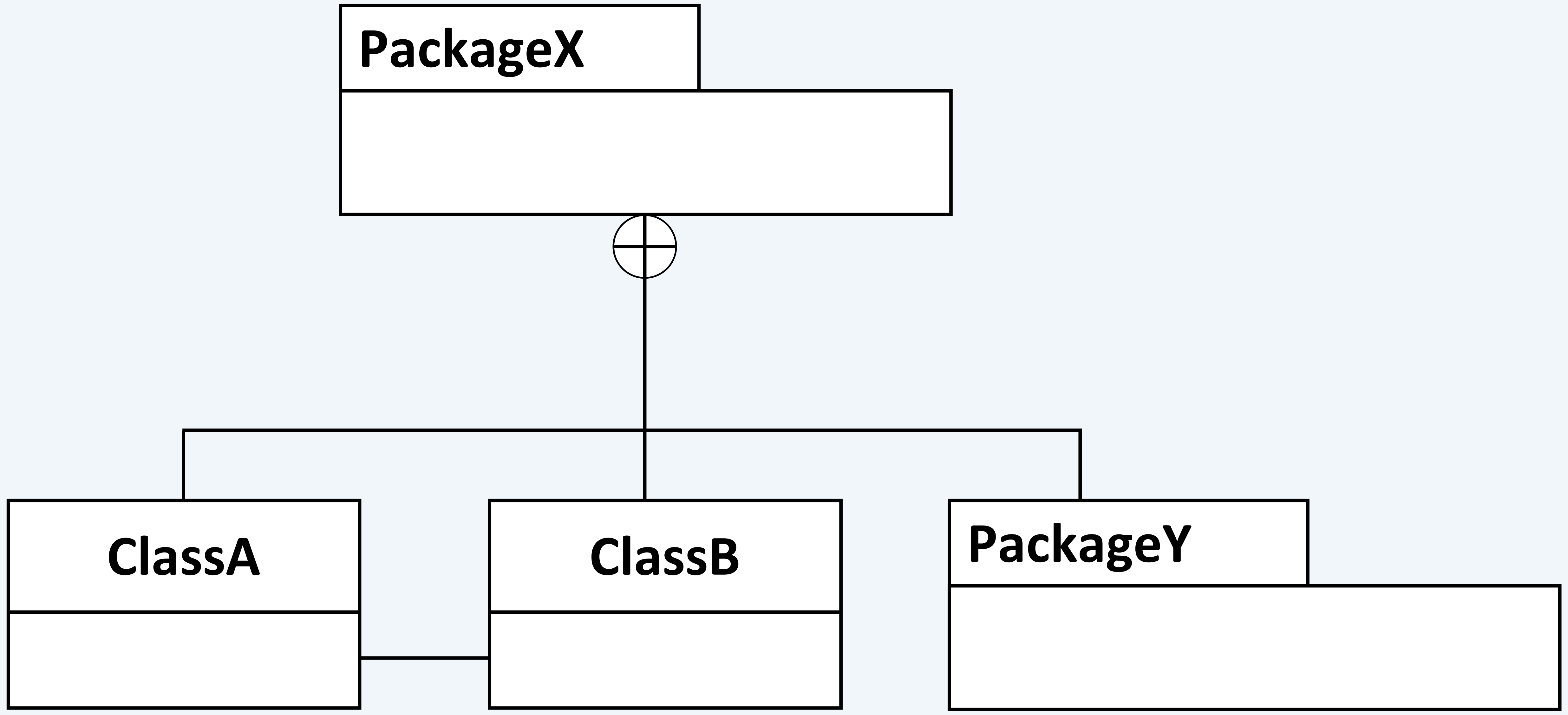


Package Hierarchies

- Packages can be nested
  - Any depth
  - Package hierarchy forms a tree
- Two forms of representation



or



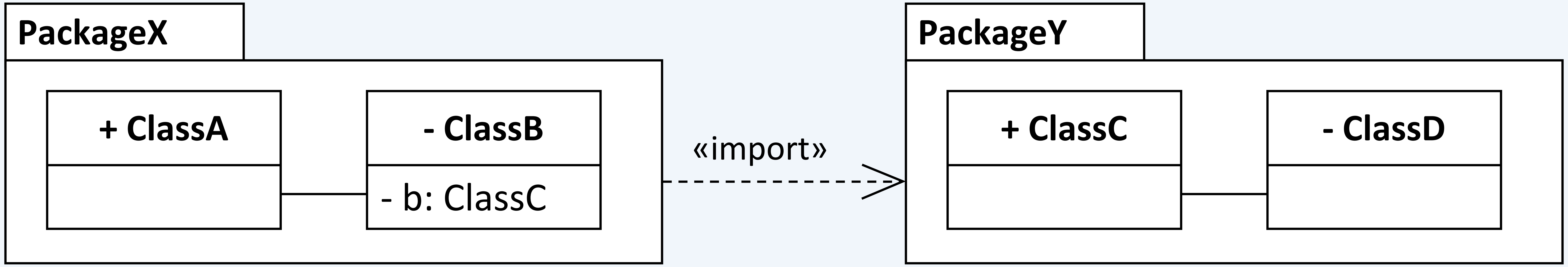
Importing Elements and Packages

- Importing individual elements
  - Prerequisite: Visibility of the element is public
- Importing entire packages
  - Equivalent to the element import of all publicly visible elements of the imported package

- Visibilities
  - The visibility of the imported elements and packages can be redefined during import
  - Visibility only public or private ("+" or "-")
  - «**`import`**» - Relationships for public visibility
  - «**`access`**» - Relationships for private visibility

Importing Elements and Packages – **«import»** (1/2)

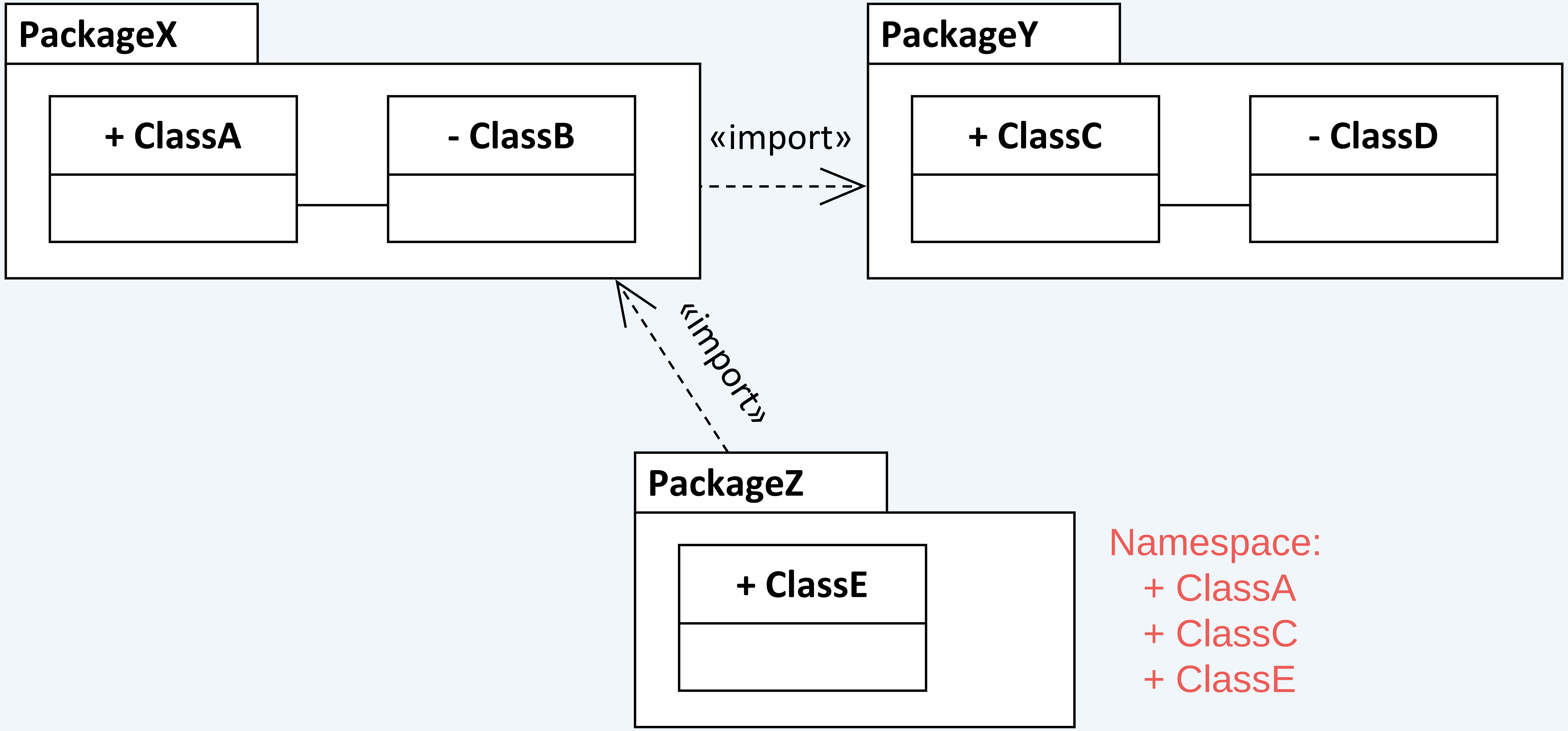
- Changing the namespace
  - Loads the names of the imported package into the namespace of the client
  - Changes the namespace of the client
  - Qualified names are no longer necessary



Importing Elements and Packages – **«import»** (2/2)

- Transitivity
  - The imported names are public and are therefore taken into account when importing again

Namespace:
+ ClassA
- ClassB
+ ClassC

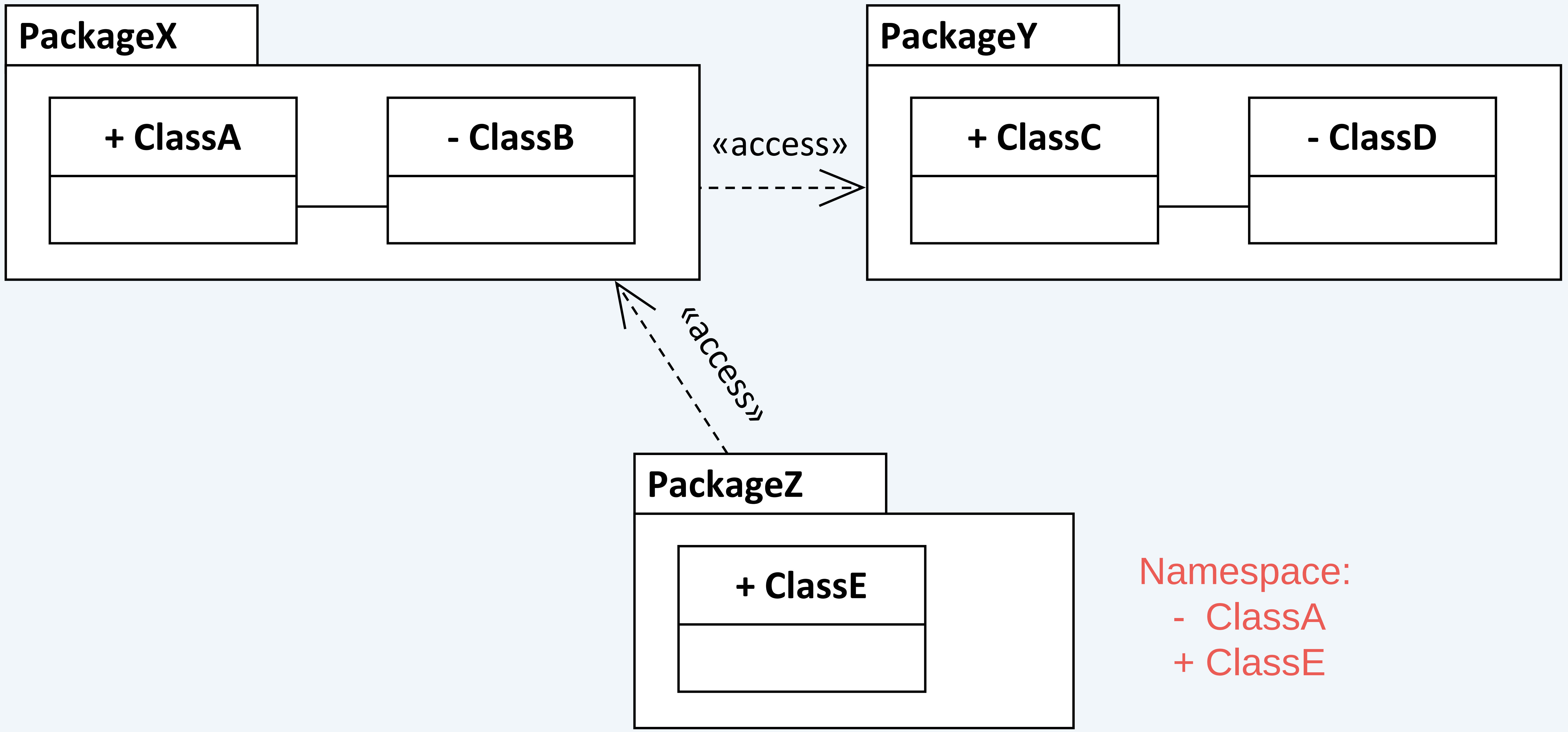


Namespace:
+ ClassA
+ ClassC
+ ClassE

Importing Elements and Packages – **«access»**

- Non-transitive
  - Changes the visibility of imported elements to private

Namespace:
+ ClassA
- ClassB
- ClassC



Namespace:
- ClassA
+ ClassE