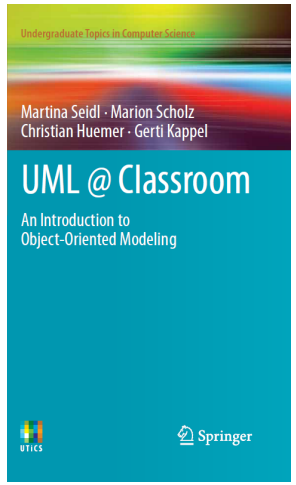
Business Informatics Group

Vienna University of Technology

Object-Oriented Modeling

Structure Modeling – **Advanced Mode**

Slides accompanying UML@Classroom
Version 1.0

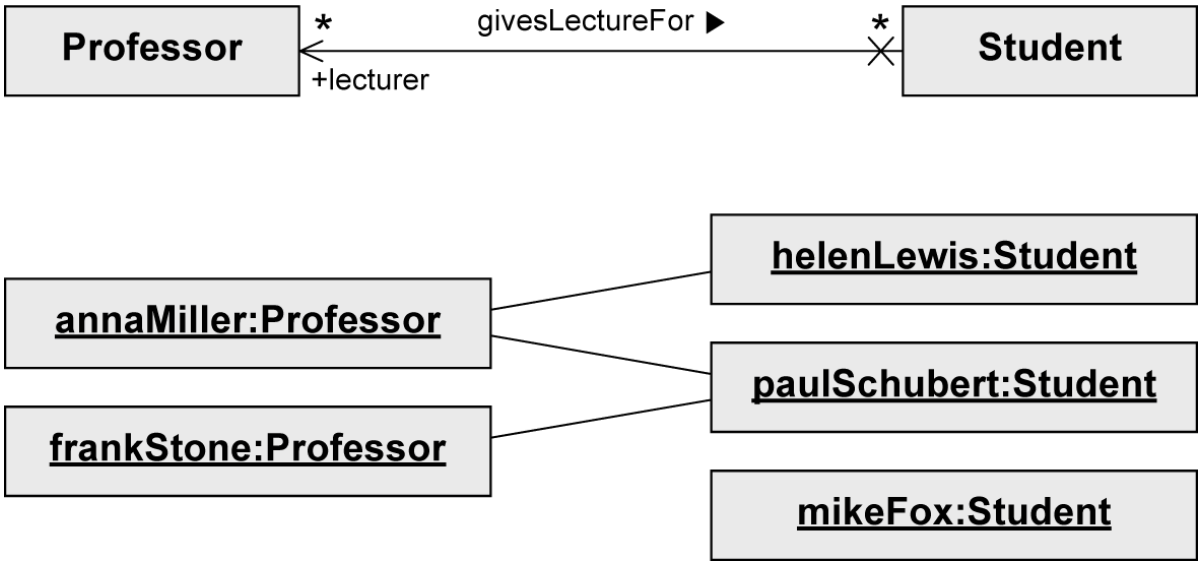


Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology

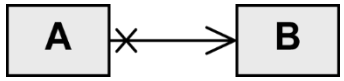
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

Association

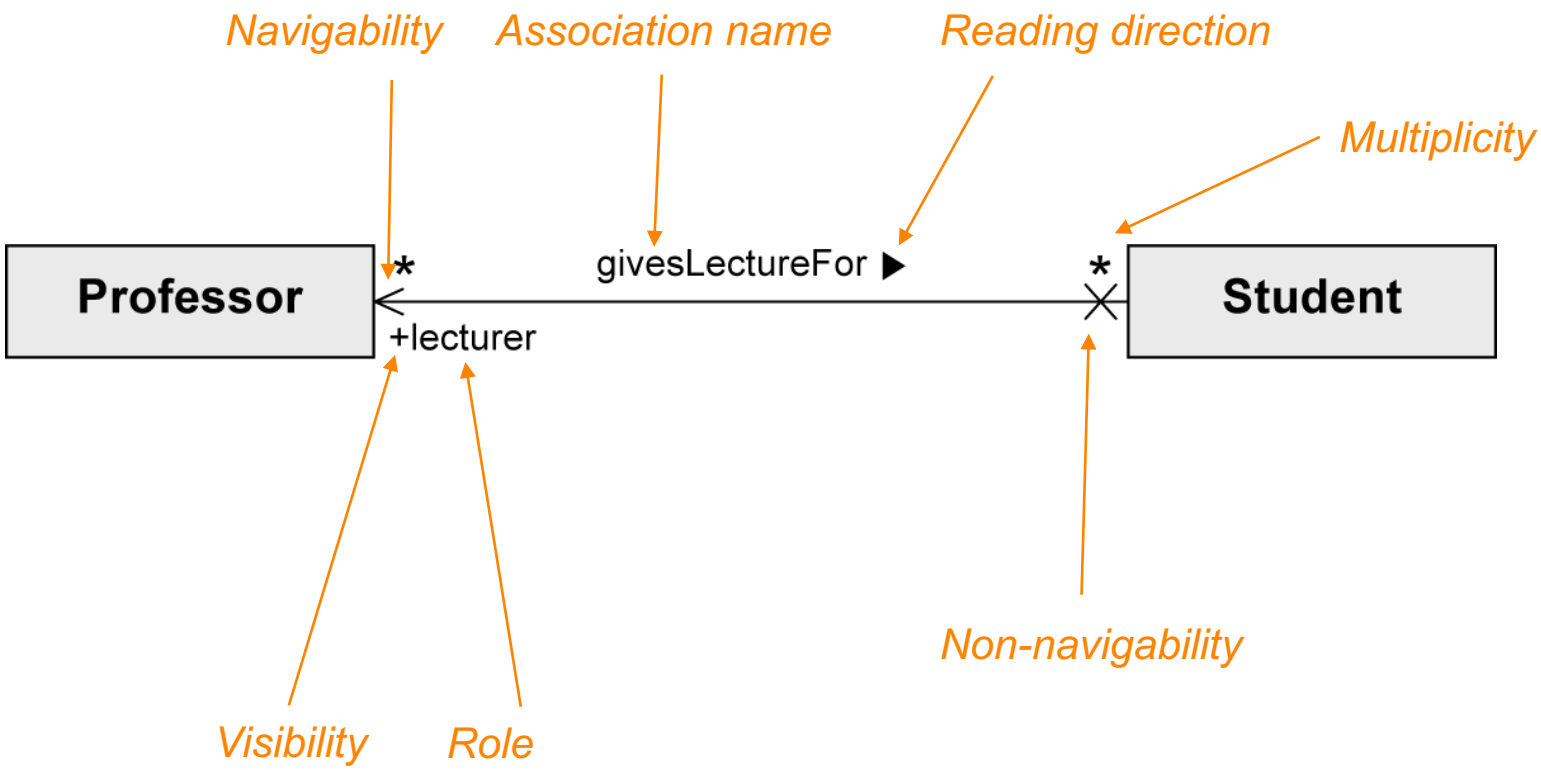
- Models possible relationships between instances of classes



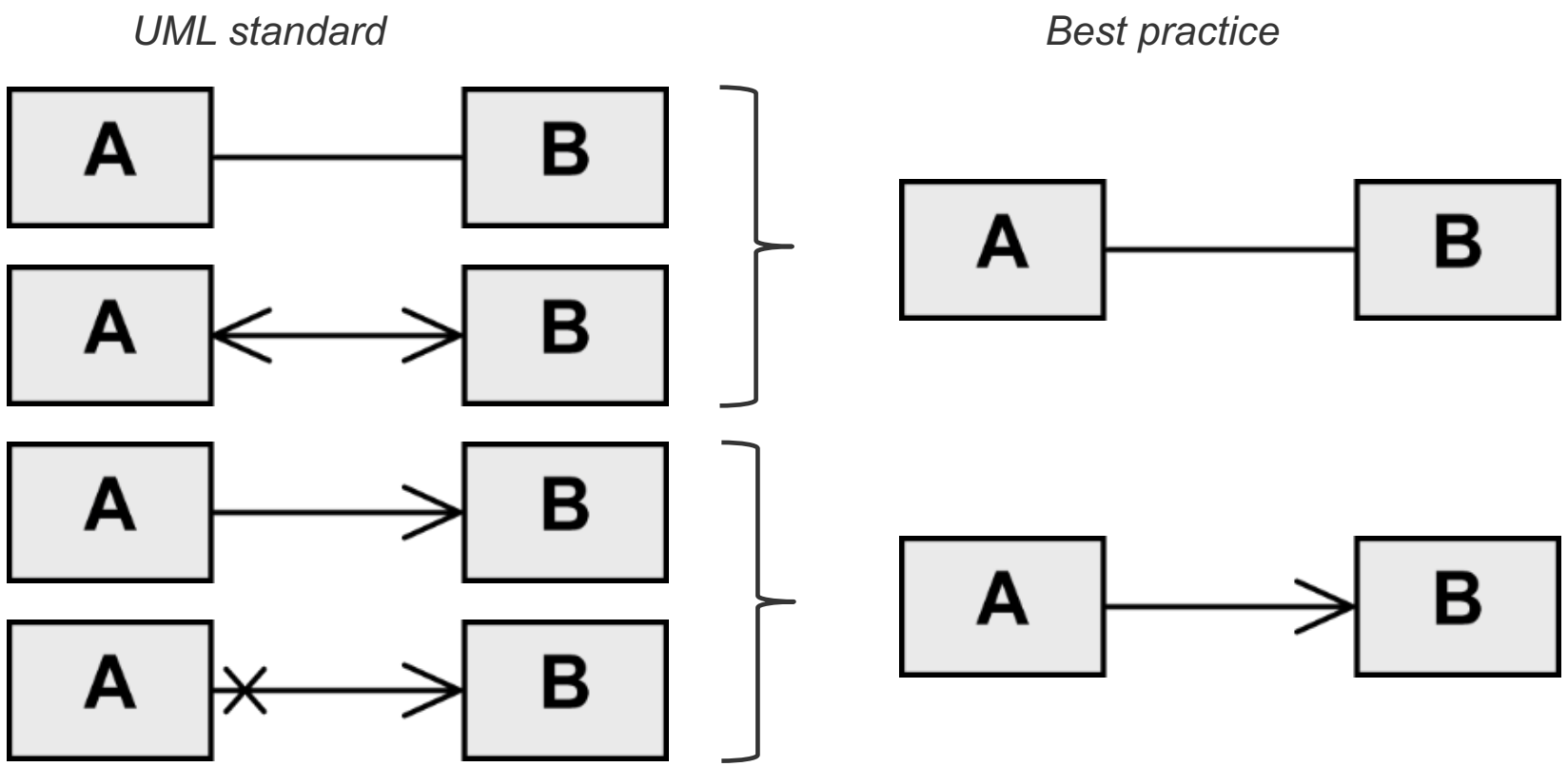
Binary Association



- Connects instances of two classes with one another

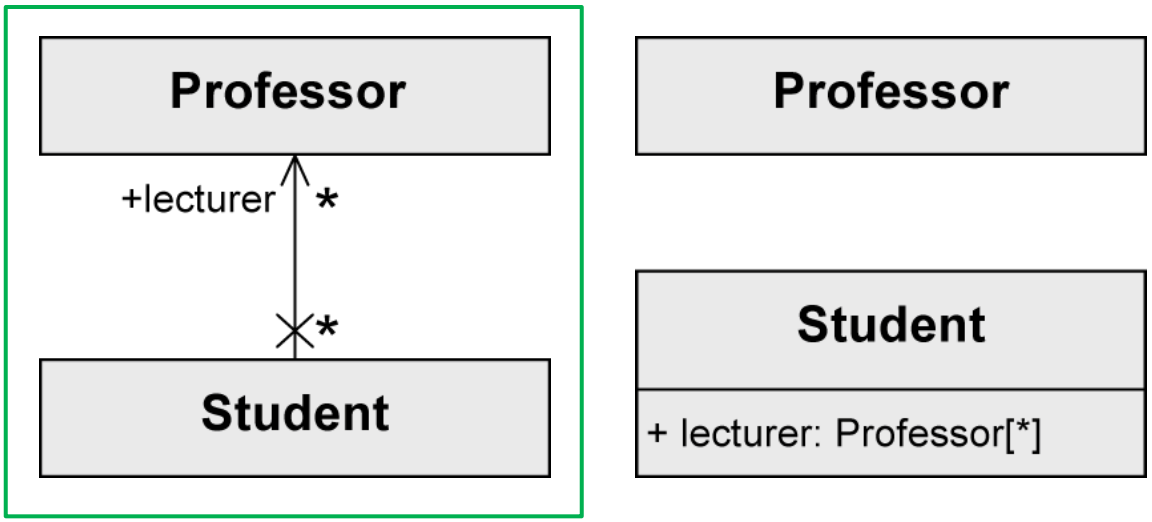


Navigability – UML Standard vs. Best Practice



Binary Association as Attribute

Preferable



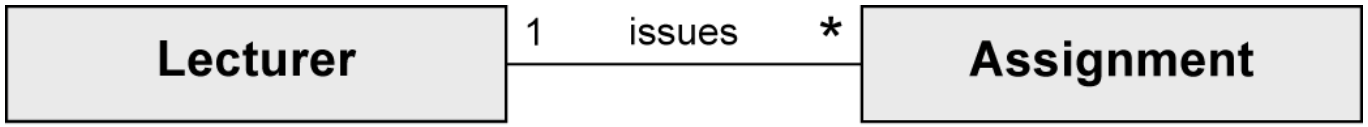
- Java-like notation:

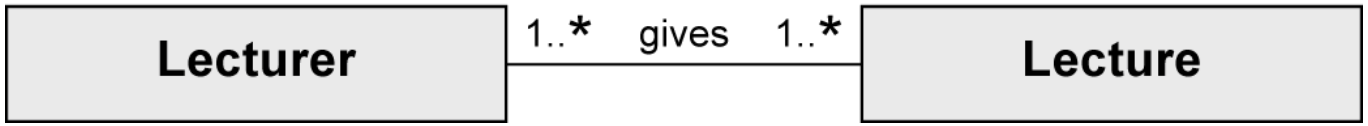
```
class Professor {…}

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

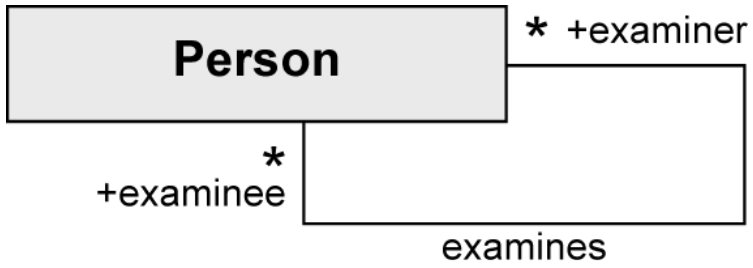
Binary Association – Multiplicity and Role

- Multiplicity: Number of objects that may be associated with exactly one object of the opposite side

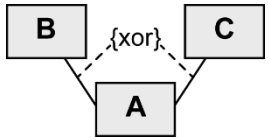




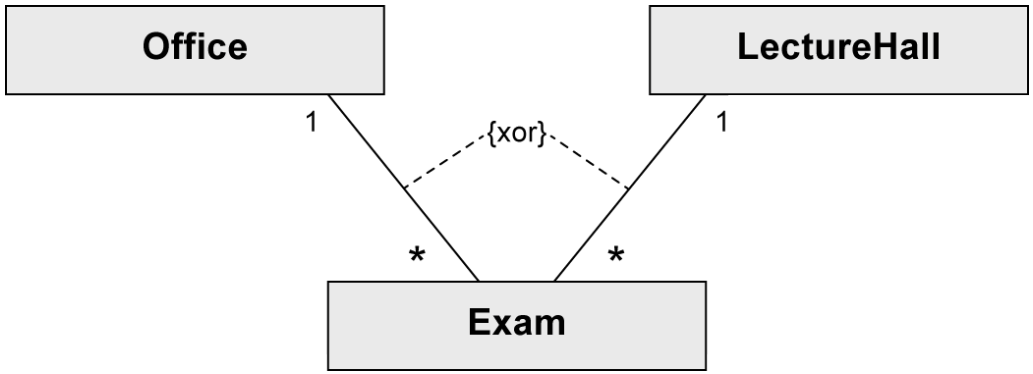
- Role: describes the way in which an object is involved in an association relationship



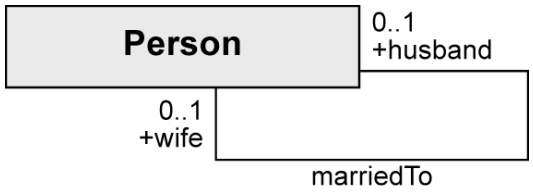
Binary Association – `xor` constraint

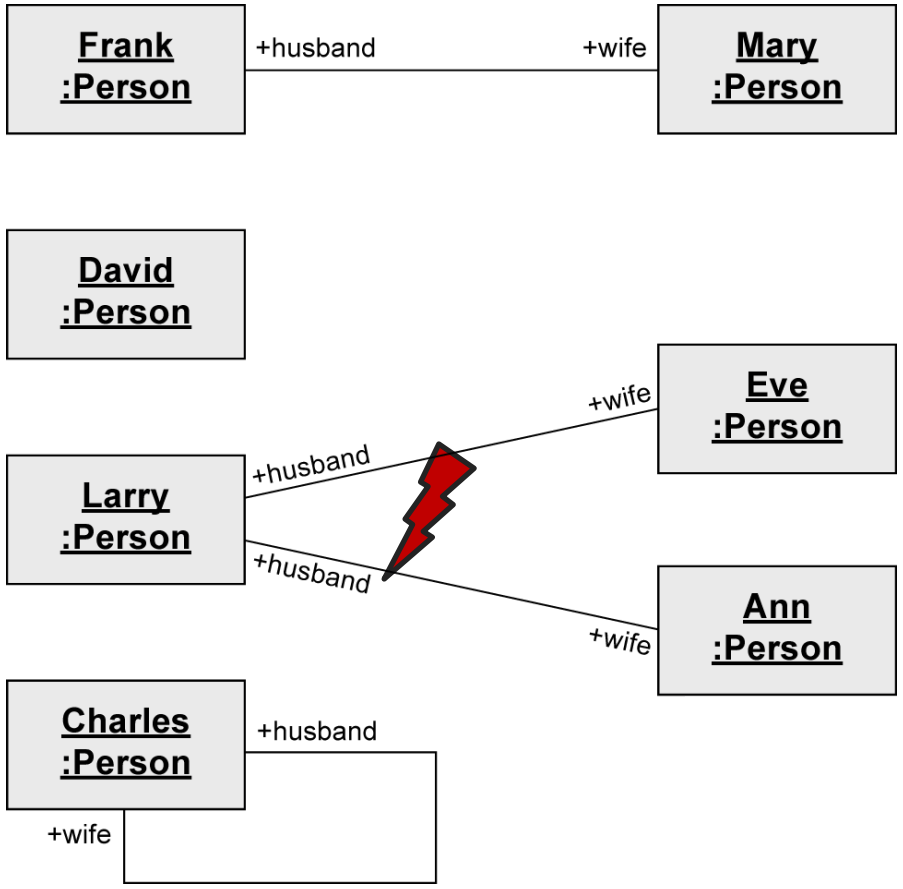


- "exclusive or" constraint
- An object of class **`A`** is to be associated with an object of class **`B`** or an object of class **`C`** but not with both.

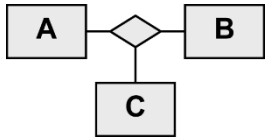


Unary Association - Example

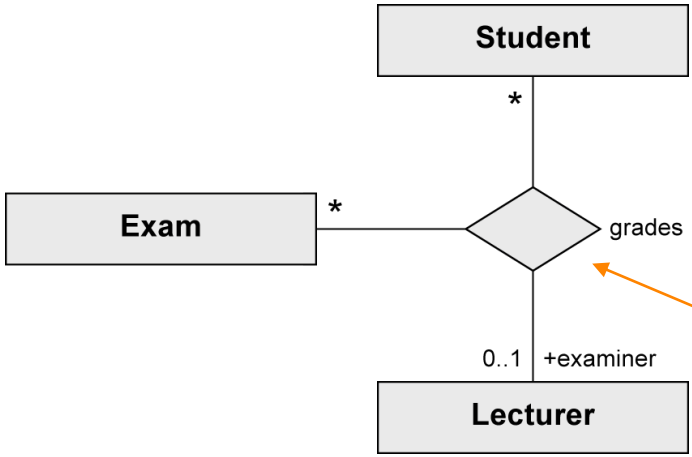




n-ary Association (1/2)



- More than two partner objects are involved in the relationship.
- No navigation directions

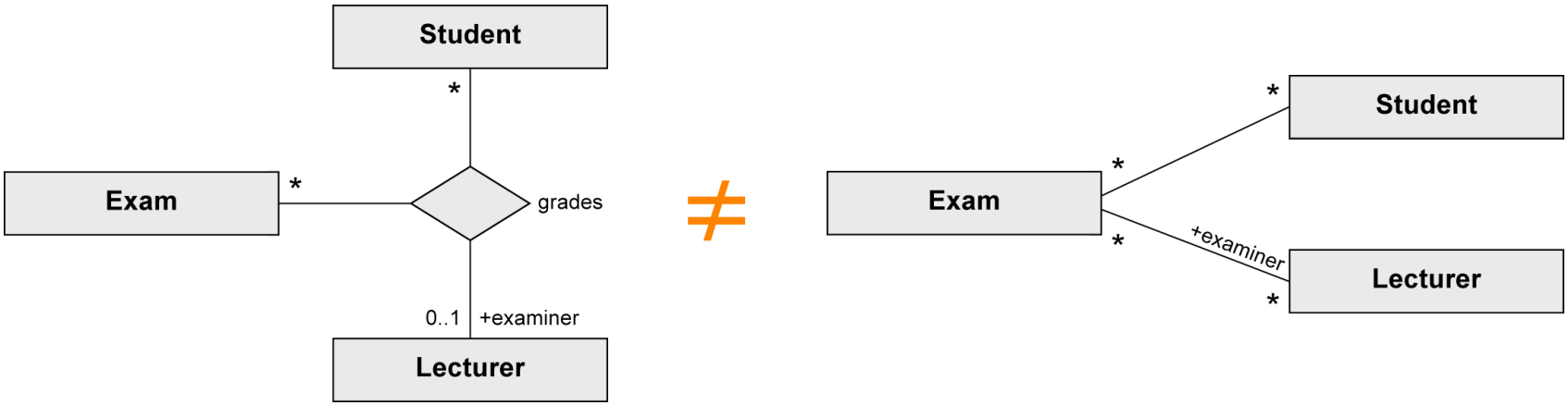


Ternary association

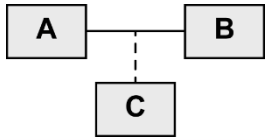
DO NOT USE

n-ary Association (2/2)

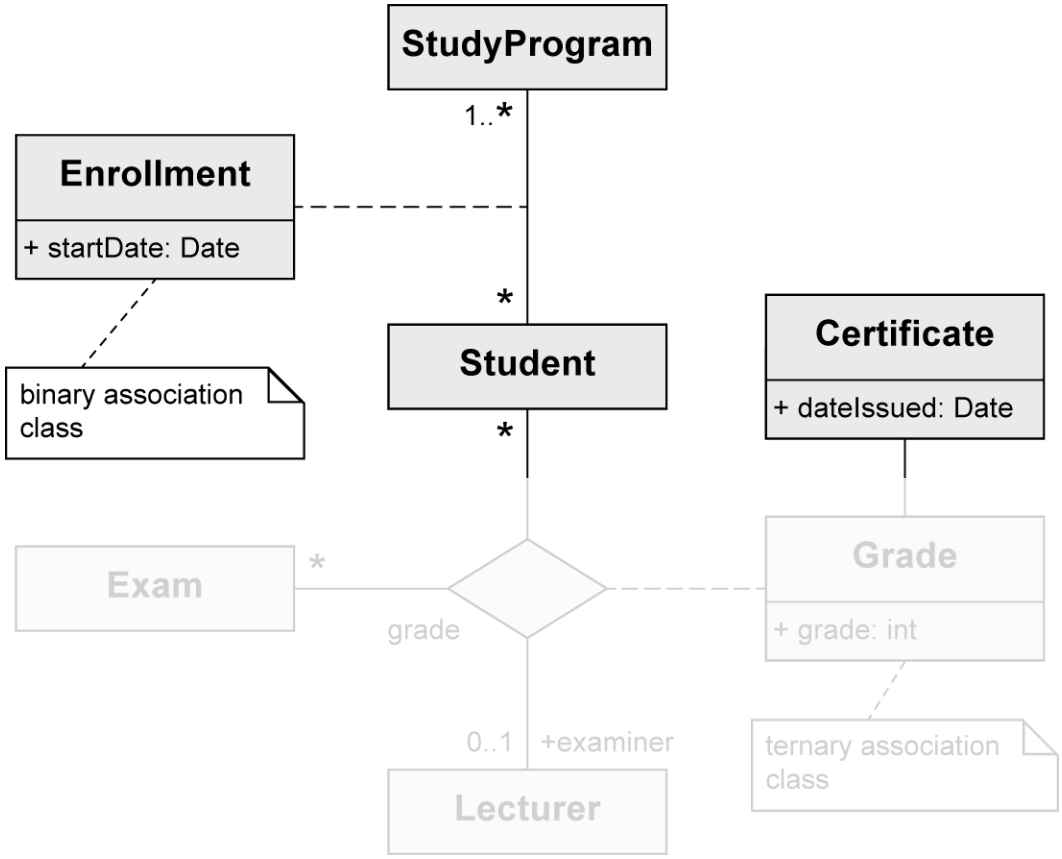
- Example
  - (**`Student`**, **`Exam`**) → (**`Lecturer`**)
    - One student takes one exam with one or no lecturer
  - (**`Exam`**, **`Lecturer`**) → (**`Student`**)
    - One exam with one lecturer can be taken by any number of students
  - (**`Student`**, **`Lecturer`**) → (**`Exam`**)
    - One student can be graded by one **`Lecturer`** for any number of exams



Association Class

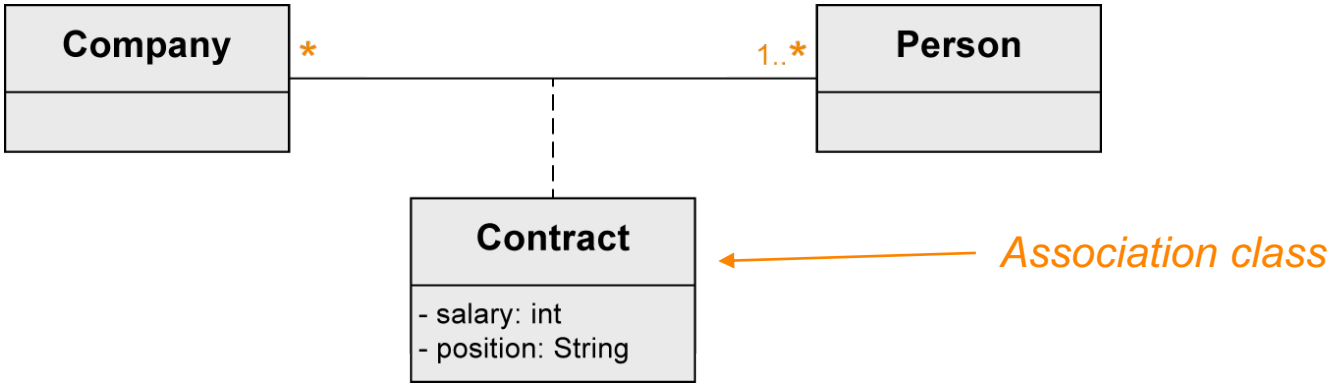


- Assign attributes to the relationship between classes rather than to a class itself

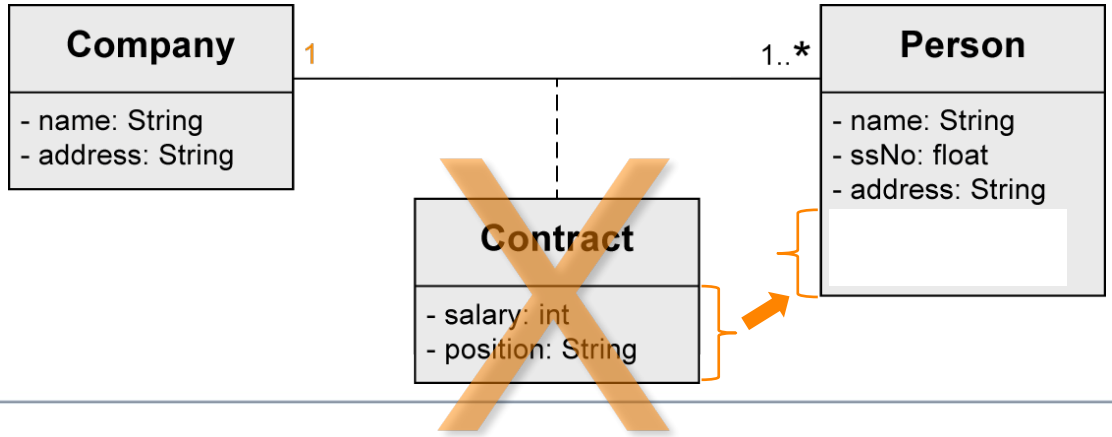


Association Class

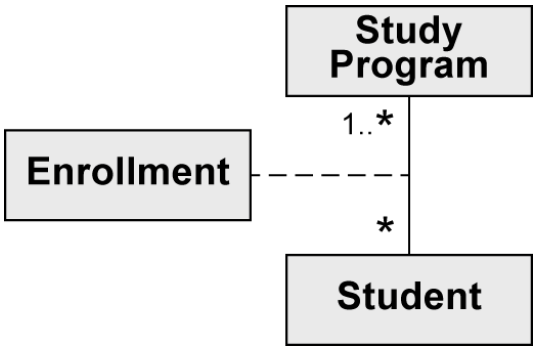
- Necessary when modeling n:m Associations



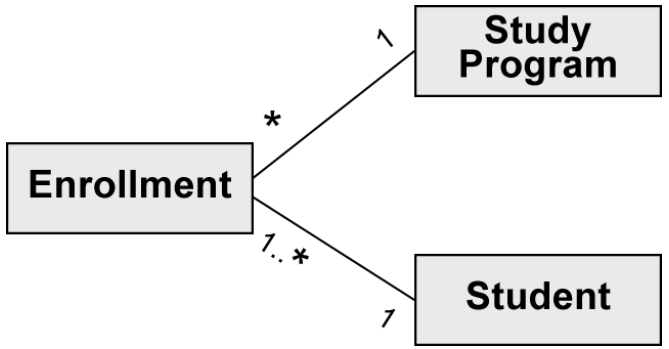
- With 1:1 or 1:n possible but not necessary



Association Class vs. Regular Class



≠

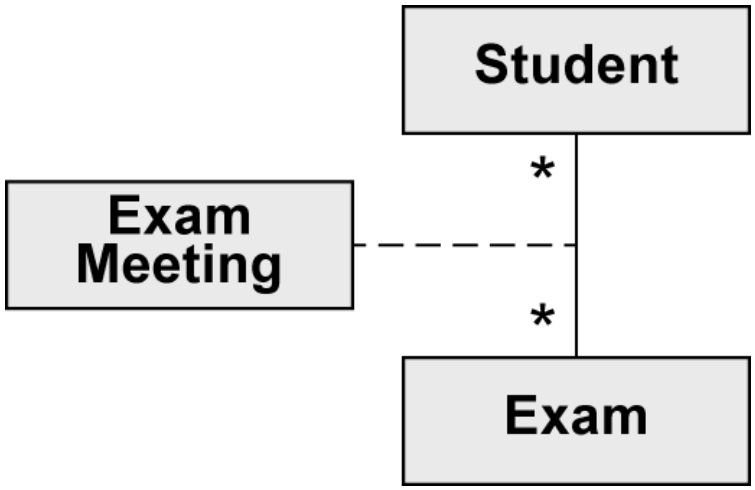


A `Student` *can enroll for one particular* `StudyProgram` *only once*

A `Student` *can have mutiple* `Enrollments` *for one and the same* `StudyProgram`

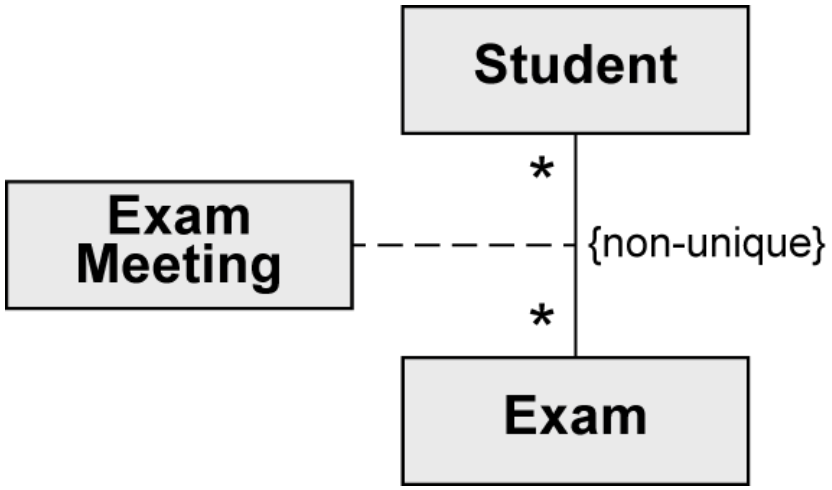
Association Class – `unique/non-unique` (1/2)

- Default: no duplicates



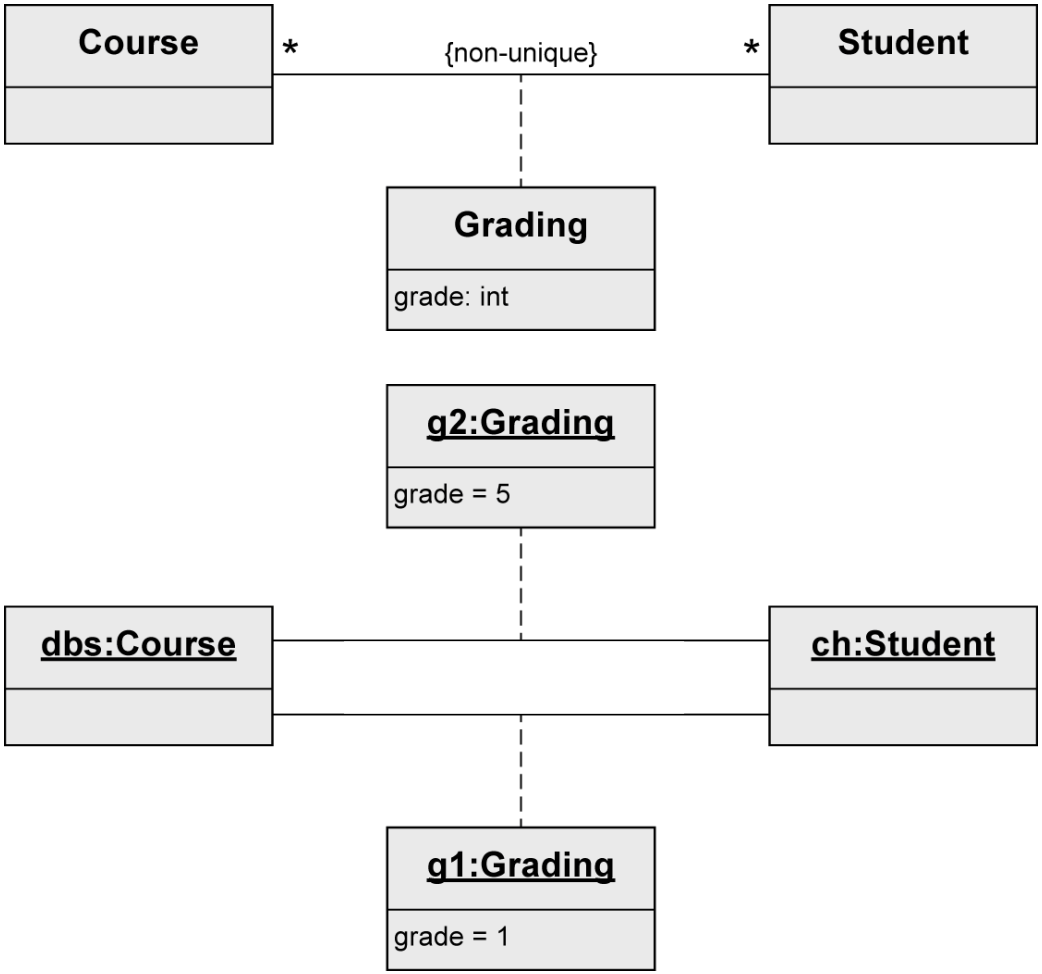
A student can only be granted an exam meeting for a specific exam once.

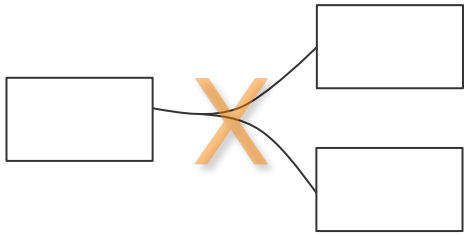
- non-unique: duplicates allowed



A student can have more than one exam meetings for a specific exam.

Association Class – `unique/non-unique` (2/2)



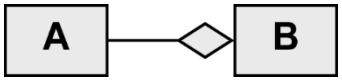


Aggregation

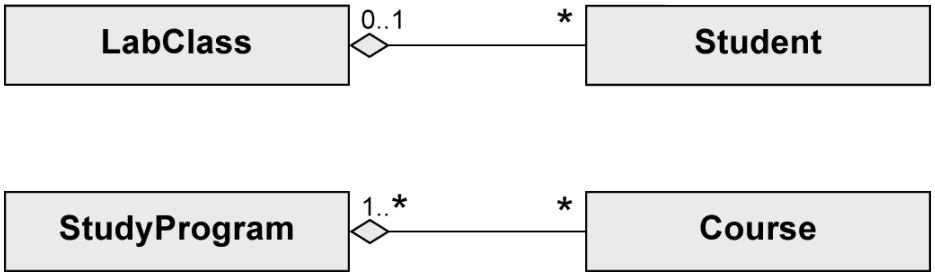
- Special form of association
- Used to express that a class is part of another class
- Properties of the aggregation association:
  - **Transitive**: if **`B`** is part of **`A`** and **`C`** is part of **`B`**, **`C`** is also part of **`A`**
  - **Asymmetric**: it is not possible for **`A`** to be part of **`B`** and **`B`** to be part of **`A`** simultaneously.

- Two types:
  - Shared aggregation
  - Composition

Shared Aggregation



- Expresses a weak belonging of the parts to a whole
  = Parts also exist independently of the whole
- Multiplicity at the aggregating end may be >1
  = One element can be part of multiple other elements simultaneously
- Spans a directed acyclic graph
- Syntax: Hollow diamond at the aggregating end
- Example:
  - **`Student`** is part of **`LabClass`**
  - **`Course`** is part of **`StudyProgram`**



Composition

- Existence dependency between the composite object and its parts
- One part can only be contained in at most one composite object at one specific point in time

  Multiplicity at the aggregating end max. 1

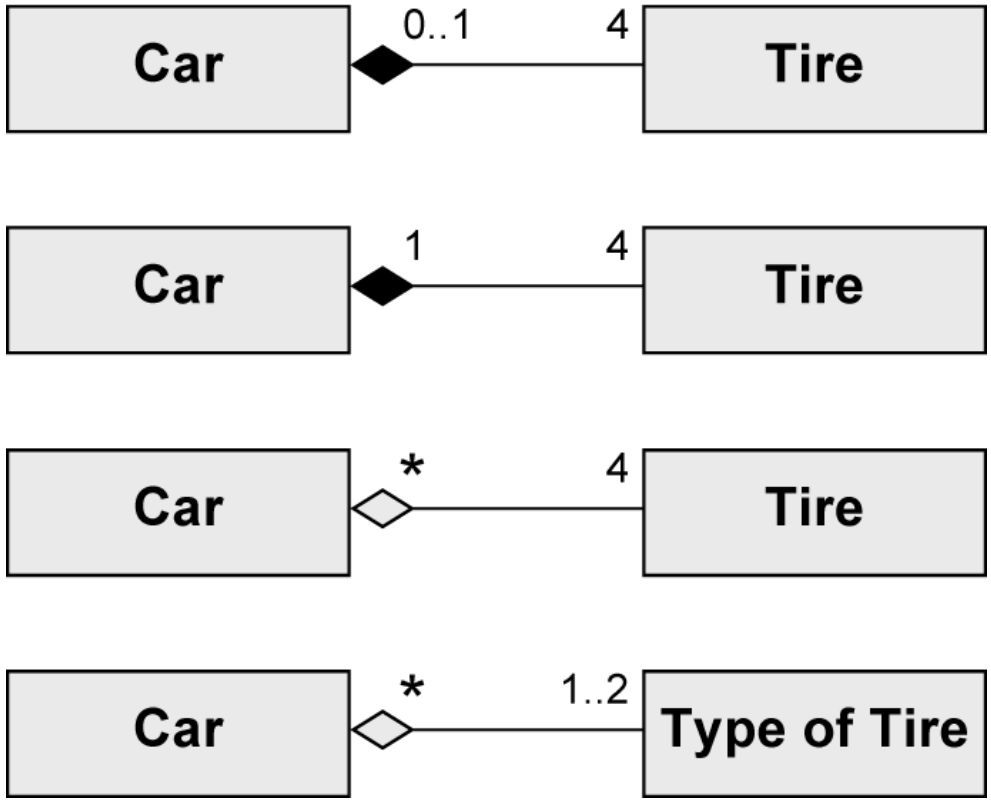
  -> The composite objects form a tree
- If the composite object is deleted, its parts are also deleted.
- Syntax: Solid diamond at the aggregating end
- Example: **`Beamer`** is part of **`LectureHall`** is part of **`Building`**

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

The `Beamer` can exist without the `LectureHall`, but if it is contained in the `LectureHall` while it is deleted, the `Beamer` is also deleted

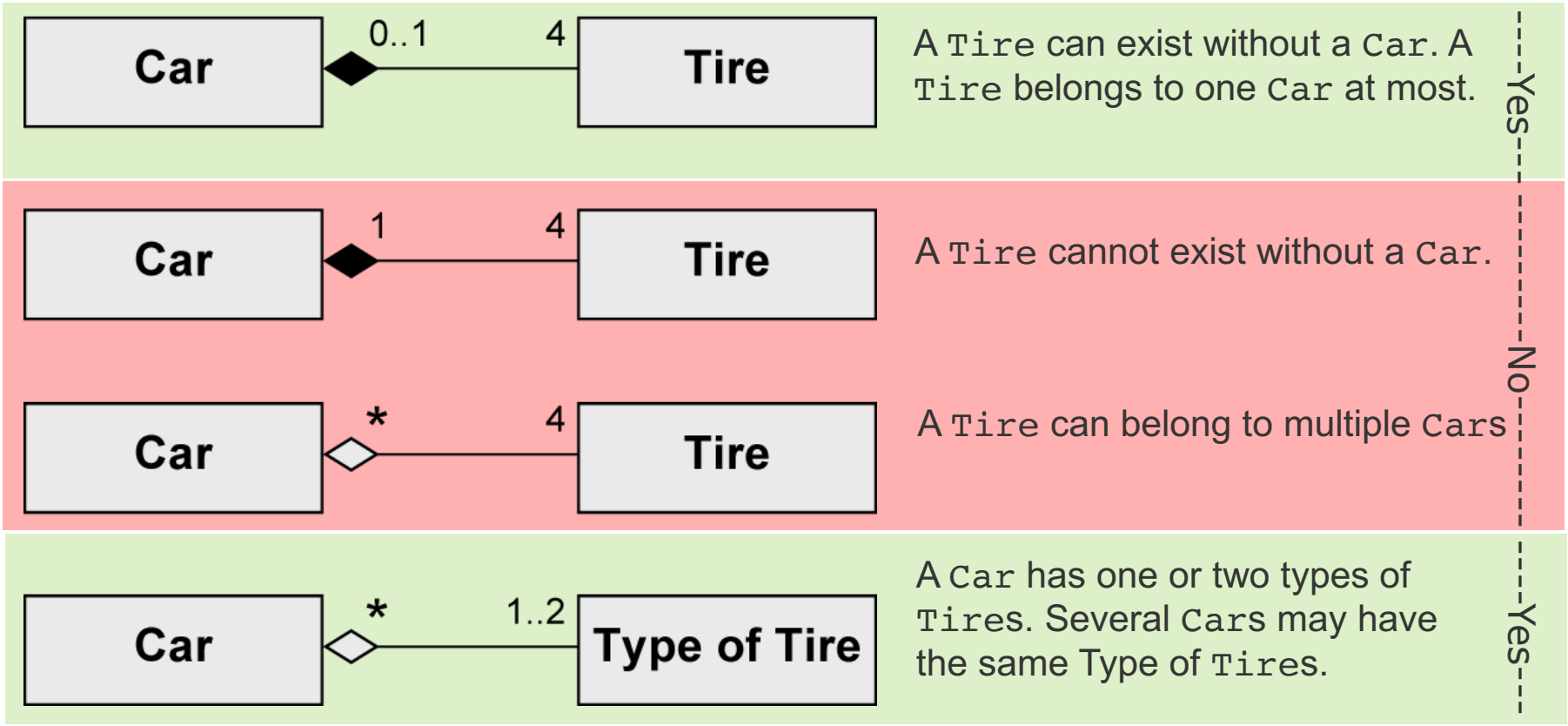
Shared Aggregation and Composition

- Which model applies?

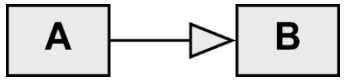


Shared Aggregation and Composition

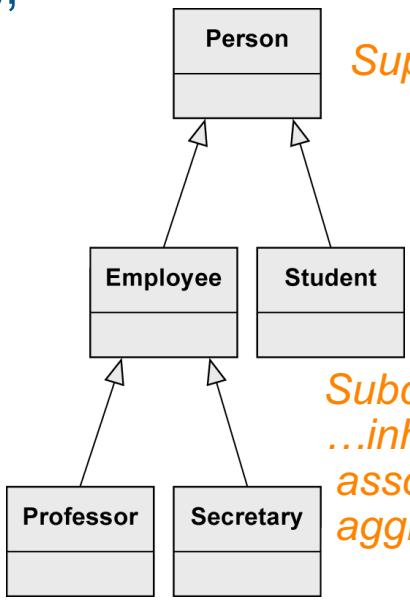
- Which model applys?



Generalization

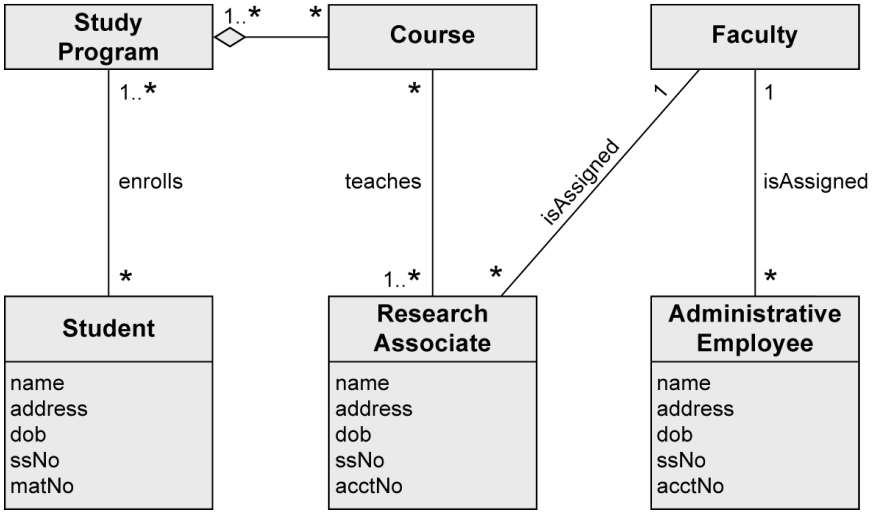


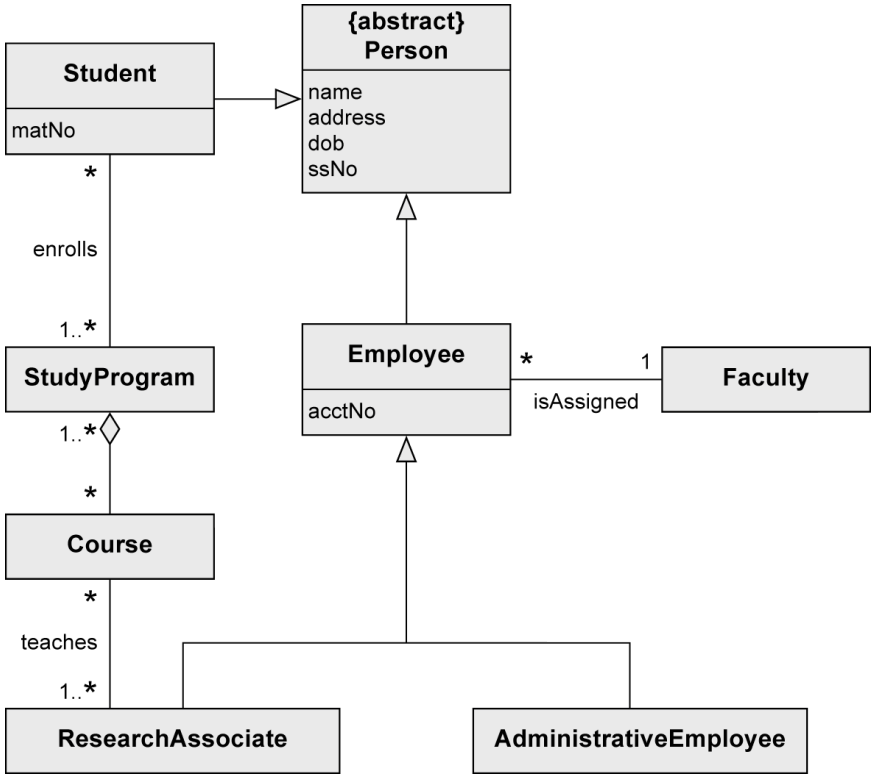
- Characteristics (attributes and operations), associations, and aggregations that are specified for a general class (superclass) are passed on to its subclasses.
- Every instance of a subclass is simultaneously an indirect instance of the superclass.
- Subclass inherits all characteristics, associations, and aggregations of the superclass except private ones.
- Subclass may have further characteristics, associations, and aggregations.
- Generalizations are transitive.



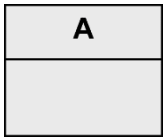
A `Secretary` is an `Employee` and a `Person`

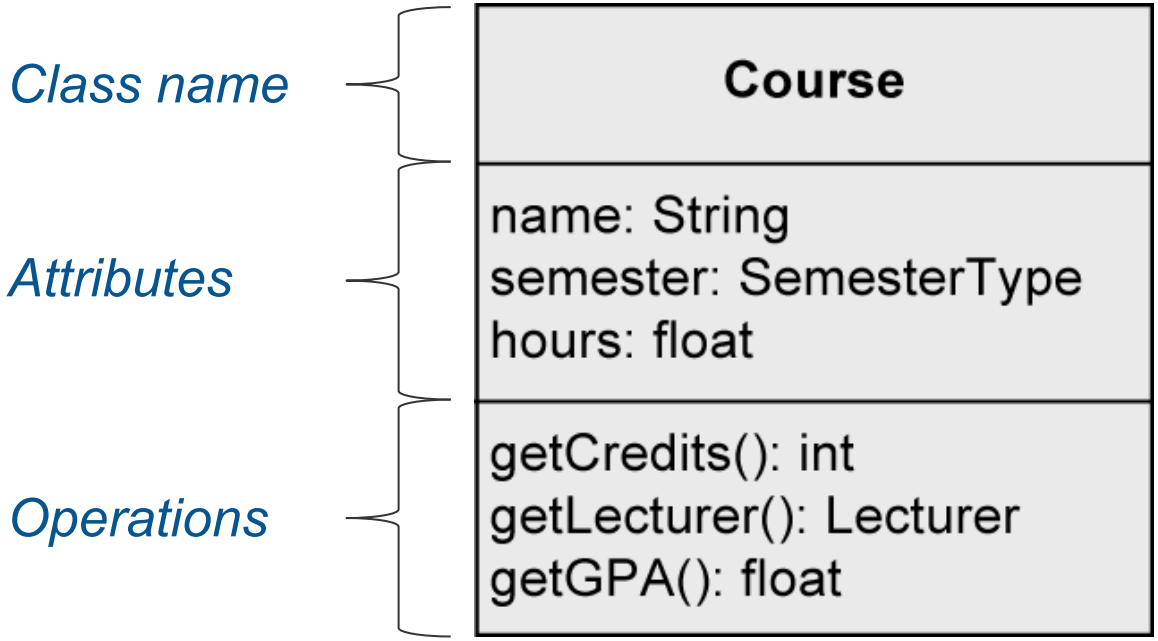
With and Without Generalization



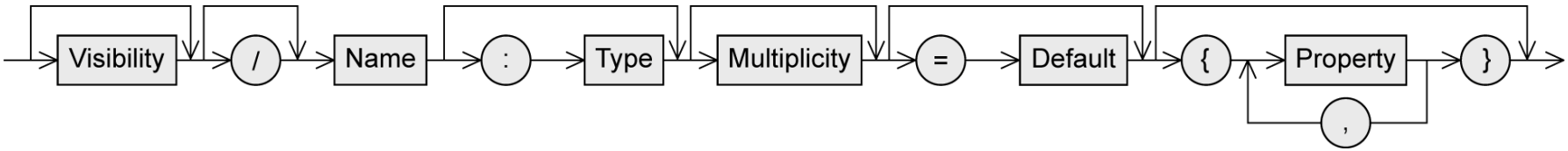


Class

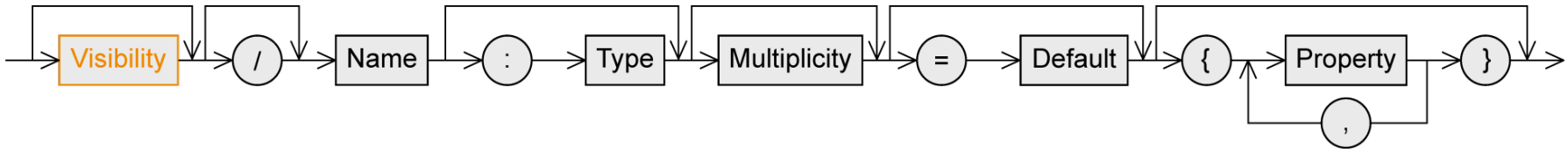

A


Class name
Attributes
Operations
Course
name: String
semester: SemesterType
hours: float
getCredits(): int
getLecturer(): Lecturer
getGPA(): float

Attribute Syntax



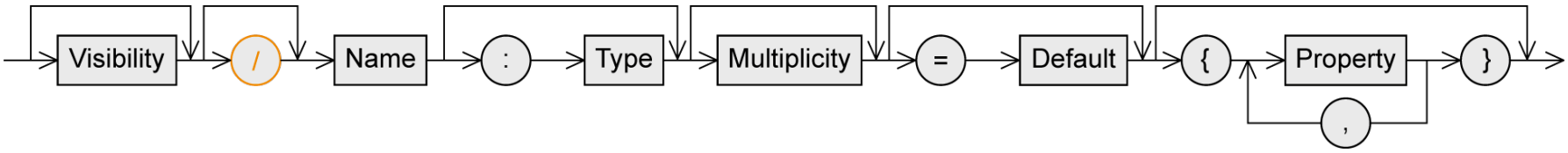
Attribute Syntax - Visibility



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

- Who is permitted to access the attribute
  - + ... public: everybody
  - - ... private: only the object itself
  - # ... protected: class itself and subclasses
  - ~ ... package: classes that are in the same package

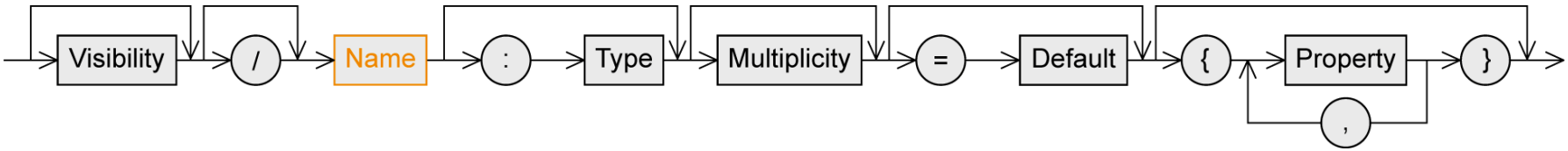
Attribute Syntax - Derived Attribute



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

- Attribute value is derived from other attributes
  - **age**: calculated from the date of birth

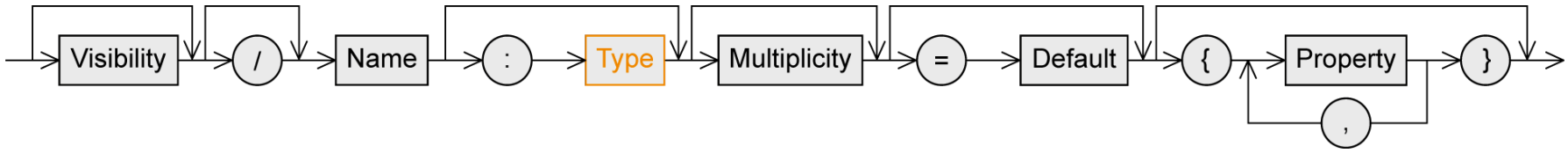
Attribute Syntax - Name



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



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

- Type
  - User-defined classes
  - Data type
    - Primitive data type
      - Pre-defined: Boolean, Integer, UnlimitedNatural, String
      - User-defined: «**primitive**»
      - Composite data type: «**datatype**»
    - Enumerations: «**enumeration**»

«primitive» **Float**
round(): void

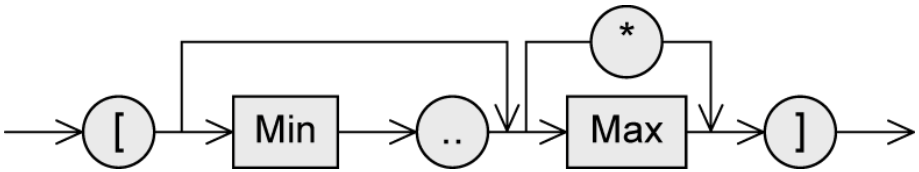
«datatype» **Date**
day month year

«enumeration» **AcademicDegree**
bachelor master phd

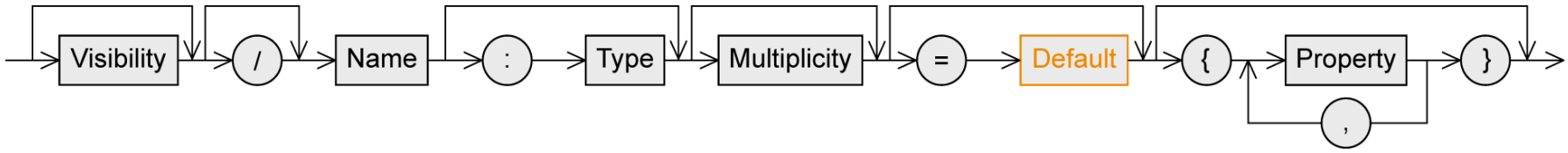
Attribute Syntax - Multiplicity

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 may contain
- Default value: 1
- Notation: **[min..max]**
  - no upper limit: `[*]` or `[0..*]`



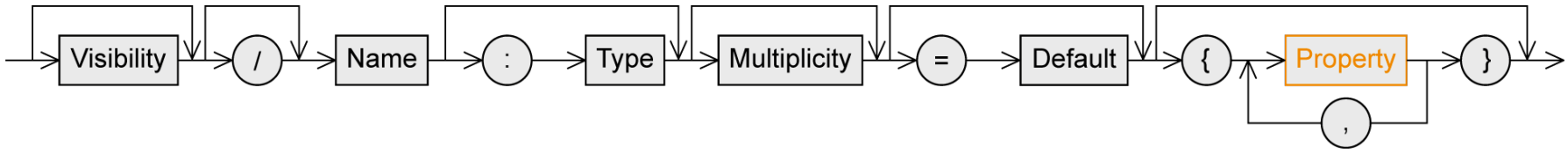
Attribute Syntax – Default Value



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 attribute value is not set explicitly by the user

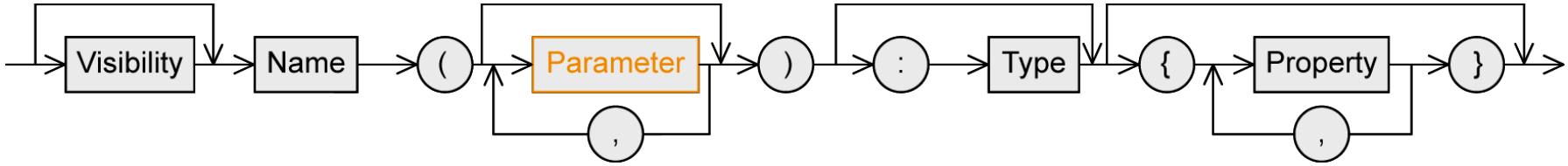
Attribute Syntax – Properties



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

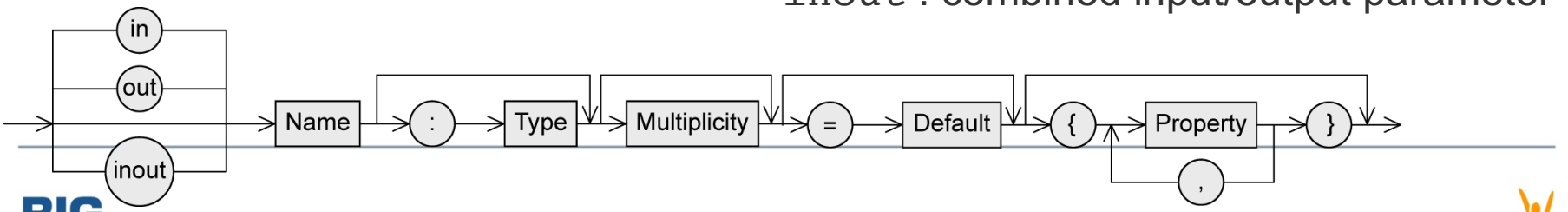
- Pre-defined properties
  - `{readOnly}` … value cannot be changed
  - `{unique}` … no duplicates permitted
  - `{non-unique}` … duplicates permitted
  - `{ordered}` … fixed order of the values
  - `{unordered}` … no fixed order of the values
- Attribute specification
  - Set: `{unordered, unique}`
  - Multi-set: `{unordered, non-unique}`
  - Ordered set: `{ordered, unique}`
  - List: `{ordered, non-unique}`

Operation Syntax - Parameters

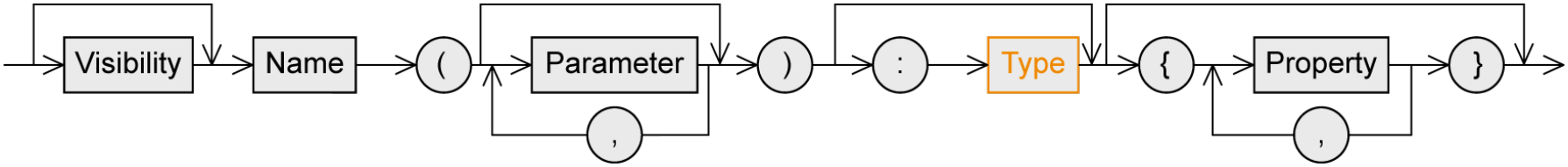


Person
...
+ getName(out fn: String, out ln: String): void + updateLastName(newName: String): boolean + getPersonsNumber(): int

- Notation similar to attributes
- Direction of the parameter
  - `in` … input parameter
    - When the operation is used, a value is expected from this parameter
  - `out` … output parameter
    - After the execution of the operation, the parameter has adopted a new value
  - `inout` : combined input/output parameter



Operation Syntax - Type

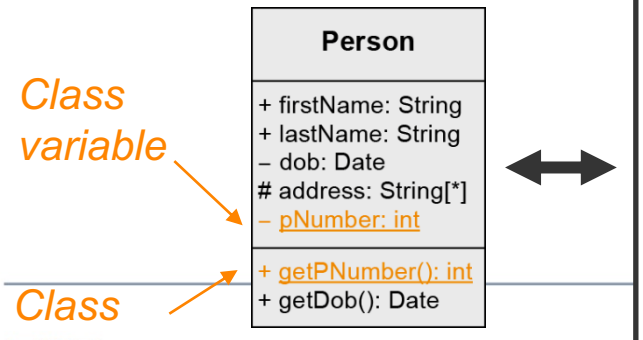


Person
...
getName(out fn: String, out ln: String): void updateLastName(newName: String): boolean getPersonsNumber(): int

- Type of the return value

Class Variable and Class Operation

- Instance variable (= instance attribute): attributes defined on instance level
- Class variable (= class attribute, static attribute)
  - Defined only once per class, i.e., shared by all instances of the class
  - E.g. counters for the number of instances of a class, constants, etc.
- Class operation (= static operation)
  - Can be used if no instance of the corresponding class was created
  - E.g. constructors, counting operations, math. functions (sin(x)), etc.
- Notation: underlining name of class variable / class operation

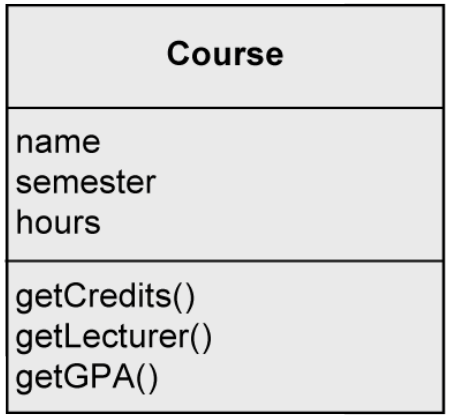


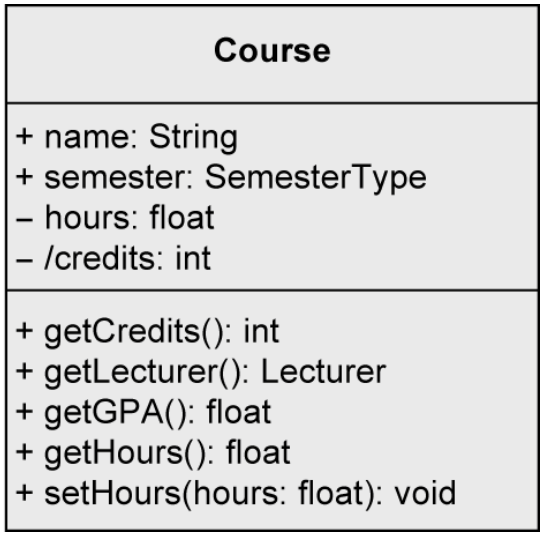

```
class Person {

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

Specification of Classes: Different Levels of Detail

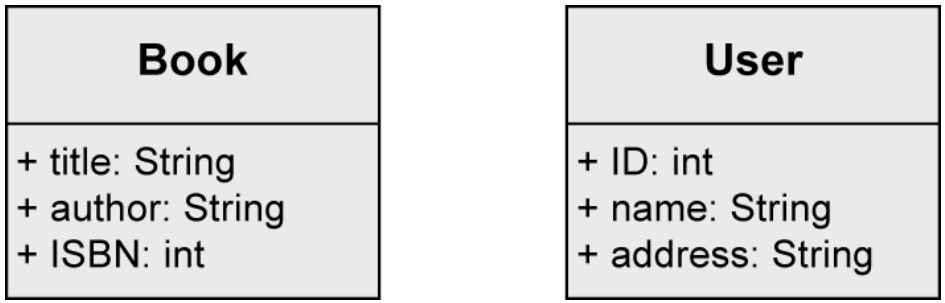
coarse-grained *fine-grained*





Creating a Class Diagram

- Not possible to completely extract classes, attributes and associations from a natural language text automatically.
- Guidelines
  - Nouns often indicate classes
  - Adjectives indicate attribute values
  - Verbs indicate operations
- Example: The library management system stores users with their unique ID, name and address as well as books with their title, author and ISBN number. Ann Foster wants to use the library.



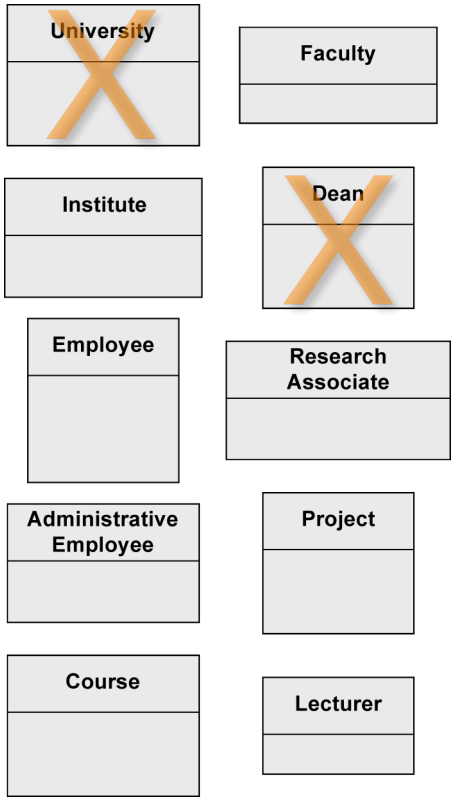
Example – University Information System

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for 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 email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

Example – Step 1: Identifying Classes

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for 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 email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

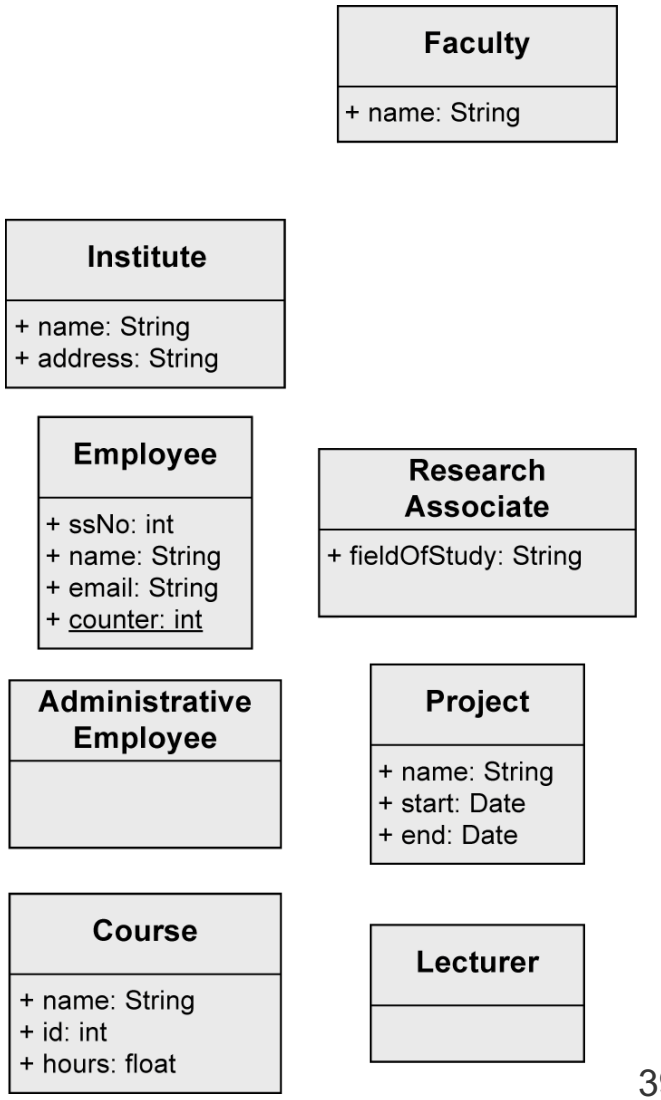
We model the system „University“



Dean has no further attributes than any other employee

Example – Step 2: Identifying the Attributes

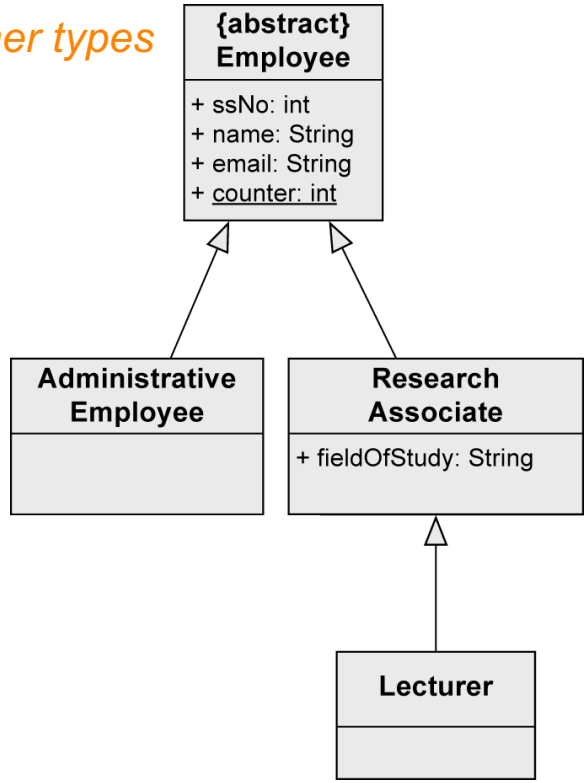
- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.



Example – Step 2: Identifying Relationships (1/6)

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

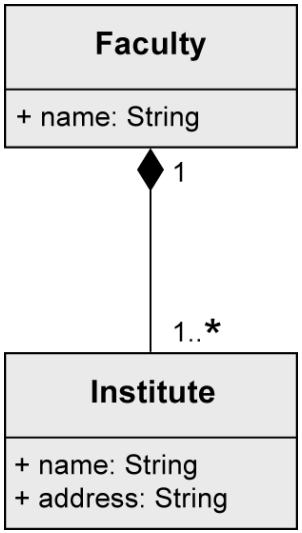
Abstract, i.e., no other types of employees



- Indication of a generalization
- *"There is a distinction between research and administrative personnel."*
- *"Some research associates hold courses. Then they are called lecturers."*

Example – Step 2: Identifying Relationships (2/6)

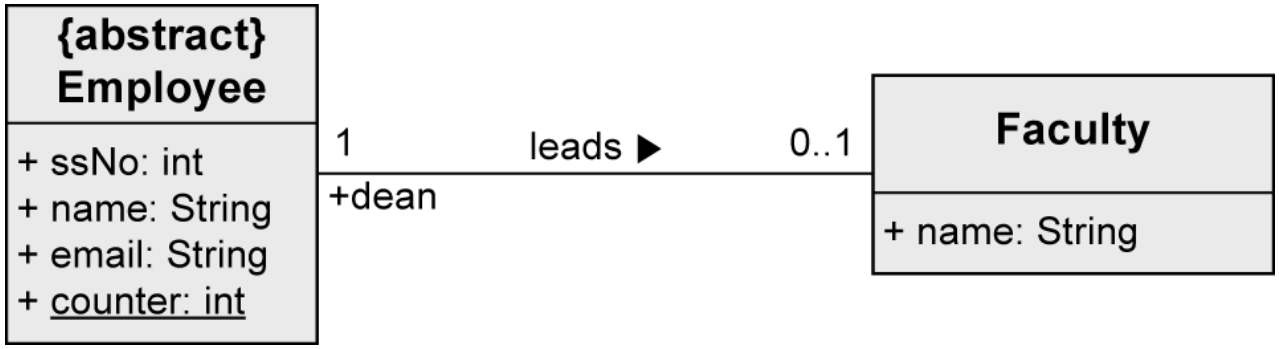
- *"A university consists of multiple faculties which are composed of various institutes."*



Composition to show existence dependency

Example – Step 2: Identifying Relationships (3/6)

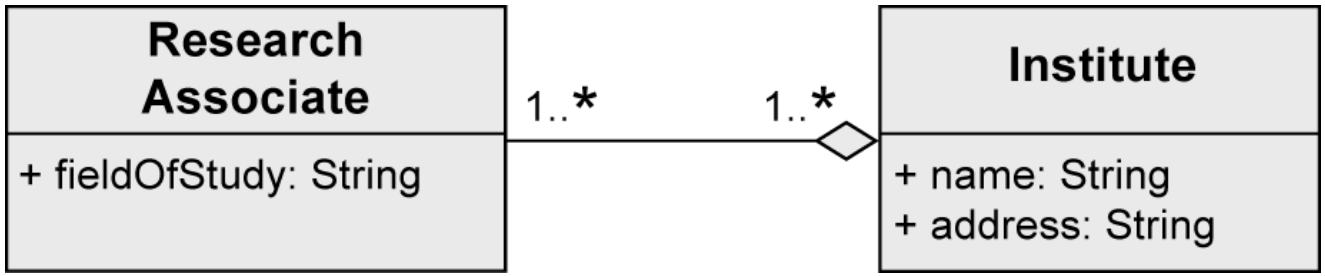
- *"Each faculty is led by a dean, who is an employee of the university"*



In the `leads`-relationship, the `Employee` takes the role of a `dean`.

Example – Step 2: Identifying Relationships (4/6)

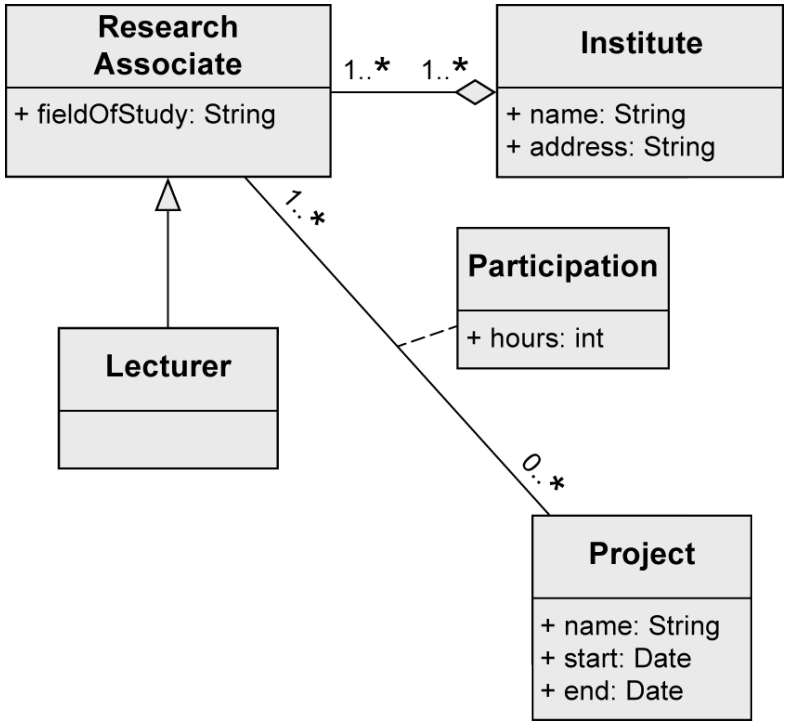
- "Research associates are assigned to at least one institute."



Shared aggregation to show that `ResearchAssociates` are part of an `Institute`, but there is no existence dependency

Example – Step 2: Identifying Relationships (5/6)

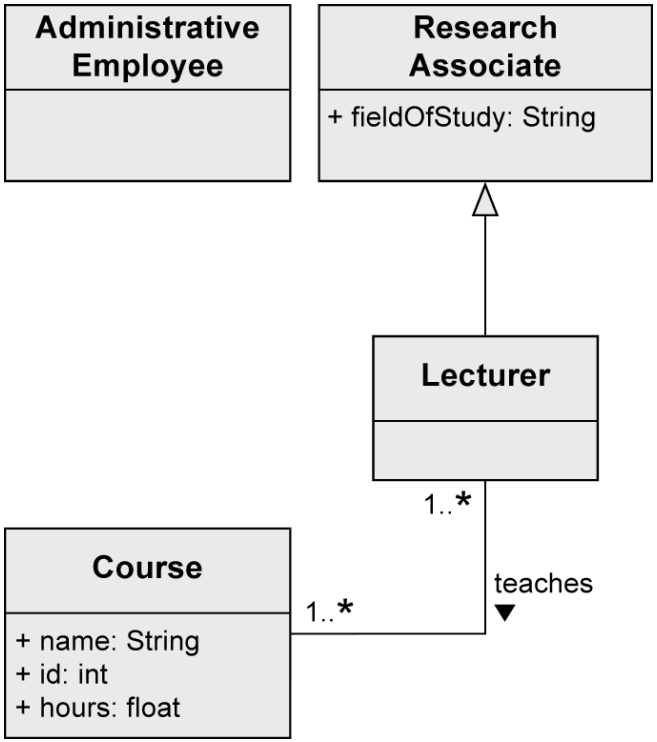
- *"Furthermore, research associates can be involved in projects for a certain number of hours."*



Association class enables to store the number of hours for every single `Project` *of every single* `ResearchAssociate`

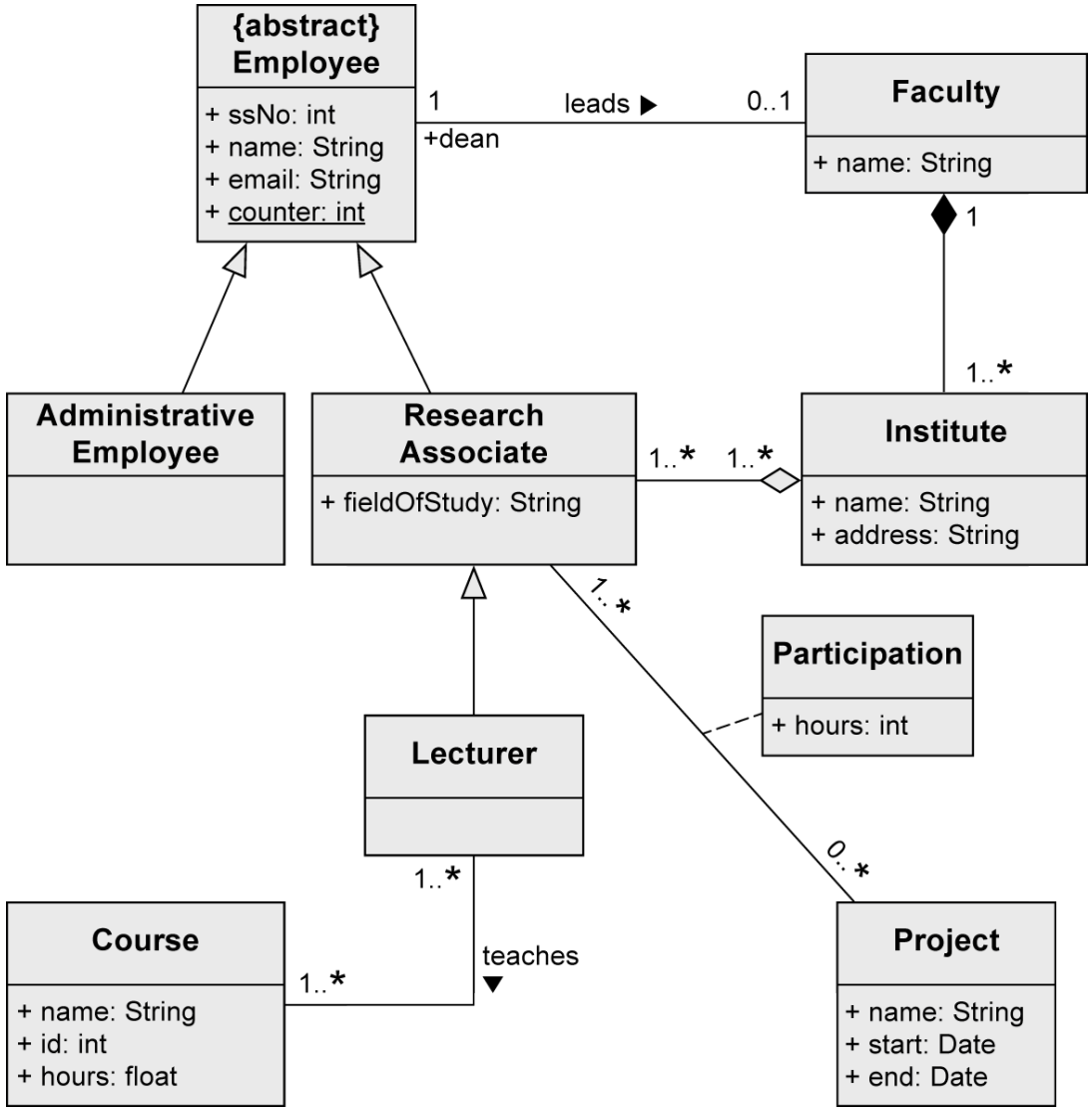
Example – Step 2: Identifying Relationships (6/6)

- *"Some research associates hold courses. Then they are called lecturers."*



`Lecturer` inherits all characteristics, associations, and aggregations from `ResearchAssociate`.
In addtion, a `Lecturer` has an association `teaches` to `Course`.

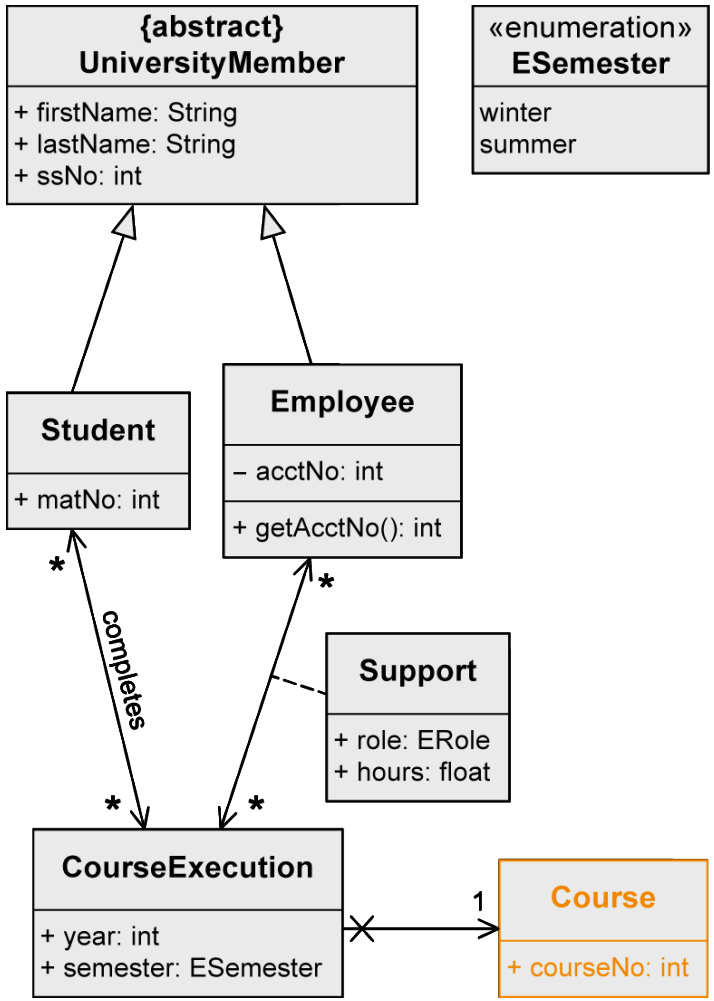
Example – Complete Class Diagram



Code Generation

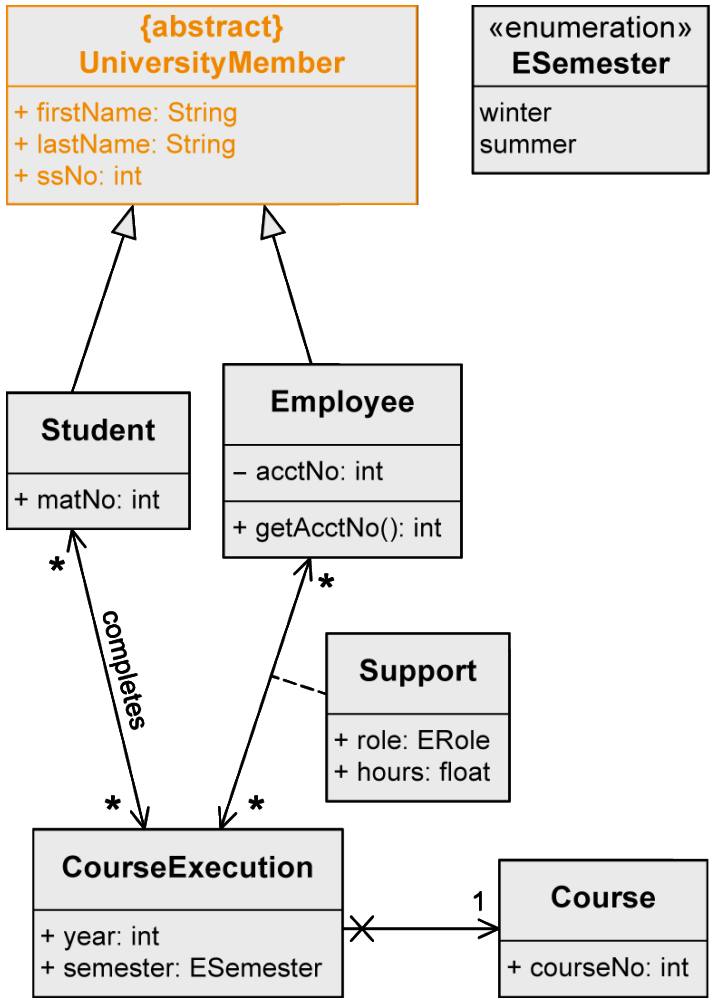
- Class diagrams are often created with the intention of implementing the modeled elements in an object-oriented programming language.
- Often, translation is semi-automatic and requires only minimal manual intervention.

Code Generation – Example (1/6)



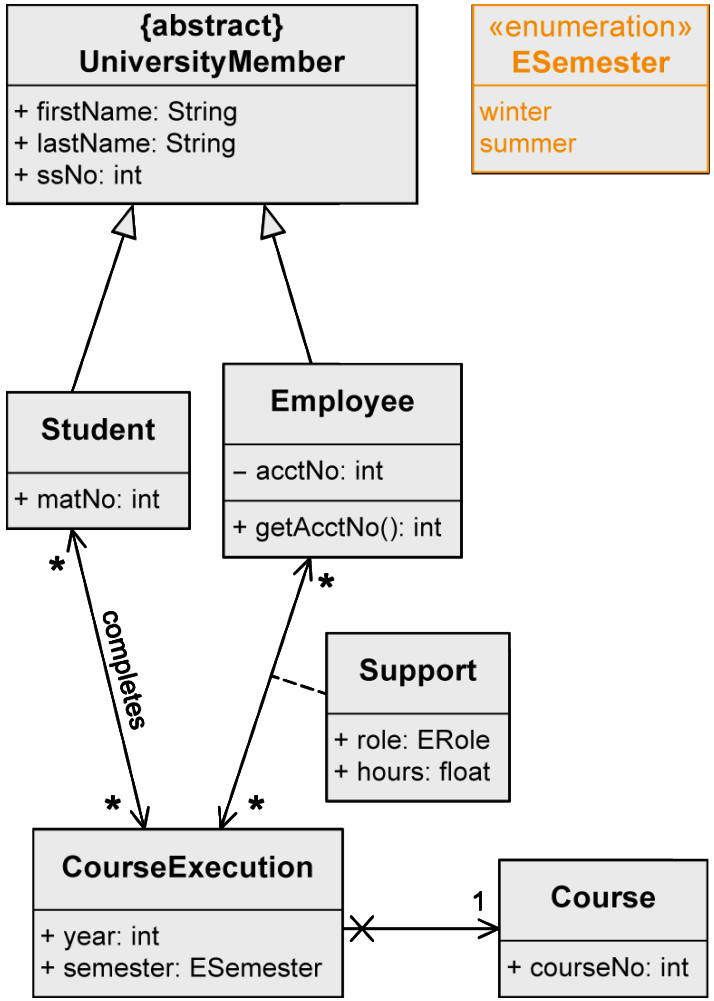

```
class Course {
    public int courseNo;
}
```

Code Generation – Example (2/6)




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

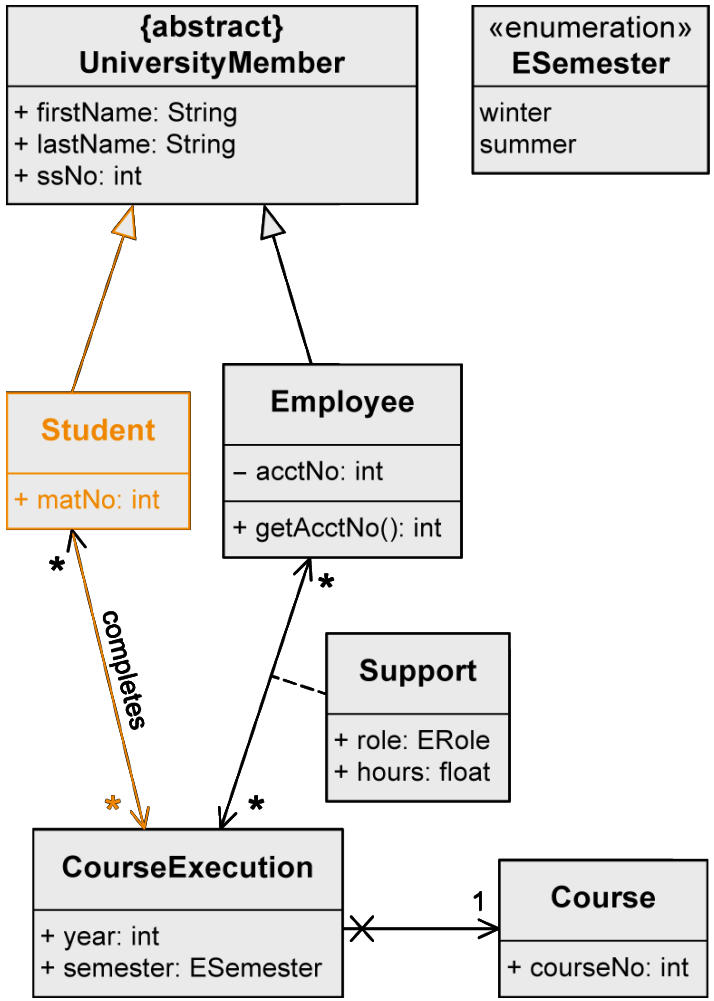
Code Generation – Example (3/6)




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

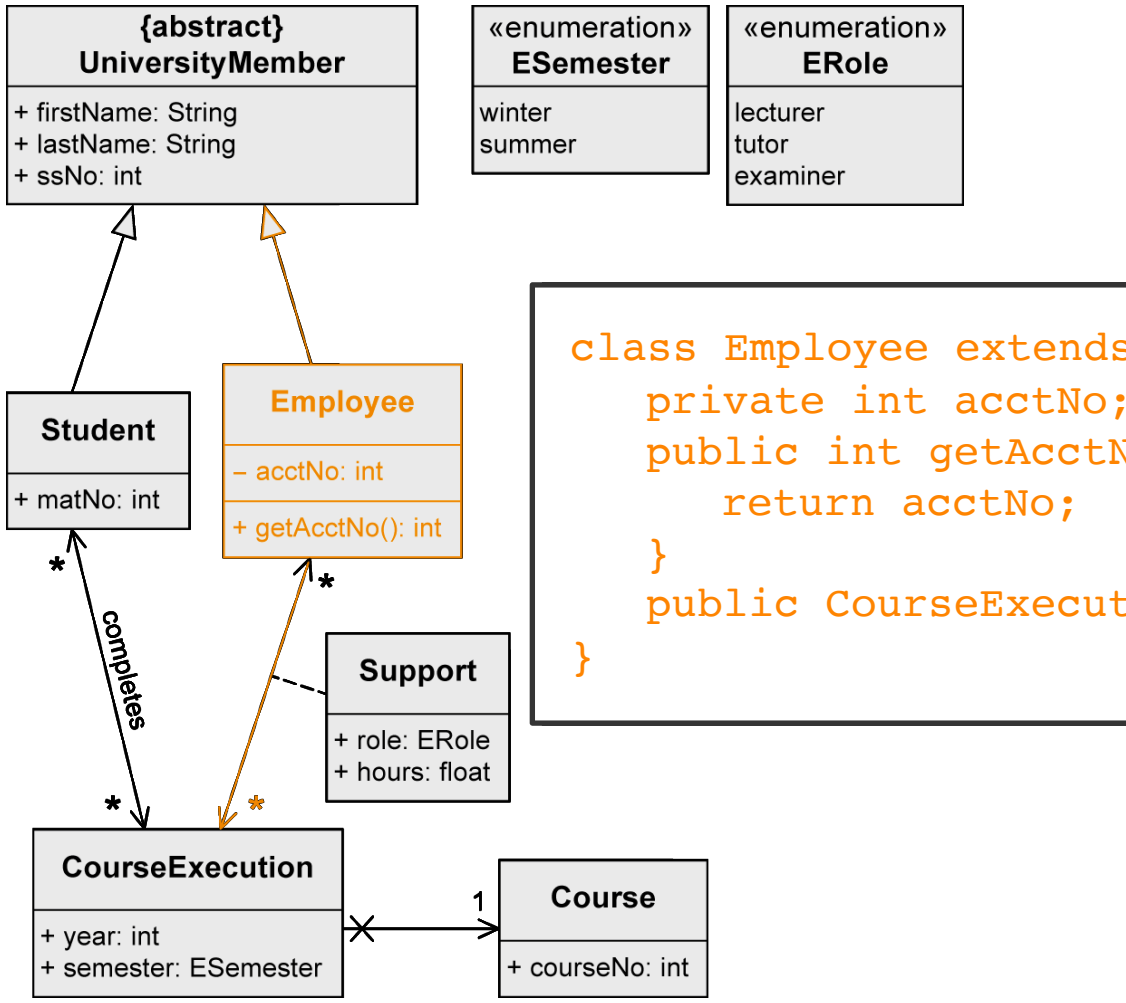
```
Enumeration ERole {
    lecturer,
    tutor,
    examiner
}
```

Code Generation – Example (4/6)



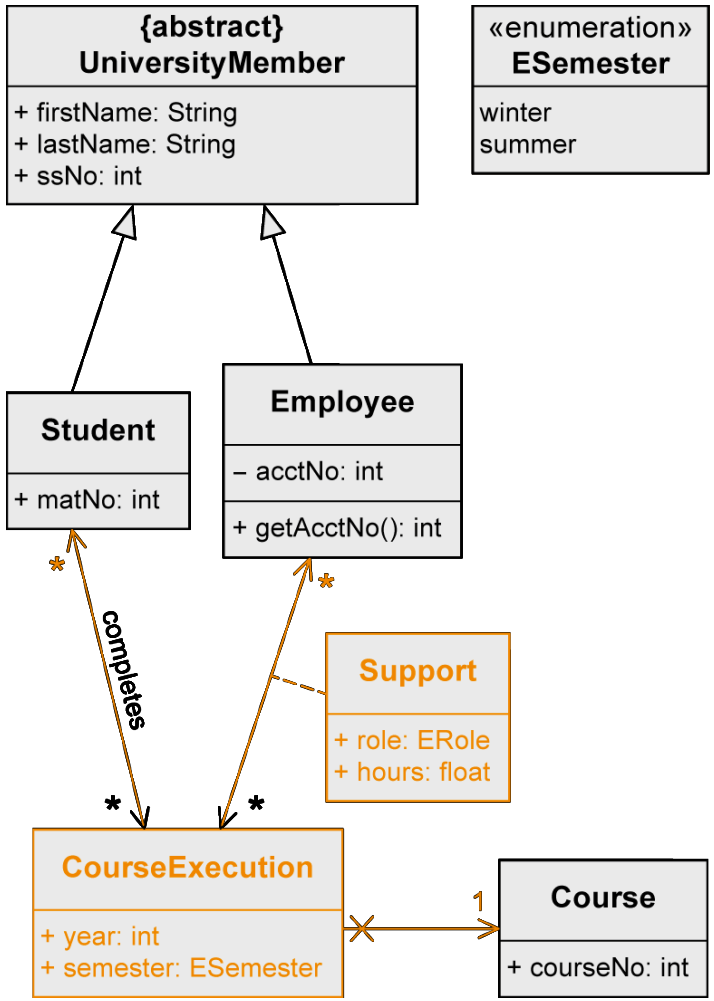

```
class Student extends
                UniversityMember {
   public int matNo;
   public CourseExecution []
                completedCourses;
}
```

Code Generation – Example (5/6)



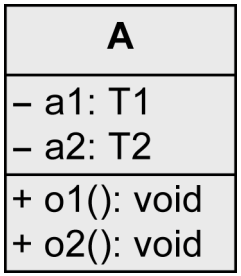
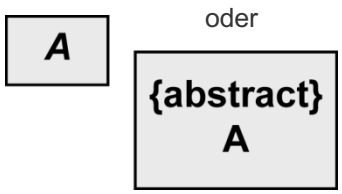
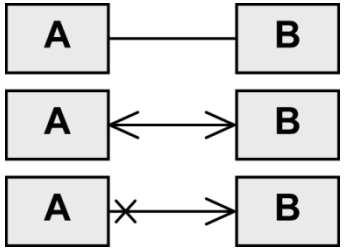

```
class Employee extends UniversityMember {
   private int acctNo;
   public int getAcctNo () {
      return acctNo;
   }
   public CourseExecution [] courseExecutions;
}
```

Code Generation – Example (6/6)

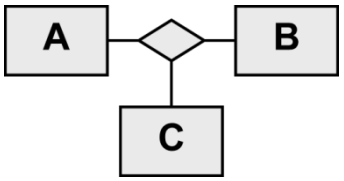
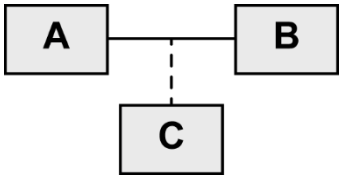
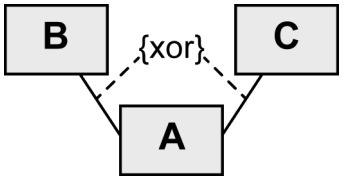



```
class CourseExecution {
   public int year;
   public ESemester semester;
   public Student [] student;
   public Course course;
   public Map support;
      // Key: employee
      // Value: (role, hours)
}
```

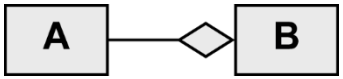
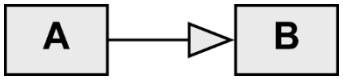
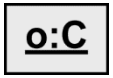
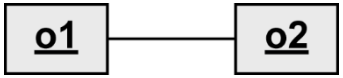
Notation Elements (1/3)

Name	Notation	Description
Class		Description of the structure and behavior of a set of objects
Abstract class		Class that cannot be instantiated
Association		Relationship between classes: navigability unspecified, navigable in both directions, not navigable in one direction

Notation Elements (2/3)

Name	Notation	Description
n-ary association		Relationship between n (here 3) classes
Association class		More detailed description of an association
`xor` relationship		An object of **`C`** is in a relationship with an object of **`A`** or with an object of **`B`** but not with both

Notation Elements (3/3)

Name	Notation	Description
Shared aggregation		Parts-whole relationship (**A** is part of **B**)
Strong aggregation = composition		Existence-dependent parts-whole relationship (**A** is part of **B**)
Generalization		Inheritance relationship (**A** inherits from **B**)
Object		Instance of a class
Link		Relationship between objects