

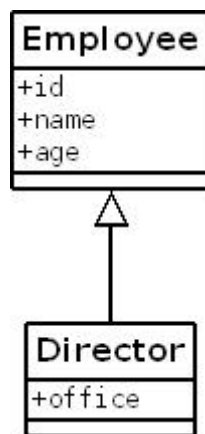
GENERALIZATION

DEFINITION

A generalization is a relationship between a more general class and a more specific class.

Each instance of the specific class is also an indirect instance of the general class. Thus, the specific class inherits the attributes of the more general one and is constrained by the constraints applicable to the general class. In particular, the identifiers of the general class are also identifiers of the specific one. In most cases, the specific class may act in place of the general one. For instance, the specific class participates in the associations that the general one participates in.

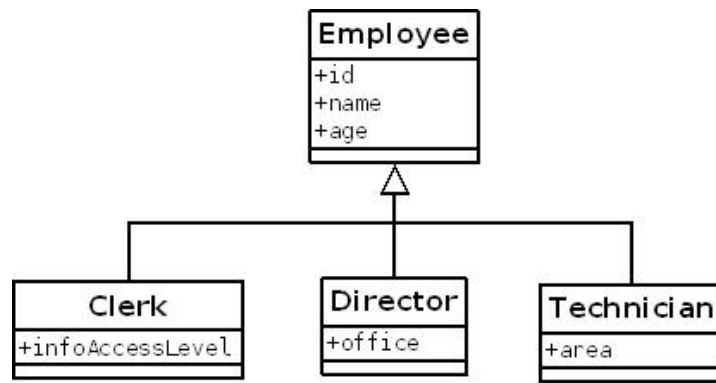
The specific class may define new attributes and participate in new associations in addition to the ones inherited from the general class. In the following example, *Employee* is the general class and *Director* the specific one. This class has four attributes: three inherited from *Employee* and one, *office*, defined by itself.



SUBDIVIDING A CLASS

Sometimes several generalizations with a common general class are defined collectively in such a way that they describe how the general class may be divided into subclasses. Note that for practical issues, we may consider a single generalization as the one mentioned before as a special case of the one we are considering now.

In the example below, *Employee* is subdivided into three subclasses: *Clerk*, *Director* and *Technician*.



In this case, we must specify the exact meaning of the generalizations with respect to two dimensions:

- Complete/Incomplete. If every instance of the general class is also an instance of some of the subclasses, the generalization is said to be **complete**. Otherwise, if there may exist instances of the general class that are not instances of any subclass, the generalization is **incomplete**.
- Disjoint/Overlapping. If an instance of a subclass (and, hence, of the general class) may also be an instance of another subclass, the generalization is an **overlapping** one. Otherwise, the generalization is **disjoint**.

A pair of capital letters, such as {D, C} for Disjoint/Complete, is attached near the generalization arrow to specify in the class diagram the nature of the generalization.

TRANSLATION INTO RELATIONAL MODEL

There are three options to represent the information of a subdivision structure including some classes and generalizations defined collectively:

1. A single relation for all the classes. This relation will have the union of all the attributes defined in the classes.

Employee(id,name,age,infoAccessLevel,office, area)

2. A relation for each subclass, but no relation for the general one. Each relation will have the attributes of the corresponding class together with the ones defined by the general class. The above example would turn to:

Clerk(id, name, age, infoAccessLevel)

Director(id, name, age, office)

Technician(id, name, age, area)

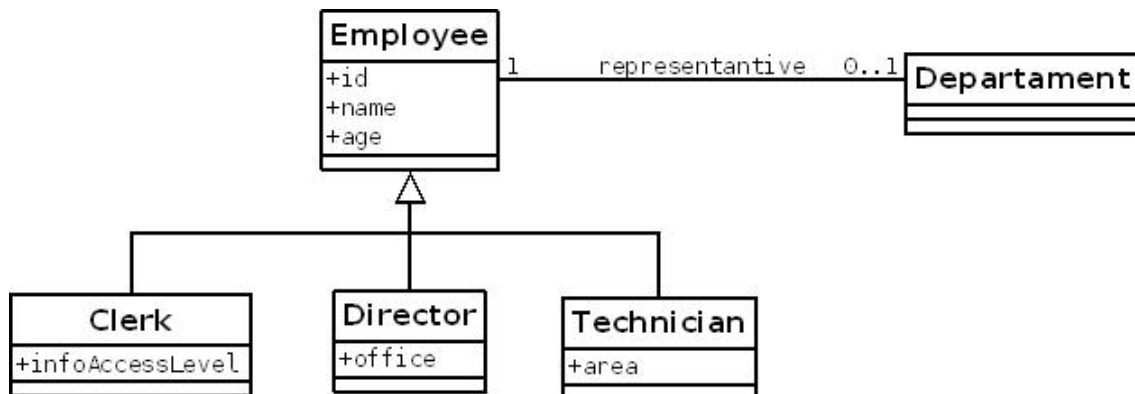
3. A relation for each class. The relations will be defined with the attributes of the corresponding class. The relations corresponding to the subclasses will hold, in

addition, a foreign key constraint referencing the relation corresponding to the general class (keep in mind that identifiers are inherited). Now, we would have:

```
Employee(id, name, age)
Clerk(id, infoAccessLevel) {id references Employee(id)}
Director(id, office) {id references Employee(id)}
Technician(id, area) {id references Employee(id)}
```

In order to choose between the three possible representations of a generalization we can consider a number of circumstances. A first issue that we must always take into account is which of the possible structures can store all the information of all the possible instances (e.g., if we choose option 2 for an incomplete generalization, there is no relation to store those instances that do not belong to any subclass). Another important criterion is the number of null values generated by each structure (e.g., if we choose option 1, each tuple will have null values for those attributes corresponding to the subclasses that the instance does not belong to). It has also to be said that some options lead to redundancy when the generalization is overlapping (e.g., if we choose option 2 for an overlapping generalization, the common attributes of the superclass will be stored once per subclasse the instance belongs to). Finally, performance issues have to be considered: depending on the query, option 3 generates joins and option 2 generates unions while the alternative option does not. We, of course, will choose the alternative that allows storing all instances and leading to no (or the minimum number of) null values nor redundancy if there are no performance reasons demanding a different option.

This just mentioned rule can be used as a general guide but one must consider the whole conceptual schema in which the generalization is involved. For instance, if there are references from other relations (coming from the translation of another part of the schema), some of the options become invalid. Consider, for instance, the shema below. If we decide to translate *representative* into a foreign key in *Department* referencing *Employee*, option 2 is not valid.



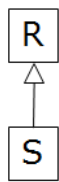
ACCESSING DATA

We address now the problem of obtaining all the instances of the general class of a generalization from the relational model used to represent the generalization. We want to obtain the result as a set containing a tuple for each instance, no more no less, of the general class. And the tuples must have as attributes the union of all attributes.

If we have turned the generalization into a single relation it's as easy as selecting all the tuples of that relation. But if we have used option 2 or 3 we have to take into account the following two facts:

- The same instance may be present in more than one of the relations corresponding to subclasses if the generalization is overlapping.
- If we have chosen to have all the relations, the instances in the relations corresponding to subclasses are also present in the relation corresponding to the general class, but the opposite is not true (i.e., the instances of the superclass do not belong to all subclasses).

Thus, it is not as easy as performing the union of the relations because the same instance would result in several tuples. For one subclass, the solution is executing a union as follows:



```
SELECT R.a, R.b, S.c
FROM R, S
WHERE R.b=S.b
UNION
SELECT R.a, R.b, NULL
FROM R
WHERE R.b NOT IN ( SELECT S.b
                   FROM S);
```

In general, with more than one subclass, we must unite the following sets of tuples:

- Instances belonging to the general class only.
- Instances belonging only to the first subclass.
- Instances belonging only to the second subclass
- All other similar sets
- Instances belonging to the first and second subclasses
- All other combinations of two subclasses
- All other combination of subclasses

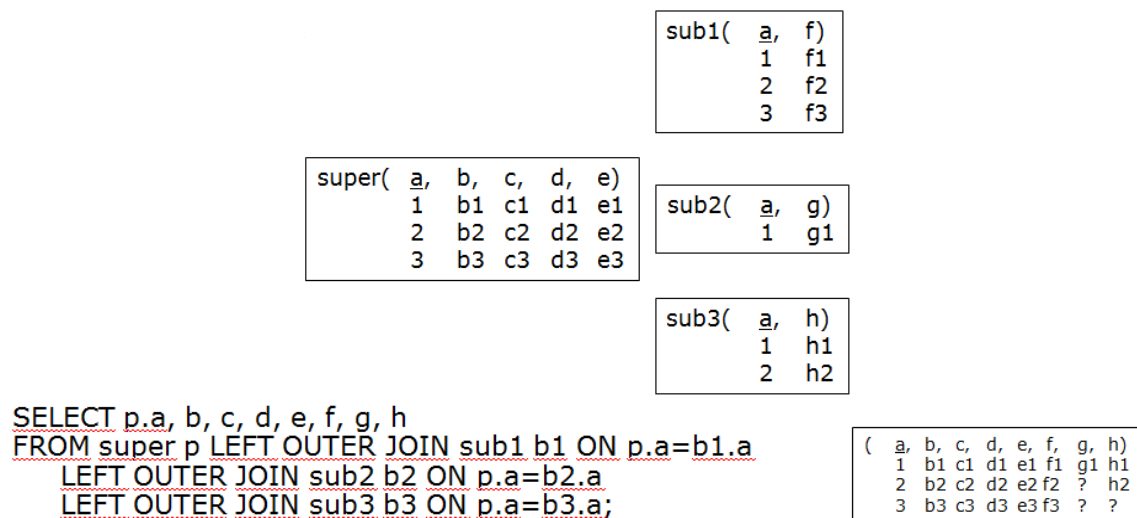
Although some of the sets above are known to be empty depending on the complete/disjoint nature of the generalization, this solution is clearly unfeasible. A new kind of JOIN operation, the OUTER JOIN, will help us a lot in solving the problem stated at the beginning of this section. There are three variations: left, right and full outer join.

```
SELECT xxx FROM A LEFT OUTER JOIN B ON cond WHERE yyy
```

performs a normal JOIN (also called INNER) but the rows in *A* not joining any row in *B* by means of the condition, will be joined with an artificial row in *B* with null values in all the attributes.

Outer join operations may be chained and one have to keep in mind that will be performed from left to right and prior to the evaluation of *where* conditions (this is important to know as long as outer joins may produce null values). Note that all we need to do in order to solve our problem is to produce the LEFT OUTER JOIN of all the relations representing the generalization.

In the next figure, you can see how we can easily retrieve all data corresponding to all the instances of “super” (question marks in the result of the query correspond to null values).



Right outer join works in a symmetric way (all rows in *B* will be present in the result, even those that don't match any row from *A*) and full outer join guarantees the presence of all rows of *A* and *B* in the result.

Different kinds of outer join can be used in the same statement. Thus,

```
SELECT xxx FROM A LEFT OUTER JOIN B ON cond1 RIGHT OUTER JOIN C ON cond2 WHERE yyy
```

will perform A LEFT OUTER JOIN B and the result will be RIGHT OUTER JOINed with C.