REVERSE ENGINEERING

What is reverse engineering?

Example: How to use basic UML for reverse engineering

Tools

Class Diagram Tips

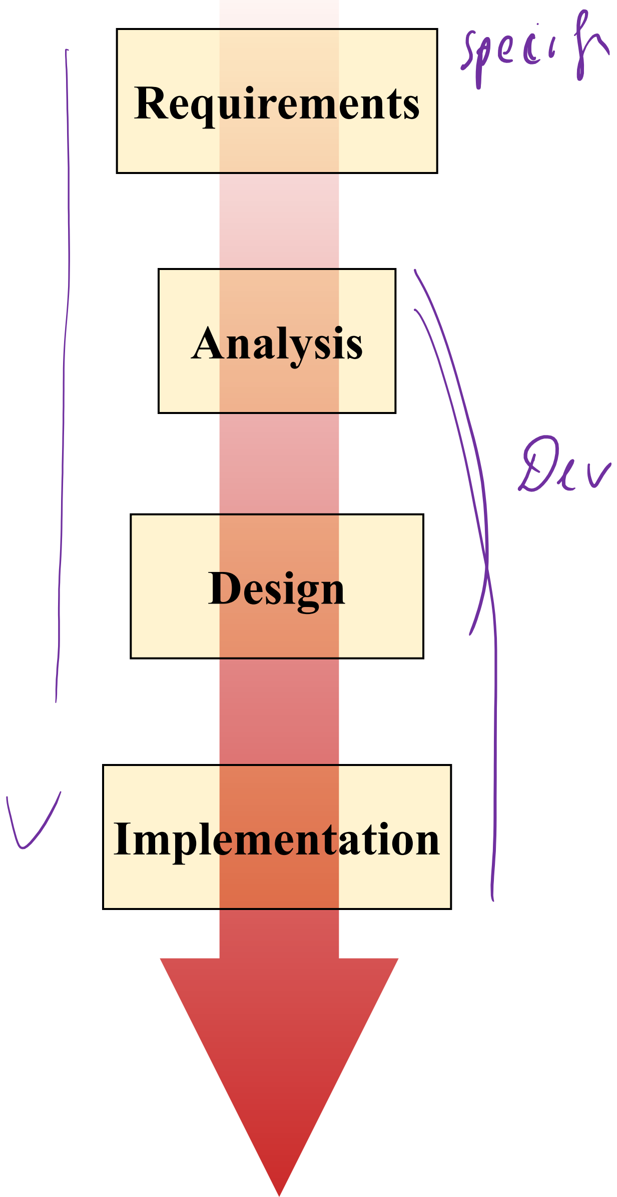
In Your Assignment

WHAT IS REVERSE ENGINEERING?

REVERSE ENGINEERING

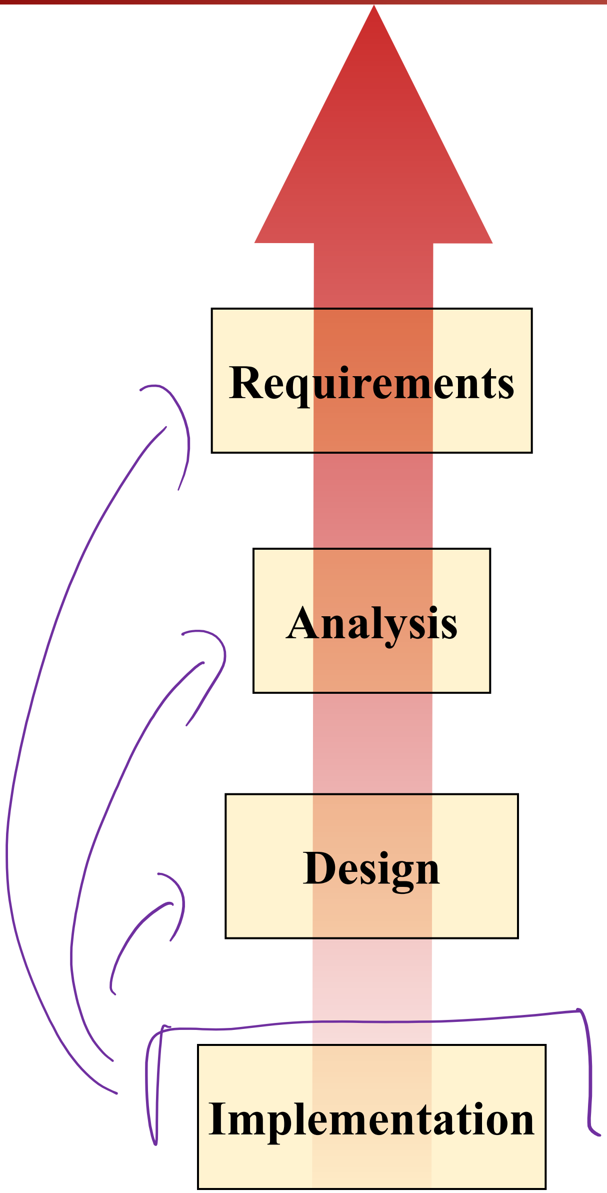
What do you think is reverse engineering?

BACK TO FORWARD ENGINEERING



Goal: Implementation that fulfills requirements

REVERSE ENGINEERING



Requirements

Analysis

Design

Implementation

Goal: Start with implementation and go back to requirements (or anywhere in between)

Reverse Engineering

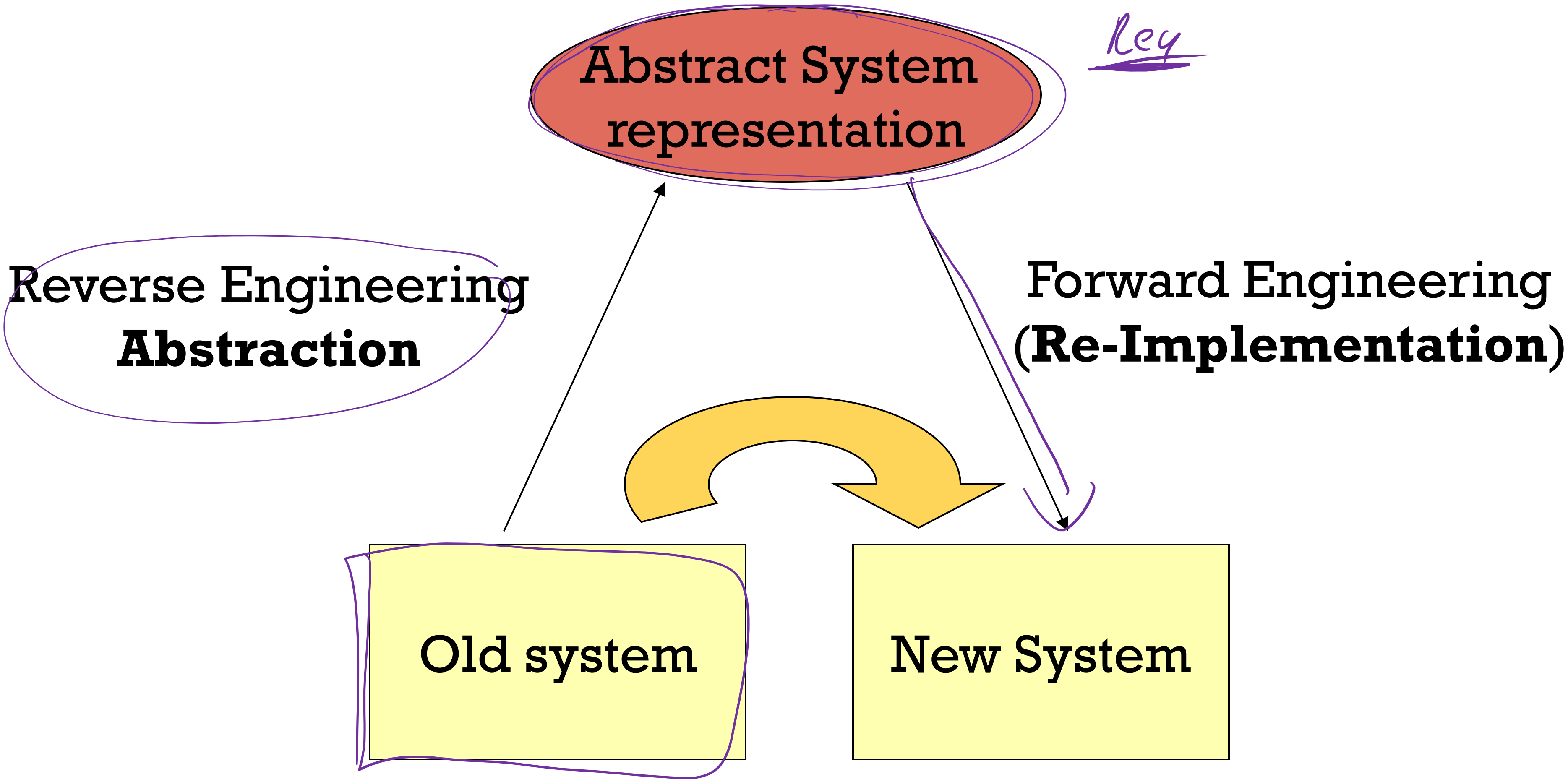
- Sommerville:
  - The program is analyzed and information extracted from it. This helps to document its organization and functionality.

- Wikipedia:
  - Reverse engineering is the processes of extracting knowledge or design information from anything man-made and reproducing it or reproducing anything based on the extracted information.

Scope and Task of Reverse Engineering

- Understanding legacy code
- Understanding code structure
- Understanding code and design at different levels of abstraction
- Identifying reusable code
- Identifying how to enhance the system
- Re-documentation

ABSTRACTION



Abstract System representation

Reverse Engineering
Abstraction

Forward Engineering
(**Re-Implementation**)

Old system

New System

UML
Req

CLEAN ROOM DESIGN

- Reengineer/copy software by imitating the behavior, but write the code from scratch (black box view)
- Can be used to replicate software while avoiding copyright claims
- But it can not avoid patent claims
- Process:
  - View program/function/method as a black box
  - Evaluate behavior (like in a black box analysis)
  - Write own program/function/method from scratch and copy the behavior of the black box analysis

SECURITY AUDIT

- Security audit through reverse engineering
  - Checking if critical components behave as specified
  - Process:
    - Reverse engineer the program
    - Compare observed behavior with specifications
    - Find mismatches between observed behavior and specifications
  - Can be done with whitebox and blackbox analysis

COUNTER MEASURE: OBFUSCATION

- Obfuscation
  - Changing the code…
    - … in a way that it can’t be easily read anymore
    - … in a way that it doesn’t change its behaviour
  - Example:
    - Changing all variable names into different combinations of lowercase “L” and uppercase “i” (l & I) or uppercase “o” and zero (O & 0)
      - E.g. IIIllIll & lIIlllIl (two different variables)
  - A counter measure against Reverse Engineering

SUMMARY

- Reverse Engineering going from code to a different level of abstraction
- Can be used to
  - Understand legacy code
  - Identify reusable code
  - Create new version of code
  - Etc.
- Often done through using UML models

3/6

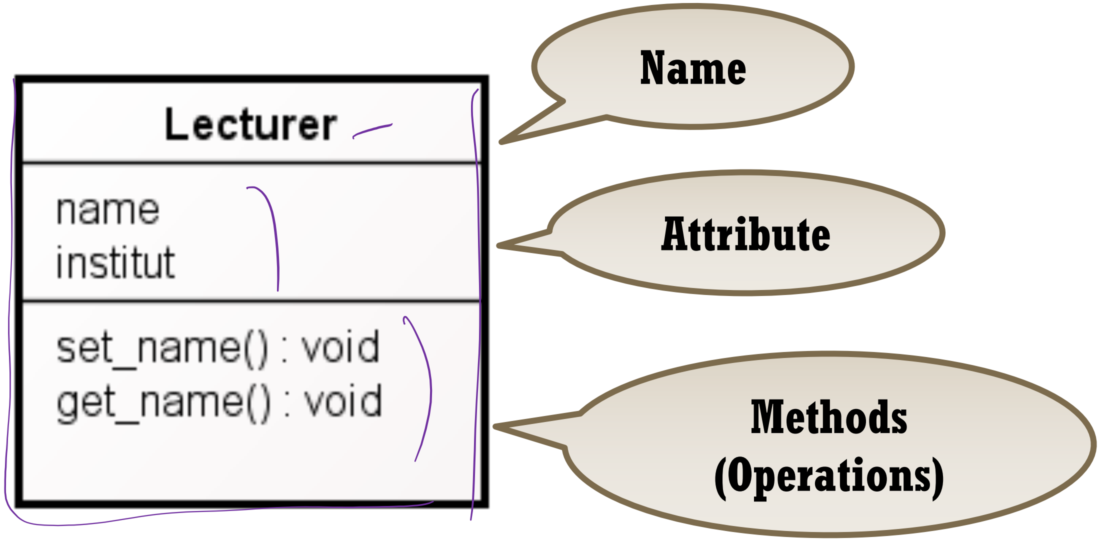
HOW TO USE BASIC UML FOR REVERSE ENGINEERING

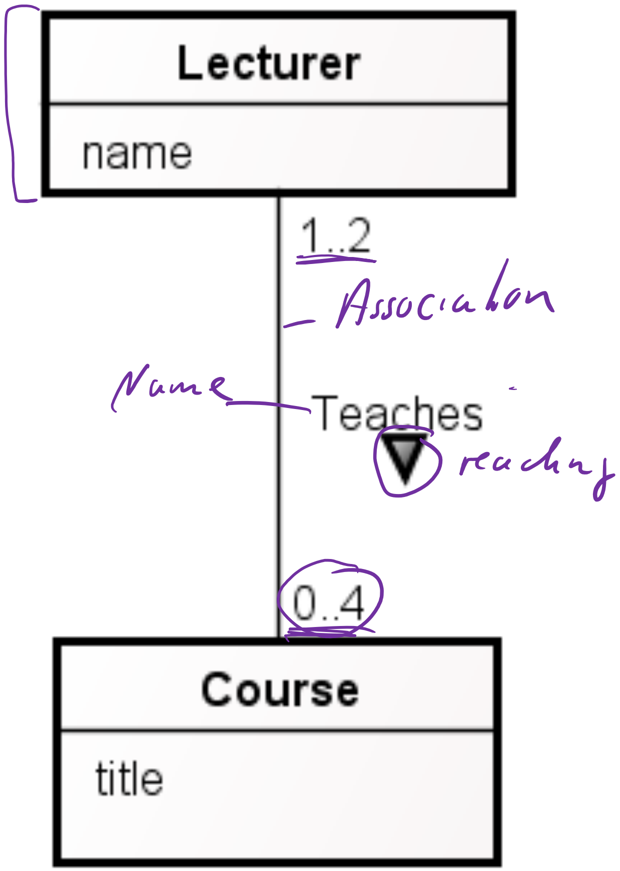
Basics UML Class diagram

UML Class diagram to code and back

BASIC UML CLASS DIAGRAM

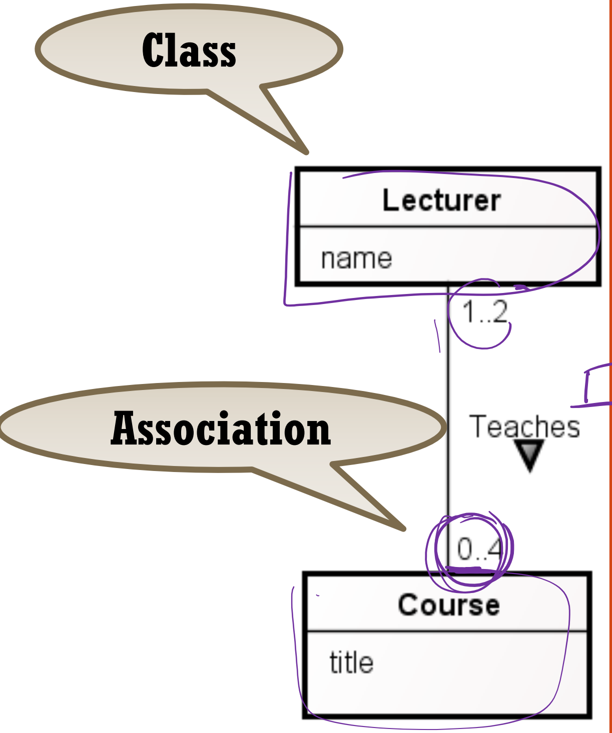
MODELING OF CLASSES



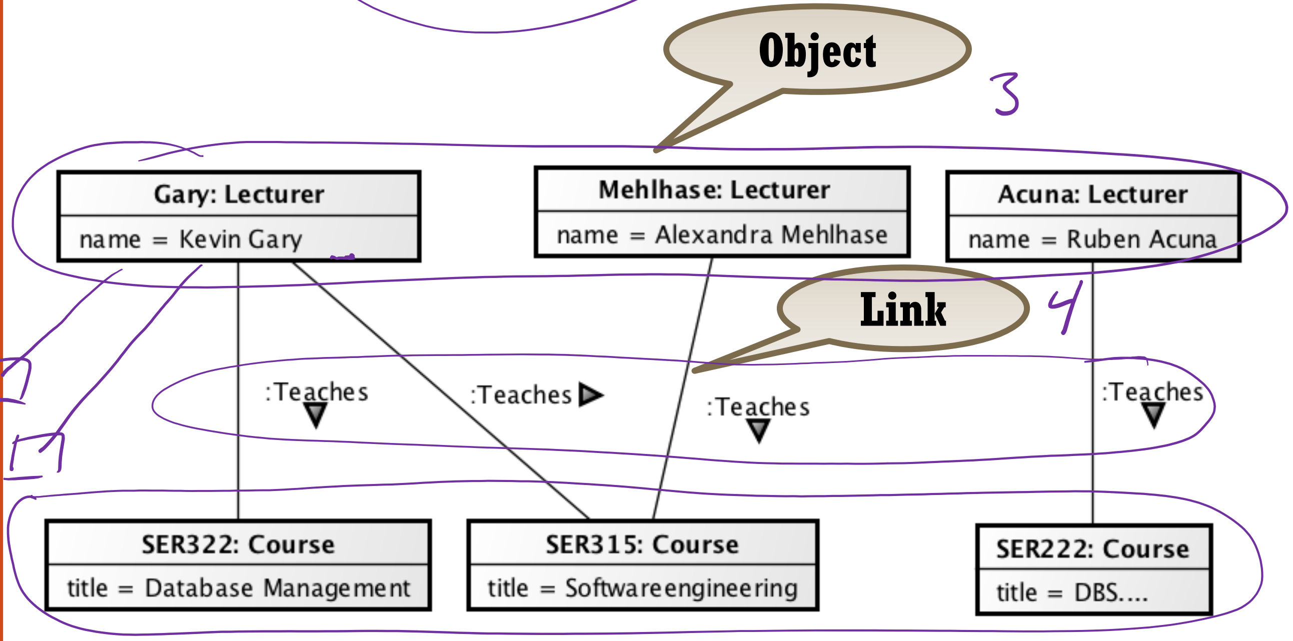


CLASS VIEW VS. OBJECT VIEW

Classes and associations

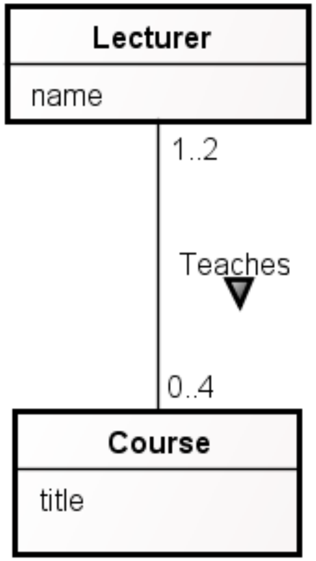


Objects and links

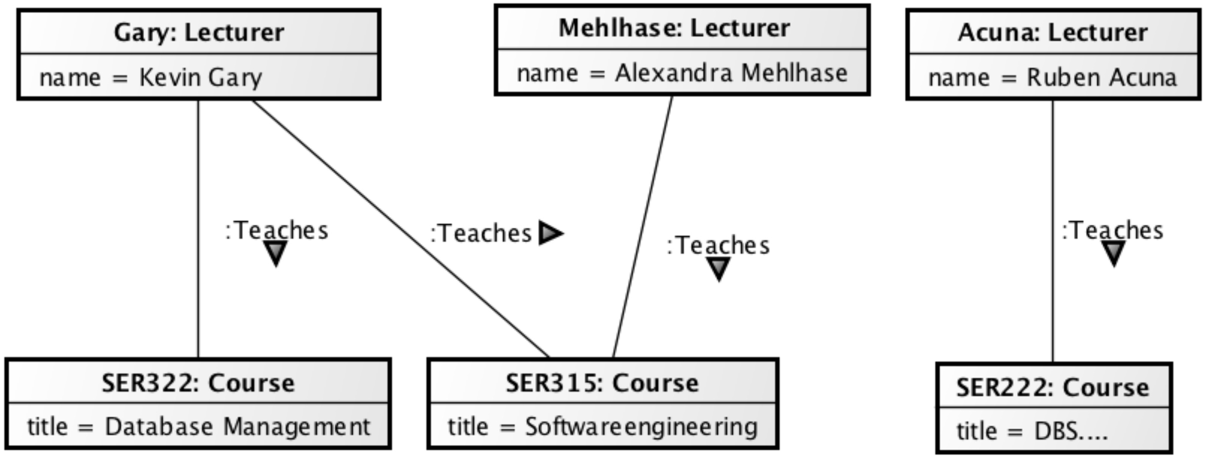


CLASS VIEW VS. OBJECT VIEW

Classes and associations

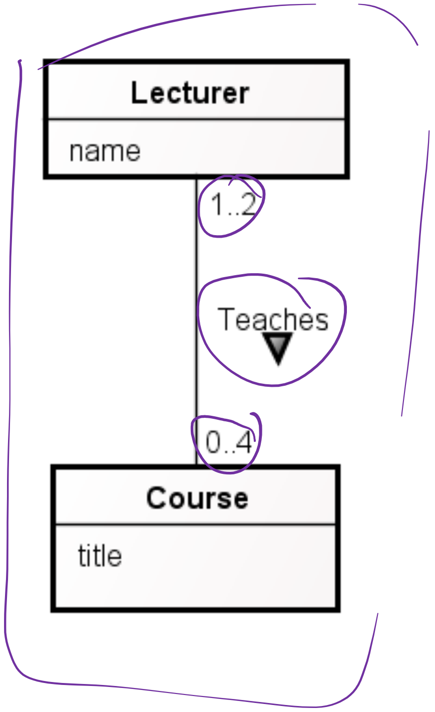


Objects and links

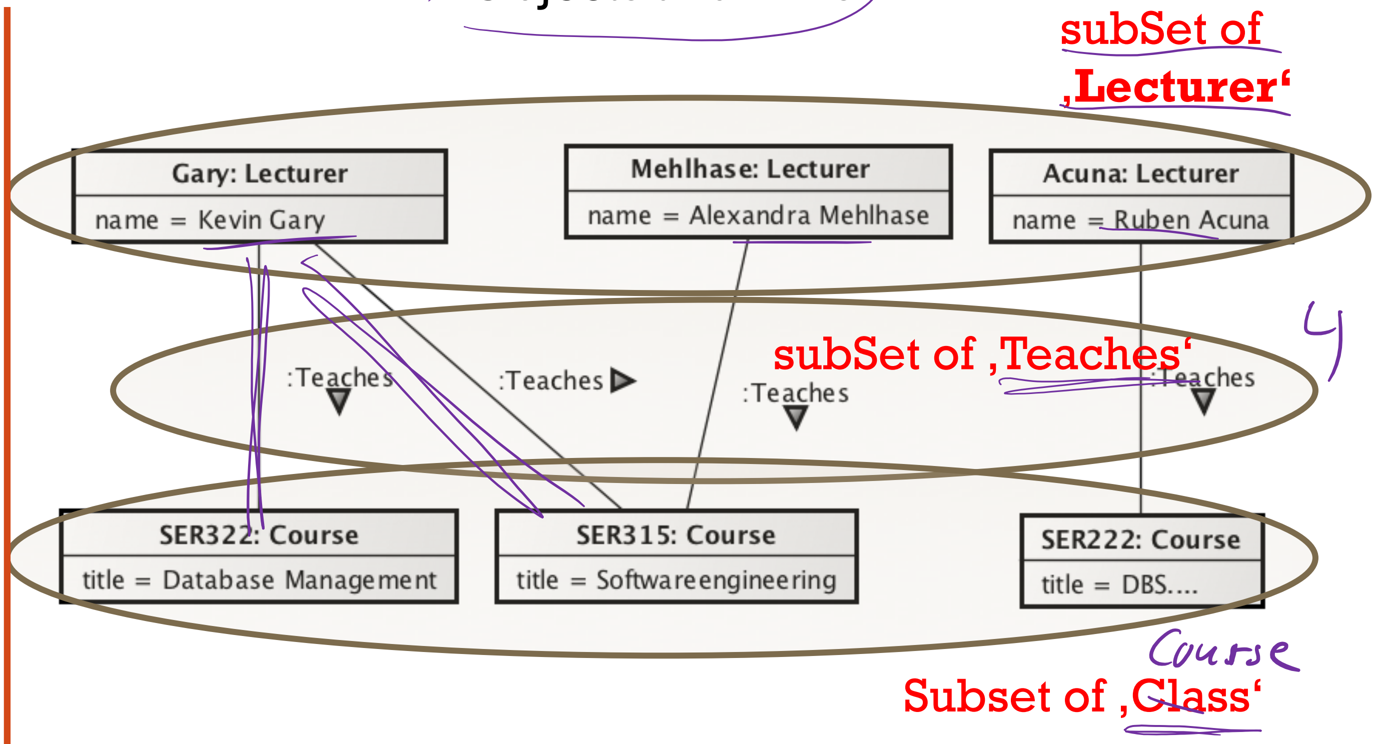


CLASS VIEW VS. OBJECT VIEW

Classes and associations



Objects and links



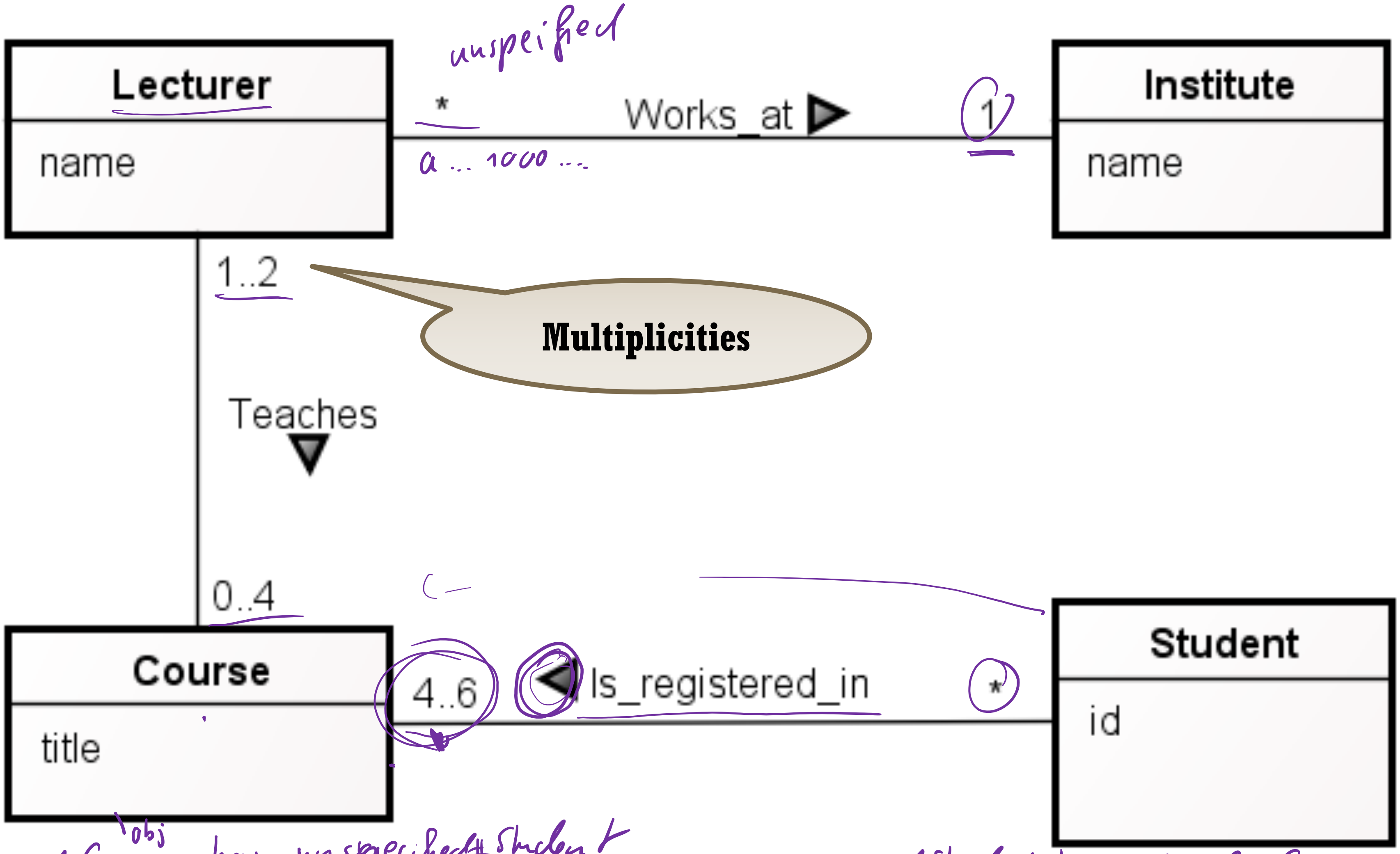
Lecturer, *Course* and *Teaches* sets of concrete system state

Lecturer = {Gary, Mehlhase, Acuna} : $\mathbb{P}$ **Lecturer**

Course = {SER322, SER315, SER222}: $\mathbb{P}$ **Course**

Teaches = {(Gary, SER322), (Gary, SER315),
(Mehlhase, SER315), (Acuna, SER222)} : $\mathbb{P}$ **Teaches**

MULTIPLICITIES



unspeified

Lecturer

name

*

a ... 1000 ...

Works_at ▶

1

Institute

name

1..2

Multiplicities

Teaches ▼

0..4

Course

title

4..6

◀ Is_registered_in

*

Student

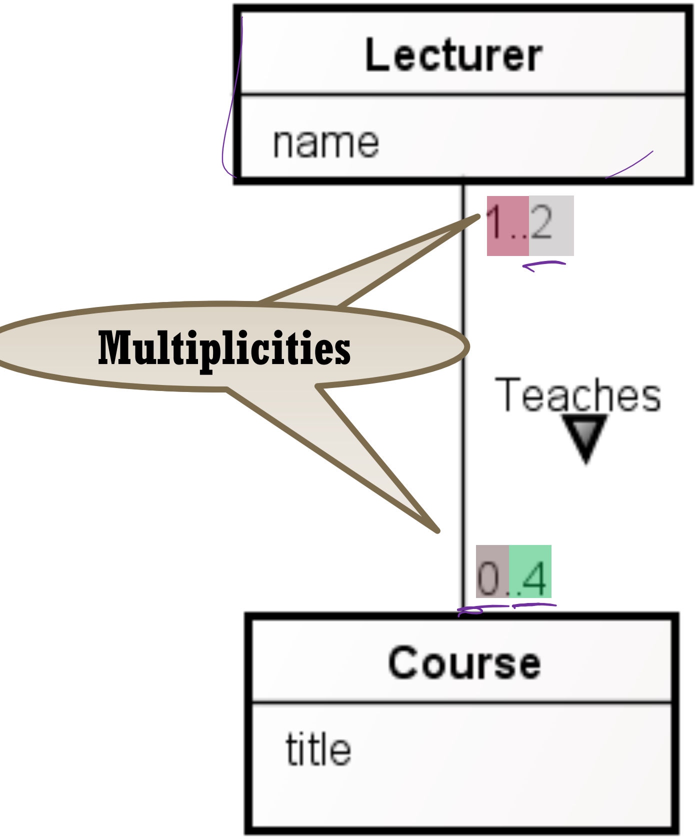
id

obj

1 Course has unspecified # Student

1 Student has 4-6 Courses

MULTIPLICITIES



Determine, how elements are connected. Muliplicities are statements about the set of links!

In this example, the statements about the **Teaches** relation are:

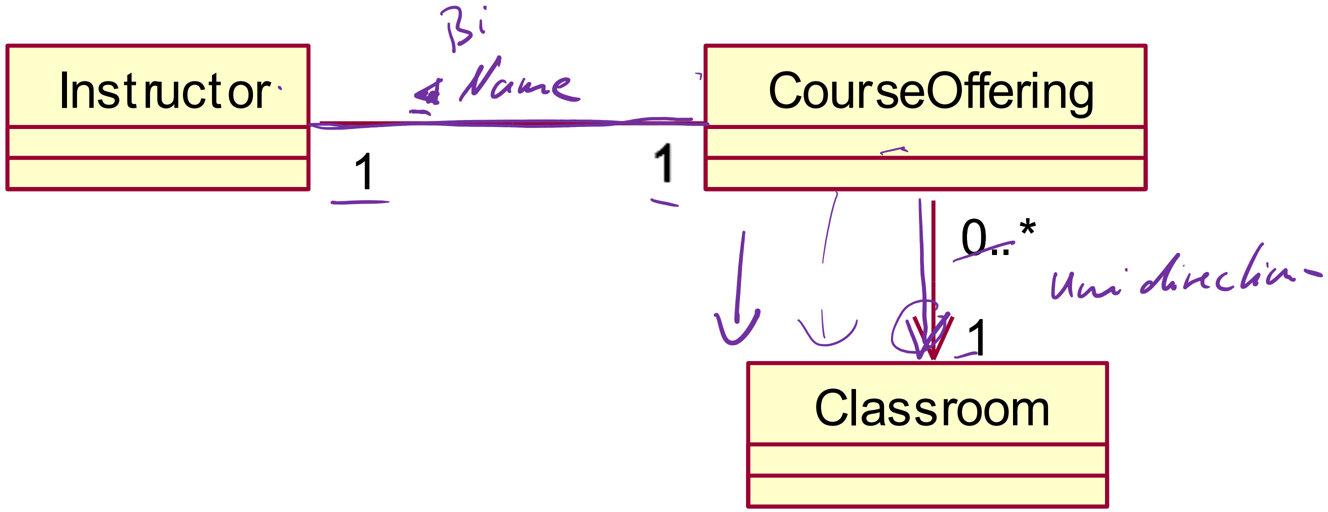
- Each element of the set **Course** (in this system) occurs in Teaches
  - Each class is therefore taught by at least one lecturer
- A class is taught by at most two lecturers
- A lecturer might not teach a class
- A lecturer teaches not more than four classes

SUMMARY

- Classes can be real world objects or classes to be implemented
- Objects are instances of classes
- Associations represent relation between classes
- Association names help with readability and should have reading directions
- Associations should have multiplicities that describe allowed system states

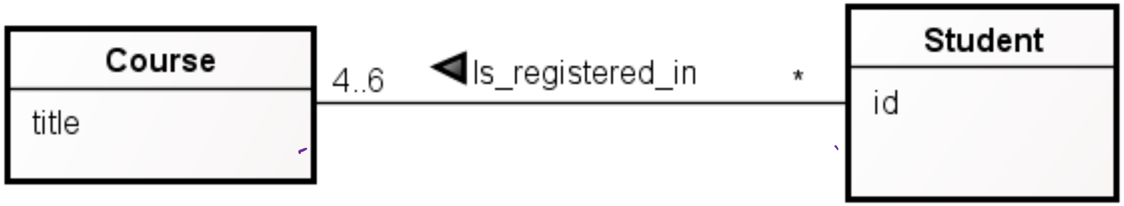
ASSOCIATIONS

UNI- AND BI-DIRECTIONAL

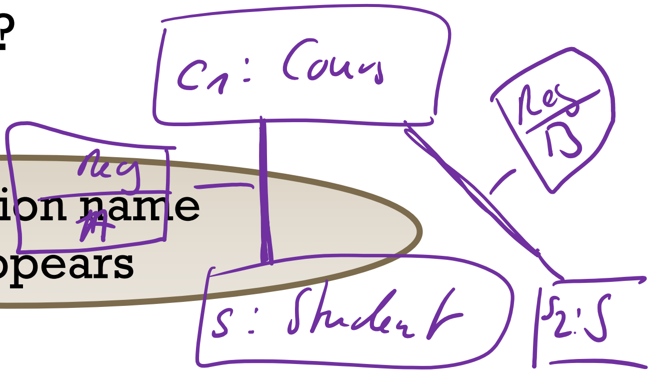


CLASS DIAGRAMS: ASSOCIATION CLASS

some attributes are difficult to be allocated to classes:

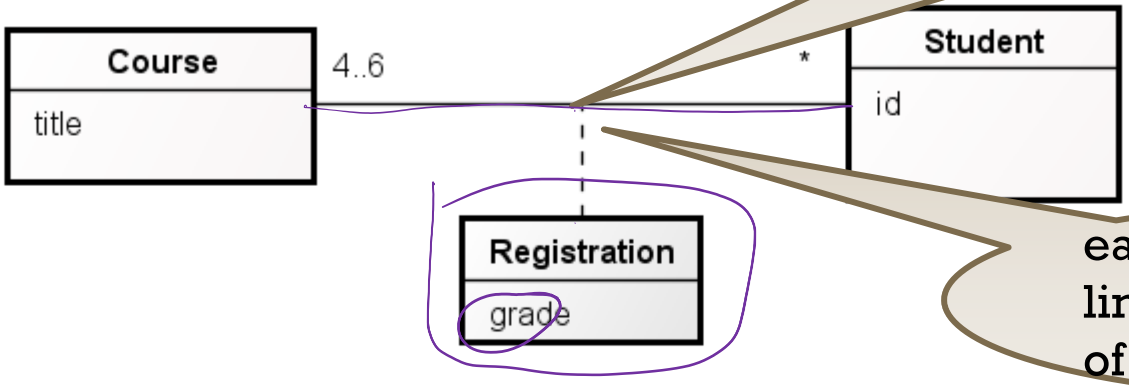


- where should the attribute **grade** be allocated?



use an association class:

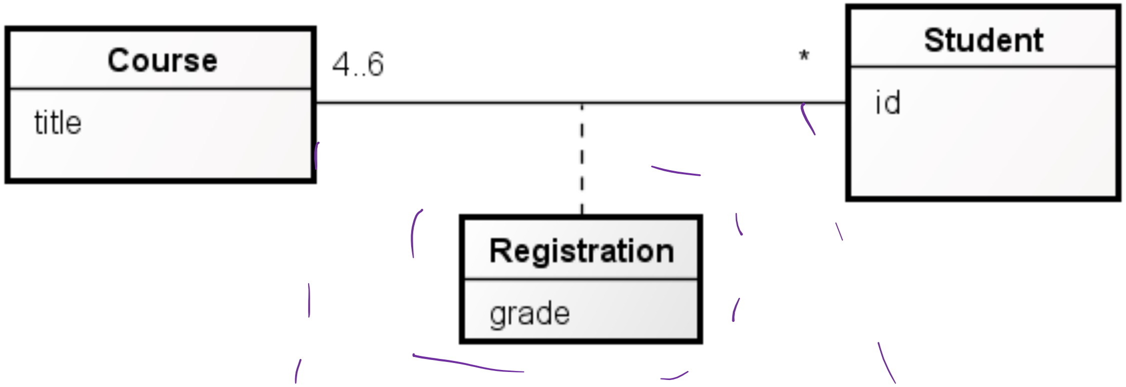
association name disappears



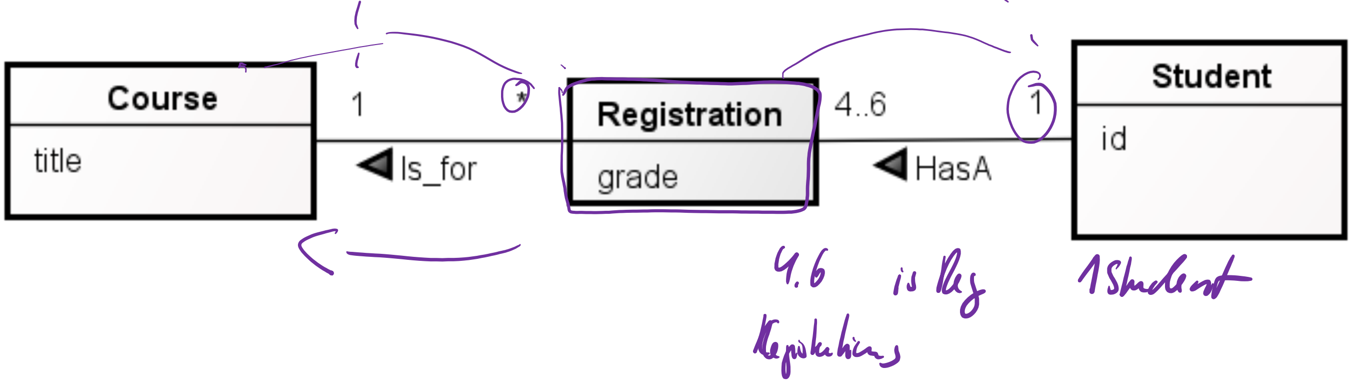
each **Student-Course** link contains an object of type **Registration**

ASSOCIATION CLASSES CAN BE ELIMINATED

- replace the association and the association class



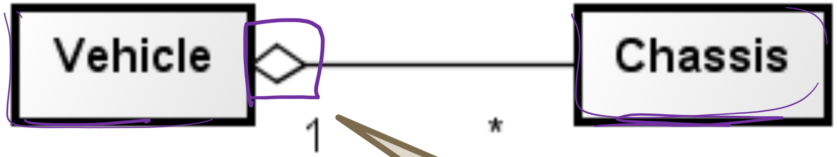
- by two (one-to-many) associations:



AGGREGATION

Aggregates represent **part-whole** associations between instances.

aggregate
class

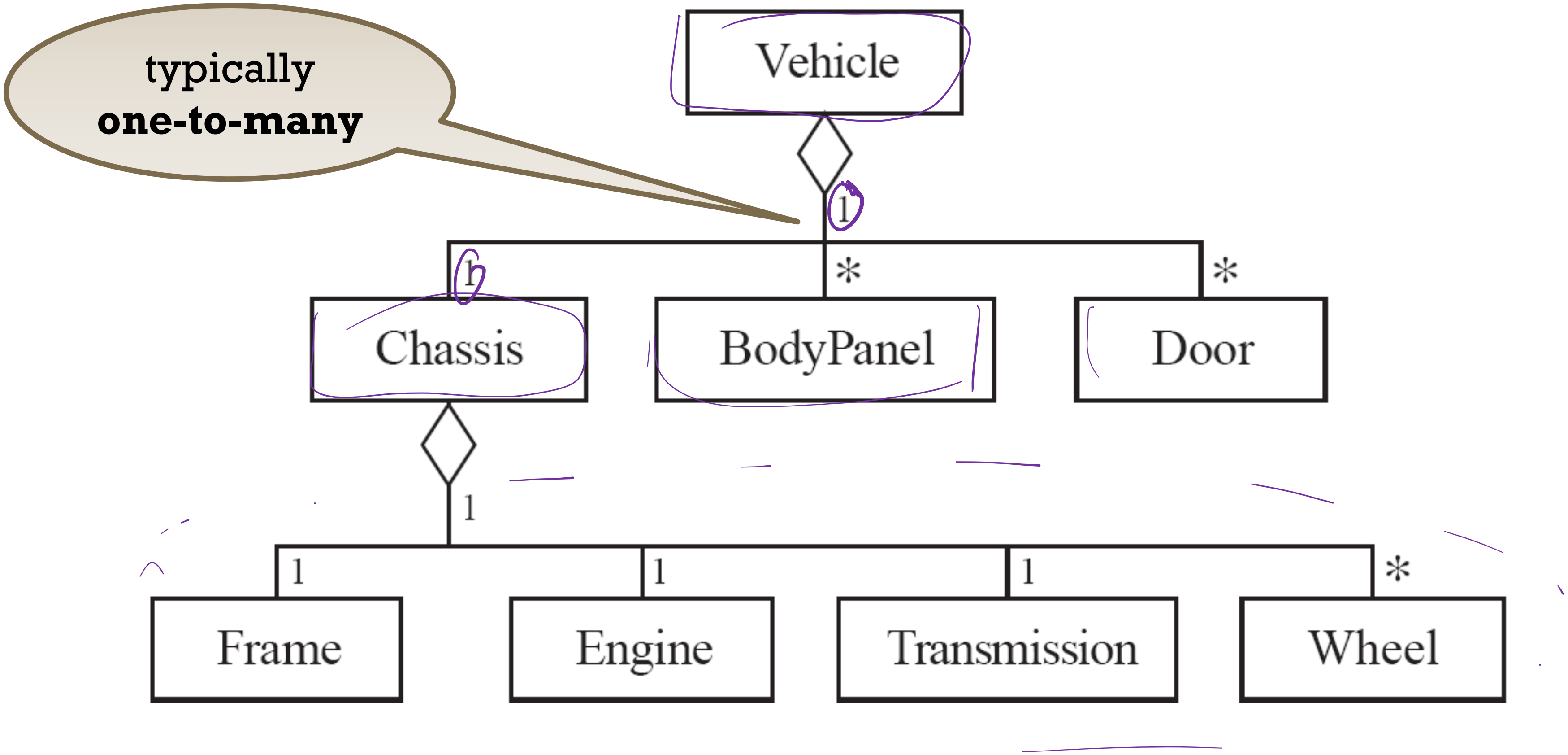


implicit **partOf**
association

To be used when

- parts are part of the aggregate (or the aggregate is composed of the parts)
- whoever owns or controls the aggregate, also owns or controls the parts

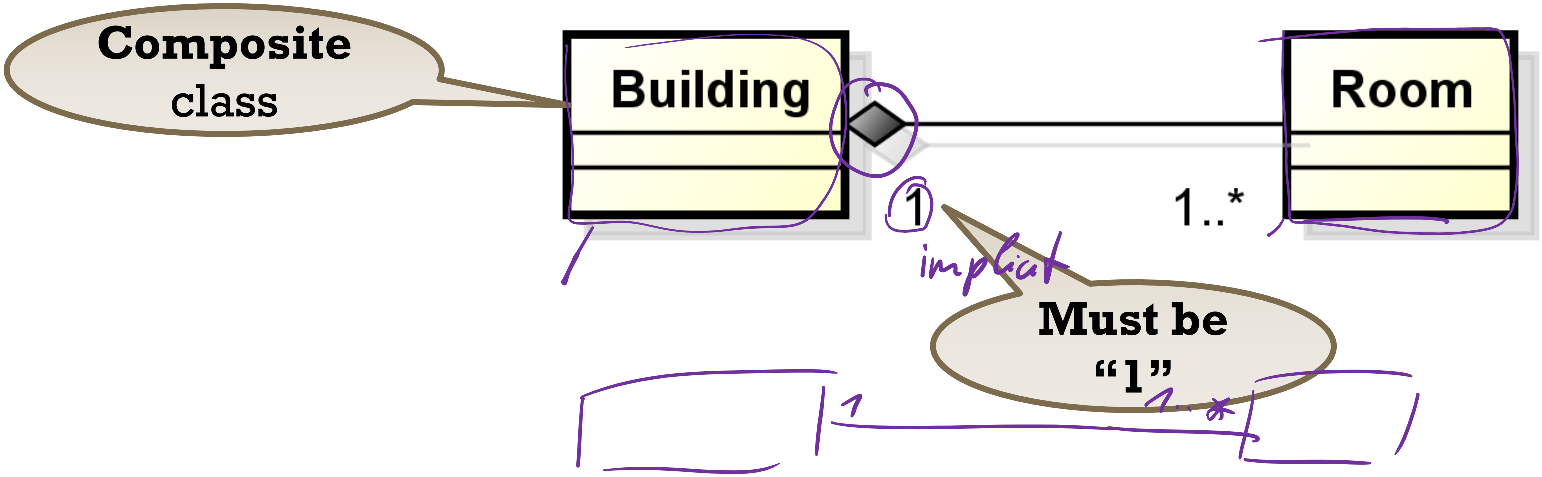
AGGREGATION HIERARCHIES



- Aggregation can be arranged hierarchically
- Note: Different from inheritance!

COMPOSITE ASSOCIATIONS

Composite associations represent **strong part-whole** associations between instances.



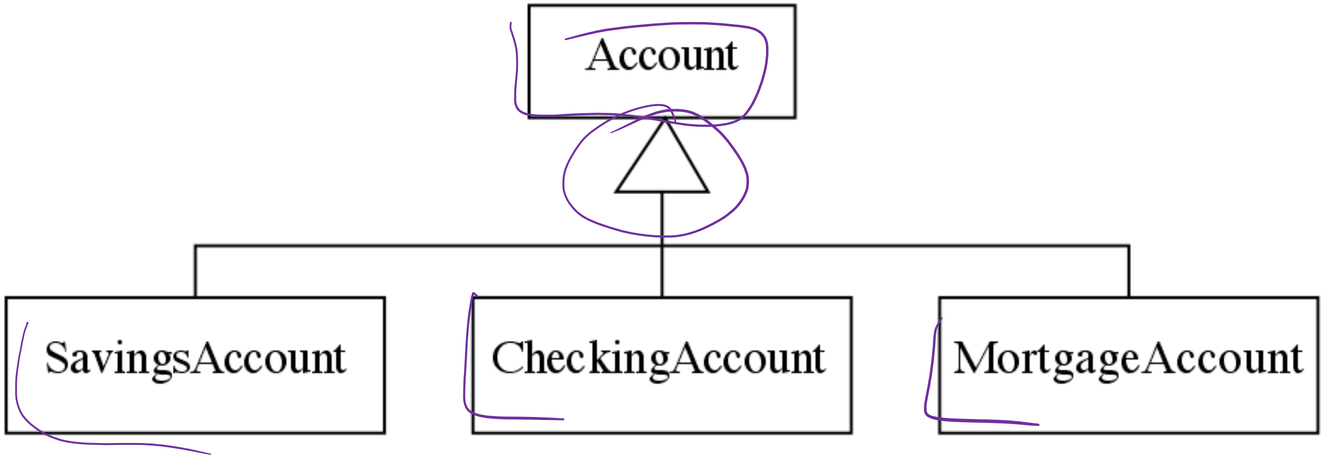
Composite class

Must be "1"

This is used when the part cannot exist independently from the aggregate.

GENERALIZATION

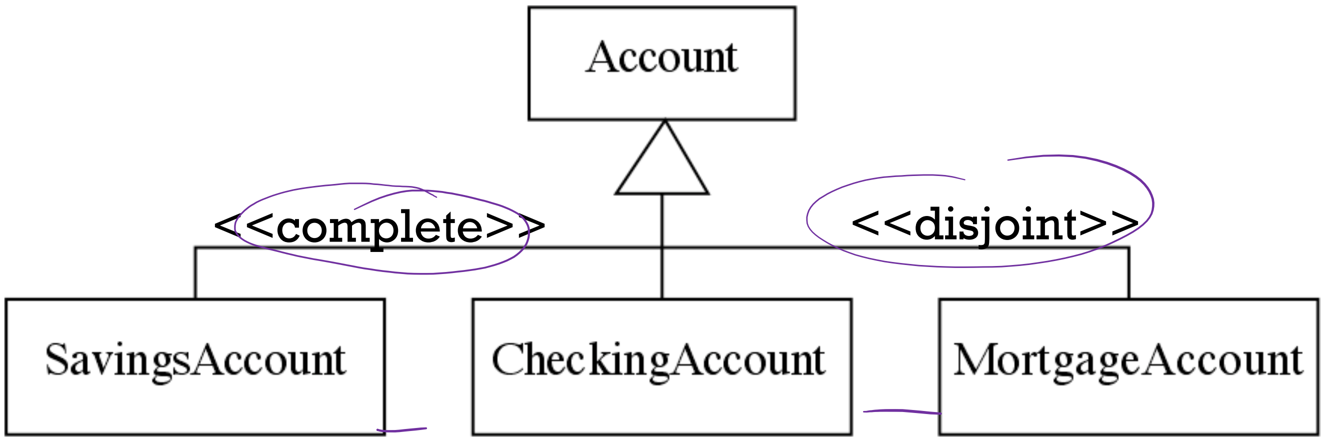
Generalization represents **inheritance hierarchies between classes** and not an association between objects.



Inheritance must obey the "is-a" rule:

- "*A <subclass> is a <superclass>*" must make sense.

GENERALIZATION



Stereotypes:

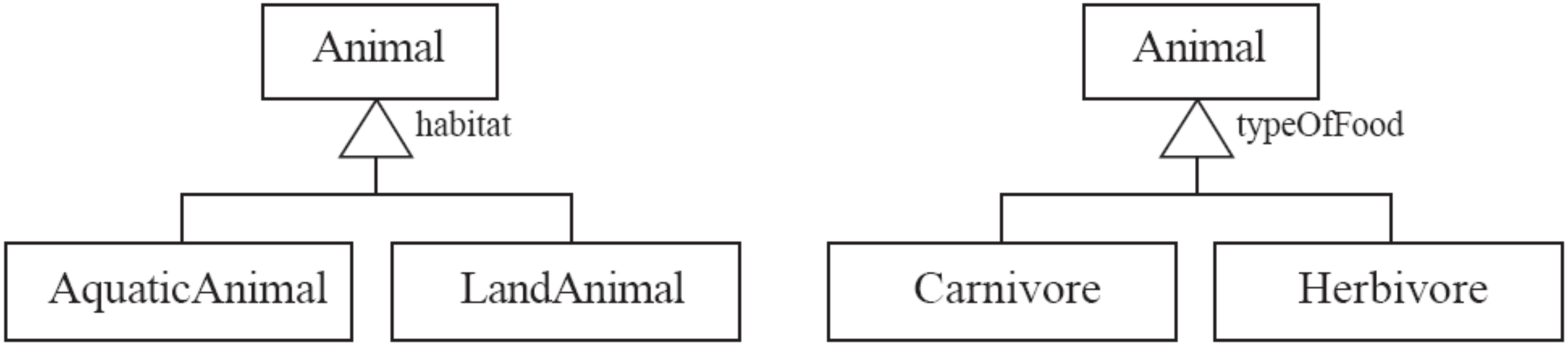
- <<*complete*>> indicates an abstract superclass

  (i.e. all instances belongs to a subclass)

- <<*disjoint*>> indicates that each instance belongs only to a single subclass

  (opposite: <<overlapping>>)

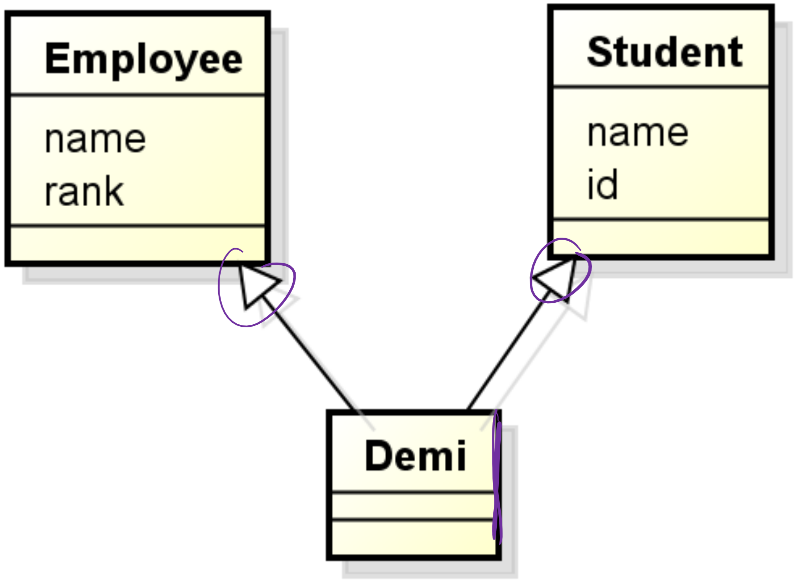
GENERALIZATION



- A discriminator label can be used to denote the criterion
- which distinguishes different subclasses.

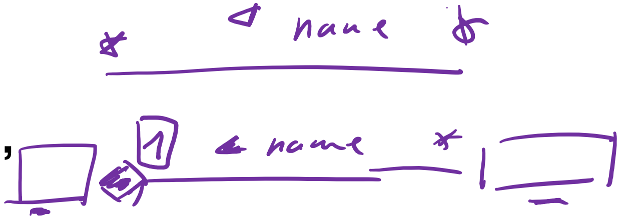
This is useful, especially for disjoint generalizations.

PITFALLS - MULTIPLE GENERALIZATIONS



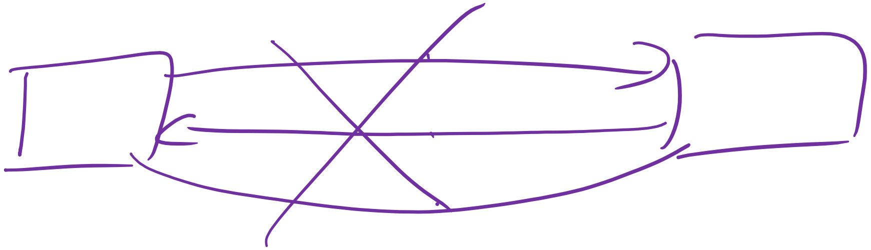
- UML allows multiple inheritance
  - but Java does not
  - in addition, problems pop up when using the same name for attributes in the superclasses

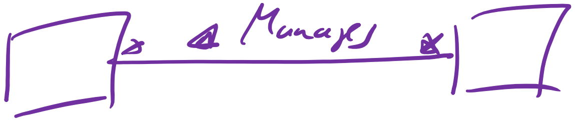
SUMMARY



- Association between classes are the "base case"
- Aggregations and Compositions are special cases of Association with specific meaning

- Generalization is not an association
- An Association is always between 2 classes
- Two classed should only be connected through one association





UML CLASS DIAGRAM TO CODE AND BACK

ABSTRACT DATA TYPES OR STRUCTURED CODE

- Self-defined data structures:
  - Classes as a data set (records)
  - Collection of labeled fields (attributes)

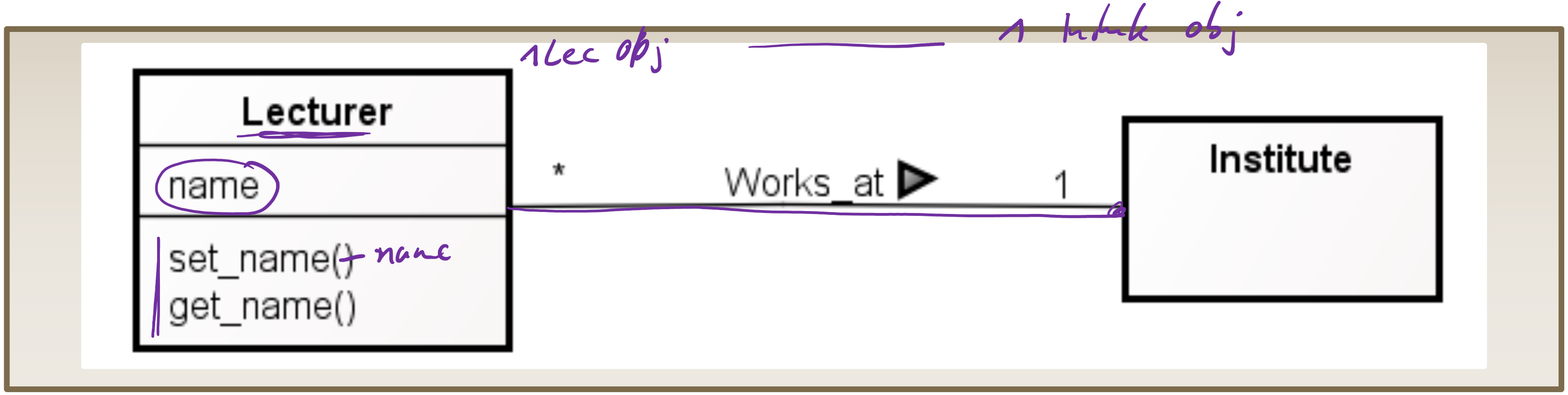
```
class Lecturer{
      String name;
      Institute institute;
}
class Institute {
      // …
}
```

CLASS-INSTANTIATION

- Declaration of variables
- Creation of instances / objects

```
class Lecturer{
        String name;
         Institute institute;
}
...
Lecturer kg = new Lecturer();
kg.name = „Gary";
kg.institute = new Institute(„SE")
```

REFERENCES IN CLASS-DESCRIPTIONS



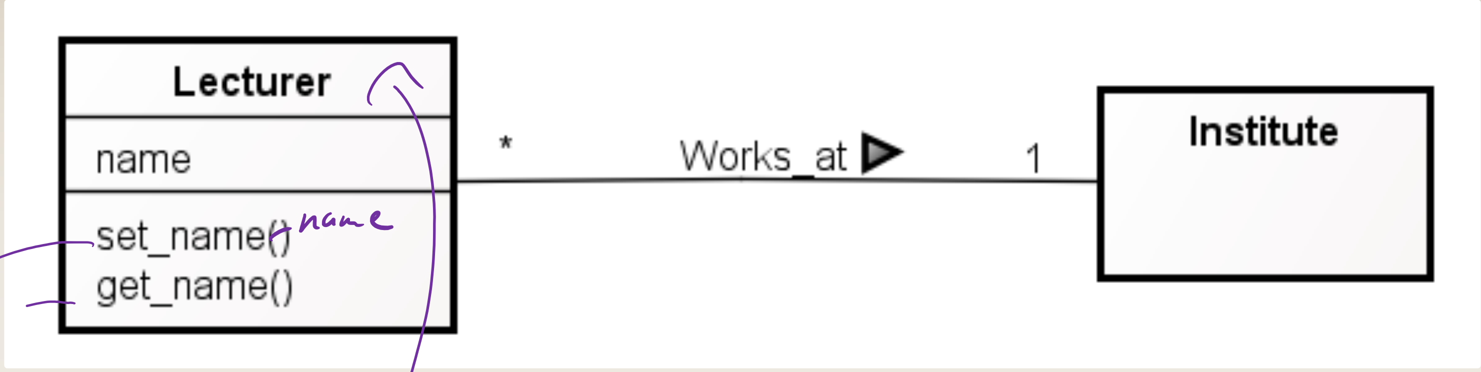
```
class Lecturer{
    String name;
    Institute institute;
}
...
Lecturer kg = new Lecturer();
kg.name = „Gary“;
kg.institute = new Institute(„SE“)
```

OPERATIONS WITHIN CLASSES

- Operations, „methods“
  - used to implement the behavior of instances of the class
  - have a hidden argument like `this, self, current` (dependent on language)

```
class Lecturer{
…
   void set_name(String name) {
       this.name = name;
   }
   String get_name() {
       return this.name;
   }
}
```

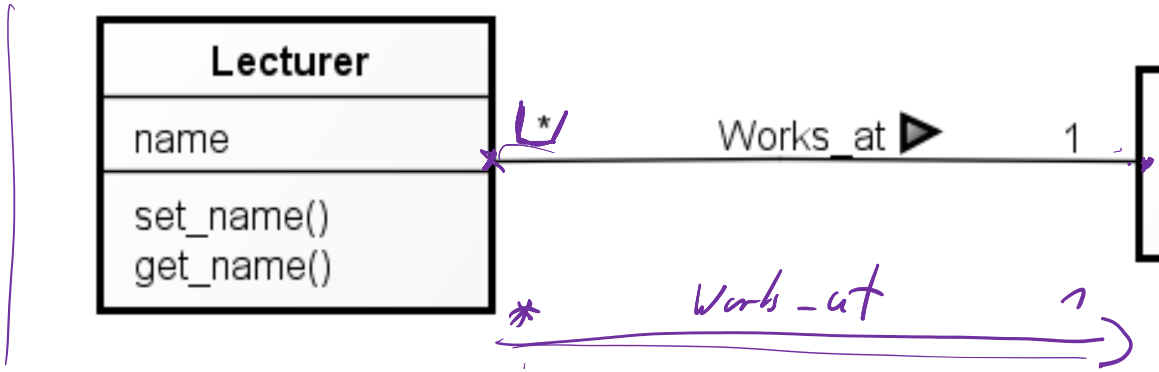
OPERATIONS IN CLASS-DESCRIPTIONS



```
class Lecturer{
…
  void set_name(String name) {
       this.name = name;
  }

  String get_name() {
       return this.name;
  }
}
```

BI-DIRECTIONAL

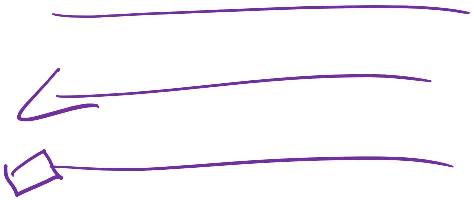


```
class Institute{
        Lecturer lecturer[];
…
}
```

TOOLS

ASTAH

- Modeling tool for UML
- Can model different UML diagrams and has some dependency rules
  - We will talk about these rules in due course
  - Be careful to remember that UML is a language and therefore different lines etc. mean different things
- Astah can forward engineer (Class diagram to Java)
- Astah can reverse engineer (Java to UML class diagram)



OTHER TOOLS

- Microsoft Visio
- Rational Software Modeler ✓
- UML Designer ✓
- And many many more
  - https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

CLASS DIAGRAM TIPS

TIPS

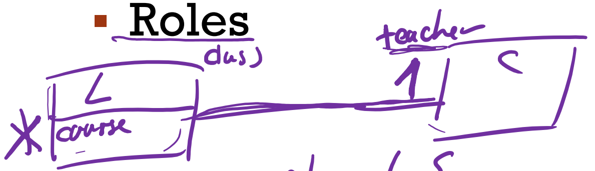
- Make sure you know what level of abstraction you want to use. E.g. what is included
  - Methods
  - Types
- Start of with using general associations (bi-directional)
  - If you think you need something stronger then consider aggregation
  - Consider if you need a composition
  - Make sure you understand the implication of these two things
    - They can be used in a wrong way, using a "normal" association is usually not wrong
  - Remember that Generalization is not an Association
- Name your classes and associations well, so the diagram can be understood

IN YOUR ASSIGNMENT

Reverse Engineering

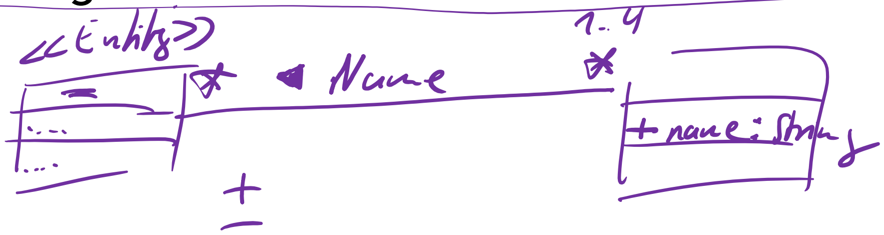
REVERSE ENGINEERING ASSIGNMENT

- You should use a specific level of detail for the class diagram you create
  - Create classes for each class in the code
  - Include attributes and methods (with types), multiplicities, association names and reading directions
  - Include access specifiers if they are used in the code (if they are not specified in the code you don't need them)
  - You **do not have to** include constructors
  - You **should not** include
    - Sterotypes
    - Classes you have in the code/diagram as attributes in other classes
    - ➔Use Associations for that
    - Roles

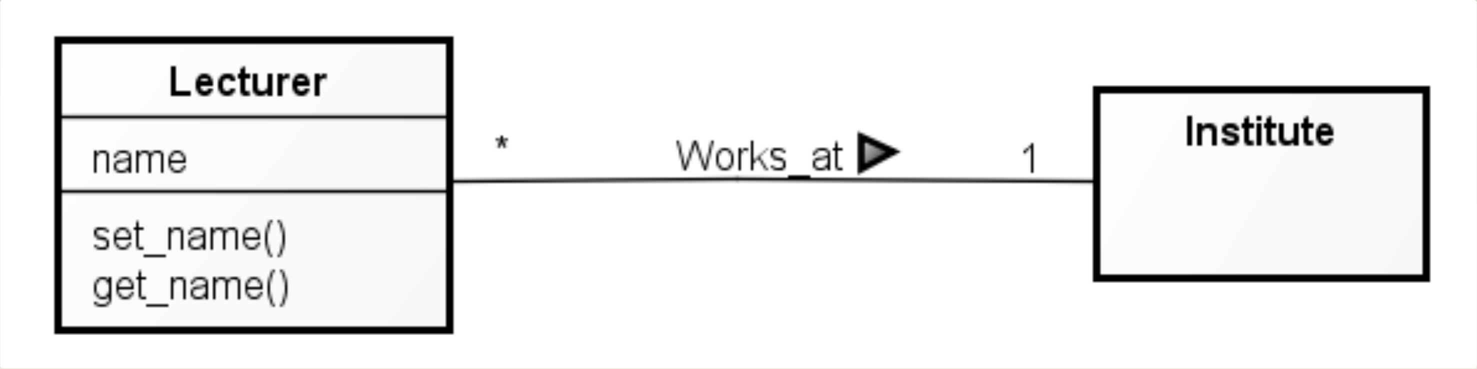


class L {

Course ccourse ;



OPERATIONS IN CLASS-DESCRIPTIONS



```
class Lecturer{
…
 void set_name(String name) {
   this.name = name;
 }

 String get_name() {
   return this.name;
 }
}
```