Introduction

1. Basic concepts

Computer systems → applications which access to data and process them

Software engineering guarantee the functionality of applications

Databases: way to store data ensuring persistent content and efficient access

Personal use → each user application has its own system with its particular format and access

Shared big data: there is a **centralized system** with standarized format and access for different applications

Access to large volumes of data

- A program with access by address it's inefficient

Access to large volumes of modifiable data

- What would happen to a standard program using access by file memory address if either one attribute is removed or the order/type of attributes are changed?

Shared files (centralized system) with linked data

- It's necessary to guarantee a correct propagation of changes (consistency of the information)

A system that…

- gives **efficient** acces (acces time, computer resources) to large data volumes
- allow **different** requirements (format, sorting) of user data (multiuser centralized system)
- stores data with **modifiable** structure

… is a **database system**

Database systems (DBS): computer storage systems for the manipulation of related data volumes in centralized system (multiuser). They allow to guarantee the coherence of the content and describe objects from the real world at different abstraction levels. DBS must comply:

- Insert new data into existing files
- View data content
- Update data (delete, insert)
- Delete and add files

Some examples are: PACS (Picture Archiving and Communication System) in health, banking and accounting systems, ESCI academic administration, applications…

Paradigms of the DBSs

Relational

- Data homogenity: logical structure which always keeps the same type of information (varying on number and type)
- Data integrity: always guarantee ACID properties (Atomicity, Consistency, Isolation, Durability)
- Examples: PostgreSQL, Oracle

Non-relational

- Data heterogenity: flexible logical structure that allows to keep the same type of information (varying on number and type)
- Data integrity: always guarantee ACID properties (Atomicity, Consistency, Isolation, Durability)
- Examples: mongoDB, OrientDB

DBS requirements → in order to guarantee a proper operation of all applications using data stored in a DB, the system must:

- Guarantee the **data integrity**
- Be **efficient**
- Grant **independency** between program code and data format

→ **Integrity**

- Content **consistency** and **coherence** of information stored in the DB
- Avoid **redundancy** which are incorrect or duplicated values in the requested information
- DBS must guarantee the data **integrity when content is modified** (insert, update and changes propagation)

→ **Incoherence**

- Impossible value of the requested information
- Avoid incorrect values in the requested information

→ **Inconsistency**

- Same query gives different results depending on its implementation
- Two sources of information (files, field's files) which are equivalent (same semantical content) have different values

→ **Redundancy**

- Repeated information has the following problems:
  - Unnecessary disk usage
  - Duplication on updates
  - Risk of inconsistencies
- DBS must guarantee minimum redundancy and update on information to all files if one is changed (change propagation)

Efficiency

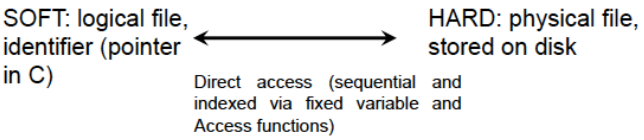
DBS should minimize the maximum time for a query (given for access to all records) based on contents and functional needs of the DB:

- Storage structures. Restructuring the disk information distribution (minimize access to disk)
- Compression techniques. DBS compress the information to be stored into the disk (minimize content)
- Characteristics of the devices where the data is stored
- Concurrence control of accesses of several users (priorities)

Independence

→ Access by **address**

- Access to data (disk) in 3rd generation programming languages (C, Pascal, C++) depending on the physical format of the data stored in HW devices

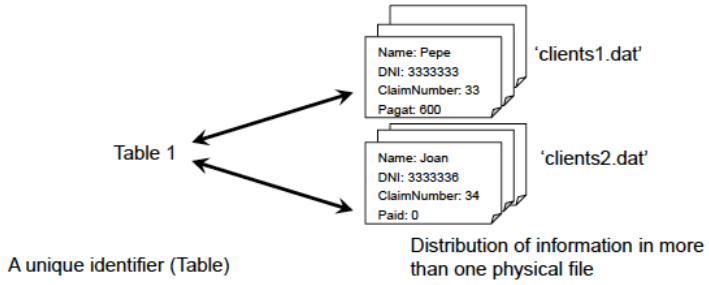


→ **Modifiable** structure

- What would happen if we add or remove fields? Change the type of fields?

→ Access by **content**

- Independence between physical and logical
- **Table**: object (logical) from which you access to the data (physical)
  - **Rows**: tuples or records ('individuals', 'objects')
  - **Columns**: attributes or fields ('properties')
- DBS must ensure that changes in the storage structure and data access **do not affect** the applications that use them
- Main requirement of a DBS:
  - DBS guarantees simultaneous access to **more than one** physical file with the **same** logical file (**identifier**)



2. Components of a Database System (DBS)

Computers:

- Databases
- HW (where DBS is installed)
- Database-Management System (DBMS)
- Applications

Humans (actors):

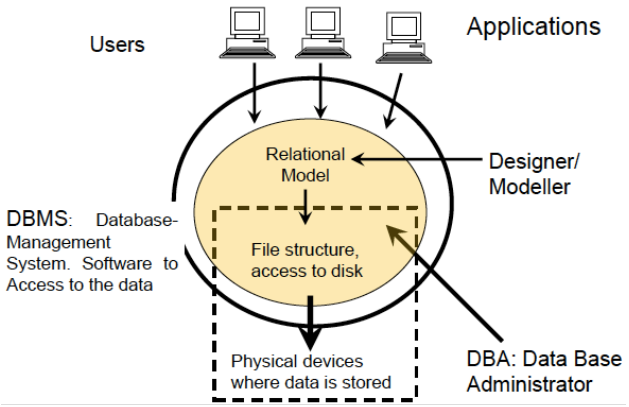
- Modelers
- DB Administrators DBA
- Programmers
- Final users

Components

→ **Users**

- **Final user**: access to the DBS using applications developed by programmers

- **Programmer**: design applications that perform queries to the DBS using the DBMS language
- **Data Base Administrator** (**DBA**): responsible to maintenance of the DBS:
  - File system (physical design)
  - Permissions for data access
  - Updating of information
  - Security backups
  - Performance system



→ **Database**
- Description of the data in related tables (relational model) with access by content regardñess of the structure of file which save the data and the type of disk access

→ **HW**
- Disks (save data)
- RAM (save data and execute SW)
- CPU (execute SW)

Capacity and access time are important for the DBS performance

Administrator manages the selection of the HW

→ **Data Base Management System** (**DBMS**)
- SW which allows create, keep and access to a DB
- Examples: MySQL, Microsoft SQL Server, SYBASE, PostgreSQL, Oracle
- Manage access to DBS:
  - Creation of the DB
  - User requests (insert, consult, delete)
  - Write data to the files
  - Data protection and access permissions
  - Provides an independent view of HW operating system through tables, attributes and a handling data language (SQL: Structured Query Language)
- Manage and guarantee many properties of a DBS:
  - Data integrity (propagation of changes)
  - Access control (security)
  - Security backups
  - Management of multiple user interfaces
  - Performance of a DB (distribution of data in file)

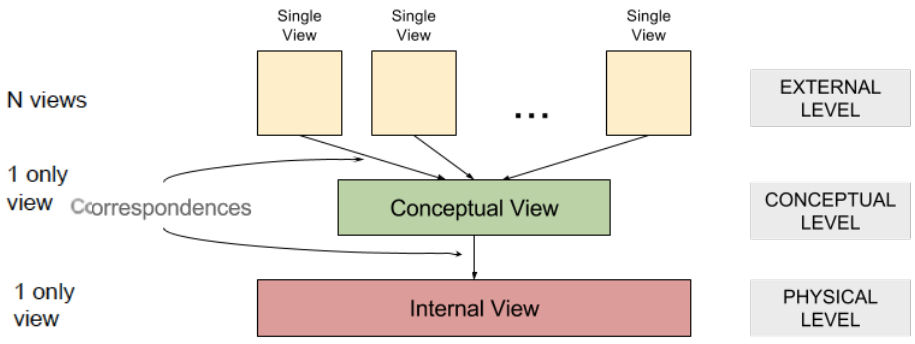
Architecture

1. Definition

Structure and description of DBS in modules to guarantee the independence. Most common architectures:
- ANSI/SPARC
- Client-Server:
  - Back-end/Front-end
  - Distributed systems

2. ANSI/SPARC

Structure the DB according to the description of the data in three levels of abstraction

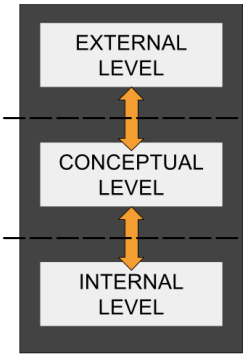


ANSI/SPARC → Abstraction Levels

- **External** level: presentation of data to users (applications that use a relational model)
- **Conceptual** level: logical description of data → **tables** (relational), **collections** (non-relational)
- **Internal** level: organization and storage on physical files (↓-level, pointers, indexes…)

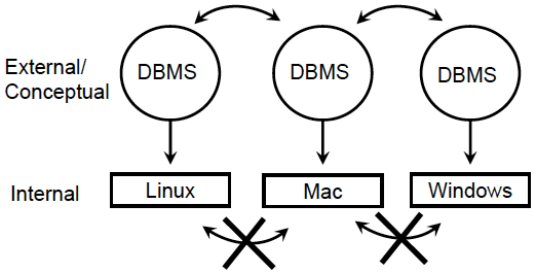
Data independence

Modification in the external/conceptual level **do not** affect the internal level.



Logic: modification of the conceptual level without affecting the external level

Physical: internal distribution does not affect the functionalities of other levels



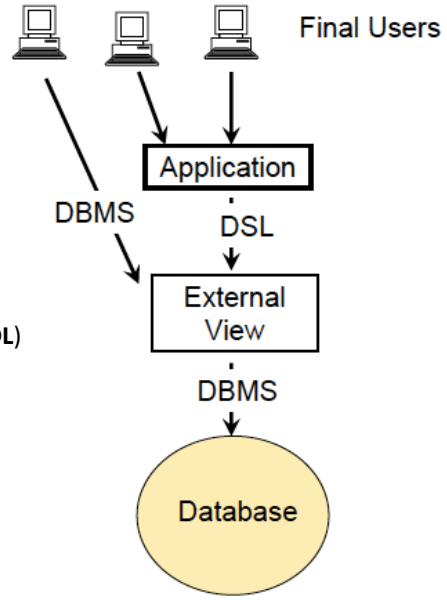
External and conceptual levels **do not** depend of the operating system. Internal level **does** depend of the operating system.

External level

Presentation of the data to users

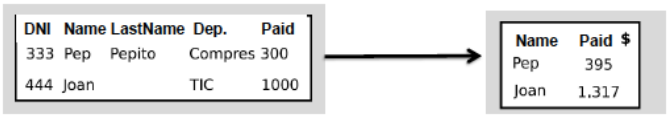
- Final user: accessing via **SQL** or some application
- Application: developed with ↑-end programming languages (C++, Pascal…) which incorporate a Data Sub Language (**DSL**)

View: content (attributes values) on the **part** of the DB as seen by a user/application at **a given time**



Name	Paid €
Pep	300
Joan	1000

DSL sub-language in the ↑-end language is able for interaction with the DBS: Data Definition Language (**DDL**) → to create data; tables, views, Data Manipulation Language (**DML**) → to modify data; select option…

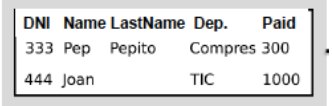


DNI	Name	LastName	Dep.	Paid
333	Pep	Pepito	Compres	300
444	Joan		TIC	1000

Name	Paid $
Pep	395
Joan	1.317

Conceptual level

View: content (attribute values) of **all** the DB **at any time**



DNI	Name	LastName	Dep.	Paid
333	Pep	Pepito	Compres	300
444	Joan		TIC	1000

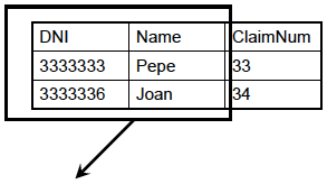
- Defined based on **DDL** conceptual (script SQL: create table, create domain, create foreign key…)
- Specify **integrity** and **security** controls
- There must be a correspondence between the **outer** and the **conceptual** scheme

Internal level

Content of the files which **store** the content of tables

Does not match with conceptual view (data distributed in **different** files)

Files description according the operating system format



DNI	Name	ClaimNum
3333333	Pepe	33
3333336	Joan	34

Filename, table's attributes, access (indexing)

3. Client-Server

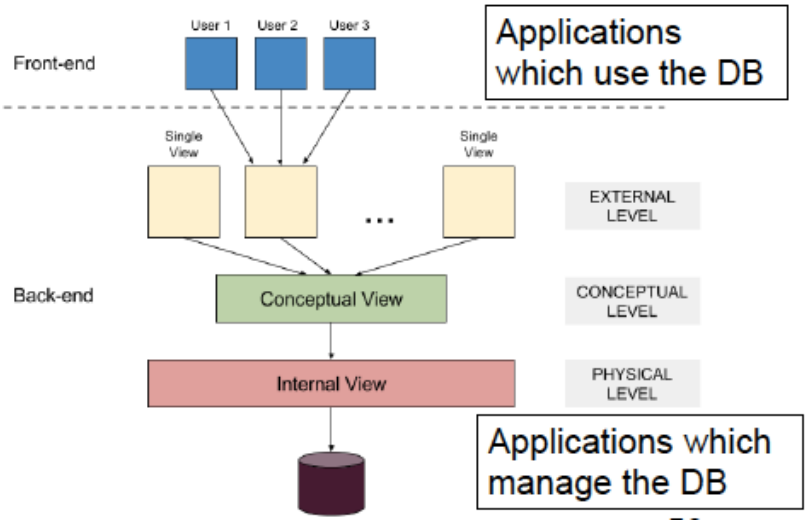
3.1 Back-end/Front-end

Structuring the DB according the type of application (SW) that are executed on the DB in two levels:

Back-end (behind): SW which executes all functions specified in a DBMS (Oracle, PostgreSQL, MySQL)

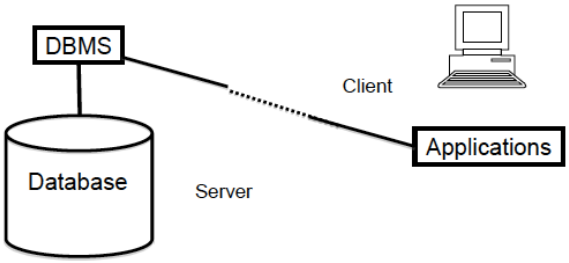
Front-end (on front): applications executed over a DBMS (web browser, smartphone app, SQL developer…)

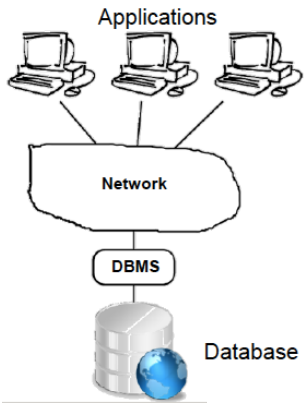
Usually **1 back-end** and **n front-ends**.



3.2 Distributed system

The separation of the DB according to the SW allows each part to be in different physical nodes (or computers)





Distributed system: **Client-Server**

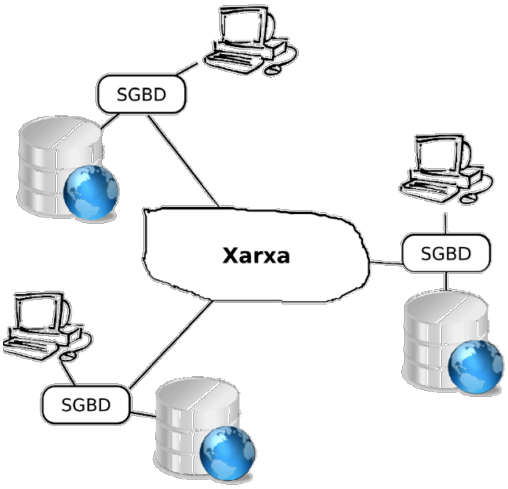
Client (Front-end): machine which executes the **application**

Server (Back-end): machine which executes the **DBMS**

Distributed system on **network**

Distribution of data on several machines and operating systems
N client-server locals (**nodes**) are communicating between them using SQL
Each machine contains a DBS that works like:

- Server **per** certain users
- Client **for** other users



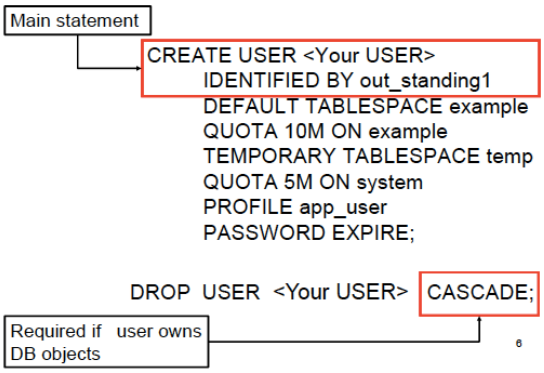
Database administration: users and security

Users, roles and privileges

- **Privilege**: right to run a particular type of SQL statement → create a season, execute SQL query, run a PL/SQL procedure…
- **Users**: access name to the Oracle DB with a **disk quota** and a **set of privileges** assigned to
- **Roles**: group of privileges
  - Users can be assigned to one or more roles
  - They inherit role privileges
  - They **do not** have assigned any disk quota

Users in Oracle

- **Sys**: perform all administrative functions to mantain DB integrity. Never creates any table in the SYS schema
- **System**: same as Sys user except backup and recovery, and database upgrade
- **Other users**: see right image



Create and drop users

Grant/revoke user privileges

Grant privileges:

- **grant** create session, select any table **to** <Your User>;
- **grant** unlimited tablespace **to** <Your User>;

Revoke privileges:

- **revoke** create session, select any table **from** <Your User>;
- **revoke** unlimited tablespace **from** <Your User>;

Managing user roles:

- Create: ***CREATE ROLE <Role Name>;***
- Drop: ***DROP ROLE <Role Name>;***
- Grant/revoke privileges
  - ***GRANT create session TO <Role Name>***
  - ***REVOKE create session FROM <Role Name>***

Design introduction

0. Review of previously discussed

Key points of a DBS

- Data integrity (consistency and coherence of the content)
- Independency program/format data
- Efficient system (access to data)

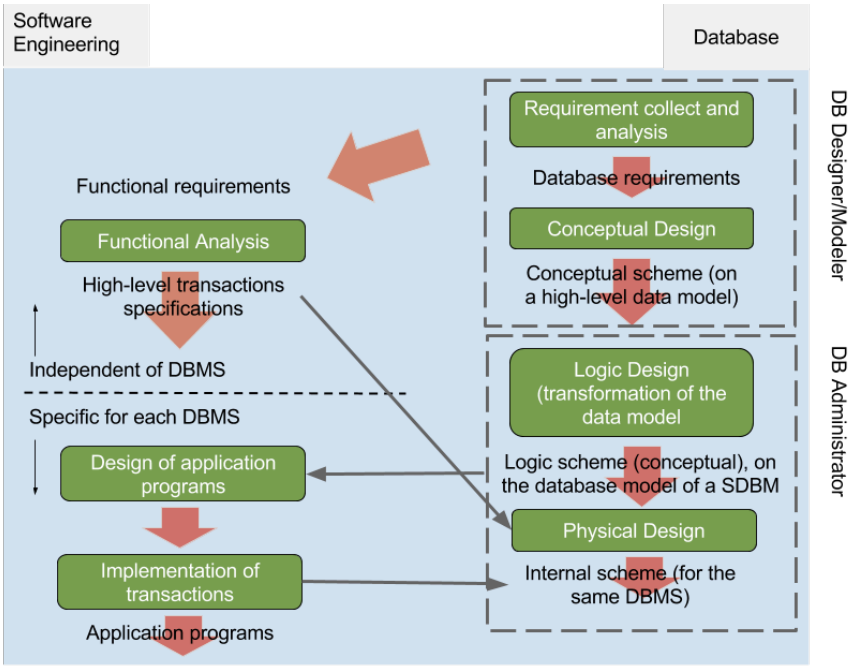
→ Description at Conceptual Level (Relational Model and E-R Diagram)

→ Physical level description (file system and devices, DBM)

Data description

The DBS must satisfy the necessities of information to different users who share the data. Description of the data (according to user's requirements) guaranteeing the efficiency and consistency.

→ Design of the database



1. Design phases

Design of an application

Design phases of a DB
- Capture and analysis of requirements
- Conceptual design

————————————————————

- Logic design
- Physical design

→ Specific for each DMBS

Requirements
To charectarize the DB user necessities, their data and usage:
- **Data requirements** (what information we have): description of data and the relationship between them
- **Functional requirements** (what we want to do with the data): description of the operations (transactions) to be carried out with the data

2. Conceptual design
2.1 Basic structures
Translation of the data requirements to the abstract model that has been choosen (conceptual scheme):
- ER Diagram
- Descriptive report

It is necessary to validate its functionality according the transactions indicated in the functional requirements (set of tests). Is an abstract description (↑-level model) of independent data of the DBMS.

Abstract data models
They allow semantically to model the data and links (relationships, inter-relations) that exist between them. Developed to ↑ the effectiveness and accuracy of the DB design. Existing models:
- Entity-Relationship
- Object Oriented-extended ER model

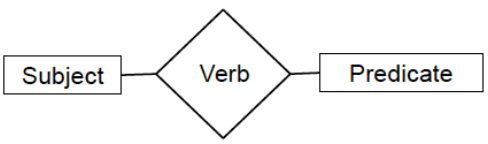
2.2 Properties of the links
Logical design
Translation of conceptual design to the data model (generally relational) of the DBMS. Precise implementation depends of the DBMS of our application:
- Table diagram, SQL implementation → DBM
- Classes diagram, UML implementation → Functional analysis (SW engineering)
- Object oriented DB (ODL, OSL) → Management and administration of a DB

Physical design
- Structure the tables/classes in files and devices
- Depends of the DBMS and the selected operating system
- The device distribution must guarantee the efficiency in access and space
- Depends of the transactions specified in the functional requirements

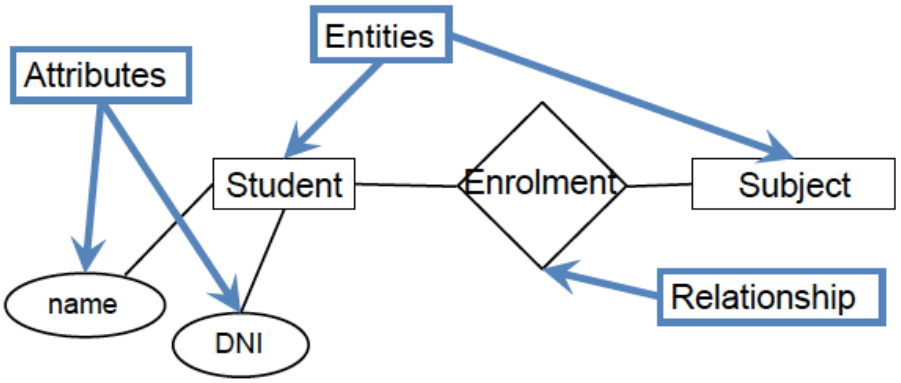
Basic ER-Design
1. E-R Model introduction
Description/Schematic representation of a real situation (which our application must manage) that requires **saving information** from different objects (**entities**) that are **related** to each other. Example:
- University Enrolment Management: in an enrolment management system I want to know which students have enrolled in engineering subjects. For each student (and subject) I want to keep certain information (name, contact details…). In particular, I will have to save the **necessary data** so as **not** to **confuse** two students.

Components: they let us describe **who does what**
- **Entities**: object (event) from real world → "Subjects, complements"
- **Relationships** (relations, inter-relations): association relating several entities → "Verbs"
- **Attributes**: entities and relationships information or features

E-R Diagram: graphical representation of all entities and relationships with their attributes
- **Relationships** → rhombus



- **Attributes** → ellipses
- **Entities** → rectangles



2. Basic structures

2.1 Entities

2.1.1 Definition

Real-world object or action distinguishable from the rest of which we are interested in **keeping** some **properties**. Semantic description of an object. They are represented by squares/rectangles.

- They can be **concrete** (corresponds to a physical object) or **abstract** (corresponds to an action or concept)
- Entities have a set of characteristics (**attributes**) with values that uniquely identify each instance (case, tuple) of the entity

Examples:

- A university student is a **concrete** entity. Each student must have a characteristic that uniquely identifies them. For example, NIU 1007899 could uniquely identify a particular student.
- Book loans can be considered as **abstract** entities, and the UAB library's loan code L-15AJY9 uniquely identifies "loan" instances

2.1.2 Candidate keys

Every entity must have a subset of its attributes with values that uniquely identify each instance (case, tuple) of the entity: these are candidate keys (CK).

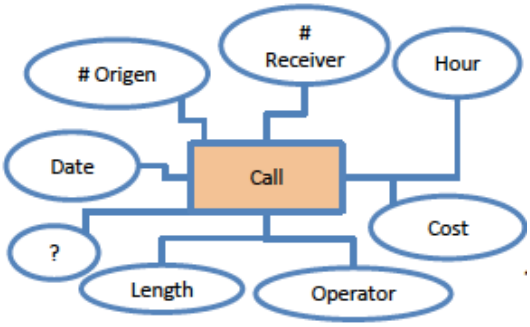
→ Minimum set of attributes that uniquely identify each instance. A set of attributes that uniquely identify an instance and **do not** contain other keys inside or redundant attributes.

- **How to identify them**: CK attribute values are **not repeated**. Different instances have some of the CK attributes different
- **Minimality**: if we drop any of CK attributes we could have instances with the same values of CK and therefore they would no longer identify

Examples:

What are the necessary attributes to uniquely identify…

- A member?
  - Name, surname, date of birth, DNI, marital status, telephone, member code, address
- A phone call?



From these attributes we can choose the candidate keys:

- A member
  - DNI
  - Member code

DNI		Date	Member Code	Name
3676373L		1-1-75	123	Juan
4748474P		1-1-01	333	Juan
6727271Q		1-1-75	667	Pere

Equivalents as a minimal identifiers

- Phone call
  - (Date, Hour, #Origen)
  - (Date, Hour, #Receiver)

#Origen	Cost	Hour	Date	#Receiver	Min
66234560	0.8	10:00	1-9-18	93581444	1
93581444	0.4	10:20	1-9-18	66234501	5
66234501	0.8	10:00	2-9-18	93581444	1

All 3 attributes are required to identify. If we drop anyone, there are repetitions

2.1.3 Primary keys

Primary key (PK): CK selected by the database designer. An entitty without PK is **not well-defined**

DNI	Date	Member Code	Name
3676373L	1-1-75	123	Juan
4748474P	1-1-01	333	Juan
6727271Q	1-1-75	667	Pere

A member → Member code

#Origen	Cost	Hour	Date	#Receiver	Min
66234560	0.8	10:00	1-9-18	93581444	1
93581444	0.4	10:20	1-9-18	66234501	5
66234501	0.8	10:00	2-9-18	93581444	1

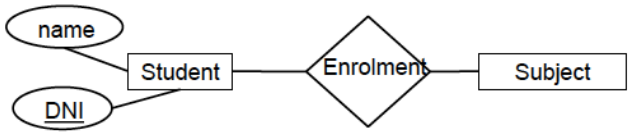
Phone call → (Date, Hour, #Origen)

2.2 Attributes

2.1.1 Definition

Characteristics that **describe** each entity/relationship. They are represented by ellipses.

- Their values define the instances of the entities that is part of the information that is stored in the DB
- **OBS**: the attribute values can change over time

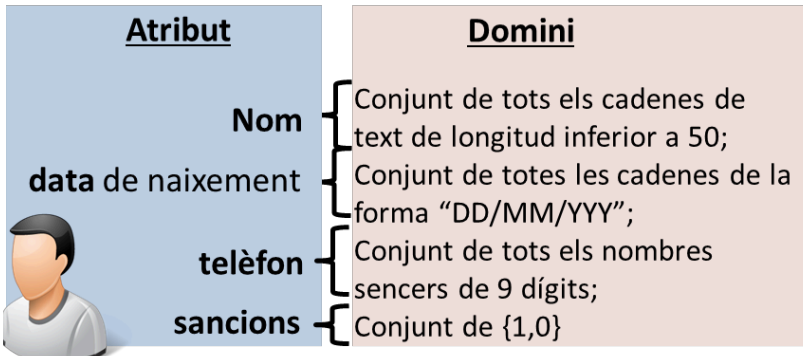


DNI	Name
3676373L	Pere
4748474P	Juan
6727271Q	Pere

Student instances

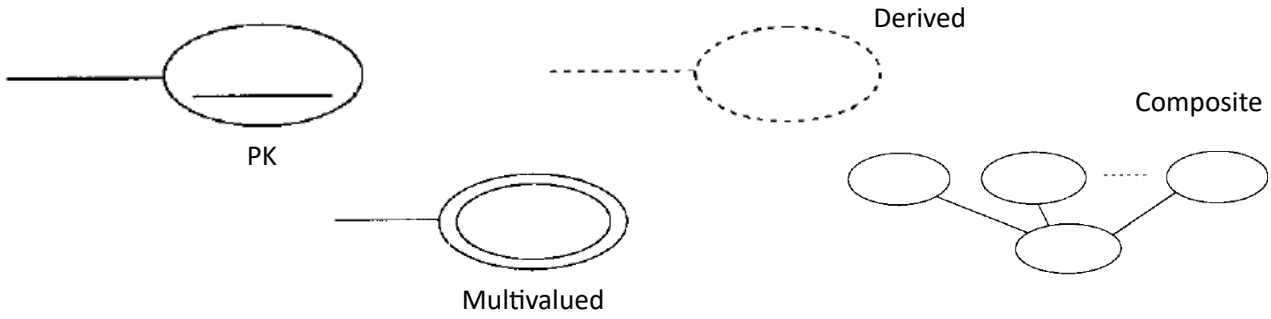
2.1.2 Domain

Each attribute has associated a domain or values set (type and range) permitted



2.1.3 Attributes types

PK: set of attributes that uniquely identify entity's instances
Multivalued: the attribute has more than one value for each instance (they are "vectors" of variable legth for each instance)
Derived: can be computed from the values of the other attribuets
Composite: can be splitted into simplier attributes

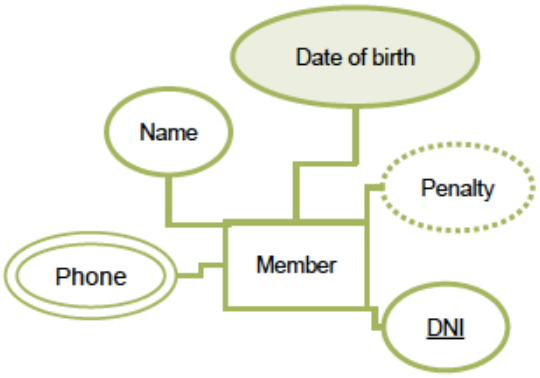


Examples:

→ **Multivalued**

A member can have more than one phone number.
Dani has three phone numbers and Peio has only one

DNI	Name	Surname	Phone
3676373L	Peio	Artola	938373893
4748474P	Dani	Alves	617232066, 623344112, 617734333

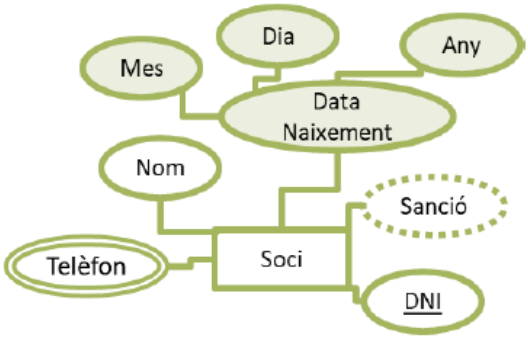


→ **Composite**

Date of birth can be splitted into day, month and year

DNI	Name	Surname	Phone	Date of birth
3676373L	Peio	Artola	938373893	23-Gen-1965
4748474P	Dani	Alves	617232066, 623344112, 617734333	12-Juny-2001

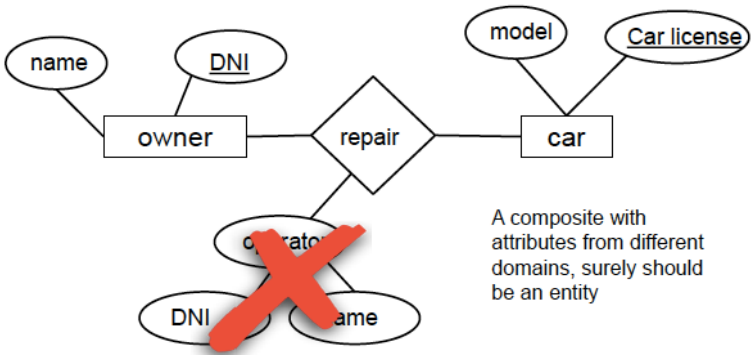
DNI	Name	Surname	Phone	Birth day	Birth month	Birth year
3676373L	Peio	Artola	938373893	23	Gen	1965
4748474P	Dani	Alves	617232066, 623344112, 617734333	12	Juny	2001



→ **Composite vs Entity**

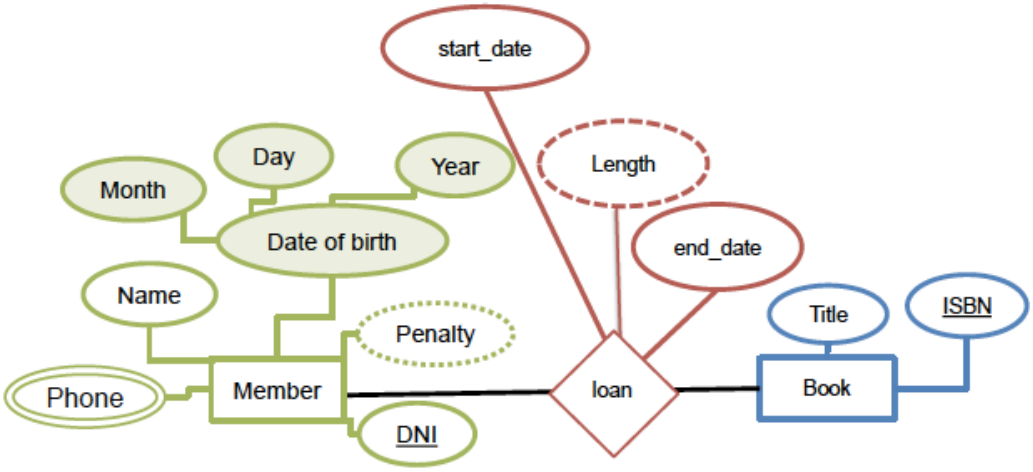
In the composite attributes the pieces are always of the **same type**. A composite attribute with different domains, surely should be an **entity**.

- **Car repair shop**: a car repair shop wants to keep information about its customers and repairs. We want to keep the ID and name of the clients, the attendant and the car being repaired. We want to save the name, ID and the category of the operator. Of the car, the license plate, model, who repairs it and to whom it belongs.



→ **Derived**

The **penalty** can be obtained from the list of overdue books represented by the relation. **Length** is the difference between **start date** and **end date**.



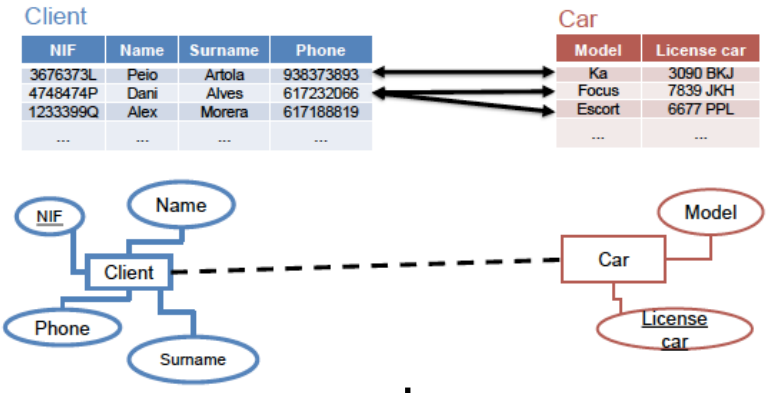
Basic ER-Design

2. Basic structures

2.3 Relationships

Association between different related entities instances. They are represented by rhombuses. Each instance of the relationship is defined by the **values of the PKs** of the associated entities instances

Relationships **never** have PKs, they are already **uniquely defined** by the PKs of the two related instances



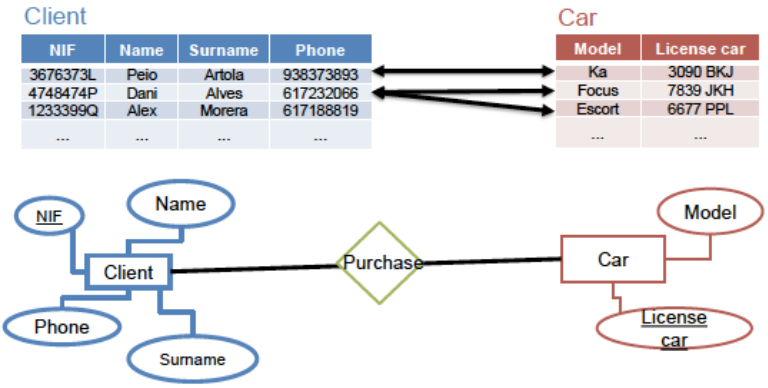
Relationships are **correlated** with the real world

Real world:

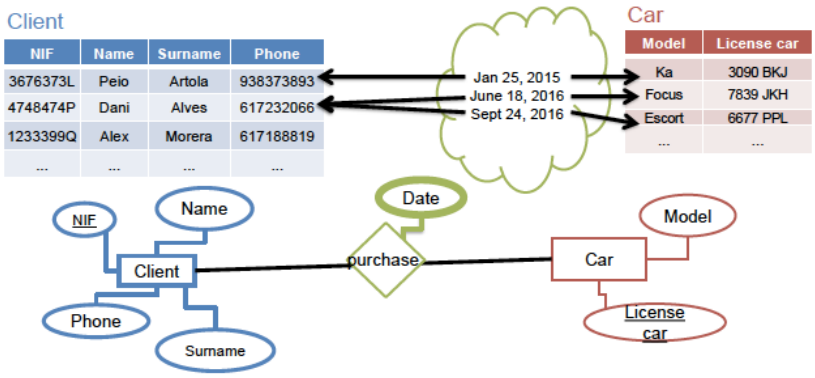
→ Peio Artola buys a Ka with license 3090BKJ

→ Dani Alves buys a Focus with license 7839JKH and an Escort with license 6677PPL

→ …



Sometimes it may be appropiate to associate attributes with relationships in themselves



Real word:

→ Peio Artola buys a Kia with lincense 3090BKJ on 25/01/2015

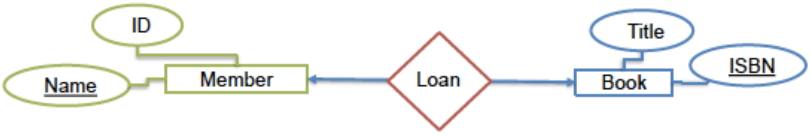
→ Dani Alves buys a Focus with license 7839JKH on 18/06/2016 and an Escort with license 6677PPL on 24/09/2016

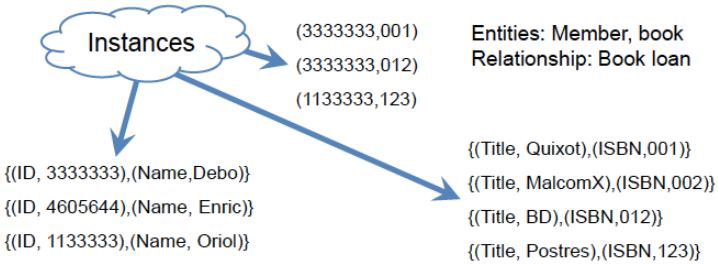
Examples

→ Information contained in the relationship

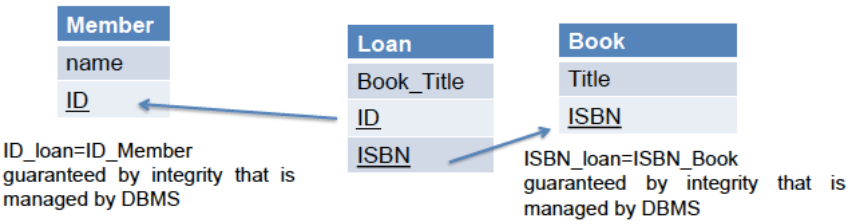
Library: we want to manage a library network loans and know the free copies. From a **book** we want to save the ISBN and the title. From a **member** we want to save the name and ID. We want to know what books each member has.

- Entities: book, member
- Attributes: (title, ISBN), (name, ID)
- Relationship: book loan

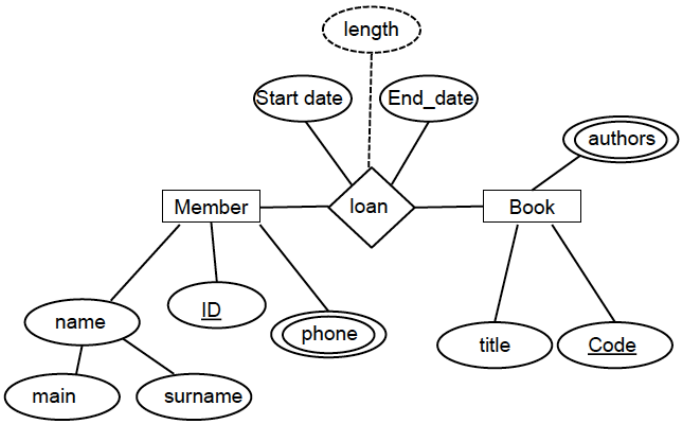




With this design, can we know the titles of the books that are on loan? How would you modify it? Should the title of the book be incorporated as an attribute of the "loan" relationship? The relationship contains (indirectly, through the PKs of the related entities) all information (attributes) of the entities that it relates. You do not need attributes to the link that are already in the related entities (relationships **never** have PKs).
In addition, these repetitions introduce **redundancy** of attributes that the DBMS will not handle (updates, FK modifications only) and may **violate integrity**.



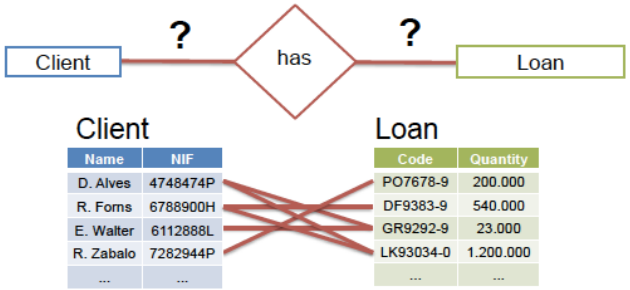
Relationships can have attributes that describe the action they represent



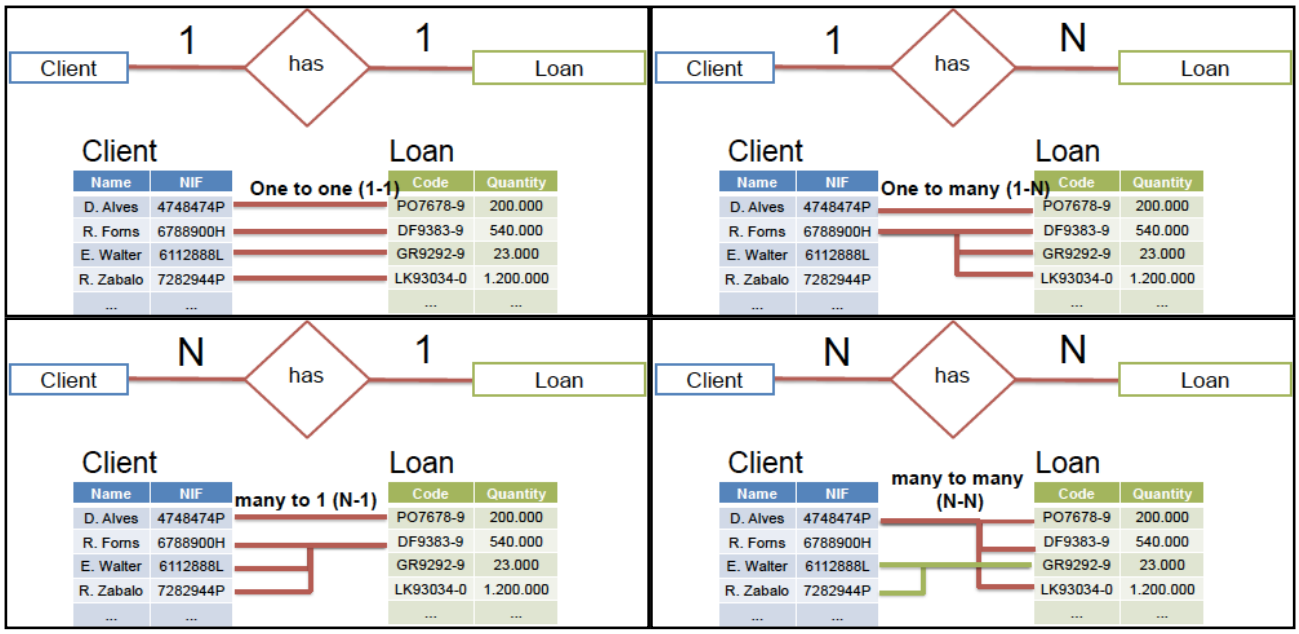
→ Identify entities and relationships
Database subjects: students of this subjects are distributed in different groups according to type of teaching (theory, problems and practices). Each of these courses is taught by a teacher (which can always be the same). In addition, depending on the number of students more than one teacher can do the same type of teaching. In addition, several evaluation tests will be performed for each type of teaching.

- Entities: students, groups, courses, teacher, evaluation tests
- Relationship: students are distributed in different groups, groups vary by courses, teachers teach various types of teaching, one teacher can do the same type of course, evaluation tests will be performed for each type of course

3. Relationships properties
3.1 Cardinality
Maximum number of instances of an entity that may be associated with an instance of the other entity involved in a relationship (there are several types)



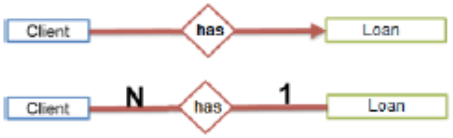
- **Cardinality 1-1:** a customer can only have one loan (and one loan can only be took by one client)
- **Cardinality 1-N:** a client can have more than one loan (and a loan can be took by a client)
- **Cardinality N-1:** a loan can be shared among multiples clients (but one client **cannot** have more than one loan)
- **Cardinality N-N:** no restrictions (one loan can be shared between multiple clients and one client can take more than one loan)



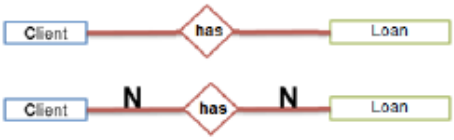
Symbolic representation: the simple or directed line (arrow) serves to distinguish different relationships

The **double arrow** indicates a **one-to-one** relationship

A **simple arrow** indicates a **many-to-one** relationship

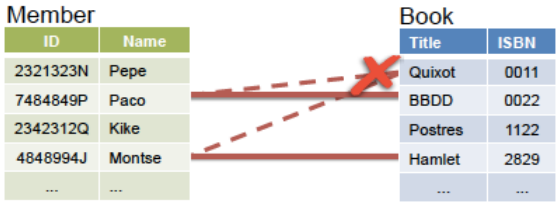


A **simple line** indicates a **many-to-many** relationship

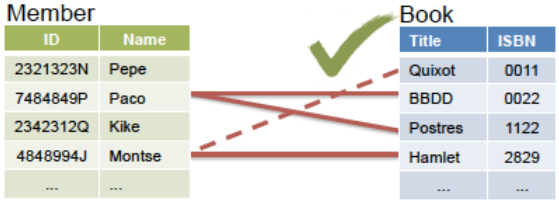


Examples
→ Cardinality type

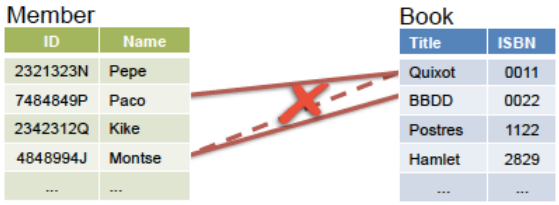
Consider the ER design where the "loan" relationship has **1-1** cardinality. Does this design allow Montse or Paco to take Don Quixote? No, because one book could be load by one member



Consider the ER design where the "loan" relationship has **1-N** cardinality. Does this design allow keeping Montse to take Don Quixote? Yes, because one member can have more than one book on loan and one book can only be loaned to one member



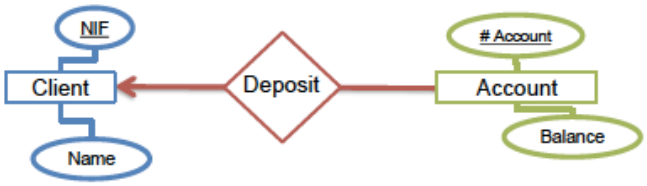
Consider the ER design where the "loan" relationship has **N-1** cardinality. Does this design allow keeping Montse to take Don Quixote? No. Only if we remove from the DB the book that Montse already has on loan (BBDD, 0022)



→ Attributes definition

Where is it more convenient to define an attribute? The cardinality of a relationship can determine where it is more convenient to define an attribute.

Consider the ER design where the cardinality of the "deposit" relationship is **1-N** (one-to-many). Suppose we want to keep **access date** for each bank account.



Client

Name	NIF
D. Alves	4748474P
R. Forns	6788900H
E. Walter	6112888L
R. Zabalo	7282944P
...	...

Deposit

Account

# Account	Balance
CTB-246800	500
CTB-357900	1.250
CTB-135700	13.000
CTB-975300	100.056
...	...

Three possible options

→ Option A

Client

Access date	Name	NIF
22-05-2016	D. Alves	4748474P
15-04-2016	R. Forns	6788900H
22-02-2017	E. Walter	6112888L
25-01-2017	R. Zabalo	7282944P
...	...	...

Deposit

Account

# Account	Balance
CTB-246800	500
CTB-357900	1.250
CTB-135700	13.000
CTB-975300	100.056
...	...

Each time D. Alves accesses the CBT-135700 account he will **overwrite** the acces date of his other account (the CTB-246800). Then, this option is unable to record access dates.

→ Option B

Client

Name	NIF
D. Alves	4748474P
R. Forns	6788900H
E. Walter	6112888L
R. Zabalo	7282944P
...	...

Deposit

Account

# Account	Balance	Access date
CTB-246800	500	22-05-2016
CTB-357900	1.250	15-04-2016
CTB-135700	13.000	22-02-2017
CTB-975300	100.056	25-01-2017
...	...	...

It is now possible to record the date of the last entry in each account separately. But, new access **overwrites** the previous date. Then, we obtain a possible solution (**last acces date**)

→ Option C

Client

Name	NIF
D. Alves	4748474P
R. Forns	6788900H
E. Walter	6112888L
R. Zabalo	7282944P
...	...

Deposit

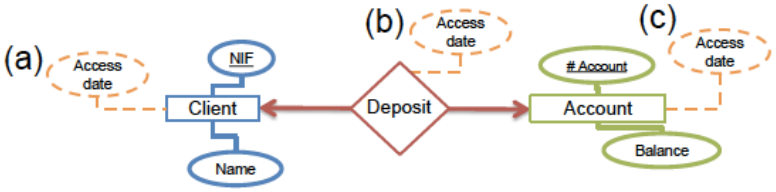
Access date
22-05-2016
15-04-2016
22-02-2017
25-01-2017
...

Account

# Account	Balance
CTB-246800	500
CTB-357900	1.250
CTB-135700	13.000
CTB-975300	100.056
...	...

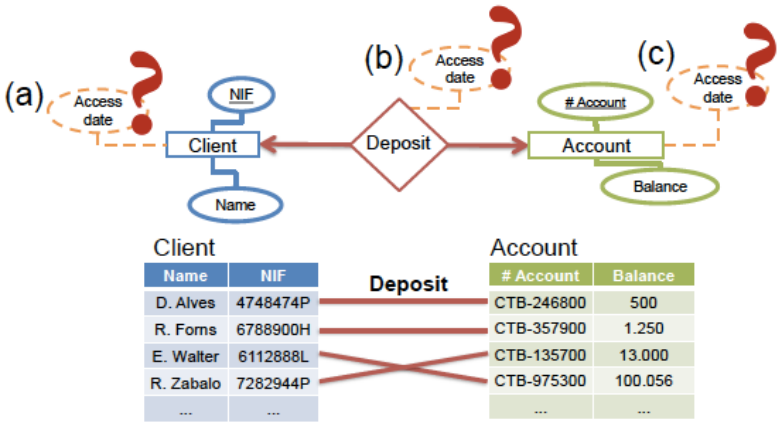
Each account is now also related to its last acces date, similarly to option B. Then, we also obtain a possible solution (**last acces date**)

Consider the ER design where the cardinality of the "deposit" relationship is **1-1** (one-to-one). Suppose we want to keep **access date** for each bank account.



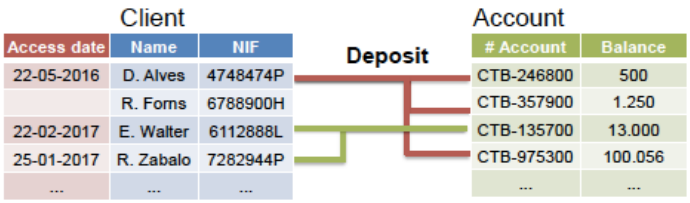
Again, we have three possibilities.

In this case, all three designs retain the **last acces date** for each account and are equivalent.



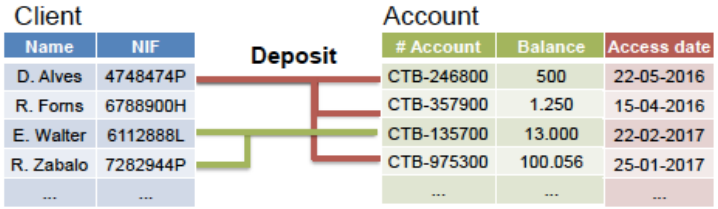
Consider the ER design where the cardinality of the "deposit" relationship is **N-N** (many-to-many). Suppose we want to keep **access date** for each bank account.
Again, we have three possibilities.

→ Option A



Each time D. Alves accesses the CBT-357900 account he will **overwrite** the acces date of his other account (the CTB-246800).

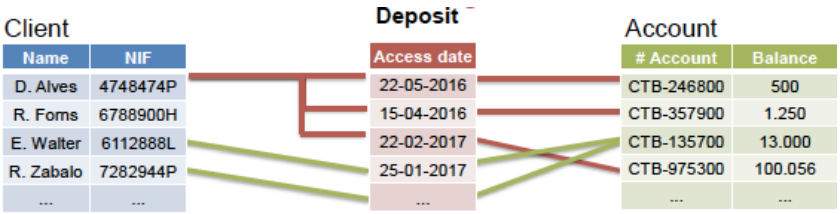
→ Option B



The same as in option A, each time E. Walter accesses the CBT-135700 account, the access date of R. Zabalo to the same account will be **overwritten**.

In no case A and B it is possible to register the clients and the dates of access to their respective accounts.

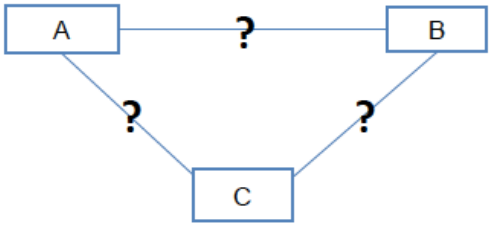
→ Option C

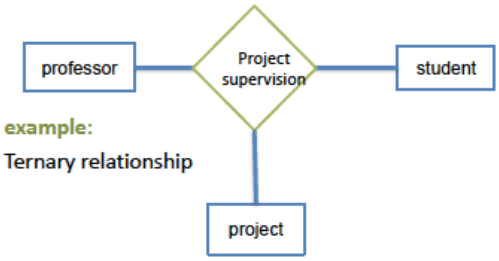


This is the only case where it is possible to register the clients and the last dates of access to their respective accounts.

Final exam question: is it possible for these designs to record a history of clients that includes all access dates to their respective accounts?

Basic ER-Design
3. Relationships features
3.2 Degree
Conceptual model: entity-relation design
→ **Multiple relationships:** while most relationships are binary, there are times when it is more convenient to associate **elements from more than one entity**



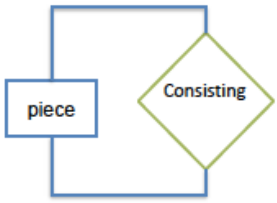


Type of relationship: as the number of entities involved, a relationship can be classified as:
- **Unary** (**1**-degree): relation to the same entity (hierarchical structure or equivalent structure)
- **Binary** (**2**-degree): relationship between two entities. It is the most common
- **Ternary** (**3**-degree): relationship between three entities
- **N-ary** (**n**-degree): relationship between n (known) entities

Unary (1-degree): relation of an entity with itself (**hierarchical**)

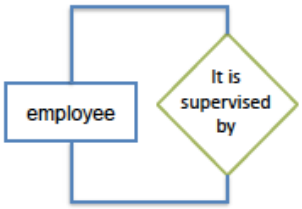
A piece can be composed of other pieces.

example: Hinge comprising a bis and a lid.



A worker can work under the supervision of other employees.

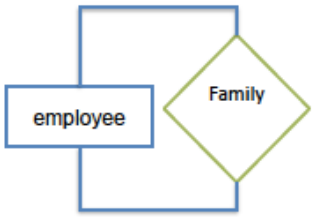
example: an employee of a bank can be supervised by a manager (another employee of the same bank).



Unary (1-degree): relation of an entity with itself (**equivalent structure**)

Unary equivalent relationships no establish hierarchy

example: A worker can be family of another worker (and this for the first).

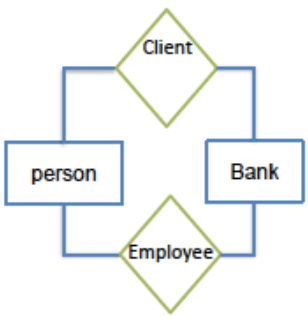


Binary (2-degree)

Multiple binary relations:

It may be the case of more than one binary relation between two entities.

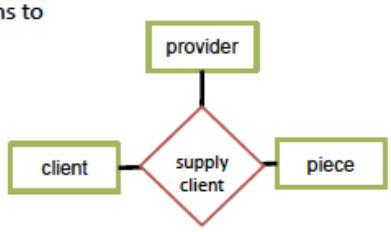
example: An employee can be,also, a bank client



Ternary (3-degree)

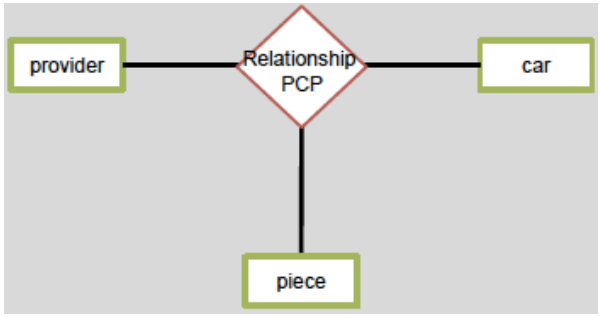
- it is usually possible replace a relation of degree > 2 by a number of different binary relations
- In some cases these two designs are not equivalent
- They are very rigid structures

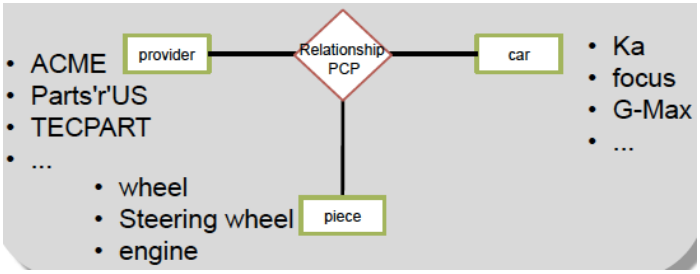
example: Some vendors provide certain items to certain customers



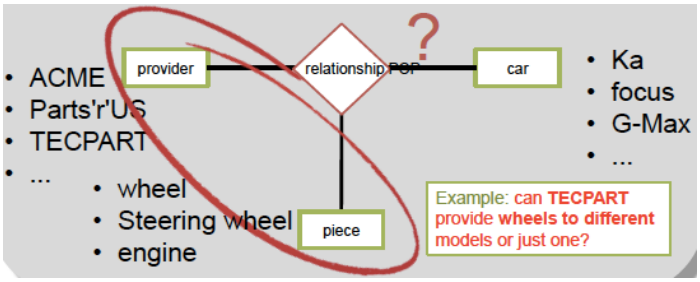
Discussion 2: Ternary relationships cardinality (degree 3)

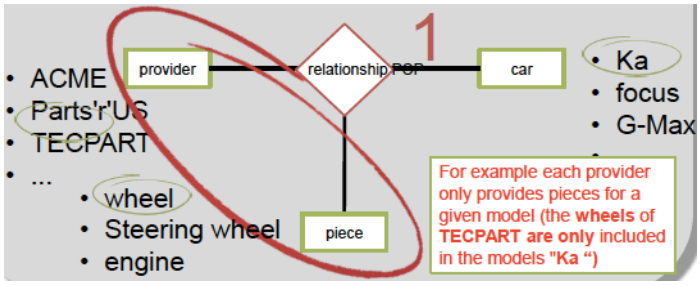
Consider the following ER design, expressing cardinalities. Consider the following instances





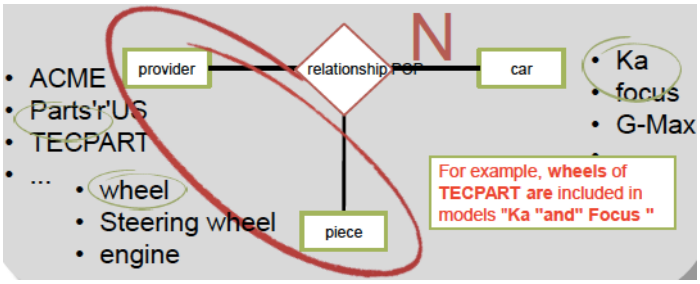
For a given instance of a given supplier of a given piece… Can this particular combination (Provider/Piece) satisfy several cars or at most just a car?

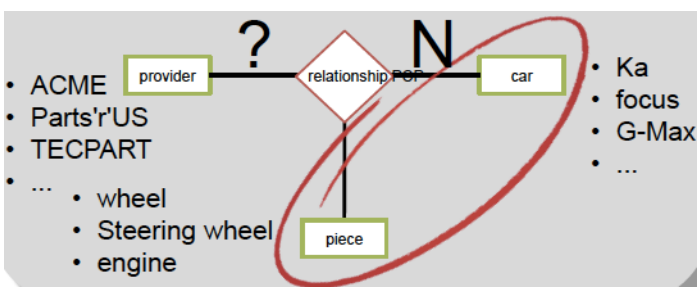




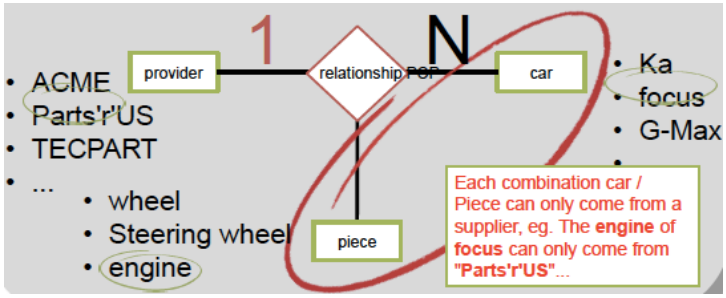
If combination of provider and piece can only meet one model car, then car cardinality is **1**

If combinations of provider and piece can satisfy several cars, then the car cardinality is **N** (many)



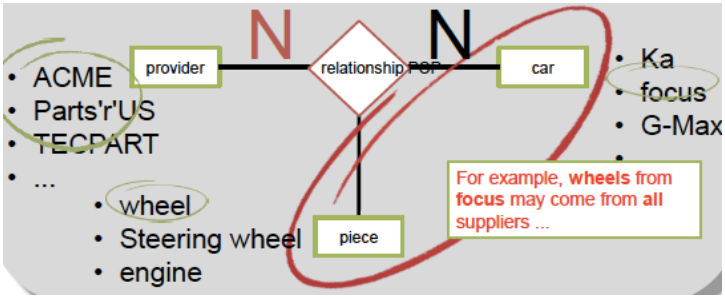


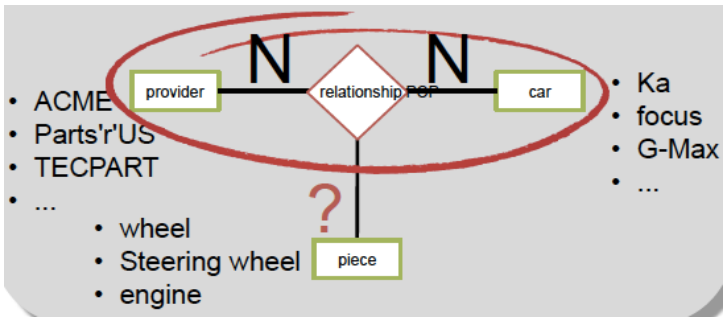
Same for the other combinations: *for a given instance of a car and a given piece… Can this particular combination (car/piece) relate to several suppliers or simply with a single supplier at most?*



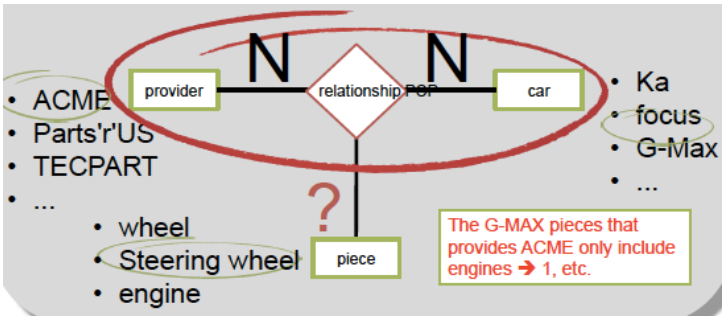
If a combination car-piece can only come from a single provider, then the cardinality of provider is **1**

If a combination of car and piece may come from different vendors, then the cardinality of provider is **N** (many)

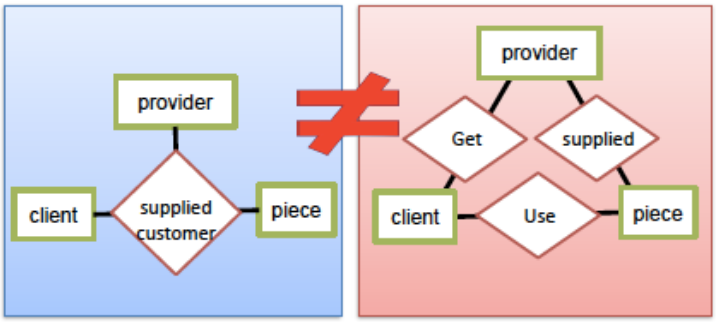




Also for a given instance of a given supplier and a given car… Can this particular combination (provider/car) relate to various pieces or at most to just one pice?



In this case it is not the same a **ternary relation** that a **three binary** relations



The ternary relationship between entities 'supplier', 'piece' and 'client':
→ Some vendors provide certain items to certains customers
→ **NOT** semantically equivalent to:
- Providers supply pieces
- Customers use pieces
- Client receives supplies from provider

This are **binary relations**

Consider the following individual instances
- Provider = IKEA
- Pieces = hinges
- Client = UAB

→ IKEA supplies hinges to UAB

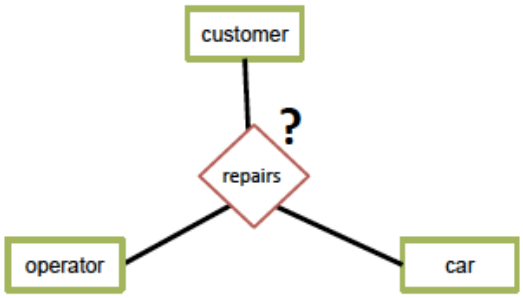
This is semantically equivalent to…

- IKEA supplies hinges
- Hinges are used for UAB
- UAB receives IKEA products

→ Binary relations

Example 2

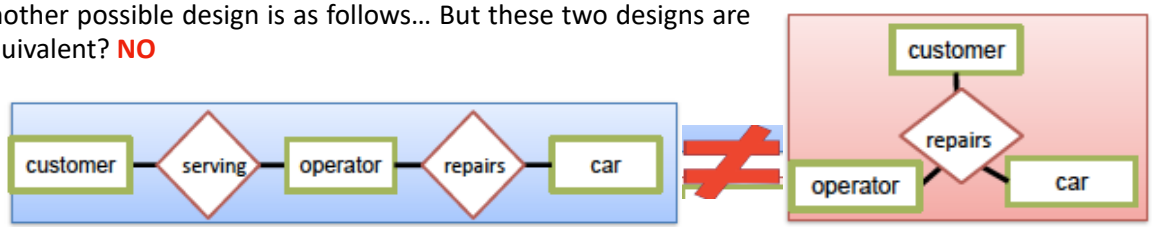
Car repair: a car repair shop wants to keep customer information, the operator that serves them and the car being repaired



The interrelationships with degree > 2 are more specific, but more rigid: we have to know all instances when they are created. For example, when a new client comes to your car you have to assign a mechanic (operator)

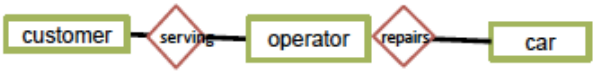
Not practical

Another possible design is as follows… But these two designs are equivalent? **NO**



This scheme allows new customers who do not come for reparations ('partners') but I can not relate each car with its owner. I have ambiguity: **trap connection**

Discussion: Trap connection

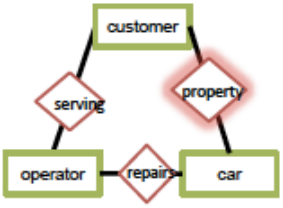


example:

- *Operator 7 serves Mr. Puig*
- *The operator 7 repairs a car with licenes plate 567 HYT*

Is repairing the Mr. Puig a car? We don't know. I can not connect cars with their owners

Solution:

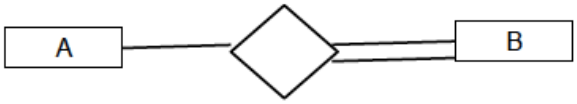


- *Mr. Puig is the owner of license plate HYT 567 car*
- *Operator 7 serves Mr. Puig*
- *The operator 7 repairs a car with license plate 567 HYT*

3.3 Participation

Conceptual model: entity-relation design (E-R)

Completetness: all B items have to be related with some A item. It is represented by a double line from B to the relationship. To insert and instance B, I must have inserted an A instance to relate with



Participation restrictions

Specify whether the existence of an entity A **depends** on its relation to another entity B. There are two types:

- **Total participation**: **all** elements of A must be related to some element of B
  Indicated by the double line
- **Partial participation**: **not** all elements of A must be related to some element of B
  Suitable for plain line

Example 1: relation car-citizen

- **All** cars must have an owner
- If all cars must have at least one owner (citizen), 'car' participation in the relations is **total**

- **Not all** citizen have to be car owner. The 'citizen' participation in the relation is partial

The combination cardinality/participation appropiate for a particular relation obviously depends o the situation of the real world of this relationship.

Example 2: relation person-book

- Each book should belong to anyone?

No. Depends on the context, but in most cases it is not necessary that every book has an owner…

Yes. In the case that any book must necessarily have an owner, then participation is **total**.

Advanced ER-Design 1

1. Weak entities

1.1 Definition and examples

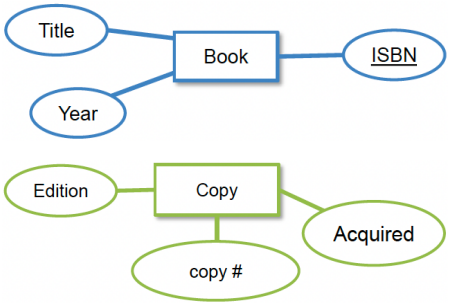
1.2 Design with weak entity

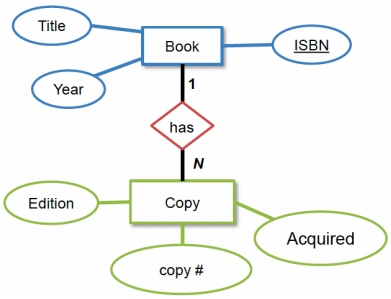
Entities: real world object or action distinguishable from the rest from which we are interested in some properties.

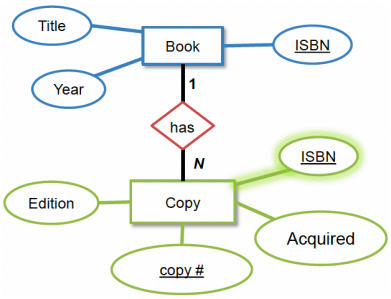
For each book we want to know the title, ISBN, publisher, edition and author. Consider also that for every book there are one or more 'copies' that share some features of 'book'.

Each 'copy' is a new entity. Instances of this new entity has also its own attributes.

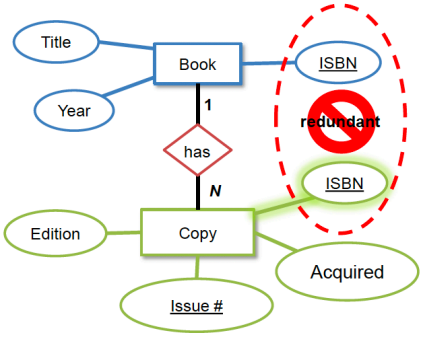
For each copy we want to know the copy number, date of acquisition and edition.







'Book' and 'Copy' are related through relation 'has' one-to-many cardinality. Since the copy are also books, can be uniquely identifie by the ISBN code and a new attribute #_copy

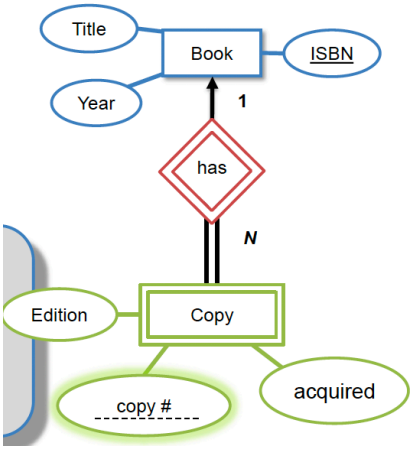


Observe that ISBN attribute is redundant, since book already contains that information

One solution is **not** to keep the ISBN attribute for entity 'Copy' but, then there would not be enough attributes in 'Copy' in order to distinguish each book:

→ We need ISBN to distinguish between the copy #3 of 'Don Quixote' and the copy #3 of 'La Galatea'

To solve this problem, we will deal with the relationship 'has' as a special relationship that provides extra information (in this case 'ISBN') needed to identify each book uniquely



Weak entities are entities that do **not** have **enough** attributes to define a PK while entities that have a PK are strong entities.

Example: suppose again books case. While a book is identified with a unique ISBN, each copy is identical to another copy. In a library may have many copies of a book (weak entities)

→ The solution is to associate to each of these weak entities a set of attributes that allow us to identify them without confusion

Discriminant: set of attributes which allow to distinguish between instances of the weak entity that depend on the same instance of strong entity. PK_WeakEntity is Discriminant U PK_StrongEntity

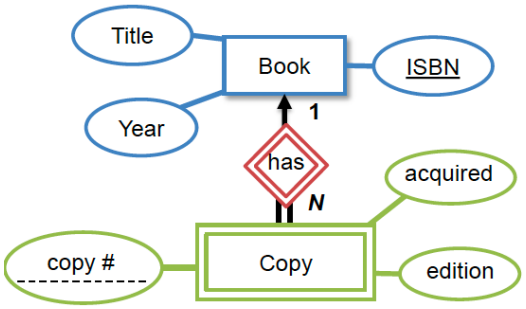
Desing ER with weak entity

Are represented by double lines in the ER diagram:

- **Double rectangle**: weak entities
- **Double diamond**: relation between a weak entity and its associated strong entity
- **Double relation line**: total participation of an entity in a relationship
- **Underline points**: key attribute (**discriminant**) of a weak entity

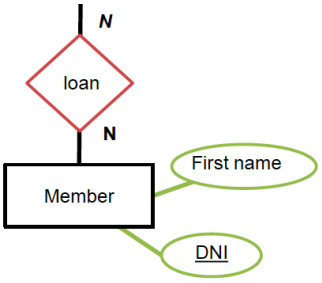
Example: a library lends books with the following conditions

- A user can take different loan books and a book can be taken for different members
- For each book there is more than one copy



→ Suppose some instances

Instances of copy entity can not be uniquely identified neither in the copy (N-1 relationship) nor in loan (NN relationship)



Book

Title	ISBN	Year
Don Quixote	12455	1605
Hamlet	23900	1603
....	....	....

FK

Copy

Purchased date	Edition	copy #	ISBN	DNI
02/04/2015	2010	0001	12455	3676373L
10/23/2014	2011	0002	12455	4748474P
12/12/1998	1990	0001	23900	3676373L

FK

Loan

DNI	copy #
3676373L	0001
4748474P	0002
3676373L	0001

FK

Member

DNI	First name
3676373L	Pepito Perez
4748474P	Vladimir Sanchez
....	....

Book

Title	ISBN	Year
Don Quixote	12455	1605
Hamlet	23900	1603
....	....	....

FK

Copy

Purchased date	Edition	copy #	ISBN	DNI
02/04/2015	2010	0001	12455	3676373L
10/23/2014	2011	0002	12455	4748474P
12/12/1998	1990	0001	23900	3676373L

FK

Loan

DNI	ISBN	copy #
3676373L	12455	0001
4748474P	12455	0002
3676373L	23900	0001

FK

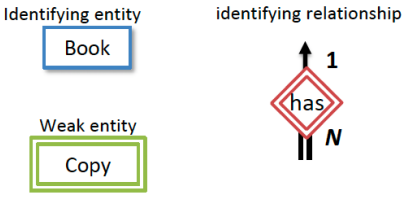
Member

DNI	First name
3676373L	Pepito Perez
4748474P	Vladimir Sanchez
....	....

This conflict is solved because 'Copy' is a weak entity and, therefore,,, the discriminant (copy #) and book PK do the role on Copy table.

Weak/Strong dependence and identifying entity

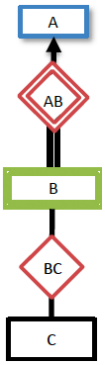
For a weak entity to be meaningful, it must be associated with another entity 'strong' or 'identifying' whose existence depends on.



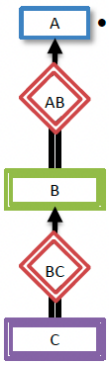
The relationship between the weak entity and its identifying entity is called '**identifying relationship**' and it is **many to one** (from the weak entity to the identifying entity) with **total participation** of the weak entity in the relationship

- The PK of a weak entity consists on the **union** of the PK of the strong entity and the weak entity discriminant
- Identifying relationship can **not** have descriptive attributes (these must belong to the weak entity)

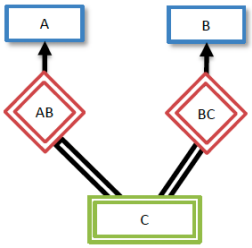
Relations with other entities:



A weak entity can participate in relationships a part of its **identifying relationship**



A weak entity can participate as '**strong**' entity with another relationship with weaker entities



It is **not** possible to have a weak entity with **more than one identifying** relation

Since participation in the relationship is total (each element of the weak entity is related to one of the strong entity)…
→ What will happen if an element is removed from the strong entity?
→ For example, one result will be in 'copy' if a book is removed from the entity 'book'? It makes sense a copy without a book?

The existence of the weak entity depends on a strong entity (subordinate) → can only exists if it is linked to a strong entity (existence dependency)

Book

Title	ISBN	Year
Don Quixote	12455	1605
Hamlet	23900	1603
....	....	....

Copy

Purchased date	edition	copy #	ISBN
02/04/2015	2010	0001	12455
10/23/2014	2011	0002	12455
02/13/2016	2005	0004	23900
10/23/2014	2011	0007	12455
....	....	....	...

In addition, **participation** of the weak entity in the relation is **total**: all entity instances are associated with a strong entity instance (otherwise could **not** be identified)
But, there may be instances of **strong entities not linked** with **weak entity** instances (**partial** participation)

Advanced ER-Design 2

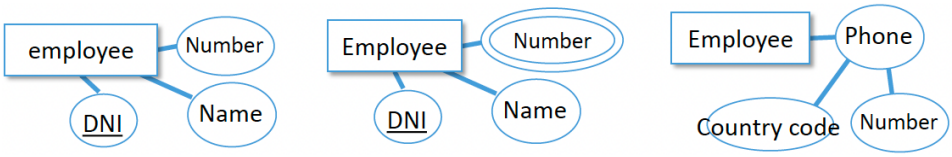
2. Design criteria

To meet the requirements of the data we have to decide:

- Using entities or attributes
- Using entities or relationships
- Relations degree (binary n-ary)
- Use of strong or weak entities)

Using attributes or entities

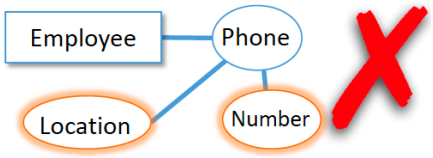
If an attribute has other attributes with different domains → entity
Example: employees in the company phone number



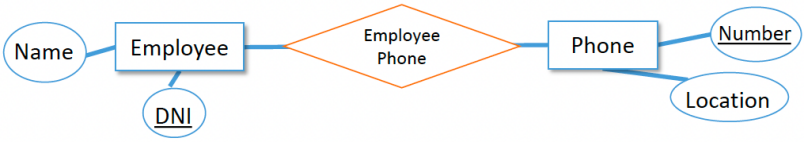
Attribute:
unique number per employee

Multivalued attribute:
more than one phone, but without own information (type, location)

Composite attributes:
is splitted on attributes of the same type



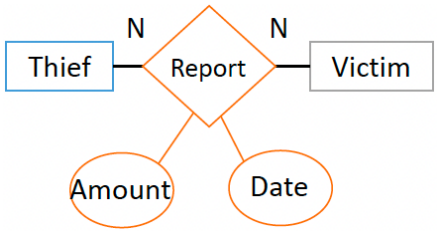
In the case of composite attributes, the pieces must be of the same type. A composite with attributes from different domains, probably should be an entity



<u>Using entities or relationships</u>
There is not a golden rule, but…

- Interrelationships is a '**verb**'. Action between two entities. Depending on their properties (attributes) is better representing it as an enitty
- If the cardinality is many-to-many (N-N) and we want a historical of the attributes of the interrelationship → better entity

Example: complaints
→ What about the previous complaints if someone denounces twice the same offender? *Can this design keep a record?*



Thief

DNI
7630556K
9894414J
9399993K
…

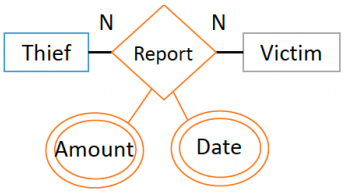
Report

ThiDNI	Amount	Date	Vict.DNI
7630556K	100	22-01-17	9930556P
9894414J	500	25-01-17	8499409K
9894414J	50	25-02-17	7363778Q
…	…	…	…

Victim

Vict.DNI
9930556P
8499409K
7363778Q
…

Suppose thief stealing to the same victim again… This new instance with the same pair of elements (thief and victim) will replace the previous amount and date. Is NOT possible to keep a record of robberies



Yes, we can keep a record of robberies, but only for one attribute (e.g. Date)

Thief

DNI
7630556K
9894414J
9399993K
…

Report

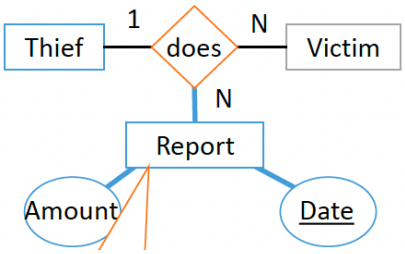
Thi.DNI	Amount	Date	Vict.DNI
7630556K	100	22-01-17	9930556P
9894414J	500 20 140	25-01-17 25-12-16 22-01-17	8499409K
9894414J	50 22	25-02-17 15-11-16	7363778Q
…	…	…	…

Victim

Vict.DNI
9930556P
8499409K
7363778Q
…

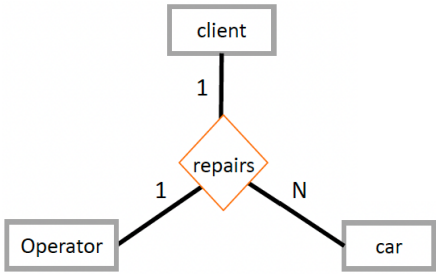
And if we want to save only the date and amount?
In this case we can not link the dates of complaints with amounts…

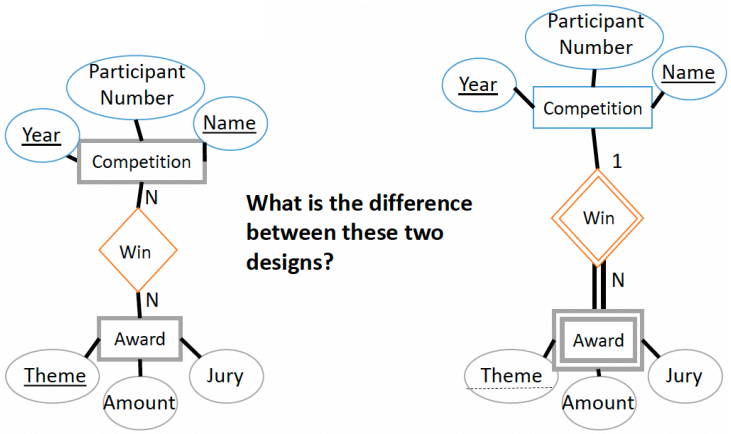
Can this other design keep a record? And if we keep only the date?
If we want to save the historic amounts stolen and the date, report should be an entity



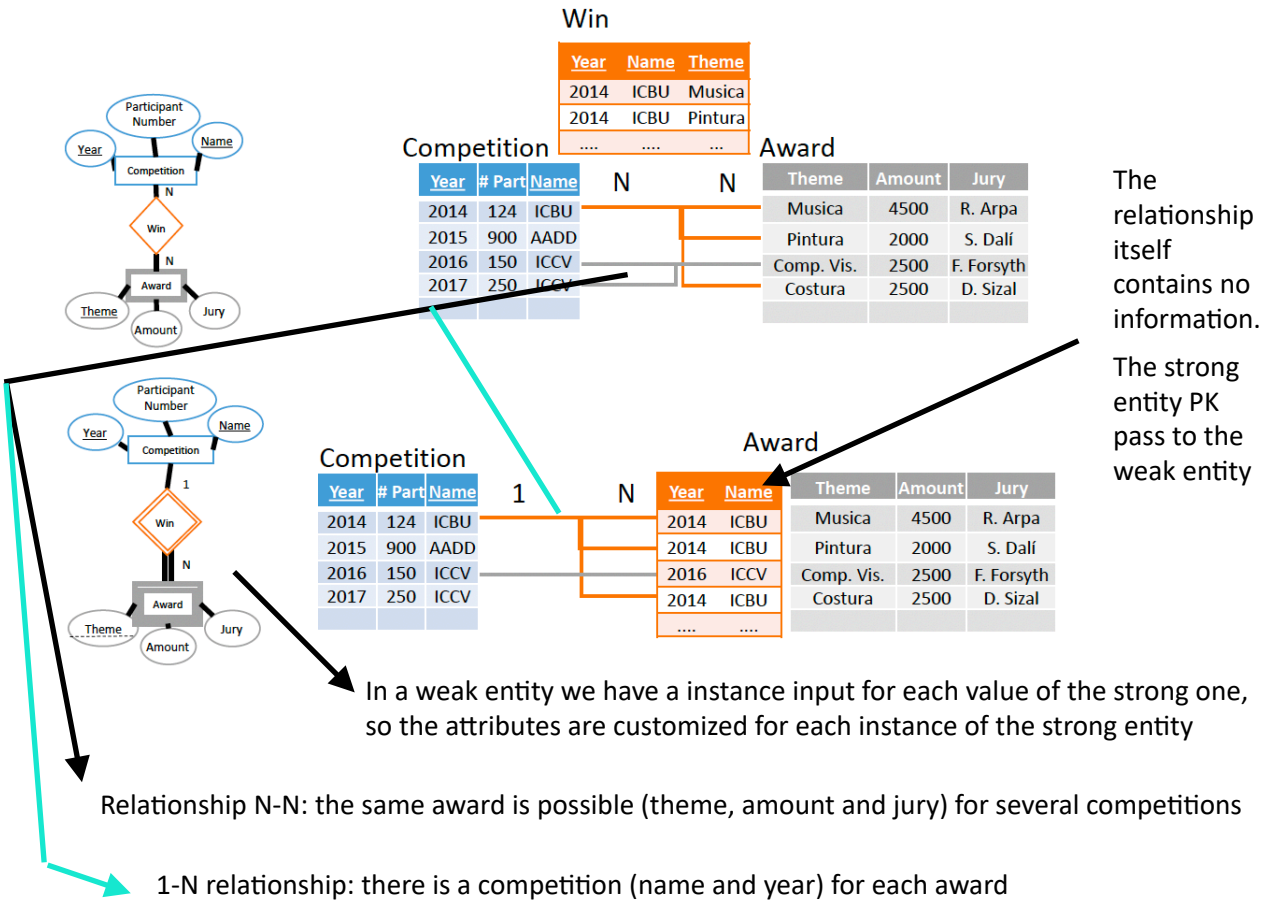
<u>Relations degree (binary n-ary)</u>
Interrelations of degree>2 are more specific but more rigid. We have to know all instances when are created

Example 2: a car repair shop wants to keep customer information, the operator and the car is being repaired
For example, when a new client comes with their car we have to assign a mechanic (operator)
→ It is not always practical: the customer may only want to become a member and oes not need a mechanic





Use of strong or weak entities
Example 3



Final criteria

- *Using entities or attributes*: Attributes of an attribute → entity
- *Using entities or relationships*: If N-N and historic → entity
- *Relations degree (binary n-ary)*: degree > 2 are more specific but more rigid (need to know more to define an instance)
- *Use of strong or weak entities)*: one instance for each instance of the strong entity/different attributes depending on instances of the strong entity → weak entity

Advanced ER-Design 3

3. Model ER extended

3.1 Aggregations

3.1.1 Specialization

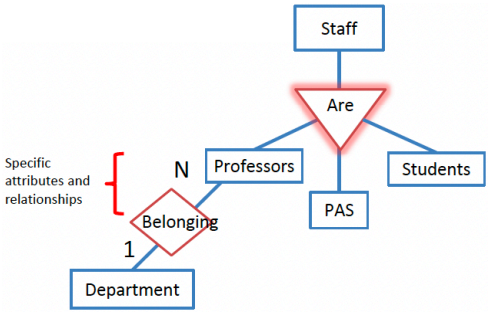
Partition of an entity in specific groups (↓-level entities)

An entity may include entities subgroups that are somehow different from other entities. The process of appointment of subgroups within a set of entities is called **specialization**.

Example: instances of 'staff' entity can be classified as one of the following:

- Employees
- Students
- Administration and services (PAS)

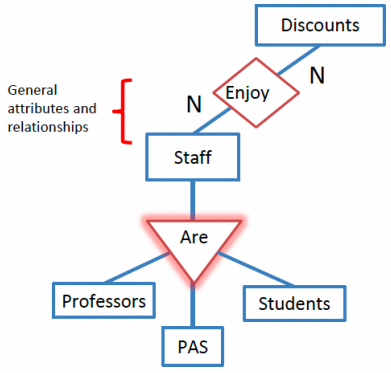
The specialization of 'staff' allows us to distinguish between entities 'staff' according to whether they correspond to the 'teachers', 'PAS' or 'students'. In general terms, the staff could be an employee, PAS, a student, a combination, or none of them.



3.1.2 Generalization

Process summarizing multiple entities into a ↑ level entity based on common characteristics (the reverse process of specialization).

There may be similarities between entity 'A' and entity 'B', for instance, several attributes that are conceptually the same in both entities. This coincidence can be expressed by **generalization**, which is a containment relationship between a ↑ level entity and one or multiple ↓ level entities.



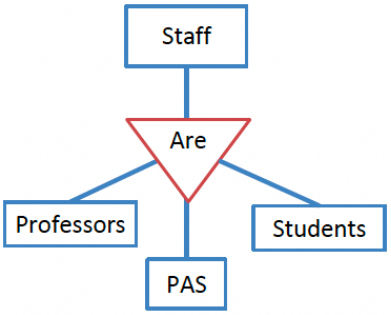
The generalization is used to merge a set of entities that share the same characteristics on a set of ↑-level entities.

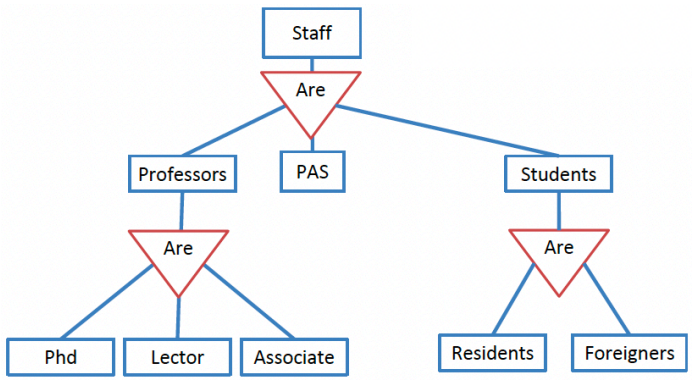
Allows to emphasize the similarities among ↓-level entities and to hide the differences.

Economic representation: shared attributes are not repeated

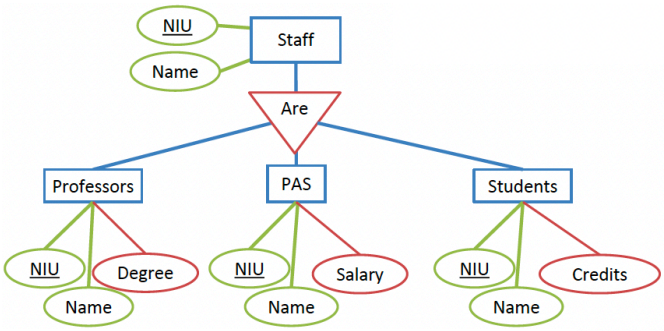
Common features

- Grouping of entities with common attributes into a ↑ level entity
- Eliminate redundancies
- The ER diagram itself does **not** distinguish between specialization and generalization
- Attributes of the ↑ level entities are inherited by ↓ level entities
- For example: students, teachers and PAS inherit 'staff' attributes



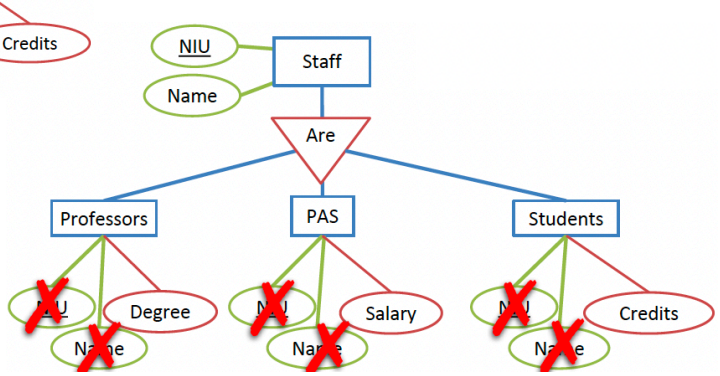


We can apply the specialization/generalization more than once to refine a design scheme



Each of these 'staff' sub-entity is described by a set of attributes that includes all the 'staff' attributes adding possibly some additional attributes.

Attributes and relationships of ↑ level entities are inherited by the ↓ level entities (in particular CP) and it is **not** necessary to specify them



Restrictions

Database designer can decide to place certain restrictions on a particular specialization or generalization

Participation:

- Total: all the ↑ level entity instances must belong to a ↓ level entity
- Partial: there are instances of the ↑ level entity that can not be classified (default)

Observation → generalizations are usually of total participation

Participation overlap:

- **Disjointed**: each instance of the ↑ level entity belongs to only one ↓ level entity. For example, a member of 'staff' cannot be student and PAS
- **Overlapped**: some instance of the ↑ level entity can belong to more than one ↓ level entity. For example, a 'staff' member can be both student and PAS

Assignment:

- **By condition**. Some condition is mandatory over some attributes of the ↑ level entity

Example:
→ Students: people that are enroled to some course
→ Professors: people that teach some course
→ PAS: no teaching hired staff

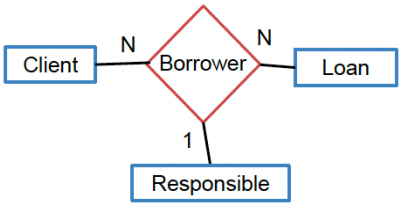
- **By user**: instance manually assignment

3.1.3 Aggregation

Grouping of relationships and participating entities into a new entity

- A limitation of the ER model is that it **cannot** express relationships between relationships
- To solve this problem, we create **aggregations** that are abstractions through which relationships are treated as ↑ level entities

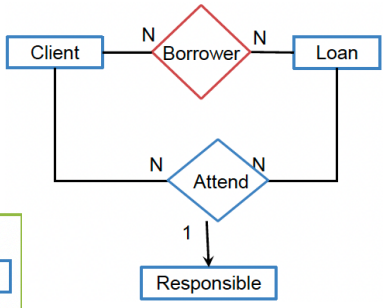
Example 1: a bank wants to keep information on loans, their clients and a bank responsible for each loan. The relationship client-loan is N-N.
→ *We need to assign the responsible when the loan is accepted*

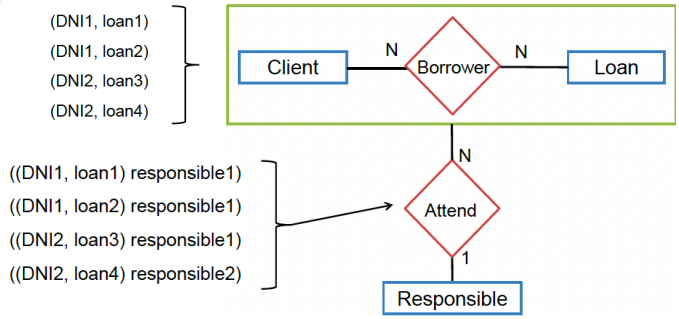


Possible solution… **not as it seems**

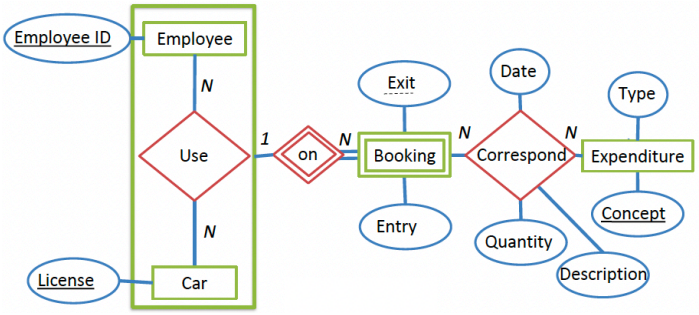
We must ensure that every instance of 'attend' is also an instance of 'borrower' and this is not reflected in this diagram (Trap Connection)

A solution will be… a relationship between 'responsible' entity and borrower relationship





Aggregation is a new entity that can be treated as a single one on the rest of the design



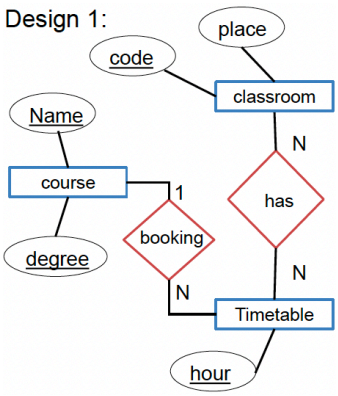
Example 2

We want to manage ETSE classroom timetable booking and know the availability. For each school day, reservations are *fractions of an hour* from 9 to 20. For every classroom we keep the *code*, *location* and *capacity*. For each course we want to know the *code*, *name* and *degree* to which it belongs.

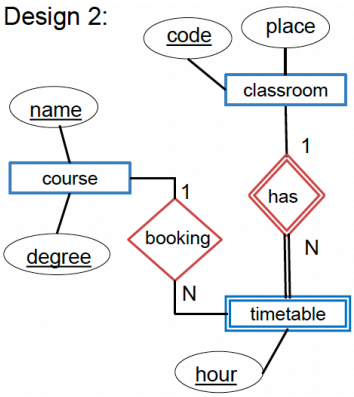
a. There are only morning and only afternoon scheduled classrooms
b. All classrooms have the same timetable

We propose 2 designs:

Design 1:



Design 2:

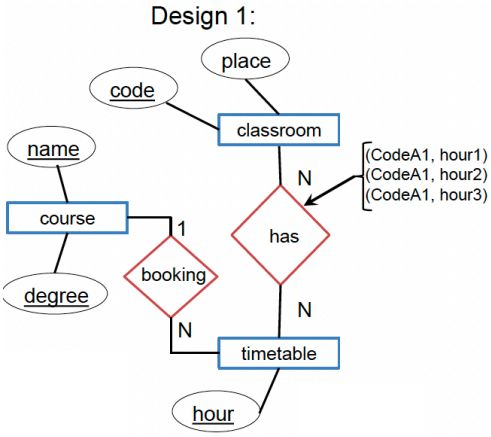


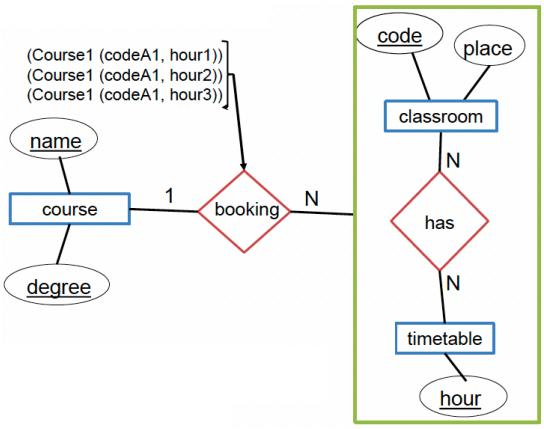
Which is the difference between these 2 designs? Can we know timetable for requirement a)? How?

If timetable is a strong entity we have generic (list of all slots) timetables, so every classroom timetable (a. change depending on the classroom) are in the relationship.

With this design manage availability is **not** allowed

Design 1:

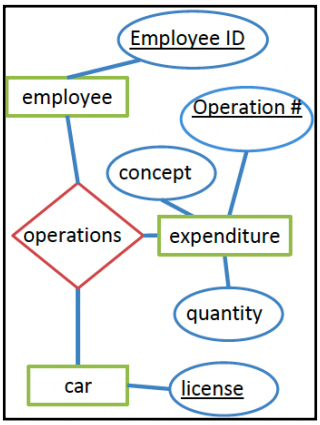




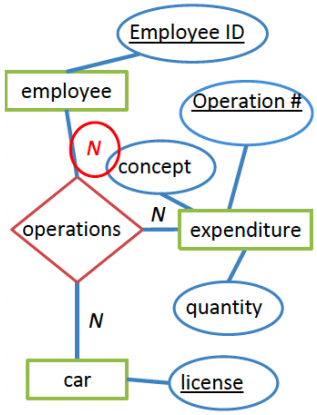
An aggregation would manage the hours available, although it is more complex

Example 3: a company has a fleet of **cars** for **employees** to perform **operations** outside their workplace. In addition, each time an employee uses a car generates a sheet with all associated **costs**. Particularly, for each operation…

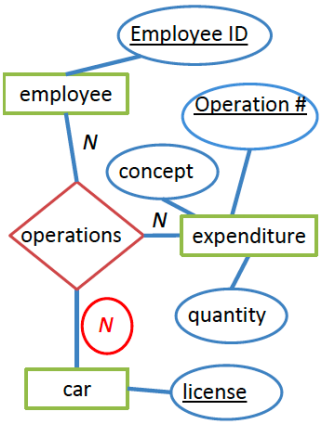
- May be required more than one worker
- No limits on the number of cars in use
- The cost sheet should indicate all associated costs
- One of workers involved in the operations (the manager) is in charge (before use) to reserve cars (among those available), and once the operation has been completed, delivering the sheet with all the costs



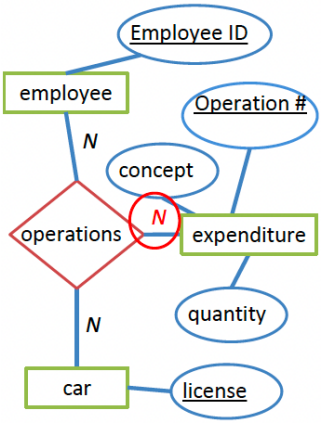
May be required more than one worker



No limits on the number of cars in use



The cost sheet should indicate all associated costs



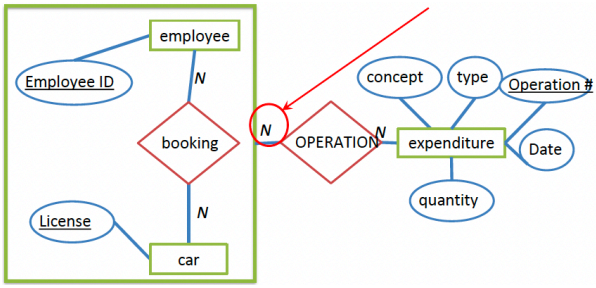
Problems

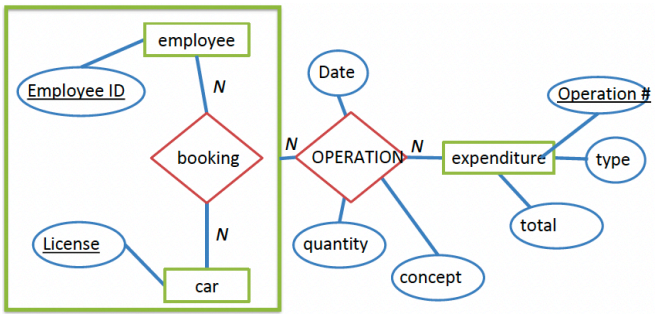
→ The ternary relationship does **not** allow booking first and then to pass expenses

In this case should be allowed operations between employee and car before issuing costs (requires to be independent 'operations' of 'spending')

→ Costs associated with a real operation correspond to the same number of operation (Operation #) are not checked

In one operation, employee can take more than one car



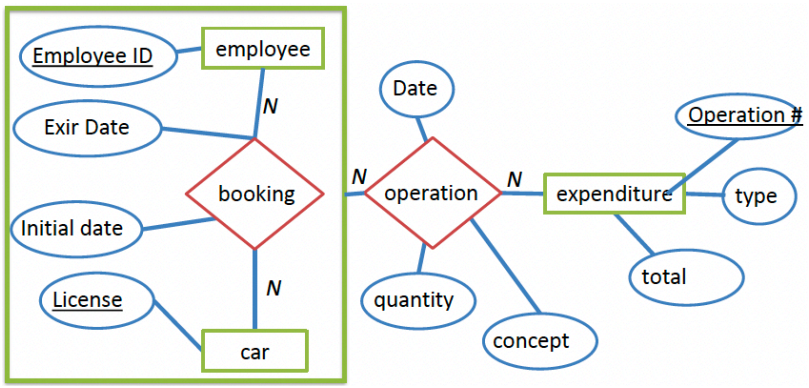


This design still has problems…

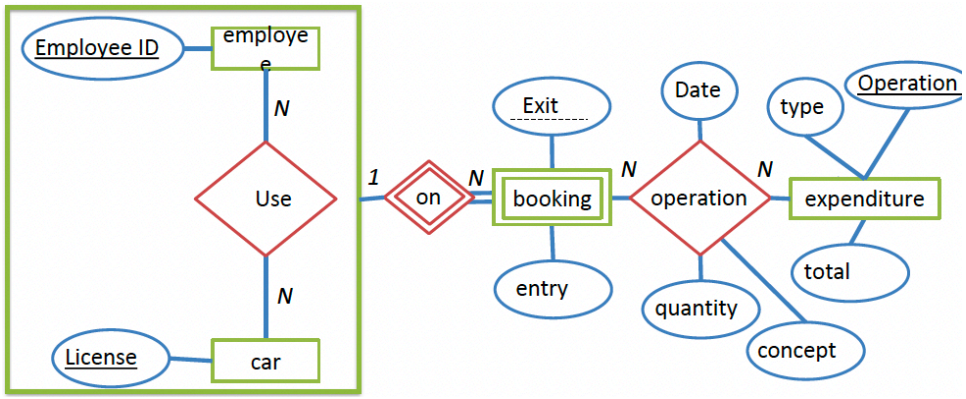
- *What happens when an employee takes the same car for different operations?*

Solution: we should add the **date of reservation** and make a change in design

Solution 1



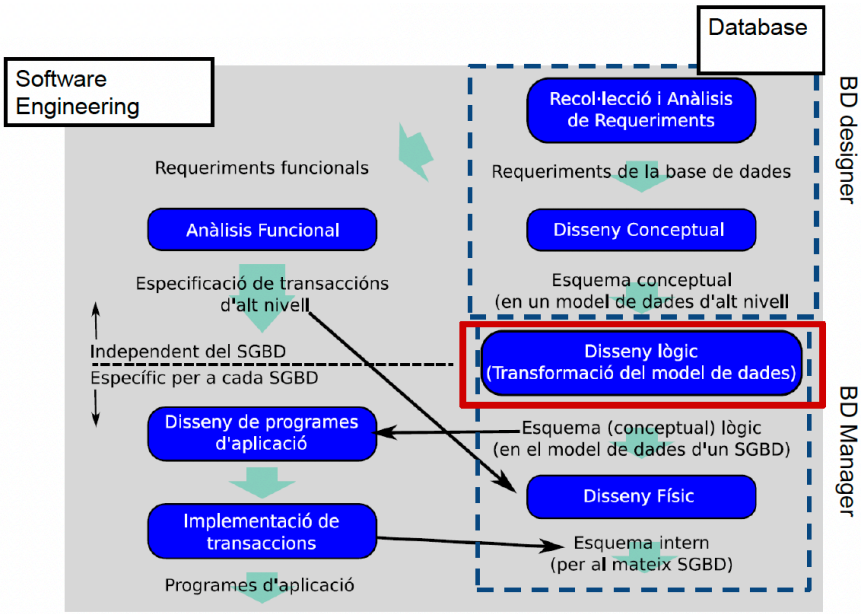
Solution 2



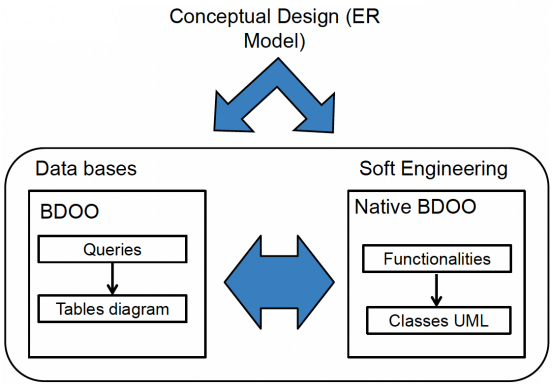
Relation design

1. Introduction

Phases design



Logical design



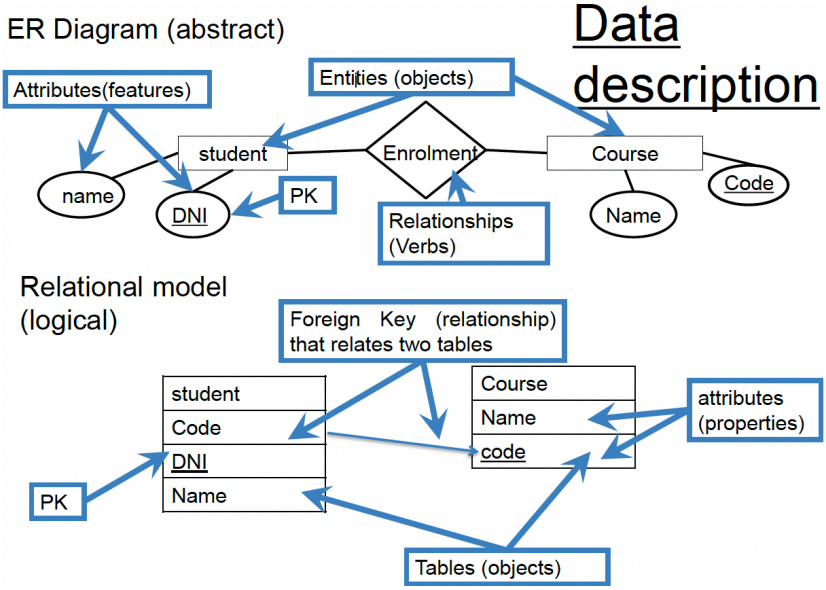
2. ER design to Relational Model

2.1 Relational Model

- Data ↑ level description that guarantees its integrity
- Is the unique theorical model implementable with basic rules
- It's the DBS reference although another models exist

Components

- Data structure: description by relations/tables
- Integrity rules: rules to ensure the queries consistency
  - Entity (Table, PK)
  - Referential (relations between tables, FK)
- Operators to acces data by content:
  - Relational algebra
  - Relational calculus

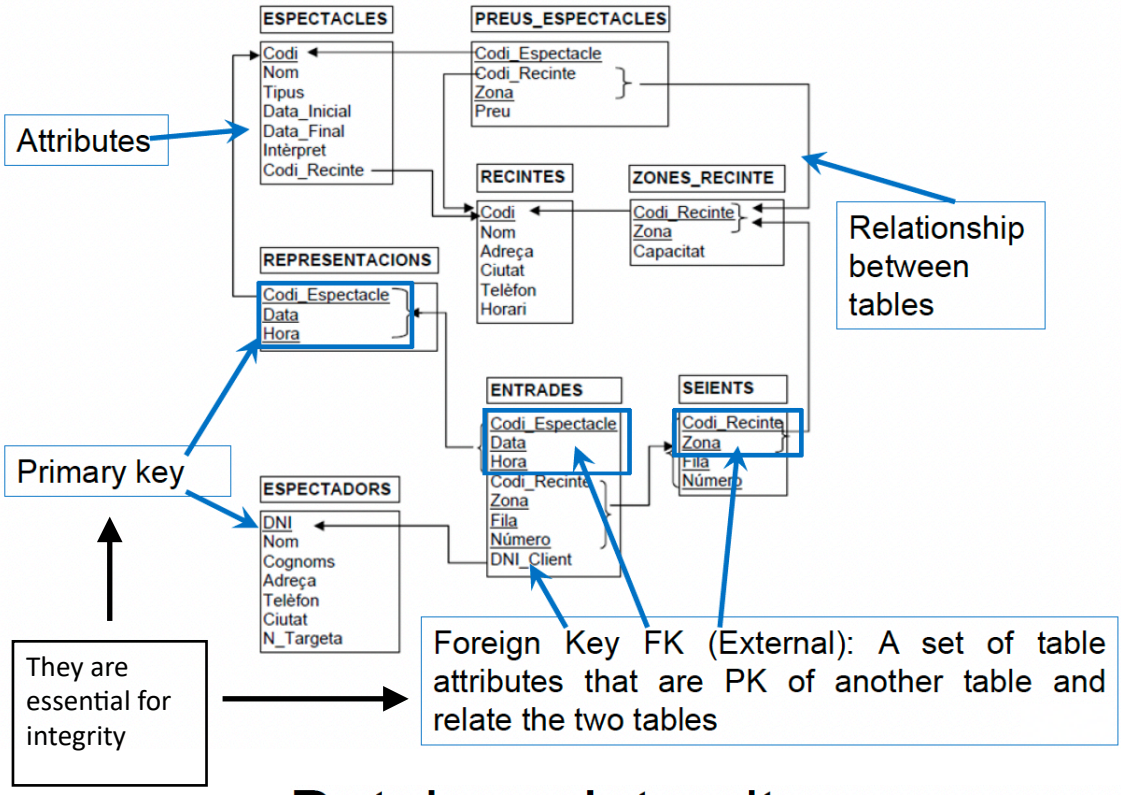


Relational components

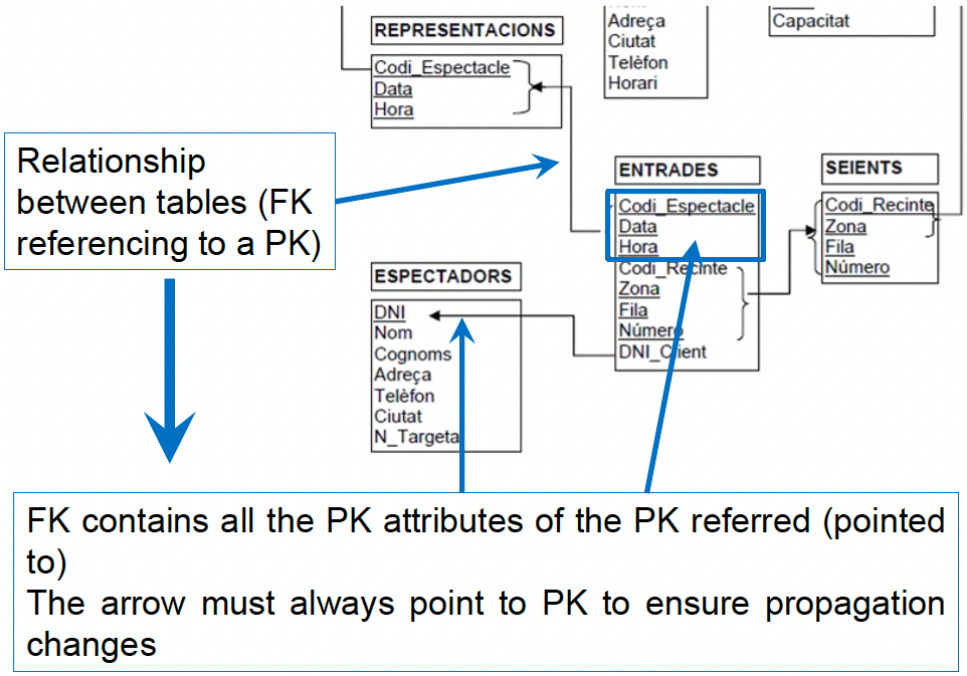
Objects		Relation (Table)
Features	→	Domain (allowed values of attributes)
Related information		Related database (set of referenced tables)

Database

Collection of related tables (referenced, linked) together

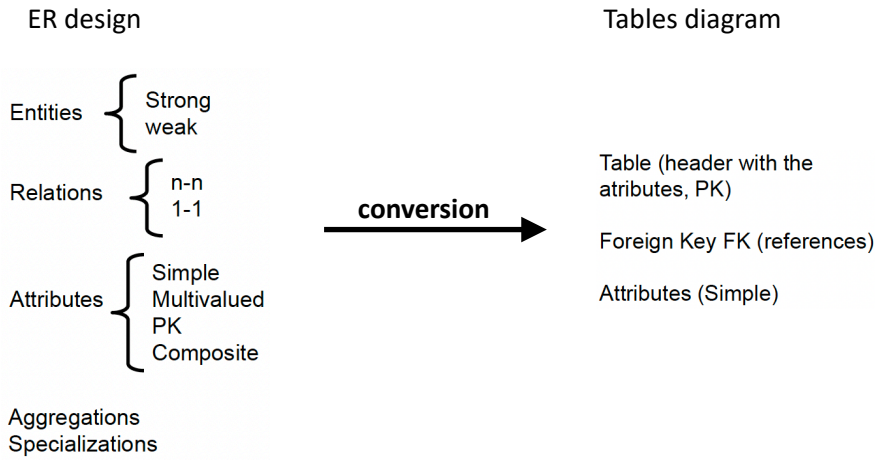


Database Integrity



2.2 Tables Diagram from an ER Design

Structures



Conversions

Entities → Table

- **Strong**: entity PK
- **Weak**: weak entity PK + FK with value strong PK

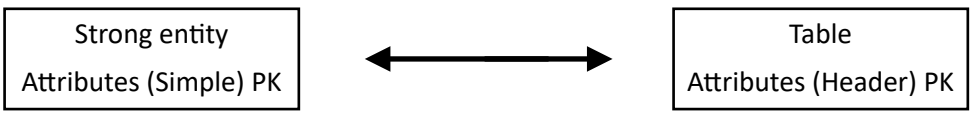
Relationships

- **Binary 1-n**: FK on n-side entity with PK value on 1-side entity
  - Ex: Table A - PK A Table B - PK B + FK A
- **Binary 1-1**: FK on 1-side entity with PK values on 1-side entity (+ FK uniqueness constraing)
  - Ex: Table A - PK A + FK B Table B - PK B + FK A
- **Binary n-n**: table with PK union n-side participating entities PK's + FK for each participating entity
  - Ex: Table A - PK A Table B - PK B Table C - PK(FK A + FK B)
- **Ternari**: table with PK union n-side participating entities PK's + FK for each participating entity

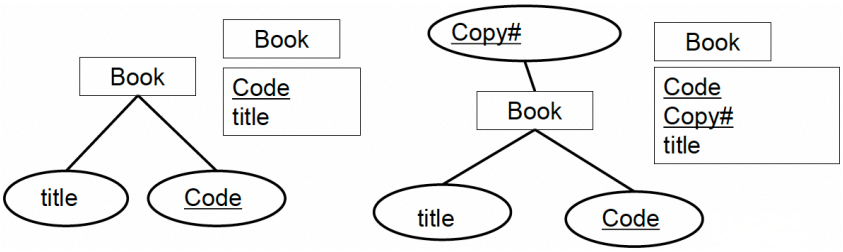
Attributes

- **Simple**, **composite** → attributes
- **Multivalues**: table with attribute PK and entity PK + FK with value entity PK

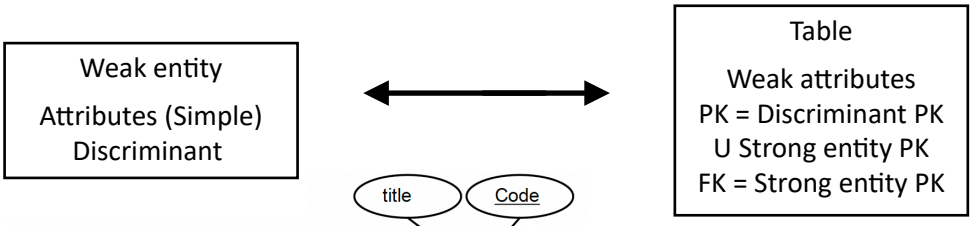
Strong entities



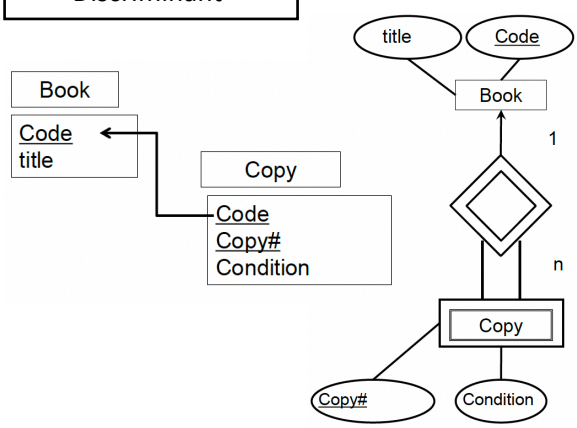
Example:



Weak entities



Example:

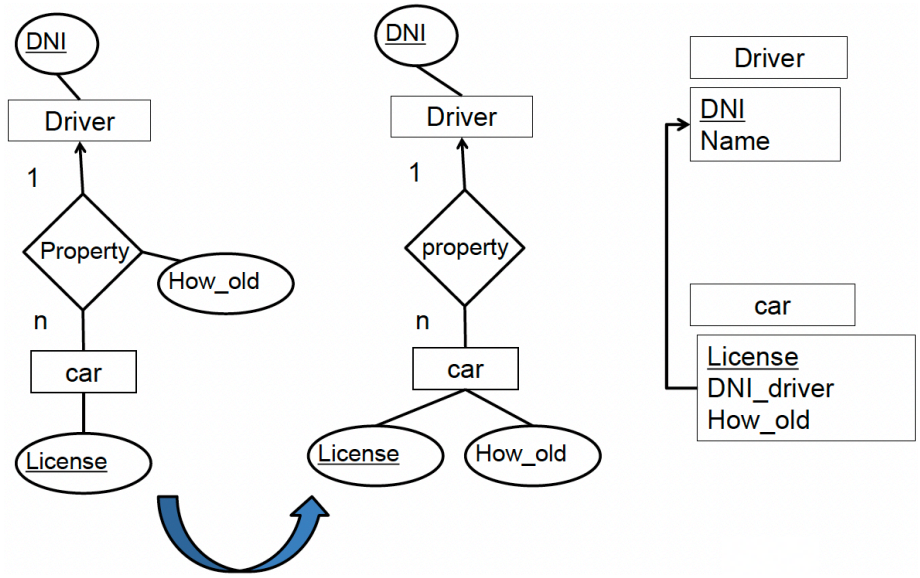


Relationships

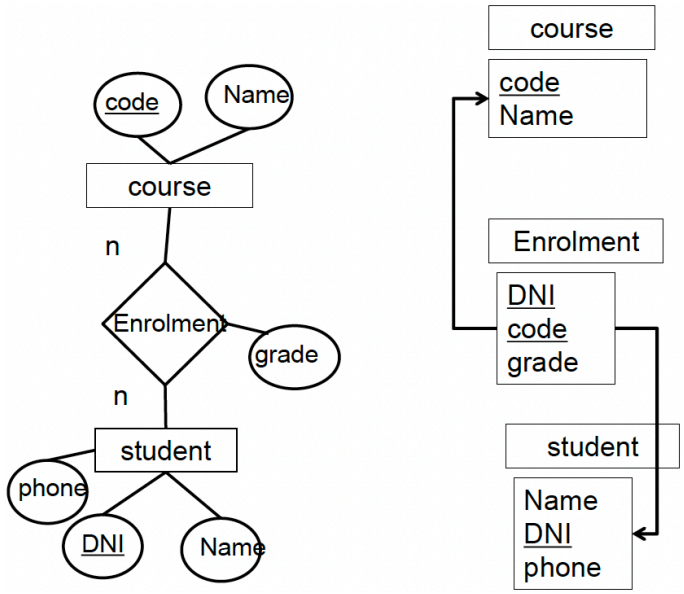
They become a table or a relationship between interrelated entities depending on the cardinality:

- Cardinality 1-N: attributes are passing to n-side entity and the interrelationship is converted to relationship between tables (FK to the n-side entity)
- Cardinality N-N: relationship is converted to a new table with FK linking two related tables. The new table PK is the union of the related tables PK

Example cardinality 1-N



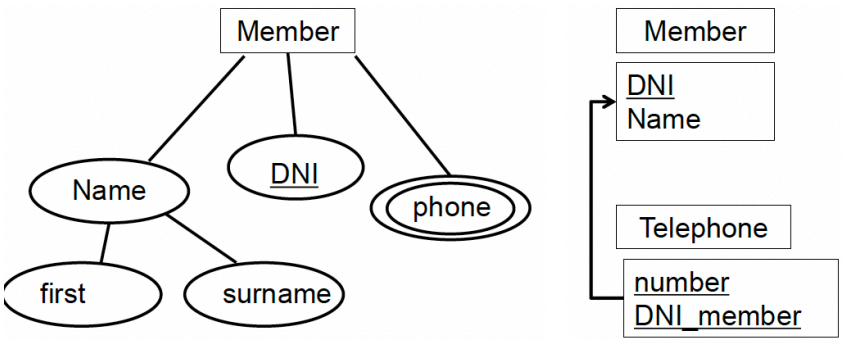
Example cardinality N-N



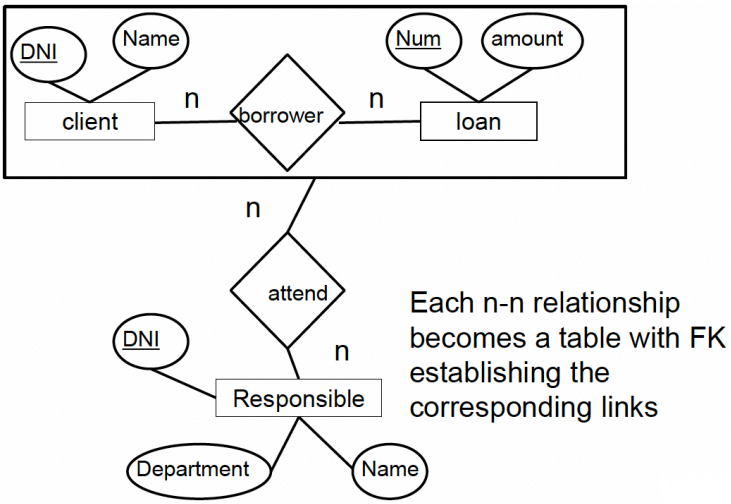
Attributes

Simple and composite attributes have direct translation into simple attributes with a certain domain

Multivalued generate a table with FK:

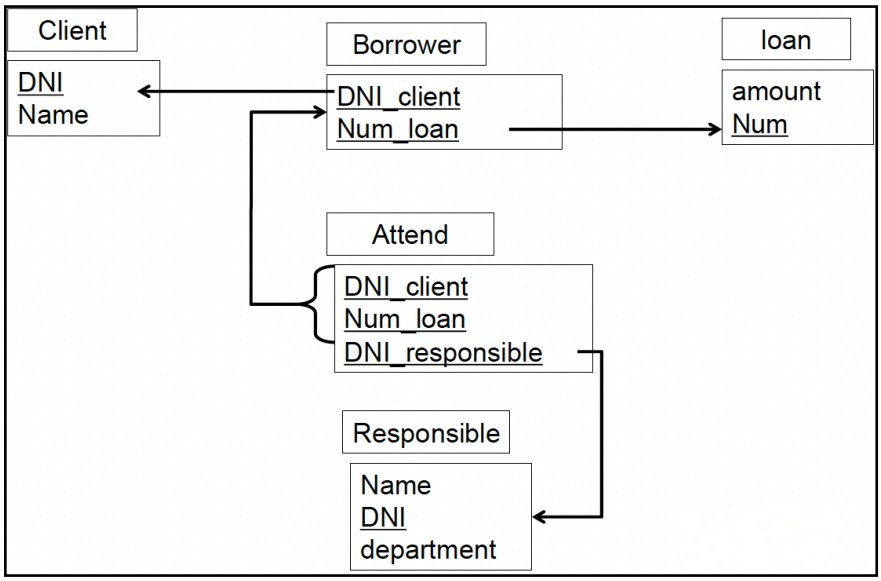


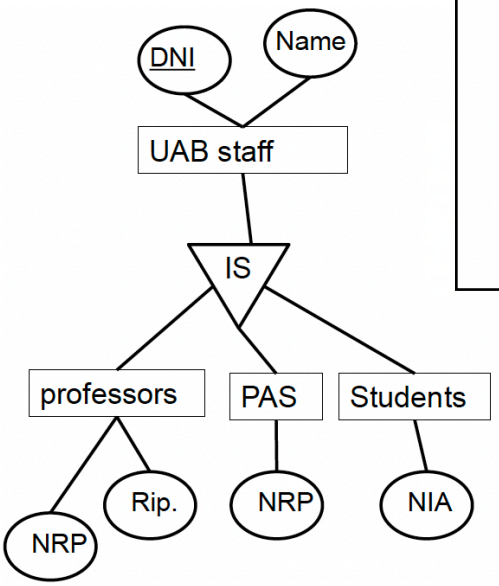
Aggregations



Each n-n relationship becomes a table with FK establishing the corresponding links

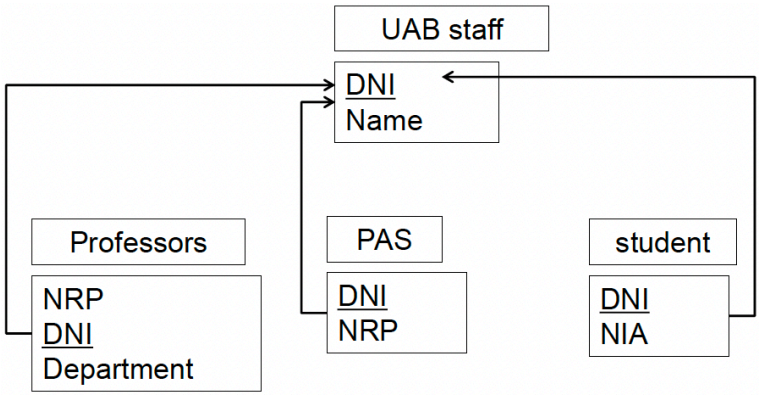
Example

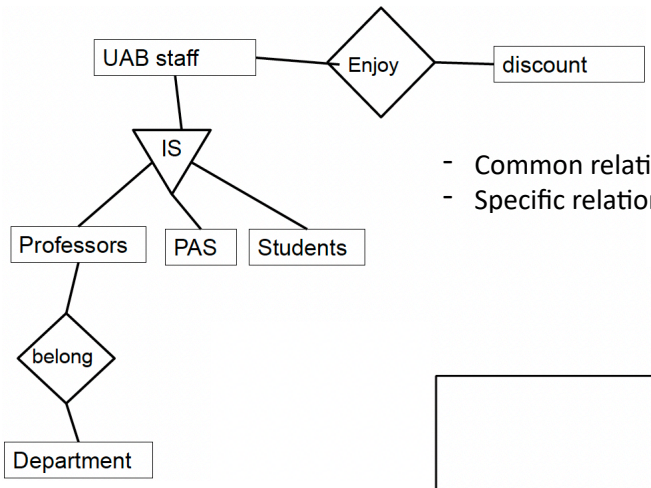




Specializations
- Each entity becomes a table
- ↓ level entities have FK linking them to ↑ level entity

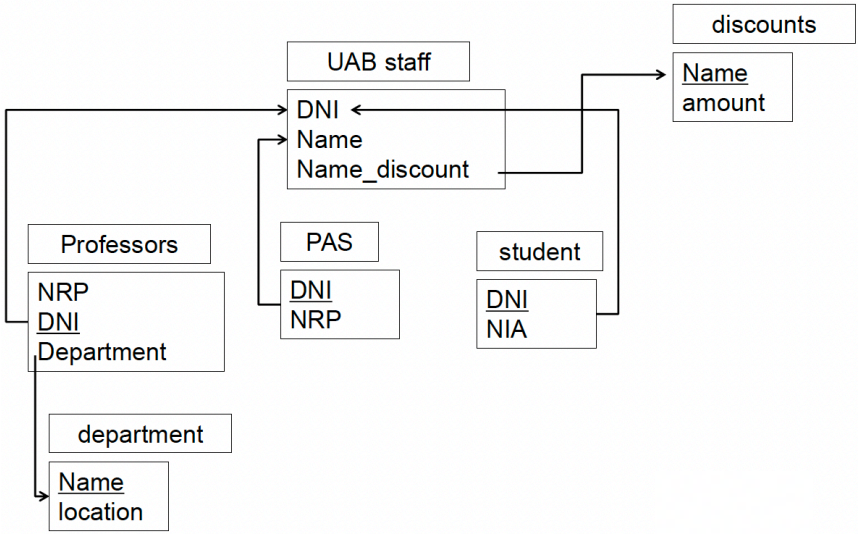
Example





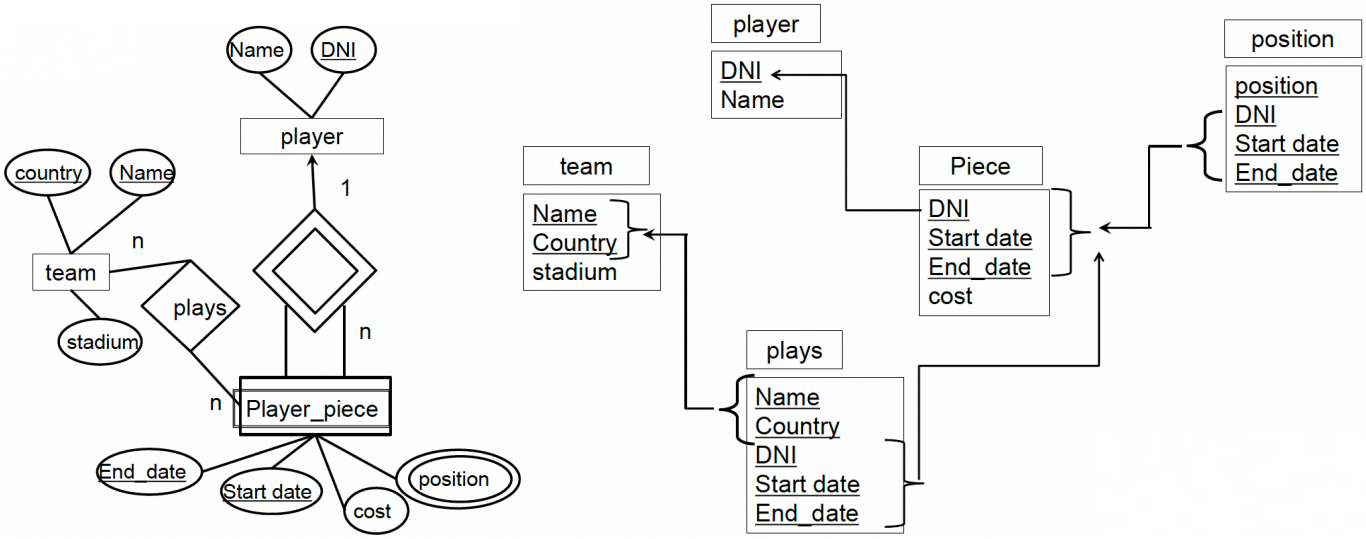
- Common relationships are specified through ↑ level table
- Specific relationships only by ↓ level involved tables

Example

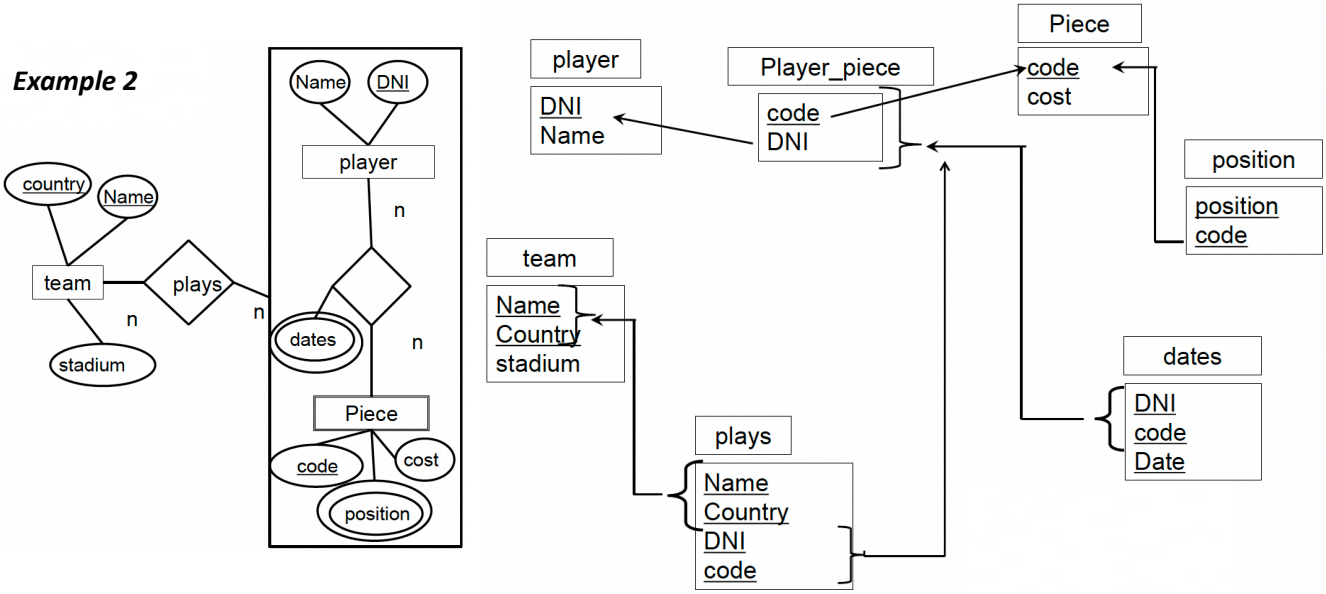


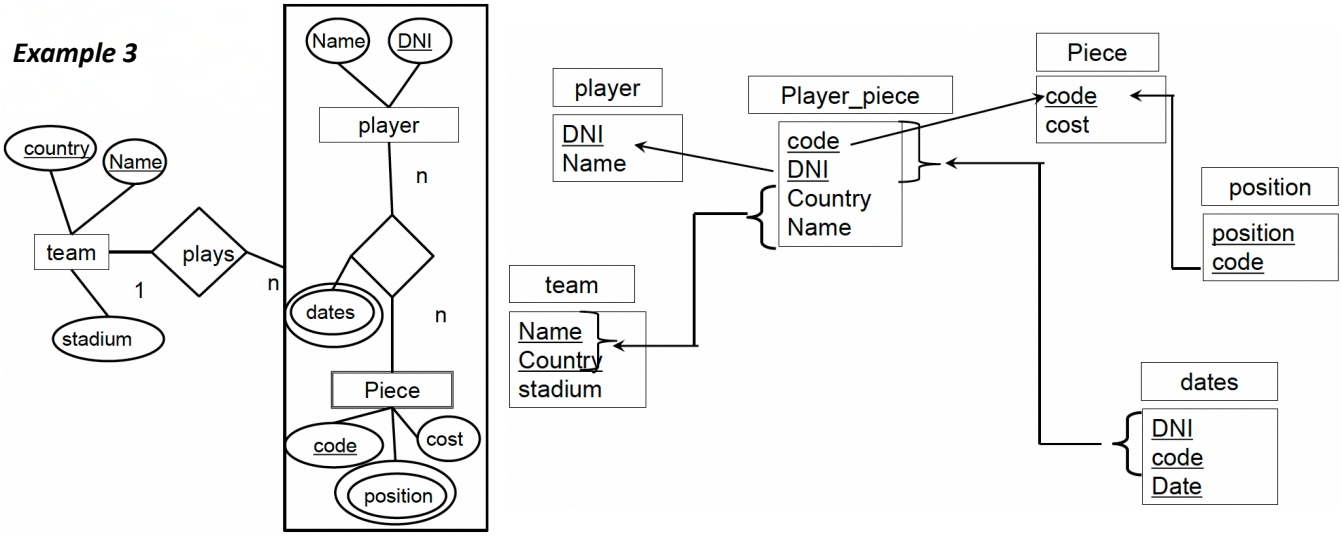
2.3 Examples

Example 1



Example 2





Cuando es **N-N**, una nueva tabla (new relational table) se añade
Cuando es **1-N**, la table N se convierte en la relation
→ la **PK** de la 1 table, se va a la N table y se convierte en **foreigner (FK)**

Data Integrity
Integrity rules
1. Basic concepts
Rules (constraints) used to ensure validity and accuracy of a data set. Most are incorporated into the DBMS.
Two levels: specifics and generals

Specific rules:
- Specific to the type of data that each BD contains
- They are defined through specific domains
- Examples:
  - Weight is positive
  - Boarding gates are identified by a letter
  - Number of pieces per box are multiple of 100

General rules:
Rules applicable to any type of data and BD. Three rules for data integrity:
- **Domain** integrity
- **Entity** integrity
- **Reference** integrity

Domain integrity
Every value of an attribute must belong to the domain on which we defined it. Many DBMS (Oracle) do not have it incorporated due to efficiency issues → application-level checking

Entity integrity
1. **Motivation**
- Tuples (rows) in a table represent objects/individuals in the real world
- The real-world objects are identifiable and distinguishable from other (uniqueness)
- As a representant of a real object, each tuple is **unique** (although some attribute values match those of another tuples)

2. **Primary Key**
- Candidate keys (CK): minimum set (there are not redundancies) of attributes that uniquely identify each instance
- Primary key (PK): CK chosen by the designer

Example 1

- Requirements: the ISBN is a code that is assigned according to the title, author and book publisher
- The barcode is a library internal encoding that is assigned to each copy

Identify CKs and PK →

Copy
ISBN
Copy #
Barcode
MaxDuration
Penalisation

3. **Entities integrity**

No component of the PK can be **null** (missing information). NULL is equivalent to an existint object without identification or with an unknown identification

Remarks:

- DBMS checks integrity when the PK is defined
- If you do not specify PK, some DBMS assign PK automatically. Enter an extra attibute to enumerate the tuples → Automatic numbering is **not** recommended:
  - It depends on the insertion order (Independence)
  - It has no semantic meaning (Domain)
  - Enter redundancies and dependencies between data

4. **SQL Syntax**

```
CREATE TABLE <nameT> (
<nameA1>  <domain1> <values>,
....
<nameAk1><domaink1> NOT NULL,
<nameAk2><domaink2> NOT NULL,
....
<nameAN>  <domainN> <values>,
CONSTRAINT nameT_PK   PRIMARY KEY (nameAk1,nameAk2)
);
```

Example 1

Simple PK

```
CREATE TABLE RECINTES (
 code NUMBER NOT NULL,
 name VARCHAR2 (50)
 Address VARCHAR2 (50)
 Telephone NUMBER
 Timetable VARCHAR2 (50)

 Constraint RECINTES_PK PRIMARY KEY ( code ) )
```

Example 2

Compund PK

```
CREATE TABLE PREUS_ESPECTACLES (
 Codi_Espectacle NUMBER NOT NULL,
 Codi_Recinte NUMBER NOT NULL,
 Zone VARCHAR2 (20) NOT NULL,
 Price NUMBER,
 Constraint Preus_Espectacles_PK
 PRIMARY KEY ( Codi_Espectacle, Codi_RecinteArea) )
```

Reference integrity

1. **Motivation**

Relational databases are a set of referenced tables

Student

NIA	Name	DNI	Tel
1	Pepe	46956514	657545454
2	Joan	56056512	666666666

Subject

PK_Sub	Name	Classroom
100	BD	1011
202	IA	1013
305	SO	2010

PK_St	PK_Sub
1	100
2	100
2	202
2	305

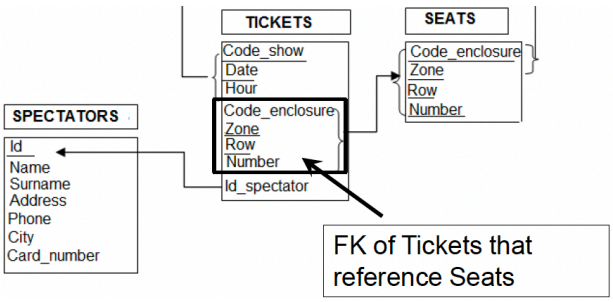
Enrolment

References to tables containing data

2. **Foreign Key (FK)**

Set of attributes of a relation R2 whose values are the same values as another PK takes in a relationship which refers to R1.

Obs: PK and FK must be of the same domain



Concepts

- Referenced **relation**: table containing PK
- Referenced **tuple**: tuple where exists PK value of FK

Student

NIA	Name	DNI	Tel
1	Pepe	46956514	657545454
2	Joan	56056512	666666666

Subject

PK_Sub	Name	Classroom
100	BD	1011
202	IA	1013
305	SO	2010

Enrolment

PK_St	PK_Sub
2	100
1	100
2	202
2	305

Referential relationship: value containing the FK

3. **Reference integrity**

Basic concepts

→ There may be PK value of R1 that are **not** referenced by any FK of R2

Examples:

- There can be subjects with no students enrolled
- Students can be **not** enrolled

→ A value of FK in a R2 relation referencing to a R1 relation is only allowed if this value appears in the PK of one of the tuples of R1

Examples:

- It makes no sense that a student enrols for a subject that does not exist
- We can not sell tickets for a flight to the Moon

→ Definition

- The DB can **not** have values (NO_NULL) for FK non-PK of a tuple of the referenced relation
- If R2 refers to R1, R1 must exist
- The DBMS makes checking whether FKs are specified

4. **SQL Syntax**

Constraint <nameFK> **FOREIGN KEY** (<AttributseFK>) REFERENCES <nameReferencedTable> (<AttributesPKReferencedTable>)

NULL [NOT] ALLOWED
DELETE OF < nameReferencedTable > <effect>
UPDATE OF < AttributesPKReferencedTable > < effect >

Where <effect> can be
RESTRICTED |
CASCADES |
NULLIFIES

Example

Fks are defined once all tables have been created using the following syntax:

ALTER TABLE <nameTable> ADD CONSTRAINT

```
ALTER TABLE AUXILIAR ADD CONSTRAINT
AUXILIAR_PERSONAL_FK FOREIGN KEY ( SS ) REFERENCES
PERSONAL ( SS ) ;

ALTER TABLE TICKETS ADD CONSTRAINT TICKETS_TRIP_FK
FOREIGN KEY ( Code_ticket, Passenger ) REFERENCES
TRIP( Code_ticket, NIF_Passenger ) ;
```

5. **Referential Diagram**

R1 ← **R3** → **R2**

Refered relation (contains PK) | Refered relation (containing PK) | Refered relation (contains PK)

Relations can be labeled with FKs:

Student ← PK_St **Enrolment** PK_Sub → **Subject**

A relation can be referenced and referential

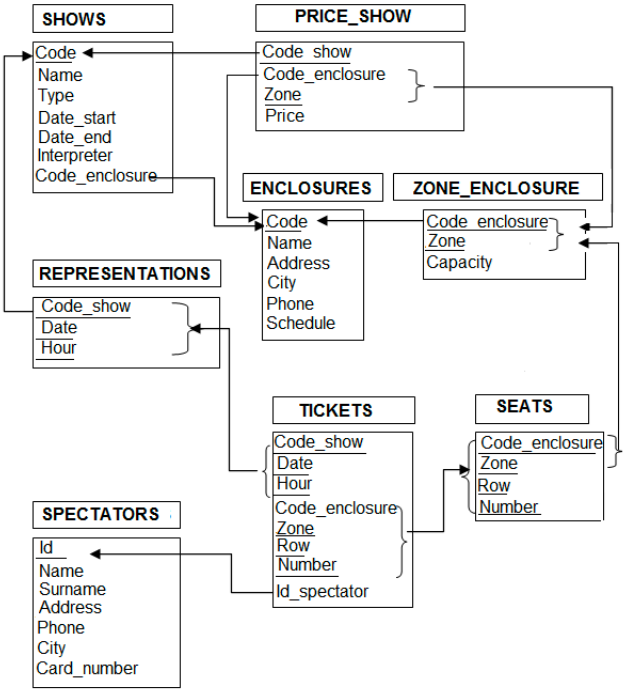
b **a**

R3 → R2 → R1

Referential path

- Referential chain constraints
- Determines the order to follow when it modifies (Inserts, Deletes) DB

$$R(n) \longrightarrow R(n-1) \longrightarrow \cdots \longrightarrow R2 \longrightarrow R1$$



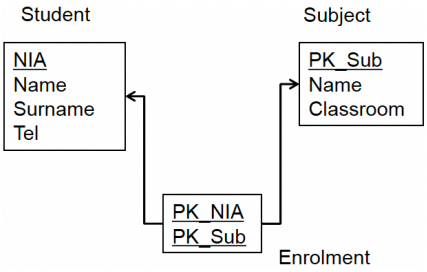
Referential Diagram and Insert/Delete order:

1. A enclosure
2. A zone enclosure
3. A seat

Propagation constrains

1. **Motivation**

What happens if we delete a referenced/referential tuple?



- FKs **determine** the link between tables
- Modifying a referenced tuple affects **ALL** those referring to it

Rules for FKs to guarantee reference integrity

- NULLS acceptation
- Tuples deletion (Delete)
- PK modification (Update)

→ Restrict (**Restricted**)
Propagate (**Cascade**)
Accept nulls (**Nullifies**)

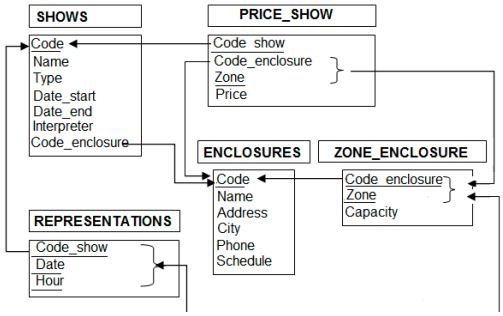
2. **Null allowed** (**Nullify**)

NULLS acceptation

Admitting NULL in a FK not always makes sense. It depends on the meaning of the DB and the data policy to be modelled.

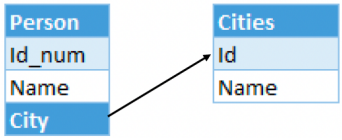
→ SQL: by default FK supports NULLs

A show without a venue could make sense in an event organization company.



NULL values meaning

FK can accept NULL values → *What does it mean?*

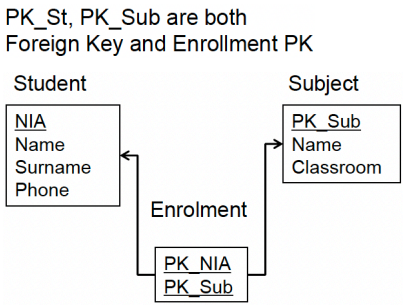


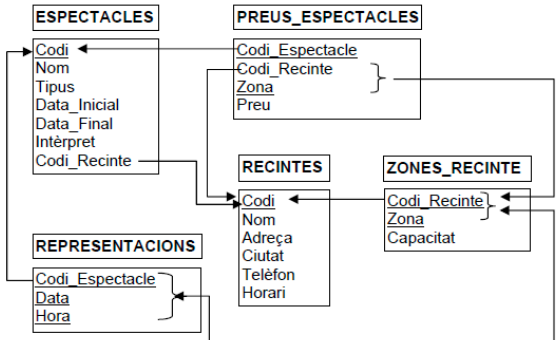
Id_num	Name	City
234234234	Lucia	002
453454534	Paul	~~003~~ NULL
585765837	Ernest	001

Id	Name
001	Nápoles
002	Barcelona
~~003~~	~~Paris~~

If FK consists of more than one attribute, they must be entirely NULLS or NOT NULLS

A FK may be part of the PK of the referential relation…





3. **Delete**

1. **Restricted**

- Delete S2 supplier from S Relation ⇨ **Not possible**

S

S#	Snom	Zona	Ciutat
S1	Salazar	20	Londres
S2	Jaime	10	Paris
S3	Bernal	30	Paris
S4	Corona	20	Londres
S5	Aldana	30	Atenas

SP

S#	P#	Cant
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P3	200
S4	P2	200

P

P#	Pnom	Color	Pes	Ciutat
P1	Femella	Vermell	12	Londres
P2	Pern	Verd	17	Paris
P3	coixinet	Blau	17	Paris

2. **Cascade**

Delete S2 supplier from S Relation ⇨ **Deleted in S, SP**

S

S#	Snom	Zona	Ciutat
S1	Salazar	20	Londres
S2	Jaime	10	Paris
S3	Bernal	30	Paris
S4	Corona	20	Londres
S5	Aldana	30	Atenas

SP

S#	P#	Cant
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P3	200
S4	P2	200

P

P#	Pnom	Color	Pes	Ciutat
P1	Femella	Vermell	12	Londres
P2	Pern	Verd	17	Paris
P3	coixinet	Blau	17	Paris

Delete S2 supplier from S Relation ⇨ **Deleted in S, SP**

S

S#	Snom	Zona	Ciutat
S1	Salazar	20	Londres
S3	Bernal	30	Paris
S4	Corona	20	Londres
S5	Aldana	30	Atenas

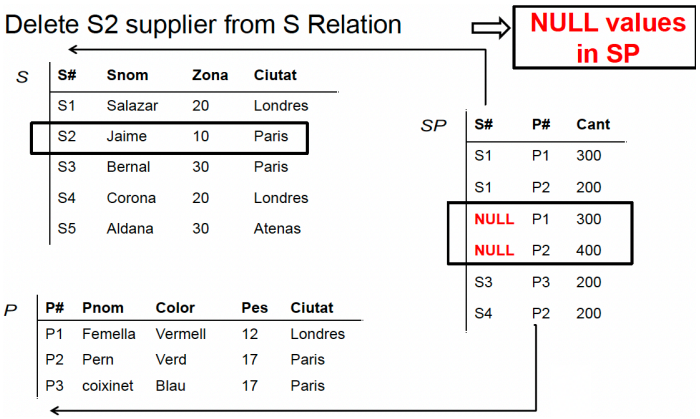
SP

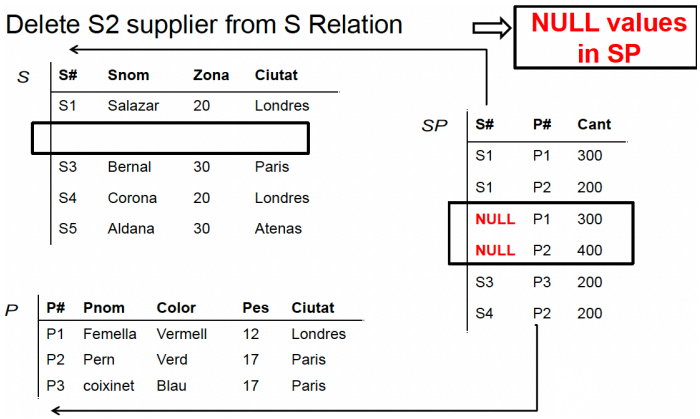
S#	P#	Cant
S1	P1	300
S1	P2	200
S3	P3	200
S4	P2	200

P

P#	Pnom	Color	Pes	Ciutat
P1	Femella	Vermell	12	Londres
P2	Pern	Verd	17	Paris
P3	coixinet	Blau	17	Paris

3. **Nullifies**





4. **Updates** (**UpDate**)

1. Restriced

Update S2 value from S Relation→ Not possible

2. Cascade

Update P2 value from P Relation → We change the values in P, SP

3. Nullify

Update P2 by P5 value from P Relation → NULL values in SP

In what order should it be done?

Observations

→ CASCADES FK rule can lead to problems

R3 **b** → R2 **a** → R1

Reference: CASCADES

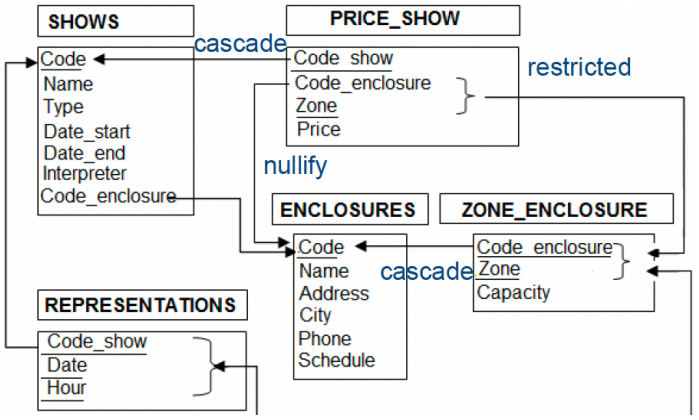
What if we delete tuples at R1?

If the rule a b does not allow to delete tuples in R3 as a result of removing elements in R2 → R1 tuple can **not** be deleted

Restricted (default mode) is the rule that has fewer risks and it is easier to implement but less comfortable for the user (deleting with a certain order)

Example 1

What happens if we delete an enclosure?



Example 2

Do these rules allow deleting a record in the table Enclosures? Why? If that is not possible, what should we do before? If that is possible, which effects it has on the other tables?