Extended Entity Relationship Model

Is a conceptual data model incorporating extensions to the original ER Model.

Was developed to reflect more precisely the properties and constraints that are found in more complex databases

Developed to be used in engineering design and manufacturing (CAD/CAM), telecommunications, complex software systems and geographic information systems (GIS).

Includes all of the concepts introduced by the ER model, plus:

- concepts of a subclass and superclass
- specialization and generalization
- Introduces the concept of a union type or category, which is used to represent a collection of objects that is the union of objects of different entity types.

Subclass and superclass

Entity type Y is a subtype (subclass) of an entity type X if and only if every Y is necessarily an X. A subclass entity inherits all attributes and relationships of its superclass entity. A subclass entity may have its own specific attributes and relationships (together with all the attributes and relationships it inherits from the superclass). Most common superclass examples is a vehicle with subclasses of Car and Truck. There are a number of common attributes between a car and a truck, which would be part of the Superclass, while the attributes specific to a car or a truck (such as max payload, truck type...) would make up two subclasses.

User Defined Abstract data types

At the most basic level, and abstract data type is nothing more than a collection of smaller, basic data types that can be treated as a single entity. While this is a simple concept, the changes in database object design will be dramatic. Some argue that a database must be able to support data types which contain lists rather than finite values, and some of the object/relational databases such as UniSQL allow for a single data type (a column) to contain lists of values, or even another table.

Unlike traditional database management system which only provide for primitive data types such as INTEGER and CHARACTER, the object-oriented programming languages allow for the creation of abstract data types. The data types offered in commercial database systems, CHAR INTEGER NUMBER, VARCHAR, BIT, are sufficient for most relational database applications but developers are now beginning to realize that the ability to create user-defined data types can greatly simplify their database design. While these data types were popular within the programming languages, they have only recently been introduced into the mainstream world of database objects.

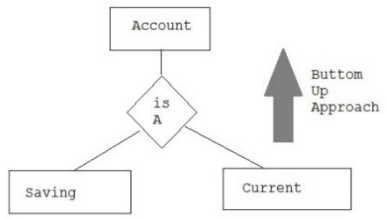
```
CREATE TYPE person_typ AS OBJECT (
  idno          NUMBER,
  name          VARCHAR2(30),
  phone         VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER );
----------------------------------------------------
CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
```

```
END;
-----------------------------------------------
CREATE TABLE contacts (
  contact         person_typ,
  contact_date    DATE );
INSERT INTO contacts VALUES (
  person_typ (65, 'Vrinda Mills', '1-800-555-4412'), '24 Jun 2003' );
-----------------------------------------------
SELECT c.contact.get_idno() FROM contacts c;
-----------------------------------------------
```

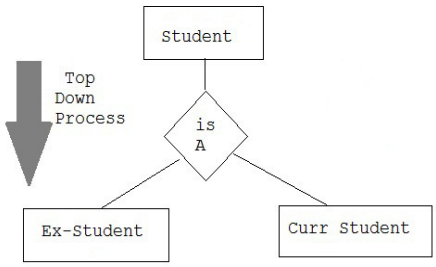
Specialization and Generalization

Are inverse to each other.

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity.



Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.



Constraints:

Disjointness vs. overlapping constraint: whether one entity can be in two or more subclasses.

In a disjoint specialization, also called an exclusive specialization, an individual of the parent class may be a member of only one specialized subclass.

In an overlapping specialization, an individual of of the parent class may be a member of more than one of the specialized subclasses.

Completeness Constraint: partial vs. total: other an entity is required to be in one of subclasses.

Total specialization exists when every instance of a supertype must also be an instance of a subtype. Partial specialization exists when every instance of a supertype does not have to be an instance of a subtype.

Basic concepts of UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously putting effort to make a truly industry standard.

1. UML stands for **U**nified **M**odeling **L**anguage.

2. UML is different from the other common programming languages like C++, Java, COBOL etc.

3. UML is a pictorial language used to make software blue prints.

So UML can be described as a general purpose visual modeling language to visualize, specify, construct and document software system. Although UML is generally used to model software systems but it is not limited within this boundary. It is also used to model non software systems as well like process flow in a manufacturing unit etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization UML is become an OMG (Object Management Group) standard.

The Degree of a Relationship Type

The degree of a relationship type concerns the number of entities within each entity type that can be linked by a given relationship type. There are two directions of a relationship type. Each is named and each has a minimum degree and a maximum degree.

The degree of relationship (also known as cardinality) is the number of occurrences in one entity which are associated (or linked) to the number of occurrences in another. There are three degrees of relationship, known as:

- one-to-one (1:1)
- one-to-many (1:M)
- many-to-many (M:N)

One-to-one (1:1)

This is where one occurrence of an entity relates to only one occurrence in another entity. A one-to-one relationship rarely exists in practice, but it can. However, we may consider combining them into one entity. For example, an employee is allocated a company car, which can only be driven by that employee. Therefore, there is a one-to-one relationship between employee and company car.

One-to-Many (1:M)

Is where one occurrence in an entity relates to many occurrences in another entity. For example, taking the employee and department entities shown on the previous page, an employee works in one department but a department has many employees. Therefore, there is a one-to-many relationship between department and employee.

Many-to-Many (M:N)

This is where many occurrences in an entity relate to many occurrences in another entity. The normalization process would prevent any such relationship. As with one-to-one relationships, many-to-many relationships rarely exist. Normally they occur because an entity has been missed. For example, an employee may work on several projects at the same time and a project has a team of many employees. Therefore, there is a many-to-many relationship between employee and project.

Relational database design by EER-to-relational mapping

ER Model	Relational Model
Entity type	Entity relation
1:1 and 1:N relationship type	Foreign key or relationship relation
M:N relationship type	Relationship relation and two foreign keys
n-ary relationship type	Relationship relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary key or secondary key

Step 1: Regular Entity Types

Create an *entity relation* for each strong entity type. Include all single-valued attributes. Flatten composite attributes. Keys become secondary keys, except for the one chosen to be the primary key.

Step 2: Weak Entity Types

Also create an entity relation for each weak entity type, similarly including its (flattened) single-valued attributes. In addition, add the primary key of each owner entity type as a foreign key attribute here. Possibly make this foreign key CASCADE.

Step 3: Binary 1:1 Relationship Types

Approach	Join cost	Null-storage cost
Foreign key	1	low to moderate
Merged relation	0	very high, unless both are total
Cross-reference	2	none

Step 4: Binary 1:N Relationship Types

Let the relationship be of the form [S]——^{N}<R>^{1}——[T]. The primary key of T is added as a foreign key in S. Attributes of R are moved to S. This is the foreign key approach. The merged relation approach is not possible for 1:N relationships. (Why?) The cross-reference approach might be used if the join cost is worth avoid null storage.

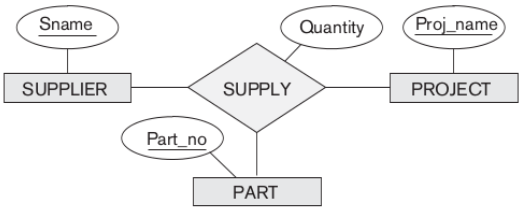
Step 5: Binary M:N Relationship Types

Here the cross-reference approach (also called a *relationship relation*) is the only possible way.

Step 6: Multivalued Attributes

Let an entity S have multivalued attribute A. Create a new relation R representing the attribute, with a foreign key into S added. The primary key of R is the combination of the foreign key and A. Once again this relation is dependent on an "owner relation" so its foreign key should CASCADE.

Step 7: Higher-Arity Relationship Types



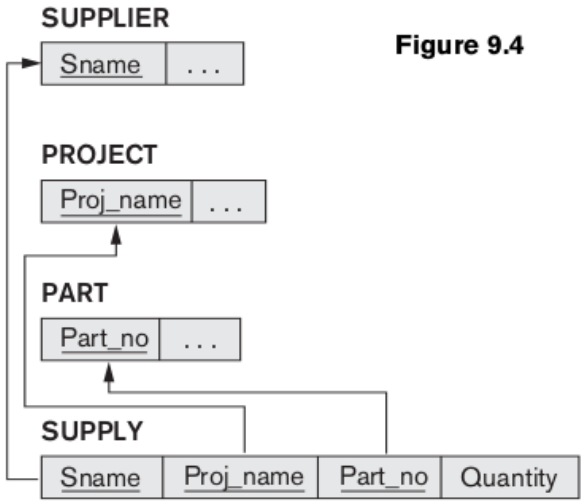


Figure 9.4

Here again, use the cross-reference approach. For each n-ary relationship create a relation to represent it. Add a foreign key into each participating entity type. Also add any attributes of the relationship. The primary key of this relation is the combination of all foreign keys into participating entity types *that do not have a max cardinality of 1*.

Step 8: Specialization/Generalization

Use Attrs(R) to mean the attributes of relation R, and PK(R) to mean the primary key of R. Let the subclasses be {S1,S2,…,S_m} and the superclass be C, and let Attr(C) = {k,a1,…,a_n} and PK(C) = k.

A. **Multiple relations—superclass and subclasses** Create a relation to represent C, having its attributes and primary key. Also create relations for eachS_i, having attributes Attr(S_i) ∪ {k}. In these relations k is *both* the primary key and a foreign key into C.

B. **Multiple relations—subclass relations only** Only create relations for each S_i. The attributes of each of these relations will be Attr(S_i) ∪ {k} ∪ Attr(C). The primary key will be k.

C. **Single relation with one type attribute** Create one relation representing the entire specialization set. Its attributes are Attr(C) ∪ {t} ∪ $_{i=0}U^m$ Attr(S_i). Attribute t is a type attribute (discriminator) that identifies which entity subtype an entity belongs to. If the specialization is already attribute-defined it uses that as t, otherwise t is a new attribute.

D. **Single relation with multiple type attributes** Proceed as in the previous approach, except instead of one t create *m* ts, each one a Boolean indicating whether a tuple is a member of its associated subtype. Together the ts form a bitmap of the entity's type.

Step 9: Union Types

Let C1,C2,…,Cm be the entity types participating in the union and S be the union type. Create a relation for S. If the primary keys of the Ci relations differ, create a surrogate key ks so that PK(S)=ks, and also add ks to each Attr(Ci) as a foreign key into S. If all the Cis have the same primary key type, use that as PK(S) instead.