BDA NOTES 3

UNIT 3: DATABASE MANAGEMENT SYSTEMS (DBMS)

1.-THE ANSI/SPARC ARCHITECTURE

1.1.-SCHEMAS

The working group ANSI/SPARC proposed the **database definition** with **3 levels of abstraction**:

Internal level (internal schema): description of the DB in terms of its physical representation.

Conceptual level (conceptual schema): description of the DB independently of the DBMS. It is usually a graphical representation.

External level (external schemas): description of the different users' partial views.

Since there was no generalized conceptual model for different kinds of DBMS, the "logical" level was added:

Internal (physical) **level** (internal schema): DB description in terms of its physical representation. Describe how the database is store in secondary memory.

Logical level (logical schema): DB description in terms of the DBMS data model. It does not include any details of the physical representation.

Conceptual level (conceptual schema): description of the information system from the organizational point of view. It is independent of the DBMS. We will use a UML class diagram.

External level (external schemas): description of the partial views which the different users have on the DB.

A DBMS that supports **the 3-level architecture must**:

- Allow the **definition** of the different **schemas** for the database (except the conceptual schema).
- Establish the **correspondence** between schemas.
- **Isolate the schemas**: changes in one schema should not affect neither the schemas at upper levels nor the application programs.

1.2.-DBMS FUNDAMENTALS

A **DBMS** is the software which allows the creation and manipulation of databases (DB). A DBMS must maintain the **independence**, **integrity** and **security** of data.

The **objectives of DB techniques** are:

- Unified and independent **data description**
- **Application** independence
- Partial **view** definition.
- Information **management**.
- Data **integrity** and **security**.

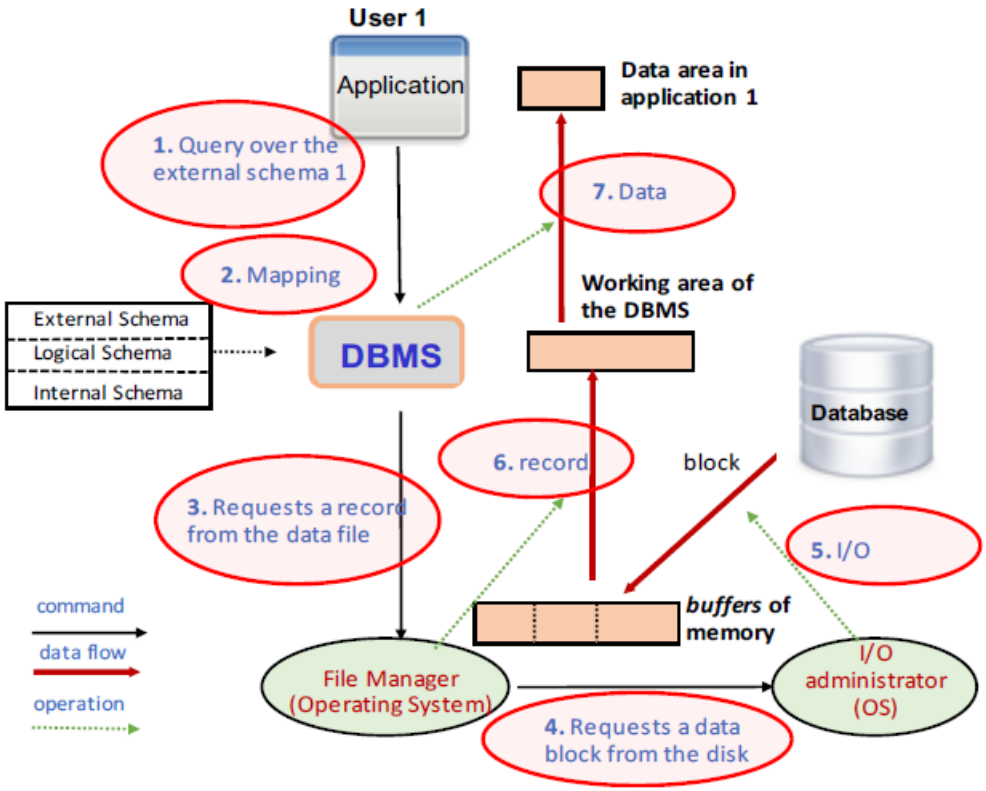
The **DBMS functions** are:

- **Data definition** at several **levels**: logical schema, internal schema and external schema.
- Data **manipulation**: query and update.
- Management and **administration** of the datab.ase
- Control of: semantic integrity, concurrent access, recovery in case of failure and security (privacy).

The **DBMS components** are:

- Schema definition languages and their associated translators.
- Manipulation languages and their associated translators.
- Tools for: restructuring, simulation, statistics, printing and reporting
- Tools for: integrity control, reconstruction and security control.

Overview for **accessing the data**:



1.3.-DATA INDEPENDENCE

Data independence is the property which ensures that the **application programs are independent of**:

- The **changes** which are performed on **data** which they do **not used**.
- The **physical representation** details of the accessed data.

The **logical independence** is between the logical schema and the external schemas. It makes that the external schemas and the application programs cannot be affected by the modifications in the logical schema of data which are not used by these programs.

The **physical independence** is between the internal schema and the logical schema. It makes that the logical schema cannot be affected by changes in the internal schema which refer to the implementation of the data structures, access modes, page size, search path, etc.

A **binding** is a transformation of the external schema into the internal schema. There are two types: **logical binding** and **physical binding**. When the binding is performed, the independence disappears. It is important to determine the binding moment. Currently, most DBMS do the binding for each query.

2.-TRANSACTIONS, INTEGRITY AND CONCURRENCY

The **objective** of DB technology is to have **information quality**. Data must be structured in such a way to adequately **reflect** the **objects**, **relations**, and **constraints** which exist in the part of the real world modelled by the database model.

When **reality changes**, the user **updates** the database. The information contained in the DB must preserve the schema definition.

Information quality (integrity perspective) means that:

- The DBMS must ensure that the data are **correctly stored**.
- The DBMS must ensure that **user updates** over the DB are correctly executed and become **permanent**.

The **DBMS tools** are oriented towards integrity:

- Check (when an update is performed) the **integrity constraints** defined in the schema.
- Control the correct execution of the **updates** in a concurrent environment.
- Recover (**reconstruct**) the DB in case of loss or accident.

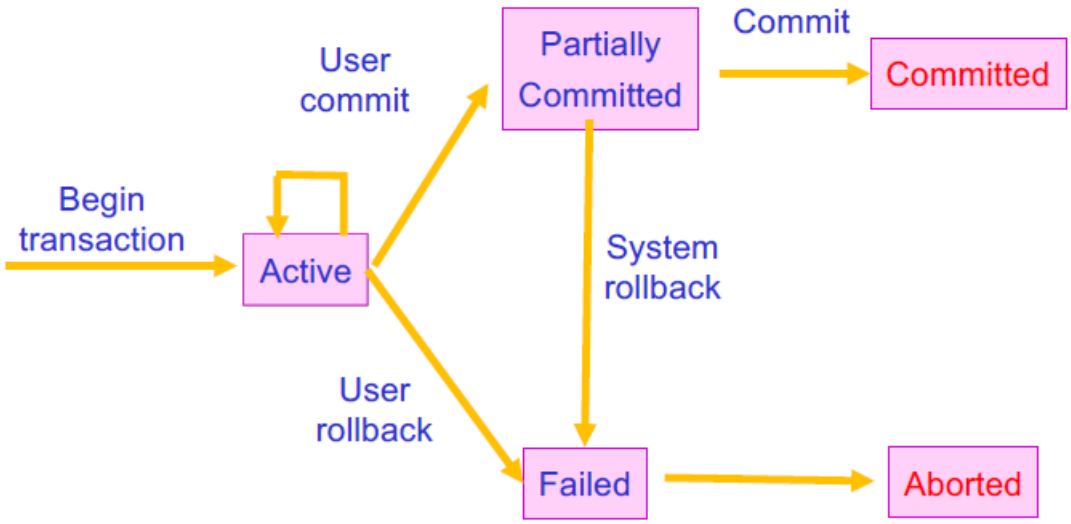
2.1.-TRANSACTIONS

The operations in a DB are organized in transactions. A **transaction** is a sequence of access operations to the DB which constitute a **logical execution unit**.

The **actions** which **change transactions states** are:

- **Begin**: indicates the **beginning** of the execution of the transaction.
- **Cancelation (user rollback)**: the user aborts the transaction.
- **Confirmation (user commit)**: the user considers the transaction as ended. Then the DBMS performs some checking to determine how the transaction will end:
    - **Success (system commit)**: indicates the **success** of the transaction, making the DBMS store the changes performed on the DB.
    - **Failure (system rollback)**: indicates the **failure** of the transaction, or that the transaction hasn't passed the checking. The DBMS undoes all the possible changes performed by the transaction.

The **states of a transaction** are:



The **properties of the transactions (ACID)** are:

- **Atomicity**: a transaction is an indivisible unit that is either **performed** in its **entirety** or is not performed at all ("All or nothing").
- **Consistency**: the transaction must transform the DB from one consistent state to **another consistent state** (all integrity constraints must be met).
- **Isolation**: concurrent transactions are executed independently, since all the partial effects of **incomplete transactions should not be visible** to other transactions.
- **Durability**: the effects of a **successfully** completed (committed) transaction are **permanently recorded** in the DB and must not be lost because of a subsequent system or other transaction failure.

2.2.-SEMANTIC INTEGRITY

An **integrity constraint** is a property of the real world which is modelled by the DB.

Constraints are defined in the **logical schema** and the DBMS must ensure that they are met. The checking is **performed** whenever the **DB changes** (when any updating operation is executed). Constraints **not included in the DB** schema must be maintained by the application programs. This last situation is, in general, **inappropriate** if the constraints are common to more than one application, since the responsibility to check them is dispersed.

There are **two types of integrity constraints**:

- **Static**: they must be met in each state of the DB (they can be represented by logical expressions).
- **Dynamic (Transition)**: they must be met regarding two consecutive states.

Static constraints can be expressed over:

- **Data values**.
- **Attributes**.
- **Relations**.
- The DB (**general constraint**).

They are checked after every command (**IMMEDIATE**) or at the end of the transaction (**DEFERRED**).

The **triggers** are used when a designer wants to have a system response when some events are produced. This allows to incorporate complex constraints into the DB. These triggers include:

- **Events**: operations over the DB which trigger it.
- **Conditions** to determine if the actions must be executed or not.
- **Actions** to be executed when an event happens and the conditions are met. They are usually written in a data-oriented high level programming language, that can include SQL commands.

2.3.-CONCURRENT ACCESS CONTROL

In order to keep the integrity of the database, the DBMS must control concurrent access to the database to avoid that the results of the execution of **several processes** (users or programs) lead to **incorrect**, **incoherent** or **lost results** because of the simultaneous execution of other program accessing the same data.

The **basic operations** in a transaction which are relevant to the DBMS:

- **read(X)**: reading or access to a piece of data X in the DB over the program variable with the same name. Steps:
    1. Seek the address of the block which contains the datum X.
    2. Copy the block to a buffer into main memory.
    3. Copy the datum X from the buffer to the program variable X.
- **write(X)**: update (insertion, deletion, or modification) of a piece of data X in the DB by using the program variable with the same name. Steps:
    1. Find the address of the block containing the datum X (if not read before).
    2. Copy the block into a database buffer in main memory (if not read before).
    3. Copy the datum X from the program variable into the database buffer.
    4. Write the updated block from the database buffer to the disk.

The **problems** that arise due to the **interference of concurrent accesses** are:

- **Loss of updates**. An apparently successfully completed update operation by one user can be overridden by another user.
- **Inconsistent** information corresponding to several valid database states. One transaction reads several values but a second transaction updates some of them during the execution of the first.

- Access to updated data (but **still not confirmed**) that can still be cancelled. One transaction is allowed to use the intermediate results of another transaction before it has committed (dirty read).

The **techniques** to solve these problems are:

- Reserving some data occurrences (**locks**).
- Cascade cancellation (only for the third problem).
- Transaction isolation (only for the third problem).

3.-RECOVERY AND SECURITY

A database must guarantee:

- **Recovery**: a database must always be recovered from any type of failure.
- **Security**: a database cannot allow non-authorized access.

3.1.-DB RECOVERY

The transaction properties of **atomicity** and **durability** force a DBMS to ensure that:

- If **confirmed**, the changes performed are recorded in the DB to make them persistent.
- If **cancelled**, the changes performed over the DB are **undone**.

Backups alone are **not** the **solution** to the recovery problem. The increase of the backup frequency is not a feasible solution. DB technology provides much more efficient and robust techniques for DB recovery. The lost of confirmed data is inadmissible with current technology.

The **causes of transaction failure** are:

- **Local to the transaction** (normal system operation):
  - **Transaction error**.
  - **Exceptions**.
  - **Concurrency control** (locked state between two transactions).
  - Human **decision** (inside a program or explicitly).
- **Extern to the transaction** (system error):
  - System failures with **loss of main memory**.
  - Failures in the storage system with **loss of secondary memory**.

The changes performed by a transaction are located in memory buffers (main memory). When the transaction is **confirmed** its changes must be recorded in secondary memory. If a failure with loss of main memory occurs between the **transaction confirmation** and **flushing the buffers** to secondary memory, the blocks in the buffers will be lost.

The changes performed by a **confirmed** transaction are recorded into the DB. If there is a **failure in secondary memory**, the **changes will be lost**.

3.1.1.-RECOVERY FROM FAILURES OF SYSTEM MAIN MEMORY

We have to **recover confirmed transactions** which have **not been recorded** and can **cancel transactions** which have **failed**. These operations are effectuated by the recovery module. The most used technique is the **journal file** or **log**.

There are **two ways of implementing transactions**:

- **Transactions with Immediate Update**: updates have an **immediate effect on secondary memory**. In case of cancellation, they have to be undone.
- **Transactions with Deferred Update**: updates only have immediate effect on main memory. The updates will be transferred to secondary memory **when confirmed**.

Both ways need a log file.

The **activities and events are recorded in the log file**. It **records** the update operations performed by existing transactions during a period of time. The log file is stored on **disk** to avoid loss after a system failure. It is **dumped periodically** into a massive storage unit.

Types of **entries in a log file**:

- **[start, T]**: a transaction with identifier T has been started.
- **[write, T, X, value_before, value_after]**: the T transaction has performed an update instruction on data X.
- **[read, T, X]**: the T transaction has read data X.
- **[confirm, T]**: the T transaction has been confirmed.
- **[cancel, T]**: the T transaction has been cancelled.

There are **two problems** with this technique:

- Size of log file can increase very quickly.
- Recovery in case of failure is very expensive (many instructions have to be redone).

The solution to these problems is **checkpoints**. Checkpoints are recorded in the log file periodically. The **steps to perform a checkpoint** are:

1. **Suspend** the execution of transactions temporally.
2. **Record** a checkpoint in the log file.
3. **Record** all updates performed by confirmed transactions (copy all main memory **buffers to disk**).
4. **Resume** the execution of the suspended transactions.

3.1.1.1.-INMEDIATE DATABASE UPDATES

Updates have an **immediate effect on secondary memory**. In case of cancellation, they have to be undone. In case there is a **failure of a transaction T**, we have to **undo** changes performed by T. To do this, we have to update the data which has been modified by T with its original value, that is, **searching for the entries in the logfile** with [**write**, T, X, value_**before**, value_**after**].

In case of **unconfirmed transactions**, [start, T] in the log file without [confirm, T], we have to **undo** changes performed by T.

In case of **confirmed transactions**, [confirm, T], we have to **execute** (redo) them again, that is, search the write entry and change the values.

Basic considerations:

- **Updates** performed by **confirmed** transaction (a transaction commit appears in the log file) could have **not** been transferred to **disk** because the buffer block where they are, has not been recorded yet: **redo**.
- **Updates** performed by **non-confirmed** transactions (there is no transaction commit in the log file) could be **in disk** because their main memory blocks were transferred to disk: **undo** (only used in immediate updates).
- When a **checkpoint** is recorded, the DBMS **records** all the updates performed by the **confirmed** transaction.

3.1.1.2.-DEFERRED DATABASE UPDATES

Updates only have **immediate effect on main memory**. The updates are only written to the secondary memory **until confirmed** (after the commit).

In case of **unconfirmed transactions**, [start, T] in the log file without [confirm, T], we have to **do nothing** since they are not in secondary memory.

In case of **confirmed transactions**, [confirm, T], we have to **execute** (redo) them again.

Basic considerations:

- **Updates** performed by **confirmed** transaction (a transaction commit appears in the log file) could have **not** been transferred to **disk** because the buffer block where they are, has not been recorded yet: **redo**.
- **Updates** performed by **non-confirmed** transactions (there is no transaction commit in the log file) are **not in disk** : **do nothing**.
- When a **checkpoint** is recorded, the DBMS **records** all the updates performed by the **confirmed** transaction.

3.1.2.-RECOVERY FROM FAILURES OF SECONDARY MEMORY

When a failure of the **storage system** occurs, the database might be **damaged** totally or partially. The technique to solve this is **reconstruction of the database**:

- Using the most recent **backup**.
- From the backup instant, the system uses the **log file** to redo all the instructions performed by the confirmed transactions.

3.2.-SECURITY

The **objective** is that information can only be accessed by the people and processes that are authorized and in the authorized way.

There are **two techniques**:

- **User identification**.
- **Establishment of allowed accesses**. It has **two modes**:
  - **Authorization list** associated to each user containing the allowed objects and operations. (GRANT).
  - **Level of authorization** (less flexible). There several users groups with different authorization.
- **Management of transferrable authorizations**: handover of authorizations from one user to another. (**WITH GRANT OPTION**).

In the last case, it is necessary to know the access **authorizations** of each **user** (some authorizations will be transferable to other users). One authorization can be transferred to other user in **mode transferable or not**. When one authorization is **revoked**, ff the authorization was transferable, it is necessary to revoke all the transferred authorizations. If one user receives more than one authorization, each of the authorizations can be independently revoked.

3.2.1.-PRIVACY AND SECURITY

We have to have extreme care on:

- Protection against the access or spreading of **personal data** to **non-authorized** users.
- Control the **flow** to third parties of **information** that can contain personal data or **information** that is apparently **aggregated** (parameterized queries) but that might **reveal particular** information for some parameters.
- **Custody of security backups**, retired or malfunctioning disks, etc.
- **Small devices**: **lost** or **stolen** very easily.

Personal data is “any information relating to an identified or identifiable natural person.”

UNIT 4: RELATIONAL DATABASE DESIGN

CHAPTER 1.-DATABASE DESIGN FUNDAMENTALS

1.-INTRODUCTION

Methodology issues are strategies and recommendations to address the design problem.

Modelling languages issues are a presentation of an appropriate (graphical) language to represent the system (data model).

2.-METHODOLOGY

A **methodology** is a set of standard **procedures**, **techniques** and **documentation** for the development of a product (a database in our case). A methodology is supported by:

- **Techniques**: how to deal with each of the steps and activities in the methodology
- **Models**: Way to represent or think abstractly about the reality, the problem or the solution.
- **CASE tools**: (optionally) software tools to automate or assist on the development of techniques and models.

The **analysis** is to obtain the set of requirements of information and of process that the organization needs for achieve its aims.

There are **different types of design**:

- **Conceptual Design**: obtain a representation of reality including static and dynamics properties necessaries to meet the organization's requirements.
- **Logical Design**: translation of the conceptual scheme to the data model of the DBMS to use.
- **Physical Design**: selection of storage structures taking into account details of the physical representation for obtaining a good performance.

The **deployment** is the incorporations of the database and applications into the organization.

3.-DATA MODELS

A **data model** is a way in which **static** and **dynamic** properties of reality are represented. We are going to use a conceptual data model that:

- Incorporates notions from the **Entity-Relationship** Model using **UML**.
- Is more **abstract**, **expressive**, and **system-independent** than the classical relational model.
- Is essentially **graphical** (based on UML).

CHAPTER 2.-CONCEPTUAL DESIGN

1.-INTRODUCTION

The **conceptual design** is the stage of the database design process which aims at "obtaining a **representation** of reality which captures its **static** and **dynamic** properties such that requirements are satisfied. This representation must be a truthful image of the actual world". For the static design we will use **UML Class Diagrams**.

Our enhanced **UML diagrams** will represent:

- The structures which constitute the content of the information system.
- The constraints which limit the occurrences of the data.

Some of the **elements** that we will see:

- Class (entity).
- Attribute.
- Association (common relationship).
- Generalization/specialization (other kind of relationship).
- Constraints.

2.-THE UML CLASS DIAGRAM

2.1.-CLASS

The observation of reality leads to the detection of (physical or abstract) "**objects**" or "entities". Through **classification** (a very simple abstraction mechanism) we identify the kind of "classes" or "entity types" (types of objects) which are interesting for the organization.

Objects which are of the same kind are represented by a class (or entity). A **class** is a description of a set of objects which share the same properties.

2.2.-ATTRIBUTE

An **attribute** is a property of a class which is identified by a name, and whose instances can have different values from a given specified set. There can be composite attributes.

For each attribute we can specify:

- **Data type or domain**. A predefined datatype which determines the possible values for the attribute:
    - The name of a datatype.
    - An enumeration of possible values (in parentheses).
    - A record (composite attribute).
- **Constraints**:
    - Uniqueness.
    - Multiplicity.
    - Identification.

Derived attributes are special attributes whose value is determined by a rule or formula.

Different occurrences of a class must take **different values** for the attribute (or set of attributes) with the **uniqueness constraint**.

The **cardinality** expresses the **number of values** that each attribute can take for each object in the class:

- **{1..1}**: The attribute has exactly one value for each occurrence in the class.
- **{0..1}**: The attribute can have no value or just one (this is the **default case**).
- **{1..*}**: The attribute must have one or more values (but at least one).
- **{0..*}**: The attribute can have no values of any number or values.

An **identifier** is as set of attributes with uniqueness constraint and multiplicity {1..1}. It allows distinguish any two occurrences of the class. There is only one identifier per class (but may contain several attributes).

2.3.-ASSOCIATION

The **associations** represent **relationships** between the classes. An association connecting two (and only two) classes is said to be **binary**. Associations are annotated with:

- **Name**: verbs are preferable.
- **Arrow** (optional): to determine the subject and the object.
- **Role** (optional): to clarify the relation, especially when the association is reflexive (a class is associated with itself).
- **Multiplicity**: participation-cardinality constraint.

When the minimum multiplicity (minA) of a class A is **greater than 0** we say that class A has an **existence constraint** relative to the association.

To include information to a relation, we use **anonymous classes** or **link attributes**.

To make an association within an association we use **association classes**, that are classes that have a relation with an anonymous class.

2.4.-WEAK CLASS

A class has an **identification dependency constraint** when it **cannot be identified with its own attributes**. Its occurrences must be distinguished through the association to other class(es).

These classes are called **weak classes**. When a weak class depends on another class (which might be weak as well), we talk about the **subordinate** and the **dominant** classes.

This constraint is represented by the label **{id}** instead of the multiplicity (“id” is a special case of ”1..1”). The weak class counts on some attribute of the dominant class in order to be identified.

2.5.-TERNARY ASSOCIATIONS

An association connecting two classes is called **binary**. In UML, we can also represent N-ary associations. Nonetheless, for simplicity, we are going to use only binary associations. **Ternary associations** are usually **represented** using:

- **Association classes**.
- **Weak classes**.

2.6.-SPECIALIZATION / GENERALIZATION

When several classes **share properties** that might well be represented as part of a more abstract class, taking the common properties, then we have a **generalisation/specialization association**. The general class is said to be specialized into one or more **subclasses**, and the specific classes are said to be generalized into a **superclass**. All subclasses **inherit** the attributes of the superclass.

There are different **kinds of generalization/specialization**:

- **Total (complete)**: Every occurrence of the superclass G must participate as an occurrence of any of its subclasses C1, C2 o C3.
- **Partial (incomplete)**: There can be occurrences of the superclass G which are not of any of its subclasses (default case).
- **Disjoint**: One occurrence of the superclass G cannot participate as more than one occurrence of any subclasses.
- **Overlapping (nondisjoint)**: One occurrence of the superclass G can participate as more than one occurrence of any of its subclasses (default case).

The subclasses must have a distinctive attribute.

3.-METHHODOLOGY TO OBTAIN THE ER-UML DIAGRAM

The **steps** are:

1. Identify classes with their attributes.
2. Identify generalization/specializations.
3. Identify associations between classes.
4. Specify integrity constraints.
5. Final checks.

3.1.-IDENTIFY CLASSES WITH THEIR ATTRIBUTES

For each object or **entity** of interest found in the requirements, we will define a **class** in the UML diagram. We will **identify the attributes** which define each class.

For each attribute we must:

- Associate a **domain** or, it is a derived attribute, a formula/expression of how it is defined.
- Specify the **multiplicity** and the uniqueness constraints.

Chose a class identifier. If there is no proper identifier, the class is considered **weak** and we must determine which dominant class/es it relies on to identify its occurrences. **Check** the diagram and reconsider the whole picture (redefine some class if it is necessary).

3.2.-IDENTIFY GENERALIZATIONS/SPECIALIZATIONS

There are **different strategies**:

- **Descending strategy**: specialization of a general class.
- **Ascending strategy**: generalization of several specific classes.
- **Hierarchization (is_a)**: one class is a special case of the other.

3.3.-IDENTIFY ASSOCIATIONS BETWEEN CLASSES

Steps:

1. **Determine the multiplicity** (minimum and maximum cardinality). In case of doubt, choose the least restrictive multiplicities.
2. **Remove redundant associations**.
3. **Remove transitive redundancies**. An association is redundant because it can be derived from other associations. Sometimes is not possible to eliminate any association.
4. **Specify roles in associations of a class with itself (reflexive)**. Be aware of the reflexive associations since we usually have to indicate some properties which don't appear in the association definition.

3.4.-SPECIFY INTEGRITY CONSTRAINTS

Any **other property** of reality that has not been expressed in the UML diagram must be included now. Use UML **annotation elements**.

3.5.-FINAL CHECKS

Names:

- We cannot use the **same** name for **different classes**, **associations** or **roles**.
- The **attribute names** in a class cannot be **repeated**.

Identifiers:

- **All classes** must have an identifier (otherwise are weak or specializations).
- **Specialized** classes (subclasses) **do not have** identifier, since they inherit it from the superclass. They are not weak classes. They cannot redefine attributes.
- **Association classes never** have identifiers.
- A class never uses attributes referring to other class. We use associations.

CHAPTER 3.-LOGICAL DESIGN

1.-INTRODUCTION

We can transform the ER UML diagram into other formal models. In **software engineering**, this can lead to the definition of classes and attributes. In **databases**, we can transform the diagram into other database models. This transformation is known as **logical design** and the output will be the **relational schema**.

The **Logical Design** is a transformation of a conceptual schema, described using a data model into another data model which will be the one used by the Database Management System.

We are going to apply **transformations**. Multiplicities, associations, and constraints are expressed by the use of **PK**, **FK**, **NNV**, and **UNI** constraints. Some properties or constraints cannot be represented using these predefined constraints and we will have to add them to the **list of general integrity constraints**. When facing **several design options**:

1. Chose the resulting schema with the **fewest general constraints**.
2. If the number of general constraints is similar, choose the solution with the **fewest relations**.

Methodology to obtain the relation schema:

1. Transform **classes** into relations.
2. Transform **associations** according to their multiplicity.
3. Those properties that can't be represented in the relational schema, will be expressed as a list of **integrity constraints**.

SEE SLIDES FOR THE FOLLOWING CONTENT