# Unified Modeling Language (UML) Part II

CS-A1153: Databases

Prof. Nitin Sawhney

Aalto University
School of Science

## Acknowledgements

These slides are based in part on presentation materials created by **Kerttu Pollari-Malmi** and **Juha Puustjärvi** in previous years and on the course text book: **A First Course in Database Systems**, Third Edition. Pearson by Jeffery D. Ullman and Jennifer Widom.

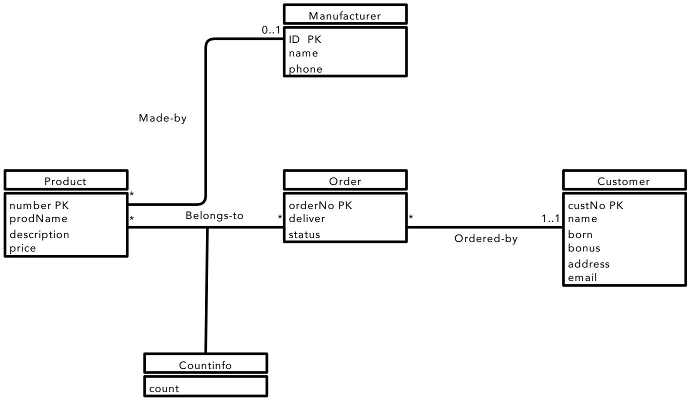
Thanks to **Etna Lindy** & **Ville Vuorenmaa** for translating prior lecture slides for this course.

## Learning Goals

- Creating a relational model based on UML diagrams, i.e. from UML define the relations and what attributes these relations should have (for a database schema).
- Explaining why one database schema may be better than another schema describing the same database.
- Understanding some concepts and techniques needed to improve the database schema, for instance:
  - Functional Dependency
  - how the key of the relation is connected to functional dependencies

# Changing UML Diagram to Database Schema

What relations would you define based on this diagram?



Aalto University
School of Science

## Changing UML Diagram to Database Schema

Basics:

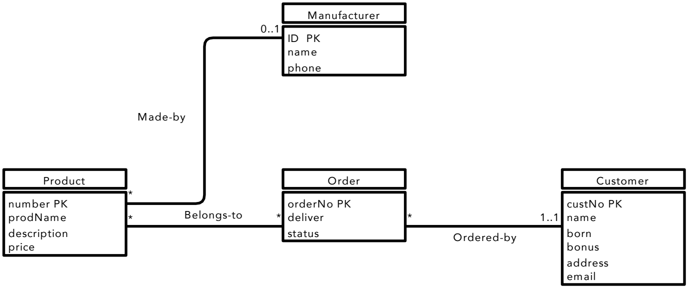
- Each class gets its own relation, which has the same attributes as the class.
- For each many-many association we create a relation, whose attributes are the keys from both of the relations the association connects, and the attributes of the association class, if there is one.
- For each many-one association we can either create a relation in the same manner as with many-many associations or we can add attributes to the relation that are on the many-side of the association.
- Rename the attributes in the relations, if needed, to make them clear and distinct.

Aalto University
School of Science

# Example: Defining Relations for UML Classes

- For the classses in the UML diagram of the online store we can define relations:

  Customers(<u>custNo</u>, name, born, bonus, address, email)
  Products(<u>number</u>, name, description, price)
  Manufacturers(<u>ID</u>, name, phone)
  Orders(<u>orderNo</u>, deliver, status)



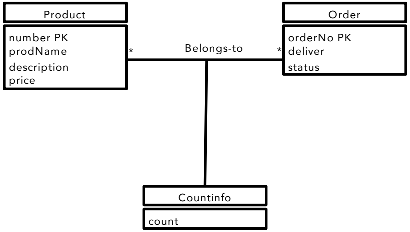
# Example: Defining Relations for Associations

- For the associations we define relations: *(later we will introduce another approach)*

  MadeBy(<u>number</u>, ID)

  OrderedBy(<u>orderNo</u>, custNo)

  BelongsTo(<u>orderNo</u>, <u>number</u>, count)



*Note: No relation needs to be created for the Countinfo class.*

## Changing UML Diagram to Database Schema

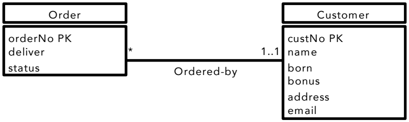
Special cases (that cannot be handled in a straight-forward manner with the basic approaches):

- Classes, with attributes inherited from other classes, and associations with such classes, must be handled differently.
- Subclasses need to be handled in a different manner.
- Consider whether or not a many-one association should have its own relation, or should the relation be replaced by simply adding the key from the one-side class to the attribute list of the many-side class.

Aalto University
School of Science

## About Keys of Relations

- A relation defined based on a class has the same key attributes as the class.
- Relations based on many-many associations inherit the key attributes from both of the classes connected by the association.
- Relations based on many-one associations only inherit the key attributes from the class in the many-side.

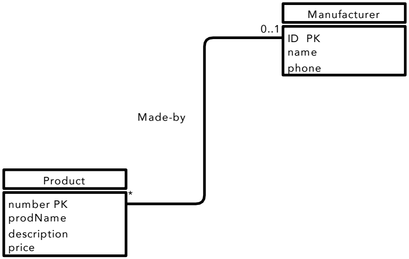


For example, OrderedBy(<u>orderNo</u>, custNo)

# About Keys of Relations, Continued

- It's important not to define too many key attributes, as with the key attributes we make sure, that two tuples with the same values in the key attributes cannot be added.

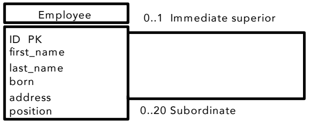
  For instance, if the relation *MadeBy* would also have *ID* as its key attribute, we could add multiple tuples with the same product number, but with a different manufacturer.

  E.g. MadeBy(<u>number</u>, ID)

## Example: Relations for Self-Associations

- Here the same class occurs in the association twice:



The key attributes will be added twice to describe both sides of the association.

- Hence, we define the relation:
  Manages(<u>subordinateID</u>, superiorID)

Aalto University
School of Science

## Combining Relations

- In the first example we defined the following relations based on the UML diagram:

  Products(<u>number</u>, name, description, price)
  Manufacturers(<u>ID</u>, name, phone)
  MadeBy(<u>number</u>, ID)

- The relation *MadeBy* was defined based on a many-one association.
- The relations *Products* and *MadeBy* can be combined into one relation by adding the information about manufacturer's ID to the relation *Products*.

  Products(<u>number</u>, name, description, price, ID)

  Here, the relation *Manufacturers* stays as it is.

## Combining Relations, Part II

- In a similar manner we may combine the relations *Orders* and *OrderedBy*.
- On the contrary, we can’t combine *BelongsTo* to any other relation, as it has been created based on a many-many association.
- The final relations (some names of the attributes have been changed for clarity):

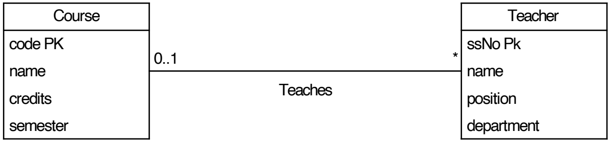
  Customers(custNo, name, born, bonus, address, email)
  Products(number, prodName, description, price, manufID)
  Manufacturers(ID, manufName, phone)
  Orders(orderNo, deliver, status, custNo)
  BelongsTo(orderNo, prodNo, count)

## Combining Relations, Part III

- If an association is a many-many association, it cannot be combined.
- Combining it would lead to repetitive information.
- If *BelongsTo*-relation would be combined with the relation *Orders*, we would end up with an association: Orders(<u>orderNo</u>, deliver, status, custNo, <u>prodNo</u>, count)
- Here, the information about the shipping method, status of the delivery and customer number of the order would be stored to the database multiple times, which is redundant.

## Exercise

Which of the following relations (see next slide) are correct, when creating a database based on this UML diagram?



Aalto University
School of Science

## Exercise Continues

Options:

a. Teachers(ssNo, name, position, department)
   Courses(code, name, credits, semester)
   Teaches(ssNo, code)

b. Teachers(ssNo, name, position, department)
   Courses(code, name, credits, semester)
   Teaches(code, ssNo)

c. Teachers(ssNo, name, position, department, coursecode)
   Courses(code, name, credits, semester)

d. Teachers(ssNo, name, position, department)
   Courses(code, name, credits, semester, teacherSsNo)

## Answers

Based on the UML diagram, one teacher can teach at most one course, but one course can have multiple teachers.

- Option a is correct, as the relation Teaches has the ID of the teacher as its key attribute.
- Option b is incorrect, as the key attribute of the relation Teaches is the course code, but this doesn't work, as the same course can have multiple teachers (for each course–teacher-combination we create one tuple).
- Option c is correct, as we have added the information about the course the teacher is teaching to the relation of the teacher (each teacher can have at most one course).
- Option d is incorrect, as we have added the information about the teacher to the relation of the course, which doesn't work since a course can have multiple teachers.

Correct answers are a and c.

## When should we combine two relations?

- Relations that have been created based on a many-one association can be combined. When should we do this?
- Combining relations results in less relations for the schema. This makes the relational schema simpler, which is desirable and more efficient for databases.

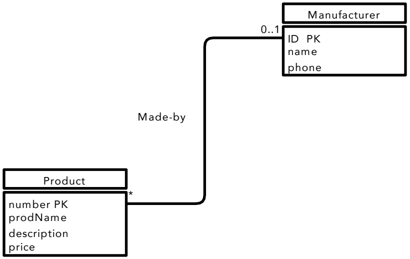
## When should we combine two relations? Part II

- If only a small number of objects of the class are associated with the other class, combining the relations leads to a situation, where many tuples have NULL-values on the attributes inherited through the association.
- In these cases combining the relations is not necessarily a good idea, since:
  - Storing the extra attributes reserves extra space, if almost all of the tuples have NULL values for the attributes in question.
  - If we perform a lot of queries on the information of the association, it might be quicker to do the queries on the relation created for the association, if its size is significantly smaller than the size of the relation created based on the class. (However, this depends on what other relations we need in this query).

## Example 1

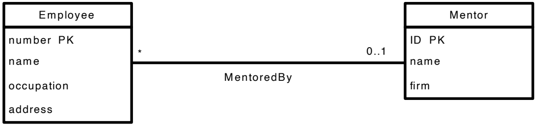
- Almost all products in the database have information on their manufacturer.
- We want to combine the relations *MadeBy* and *Products*, as the database schema becomes more simple and the tuples of the relation *Products* rarely have NULL values for the attribute manufID.

Products(<u>number</u>, name, description, price, ID)



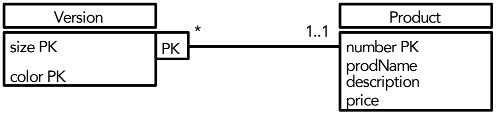
## Example 2

- Let's assume, that the workplace has a mentor program, where some of the employees have been assigned a mentor outside the workplace.
- One employee can have at most one mentor, but more than 90 % of the employees don't have a mentor



- As only a small number of employees have a mentor, we probably want to create a relation *MentoredBy* instead of combining it to the relation *Employees*.

# Attributes of the class not sufficient for a key

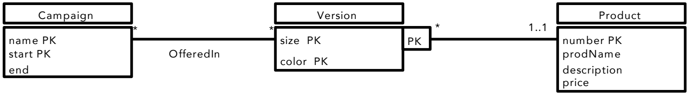


- In addition to the key attributes of the class *Version*, we add the key attributes from the class *Product*.
- We should never create a relation for the association between *Version* and *Product*.
- If the class *Version* is part of some other association, the relation created based on this association should include the key attributes from both of the classes *Version* and *Product*.
- Relations:
  Products(number, name, description, price)
  Versions(prodNo, size, color)

Aalto University
School of Science

## Class *Version* in an Ordinary Association

- Let’s assume that in the online store some of the versions are only available for temporary campaigns.

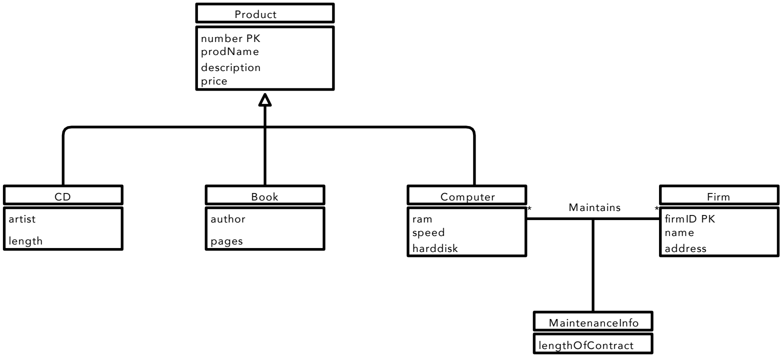


- When defining the relation based on the association *OfferedIn*, include the key attributes from *Versions* and *Campaigns*, but also the key attribute of the relation *Products*, since it’s needed for identification of the versions.
- Relations:
  Products(<u>number</u>, name, description, price)
  Versions(<u>prodNo</u>, <u>size</u>, <u>color</u>)
  Campaigns(<u>name</u>, <u>start</u>, end)
  OfferedIn(<u>prodNo</u>, <u>size</u>, <u>color</u>, <u>campaignName</u>, <u>start</u>)

Aalto University
School of Science

# Inheritance Hierarchy with Relations

How can we represent a superclass and its subclasses as relations?



Aalto University
School of Science

# Inheritance Hierarchy with Relations

- We can use three different approaches:
  1. Create a relation for the superclass on each of the subclasses. The relations of the subclasses should inherit the key attributes of the superclass, and their own attributes.
  2. Create a relation for the superclass and each of the subclasses. The relations of the subclasses inherit all attributes from the superclass, and their own attributes.
  3. Create one relation for the whole inheritance hierarchy. Any attributes not par of certain objects, will have NULL as their value.
- We use the usual approaches for creating the relations for associations concerning the superclass or subclasses.

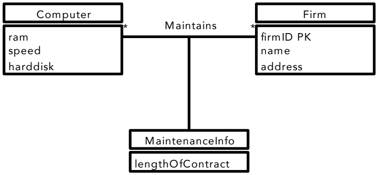
## Example 1: Relations of subclasses have their own attributes and only keys of the superclass

- We create four different relations for the classes in the diagram on slide 24:

  Products(<u>number</u>, name, description, price)
  CDs(<u>number</u>, artist, length)
  Books(<u>number</u>, author, pages)
  Computers(<u>number</u>, speed, ram, harddisk)

- In addition we need relations to represent the association *Maintains* and the class *Firm*:

  Maintains(<u>number</u>, <u>firmID</u>, lengthOfContract)
  Firms(<u>firmID</u>, name, address)



Each object of the subclass will have a tuple both in the relation of the subclass and the relation of the superclass.

## Example 2: Relations of subclasses have their own attributes and all attributes of superclass

- Now the relations of the subclasses have more attributes:

  Products(<u>number</u>, name, description, price)

  CDs(<u>number</u>, name, description, price, artist, length)

  Books(<u>number</u>, name, description, price, author, pages)

  Computers(<u>number</u>, name, description, price, speed, ram, harddisk)

- The relations for association remain the same:

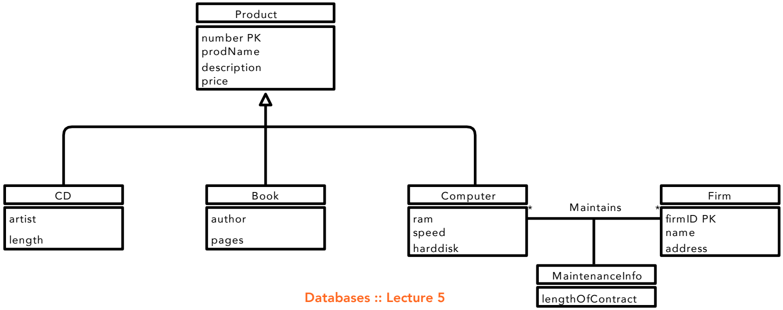
  Maintains(<u>number</u>, <u>firmID</u>, lengthOfContract)

  Firms(<u>firmID</u>, name, address)

Aalto University
School of Science

## Example 2: Relations of subclasses have their own attributes and all attributes of superclass

- Now each object represented by some subclass is in the database *only once*. In the relation *Products*, we have only those objects, that don't belong to any of the subclasses.
- If all of the tuples belong to some of the subclasses, the relation *Products* will not be needed.



## Example 3: Only one relation

- Let’s make the superclass and all the subclasses only one common relation.
- In this example we would make only one relation to represent all the products:

  Products(number, name, description, price, artist, length,
  author, pages, speed, ram, harddisk)

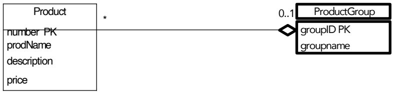
  For example, products that are not books, and the values of attributes author and pages would be NULL.
- Also, we would keep these relations:

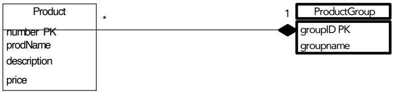
  Maintains(number, firmID, lengthOfContract)
  Firms(firmID, name, address)

## Comparison of Approaches

- In queries concerning all products, it is usually convenient if there are not too many different relations.
- In queries concerning for example only books, it is convenient if books have their own relation.
- If all tuples belong to some subclass and the superclass doesn’t have it’s own many-to-many or one-to-many associations, where the superclass is "on one-side" of the association, then approach 2 is usually the most reasonable. In other cases approach 1 is usually better.
- Approach 3 is reasonable mainly when there are a lot of objects that belong to several subclass at the same time (not always permitted), for example an audio book in CD.

# Aggregation and Composition





- Aggregation and composition are types of many-to-one associations. So instead of constructing a new relation, add the the relation for the class on the non-diamond end, the key attribute(s) of the class on the diamond end.
- Products(<u>number</u>, prodName, description, price, groupID)
  ProductGroups(<u>groupID</u>, groupname)
- In the case of aggregation, these attributes can be NULL.
- Notice: in prior examples the relation *Products* does not have attribute manufID, because here the relations are made only based on the information visible in the corresponding UML diagram and they do not include information outside the diagram.

Aalto University
School of Science

## Functional Dependencies and Normalization

- What relations should be defined in the database and what attributes they should have?
- Consider principles like Normalization to improve design by using constraints that apply to relations.
- Functional Dependencies (FD) are the most common constraint used in relations.
- For example, an FD could make an assertion that if two tuples have the same values for the title and year of movie, then they would also have the same studio.
- Decomposition can be used to split relations into smaller schemas by examining functional dependencies among attributes in the relation.

## Example

Here are two different ways to represent products and their manufacturers:

1. All the information is stored in the same relation:

   Products1(number, prodName, description, price, manufID, manufName, phone)

2. Information can be split into two relations:

   Products(number, prodName, description, price, manufID)

   Manufacturers(ID, manufName, phone)

Aalto University
School of Science

## Example: Using a Single Relation

For example, many instances of the relations could be:

Relation Products1

| *number* | *prodName* | *description* | *price* | *manufID* | *manufName* | *phone* |
|---|---|---|---|---|---|---|
| T-33441 | Galaxy A5 | cellphone | 250.0 | S123 | Samsung | 020-7300 |
| S-65221 | Brasserie 24 | pan | 33.50 | F542 | Fiskars | 020-43910 |
| T-33442 | NX 300 Smart | camera | 399.0 | S123 | Samsung | 020-7300 |
| T-33455 | Cyber-shot | camera | 463.0 | L711 | Sony | 020-6500 |
| R-43118 | Samsung LT 24 | TV | 169.0 | S123 | Samsung | 020-7300 |
| R-27113 | Sony 32 KDL | TV | 347.0 | L711 | Sony | 020-6500 |

## Example: Splitting into Two Relations

### Relation Products

| *number* | *prodName* | *description* | *price* | *manufID* |
|---|---|---|---|---|
| T-33441 | Galaxy A5 | cellphone | 250.0 | S123 |
| S-65221 | Brasserie 24 | pan | 33.50 | F542 |
| T-33442 | NX 300 Smart | camera | 399.0 | S123 |
| T-33455 | Cyber-shot | camera | 463.0 | L711 |
| R-43118 | Samsung LT 24 | TV | 169.0 | S123 |
| R-27113 | Sony 32 KDL | TV | 347.0 | L711 |

### Relation Manufactures

| *ID* | *manufName* | *phone* |
|---|---|---|
| S123 | Samsung | 020-7300 |
| L711 | Sony | 020-6500 |
| F542 | Fiskars | 020-43910 |

Aalto University
School of Science

## Example: Differences in Database Schemas

- In the first database schema the information about manufacturers' names and phone numbers are stored several times, while in the second schema only once.
- For example, if the phone number of manufacturer changes, the first schema has to be updated in several tuples, while in the second on only one tuple.
- In the first schema, if the product S-65221 is removed, the information about the manufacturer *Fiskars* is lost. In the second schema the manufacturer information is retained even if this product information is removed.
- The second schema is clearly better than the first.

## Anomalies

- Anomalies: abnormalities in the database behavior caused by a poorly structured database.

- Common Anomalies:
    - **Redundancy:** information that may be repeated unnecessarily in several tuples in the same relation (or table).
    - **Update Anomalies:** if the same information is stored in several tuples, all the changes have to be made for every instance of the information.
    - **Deletion Anomalies:** removing tuples may have side effects. For example, if information of the products and their manufacturers is stored in the same relation, removing the product may at the same time remove the information about the manufacturer's name and phone number.

## Anomalies

| *title* | *year* | *length* | *genre* | *studioName* | *starName* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | SciFi | Fox | Carrie Fisher |
| Star Wars | 1977 | 124 | SciFi | Fox | Mark Hamill |
| Star Wars | 1977 | 124 | SciFi | Fox | Harrison Ford |
| Gone With the Wind | 1939 | 231 | drama | MGM | Vivien Leigh |
| Wayne's World | 1992 | 95 | comedy | Paramount | Dana Carvey |
| Wayne's World | 1992 | 95 | comedy | Paramount | Mike Meyers |

Figure 6: The relation `Movies1` exhibiting anomalies

- **Redundancy:** same values for length and genre of movies.
- **Update Anomalies:** updating length of movie across all tuples in the table.
- **Deletion Anomalies:** removing an actor for a movie with only one tuple in the table would remove all information, unless made NULL (which may be another constraint).

Note: If an attribute is used to join two relations together it is not a redundancy (e.g. studio ID key in tables Movies & Studios).