

## 4. Beyond E/R and Relational Modelling

### This section's goal:

After completing this chapter, you should be able to

- explain the limitations of traditional E/R-based conceptual models,
- understand consequences of the 1NF restriction for relational databases,
- sketch the basic ideas of so-called “post-relational” data models,
- explain the notion of object-oriented models, and their relationship to the E/R model.

# Object-Orientation

The term “**object-orientation**” has no clear, generally accepted definition within Computer Science.

It is most commonly used for a kind of software engineering approach that encompasses analysis, modelling, and implementation in way that always keeps both,

- **structural** (an encapsulated object state) and
  - **behavioral** (a collection of interface methods) aspects
- of application-specific “units of interest” in mind.

A very diverse terminology has evolved around “objects” in different communities and/or modelling/programming languages.

# Essentials of OO modelling

**Objects** exhibit the following characteristics:

- Each object can be uniquely identified, independent of any values/other objects associated with it (**object identity**).
- Objects are instantiated from **classes**.
- Classes are organized in a **hierarchy** with **inheritance**.
- Objects can only be operated on by means of **methods** (encapsulation).

**N.B.** The basic idea of encapsulation has been developed earlier: *abstract data types (ADTs)* or the *information hiding principle*.

# E/R and object-orientation ( I )

Clearly, (extended) Entity-Relationship models cover the structural part of OO models (identity, classes, inheritance). However, the behavioral part (methods, and to a lesser extent encapsulation) is not known to E/R approaches.

An entity's **attributes** may be viewed as

- the *visible internal structure* of the entity. In that sense, E/R does not provide encapsulation.
- However, they might as well be considered (*observer*) *methods*, that read (parts of) the hidden internal state and return some information about the entity. In that sense, the internal state would be encapsulated.

## E/R and object-orientation (2)

The (mutator) **methods** of a class can not be expressed in E/R models.

- A typical reason is that type-specific methods/functions have not been supported in the target database models.
- With the advent of the 1999 version of SQL (SQL-3), this has changed (cf. “object-relational” databases).

Conversely, **relationships**, which have not been part of early object models, have been introduced into various OO models over time (including the specification of cardinalities).

## More extensions ( I )

Several additional modelling constructs have been proposed in specific E/R and/or OO models. Many of them have their origin in other “semantic data models” or “knowledge representation languages”. For example:

- **Aggregation** (hierarchies) or “part-of” relationships.  
A more complex object can be defined as a composition of subparts. This involves a special kind of existence dependency.

### Aggregation (part-of)

A *table* object is a composition of a *table board* object and four *leg* objects.

Various graphical representations for part-of relationships have been proposed.

## More extensions (2)

- **Association** (hierarchies) or “member-of” relationships. A more complex object can be obtained as a collection of subobject. As before, a certain existence dependency is induced.

### Association (member-of)

A *group* object is a collection of *person* objects.

Again, also graphical representations for association have been developed.

# The Unified Modelling Language (UML)

UML provides a couple of (mostly graphical) languages for the specification, construction, visualization, and documentation of software systems. It is an “industry standard”, widely used in practice, and offers programming language-independent notations for the

- analysis of requirements,
- visualization of the problem,
- design of programs and databases,
- communication with (application) domain experts,
- implementation, and
- documentation of the system.

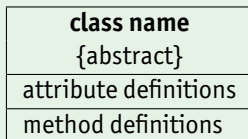
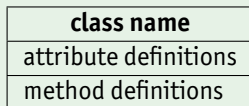
# UML classes

A **class** defines properties (attributes) and behavior (methods) of similar objects. It is described by

- a class name (with optional assertions)
- attribute definitions (with optional types and other constraints)
- method definitions (method name and parameter types).

**Abstract classes** cannot be instantiated.

A **class** is graphically represented by a rectangle (cf. entities)



# Example UML classes ( I )

A class describing 2D geometrical objects with a circular shape

Circle	
radius	{radius > 0}
center	
—Constructor—	
Circle()	
—Other methods—	
circumference():	float
area():	float

← notice the assertion

← return type

←

## Example UML classes (2)

### A class describing appointment objects

<b>Appointment</b>	
start:	DateTime
duration:	TimeIntervall = "01:30h"
description:	String
—Constructor—	
Appointment()	
—Other methods—	
ConflictWith( a:	Appointment ):
end():	DateTime

← attribute's data type

← initial value

← method argument (type)

# UML objects

It may be necessary and useful to talk about individual objects (instances of class). UML provides this capability. Instance or class name are optional (one of them needs to be given). Attribute specifications are optional.

## Single objects can be defined as well

### c1:Circle

```
radius = 1cm  
center = (15,37)
```

### a1:Appointment

```
start = "30-Nov-07;12:00h"  
duration = "01:30h"  
description = "Lunch with Anna"
```

# UML associations

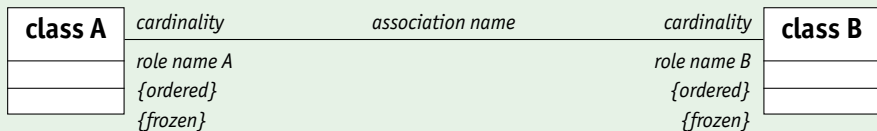
... define a connection between two classes (or objects).

## Definition (UML Associations)

... are named and depicted by a straight line, optionally, they can be

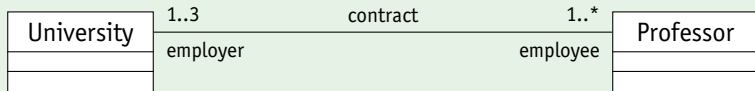
- restricted by cardinalities at both ends (min..max notation; default: 0..1),
- specified as ordered (for max cardinality > 1: list, not set),
- specified as "frozen" (connection cannot be changed, once established),
- specified with a direction (indicated by an arrow head), limiting object-to-object navigation.

## Graphical representation of UML associations (cf. E/R relationships)

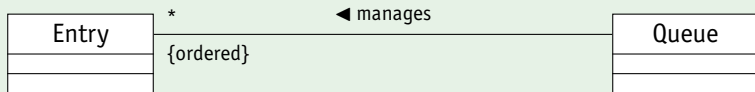


# UML associations: Examples

## ... with role names and cardinalities



## ... with order



## ... with a direction



# Cardinalities

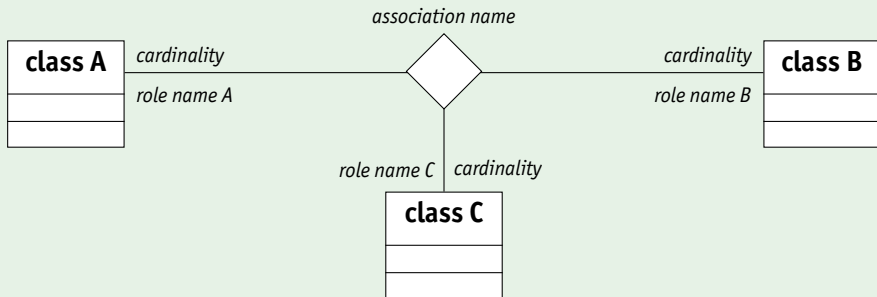
In UML, cardinalities can be specified in a variety of ways, even more flexible than in E/R models, for instance:

1	exactly one
0, 1	zero or one
0..4	zero to four (incl.)
3, 7	exactly three or seven
0..*	arbitrarily many
*	arbitrarily many
1..*	one or more
0..3, 7, 9..*	between zero and three, or seven, or at least nine

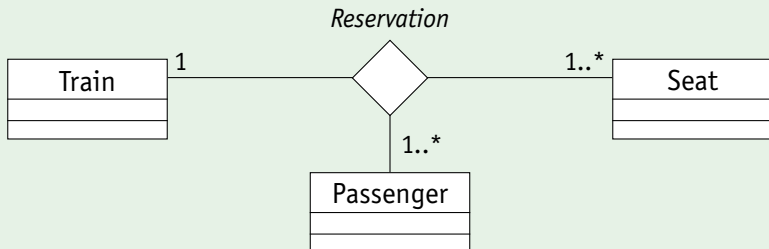
# *n*-ary associations

(Like in E/R), associations might be defined between more than two classes/objects. In that case, they are graphically represented as diamonds, like in E/R diagrams.

## Example (Ternary association)

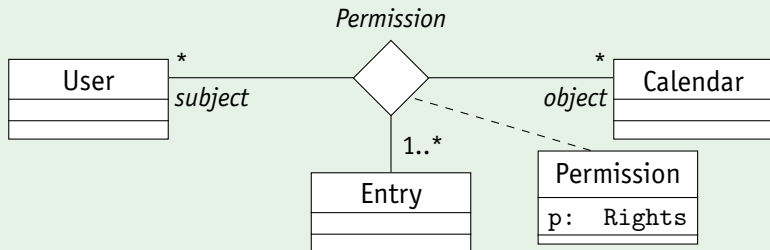


# Examples ( I )



What exactly do the cardinalities mean here?

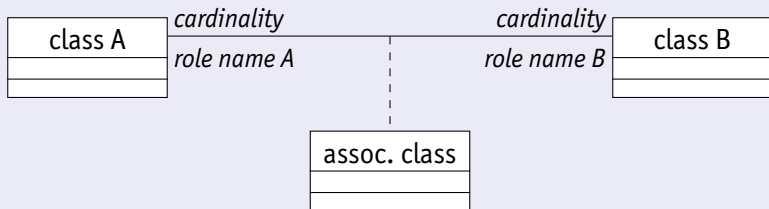
## Examples (2)



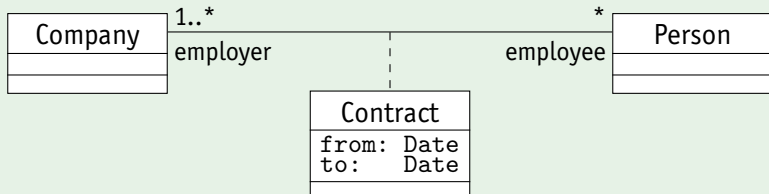
If an association shall be further described by means of attributes and/or methods, an **association class** is used (cf. some E/R models). Association classes may themselves be involved in associations with other classes.

# Association classes

## Definition (Binary association classes)



## Example



# Aggregation & Composition

UML offers support for the specification of composite objects:

- **Aggregations** are a special form of associations that characterize part-of relationships.

## Graphical representation

- **Compositions** are a stronger form of aggregation, with an existence dependency of the parts from the whole. Parts must not belong to several aggregates.

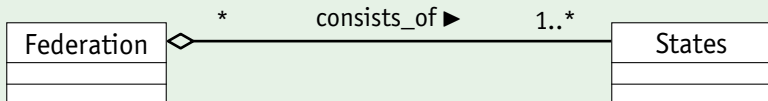
## Graphical representation

# Example: Aggregation

Simple aggregations involve no existence dependency and are non-exclusive.

## Example (Aggregation)

States can exist without Federation; a State may belong to several Federations at the same time.

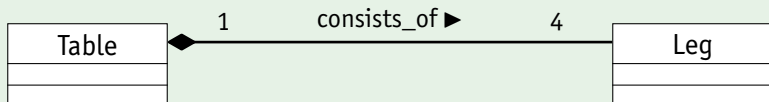


# Example: Composition

A composition involves an existence dependency and is exclusive.

## Example (Composition)

Legs belong to exactly one Table, they can not exist without a Table (in this model).



# Association, Aggregation, or Composition?

## What's the difference? When to use which?

- All three constructs describe “relationships” between objects.
- The more specific constructs imply additional constraints on
  - existence dependencies, and/or
  - exclusive vs. shared “components”.
- *Actually, those constraints are already expressible via the cardinality constraints!*

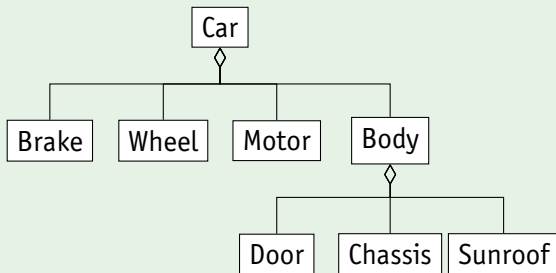
### Combinations of additional constraints

	dependent	independent
exclusive		
non-exclusive		

# Aggregation hierarchies

“Part-of” relationships may cascade over several levels.

## Example (Aggregation hierarchy)

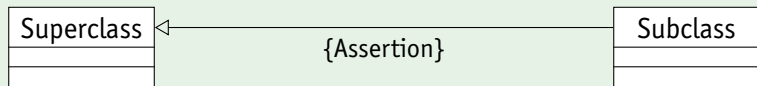


# Generalization and Specialization

Inheritance hierarchies can be expressed using the **generalization/specialization** relationship between classes.

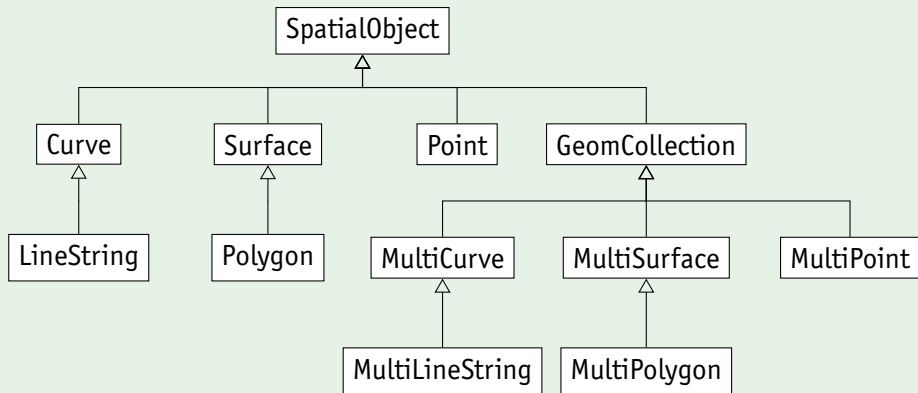
- whether it is called specialization or generalization only depends on where you start from (cf. extended E/R models),
- a subclass can have additional attributes and methods,
- specialization can be constrained by a given assertion (all instances of the subclass will fulfill the constraint).

## Graphical notation



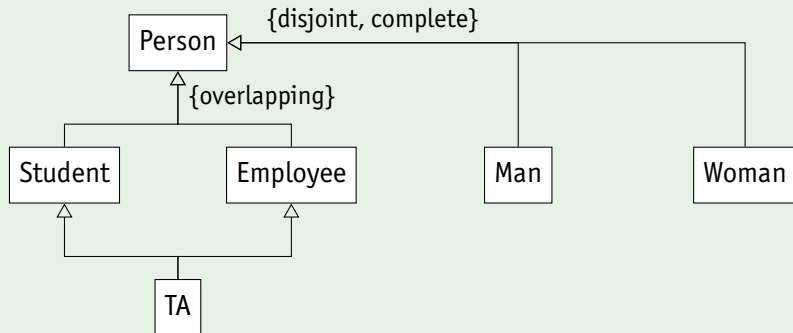
# Examples ( I )

## Example (Class hierarchy)



## Examples (2)

### Example (Specialization with assertions)



**Assertions** (cf. extended E/R models)

**disjoint** ↔ **overlapping**: can instances be shared between subclasses?

**complete** ↔ **incomplete**: does union of subclasses cover superclass?

# More on UML

UML offers *much more* than “just” class diagrams. The approach tries to cover *all* aspects relevant to the design and implementation of large software systems. We will—later—come back to other parts of UML, *e.g.*, those that deal with the description of

- dynamics (state-transition diagrams),
- workflows (use cases).

UML’s major contribution to Software Engineering lies in its attempt to bring together the tools and techniques that, at least in most cases, have been around before (but not used consistently).

# Post-Relational Database Models

The **First Normal Form (1NF)** restriction in the relational model is both,

- **an asset:**

it keeps the data structures, and hence—even more important—the query languages simple; main focus is on **bulk** data processing (set-orientation), not on operations on individual records; hence the latter have a minimal, primitive structure;

- **and a limitation:**

it prevents convenient, “native” storage of *complex* data structures; non-BDP<sup>14</sup> data, such as, CAD data, documents, ..., are not easily mapped into relational tables, furthermore—and more importantly—they are not efficiently retrieved from RDBs

---

<sup>14</sup>business data processing

# Extensions to the relational model

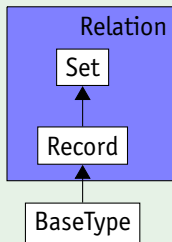
Starting from the early 1980s, many research efforts have been undertaken to enrich the relational model (or to completely replace it with something more general).

“**Relation**” can be viewed as a *type constructor* (as in a programming language, PL) that takes primitive, base types (the attribute domains) and names (attribute names) and constructs a new type (a relation’s schema) from those.

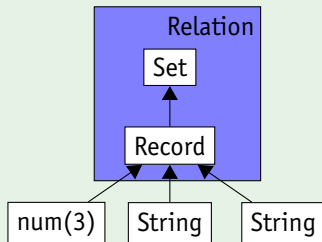
- From a PL perspective, the type constructor “relation” is a combination of the “record” and “set” constructors:  
`relation`  $\equiv$  `set-of-records`.
- This type constructor, though, is restricted in two ways:
  - 1 it can be applied **only once**,
  - 2 it can be applied to **basic types only**.

# Graphical representation

## "Type constructor" Relation



## Example



Emps		
<u>eno</u>	fname	lname
⋮	⋮	⋮
007	James	Bond
⋮	⋮	⋮

# Possible extensions

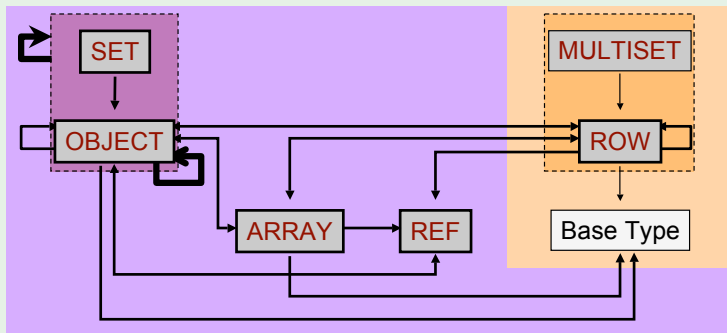
In order to generalize, one could allow

- 1 the use of many more base types,  
Such **extensible RDBMSs** stay within the relational model, yet they add, *e.g.*, arrays or matrices as attribute domains for CAD applications.
- 2 the repeated use of the “relation” type constructor,  
This so-called **nested relational model** has been studied extensively.
- 3 the individual, repeated use of the constituent type constructors, “record” and “set”,
- 4 the use of many more type constructors,  
A variety of so-called **complex object models** have been proposed pursuing this idea.
- 5 ...

It is important to keep in mind that the query languages have to be extended accordingly (see later)!

# The SQL:2003 truth about the relational model

## SQL:2003's object-relational "type system"



Don't panic! We will stick to the orange part (most of the time).

# Summary: Data Modelling

- **Entity-Relationship Models**

- Entities (“objects”) and relationships between them
- Both described by attributes
- Relationships characterized by their cardinality
- Quite a few variants, most extensions include generalization/specialization

- **Relational (Database) Model**

- Everything represented as relations
- Each with key and, possibly, foreign keys as “links” between them
- Comes with *formal* quality criteria (Normal Forms)
- Rather implementation-oriented (compared to E/R)

- **Object-Oriented Models (e.g., UML)**

- Class diagrams use all the (extended) E/R features
- Adds behavioral aspects (methods)

- There are *many* more approaches, not covered here ...