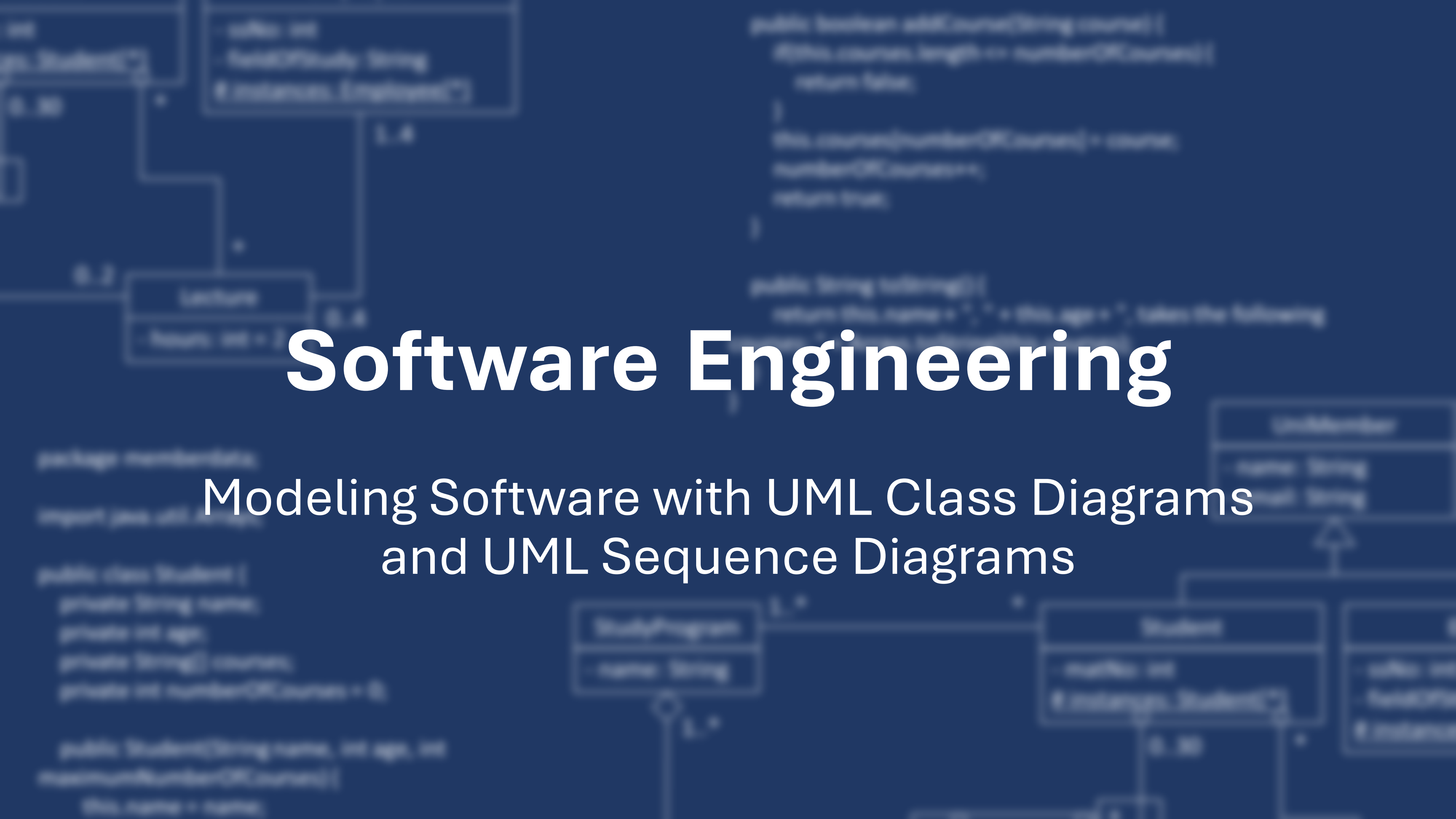


Software Engineering

Modeling Software with UML Class Diagrams and UML Sequence Diagrams

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

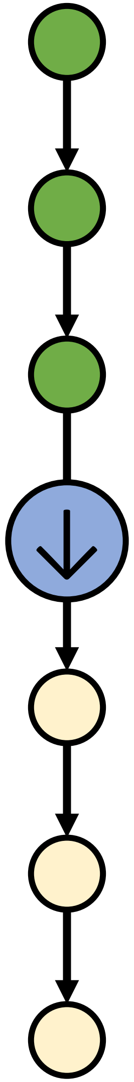
Cover advanced object-oriented principles.

How to model OO systems (UML) and map models to code.

Object-oriented modeling techniques.

Design patterns as means to realize OO concepts (I).

Design patterns as means to realize OO concepts (II).



Last Lecture

In last lecture:

- Generic programming.
- Lambdas and streams.
- Exception handling.
- Concurrency via threads.
- Object-based programming via JavaScript.

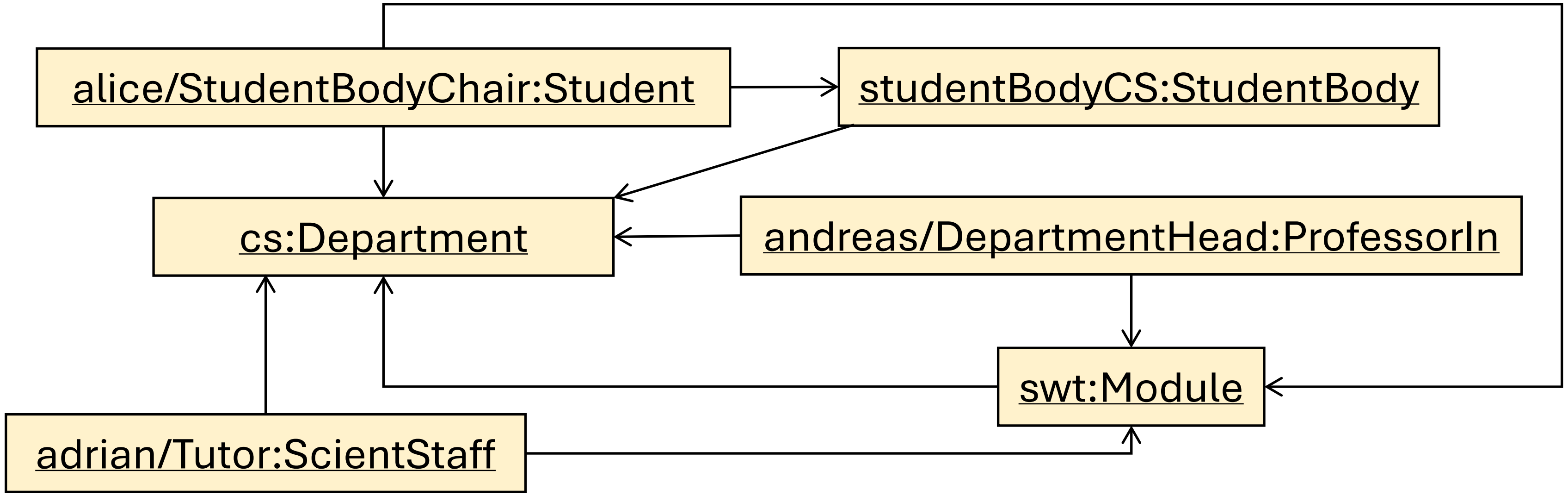
Aims of this Lecture

How to model (OO) systems?

- Object diagrams for example runtime configurations.
- Class diagrams for the structure of a system.
- Sequence diagrams for the behavior of a system.

How to map models to code?

- Class diagrams → code.
- Sequence diagrams → code.
- Activity diagrams → code.



alice/StudentBodyChair:Student

studentBodyCS:StudentBody

cs:Department

andreas/DepartmentHead:ProfessorIn

swt:Module

adrian/Tutor:ScientStaff

Object Diagrams

Objects

Objects are represented as rectangles.

The name field of objects are underlined and comprise of the following (in this exact order):

- An Identifier for the instance (optional)
- A "/" followed by the role of this object (optional)
- A ":" as divider (mandatory) followed by the Type of the instance (optional)

alice/StudentBodyChair:Student

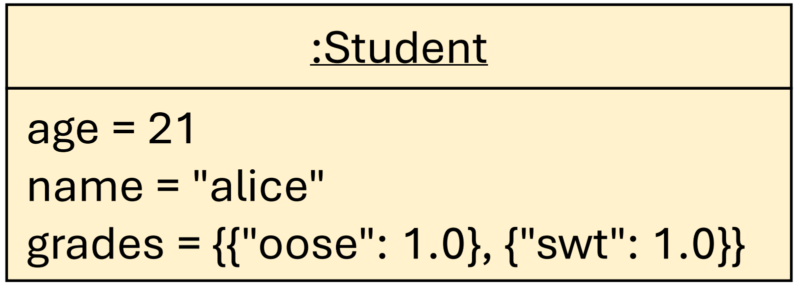
alice:Student

:Student

Objects

The objects' attributes together with their values can be displayed (optional).

Methods are not included in an object when using class-based programming languages.



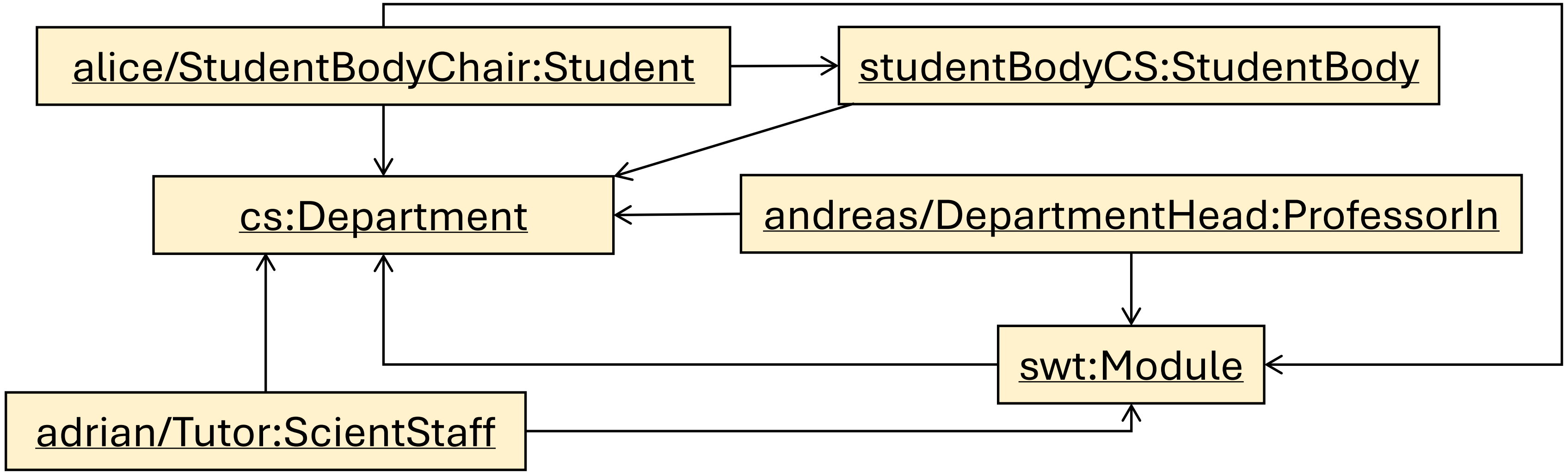
:Student

age = 21
name = "alice"
grades = {{"oose": 1.0}, {"swt": 1.0}}

Relations between Objects

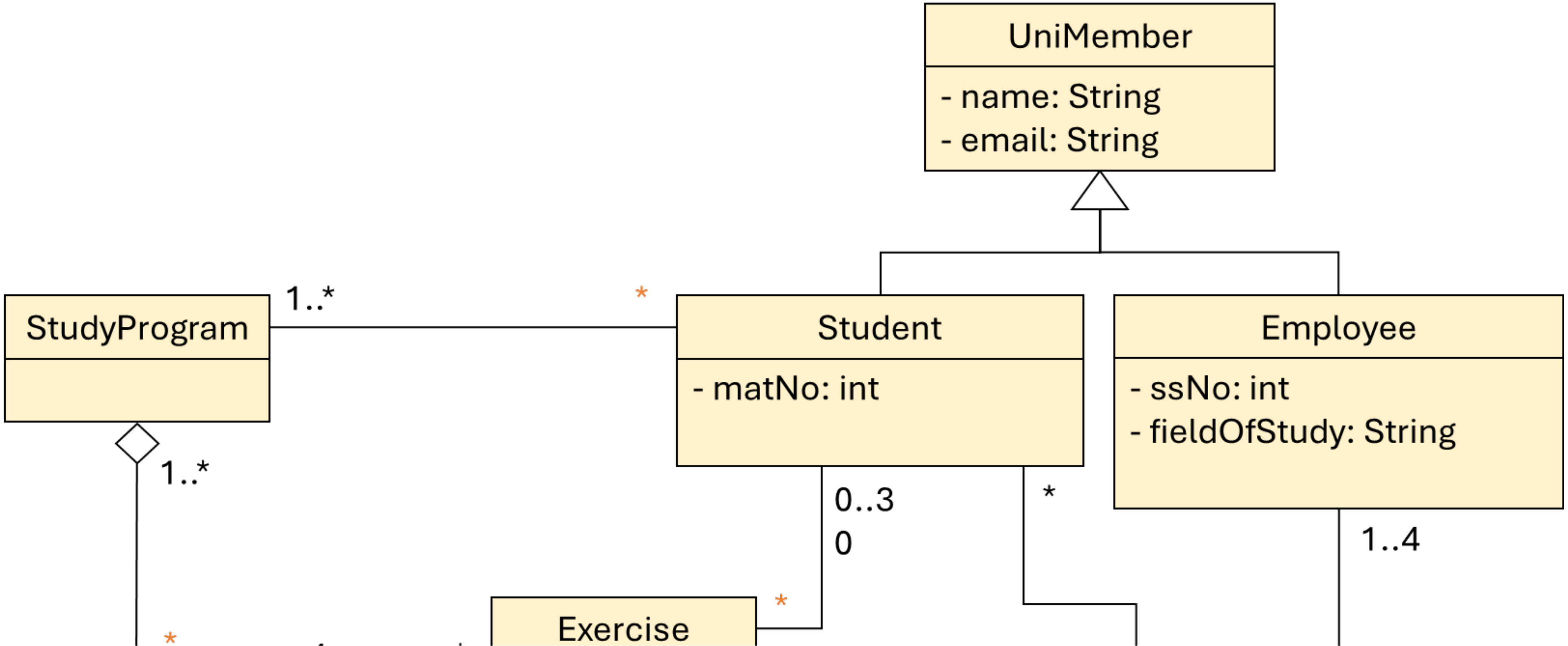
Relations are displayed using lines or arrows:

- Directed line (arrow) = specified navigability
- Undirected line = unspecified navigability


alice/StudentBodyChair:Student
studentBodyCS:StudentBody
cs:Department
andreas/DepartmentHead:ProfessorIn
swt:Module
adrian/Tutor:ScientStaff

Not according to standard, but sometimes helpful:

- Aggregation and Composition
- Roles



Class Diagrams

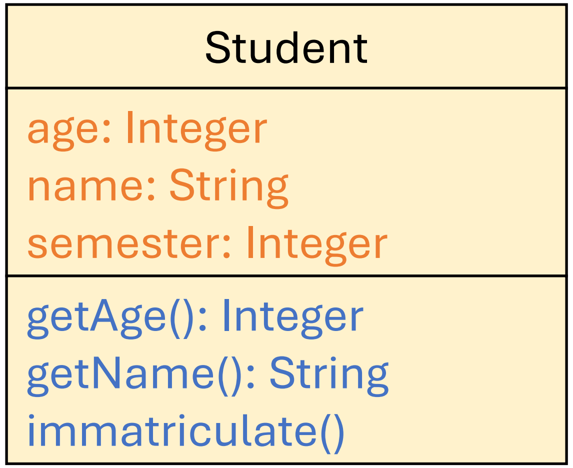
Classes

A class represents a concept.

A class encapsulates state (attributes) and behavior (operations).

The classes' name is the only mandatory aspect.

- Usually, more information is added during the development process.
- Information not important to the context may be "hidden" (not displayed).

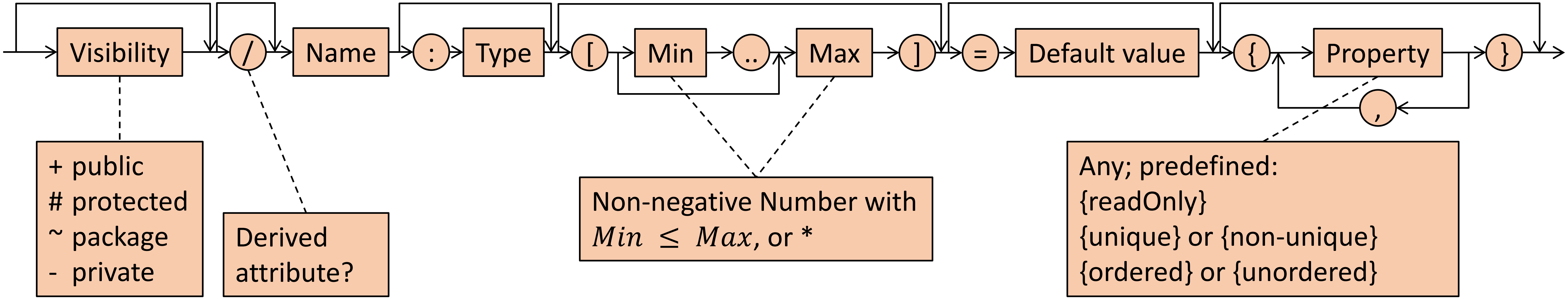


Student

age: Integer
name: String
semester: Integer

getAge(): Integer
getName(): String
immatriculate()

Attributes

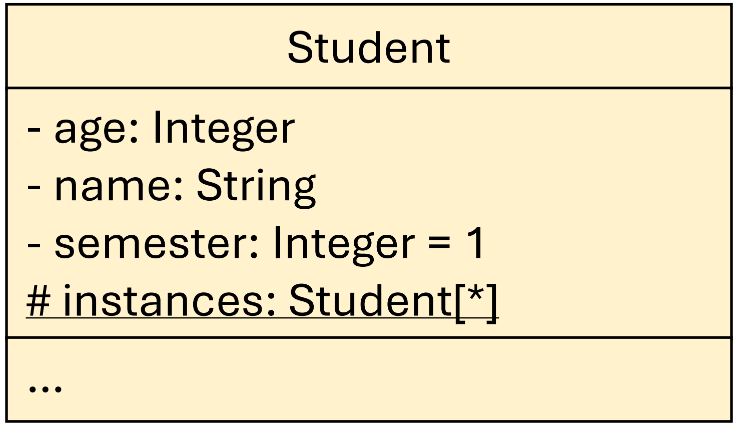


+ public
# protected
~ package
- private

Derived attribute?

Non-negative Number with $Min \leq Max$, or *

Any; predefined:
{readOnly}
{unique} or {non-unique}
{ordered} or {unordered}



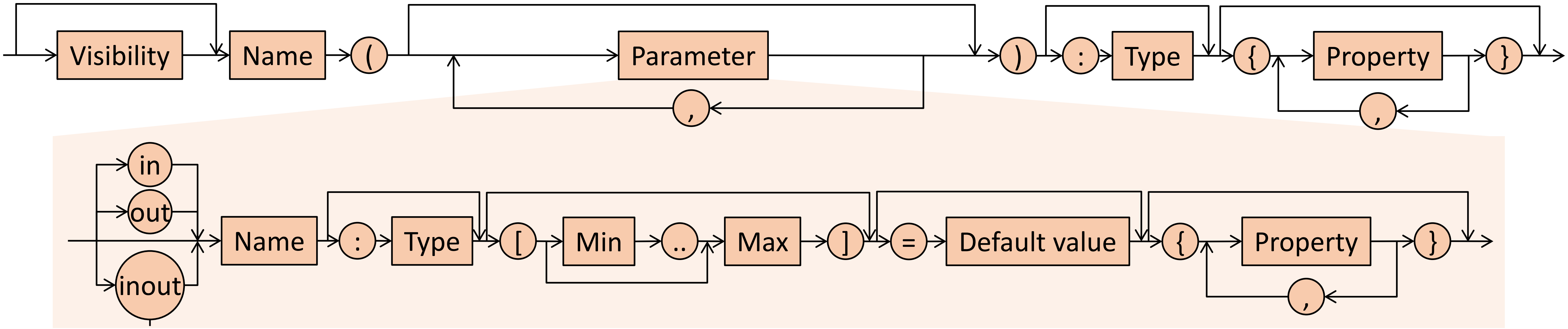
Student

- age: Integer
- name: String
- semester: Integer = 1

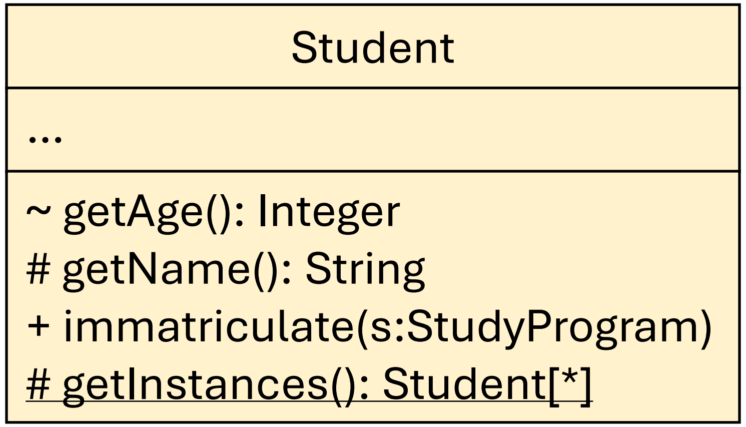
# instances: Student[*]

...

Operations



Input-, Output-, or Inout-Parameter.



Student

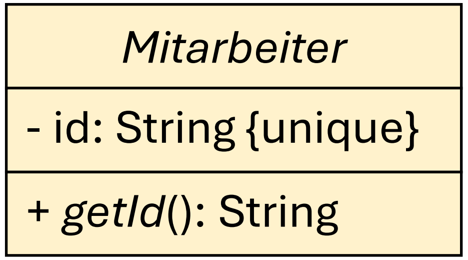
...

~ getAge(): Integer
# getName(): String
+ immatriculate(s:StudyProgram)
# getInstances(): Student[*]

Further Notation

What the previous syntax diagrams could not show:

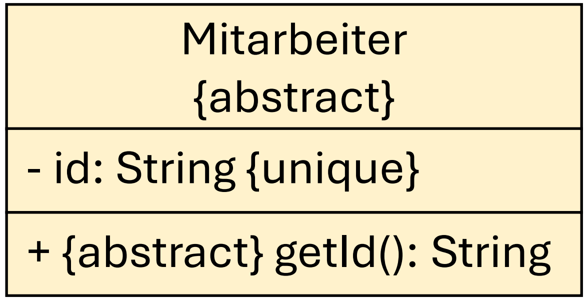
- Class methods and variables are underlined
- Abstract *classes* and *methods* are in italic



Mitarbeiter

- id: String {unique}

+ *getId*(): String

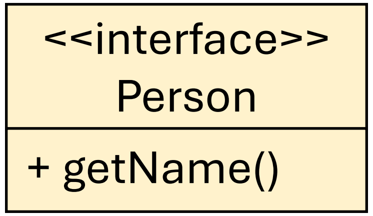


Mitarbeiter
{abstract}

- id: String {unique}

+ {abstract} getId(): String

- Interfaces are annotated with the stereotype <<interface>> and never hold any attributes



<<interface>>
Person

+ getName()

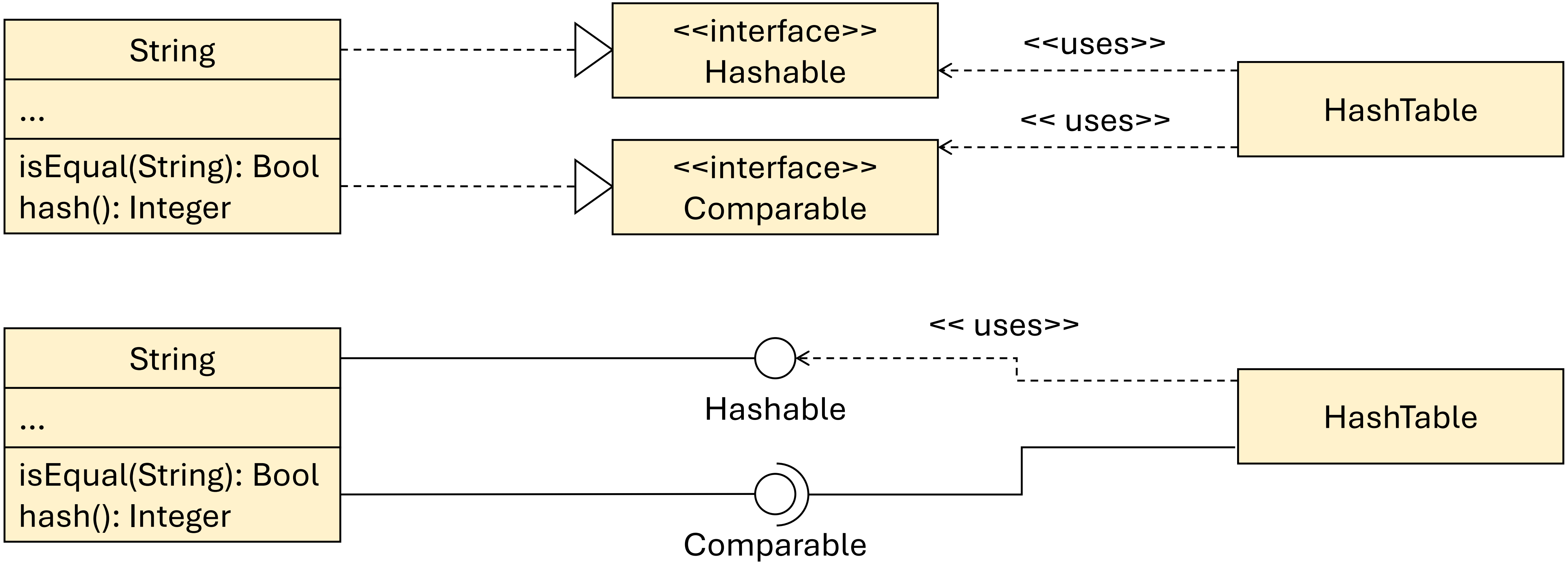
Implements and Dependency Relation

Interface Implementation

Class realizes interface

Dependency

Class depends on other class



Generalization

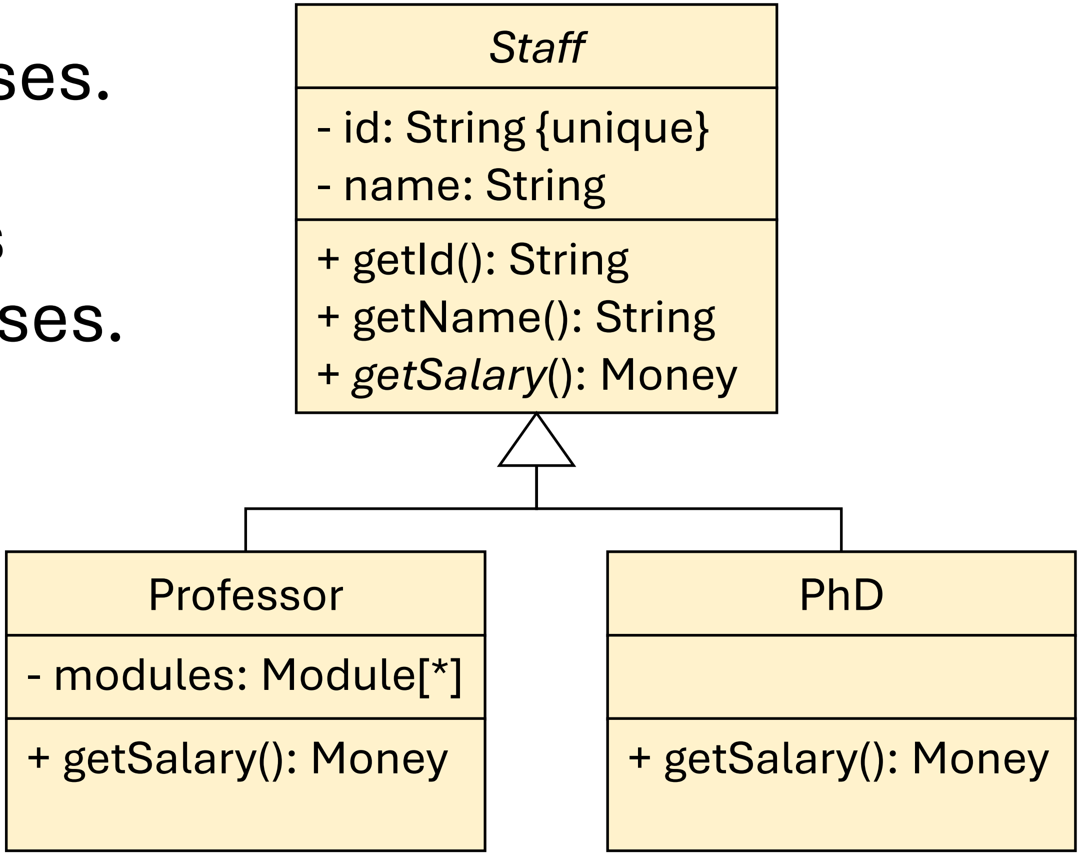
Inheritance: Inherited methods and attributes are not repeated in subclasses.

Hiding: Redefined attributes and class methods must be repeated in subclasses.

Overriding: Overridden methods are repeated in subclasses.

Overloading

- All different signatures are specified.
- Overloaded methods can be overridden in subclasses as well!



Staff

- id: String {unique}
- name: String

+ getId(): String
+ getName(): String
+ *getSalary*(): Money

Professor

- modules: Module[*]

+ getSalary(): Money

PhD

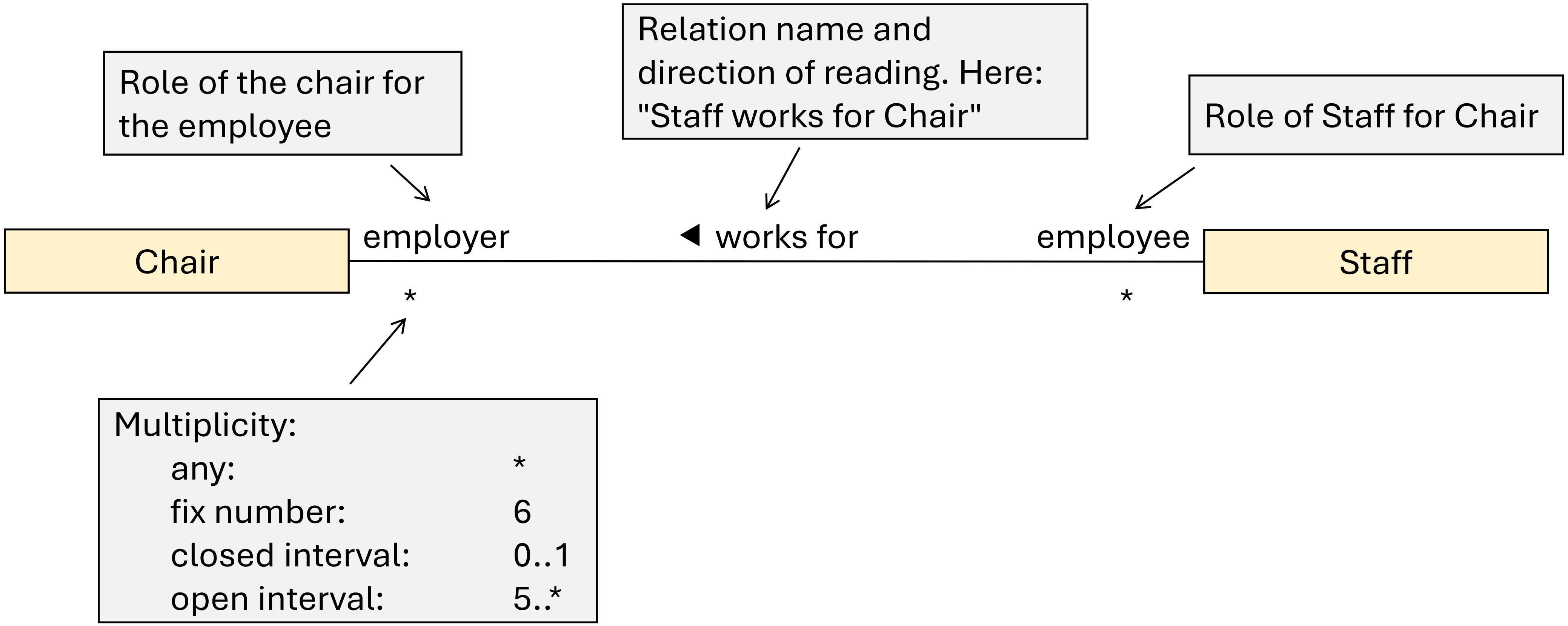
+ getSalary(): Money

Questions?

Associations

Associations

Associations define relations between instances of classes.



Role of the chair for the employee

Relation name and direction of reading. Here: "Staff works for Chair"

Role of Staff for Chair

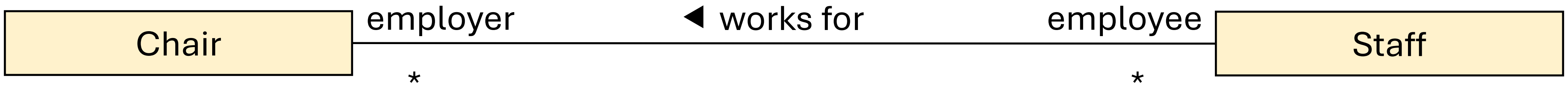
Multiplicity:

any:	*
fix number:	6
closed interval:	0..1
open interval:	5..*

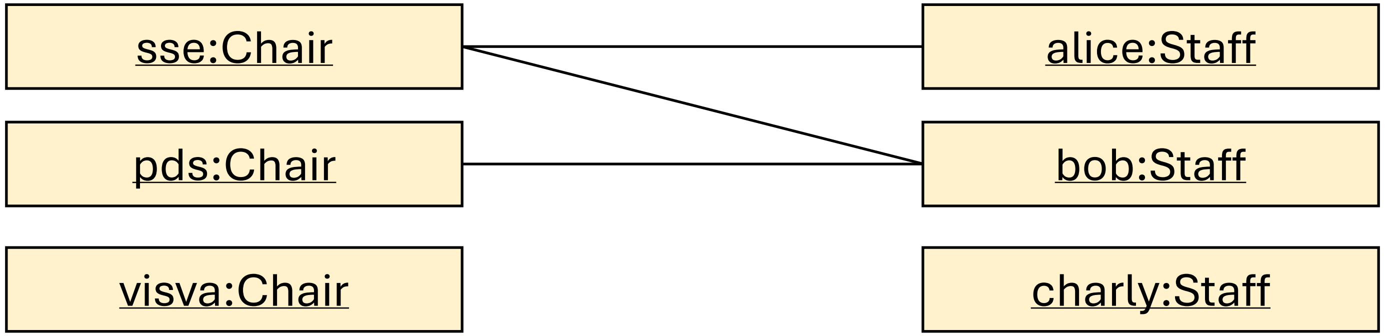
Multiplicities

The multiplicity on one end defines with how many instances of the class at this end and instance of the class on the other end can be related.

- Example: N to M association



Matching object diagram:



Legal: multiple employees

Legal: one employee

Legal: no employee

Legal: one jobs

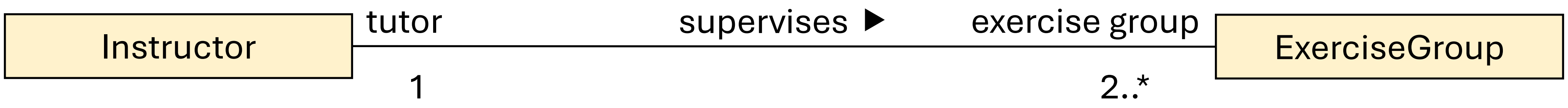
Legal: multiple jobs

Legal: no job

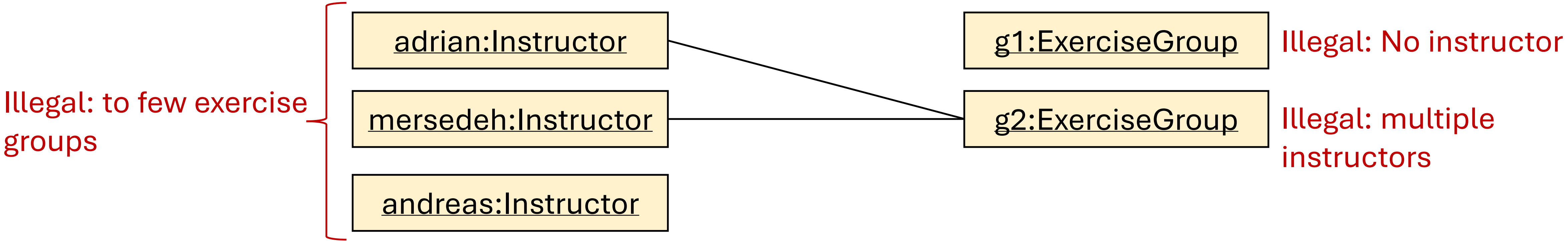
Multiplicities

The multiplicity on one end defines with how many instances of the class at this end and instance of the class on the other end can be related.

- Example: 1 to N association



Illegal object diagram:



Illegal: to few exercise groups

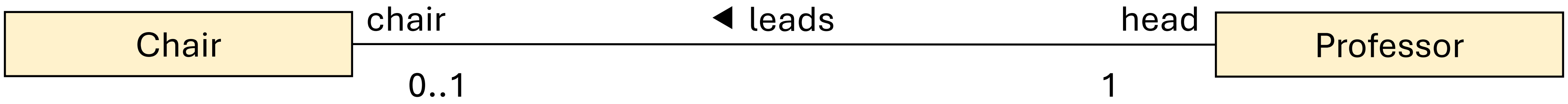
Illegal: No instructor

Illegal: multiple instructors

Multiplicities

The multiplicity on one end defines with how many instances of the class at this end and instance of the class on the other end can be related.

- Example: 1 to 1 association

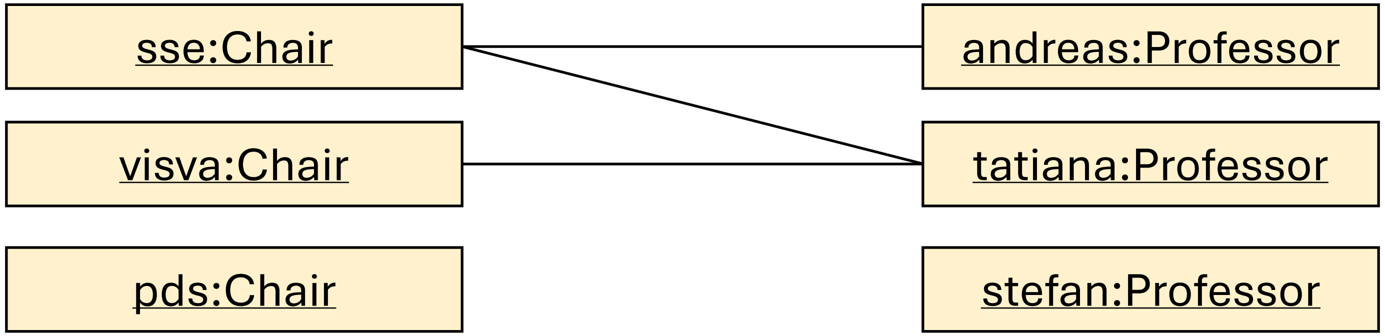


Illegal object diagram:

Illegal: multiple heads

Legal: exactly one head

Illegal: no head



Legal: exactly one chair

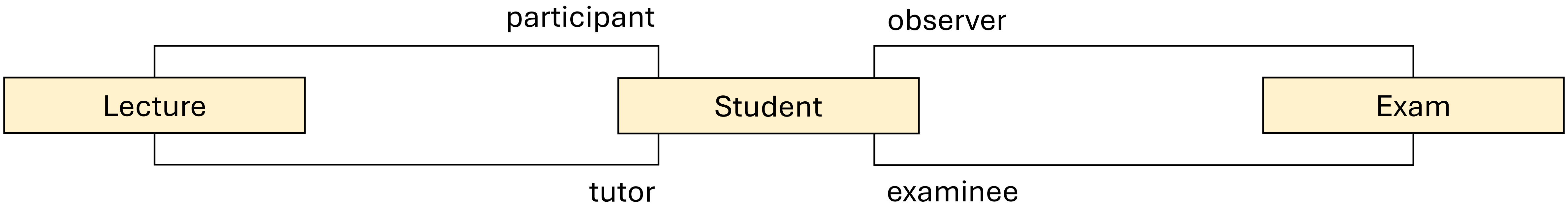
Illegal: leads multiple chairs

Legal: leads no chair

Roles

Roles describe the function of an object within an association

- The same classes can be related via different associations because their instances "play" different roles:



Associations: Navigation

Unspecified Navigation

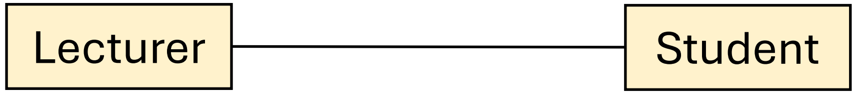
- Syntax: Simple line end without arrow or cross ———
- Semantic:
  - Navigation is not (yet) fully specified
  - Navigation might be possible
- Application: Conceptual view

Explicit Navigation

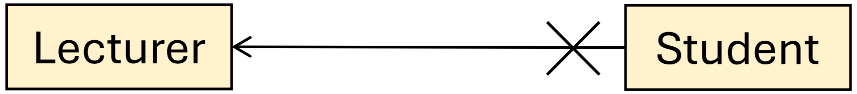
- Syntax:
  - Arrow (= navigable end) ⟶
  - Cross (= non-navigable end) —×
- Semantic:
  - Navigation to arrow end possible
  - Navigation to cross end not possible
- Application: Implementation View

Associations: Navigation

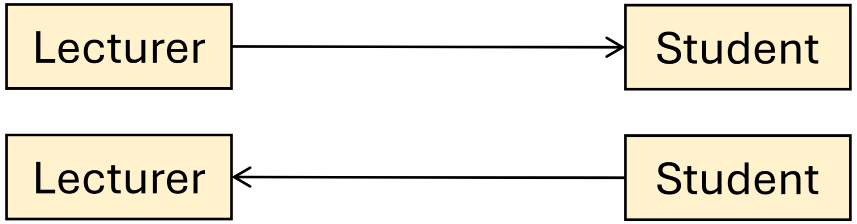
Unspecified Navigation



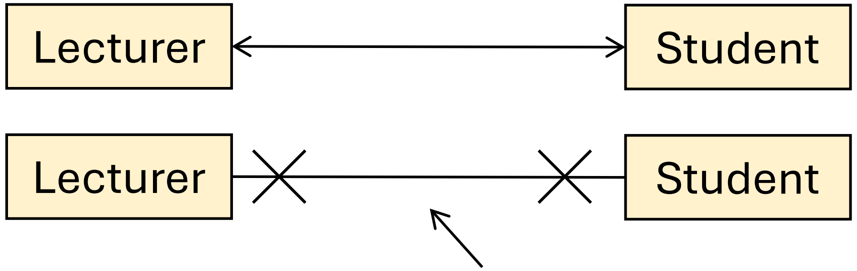
Unidirectional Navigation



Partially specified



Bidirectional Navigation



"What's that?"
→ Hint to association classes

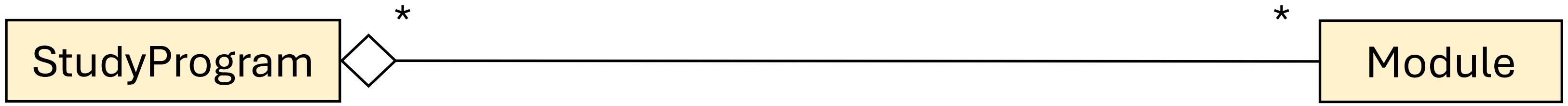
Questions?

Aggregation and Composition

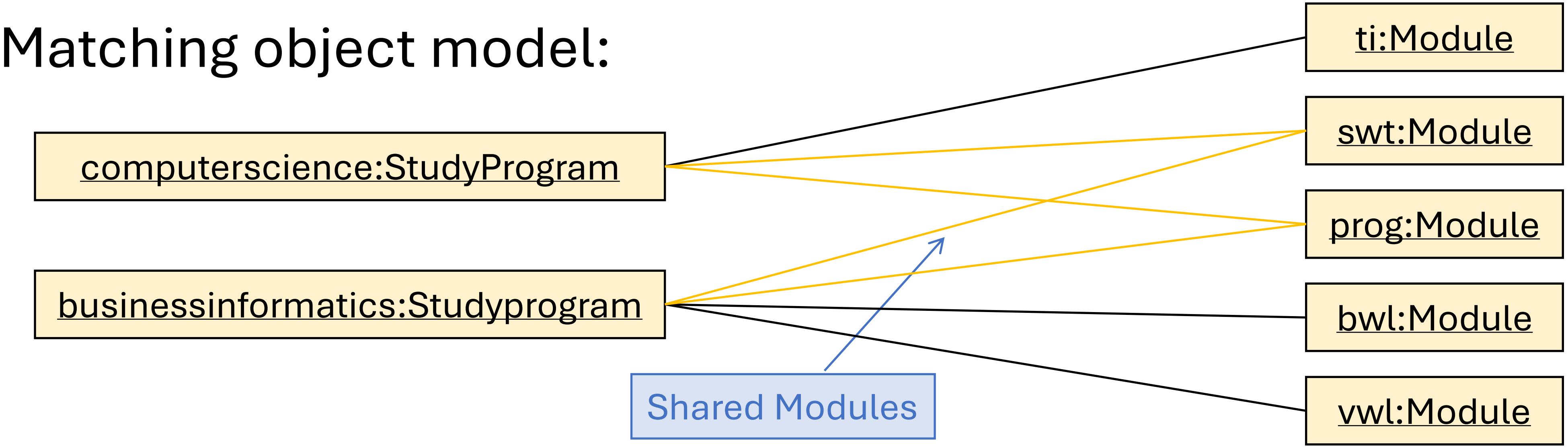
Aggregation

Aggregation = "part of"

- The "part" may be part of one or many "wholes".
- Lifetime of the "part" is not dependent on the "whole".



Matching object model:

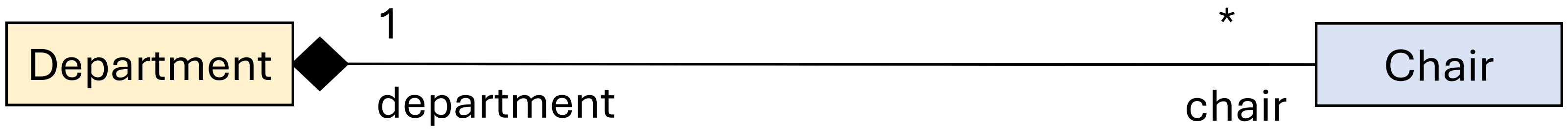


Composition

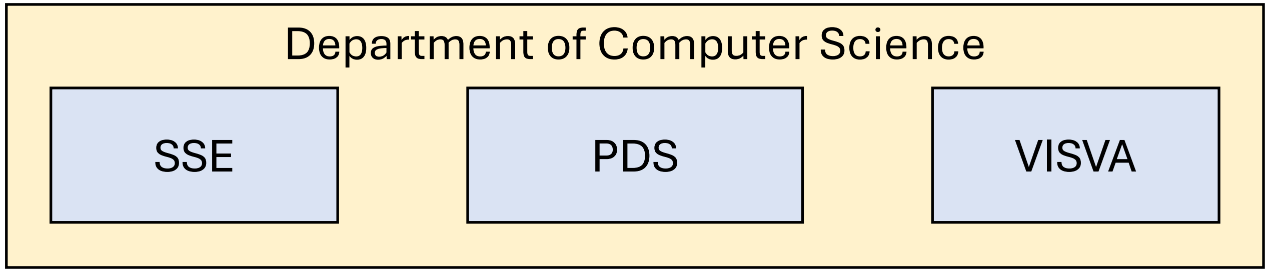
Composition = "is exclusive and existence-dependent part of"

- Part can only be part of at most one whole.
- Lifetime of the part depends on lifetime of the whole.

Example: A chair is part of exactly one Department and cannot exists without it.



Matching example of the real world:

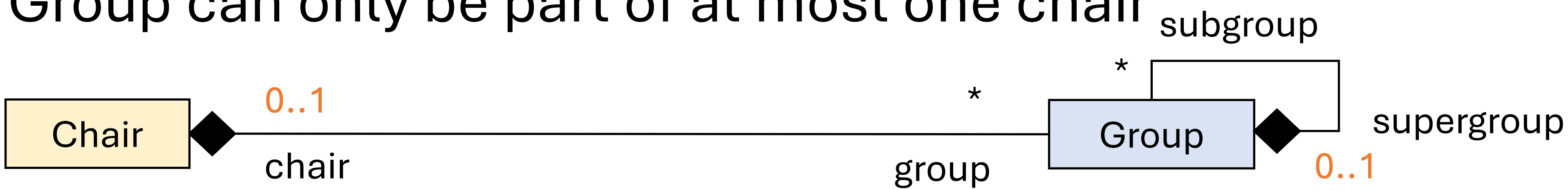


Hierarchical Composition

Composition = "is exclusive and existence-dependent part of"

- Part can only be part of at most one whole
- Lifetime of the part depends on lifetime of the whole

Example: Group can only be part of at most one chair

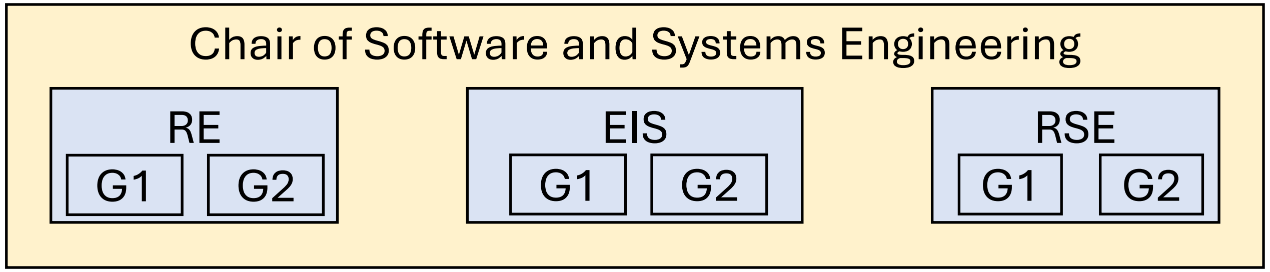


{ context Group: (chair != null)
implies (supergroup = null) }

{ context Group: (supergroup != null)
implies (chair = null) }

Subgroup is not directly part of a chair

- Multiplicity
- Explicit constraints

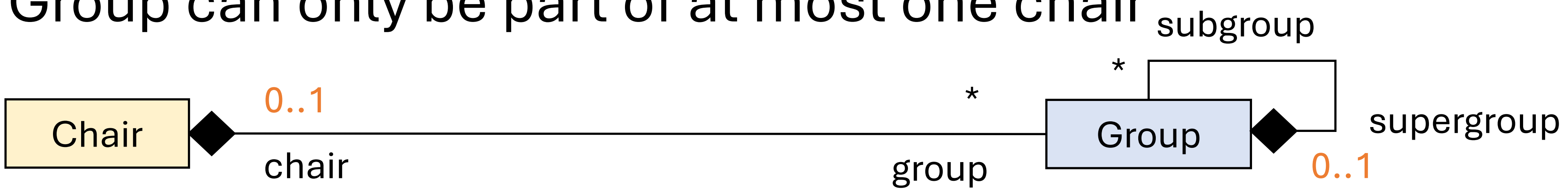


Hierarchical Composition

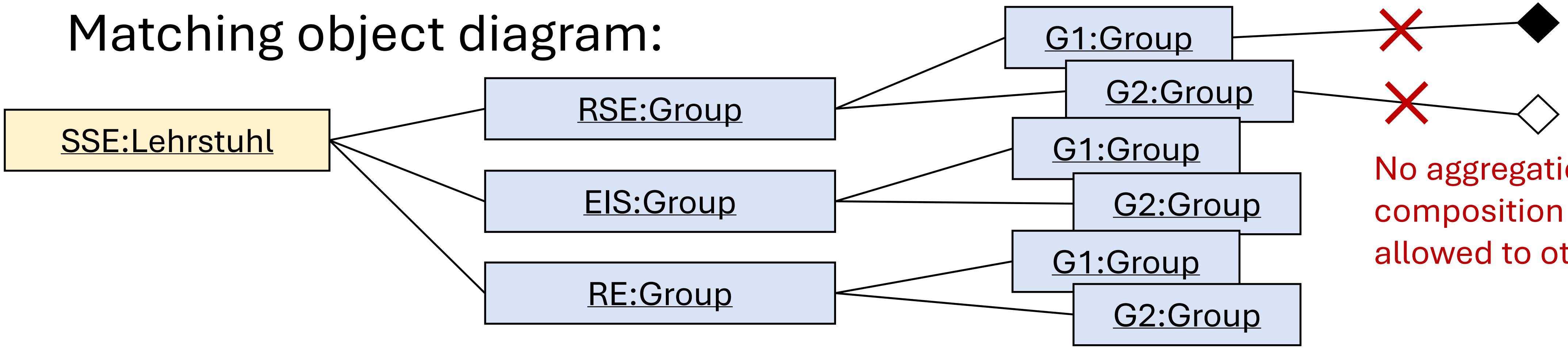
Composition = "is exclusive and existence-dependent part of"

- Part can only be part of at most one whole
- Lifetime of the part depends on lifetime of the whole

Example: Group can only be part of at most one chair



Matching object diagram:

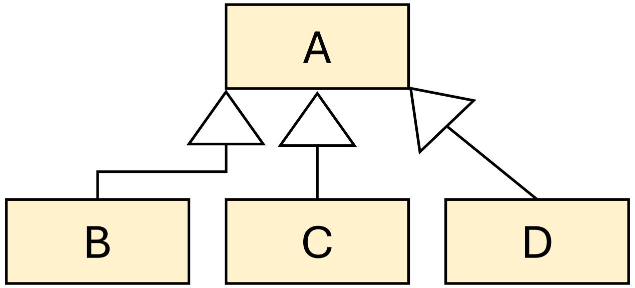


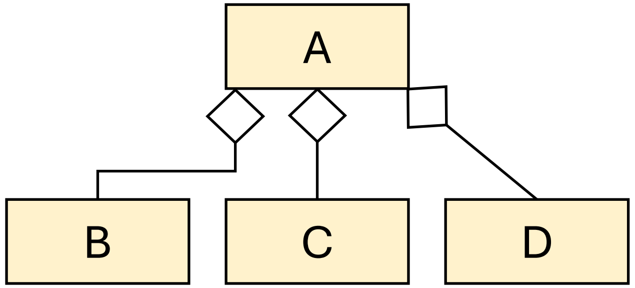
No aggregation or composition relations allowed to other objects!

Questions?

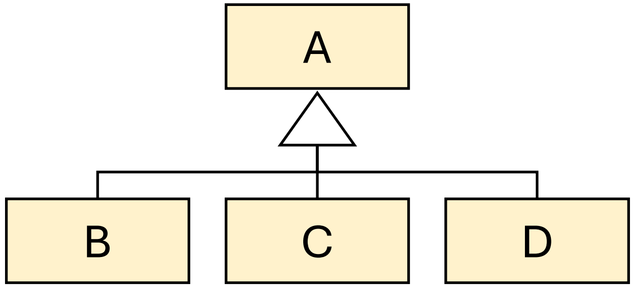
Depiction of Relations

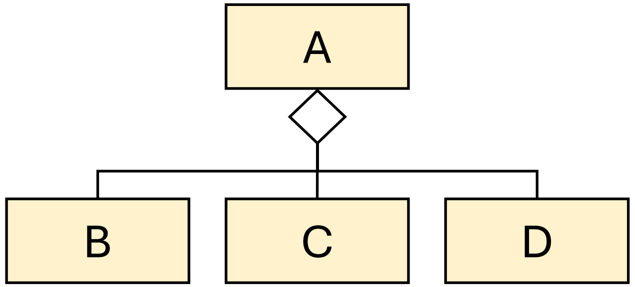
- Explicit Depiction:

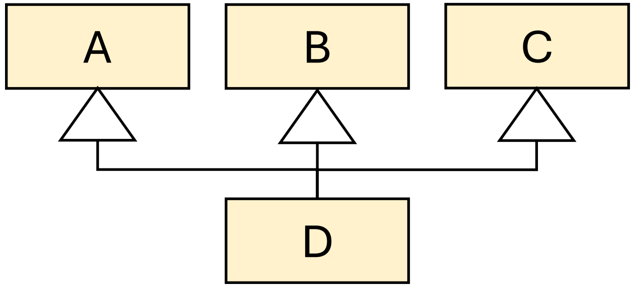


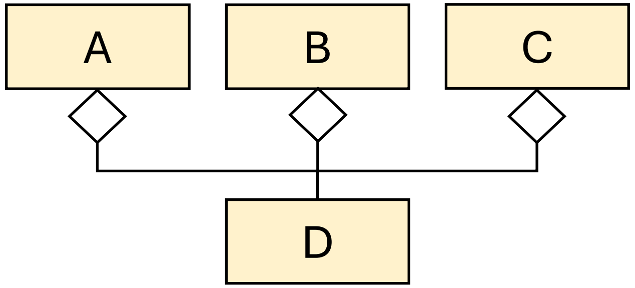


- Compact depiction:









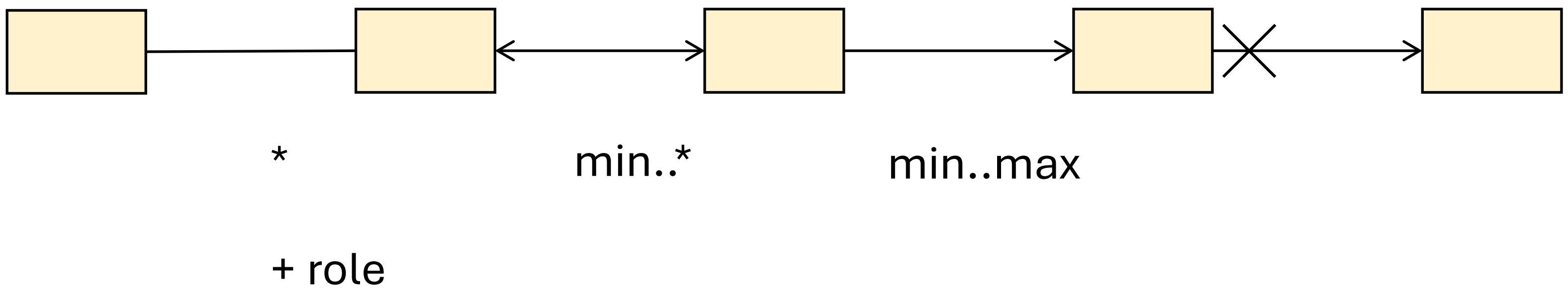
Summary

(so far)

Class Diagrams: Overview

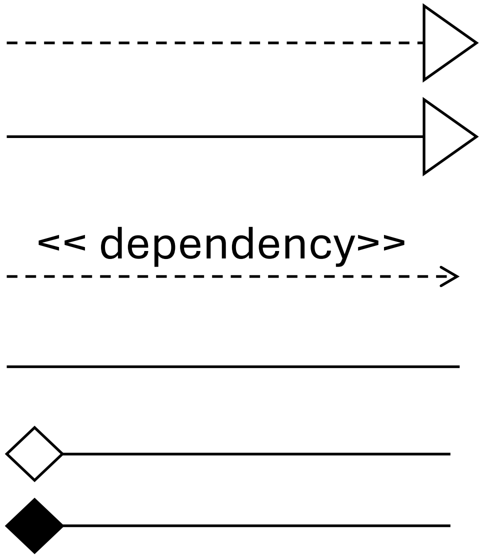
Association

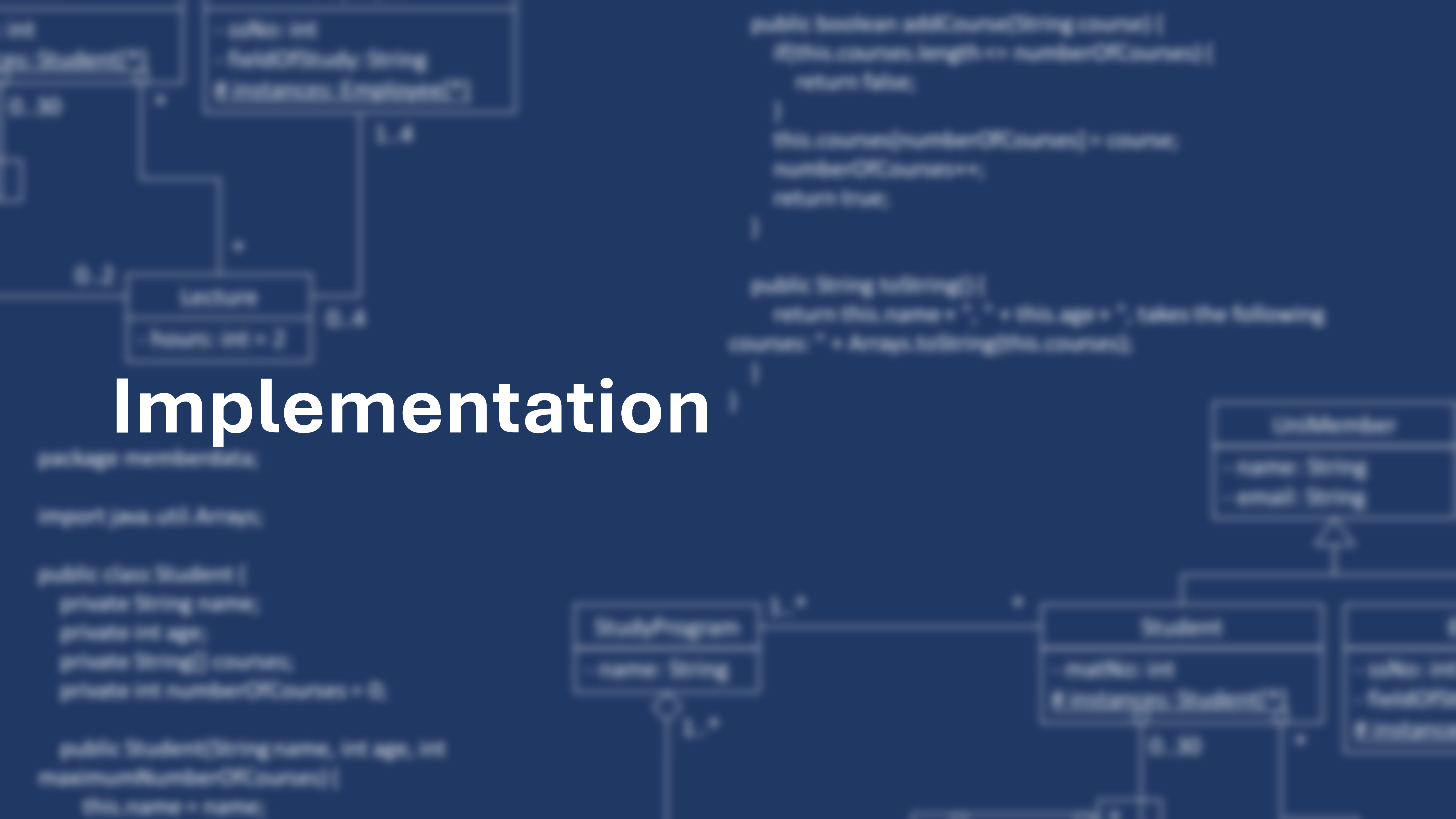
- Navigation
- Multiplicities
- Roles



Relation types

- Generalization
- Implementation
- Dependency
- Association
  - Aggregation
  - Composition

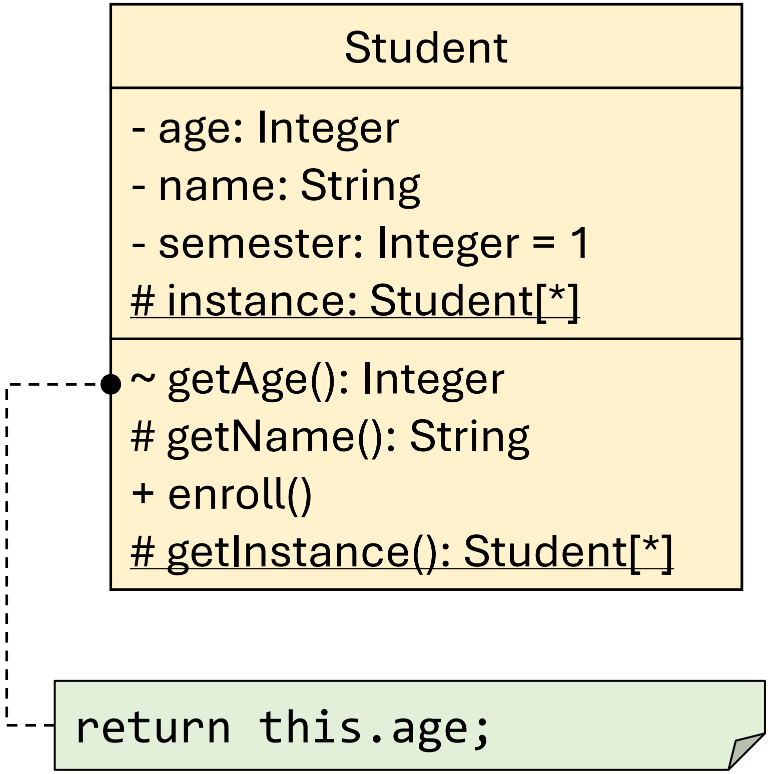




Implementation

Mapping Diagrams to Code

Class without associations



Java source code

```
class Student {
   private int age;
   private String name;
   private int semester = 1;
   protected static Collection<Student> instances;

   int getAge() {
      … return this.age;
   }
   protected String getName() {…}
   public void enroll() {…}
   protected static Collection<Student>
                                getInstances() {…}
}
```

Implementation of Associations

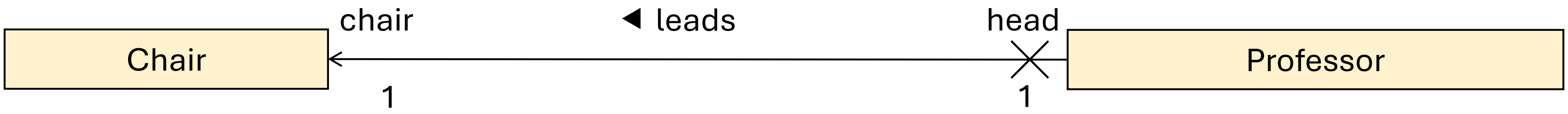
In the following slides, we will discuss the mapping of all combinations of navigability and multiplicity of associations

We will (most of the time) not make use of any particular programming language, but rather transform diagrams with association into diagrams without.

The mapping of a diagram without association has been discussed on the previous slide.

Unidirectional Association "to 1"

Object model with association



Chair

chair

◀ leads

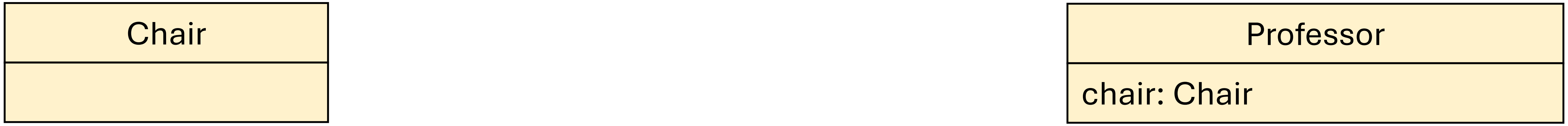
head

Professor

1

1

Object model without association



Chair

Professor

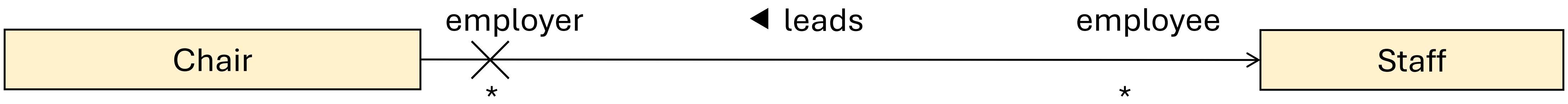
chair: Chair

Unidirectional 1:1 and 1:N associations are easy:

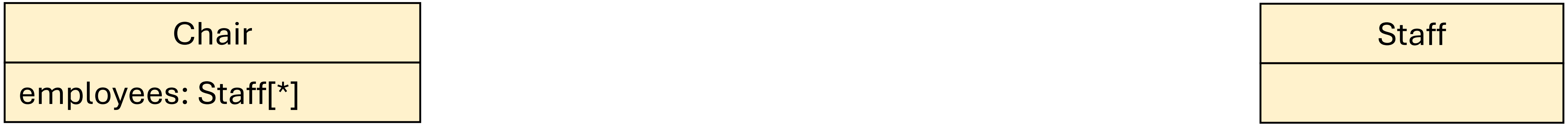
- The not navigable class has a reference to the navigable class
- The referenced class has no such variable

Unidirectional Associations 1:N and M:N

Object model with association



Object model without association

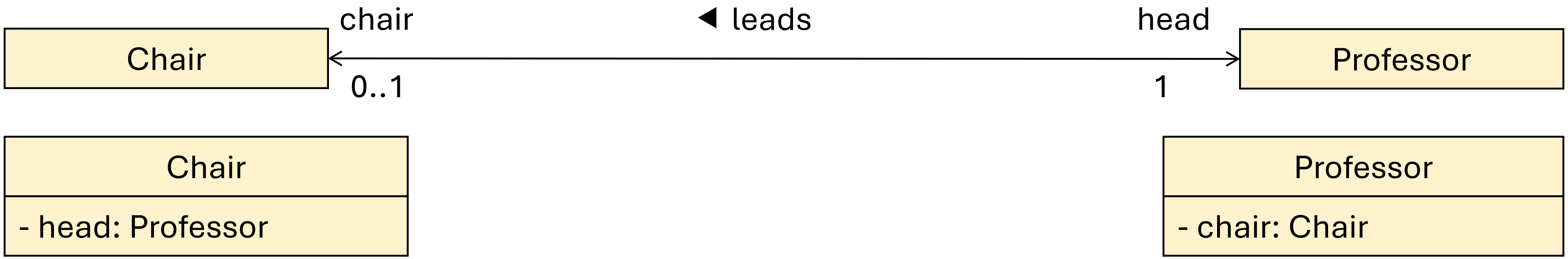


Unidirectional 1:N and M:N associations are equally easy:

- Only difference to previous: referencing class holds a collection of instances of the referenced class
- In Java, Collection is the interface to classes like List, Set, etc.

Bidirectional Association 1:1

In principle the same as two pointing opposite unidirectional associations.

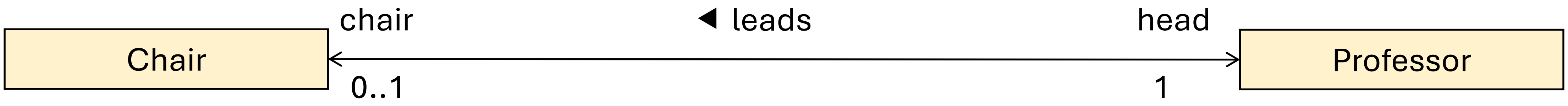


Problem: When setting the reference, we need to make sure the back reference is set as well

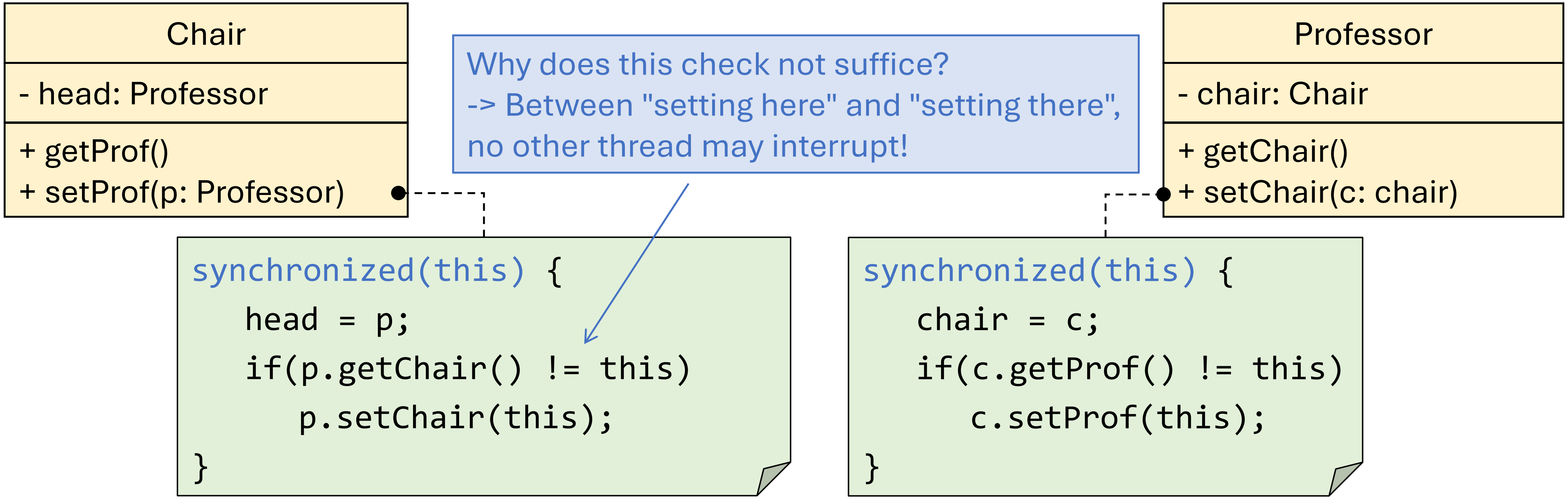
- Both assignments must happen atomically
- Synchronization in setter methods → next slide
  - Basis: synchronized-blocks and methods in Java

Bidirectional Association 1:1

Object model with association



Object model without association



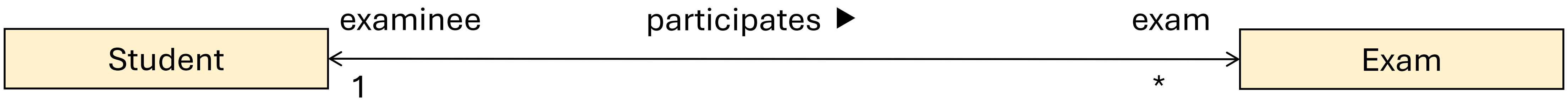
Why does this check not suffice?
-> Between "setting here" and "setting there", no other thread may interrupt!

```
synchronized(this) {
   head = p;
   if(p.getChair() != this)
      p.setChair(this);
}
```

```
synchronized(this) {
   chair = c;
   if(c.getProf() != this)
      c.setProf(this);
}
```

Bidirectional Association 1:N

Object model with association



Student

examinee

participates ▶

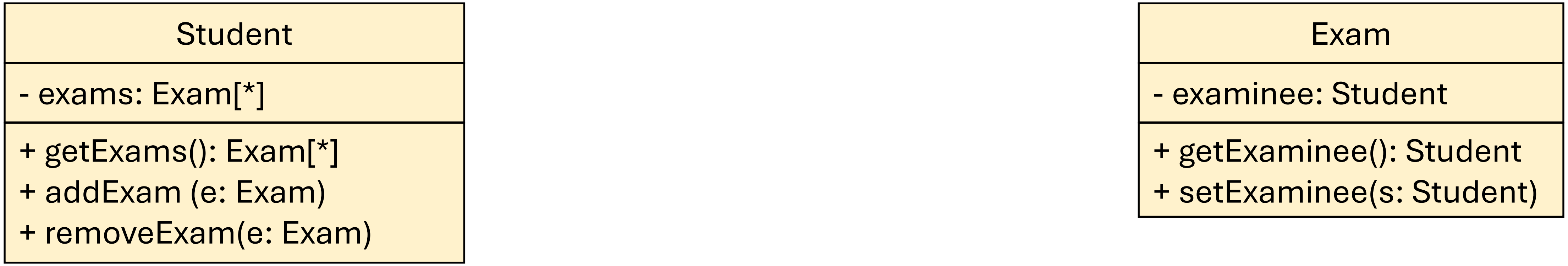
exam

Exam

1

*

Object model without association



Student

- exams: Exam[*]

+ getExams(): Exam[*]
+ addExam (e: Exam)
+ removeExam(e: Exam)

Exam

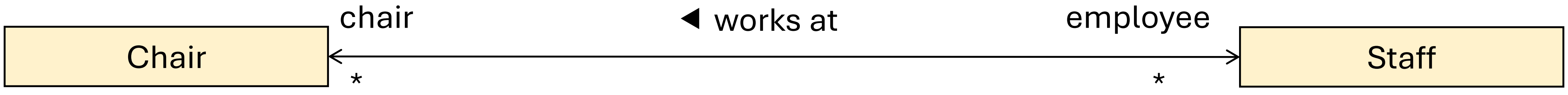
- examinee: Student

+ getExaminee(): Student
+ setExaminee(s: Student)

Synchronization approach like bidirectional 1:1 case.

Bidirectional Association N:M (naïve)

Object model with association



Chair

chair

*

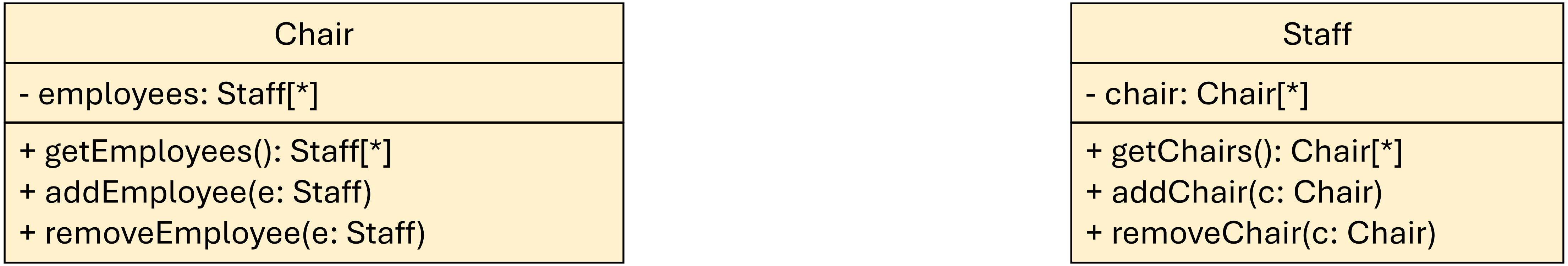
◀ works at

employee

*

Staff

Object model without association



Chair

- employees: Staff[*]

+ getEmployees(): Staff[*]
+ addEmployee(e: Staff)
+ removeEmployee(e: Staff)

Staff

- chair: Chair[*]

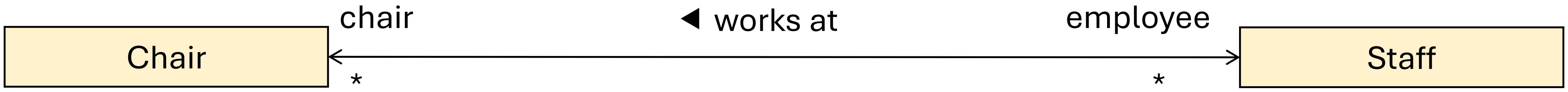
+ getChairs(): Chair[*]
+ addChair(c: Chair)
+ removeChair(c: Chair)

Problem 1: Setting back-references deadlock-free not trivial anymore

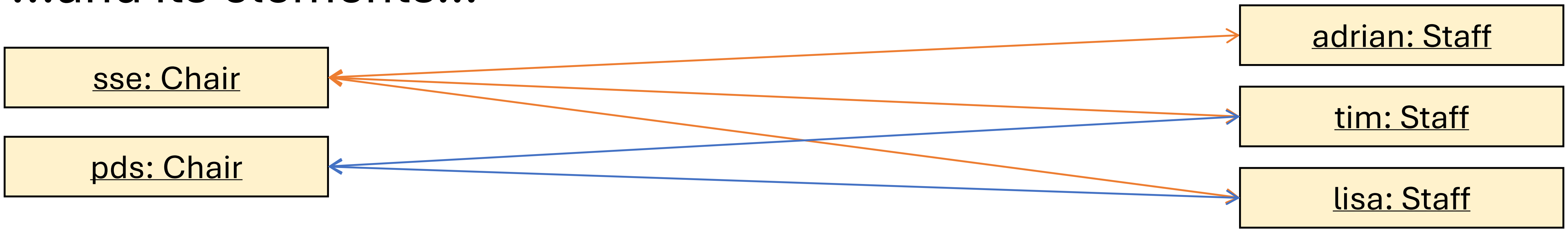
Problem 2 (of everything so far): The relation is hardcoded into the classes → high inter-dependency

Associations are implicit Classes!

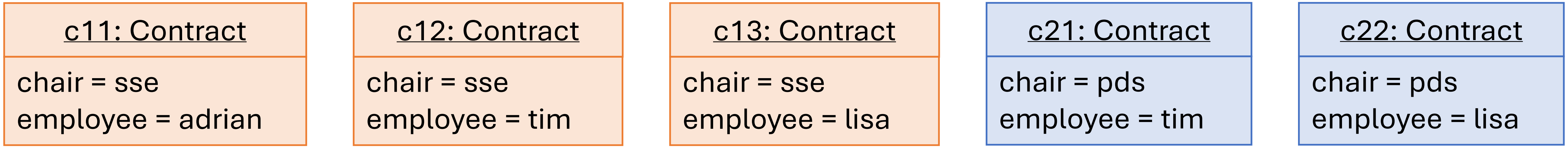
An association...



...and its elements...



...can be seen as instances of a class "Contract"



c11: Contract
chair = sse
employee = adrian

c12: Contract
chair = sse
employee = tim

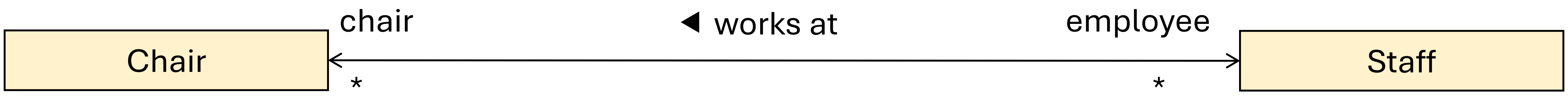
c13: Contract
chair = sse
employee = lisa

c21: Contract
chair = pds
employee = tim

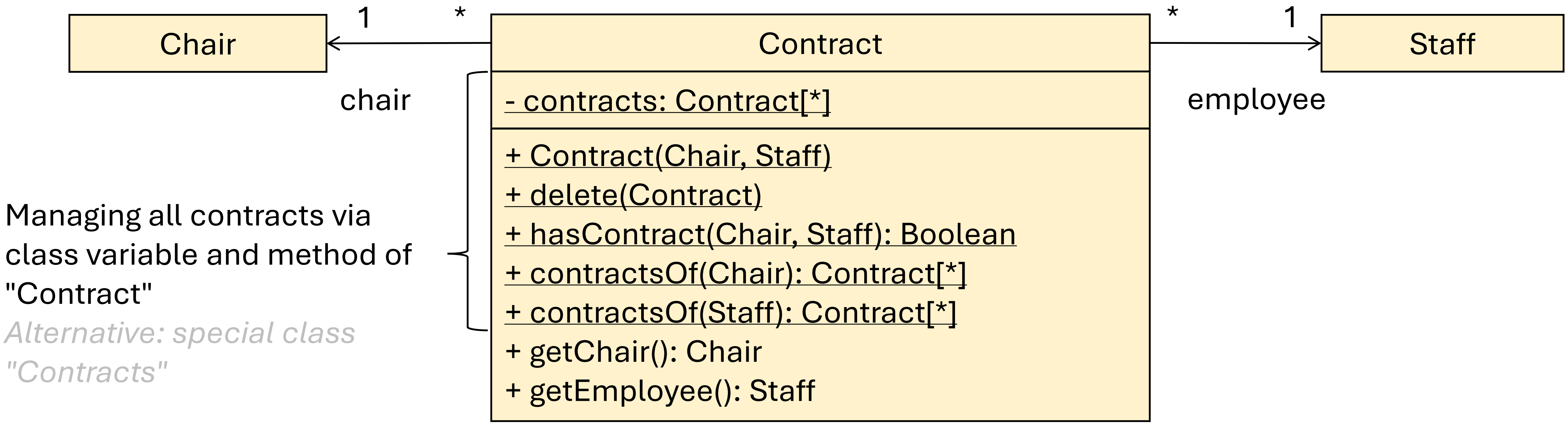
c22: Contract
chair = pds
employee = lisa

Bidirectional Associations N:M (Class)

Object model with bidirectional association



Object model without bidirectional association



- contracts: Contract[*]

+ Contract(Chair, Staff)
+ delete(Contract)
+ hasContract(Chair, Staff): Boolean
+ contractsOf(Chair): Contract[*]
+ contractsOf(Staff): Contract[*]
+ getChair(): Chair
+ getEmployee(): Staff

Managing all contracts via class variable and method of "Contract"

Alternative: special class "Contracts"

How to guarantee Multiplicities

- 0..* and 0..1: No special care needed (as seen before).
- =1: Set value at object construction; only replace; never set to `null`!
- Bounds X..Y: everywhere the attribute is modified, make sure the bounds are not violated.

```
private AcademicEmployee responsible; //=1
private List<Student> students; //X..Y

public CourseData(AcademicEmployee responsible,
      List<Student> students) {
   this.responsible = responsible;
   this.students = students;
}
...
```

```
...
public void addStudent(Student s) {
   if(this.students.size() + 1 <= Y) {
      this.students.add(s);
   }
}
public void removeStudent(Student s) {
   if(this.student.size() - 1 >= X) {
      this.students.remove(s);
   }
}
public void setResponsible(
      AcademicEmployee responsible) {
   if(responsible != null) {
      this.responsible = responsible;
   }
}
```

Implementation of Aggregation and Composition

In Java: No differentiation between association, aggregation and composition!

Proposed, but not in all terms valid approach: inner classes.

Following are some reasons for why not to use inner classes for aggregation and composition. These were not discussed in the lecture and will thus not be relevant for the exam.

"Composition via Inner Classes": Problem 1

Part cannot be instantiated without whole

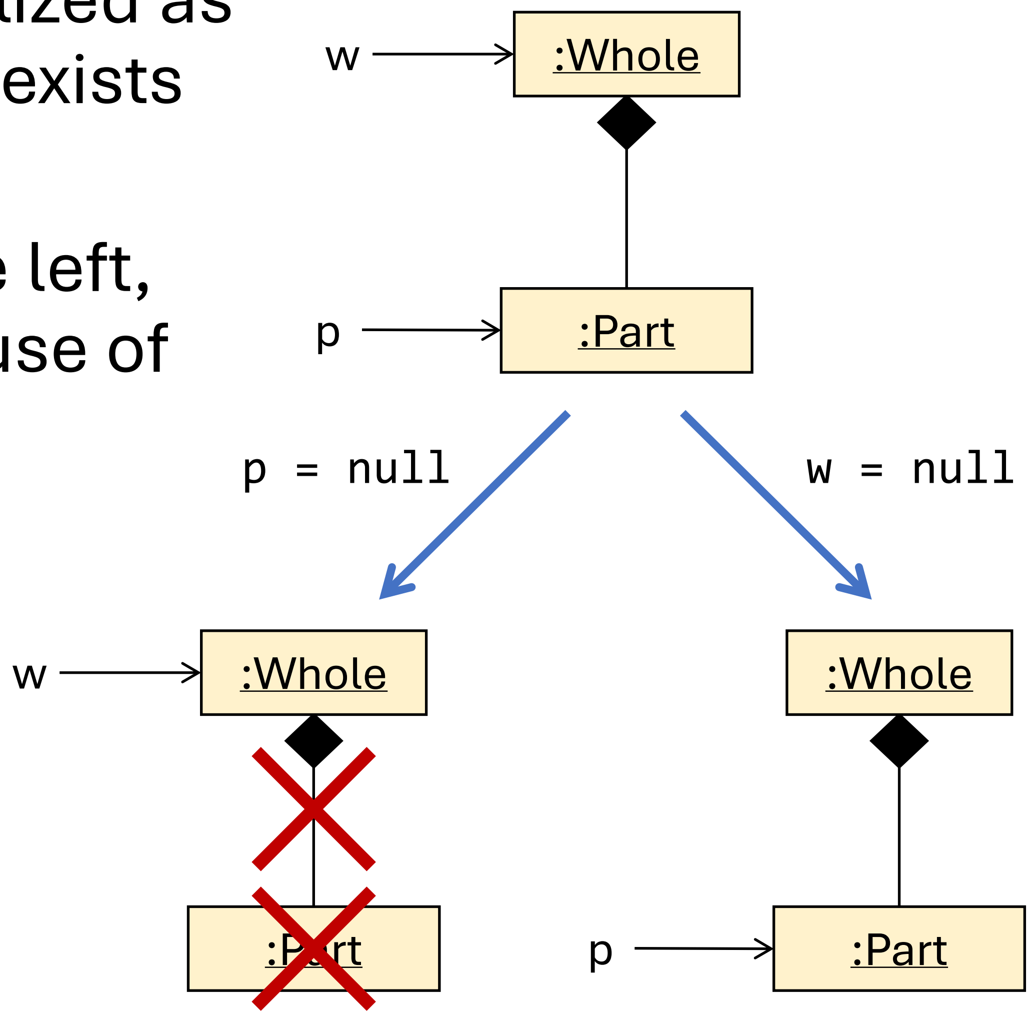
- There has to be an instance of the outer class on which the instance of the inner class can be created
- No possibility to realize 0..1 multiplicity

```
Whole w = new Whole();
Whole.Part p = w.new Part();
```

"Composition via Inner Classes": Problem 2

"Part cannot exist without whole" realized as "as long as the part exists, the whole exists as well"

- If there is no reference on the whole left, the whole continues to exists because of the reference from the part



"Composition via Inner Classes": Problem 3

No propagation of operations

- Deletion: See problem 2 and the fact that there is no explicit destruction in Java
- Handing the part to another whole: Since the reference to the whole is compiler-generated, we can not access it

```
class Whole {
   class Part {

   }
}
```

→

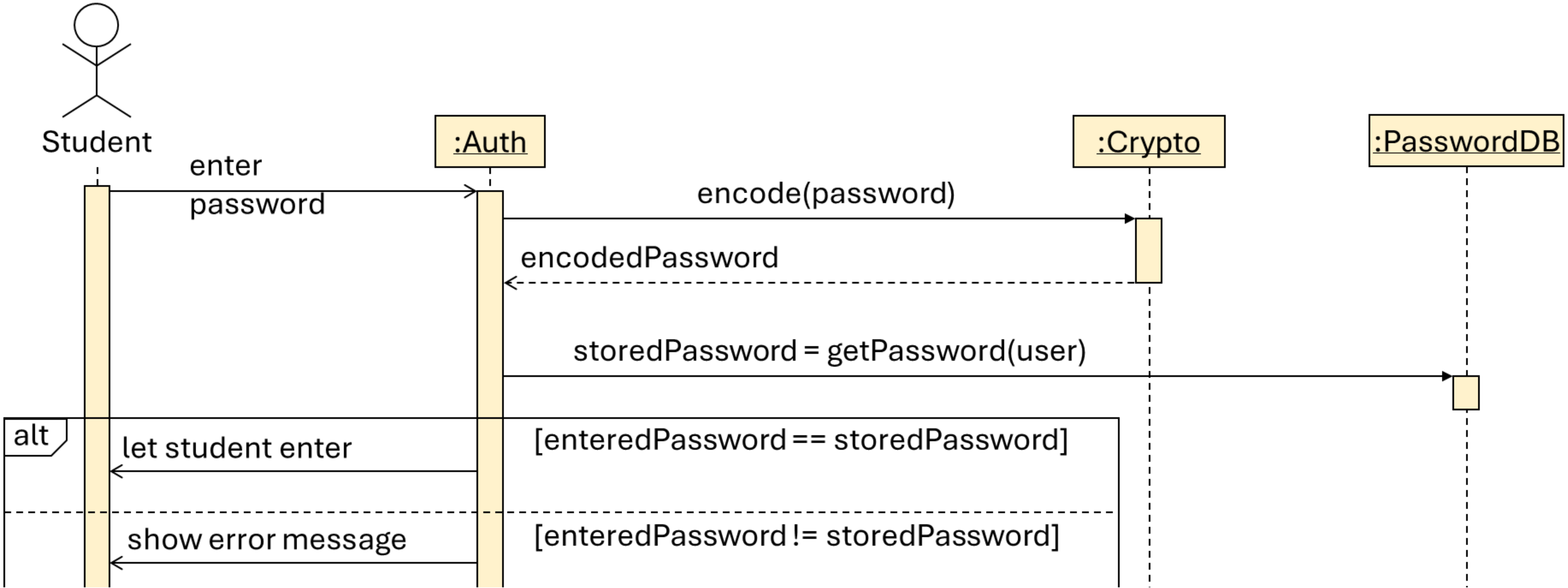
```
class Whole {...}
class Part {
   public Part(final Whole $this0) {
      parent = $this0;
   }
}
```

```
Whole w = new Whole();
Whole.Part p = w.new Part();
```

→

```
Whole w = new Whole();
Part p = new Part(w);
```

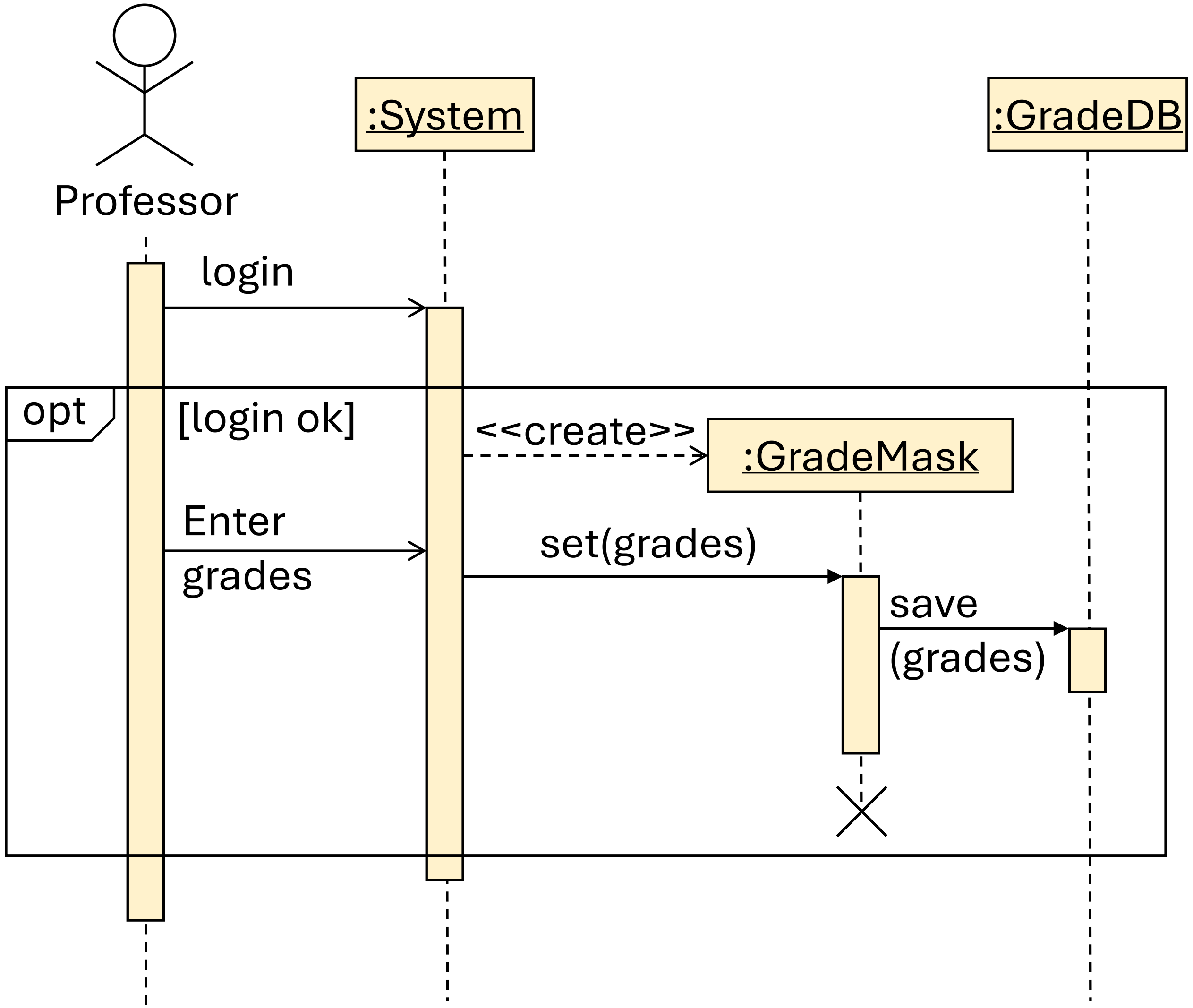
Questions?



Sequence Diagrams

Sequence Diagrams

- Visualizes messages along a timeline.
- Models interaction between
  - Actors and objects
  - Objects among each other
  - An object with itself
- Represents
  - Dataflow
  - Point in time and duration
  - Branches and loops
  - Parallelism
  - Filtering and asserts



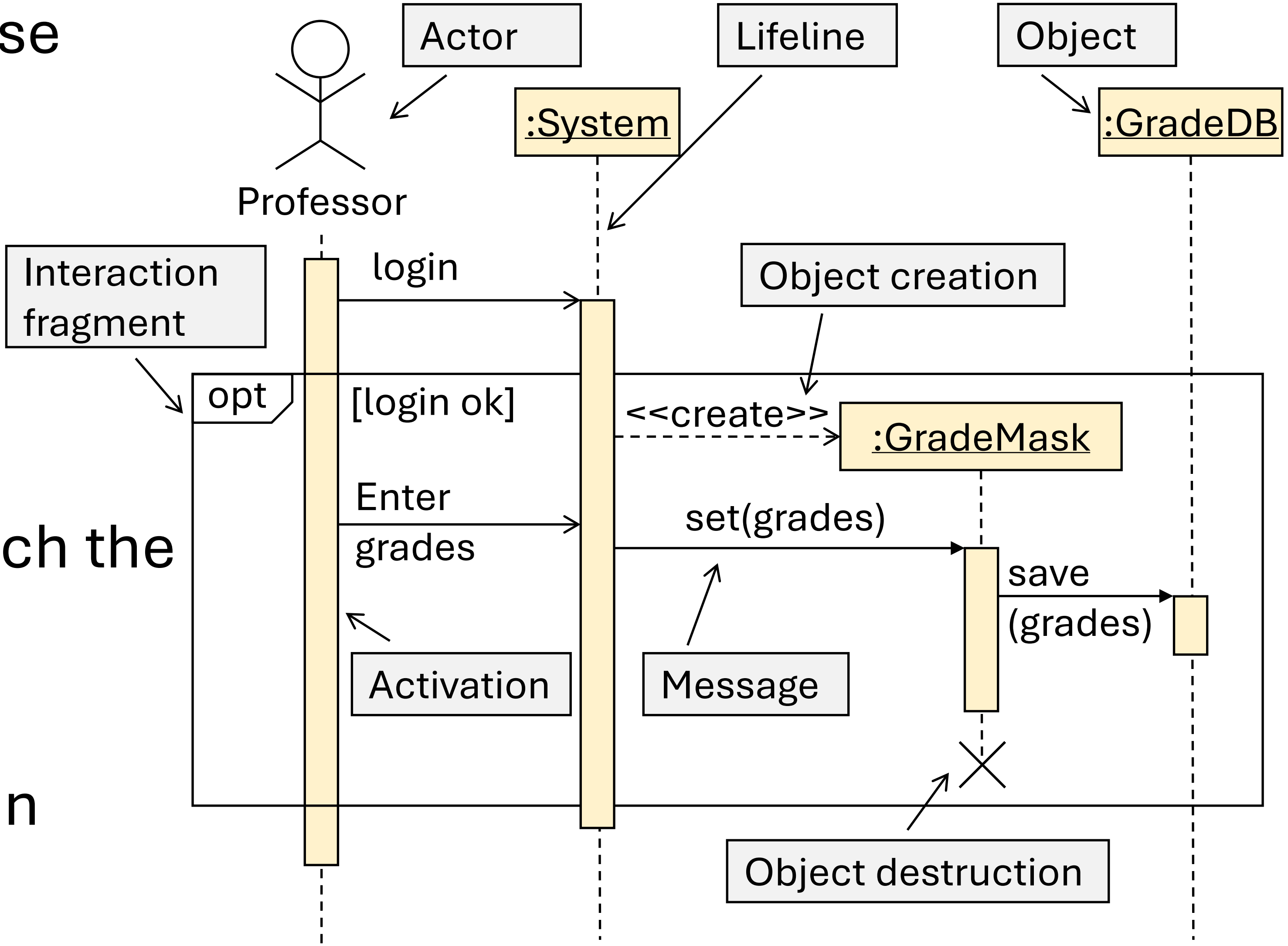
Elements

Actor: same as in use case diagrams.

Objects: same as in object diagrams.

Lifeline: Timespan in which the objects exist.

Message: Invocation of an activity.



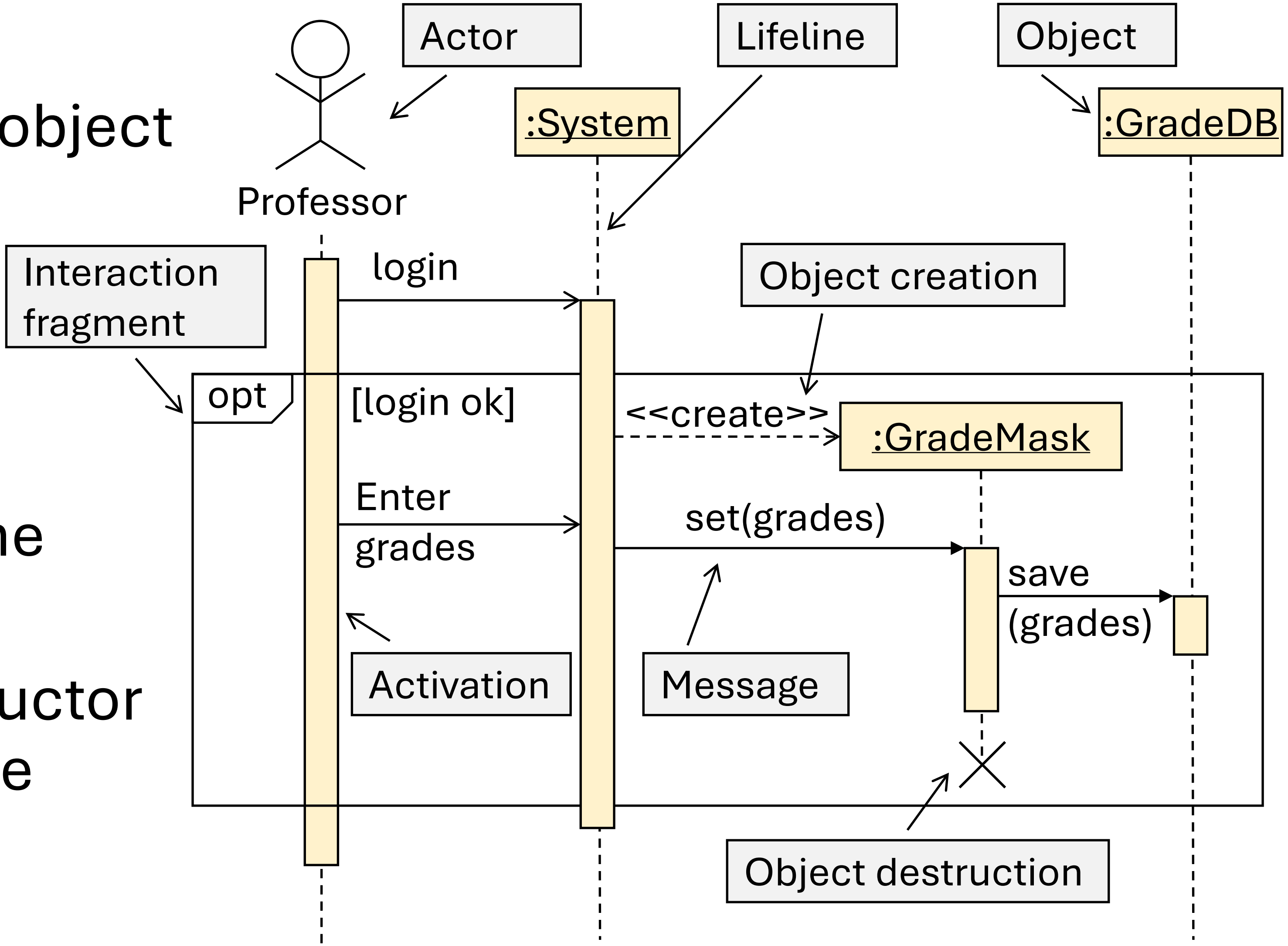
Elements

Activation

- Timespan in which the object is doing something.

Object creation and destruction:

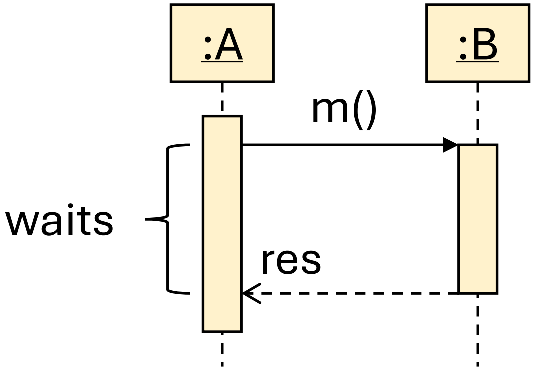
- Beginning and end of the lifeline.
- Corresponds to Constructor and Destructor/Garbage Collection



Synchronous vs. Asynchronous Messages

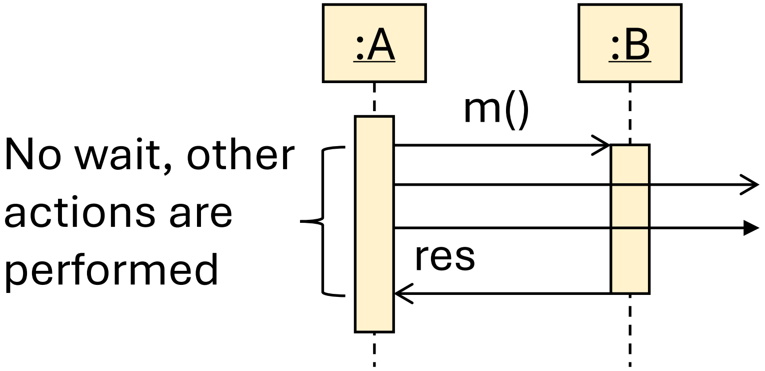
Synchronous messages:

The caller waits until it receives a response from the callee.



Asynchronous messages:

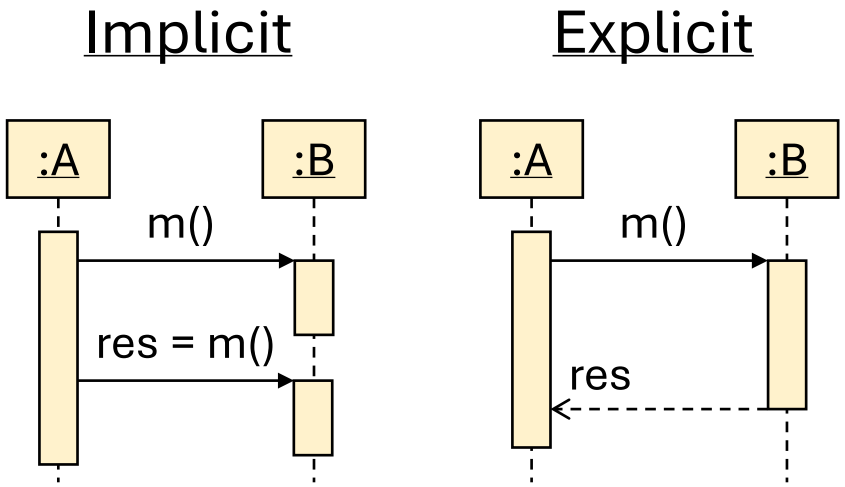
The caller continues with their next actions, not waiting on an answer.
Possible results are sent back via another asynchronous messages.



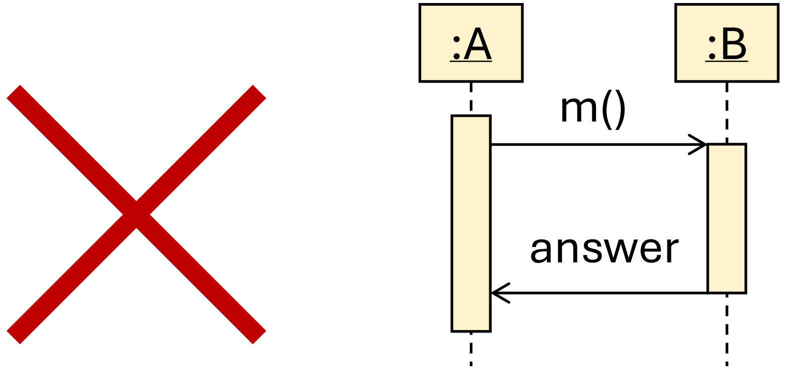
Data Flow

Synchronous messages require responses

- Either implicitly by the end of an activation
- Or explicitly via response message.



There are no response messages for asynchronous messages. Instead, we always must explicitly send back the answer (if there is any) via another message!

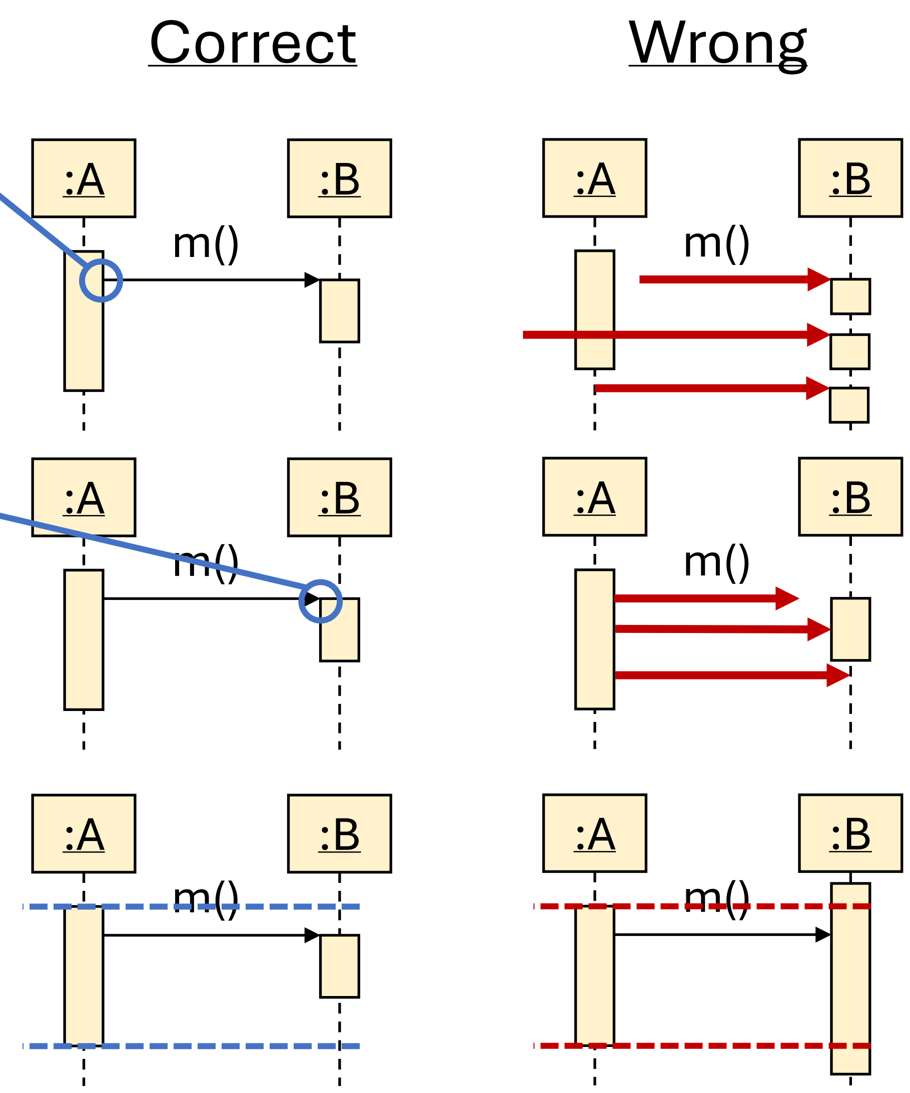


Syntax

An arrow always starts exactly on the edge of the activation that sends this message.

The activation that is targeted by the message start exactly at the arrowhead.

An activation lasts at least as long as all its nested activations which are call synchronously.



Combined Fragments

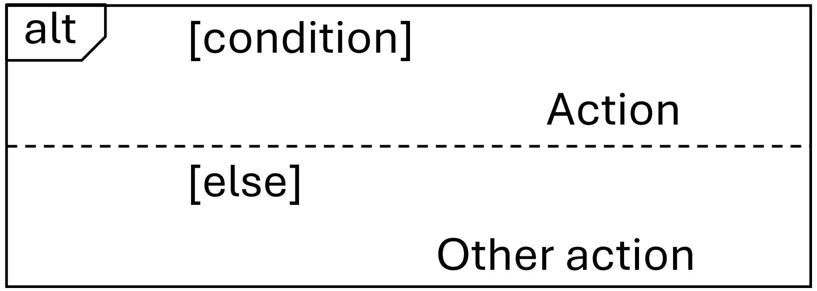
Complex control structures are realized via interaction fragments.

	Operator	Intention
Branches and loops	alt	Alternative interaction – if-then-else
	opt	Optional interaction – if-then
	break	Exceptional interaction – leaving the enclosing fragment
	loop	Iterative interaction
Concurrency and Ordering	seq	Sequential interaction of weak ordering (default)
	strict	Sequential ordering of strong order
	par	Concurrent interaction
	critical	Atomic interaction
Filtering and asserts	ignore	Irrelevant interaction
	consider	Relevant interaction
	assert	Asserted interaction
	neg	Invalid interaction

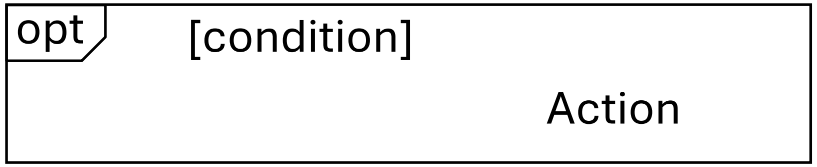
Combined Fragment

Alternative

- If condition is true, execute upper block,
- Otherwise execute lower block.
- May comprise multiple blocks (if-else if-else if...-else)



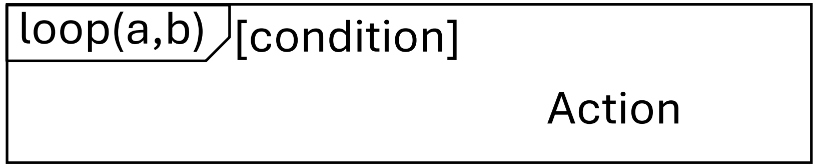
Option: Execute block if condition is true.



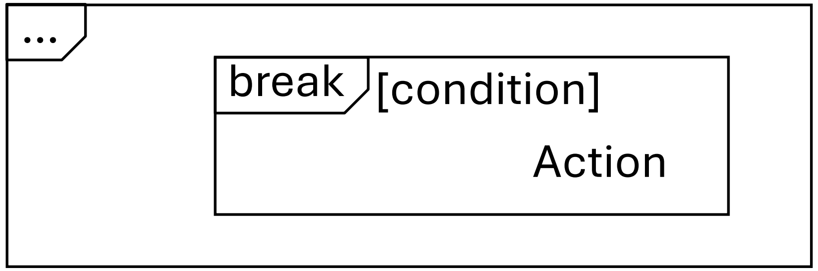
Combined Fragment

Loop: Repeat block min. a and max. b times.

Terminates if `condition` evaluates to false and at least `a` runs took place.



Break: If condition is true, run action in block and then leave the enclosing block.

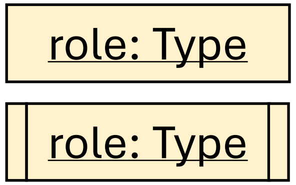


Questions?

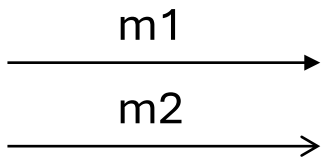
Summary

Elements

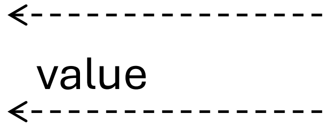
- Interaction partners
  - Passive object
  - Active object (thread)



- Message
  - synchronous (Caller wait for callee to finish)
  - asynchronous (Caller continues after sending of message)

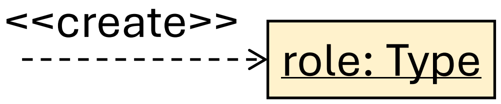


- Response message
  - No results
  - With result

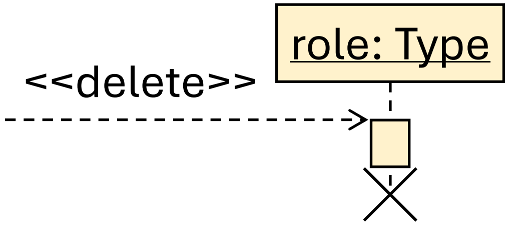


Elements

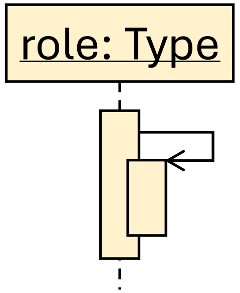
- Object creation message



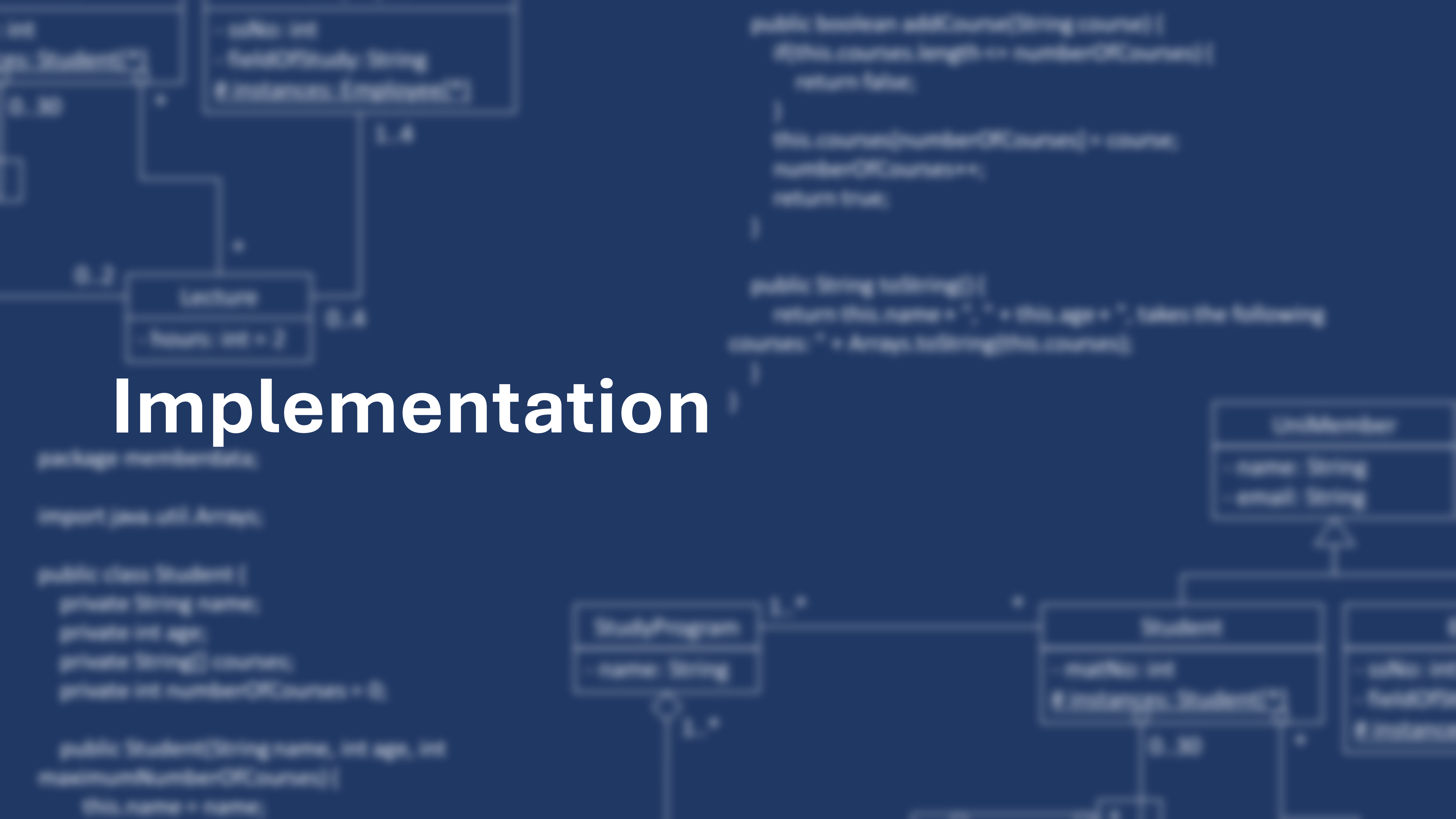
- Object destruction message



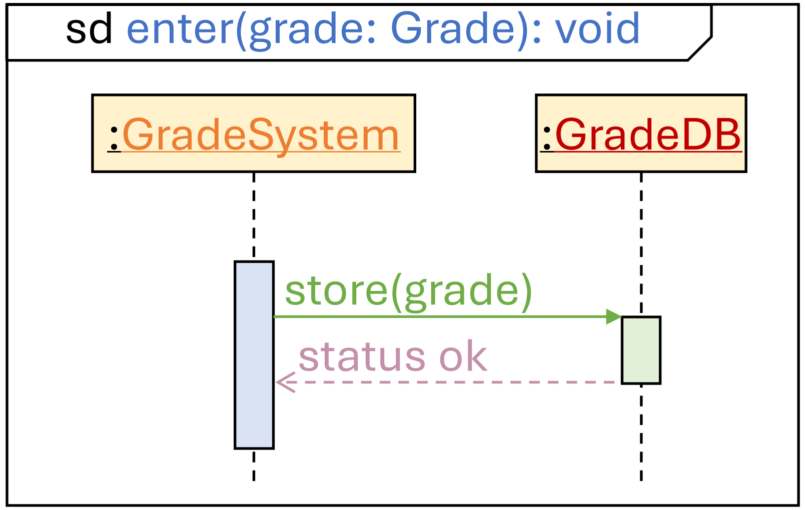
- Self-messaging
    - Recursion or
    - Message to object's other subroutines



Implementation



Sequence Diagram ▶ Code

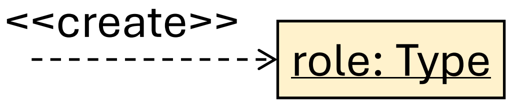


```
class GradeSystem {
   GradeDB gradedb //= ...;
   public void enter(Grade grade) {
      gradedb.store(grade);
   }
}
class GradeDB {
   public void store(Grade grade) {
      ...
      return Status.OK;
   }
}
```

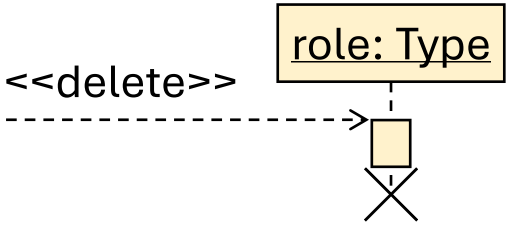
Note:

Interaction partners are either passed as arguments or are attributes of the object.

Sequence Diagram ▶ Code



```
...
Type role = new Type();
...
```



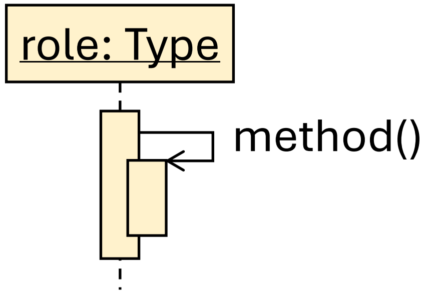
In C++ and JavaScript*!

```
...
delete role;
...
```

In Java

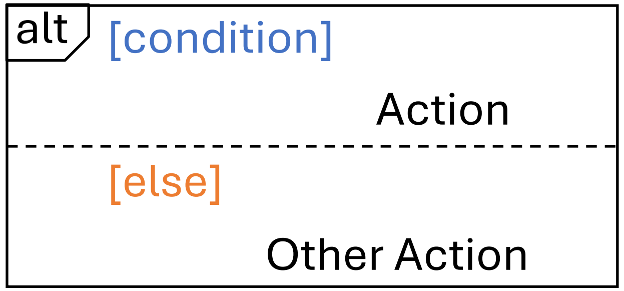
```
...
role = null; //or
other object
...
```

Also, there must be no other reference to the object. Otherwise, GC will not delete it.



```
...
this.method();
...
```

Sequence Diagram ▶ Code



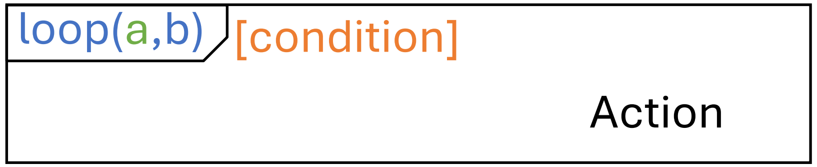
alt [condition]

Action

[else]

Other Action

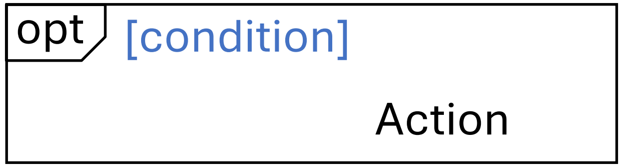
```
if(condition) {
    //Action
}
else {
    //Other Action
}
```



loop(a,b) [condition]

Action

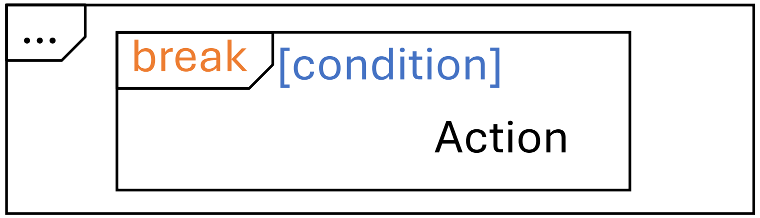
```
for(int i = 0; i < b; i++) {
    if(i >= a && !condition) {
        break;
    }
    //Action
}
```



opt [condition]

Action

```
if(condition) {
    //Action
}
```



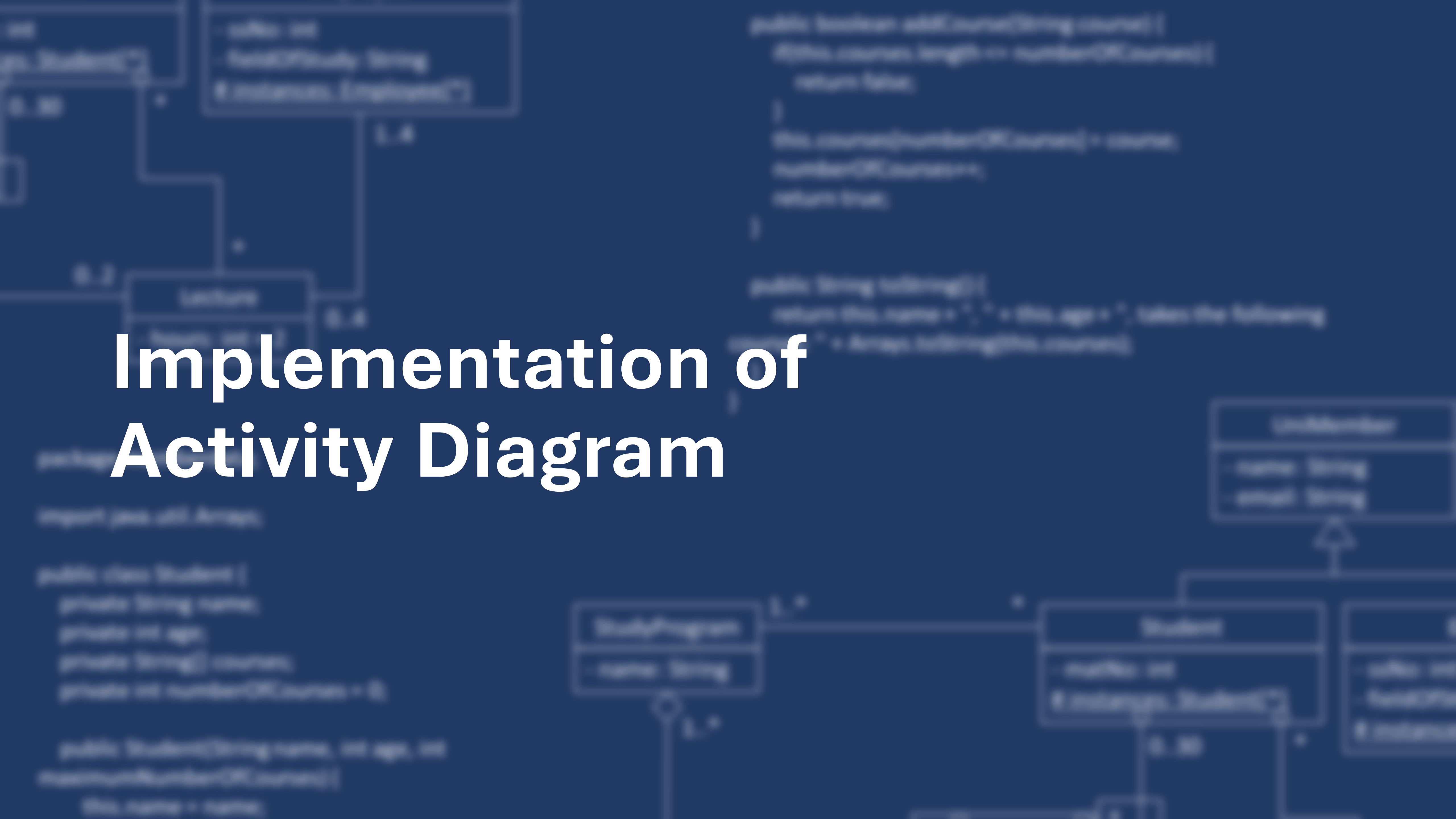
...

break [condition]

Action

```
... {
    if(condition) {
        //Action
        break; //or return;
    }
}
```

Implementation of Activity Diagram

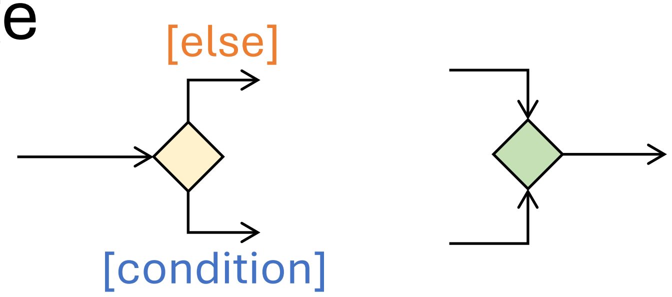


Activity Diagram ▶ Code

Activity diagrams are not necessarily coding diagrams. However, we can use them as replacement for control flow diagrams. In this case, each action corresponds to one statement.

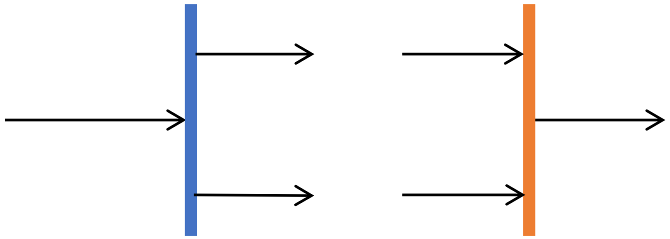
Two elements that do manifest in code:

- Decision – Merge



```
if(condition) {
   //Action
} else {
   //Other Action
}
//Here is the merge point
```

- Fork – Join



```
thread.start();
//...
thread.join();
```

Covered UML Diagrams in terms of Implementation

- Class Diagrams
- Object Diagrams
- Sequence Diagrams
- Activity Diagrams

This chapter.

- Use Case Diagrams → Too complex for specific rules.

- Component Diagrams
- State Machine Diagrams

In design patterns.

Incremental Development of a Domain Object Model

A domain object model is usually described with class diagrams.

Domain Object Model: Abbott's Textual Analysis

Mapping of language components to DOM elements:

Language element	DOM element	Example
Proper noun / name	Object	Jim Smith
Common noun	Class	Customer
"is a", "is a special kind of",...	Generalization / Role	A professor is also a researcher
"has", "contains", "comprises", ...	Aggregation / Composition	A module comprises a lecture and an exercise
Modal verb ("must", "can", "should")	Constraint	The number of participants must not exceed 200 students
Transitive Verb	Method	enter
Intransitive Verb	Event	appear
Adjective	Attribute	mandatory

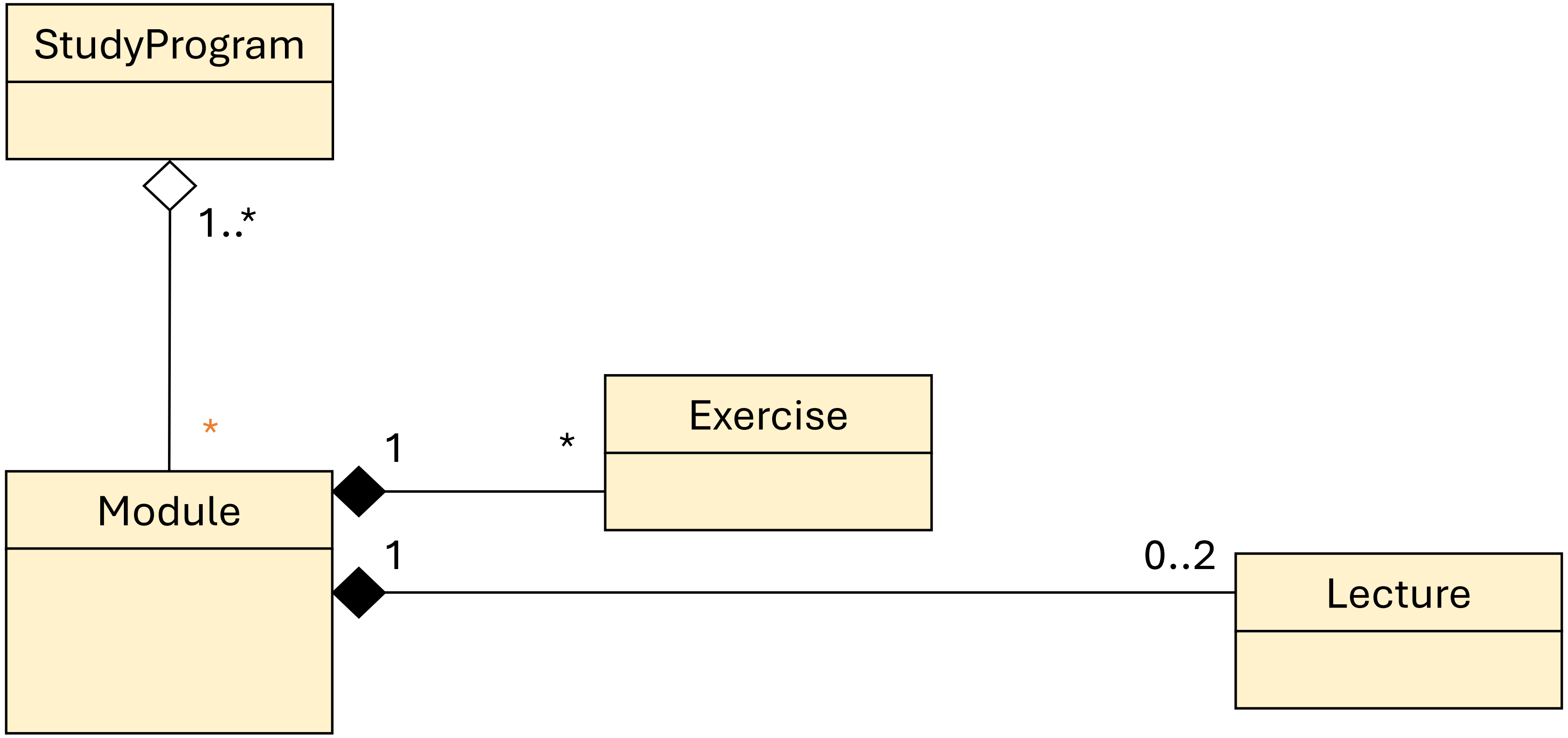
Attention! The output of Abbott's method is nothing close to final!

The Domain

- A module comprises up to 2 lecture slots and arbitrary many exercise. A module is part of at least one study program. Lectures and exercises have a certain length, usually 2 hours. The module has a name, a number of credit points given upon completion and a regular term in which the module can be taken.
- Lectures are conducted by at least one and up to four employees. Employees don't need to give any lecture, but they are allowed to give up to four.
- Lectures are not limited in size, but exercises are limited to 30 students.
- An exercise is conducted by a tutor who is a student. Each tutor can hold up to four exercises.
- Students are enrolled in any number of study programs.
- We want to keep track of the data of all university members, that is, their name and email address. Student data additionally has their matriculation number, employee data has the social security number additionally. For both categories of university members, we keep separate lists of all entries.

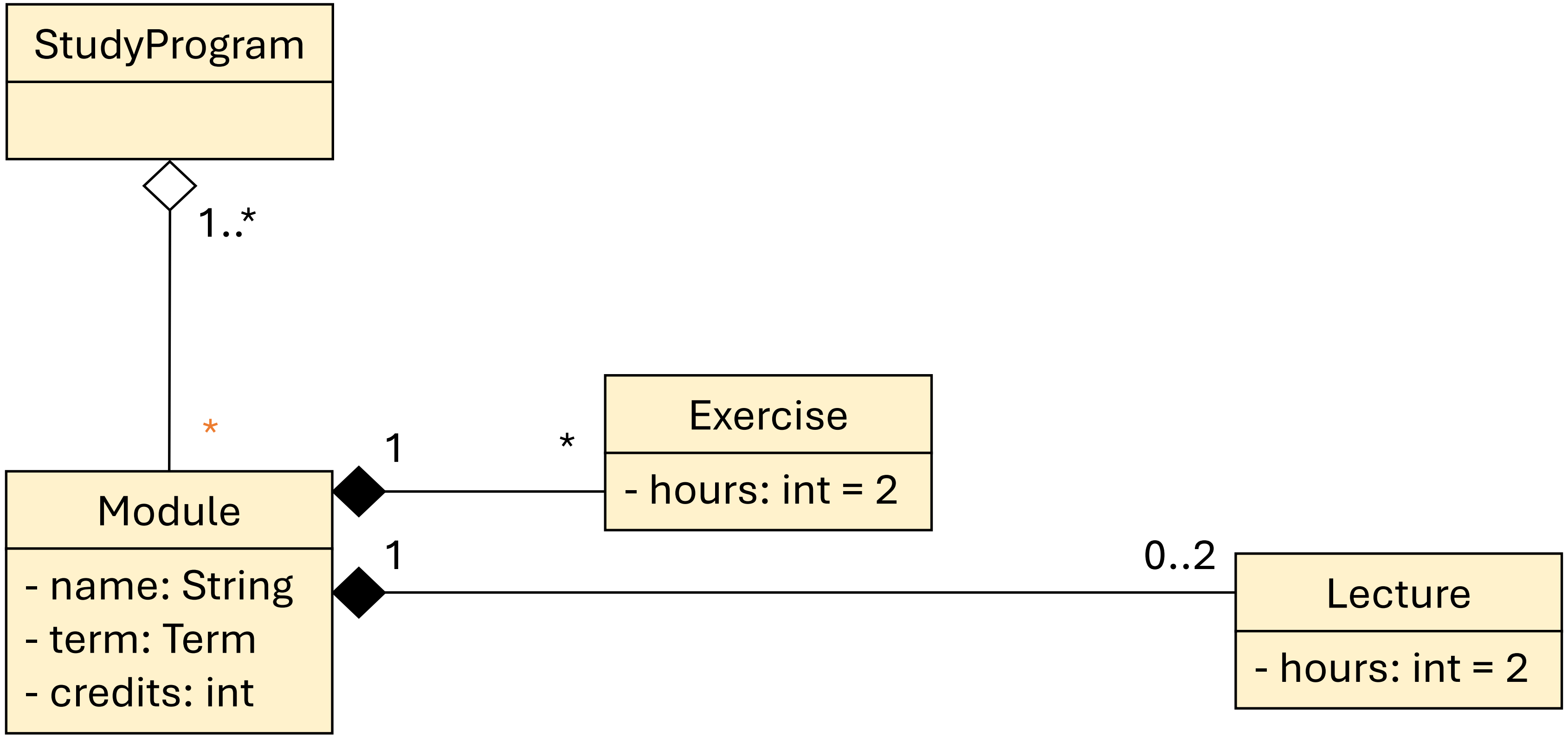
Class Diagram: University

A module comprises up to 2 lecture slots and arbitrary many exercise. A module is part of at least one study program.



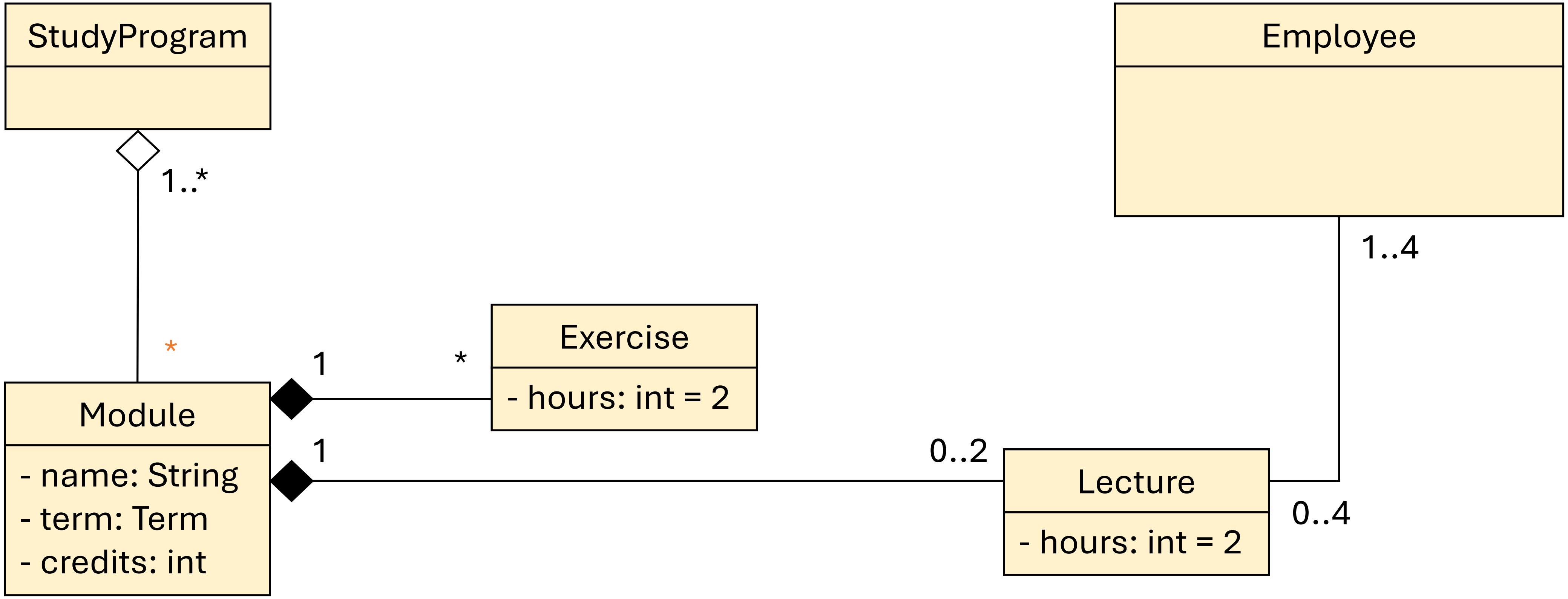
Class Diagram: University

The module has a name, a number of credit points given upon completion and a regular term in which the module can be taken. Lectures and exercises have a certain length, usually 2 hours.



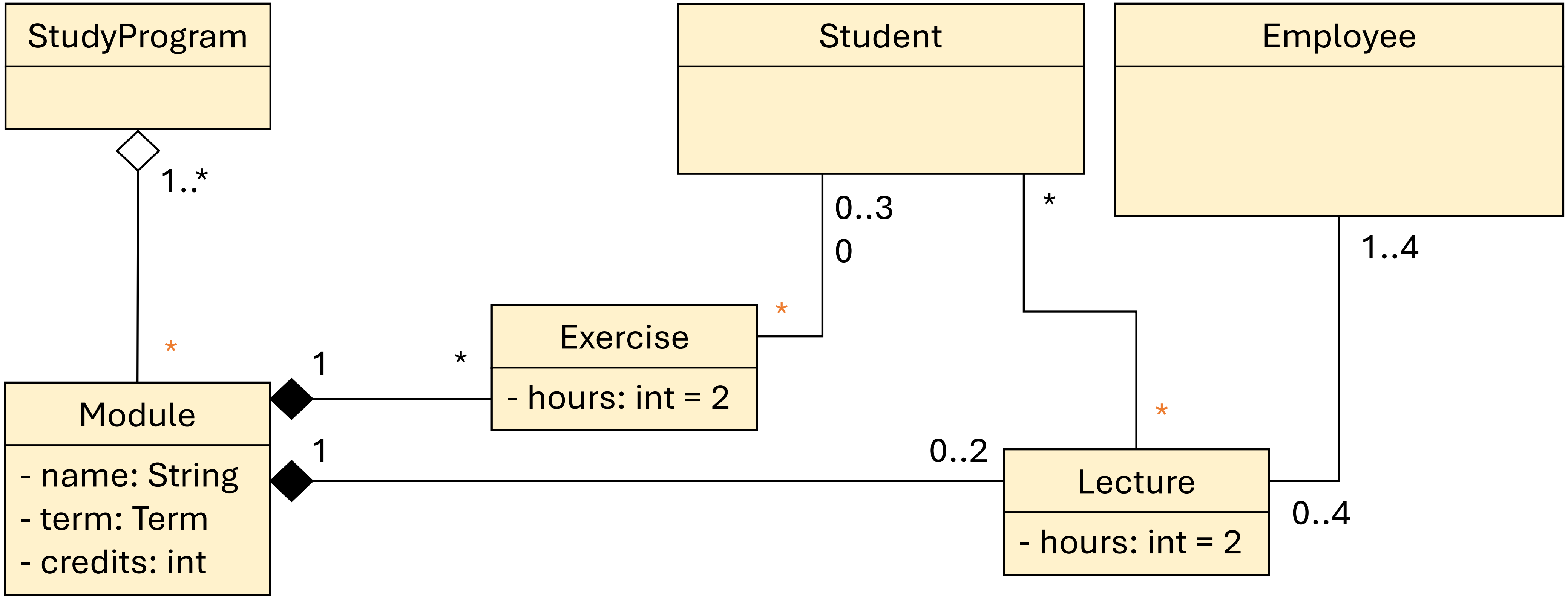
Class Diagram: University

Lectures are conducted by at least one and up to four employees. Employees don't need to give any lecture, but they are allowed to give up to four.



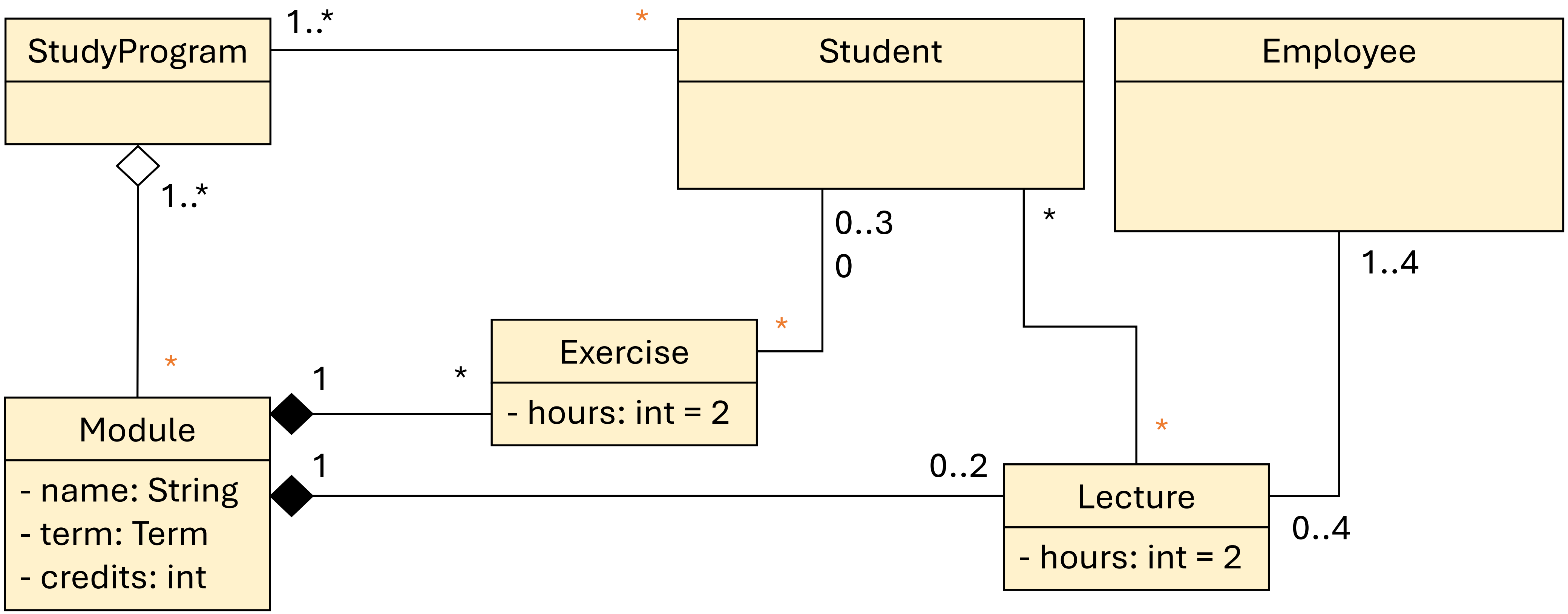
Class Diagram: University

Lectures are not limited in size, but exercises are limited to 30 students.



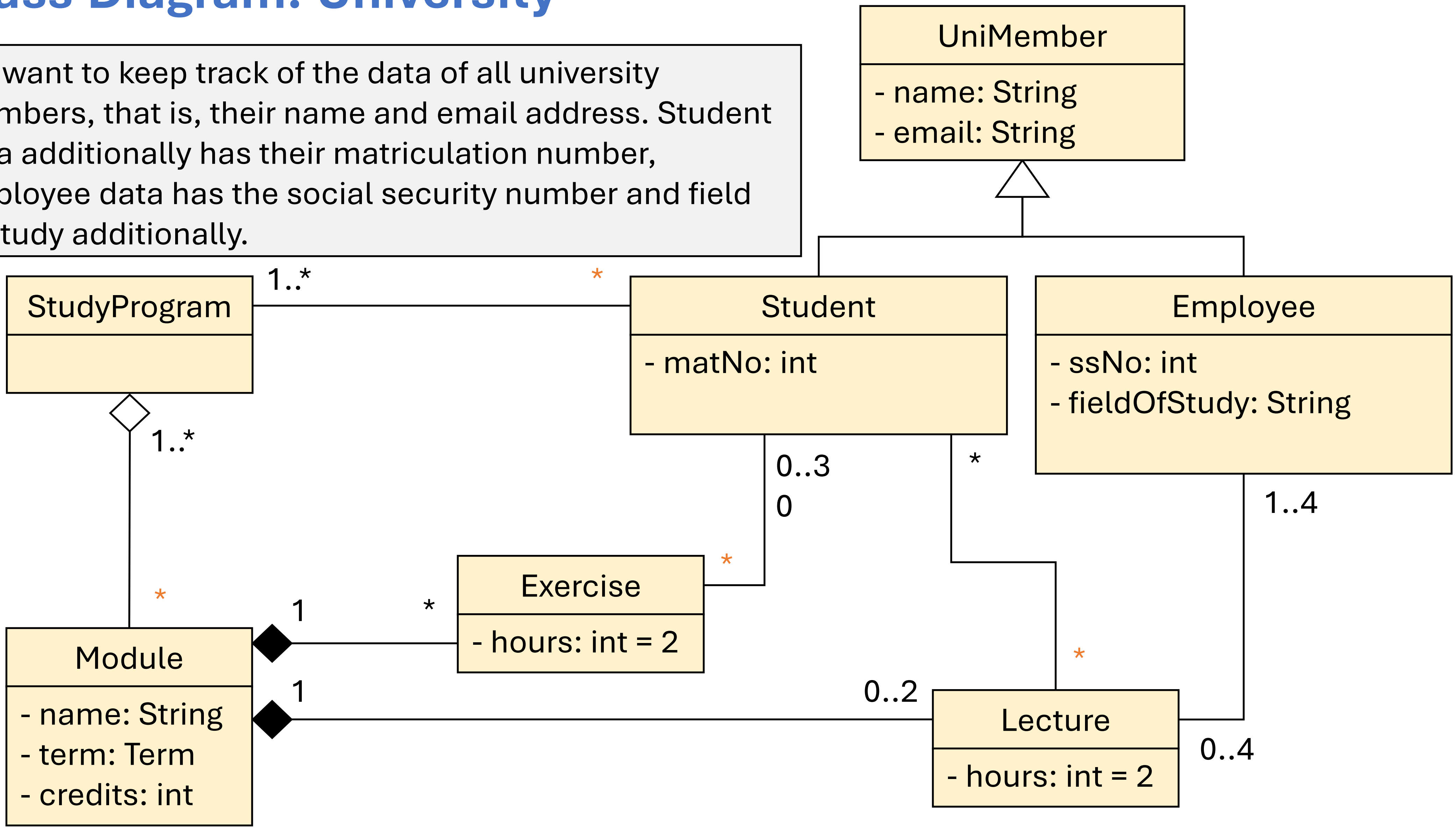
Class Diagram: University

Students are enrolled in any number of study programs.

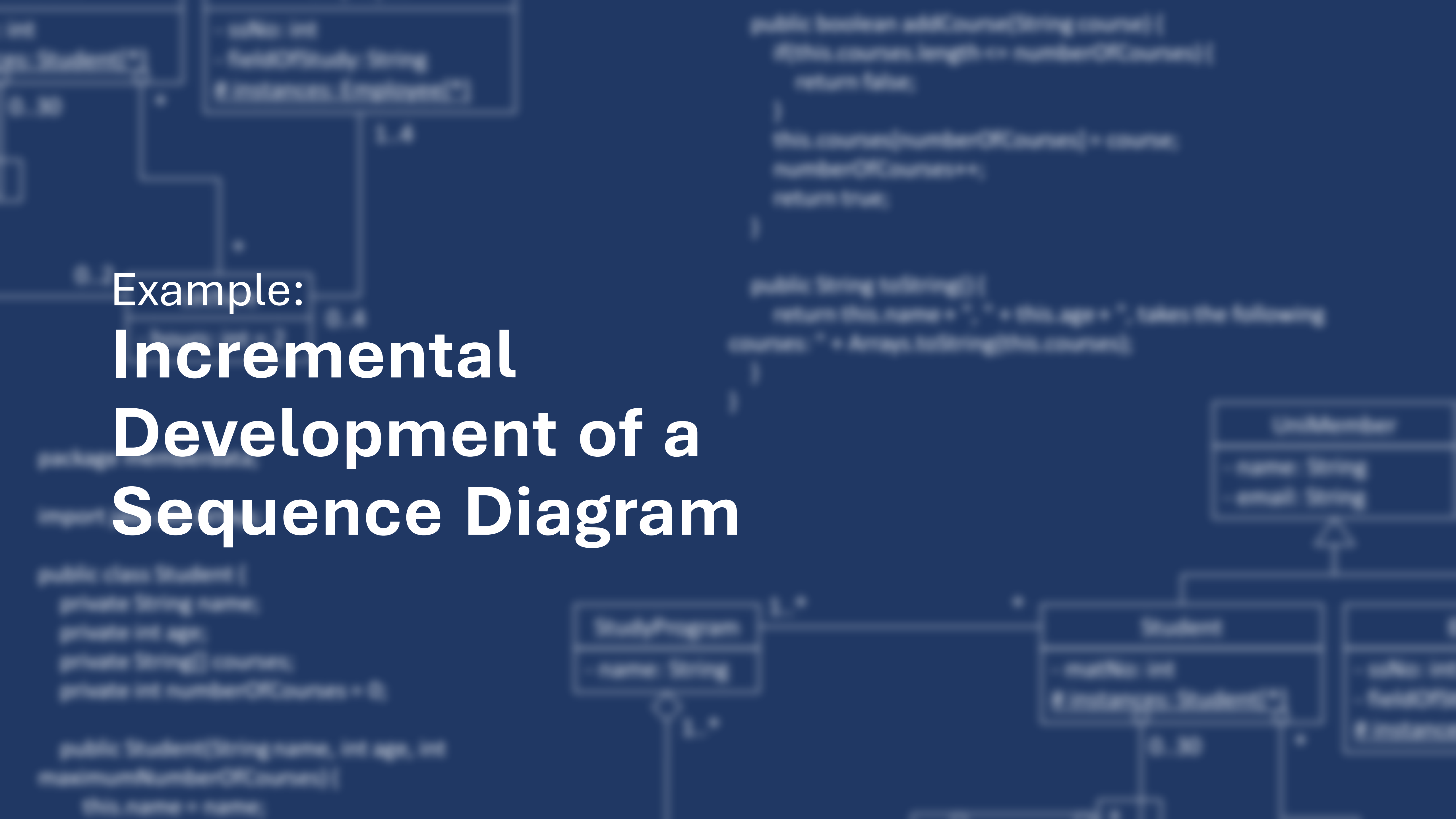


Class Diagram: University

We want to keep track of the data of all university members, that is, their name and email address. Student data additionally has their matriculation number, employee data has the social security number and field of study additionally.



Questions?

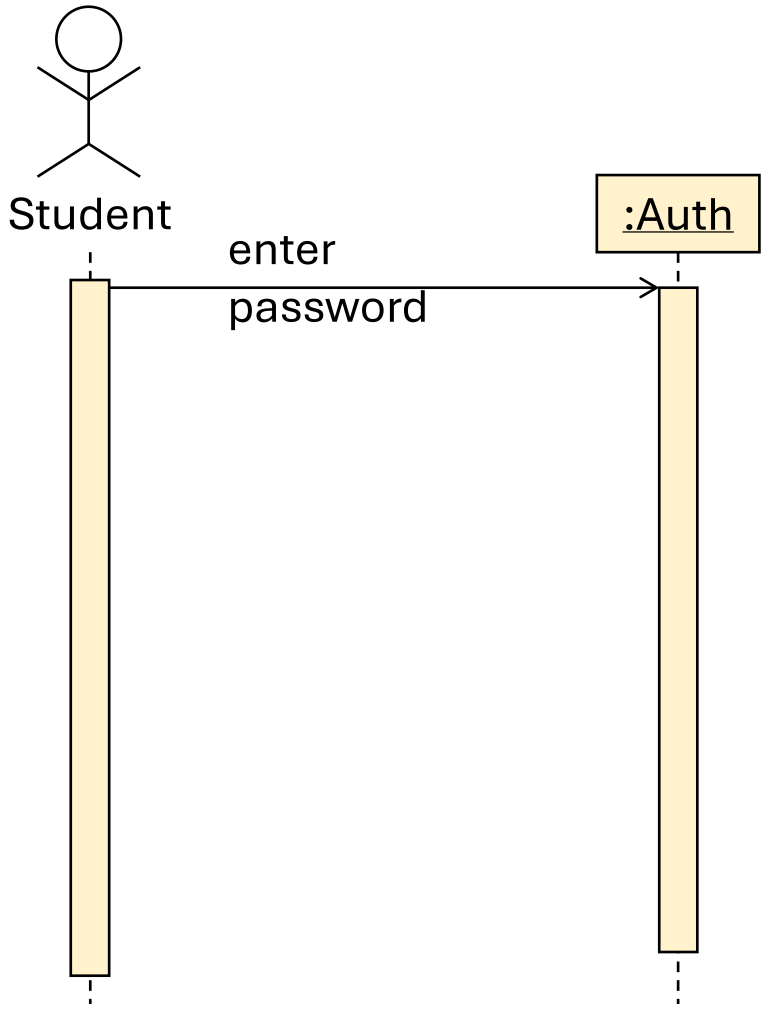


Example:

Incremental Development of a Sequence Diagram

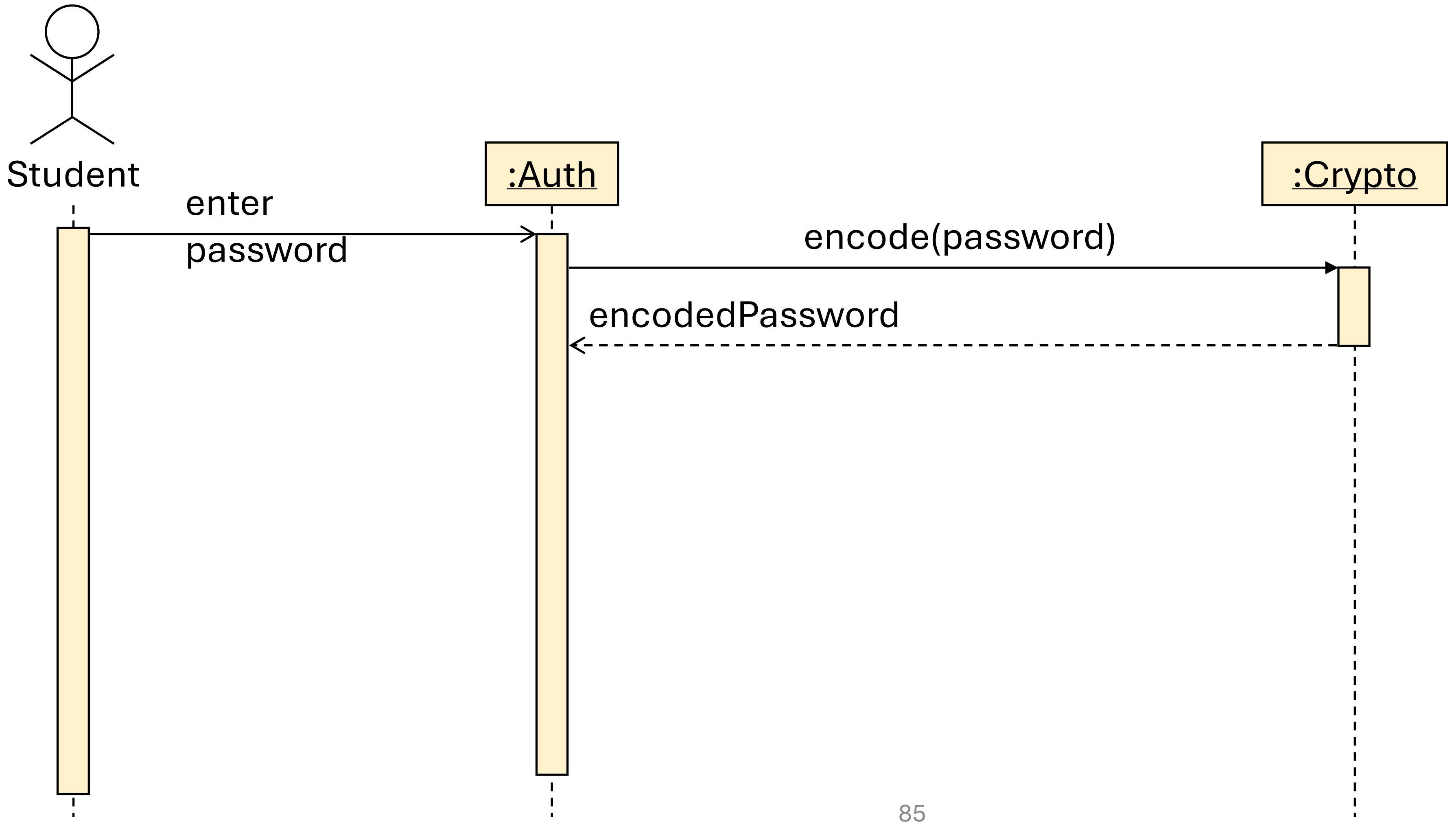
Sequence Diagram: Authentication

1. The student enters the password.



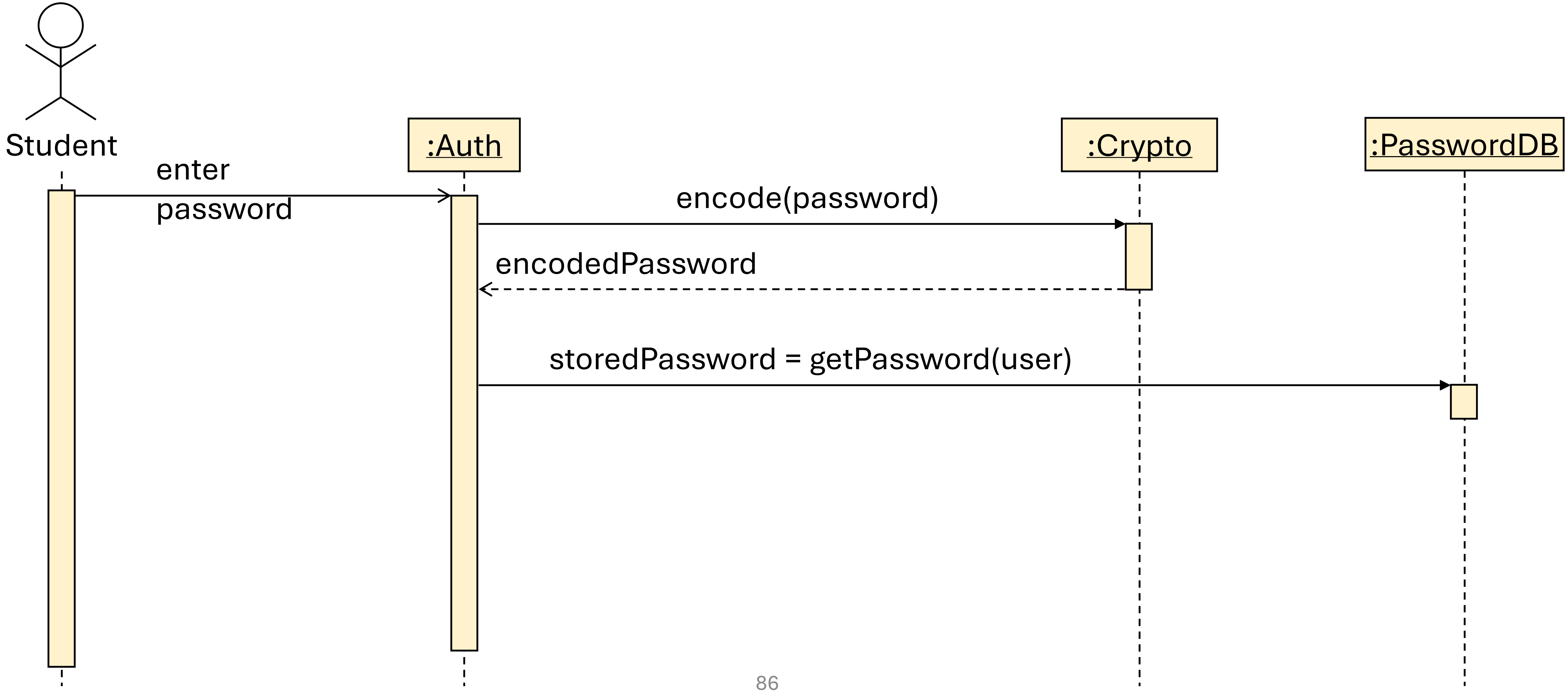
Sequence Diagram: Authentication

1. The student enters the password.
2. The password is encrypted.



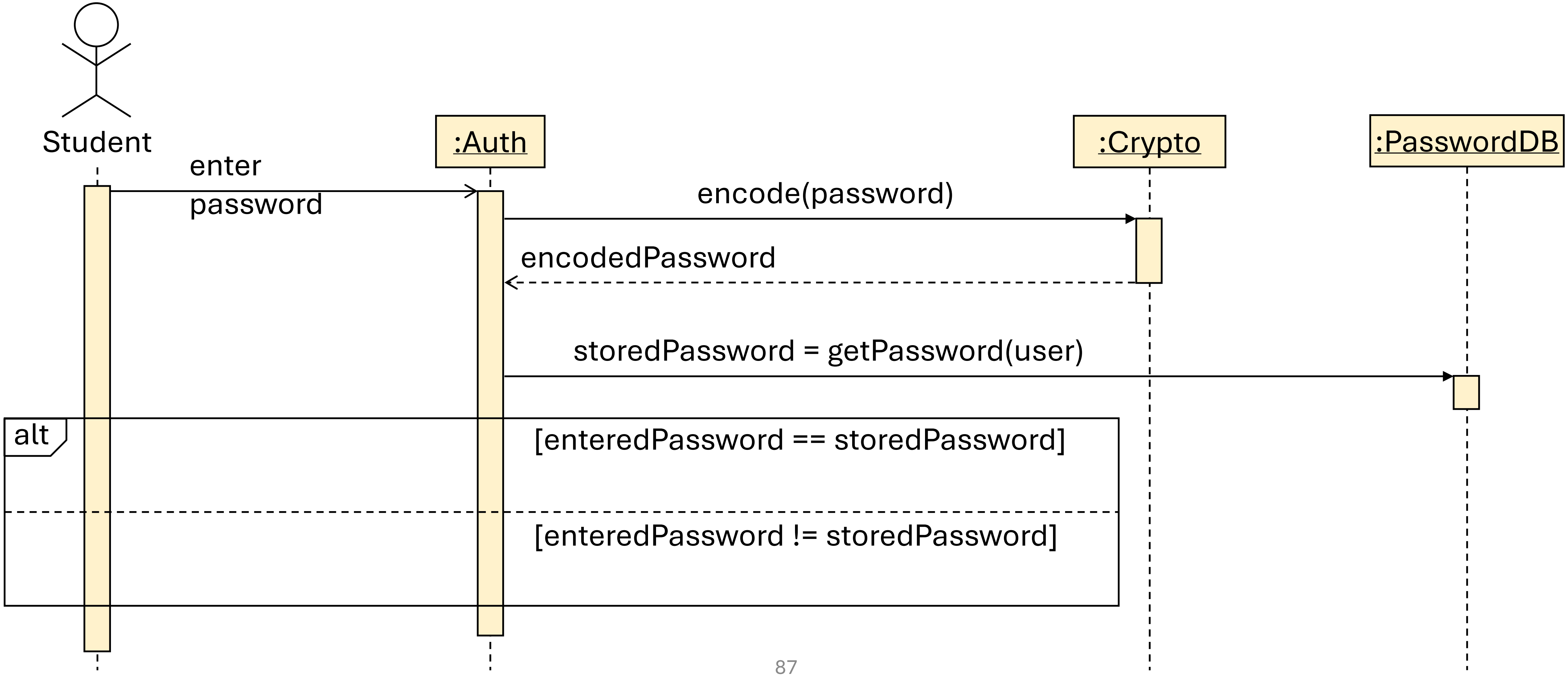
Sequence Diagram: Authentication

1. The student enters the password.
2. The password is encrypted.
3. The encrypted password is read from database.



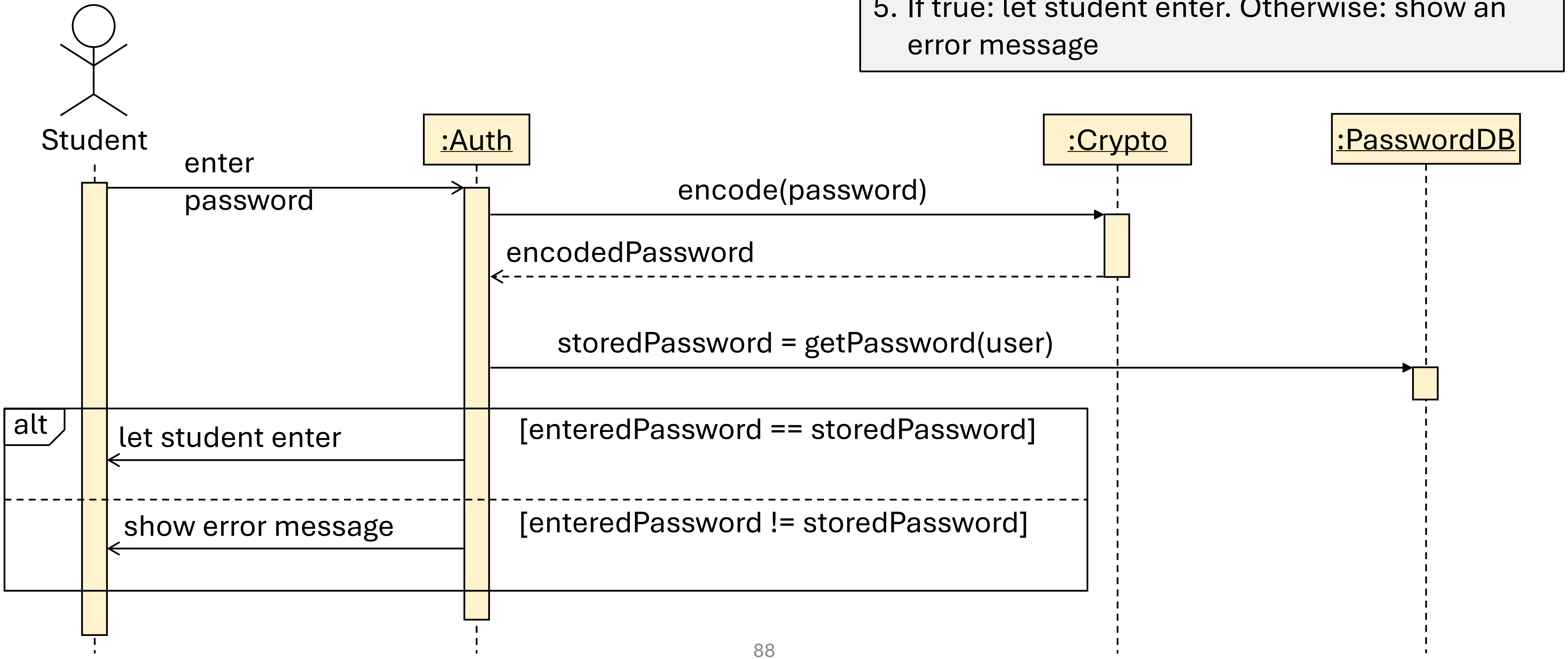
Sequence Diagram: Authentication

1. The student enters the password.
2. The password is encrypted.
3. The encrypted password is read from database.
4. Check if encrypted entered password equals stored password.


Student
:Auth
:Crypto
:PasswordDB
enter password
encode(password)
encodedPassword
storedPassword = getPassword(user)
alt
[enteredPassword == storedPassword]
[enteredPassword != storedPassword]

Sequence Diagram: Authentication

1. The student enters the password.
2. The password is encrypted.
3. The encrypted password is read from database.
4. Check if encrypted entered password equals stored password.
5. If true: let student enter. Otherwise: show an error message



Questions?