course:
Database Systems (NDBI025)
SS2011/12

lecture 2:

Logical database models, relational model

doc. RNDr. Tomáš Skopal, Ph.D.
Mgr. Martin Nečaský, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

Today’s lecture outline

- introduction to logical database models
  - object, (object-)relational model
  - other (XML, RDF)
- (object-)relational model in detail
  - introduction
  - UML schema conversion

Three layers of database modeling

abstraction

- conceptual layer
  - models a part of the **"structured"** real world relevant for applications built on top of your database
    - real world part = **real-world entities and relationships between them**
  - different conceptual models (e.g. ER, UML)
- **logical layer**
  - specifies how conceptual components are represented in logical machine interpretable data structures
  - different logical models (e.g. object, relational, object-relational, XML, graph, etc.)
- physical model
  - specifies how logical database structures are implemented in a specific technical environment
  - data files, index structures (e.g. B+ trees), etc.

implementation

More on Logical Layer

- specifies how conceptual entities are represented in data structures used by your system for various purposes, e.g.,
  - for data storage and access
    - graph of objects
    - tables in (object-)relational database
    - XML schemas in native XML database
  - for data exchange
    - XML schemas for system-to-system data exchange
  - for data publication on the Web
    - XML schemas for data publication
    - RDF schemas/OWL ontologies for publication on the Web of Linked Data in a machine-readable form
  - other

More on Logical Layer

- → different **logical models** for data representation
  - for data storage and access
    - object model
    - relational model (or object-relational model)
    - XML model
  - for data exchange
    - XML model
  - for data publication on the Web
    - XML model
    - RDF model
  - other
- modern software systems have to cope with many of these different logical models

More on Logical Layer

- problem 1: Designers need to **choose the right logical model** with respect to the nature of the prospective access to data.

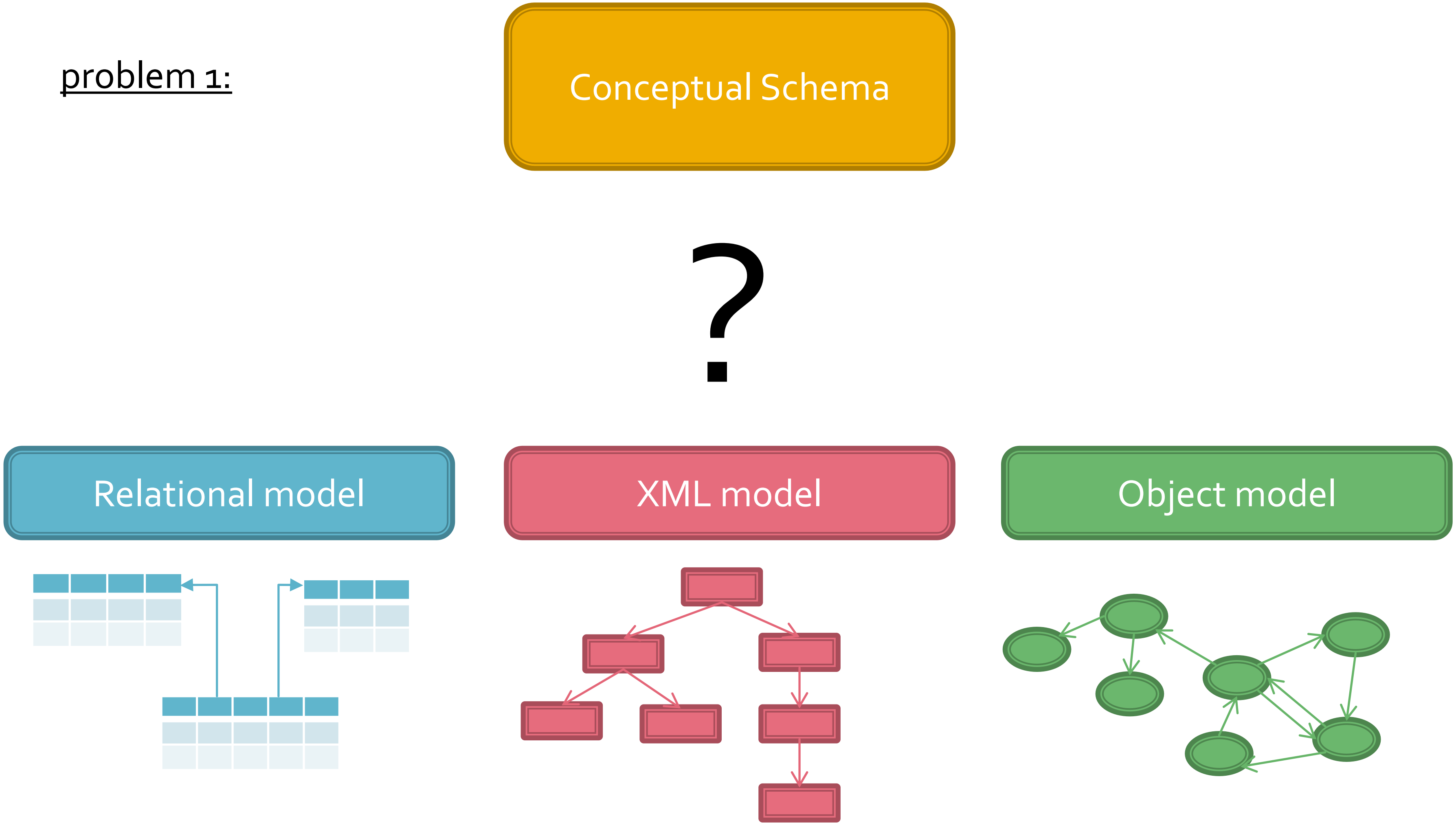
  or

- problem 2: Designers need to design **several logical schemas** for different models applied in the system.
  - logical schemas can be derived automatically or semi-automatically from a single conceptual schema

  → ***Model-Driven Development***

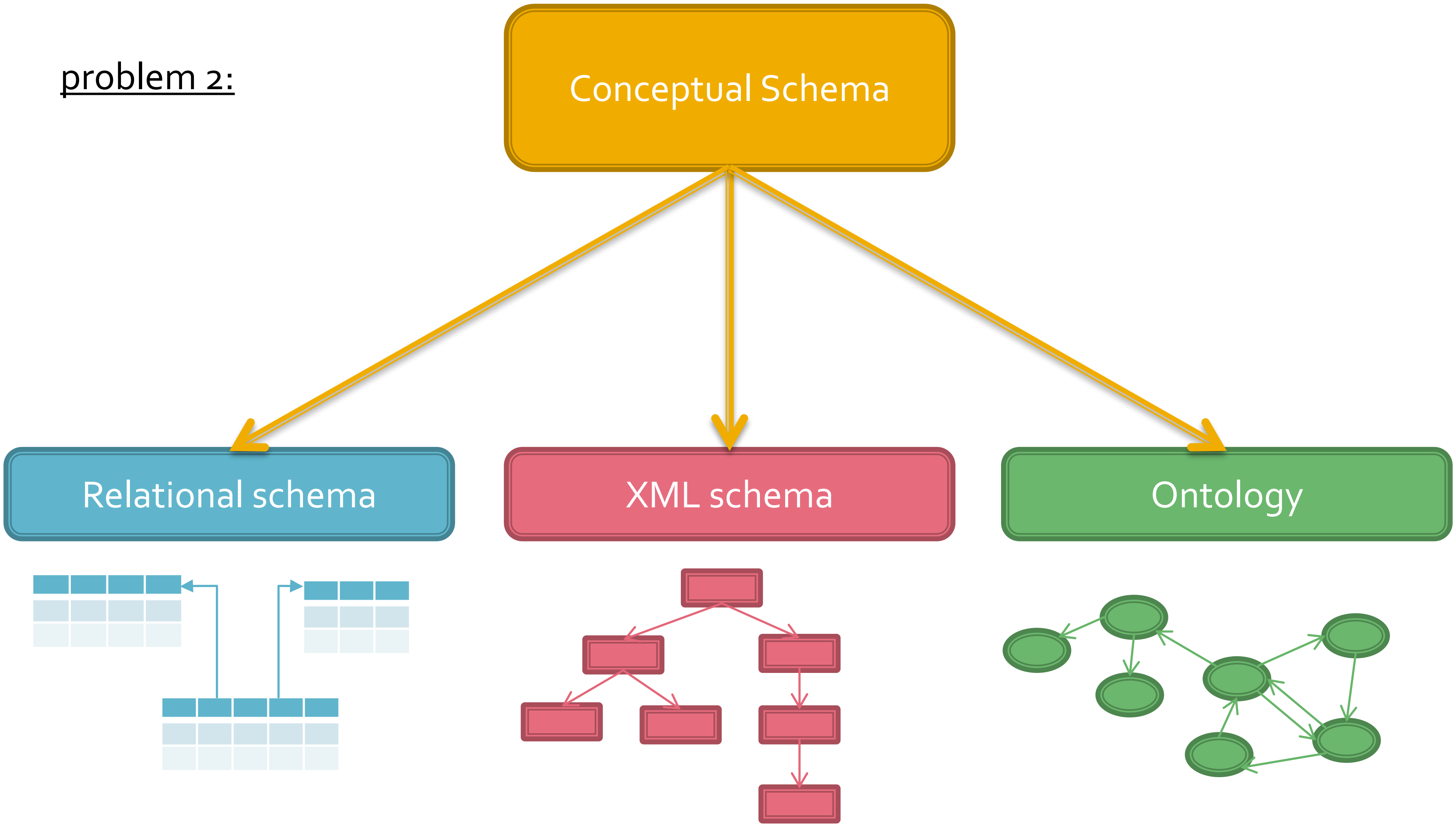
More on Logical Layer

problem 1:



More on Logical Layer

problem 2:

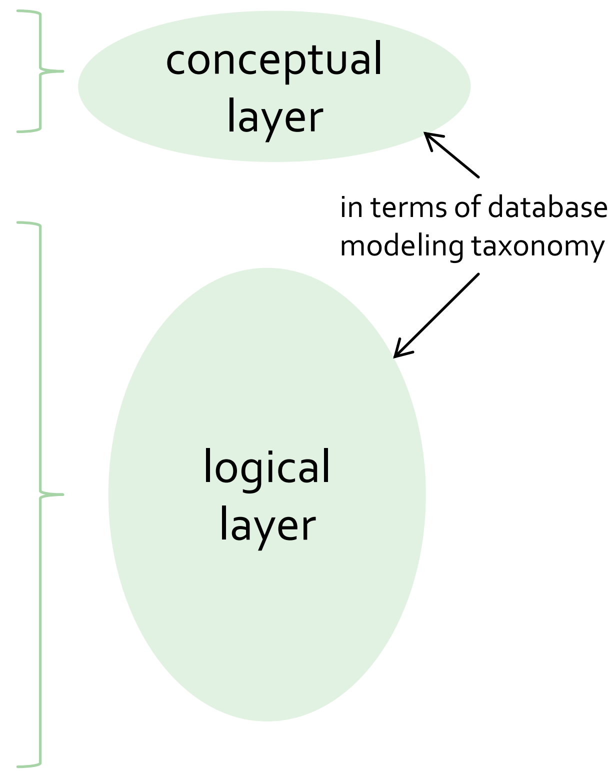


Model-Driven Development (MDD)

- an approach to software development
- theoretically, it enables to create executable schemas instead of executable code (= classical approach)
  - more precisely, schemas are automatically converted to executable code
  - recent idea, not applicable for software development in practice today
    - lack of tool support
- BUT, we will show you how it can be profitably applied for designing logical data schemas
  - allows to deal with different logical models which we need to apply in our system

MDD for Logical Database Schemas

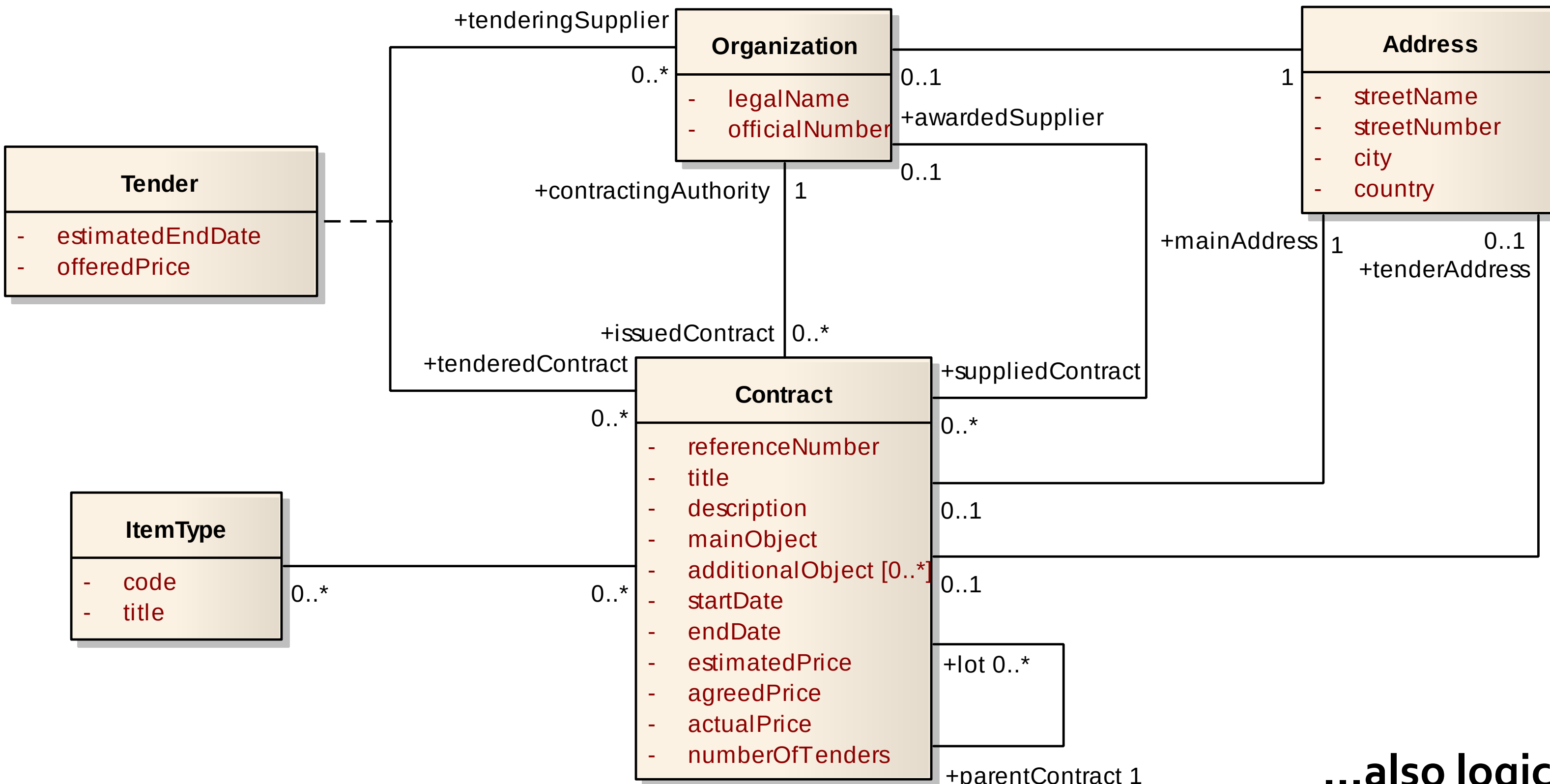
- considers UML as a general modeling language
- distinguishes several levels of abstraction of logical data schemas
  - Platform-Independent Level
    - hides particular platform-specific details
  - Platform-Specific Level
    - mapping of the conceptual schema (or its part) to a given logical model
    - adds platform-specific details to the conceptual schema
  - Code Level
    - schema expression in a selected machine-interpretable logical language
    - for us, e.g., SQL, XML Schema, OWL



Practical Example 1

- Information System for Public Procurement
  - http://www.isvzus.cz (in Czech only ☹)
  - current project of Czech eGov
  - aims at increasing transparency in Public Procurement in Czech Rep.
- system has to deal with many logical models:
  - relational data model for data storage
  - XML data model for exchanging data with information systems of public authorities who issue public contracts
  - *RDF data model for publishing data on the Web of Linked Data in a machine-readable form*
    - *this is not happening in these days (March 2012) but we, Charles University, OpenData.cz, are working hard to show how to do it ☺*

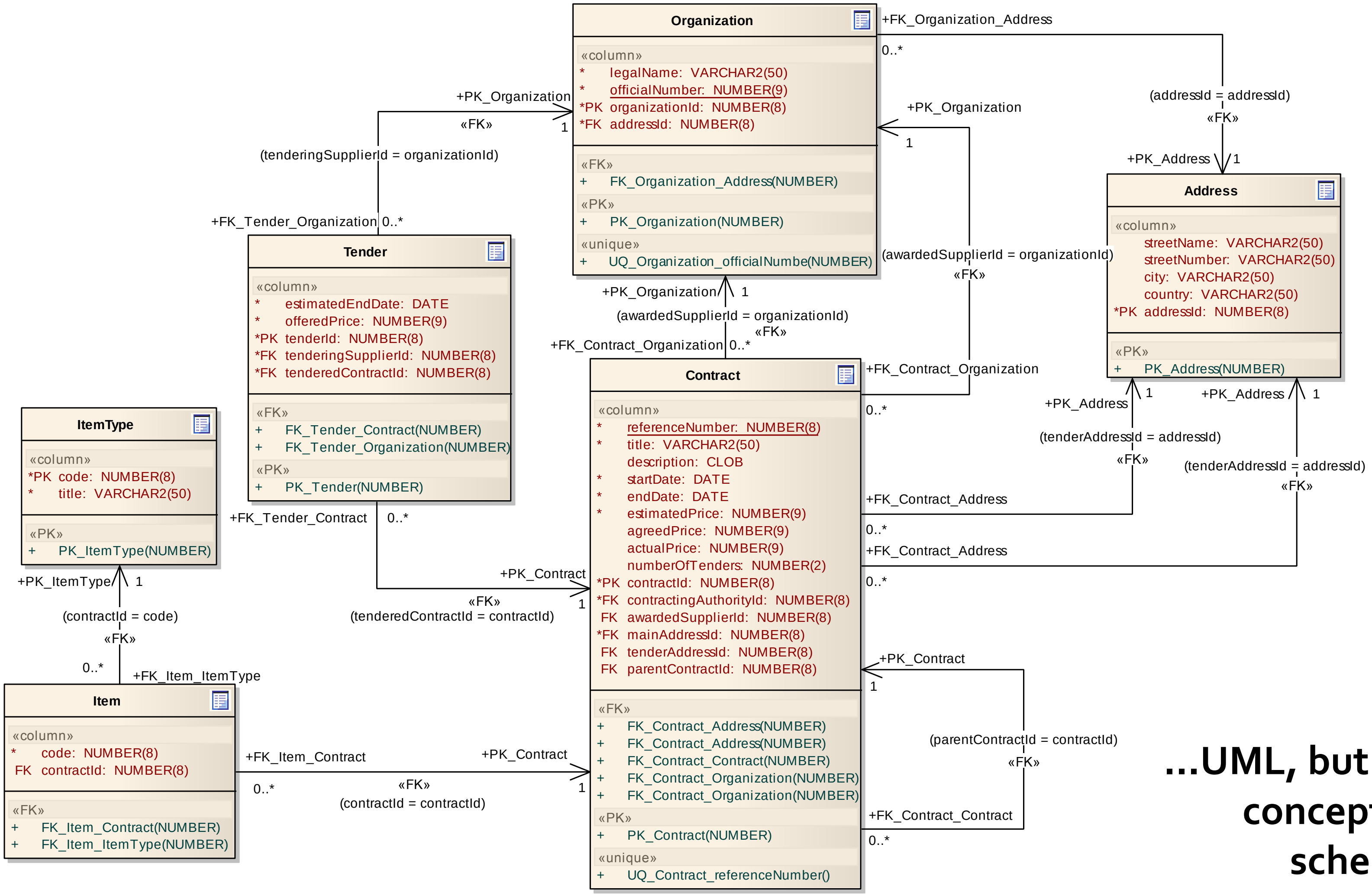
Practical Example
Platform-Independent Level (conceptual model)



...also logical object schema?

Practical Example 1

Platform-Specific Level (relational model)



...UML, but not conceptual schema!

Practical Example 1
Platform-Specific Level (relational model)

- notes to previous UML diagram
  - it is UML class diagram
  - but enhanced with features for modeling schema in **(object-)relational model**
  - enhancement = stereotypes

- stereotype = UML construct assigned to a basic construct (class, attribute, association) with a specific semantics, e.g.,
  - <<table>> specifies that a class models a table
  - <<PK>> specifies that an attribute models a primary key
  - <<FK>> specifies that an attribute/association models a foreign key
  - etc.

Practical Example 1

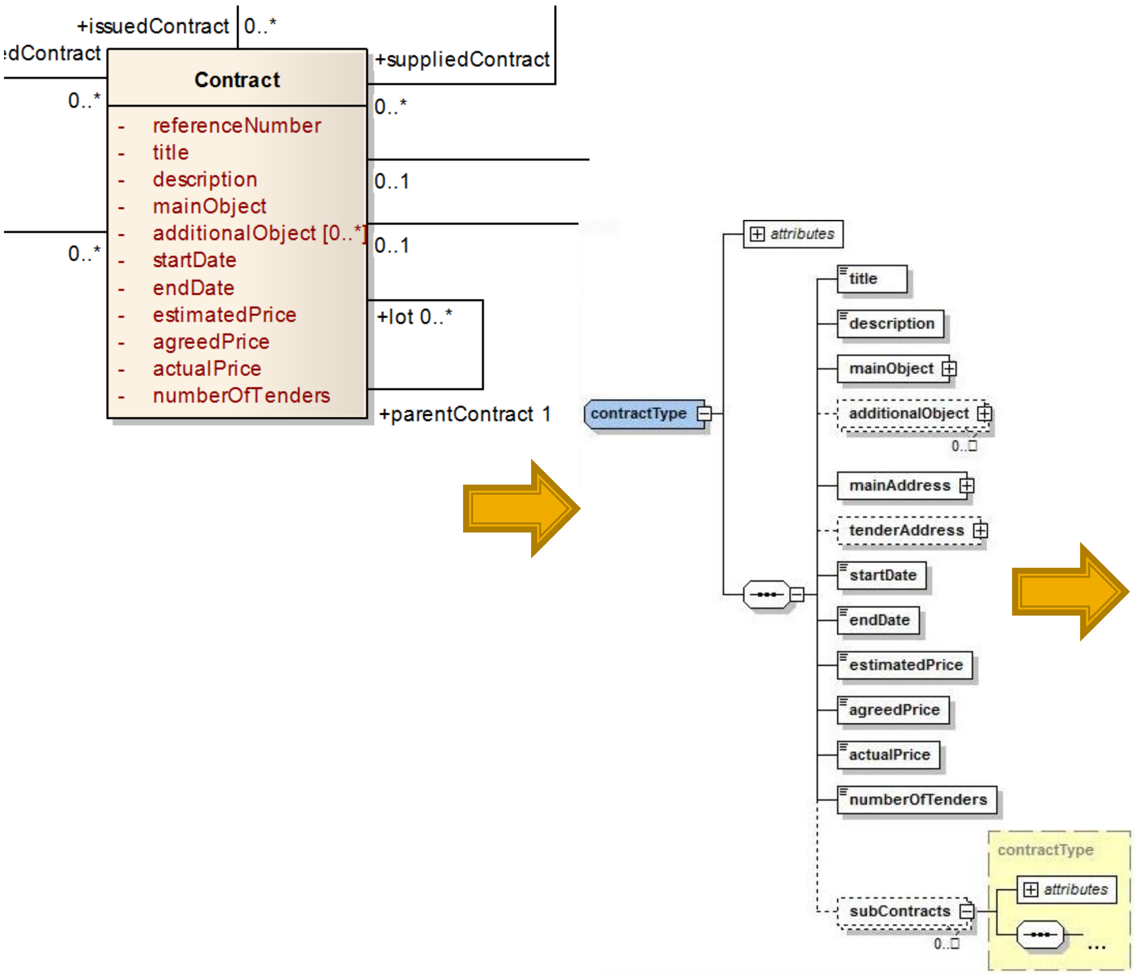
Code Level (SQL)

```
CREATE TABLE Contract (
             referenceNumber          NUMBER(8) NOT NULL,
             title                    VARCHAR2(50) NOT NULL,
             description              CLOB,
             startDate                DATE NOT NULL,
             endDate                  DATE NOT NULL,
             estimatedPrice           NUMBER(9) NOT NULL,
             agreedPrice              NUMBER(9),
             actualPrice              NUMBER(9),
             numberOfTenders          NUMBER(2),
             contractId               NUMBER(8) NOT NULL,
             contractingAuthorityId   NUMBER(8) NOT NULL,
             awardedSupplierId        NUMBER(8),
             mainAddressId            NUMBER(8) NOT NULL,
             tenderAddress            NUMBER(8),
             parentContractId         NUMBER(8));

ALTER TABLE Contract ADD CONSTRAINT PK_Contract PRIMARY KEY (contractId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Address FOREIGN KEY (mainAddressId) REFERENCES Address (addressId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Address FOREIGN KEY (tenderAddress) REFERENCES Address (addressId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Contract FOREIGN KEY (parentContractId) REFERENCES Contract (contractId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Organization FOREIGN KEY (contractingAuthorityId) REFERENCES Organization (organizationId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Organization FOREIGN KEY (awardedSupplierId) REFERENCES Organization (organizationId);

CREATE TABLE Organization(
             legalName        VARCHAR2(50) NOT NULL,
             officialNumber   NUMBER(9) NOT NULL,
             organizationId   NUMBER(8) NOT NULL,
             addressId        NUMBER(8) NOT NULL);

ALTER TABLE Organization ADD CONSTRAINT PK_Organization PRIMARY KEY (organizationId);
ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Address FOREIGN KEY (addressId) REFERENCES Address (addressId);

...
```

Practical Example 1
Code Level (SQL)

- notes to previous SQL code
  - generated fully automatically from the platform-specific diagram by a CASE tool
  - possibility to automatically generate the code requires all necessary information present in the platform-specific diagram
  - however, it is easier and less error prone to specify the information at the platform-specific level (CASE tool can detect errors and helps with the specification)

Practical Example 2
XML




```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2012 sp1 (http://www.altova.com) by IM (Charles
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:complexType name="addressType">
                <xs:sequence>
                        <xs:element name="streetName"/>
                        <xs:element name="streetNumber"/>
                        <xs:element name="city"/>
                        <xs:element name="country"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="objectType">
                <xs:sequence>
                        <xs:element name="code" type="xs:int"/>
                        <xs:element name="title" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="contractType">
                <xs:sequence>
                        <xs:element name="title" type="xs:string"/>
                        <xs:element name="description" type="xs:string"/>
                        <xs:element name="mainObject" type="objectType"/>
                        <xs:element name="additionalObject" type="objectT
                        <xs:element name="mainAddress" type="addressType"
                        <xs:element name="tenderAddress" type="addressTyp
                        <xs:element name="startDate" type="xs:date"/>
                        <xs:element name="endDate" type="xs:date"/>
                        <xs:element name="estimatedPrice" type="xs:float"
                        <xs:element name="agreedPrice" type="xs:float"/>
                        <xs:element name="actualPrice" type="xs:float"/>
                        <xs:element name="numberOfTenders" type="xs:int"/
                        <xs:element name="subContracts" type="contractTyp
                </xs:sequence>
                <xs:attribute name="referenceNumber" type="xs:int"/>
        </xs:complexType>
        <xs:complexType name="tenderedContractType">
```

Object vs. relational model

- object model
  - data stored as graph of objects (in terms of OOP)
  - suitable for individual navigational access to entities, follows the conceptual model
  - not suitable for "batch operations" (data-intensive applications)
  - comfort support in software development platforms, e.g., Hibernate in Java or ADO.NET Entity Framework
    - the goal:
      the programmer need not to care of her/his object hierarchy persistency
    - application data is loaded/stored from/to the database as needed, the data exist regardless of the application runtime

Object vs. relational model

- physical implementation of object model
  - native object DBMS (e.g., Caché)
  - object-relational mapping
    - object schema must be mapped to relational schemas
      - used also by Hibernate, etc.
    - brings overhead
  - generally, storage/serialization of general object hierarchy (graph data) is problematic
    - suitable only for enterprise applications with most of the logic not based on data processing
      - simply, structurally complex but low volume data
      - for data-intensive applications with „flat" data the relational model is crucial

Object vs. relational model

- relational model
  - data stored in flat tables
    - attribute values are simple data types
  - suitable for data-intensive “batch operations”
  - not suitable for individual navigational access to entities (many joins)
- object-relational model
  - relational model enriched by object elements
    - attributes general classes (not only simple types)
    - methods/functions defined on attribute types

Relational model

- founded by E.F Codd in article „*A relational model of data for large shared data banks*“, Communications of ACM, 1970
  - much sooner than UML and even ER modeling!

- model for storage of objects and their associations in **tables (relations)**
- object or association is represented by one **row** in the table (member of relation)
- an **attribute** of an object/association is represented by a **column** in the table
- **table schema (relation schema)** – description of the table structure (everything except the data, i.e., metadata)
  - $S(A_1:T_1, A_2:T_2, \ldots)$ – S the schema name, A_i are attributes and T_i their types
- **schema of relational database** – set of relation schemas (+ other stuff, like integrity constraints, etc.)

Relational model

- basic integrity constraints
  - two identical rows cannot exist (unique identification)
  - the attribute domain is given by its type (simple typed attributes)
  - the table is complete, there are no “holes”, i.e., every cell in the table has its value (value NULL could be included as a „metavalue“)
- every row is identified by one or more attributes of the table
  - called a **superkey** of a table (special case is the entire row)
  - superkey with minimal number of attributes is called a **key** (multiple keys allowed)
- foreign key (“inter-table” integrity constraint)
  - is a group of attributes that exists in another (referred) table, where the it is a (super)key
  - consequence: usually it is not superkey in the referring table since there may exist duplicate values in that attributes

Notation

- notation

key

other key

Relation1(**keyAttr**, **otherKeyAttr**, normalAttr)

composite key

Relation2(**keyAttrPart1, keyAttrPart2**)

Relation3(**keyAttr**, normalAttr, **foreignKeyAttr**),

foreignKeyAttr ⊆ Relation1.**keyAttr**

foreign key

Relational model

Product(<u>Name</u>: string, Producer:string, Price: float, Availability: int), Product.Producer $\subseteq$ Producer.Name

Name	**Producer**	Price	Availability
Mouse	Dell	5,80	100000
Windows	Microsoft	12,50	NULL
Printer	Toshiba	325,-	15000
Phone	Nokia	135,-	32000
Laptop	Dell	500	9000
Bing	Microsoft	0	NULL

key

foreign key
(key in table Producer)

Producer(<u>Name</u>: string, Address:string, Debt: bool)

Name	**Address**	**Debt**
Dell	USA	YES
Microsoft	USA	NO
Samsung	Korea	NO
Nokia	Norway	YES
E.ON	Germany	NO
Toshiba	Japan	NO

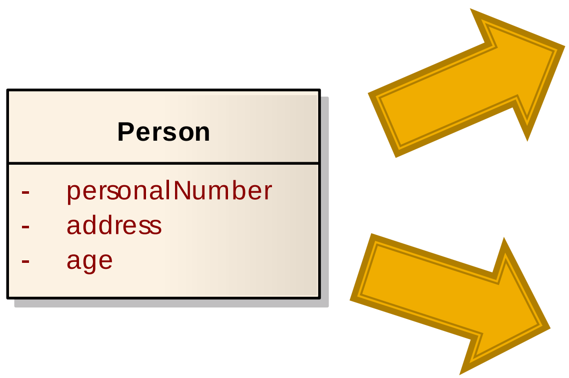
key

Translation of conceptual schema

- designing relational database based on a conceptual schema
- UML or ER diagram consists of
  - classes – objects directly stored in table rows
  - associations – must be also stored in tables
    - either separate, or together with the classes (depends on cardinalities)

Translation of Classes

- class translated to separate table
  - in some cases more classes can be translated to a single table (see next slides)



Person(**personalNumber**, address, age)

- key is an attribute derived from an attribute in the conceptual schema
  - from identifier (ER) or stereotype/OCL (UML)

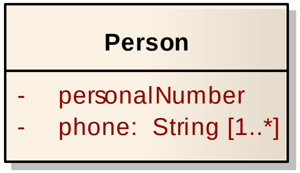
Person(**personID**, **personalNumber**, address, age)

- key is also an artificial attribute with no correspondence in real world
  - automatically generated values

- artificial identifiers are preferred candidates for primary keys!
- **NOTE: we will use only artificial identifiers in the following text**

Translation of Multiplied Attributes

- multiplied attributes must have separate table



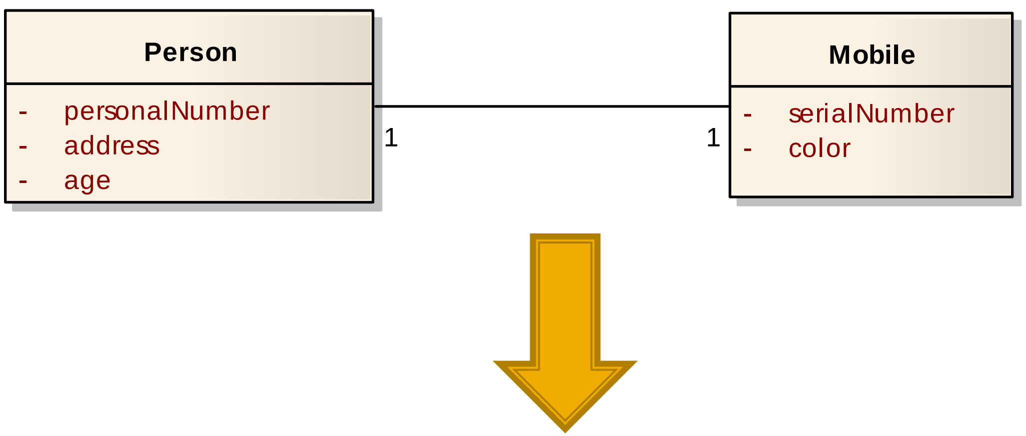
Person(**personID**, personalNumber)

Phone(**phoneID**, phone, **personID**)

personID ⊆ Person.**personID**

Translation of Associations
Multiplicity (1,1):(1,1)

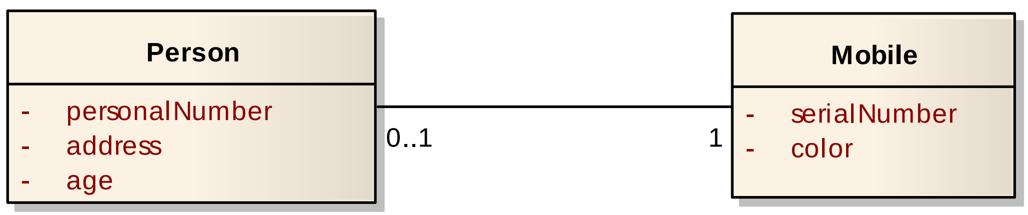
- single table, key from one class or both (two keys)



Person(**<u>personID</u>**, personalNumber, address, age, **<u>mobileID</u>**, serialNumber, color)

Translation of Associations
Multiplicity (1,1):(0,1)

- two tables T1 (0,1) and T2 (1,1)
  - T1 independent of T2
  - T2 contains a foreign key to T1



Person(<u>**personID**</u>, personalNumber, address, age, **mobileID**)

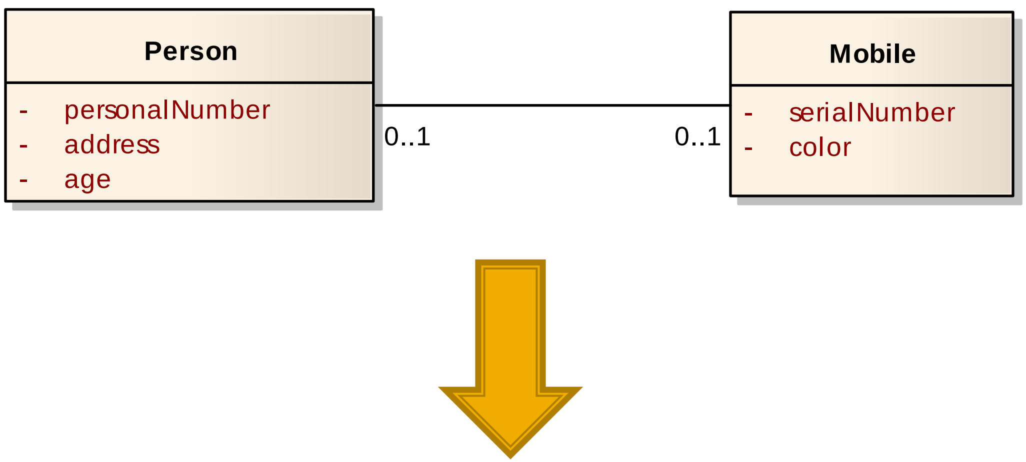
mobileID ⊆ Mobile.<u>**mobileID**</u>

Mobile(<u>**mobileID**</u>, serialNumber, color)

Why not only 1 table?
Because a mobile can exist independently of a person. Having it in a single table with persons would lead to empty "person" fields for mobiles without a person.

Translation of Associations
Multiplicity (0,1):(0,1)

- three tables T1, T2, and T3
  - T3 represents the association



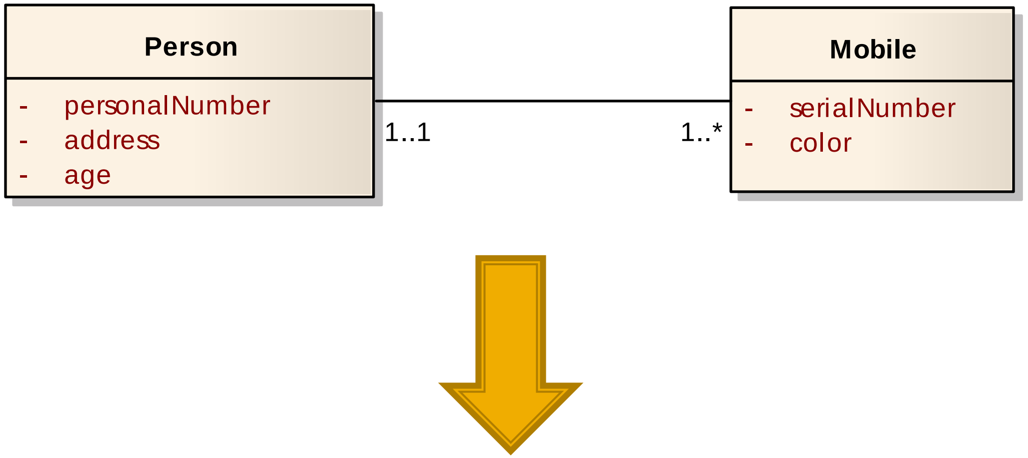
Person(<u>**personID**</u>, personalNumber, address, age)

Mobile(<u>**mobileID**</u>, serialNumber, color)

Owns(<u>**personID**</u>, <u>**mobileID**</u>), **personID** ⊆ Person.<u>**personID**</u>, **mobileID** ⊆ Mobile.<u>**mobileID**</u>

Translation of Associations
Multiplicity (1,n)/(0,n):(1,1)

- two tables T1 and T2, similarly to (0,1):(1,1)
  - T1 independent of T2

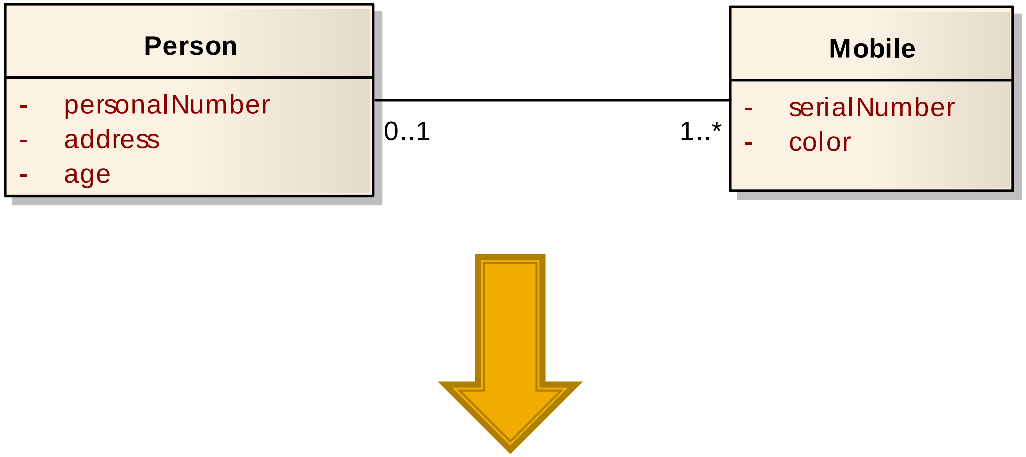


Person(<u>**personID**</u>, personalNumber, address, age)

Mobile(<u>**mobileID**</u>, serialNumber, color, **personID**) , **personID** ⊆ Person.<u>**personID**</u>

Translation of Associations
Multiplicity (1,n)/(0,n):(0,1)

- three tables T1, T2, and T3 similarly to (0,1):(0,1)



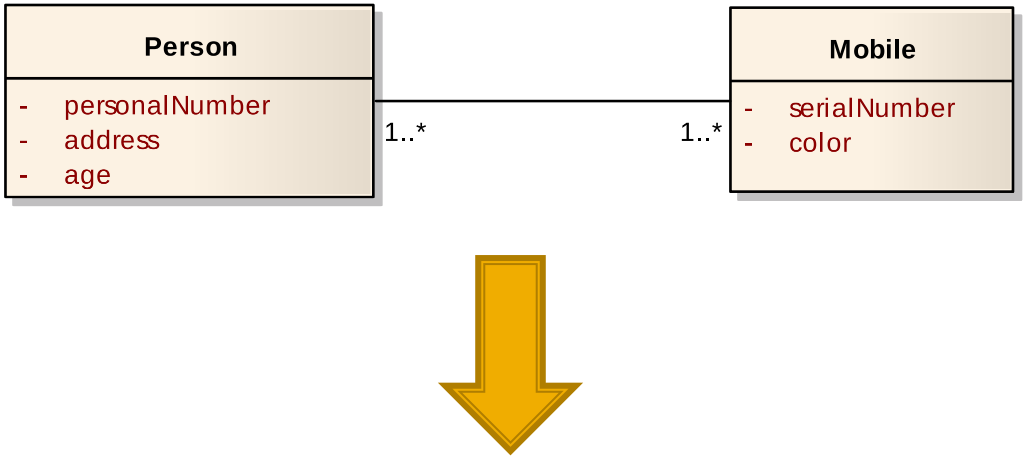
Person(**<u>personID</u>**, personalNumber, address, age)

Mobile(**<u>mobileID</u>**, serialNumber, color)

Owns(**personID**, **<u>mobileID</u>**), **personID** ⊆ Person.**<u>personID</u>**, **mobileID** ⊆ Mobile.**<u>mobileID</u>**

Translation of Associations
Multiplicity (1,n)/(0,n):(1,n)/(0,n)

- three tables T1, T2, and T3
  - T3 is association table, its key is **combination** of keys of T1 and T2**(!)**



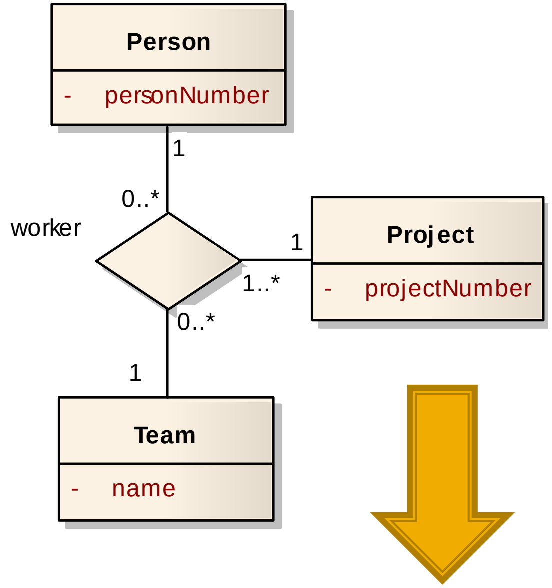
Person(**<u>personID</u>**, personalNumber, address, age)

Mobile(**<u>mobileID</u>**, serialNumber, color)

Owns(**<u>personID, mobileID</u>**) , **personID** ⊆ Person.**<u>personID</u>**, **mobileID** ⊆ Mobile.**<u>mobileID</u>**

Translation of Associations
N-ary Associations

- N tables + 1 association table



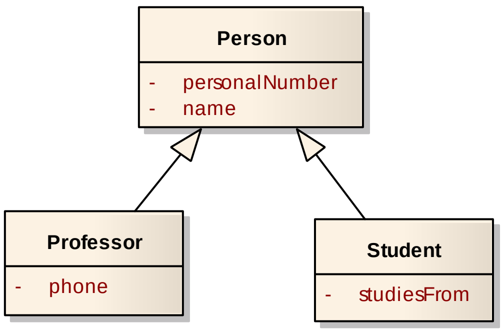
Less tables?
Yes, similarly to binary associations when ..1 appears instead of ...*

Person(<u>**personID**</u>, personNumber), Team(<u>**teamID**</u>, name), Project(**projectID**, projectNumber)

Worker(<u>**personID, teamID, projectID**</u>) ,

personID ⊆ Person.<u>**personID**</u>, **teamID** ⊆ Team.<u>**teamID**</u> , **projectID** ⊆ Project.<u>**projectID**</u>

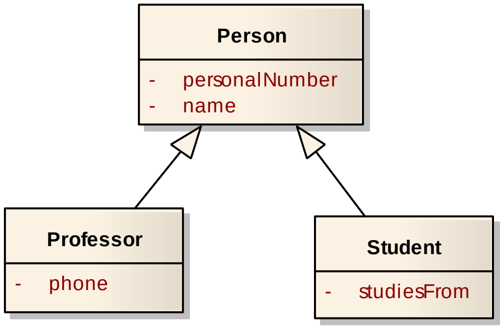
Translation of ISA hierarchy



Person(<u>**personID**</u>, personalNumber, name)
Professor(<u>**personID**</u>, phone), **personID** ⊆ Person.<u>**personID**</u>
Student(<u>**personID**</u>, personalNumber, name, studiesFrom), **personID** ⊆ Person.<u>**personID**</u>

- general solution applicable in any case
- table for each type, each has
- pros:
  - flexibility (e.g., when adding new attributes to Person, only table Person needs to be altered)
- cons:
  - joins

Translation of ISA hierarchy



Suitable when overlap constraint is false, i.e.:

$$Professors \cap Students \neq \emptyset$$

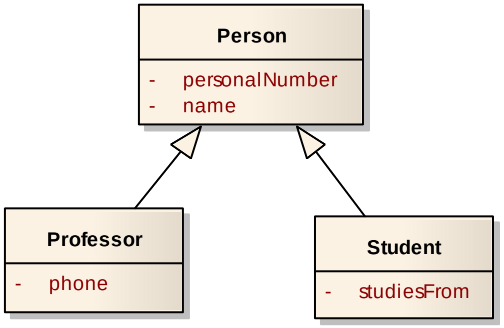
and, ideally, most of persons are both professor and student

$$|Professors \cap Students| \cong |Persons|$$

Person(<u>personID</u>, personalNumber, name, phone, studiesFrom, type)

- all instances stored in single table
- instance type (Person, Professor or Student) is distinguished by artificial type attribute
- pros:
  - one table, no joins
- cons:
  - NULL values when overlap constraint is true, or false but $|Professors \cap Students| \ll |Persons|$ (i.e. most persons are professor or student but not both)

Translation of ISA hierarchy



Suitable when covering constraint is true, i.e.:

$$Professors \cup Students = Persons$$

Unsuitable when overlap constraint is false, i.e.:

$$Professors \cap Students \neq \emptyset$$

Professor(<u>**professorID**</u>, personalNumber, name, phone)
Student(<u>**studentID**</u>, personalNumber, name, studiesFrom)

- tables only for "leaf" types
- pros: no joins
- cons:
  - problem with representing persons which are neither professors nor students (i.e. covering constraint is false)
  - redundancies when a professor can be a student or vice versa (i.e. overlap constraint is false)