3

Conceptual Data Modeling Using the (E)ER Model and UML Class Diagram

◈

Chapter Objectives

In this chapter, you will learn to:

- understand the different phases of database design: conceptual design, logical design, and physical design;
- build a conceptual data model using the ER model and understand the limitations thereof;
- build a conceptual data model using the EER model and understand the limitations thereof;
- build a conceptual data model using the UML class diagram and understand the limitations thereof.

Opening Scenario

Sober has decided to invest in a new database and begin a database design process. As a first step, it wants to formalize the data requirements in a conceptual data model. Sober asks you to build both an EER and a UML data model for its business setting. It also wants you to extensively comment on both models and properly indicate their shortcomings.

In this chapter we start by zooming out and reviewing the database design process. We elaborate on conceptual, logical, and physical database design. We continue the chapter with conceptual design, which aims at elucidating the data requirements of a business process in a formal way. We discuss three types of conceptual data models: the ER model; the EER model; and the UML class diagram. Each model is first defined in terms of its fundamental building blocks. Various examples are included for clarification. We also discuss the limitations of the three conceptual data models and contrast them in terms of their expressive power and modeling semantics. Subsequent chapters continue from the conceptual data models of this chapter and map them to logical and internal data models.

3.1 Phases of Database Design

Designing a database is a multi-step process, as illustrated in Figure 3.1. It starts from a **business process**. As an example, think about a B2B procurement application, invoice handling process, logistics process, or salary administration. A first step is **requirement collection and analysis**, where the aim is to carefully understand the different steps and data needs of the process. The information architect (see Chapter 4) will collaborate with the business user to elucidate the database requirements. Various techniques can be used, such as interviews or surveys with end-users, inspections of the documents used in the current process, etc. During the conceptual design, both parties try to formalize the data requirements in a **conceptual data model**. As mentioned before, this should be a high-level model, meaning it should be both easy to understand for the business user and formal enough for the database designer who will use it in the next step. The conceptual data model must be user-friendly, and preferably have a graphical representation such that it can be used as a handy communication and discussion instrument between both information architects and business users. It should be flexible enough that new or changing data requirements can easily be added to the model. Finally, it must be DBMS- or implementation-independent since its only goal is to adequately and accurately collect and analyze data requirements. This conceptual model will also have its limitations, which should be clearly documented and followed up during application development.

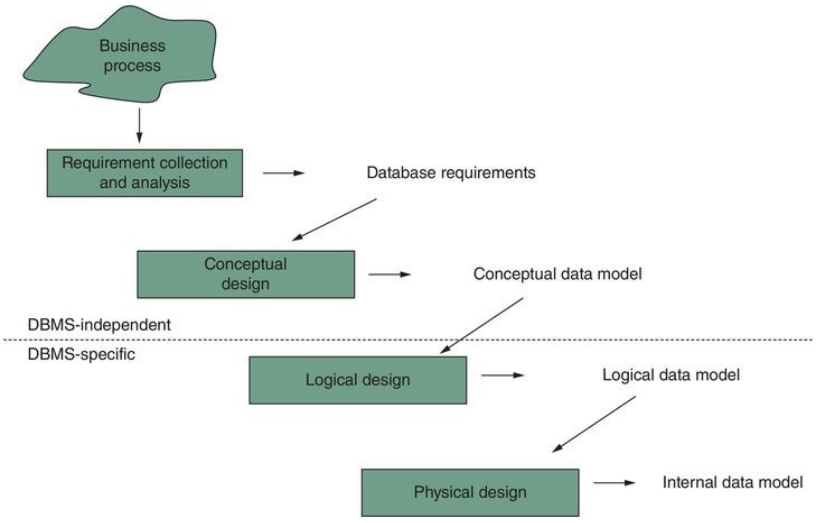


Figure 3.1 The database design process.

Once all parties have agreed upon the conceptual data model, it can be mapped to a logical data model by the database designer during the logical design step. The logical data model is based upon the data model used by the implementation environment. Although at this stage it is already known what type of DBMS (e.g., RDBMS, OODBMS, etc.) will be used, the product itself (e.g., Microsoft, IBM, Oracle) has not been decided yet. Consider a conceptual EER model that will be mapped to a logical relational model since the database will be implemented using an RDBMS. The mapping exercise can result in a loss of semantics which should be properly documented and followed up during application development. It might be possible that additional semantics can be added to further enrich the logical data model. Also, the views of the external data model can be designed during this logical design step.

In a final step, the logical data model can be mapped to an internal data model by the database designer. The DBA can also give some recommendations regarding performance during this physical design step. In this step, the DBMS product is known, the DDL is generated, and the data definitions are stored in the catalog. The database can then be populated with data and is ready for use.

Again, any semantics lost or added during this mapping step should be documented and followed up.

In this chapter, we elaborate on the ER model, EER model, and UML class diagram for conceptual data modeling. Subsequent chapters discuss logical and physical database design.

Connections

We discuss logical data models in Chapter 5 (hierarchical and CODASYL model), Chapters 6 and 7 (relational model), Chapter 8 (object-oriented model), Chapter 9 (extended relational model), Chapter 10 (XML model), and Chapter 11 (NoSQL models). Internal data models are covered in Chapters 12 and 13.

3.2 The Entity Relationship Model

The **entity relationship (ER)** model was introduced and formalized by Peter Chen in 1976. It is one of the most popular data models for conceptual data modeling. The ER model has an attractive and user-friendly graphical notation. Hence, it has the ideal properties to build a conceptual data model. It has three building blocks: entity types, attribute types, and relationship types. We elaborate on these in what follows. We also cover weak entity types and provide two examples of ER models. This section concludes by discussing the limitations of the ER model.

Drill Down

Peter Pin-Shan Chen is a Taiwanese-American computer scientist who developed the ER model in 1976. He has a PhD in computer science/applied mathematics from Harvard University and held various positions at MIT Sloan School of Management, UCLA Management School, Louisiana State University, Harvard, and National Tsing Hua University (Taiwan). He is currently a Distinguished Career Scientist and faculty member at Carnegie Mellon University. His seminal paper “The Entity–Relationship Model: Toward A Unified View of Data” was published in 1975 in *ACM Transactions on Database Systems*. It is considered one of the most influential papers within the field of computer software. His work initiated the research field of conceptual modeling.

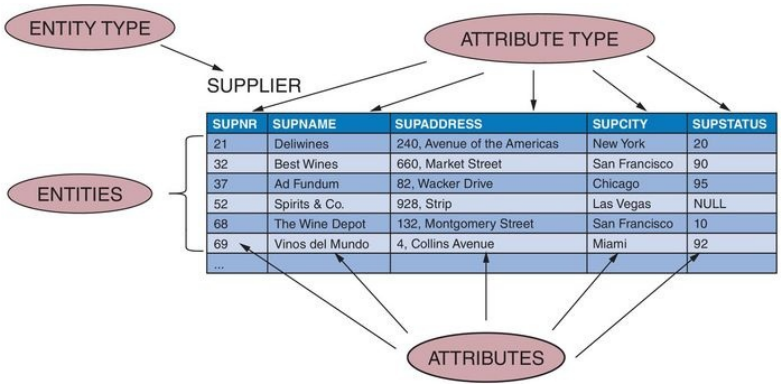
3.2.1 Entity Types

An entity type represents a business concept with an unambiguous meaning to a particular set of users. Examples of entity types are: supplier, student, product, or employee. An entity is one particular occurrence or instance of an entity type. Deliwines, Best Wines, and Ad Fundum are entities from the entity type supplier. In other words, an entity type defines a collection of entities that have similar characteristics. When building a conceptual data model, we focus on entity types and not on individual entities. In the ER model, entity types are depicted using a rectangle, as illustrated in Figure 3.2 for the entity type SUPPLIER.

Figure 3.2 The entity type SUPPLIER.

3.2.2 Attribute Types

An **attribute type** represents a property of an entity type. As an example, name and address are attribute types of the entity type supplier. A particular entity (e.g., Deliwines) has a value for each of its attribute types (e.g., its address is 240, Avenue of the Americas). An attribute type defines a collection of similar attributes, or an attribute is an instance of an attribute type. This is illustrated in Figure 3.3. The entity type SUPPLIER has attribute types SUPNR (supplier number), SUPNAME (supplier name), SUPADDRESS (supplier address), SUPCITY (supplier city), and SUPSTATUS (supplier status). Entities then correspond to specific suppliers such as supplier number 21, Deliwines, together with all its other attributes.



SUPNR	SUPNAME	SUPADDRESS	SUPCITY	SUPSTATUS
21	Deliwines	240, Avenue of the Americas	New York	20
32	Best Wines	660, Market Street	San Francisco	90
37	Ad Fundum	82, Wacker Drive	Chicago	95
52	Spirits & Co.	928, Strip	Las Vegas	NULL
68	The Wine Depot	132, Montgomery Street	San Francisco	10
69	Vinos del Mundo	4, Collins Avenue	Miami	92
...				

Figure 3.3 Entity relationship model: basic concepts.

In the ER model, we focus on attribute types and represent them using ellipses, as illustrated in Figure 3.4 for the entity type SUPPLIER and attribute types SUPNR, STATUS, and DATE OF BIRTH.

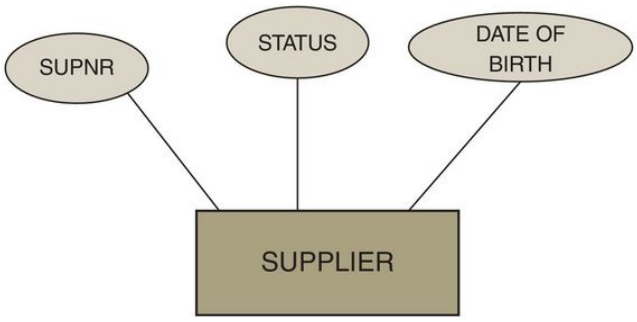


Figure 3.4 The entity type SUPPLIER with attribute types SUPNR, STATUS, and DATE OF BIRTH.

In the following subsections we elaborate on attribute types and discuss domains, key attribute types, simple versus composite attribute types, single-valued versus multi-valued attribute types and derived attribute types.

3.2.3.1 Domains

A **domain** specifies the set of values that may be assigned to an attribute for each individual entity. A domain for gender can be specified as having only two values: male and female. Likewise, a date domain can define dates as day, followed by month, followed by year. A domain can also contain null values. A null value means that a value is not known, not applicable, or not relevant. It is thus not the same as the value 0 or as an empty string of text “”. Think about a domain email address that allows for null values in case the email address is not known. By convention, domains are not displayed in an ER model.

3.2.3.2 Key Attribute Types

A **key attribute type** is an attribute type whose values are distinct for each individual entity. In other words, a key attribute type can be used to uniquely identify each entity. Examples are: supplier number, which is unique for each supplier; product number, which is unique for each product; and social security

number, which is unique for each employee. A key attribute type can also be a combination of attribute types. As an example, suppose a flight is identified by a flight number. However, the same flight number is used on each day to represent a particular flight. In this case, a combination of flight number and departure date is needed to uniquely identify flight entities. It is clear from this example that the definition of a key attribute type depends upon the business setting. Key attribute types are underlined in the ER model, as illustrated in Figure 3.5.

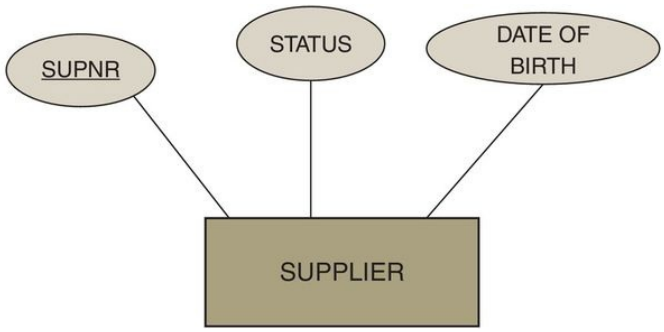


Figure 3.5 The entity type SUPPLIER with key attribute type SUPNR.

3.2.3.3 Simple versus Composite Attribute Types

A **simple or atomic attribute type** cannot be further divided into parts. Examples are supplier number or supplier status. A **composite attribute type** is an attribute type that can be decomposed into other meaningful attribute types. Think about an address attribute type, which can be further decomposed into attribute types for street, number, ZIP code, city, and country. Another example is name, which can be split into first name and last name. Figure 3.6 illustrates how the composite attribute types address and name are represented in the ER model.

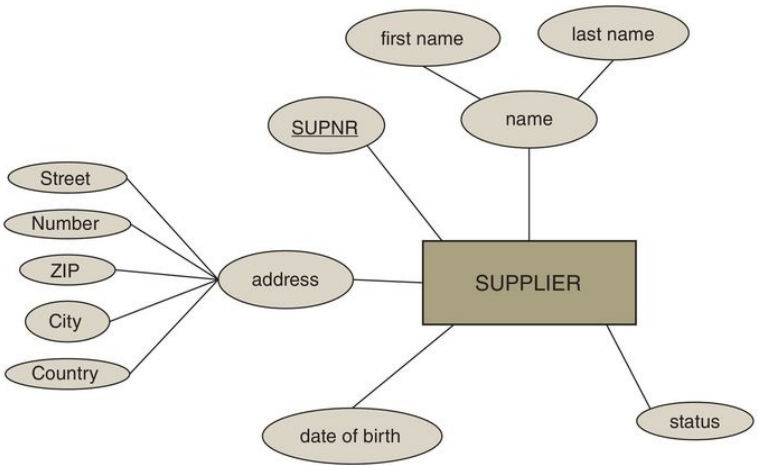


Figure 3.6 The entity type SUPPLIER with composite attribute types address and name.

3.2.3.4 Single-Valued versus Multi-Valued Attribute Types

A **single-valued attribute** type has only one value for a particular entity. An example is product number or product name. A **multi-valued attribute type** is an attribute type that can have multiple values. As an example, email address can be a multi-valued attribute type as a supplier can have multiple email addresses. Multi-valued attribute types are represented using a double ellipse in the ER model, as illustrated in Figure 3.7.

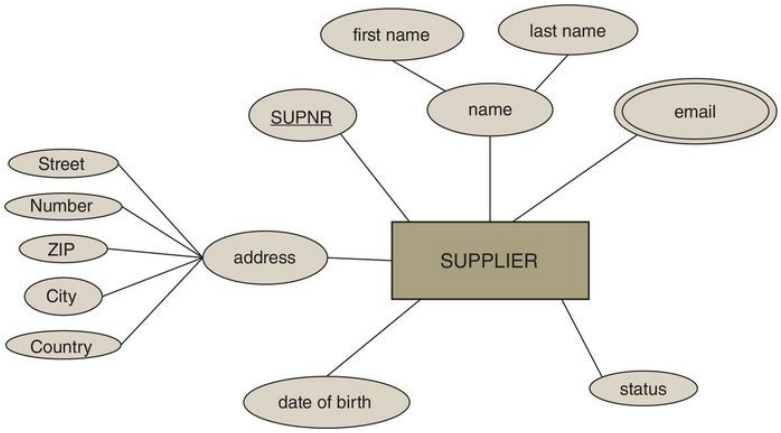


Figure 3.7 The entity type SUPPLIER with multi-valued attribute type email.

3.2.3.5 Derived Attribute Type

A **derived attribute type** is an attribute type that can be derived from another attribute type. As an example, age is a derived attribute type since it can be derived from birth date. Derived attribute types are depicted using a dashed ellipse, as shown in Figure 3.8.

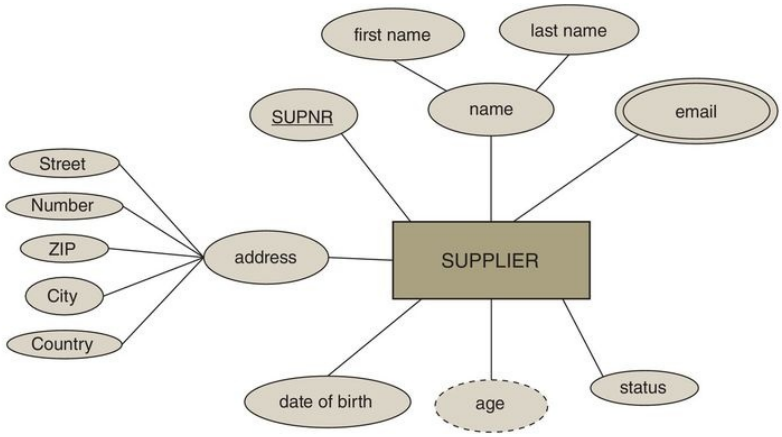


Figure 3.8 The entity type SUPPLIER with derived attribute type age.

3.2.4 Relationship Types

A **relationship** represents an association between two or more entities. Consider a particular supplier (e.g., Deliwines) supplying a set of products (e.g., product numbers 0119, 0178, 0289, etc.). A **relationship type** then defines a set of relationships among instances of one, two, or more entity types. In the ER model, relationship types are indicated using a rhombus symbol (see Figure 3.9). The rhombus can be thought of as two adjacent arrows pointing to each of the entity types specifying both directions in which the relationship type can be interpreted. Figure 3.9 shows the relationship type SUPPLIES between the entity types SUPPLIER and PRODUCT. A supplier can supply products (as indicated by the downwards arrow) and a product can be supplied by suppliers (as indicated by the upwards arrow). Each relationship instance of the SUPPLIES relationship type relates one particular supplier instance to one particular product instance. However, similar to entities and attributes, individual relationship instances are not represented in an ER model.

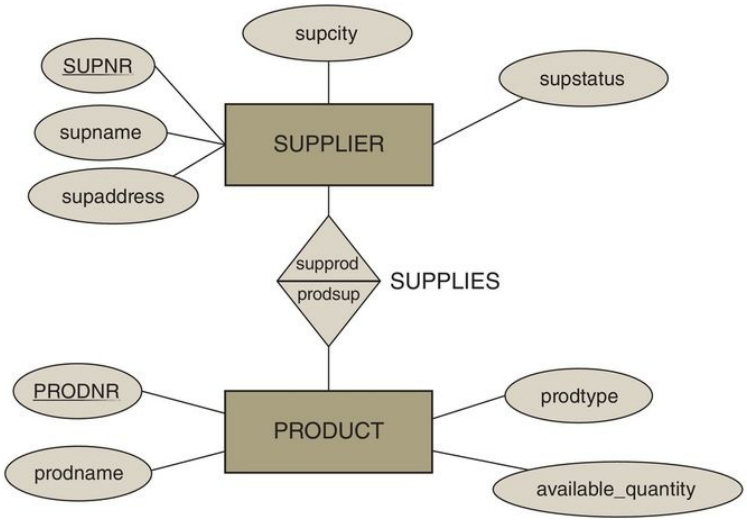


Figure 3.9 Relationship type in the ER model.

In the following subsections we elaborate on various characteristics of relationship types, such as degree and roles, cardinalities, and relationship attribute types.

3.2.4.1 Degree and Roles

The **degree** of a relationship type corresponds to the number of entity types participating in the relationship type. A unary or recursive relationship type has degree one. A binary relationship type has two participating entity types whereas a ternary relationship type has three participating entity types. The **roles** of a relationship type indicate the various directions that can be used to interpret it. Figure 3.9 represents a binary relationship type since it has two participating entity types (SUPPLIER and PRODUCT). Note the role names (supprod and prodsup) that we have added in each of the arrows making up the rhombus symbol.

Figures 3.10 and 3.11 show two other examples of relationship types. The SUPERVISES relationship type is a unary or recursive relationship type, which models the hierarchical relationships between employees. In general, the instances of a unary relationship relate two instances of the same entity type to one another. The role names *supervises* and *supervised by* are added for further clarification. The second example is an example of a ternary relationship type BOOKING between the entity types TOURIST, HOTEL, and TRAVEL AGENCY. Each relationship instance represents the interconnection between one particular tourist, hotel, and travel agency. Role names can also be added but this is less straightforward here.

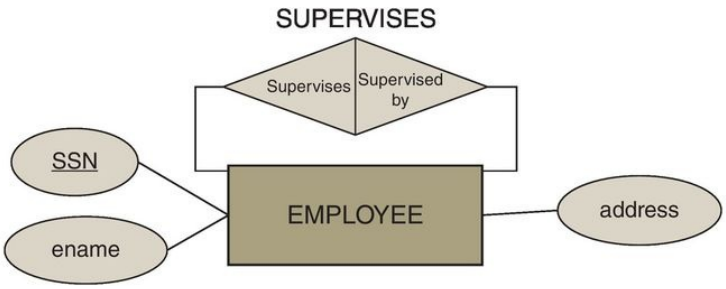


Figure 3.10 Unary ER relationship type.

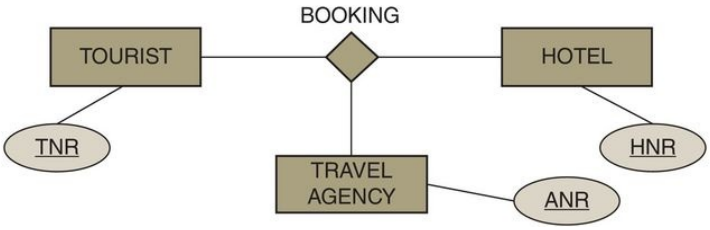


Figure 3.11 Ternary ER relationship type.

3.2.4.2 Cardinalities

Every relationship type can be characterized in terms of its **cardinalities**, which specify the minimum or maximum number of relationship instances that an individual entity can participate in. The minimum cardinality can either be 0 or 1. If it is 0, it implies that an entity can occur without being connected through that relationship type to another entity. This can be referred to as **partial participation** since some entities may not participate in the relationship. If the minimum cardinality is 1, an entity must always be connected to at least one other entity through an instance of the relationship type. This is referred to as **total participation** or **existence dependency**, since all entities need to participate in the relationship, or in other words, the existence of the entity depends upon the existence of another.

The maximum cardinality can either be 1 or N. In the case that it is 1, an entity can be involved in only one instance of that relationship type. In other words, it can be connected to at most one other entity through that relationship

type. In case the maximum cardinality is N, an entity can be connected to at most N other entities by means of the relationship type. Note that N represents an arbitrary integer number bigger than 1.

Relationship types are often characterized according to the maximum cardinality for each of their roles. For binary relationship types, this gives four options: 1:1, 1:N, N:1, and M:N.

Figure 3.12 illustrates some examples of binary relationship types together with their cardinalities. A student can be enrolled for a minimum of one course and a maximum of M courses. Conversely, a course can have minimum zero and maximum N students enrolled. This is an example of an N:M relationship type (also called many-to-many relationship type). A student can be assigned to minimum zero and maximum one master's thesis. A master's thesis is assigned to minimum zero and maximum one student. This is an example of a 1:1 relationship type. An employee can manage minimum zero and maximum N projects. A project is managed by minimum one and maximum one, or in other words exactly one employee. This is an example of a 1:N relationship type (also called one-to-many relationship type).

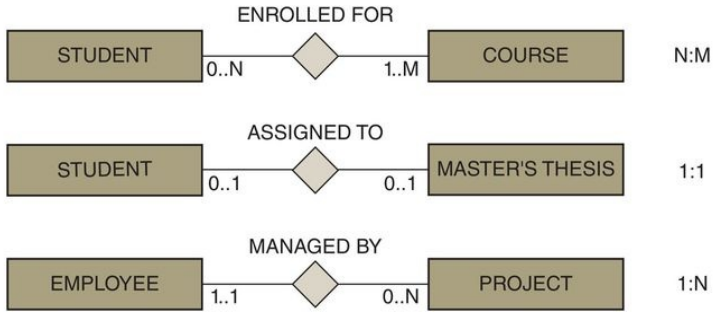


Figure 3.12 ER relationship types: examples.

3.2.4.3 Relationship Attribute Types

Like entity types, a relationship type can also have attribute types. These attribute types can be migrated to one of the participating entity types in case of a 1:1 or 1:N relationship type. However, in the case of an M:N relationship type, the attribute type needs to be explicitly specified as a relationship attribute type.

This is illustrated in Figure 3.13. The attribute type *hours* represents the number of hours an employee worked on a project. Its value cannot be considered as the sole property of an employee or of a project; it is uniquely determined by a combination of an employee instance and project instance – hence, it needs to be modeled as an attribute type of the WORKS ON relationship type which connects employees to projects.

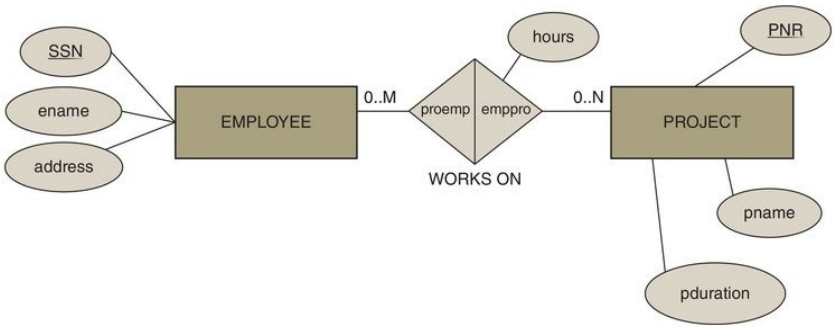


Figure 3.13 Relationship type with attribute type.

3.2.5 Weak Entity Types

A **strong entity** type is an entity type that has a key attribute type. In contrast, a **weak entity type** is an entity type that does not have a key attribute type of its own. More specifically, entities belonging to a weak entity type are identified by being related to specific entities from the **owner entity type**, which is an entity type from which they borrow an attribute type. The borrowed attribute type is then combined with some of the weak entity's own attribute types (also called partial keys) into a key attribute type. Figure 3.14 shows an ER model for a hotel administration.

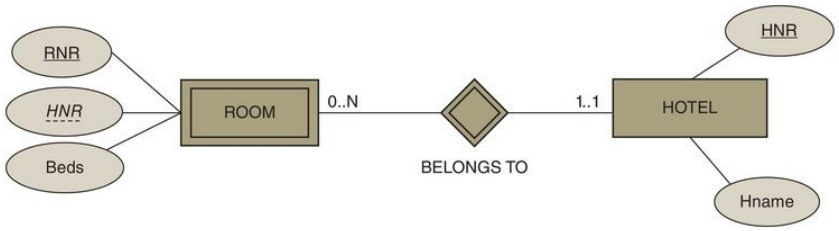


Figure 3.14 Weak entity types in the ER model.

A hotel has a hotel number (HNR) and a hotel name (Hname). Every hotel has a unique hotel number. Hence, HNR is the key attribute type of Hotel. A room is identified by a room number (RNR) and a number of beds (Beds). Within a particular hotel, each room has a unique room number but the same room number can occur for multiple rooms in different hotels. Hence, RNR as such does not suffice as a key attribute type. Consequently, the entity type ROOM is a weak entity type since it cannot produce its own key attribute type. More specifically, it needs to borrow HNR from HOTEL to come up with a key attribute type which is now a combination of its partial key RNR and HNR. Weak entity types are represented in the ER model using a double-lined rectangle, as illustrated in Figure 3.14. The rhombus representing the

relationship type through which the weak entity type borrows a key attribute type is also double-lined. The borrowed attribute type(s) is/are underlined using a dashed line.

Since a weak entity type needs to borrow an attribute type from another entity type, its existence will always be dependent on the latter. For example, in Figure 3.14, ROOM is existence-dependent on HOTEL, as also indicated by the minimum cardinality of 1. Note, however, that an existence-dependent entity type does not necessarily imply a weak entity type. Consider the example in Figure 3.15. The PURCHASE ORDER entity type is existence-dependent on SUPPLIER, as indicated by the minimum cardinality of 1. However, in this case PURCHASE ORDER has its own key attribute type, which is purchase order number (PONR). In other words, PURCHASE ORDER is an existence-dependent entity type but not a weak entity type.

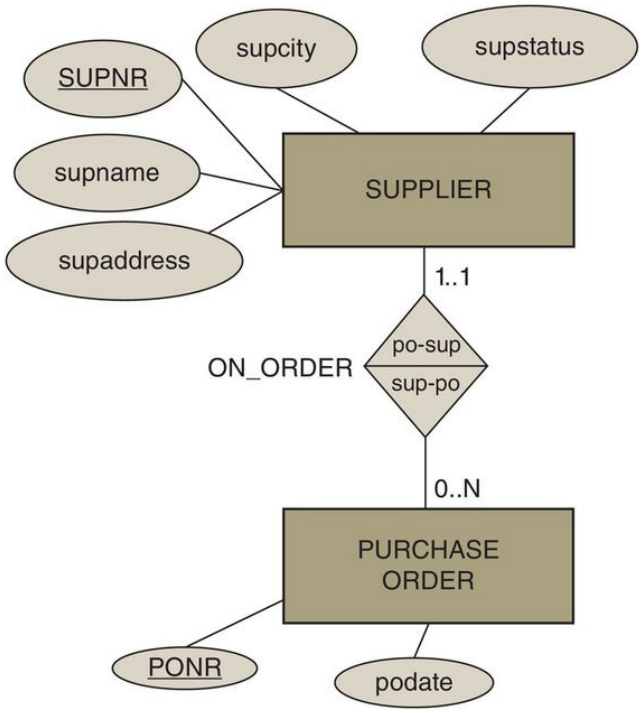


Figure 3.15 Weak versus existence-dependent entity type in the ER model.

3.2.6 Ternary Relationship Types

The majority of relationship types in an ER model are binary or have only two participating entity types. However, higher-order relationship types with more than two entity types, known as **ternary relationship types**, can occasionally occur, and special attention is needed to properly understand their meaning.

Assume that we have a situation in which suppliers can supply products for projects. A supplier can supply a particular product for multiple projects. A product for a particular project can be supplied by multiple suppliers. A project can have a particular supplier supply multiple products. The model must also include the quantity and due date for supplying a particular product to a particular project by a particular supplier. This is a situation that can be perfectly modeled using a ternary relationship type, as you can see in Figure 3.16.

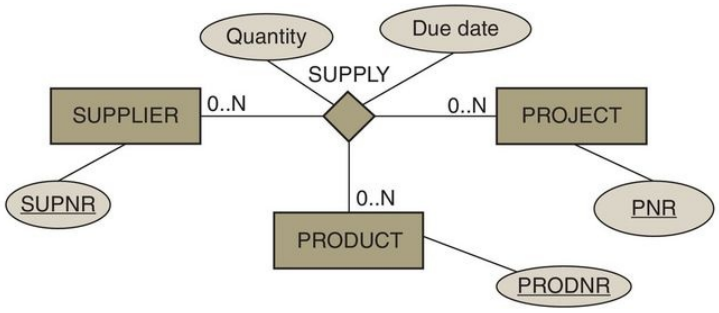


Figure 3.16 Ternary relationship type: example.

A supplier can supply a particular product for 0 to N projects. A product for a particular project can be supplied by 0 to N suppliers. A supplier can supply 0 to N products for a particular project. The relationship type also includes the quantity and due date attribute types.[1]

An obvious question is whether we can also model this ternary relationship type as a set of binary relationship types, as shown in Figure 3.17.

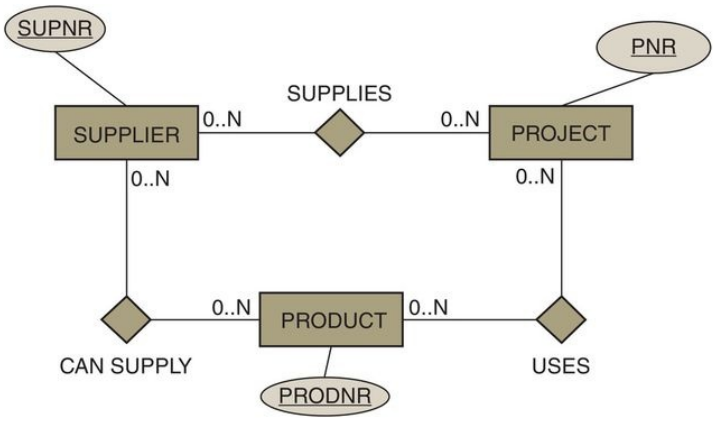


Figure 3.17 Ternary versus binary relationship types.

We decomposed the ternary relationship type into the binary relationship types “SUPPLIES” between SUPPLIER and PROJECT, “CAN SUPPLY” between SUPPLIER and PRODUCT, and “USES” between PRODUCT and PROJECT. We can now wonder whether the semantics of the ternary relationship type is preserved by these binary relationship types. To properly understand this, we need to write down some relationship instances. Say we have two projects: Project 1 uses a pencil and a pen, and Project 2 uses a pen. Supplier Peters supplies the pencil for Project 1 and the pen for Project 2, whereas supplier Johnson supplies the pen for Project 1.

Figure 3.18 shows the relationship instances for both cases. At the top of the figure are the relationship instances that would be used in a ternary relationship type “SUPPLY”. This can be deconstructed into the three binary relationship types: “SUPPLIES”, “USES”, and “CAN SUPPLY”.

SUPPLY

Supplier	Product	Project
Peters	Pencil	Project 1
Peters	Pen	Project 2
Johnson	Pen	Project 1

SUPPLIES

Supplier	Project
Peters	Project 1
Peters	Project 2
Johnson	Project 1

USES

Product	Project
Pencil	Project 1
Pen	Project 1
Pen	Project 2

CAN SUPPLY

Supplier	Product
Peters	Pencil
Peters	Pen
Johnson	Pen

Figure 3.18 Ternary versus binary relationship types: example instances.

From the “SUPPLIES” relationship type, we can see that both Peters and Johnson supply to Project 1. From the “CAN SUPPLY” relationship type, we can see that both can also supply a pen. The “USES” relationship type indicates that Project 1 needs a pen. Hence, from the binary relationship types, it is not clear who supplies the pen for Project 1. This is, however, clear in the ternary relationship type, where it can be seen that Johnson supplies the pen for Project 1. By decomposing the ternary relationship types into binary relationship types, we clearly lose semantics. Furthermore, when using binary relationship types, it is also unclear where we should add the relationship attribute types such as quantity and due date (see Figure 3.16). Binary relationship types can, however, be used to model additional semantics.

Figure 3.19 shows another example of a ternary relationship type between three entity types: INSTRUCTOR with key attribute type INR representing the instructor number; COURSE with key attribute type CNR representing the course number; and SEMESTER with key attribute type SEM-YEAR representing the semester year. An instructor can offer a course during zero to N semesters. A course during a semester is offered by one to N instructors. An instructor can offer zero to N courses during a semester. In this case, we also added an extra binary relationship type QUALIFIED between INSTRUCTOR and COURSE to indicate what courses an instructor is qualified to teach. Note that, in this way, it is possible to model the fact that an instructor may be qualified for more courses than the ones she/he is actually teaching at the moment.

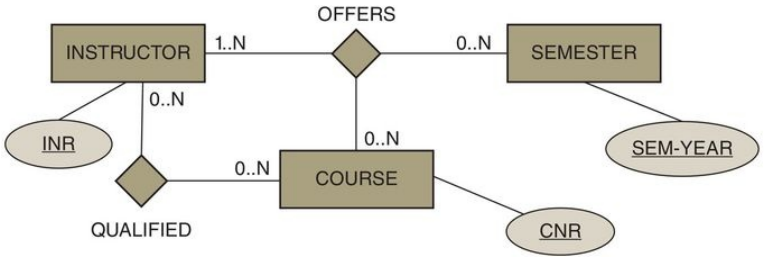


Figure 3.19 Ternary relationship type in the ER model.

Another alternative to model a ternary relationship type is by using a weak entity type as shown in Figure 3.20. The weak entity type SUPPLY is existence-dependent on SUPPLIER, PRODUCT, and PROJECT, as indicated by the minimum cardinalities of 1. Its key is a combination of supplier number, product number, and project number. It also includes the attribute types *quantity* and *due date*. Representing a ternary relationship type in this way can be handy in case the database modeling tool only supports unary and binary relationship types.

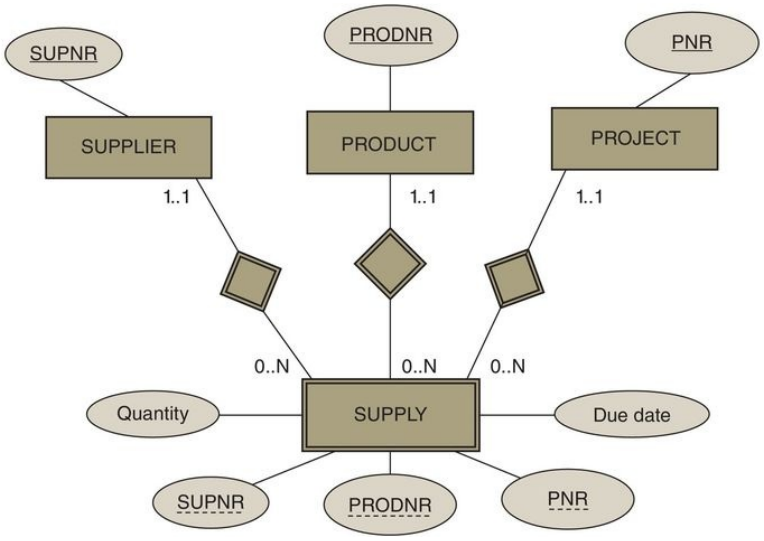


Figure 3.20 Modeling ternary relationship types as binary relationship types.

3.2.7 Examples of the ER Model

Figure 3.21 shows the ER model for a human resources (HR) administration. It has three entity types: EMPLOYEE, DEPARTMENT, and PROJECT. Let's read some of the relationship types. An employee works in minimum one and maximum one, so exactly one, department. A department has minimum one and maximum N employees working in it. A department is managed by exactly one employee. An employee can manage zero or one department. A department is in charge of zero to N projects. A project is assigned to exactly one department. An employee works on zero to N projects. A project is being worked on by zero to M employees. The relationship type WORKS ON also has an attribute type hours, representing the number of hours an employee worked on a project. Also note the recursive relationship type to model the supervision relationships between employees. An employee supervises zero to N employees. An employee is supervised by zero or one employees.

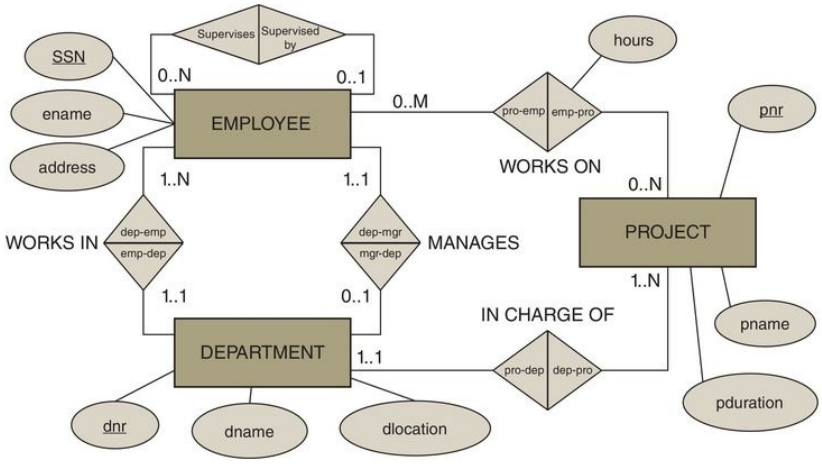


Figure 3.21 ER model for HR administration.

Figure 3.22 shows another example of an ER model for a purchase order administration. It has three entity types: SUPPLIER, PURCHASE ORDER, and

PRODUCT. A supplier can supply zero to N products. A product can be supplied by zero to M suppliers. The relationship type SUPPLIES also includes the attribute types purchase_price and deliv_period. A supplier can have zero to N purchase orders on order. A purchase order is always assigned to one supplier. A purchase order can have one to N purchase order lines with products. Conversely, a product can be included in zero to M purchase orders. In addition, the relationship type PO_LINE includes the quantity of the order. Also note the attribute types and key attribute types of each of the entity types.

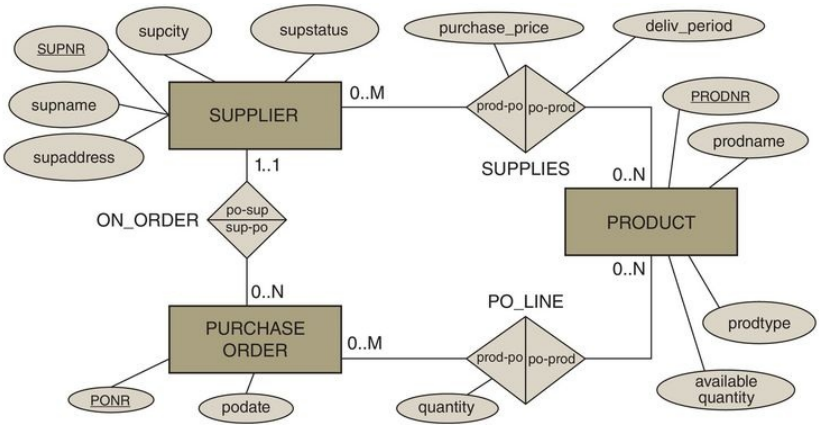


Figure 3.22 ER model for purchase order administration.

3.2.8 Limitations of the ER Model

Although the ER model is a very user-friendly data model for conceptual data modeling, it also has its limitations. First of all, the ER model presents a temporary snapshot of the data requirements of a business process. This implies that **temporal constraints**, which are constraints spanning a particular time interval, cannot be modeled. Some example temporal constraints that cannot be enforced are: a project needs to be assigned to a department after one month, an employee cannot return to a department of which he previously was a manager, an employee needs to be assigned to a department after six months, a purchase order must be assigned to a supplier after two weeks. These rules need to be documented and followed up with application code.

Another shortcoming is that the ER model cannot guarantee consistency across multiple relationship types. Some examples of business rules that cannot be enforced in the ER model are: an employee should work in the department that he/she manages, employees should work on projects assigned to departments to which the employees belong, and suppliers can only be assigned to purchase orders for products they can supply. Again, these business rules need to be documented and followed up with application code.

Furthermore, since domains are not included in the ER model, it is not possible to specify the set of values that can be assigned to an attribute type (e.g., hours should be positive; prodtype must be red, white, or sparkling, supstatus is an integer between 0 and 100). Finally, the ER model also does not support the definition of functions (e.g., a function to calculate an employee's salary).

Retention Questions

- What are the key building blocks of the ER model?
- Discuss the attribute types supported in the ER model.
- Discuss the relationship types supported in the ER model.
- What are weak entity types and how are they modeled in the ER model?
- Discuss the limitations of the ER model.

3.3 The Enhanced Entity Relationship (EER) Model

The **Enhanced Entity Relationship model or EER model** is an extension of the ER model. It includes all the modeling concepts (entity types, attribute types, relationship types) of the ER model, as well as three new additional semantic data modeling concepts: specialization/generalization, categorization, and aggregation. We discuss these in more detail in the following subsections.

3.3.1 Specialization/Generalization

The concept of **specialization** refers to the process of defining a set of subclasses of an entity type. The set of subclasses that form a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. As an example, consider an ARTIST superclass with subclasses SINGER and ACTOR. The specialization process defines an “IS A” relationship. In other words, a singer is an artist. Also, an actor is an artist. The opposite does not apply. An artist is not necessarily a singer. Likewise, an artist is not necessarily an actor. The specialization can then establish additional specific attribute types for each subclass. A singer can have a music style attribute type. During the specialization, it is also possible to establish additional specific relationship types between each subclass and other entity types. An actor can act in movies. A singer can be part of a band. A subclass inherits all attribute types and relationship types from its superclass.

Generalization, also called **abstraction**, is the reverse process of specialization. Specialization corresponds to a top-down process of conceptual refinement. As an example, the ARTIST entity type can be specialized or refined in the subclasses SINGER and ACTOR. Conversely, generalization corresponds to a bottom-up process of conceptual synthesis. As an example, the SINGER and ACTOR subclasses can be generalized in the ARTIST superclass.

Figure 3.23 shows how our specialization can be represented in the EER model. An artist has a unique artist number and an artist name. The ARTIST superclass is specialized in the subclasses SINGER and ACTOR. Both SINGER and ACTOR inherit the attribute types ANR and aname from ARTIST. A singer has a music style. An actor can act in zero to N movies. Conversely, in a movie one to M actors can act. A movie has a unique movie number and a movie title.

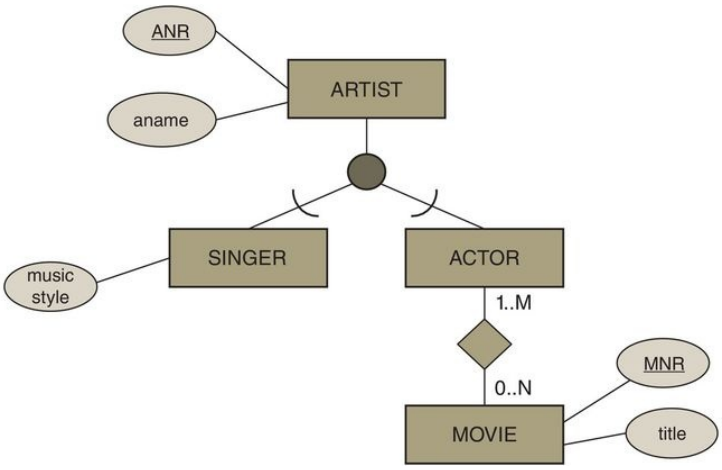


Figure 3.23 Example of EER specialization.

A specialization can be further qualified in terms of its disjointness and completeness constraints. The **disjointness constraint** specifies what subclasses an entity of the superclass can belong to. It can be set to either disjoint or overlap. A **disjoint specialization** is a specialization where an entity can be a member of at most one of the subclasses. An **overlap specialization** is a specialization where the same entity may be a member of more than one subclass. The **completeness constraint** indicates whether all entities of the superclass should belong to one of the subclasses or not. It can be set to either total or partial. A **total specialization** is a specialization where every entity in the superclass must be a member of some subclass. A **partial specialization** allows an entity to only belong to the superclass and to none of the subclasses. The disjointness and completeness constraints can be set independently, which gives four possible combinations: disjoint and total; disjoint and partial; overlapping and total; and overlapping and partial. Let's illustrate this with some examples.

Figure 3.24 gives an example of a partial specialization with overlap. The specialization is partial since not all artists are singers or actors; think about

painters, for example, which are not included in our EER model. The specialization is overlap since some artists can be both singers and actors.

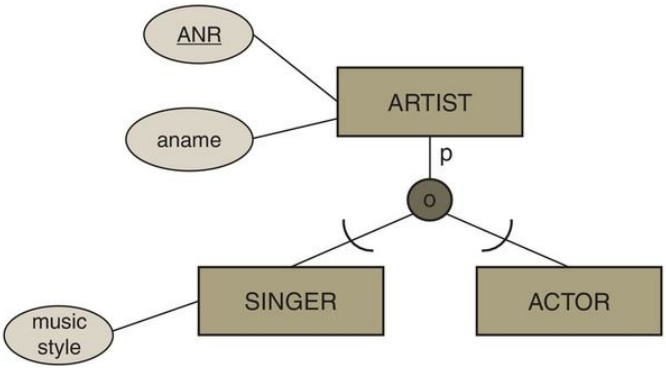


Figure 3.24 Example of partial (p) specialization with overlap (o).

Figure 3.25 illustrates a total disjoint specialization. The specialization is total, since according to our model all people are either students or professors. The specialization is disjoint, since a student cannot be a professor at the same time.

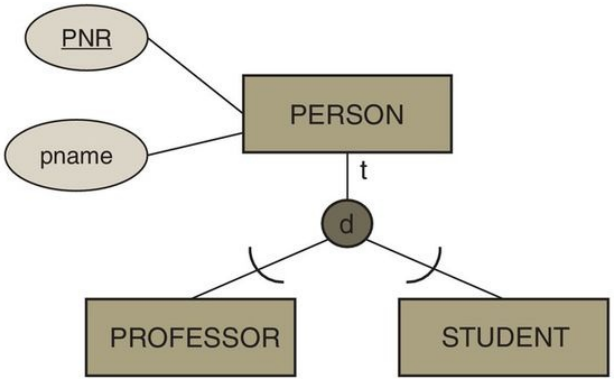


Figure 3.25 Example of total (t) and disjoint (d) specialization.

A specialization can be several levels deep: a subclass can again be a superclass of another specialization. In a specialization hierarchy, every subclass can only have a single superclass and inherits the attribute types and relationship types of all its predecessor superclasses all the way up to the root of the hierarchy. Figure 3.26 shows an example of a specialization hierarchy. The

STUDENT subclass is further specialized in the subclasses BACHELOR, MASTER, and PHD. Each of those subclasses inherits the attribute types and relationship types from STUDENT, which inherits both in turn from PERSON.

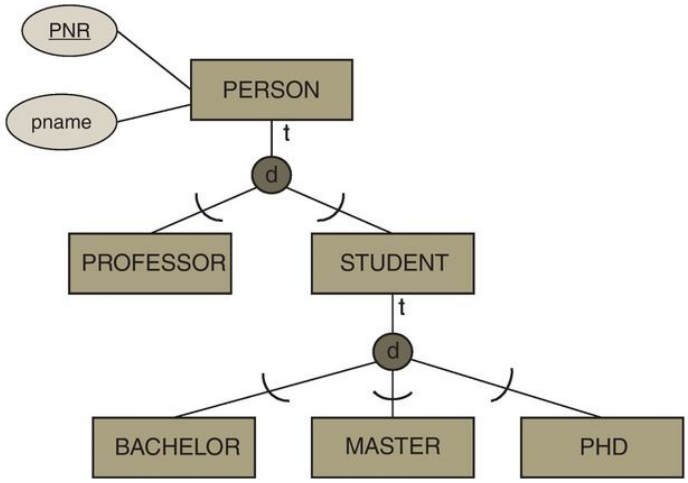


Figure 3.26 Example of specialization hierarchy.

In a specialization lattice, a subclass can have multiple superclasses. The concept in which a shared subclass or a subclass with multiple parents inherits from all of its parents is called multiple inheritance. Let's illustrate this with an example.

Figure 3.27 shows a specialization lattice. The VEHICLE superclass is specialized into MOTORCYCLE, CAR, and BOAT. The specialization is partial and with overlap. TRIKE is a shared subclass of MOTORCYCLE and CAR and inherits the attribute types and relationship types from both. Likewise, AMPHIBIAN is a shared subclass of CAR and BOAT and inherits the attribute types and relationship types from both.

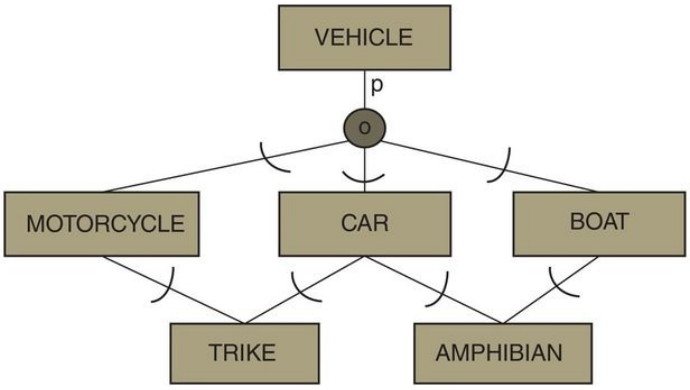


Figure 3.27 Example of specialization lattice.

3.3.2 Categorization

Categorization is the second important modeling extension of the EER model. A category is a subclass that has several possible superclasses. Each superclass represents a different entity type. The category then represents a collection of entities that is a subset of the union of the superclasses. Therefore, a categorization is represented in the EER model by a circle containing the letter "u" (from union) (see Figure 3.28).

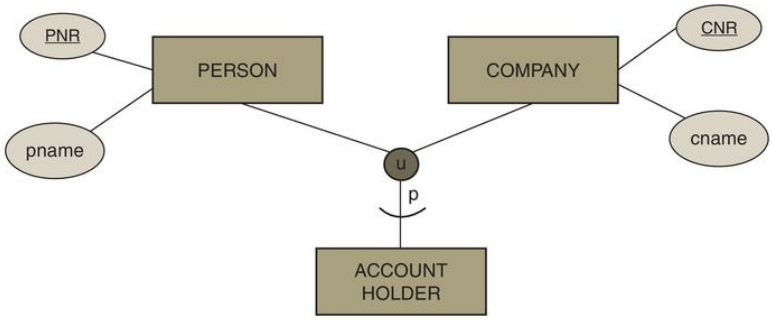


Figure 3.28 EER categorization.

Inheritance in the case of categorization corresponds to an entity inheriting only the attributes and relationships of that superclass of which it is a member. This is also referred to as **selective inheritance**. Similar to a specialization, a categorization can be *total* or *partial*. In a **total categorization**, all entities of the superclasses belong to the subclass. In a **partial categorization**, not all entities of the superclasses belong to the subclass. Let's illustrate this with an example.

Figure 3.28 shows how the superclasses PERSON and COMPANY have been categorized into an ACCOUNT HOLDER subclass. In other words, the account holder entities are a subset of the union of the person and company entities. Selective inheritance in this example implies that some account holders inherit their attributes and relationships from person, whereas others inherit them from company. The categorization is partial as represented by the letter "p". This

implies that not all persons or companies are account holders. If the categorization had been total (which would be represented by the letter “t” instead), then this would imply that all person and company entities are also account holders. In that case, we can also model this categorization using a specialization with ACCOUNT HOLDER as the superclass and PERSON and COMPANY as the subclasses.

3.3.3 Aggregation

Aggregation is the third modeling extension provided by the EER model. The idea here is that entity types that are related by a particular relationship type can be combined or aggregated into a higher-level aggregate entity type. This can be especially useful when the aggregate entity type has its own attribute types and/or relationship types.

Figure 3.29 gives an example of aggregation. A consultant works on zero to N projects. A project is being worked on by one to M consultants. Both entity types and the corresponding relationship type can now be aggregated into the aggregate concept PARTICIPATION. This aggregate has its own attribute type, date, which represents the date at which a consultant started working on a project. The aggregate also participates in a relationship type with CONTRACT. Participation should lead to a minimum of one and maximum of one contract. Conversely, a contract can be based upon one to M participations of consultants in projects.

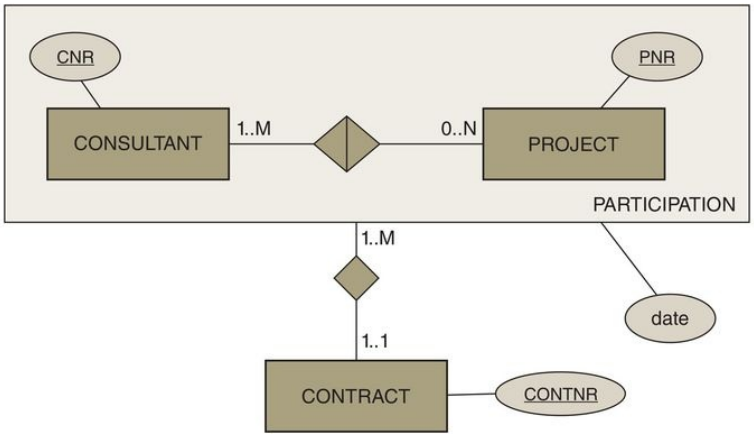


Figure 3.29 EER aggregation.

3.3.4 Examples of the EER Model

Figure 3.30 presents our earlier HR administration example (see Figure 3.21), but now enriched with some EER modeling concepts. More specifically, we partially specialized EMPLOYEE into MANAGER. The relationship type MANAGES then connects the MANAGER subclass to the DEPARTMENT entity type. DEPARTMENT and PROJECT have been aggregated into ALLOCATION. This aggregate then participates in the relationship type WORKS ON with EMPLOYEE.[2]

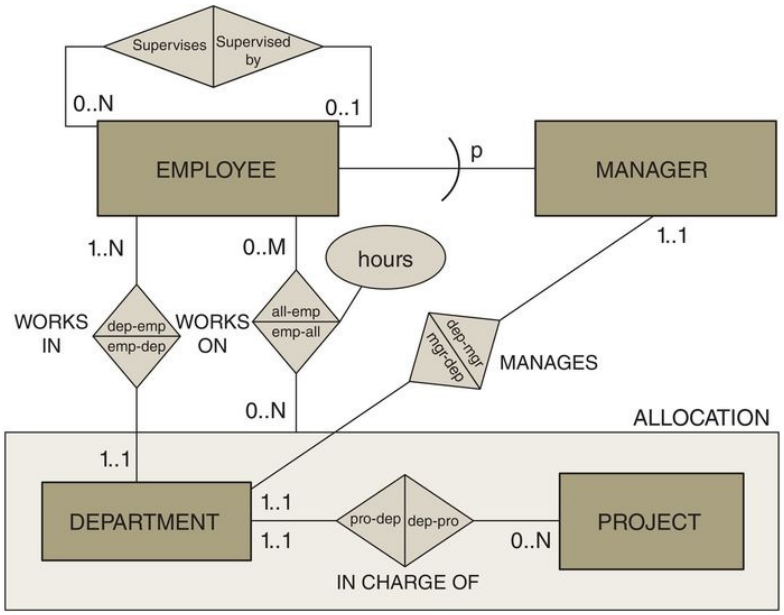


Figure 3.30 EER model for HR administration.

3.3.5 Designing an EER Model

To summarize, an EER conceptual data model can be designed according to the following steps:

1. Identify the entity types.

2. Identify the relationship types and assert their degree.

3. Assert the cardinality ratios and participation constraints (total versus partial participation).

4. Identify the attribute types and assert whether they are simple or composite, single- or multi-valued, derived or not.

5. Link each attribute type to an entity type or a relationship type.

6. Denote the key attribute type(s) of each entity type.

7. Identify the weak entity types and their partial keys.

8. Apply abstractions such as generalization/specialization, categorization, and aggregation.

9. Assert the characteristics of each abstraction such as disjoint or overlapping, total or partial.

Any semantics that cannot be represented in the EER model must be documented as separate business rules and followed up using application code. Although the EER model offers some new interesting modeling concepts such as specialization/generalization, categorization, and aggregation, the limitations of the ER model unfortunately still apply. Hence, temporal constraints still cannot be modeled, the consistency among multiple relationship types cannot be enforced and attribute type domains or functions cannot be specified. Some of

these shortcomings are addressed in the UML class diagram, which is discussed in the next section.

Retention Questions

- What modeling extensions are provided by the EER model? Illustrate with examples.
- What are the limitations of the EER model?