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

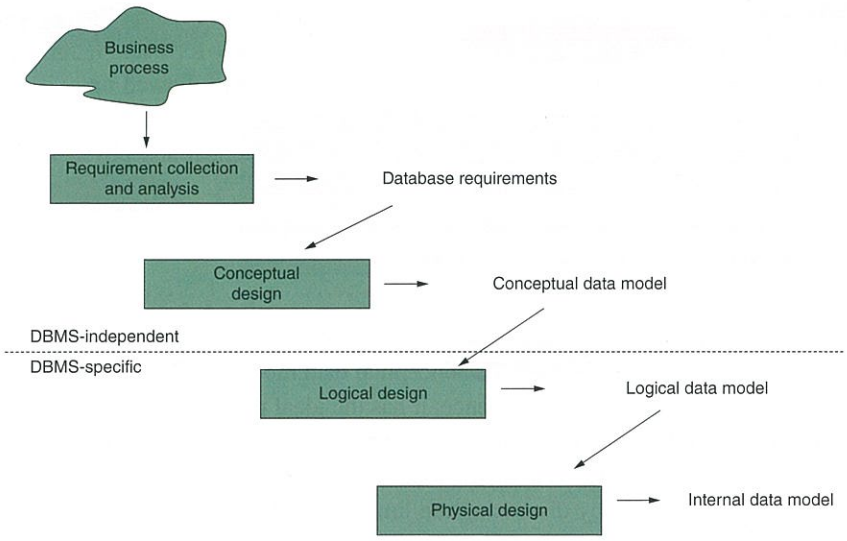


Figure 3.1 The database design process.

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.

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.

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.

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.

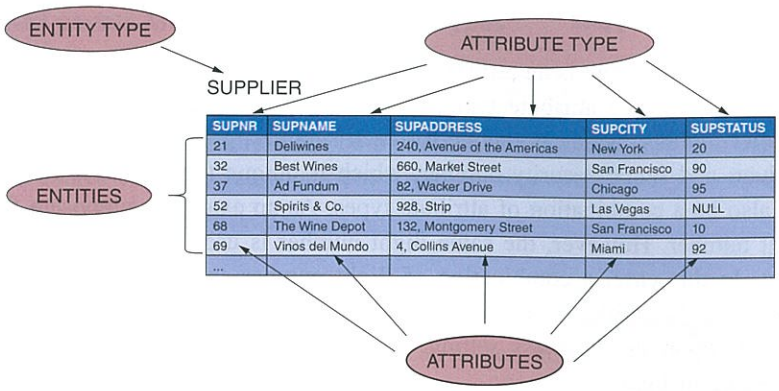
3.2.1 Entity Types

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.

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

Figure 3.2 The entity type SUPPLIER.



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.

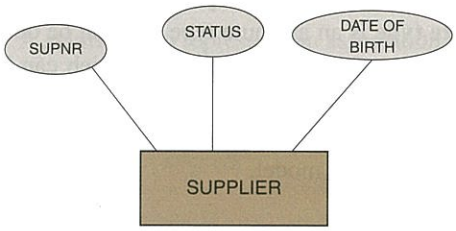


Figure 3.4 The entity type SUPPLIER with attribute types SUPNR, STATUS, and DATE OF BIRTH.

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.

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.

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.

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.

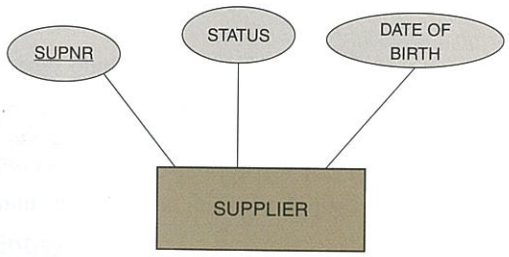


Figure 3.5 The entity type SUPPLIER with key attribute type SUPNR.

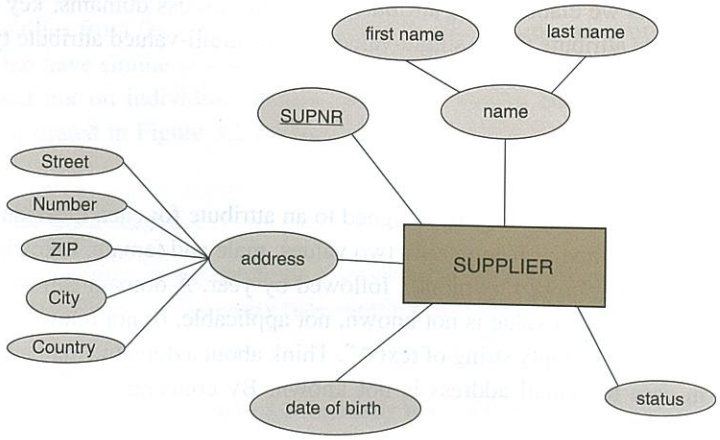


Figure 3.6 The entity type SUPPLIER with composite attribute types address and name.

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.

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.

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.).

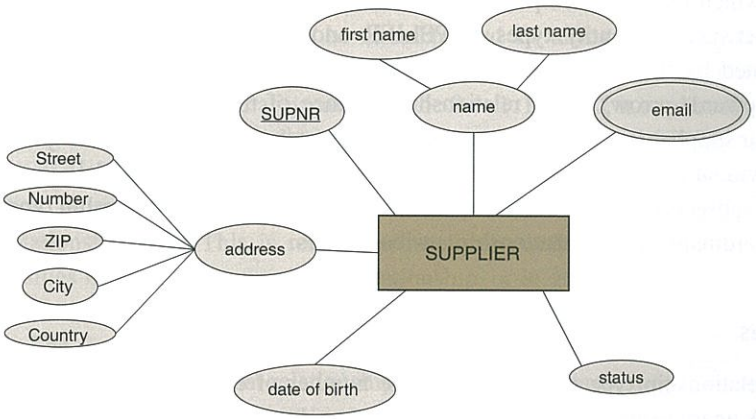


Figure 3.7 The entity type SUPPLIER with multi-valued attribute type email.

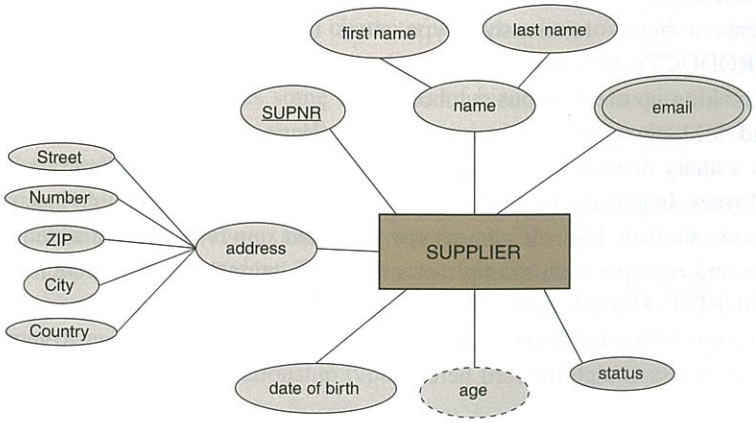


Figure 3.8 The entity type SUPPLIER with derived attribute type age.

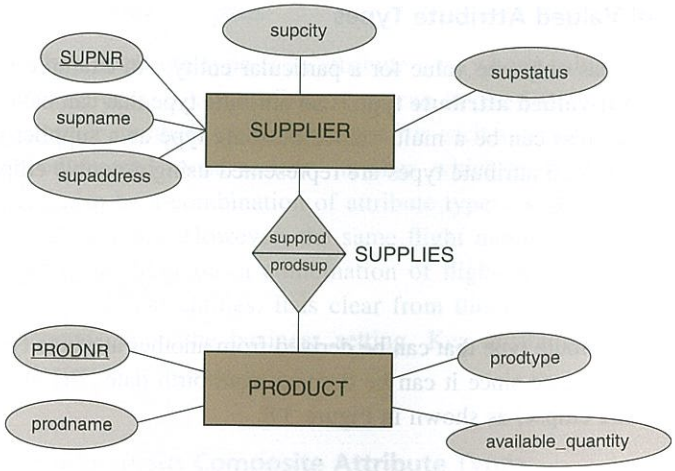


Figure 3.9 Relationship type in the ER model.

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.

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.

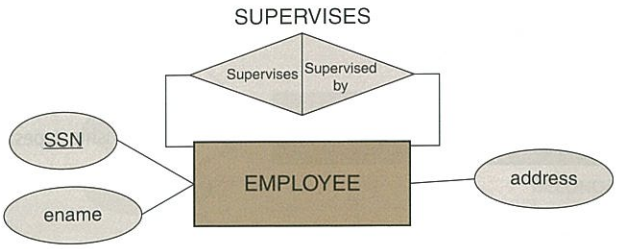


Figure 3.10 Unary ER relationship type.

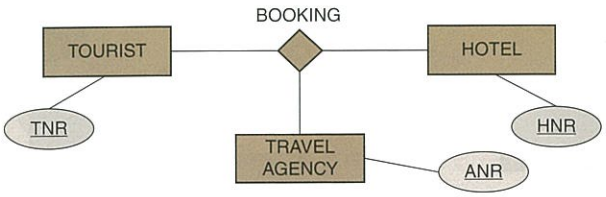


Figure 3.11 Ternary ER relationship type.

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).

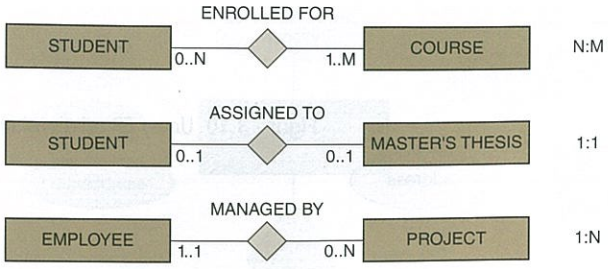


Figure 3.12 ER relationship types: examples.

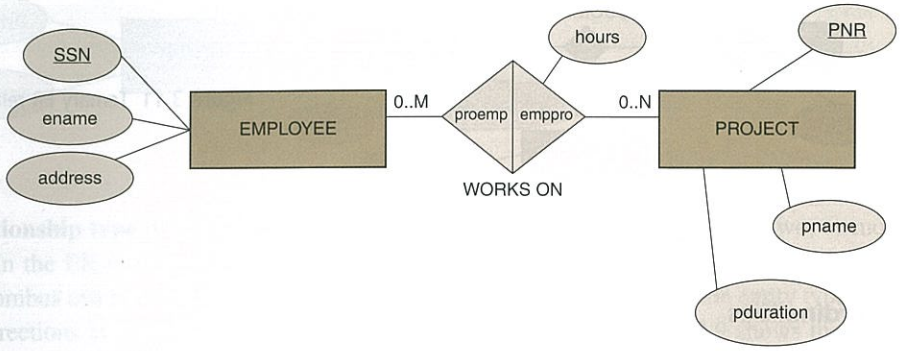


Figure 3.13 Relationship type with attribute type.

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.

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.

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)

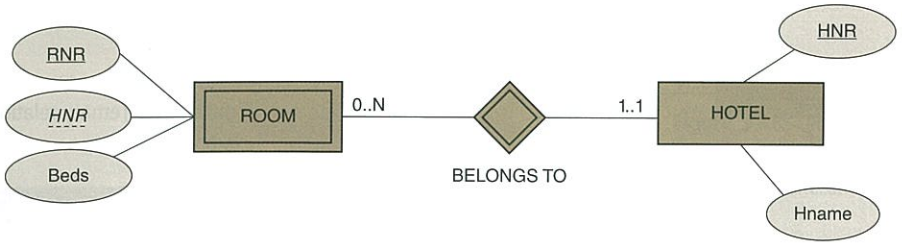


Figure 3.14 Weak entity types in the ER model.

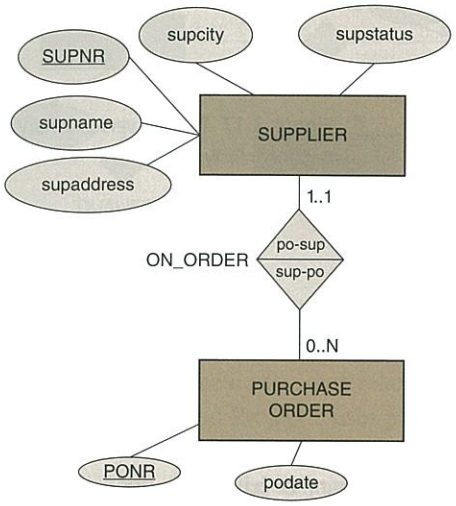


Figure 3.15 Weak versus existence-dependent entity type in the ER model.

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.

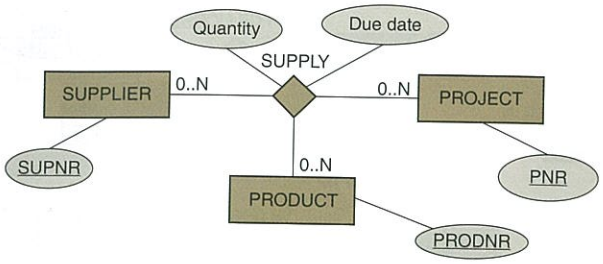


Figure 3.16 Ternary relationship type: example.

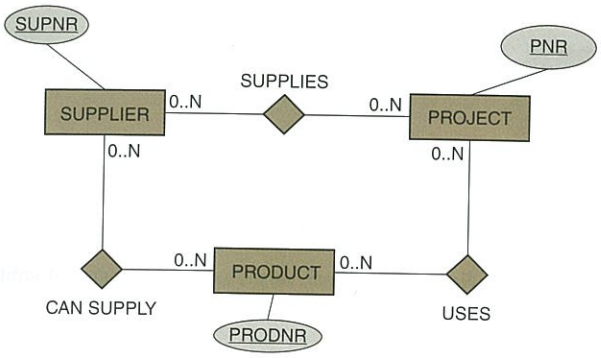


Figure 3.17 Ternary versus binary relationship types.

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.

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.

We decomposed the ternary relationship type into the binary relationship types "SUPPLIES" between SUPPLIER and PROJECT, "CAN SUPPLY" between SUPPLIER and PRODUCT, and

[1] Some textbooks put the cardinalities of each entity type next to the entity type itself instead of at the opposite side as we do. For ternary relationship types, this makes the notation less ambiguous. However, we continue to use our notation because this is the most commonly used.

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.

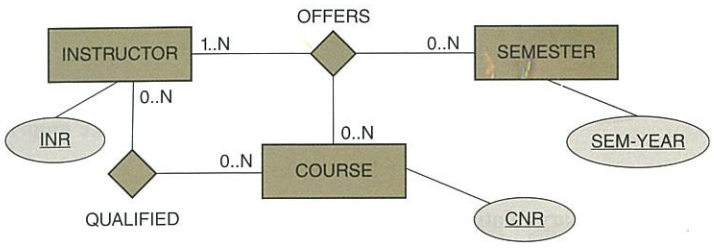


Figure 3.19 Ternary relationship type in the ER model.

"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".

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.

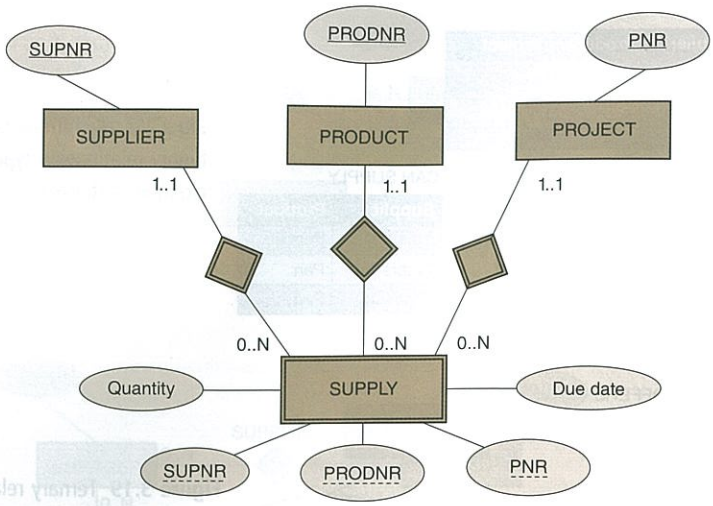


Figure 3.20 Modeling ternary relationship types as binary relationship types.

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.

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 3.22 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.

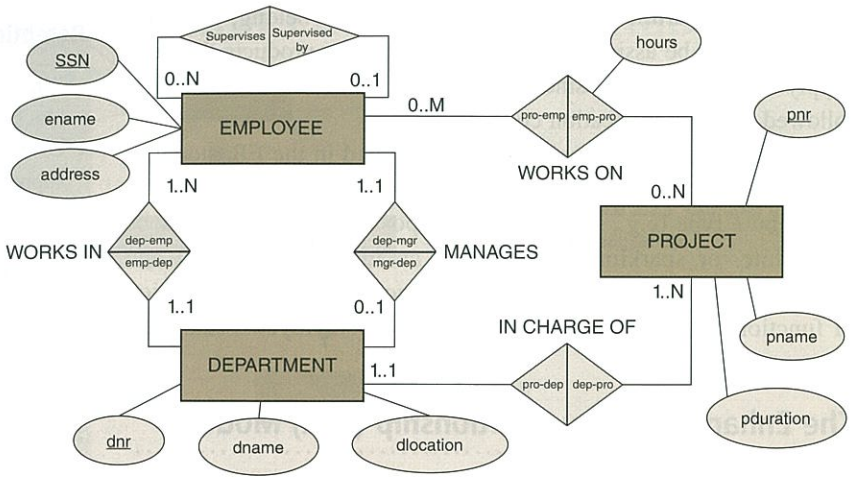


Figure 3.21 ER model for HR administration.

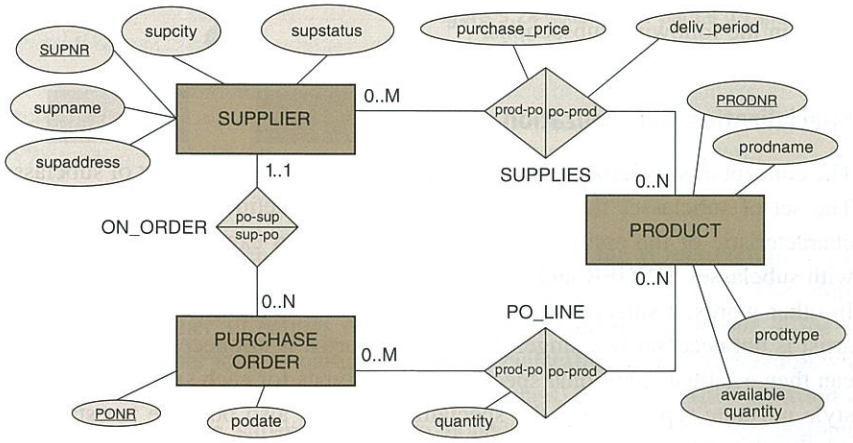


Figure 3.22 ER model for purchase order administration.

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.

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

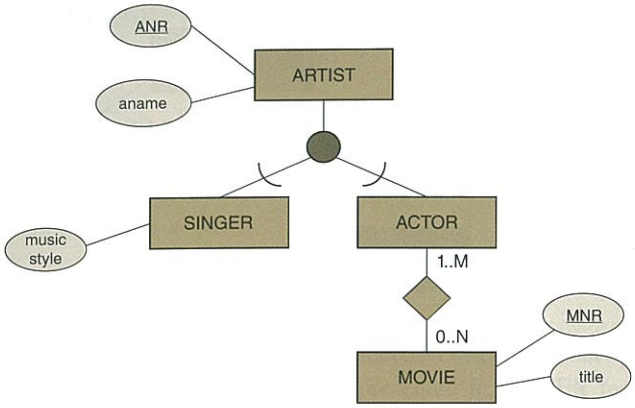


Figure 3.23 Example of EER specialization.

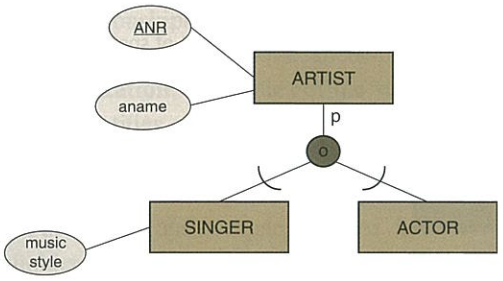


Figure 3.24 Example of partial (p) specialization with overlap (o).

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 3.25 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.

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.

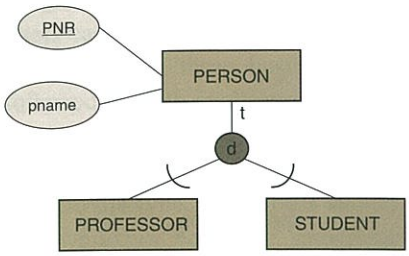


Figure 3.25 Example of total (t) and disjoint (d) specialization.

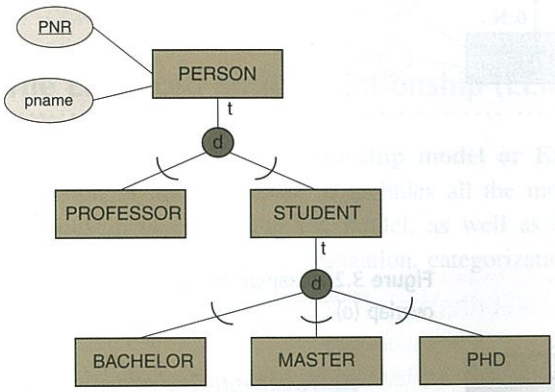


Figure 3.26 Example of specialization hierarchy.

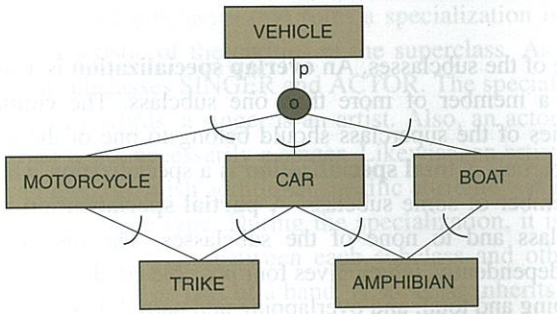


Figure 3.27 Example of specialization lattice.

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.

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.

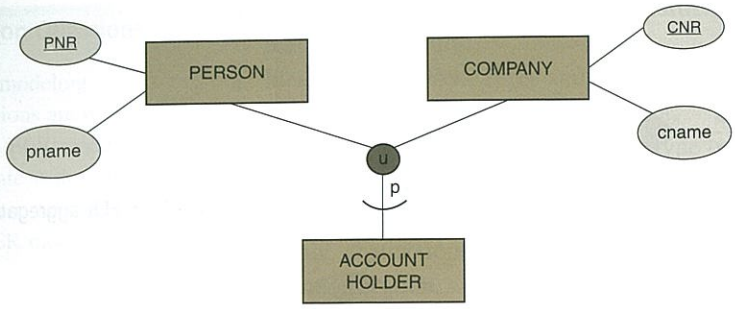


Figure 3.28 EER categorization.

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).

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.

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

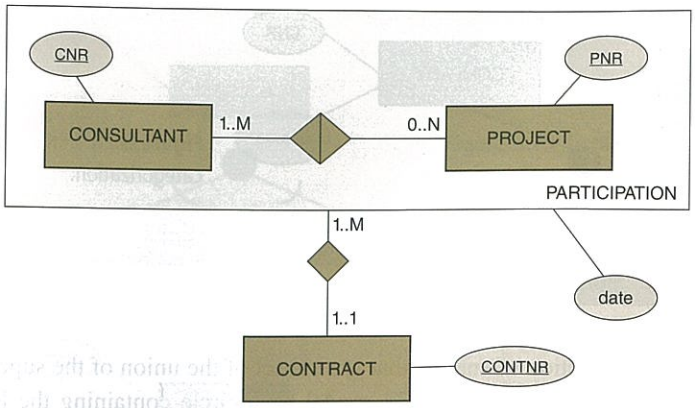


Figure 3.29 EER aggregation.

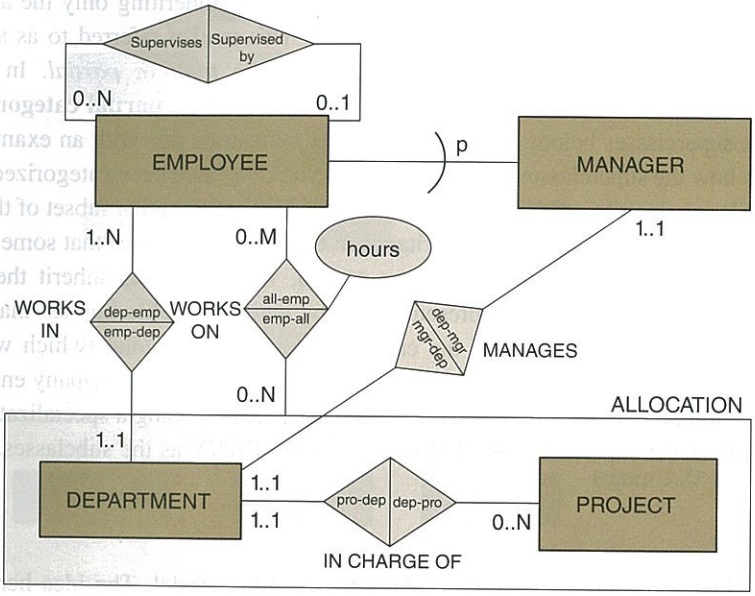


Figure 3.30 EER model for HR administration.

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]

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.

[2] We introduced the aggregation in the EER model for illustration purposes. However, since every project is assigned to exactly one department, we could also remove the aggregate ALLOCATION and draw a relationship type between EMPLOYEE and PROJECT.

Retention Questions

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

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.

3.4 The UML Class Diagram

The **Unified Modeling Language (UML)** is a modeling language that assists in the specification, visualization, construction, and documentation of artifacts of a software system.[3] UML is essentially an object-oriented system modeling notation which focuses not only on data requirements, but also on behavioral modeling, process, and application architecture. It was accepted as a standard by the Object Management Group (OMG) in 1997 and approved as an ISO standard in 2005. The most recent version is UML 2.5, introduced in 2015. To model both the data and process aspects of an information system, UML offers various diagrams such as use case diagrams, sequence diagrams, package diagrams, deployment diagrams, etc. From a database modeling perspective, the class diagram is the most important. It visualizes both classes and their associations. Before we discuss this in more detail, let's first provide a recap of object orientation (OO).

3.4.1 Recap of Object Orientation

Two important building blocks of OO are classes and objects. A **class** is a blueprint definition for a set of objects. Conversely, an **object** is an instance of a class. In other words, a class in OO corresponds to an entity type in ER, and an object to an entity. Each object is characterized by both variables and methods.[4] Variables correspond to attribute types and variable values to attributes in the EER model. The EER model has no equivalent to methods. You can think of an example class Student and an example object student Bart. For our student object, example variables could be the

[3] See www.omg.org/spec/UML/2.5 for the most recent version.

[4] In the UML model, variables are also referred to as attributes and methods as operations. However, to avoid confusion with the ER model (where attributes represent instances of attribute types), we will stick to the terms variables and methods.

Connections

We discuss object orientation in greater detail in Chapter 8.

student's name, gender, and birth date. Example methods could be calcAge, which calculates the age of the student based upon the birth date; isBirthday to verify whether the student's birthday is today; hasPassed(courseID), which verifies whether the student has passed the course represented by the courseID input parameter, etc.

Information hiding (also referred to as encapsulation) states that the variables of an object can only be accessed through either getter or setter methods. A getter method is used to retrieve the value of a variable, whereas a setter method assigns a value to it. The idea is to provide a protective shield around the object to make sure that values are always correctly retrieved or modified by means of explicitly defined methods.

Similar to the EER model, inheritance is supported. A superclass can have one or more subclasses which inherit both the variables and methods from the superclass. As an example, Student and Professor can be a subclass of the Person superclass. In OO, method overloading is also supported. This implies that various methods in the same class can have the same name, but a different number or type of input arguments.

3.4.2 Classes

In a UML class diagram, a class is represented as a rectangle with three sections. Figure 3.31 illustrates a UML class SUPPLIER. In the upper part, the name of the class is mentioned (e.g., SUPPLIER), in the middle part the variables (e.g., SUPNR, Supname), and in the bottom part the methods (e.g., getSUPNR). You can compare this with the corresponding ER representation in Figure 3.2.

Example methods are the getter and setter methods for each of the variables. The method getSUPNR is a getter method that retrieves the supplier number of a particular supplier object, whereas the method setSUPNR(newSUPNR) assigns the value newSUPNR to the SUPNR variable of a supplier object.

3.4.3 Variables

Variables with unique values (similar to key attribute types in the ER model) are not directly supported in UML. The reason is because a UML class diagram is assumed to be implemented using an OODBMS in which every object created is assigned a unique and immutable object identifier (OID) that it keeps during its entire lifetime (see Chapter 8). Hence, this OID can be used to uniquely identify objects and no other variables are needed to serve as a key. To explicitly enforce the uniqueness constraint of a variable, you can use OCL, as we discuss in Section 3.4.9.2.

UML provides a set of primitive types such as string, integer, and Boolean, which can be used to define variables in the class diagram. It is also possible to define your own data types or domains

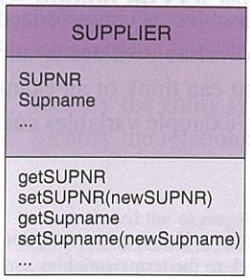


Figure 3.31 UML class.

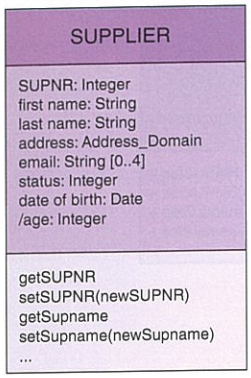


Figure 3.32 UML class with refined variable definitions.

and use them. This is illustrated in Figure 3.32. The variables SUPNR and status are defined as integers. The variable address is defined using the domain Address_Domain.

Composite variables (similar to composite attribute types in the ER model) can be tackled in two ways. A first option is to decompose them into their parts. In our example, we decomposed Supname into first name and last name. Another alternative is by creating a new domain as we did for the address variable.

Multi-valued variables can also be modeled in two ways. A first option is to indicate the multiplicity of the variable. This specifies how many values of the variable will be created when an object is instantiated. In our example, we specified that a supplier can have 0 to 4 email addresses. An infinite number of email addresses can be defined as "email: String[*]". Another option is by using an aggregation, as we discuss in what follows.

Finally, derived variables (e.g., age) need to be preceded by a forward slash.

3.4.4 Access Modifiers

In UML, **access modifiers** can be used to specify who can access a variable or method. Example choices are: private (denoted by the symbol "–"), in which case the variable or method can only be accessed by the class itself; public (denoted by the symbol "+"), in which case the variable or method can be accessed by any other class; and protected (denoted by the symbol "#"), in which case the variable or method can be accessed by both the class and its subclasses. To enforce the concept of information hiding, it is recommended to declare all variables as private and access them using getter and setter methods. This is illustrated in Figure 3.33, where all variables are private and all methods public.

You can compare this with the corresponding ER representation in Figure 3.2. From this comparison, it is already clear that UML models more semantics than its ER counterpart.

3.4.5 Associations

Analogous to relationship types in the ER model, classes can be related using **associations** in UML. Multiple associations can be defined between the same classes. Also, unary (or reflexive) and n-ary (e.g., ternary) associations are possible. An association corresponds to a relationship type in the ER model, whereas a particular occurrence of an association is referred to as a link that corresponds to a relationship in the ER model.

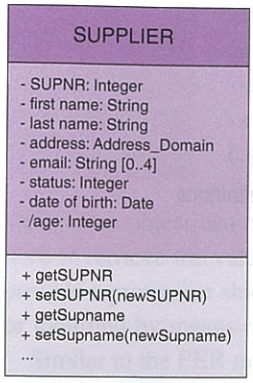


Figure 3.33 Access modifiers in UML.

Table 3.1 UML multiplicities versus ER cardinalities

UML class diagram multiplicity	ER model cardinality
*	0..N
0..1	0..1
1..*	1..N
1	1..1

An association is characterized by its multiplicities, which indicate the minimum and maximum number of participations of the corresponding classes in the association. Hence, this corresponds to the cardinalities we discussed in the ER model. Table 3.1 lists the options available and contrasts them with the corresponding ER model cardinalities. An asterisk (*) is introduced to denote a maximum cardinality of N.

In what follows, we elaborate further on associations and discuss association classes, unidirectional versus bidirectional associations, and qualified associations.

3.4.5.1 Association Class

If an association has variables and/or methods on its own, it can be modeled as an **association class**. The objects of this class then represent the links of the association. Consider the association between SUPPLIER and PRODUCT as depicted in Figure 3.34. The association class SUPPLIES has two variables: the purchase price and delivery period for each product supplied by a supplier. It can also have methods such as getter and setter methods for these variables. Association classes are represented using a dashed line connected to the association.

3.4.5.2 Unidirectional versus Bidirectional Association

Associations can be augmented with direction reading arrows, which specify the direction of querying or navigating through it. In a **unidirectional association**, there is only a single way of navigating, as indicated by the arrow. Figure 3.35 gives an example of a unidirectional association between the classes SUPPLIER and PURCHASE_ORDER. It implies that all purchase orders can be retrieved through a supplier object. Hence, according to this model, it is not possible to navigate from a purchase order object to a supplier object. Also note the multiplicities of the association.

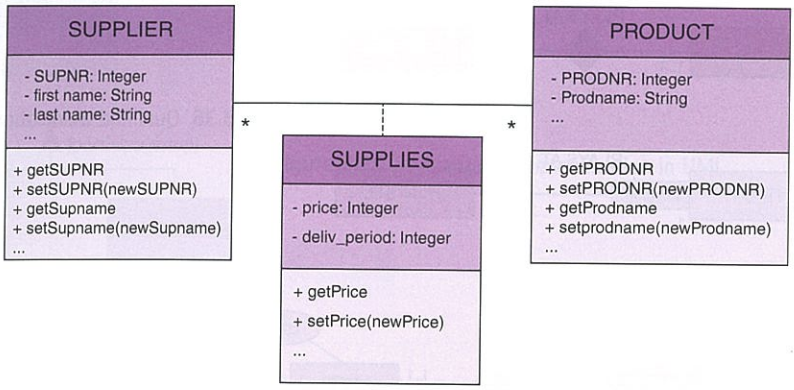


Figure 3.34 Association class.

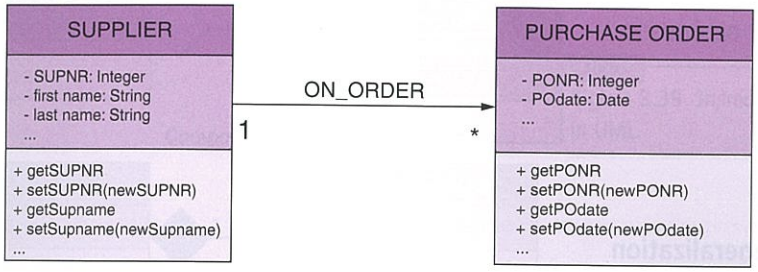


Figure 3.35 Unidirectional association.

In a **bidirectional association**, both directions are possible, and hence there is no arrow. Figure 3.34 is an example of a bidirectional association between the classes SUPPLIER and PRODUCT. According to this UML class diagram, we can navigate from SUPPLIER to PRODUCT as well as from PRODUCT to SUPPLIER.

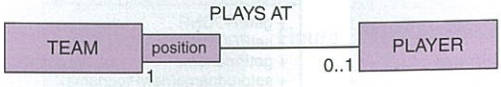
3.4.5.3 Qualified Association

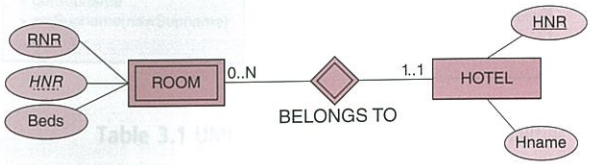
A **qualified association** is a special type of association that uses a qualifier to further refine the association. The qualifier specifies one or more variables that are used as an index key for navigating from the qualified class to the target class. It reduces the multiplicity of the association because of this extra key. Figure 3.36 gives an example.

We have two classes, TEAM and PLAYER. They are connected using a 1:N relationship type in the ER model (upper part of the figure) since a team can have zero to N players and a player is always related to exactly one team. This can be represented in UML using a qualified association by including the position variable as the index key or qualifier (lower part of the figure). A team at a given position has zero or one players, whereas a player always belongs to exactly one team.

Qualified associations can be used to represent weak entity types. Figure 3.37 shows our earlier example of ROOM as a weak entity type, being existence-dependent on HOTEL. In the UML class diagram, we can define room number as a qualifier or index key. In other words, a hotel combined with a given room number corresponds to zero or one room, whereas a room always belongs to one hotel.

Figure 3.36 Qualified association.





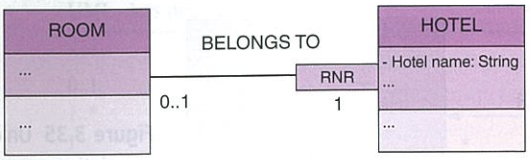


Figure 3.37 Qualified associations for representing weak entity types.

3.4.6 Specialization/Generalization

Similar to the EER model, UML also supports specialization or generalization relationships. Figure 3.38 shows the UML representation of our earlier EER specialization of Figure 3.24 with ARTIST, SINGER, and ACTOR.

The hollow triangle represents a specialization in UML. The specialization characteristics such as total/partial or disjoint/overlap can be added next to the triangle. UML also supports multiple inheritance where a subclass can inherit variables, methods, and associations from multiple superclasses.

3.4.7 Aggregation

Similar to EER, aggregation represents a composite to part relationship whereby a composite class contains a part class. Two types of aggregation are possible in UML: shared aggregation (also referred to as aggregation) and composite aggregation (also referred to as composition). In shared aggregation, the part object can simultaneously belong to multiple composite objects. In other words, the maximum multiplicity at the composite side is undetermined. The part object can also occur without belonging to a composite object. A shared aggregation thus represents a rather loose coupling between both classes. In composite aggregation or composition, the part object can only belong to one composite. The maximum multiplicity at the composite side is 1. According to the original UML standard, the minimum multiplicity can be either 1 or 0. A minimum cardinality of 0 can occur in case the part can belong to another composite. Consider two composite aggregations – one between engine and boat and one between engine and car. Since an engine can only belong to either a car or a boat, the minimum cardinality from engine (the part) to boat and car will be 0, respectively. A composite aggregation represents a tight coupling between both classes, and the part

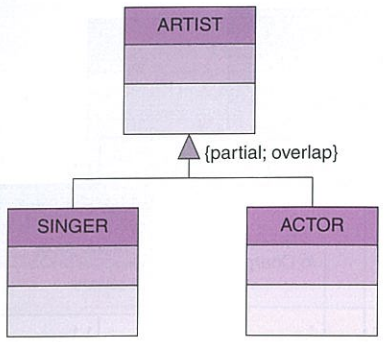


Figure 3.38 Specialization/generalization in UML.

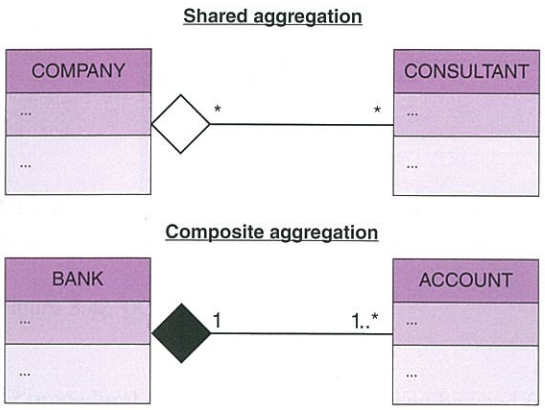


Figure 3.39 Shared versus composite aggregation in UML.

object will be automatically removed when the composite object is removed. Note that a part object can also be deleted from a composite before the composite is deleted.

Figure 3.39 illustrates both concepts. A shared aggregation is indicated by a hollow diamond and a composite aggregation by a filled diamond. We have a shared aggregation between COMPANY and CONSULTANT. A consultant can work for multiple companies. When a company is removed, any consultants that worked for it remain in the database. We have a composite aggregation between BANK and ACCOUNT. An account is tightly coupled to one bank only. When the bank is removed, all connected account objects disappear as well.

3.4.8 UML Example

Figure 3.40 shows our earlier EER HR example of Figure 3.30 in UML notation. It has six classes including two association classes (Manages and Works_On). Note the different variables and methods for each of the classes. The access modifiers for each of the variables have been set to private so as to enforce information hiding. Getter and setter methods have been added for each of the variables. We also included a shared aggregation between DEPARTMENT and LOCATION and between PROJECT and LOCATION. Hence, this implies that location information is not lost upon removal of a department or project. We have two unidirectional associations: between EMPLOYEE and PROJECT, and between DEPARTMENT and PROJECT. The unary association

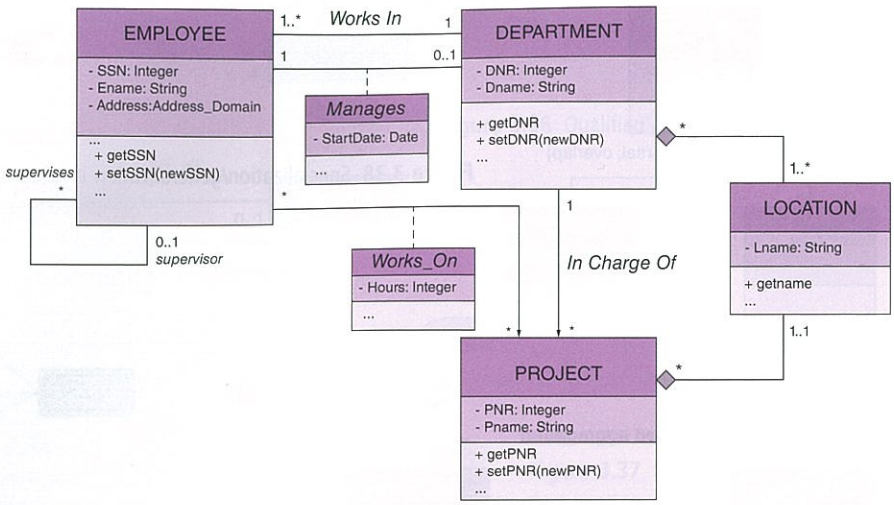


Figure 3.40 HR example in UML.

for the EMPLOYEE class models the supervision relationship. When you contrast this UML class diagram with the EER model of Figure 3.30, it is clear that the former has a lot more semantics embedded.

3.4.9 Advanced UML Modeling Concepts

UML offers various advanced modeling concepts to further add semantics to our data model. In the following subsections, we discuss the changeability property, the object constraint language (OCL), and the dependency relationship.

3.4.9.1 Changeability Property

The **changeability property** specifies the type of operations that are allowed on either variable values or links. Three common choices are: default, which allows any type of edit; addOnly, which only allows additional values or links to be added (no deletions); and frozen, which allows no further changes once the value or link is established. You can see this illustrated in Figure 3.41.

The supplier and purchase order number are both declared as frozen, which means that once a value has been assigned to either of them it can no longer change. The languages variable of the SUPPLIER class defines a set of languages a supplier can understand. It is defined as addOnly since languages can only be added and not removed from it. Also note the addOnly characteristic that was added to the ON_ORDER association. It specifies that for a given supplier, purchase orders can only be added and not removed.

3.4.9.2 Object Constraint Language (OCL)

The **object constraint language (OCL)**, which is also part of the UML standard, can be used to specify various types of constraints. The OCL constraints are defined in a declarative way. They specify what must be true, but not how this should be accomplished. In other words, no control flow

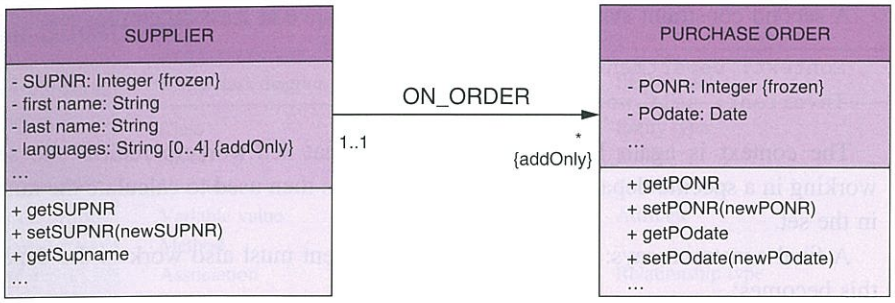


Figure 3.41 Changeability property in UML.

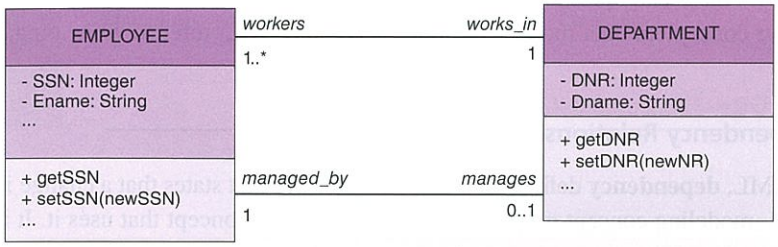


Figure 3.42 OCL constraints in UML.

or procedural code is provided. They can be used for various purposes, such as to specify invariants for classes, to specify pre- and post-conditions for methods, to navigate between classes, or to define constraints on operations.

A class invariant is a constraint that holds for all objects of a class. An example could be a constraint specifying that the supplier status of each supplier object should be greater than 100:

```
SUPPLIER: SUPSTATUS>100
```

Pre- and post-conditions on methods must be true when a method either begins or ends. For example, before the method *withdrawal* can be executed, the balance must be positive. After it has been executed, the balance must still be positive.

OCL also supports more complex constraints. Figure 3.42 illustrates the two classes EMPLOYEE and DEPARTMENT. These two classes are connected with two associations to define which employee works in which department and which employee manages what department. Note the role names that have been added to both associations. Various constraints can now be added. A first constraint states that a manager of a department should have worked there for at least ten years:

```
Context: Department
invariant: self.managed_by.yearsemployed>10
```

The context of this constraint is the DEPARTMENT class. The constraint applies to every department object, hence the keyword invariant. We use the keyword self to refer to an object of the DEPARTMENT class. We then used the role name managed_by to navigate to the EMPLOYEE class and retrieve the yearsemployed variable.

A second constraint states: a department should have at least 20 employees:

```
Context: Department
invariant: self.workers→size() >20
```

The context is again DEPARTMENT. Note that self.workers returns the set of employees working in a specific department. The size method is then used to calculate the number of members in the set.

A final constraint says: A manager of a department must also work in the department. In OCL, this becomes:

```
Context: Department
Invariant: self.managed_by.works_in=self
```

From these examples, it is clear that OCL is a very powerful language that adds a lot of semantics to our conceptual data model. For more details on OCL, refer to www.omg.org/spec/OCL.

3.4.9.3 Dependency Relationship

In UML, **dependency** defines a "using" relationship that states that a change in the specification of a UML modeling concept may affect another modeling concept that uses it. It is denoted by a dashed line in the UML diagram. An example could be when an object of one class uses an object of another class in its methods, but the referred object is not stored in any variable. This is illustrated in Figure 3.43.

We have two classes, EMPLOYEE and COURSE. Let's say an employee can take courses as part of a company education program. The EMPLOYEE class includes a method, tookCourse, that determines whether an employee took a particular course represented by the input variable CNR. Hence, an employee object makes use of a course object in one of its methods. This explains the dependency between both classes.

3.4.10 UML versus EER

Table 3.2 lists the similarities between both the UML class diagram and the EER model. From the table, it can be seen that the UML class diagram provides a richer set of semantics for modeling than the EER model. The UML class diagram can define methods that are not supported in the EER model. Complex integrity constraints can be modeled using OCL, which is also not available in the EER model.

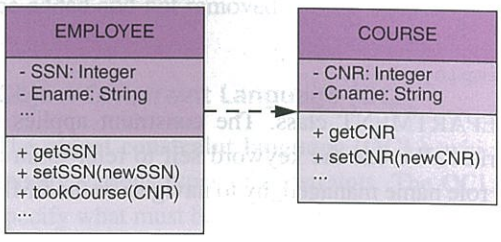


Figure 3.43 Dependency relationship in UML.

Retention Questions

- What are the key concepts of object orientation (OO)?
- Discuss the components of a UML class diagram.
- How can associations be modeled in UML?
- What types of aggregation are supported in UML?
- What advanced modeling concepts are offered by UML?
- Contrast the UML class diagram with the EER model.

Table 3.2 UML versus EER concepts

UML class diagram		EER model	
Class		Entity type	
Object		Entity	
Variable		Attribute type	
Variable value		Attribute	
Method		–	
Association		Relationship type	
Link		Relationship	
Qualified association		Weak entity type	
Specialization/generalization		Specialization/generalization	
Aggregation		Aggregation (composite/shared)	
OCL		–	
Multiplicity	*	Cardinality	0..N
	0..1		0..1
	1..*		1..N
	1		1..1

Drill Down

Some popular examples of conceptual modeling tools are: Astah (Change Vision), Database Workbench (Upscene Productions), Enterprise Architect (Sparx Systems), ER/Studio (Idera), and Erwin Data Modeler (Erwin). These tools typically provide facilities to build a conceptual model (e.g., EER or UML class diagram) and then automatically map it to a logical or internal data model for various target DBMS platforms. Most of them also include reverse engineering facilities whereby an existing internal data model can be turned back into a conceptual data model.

Summary

In this chapter we discussed conceptual data modeling using the ER model, EER model, and UML class diagram. We started the chapter by reviewing the phases of database design: requirement collection and analysis, conceptual design, logical design, and physical design. The aim of a conceptual model is to formalize the data requirements of a business process in an accurate and user-friendly way. The ER model is a popular technique for conceptual data modeling. It has the following building blocks: entity types, attribute types, and relationship types. The EER model offers three additional modeling constructs: specialization/generalization, categorization, and aggregation. The UML class diagram is an object-oriented conceptual data model and consists of classes, variables, methods, and associations. It also supports specialization/generalization and aggregation, and offers various advanced modeling concepts such as the changeability property, object constraint language, and dependency relationships. From a pure semantic perspective, UML is richer than both ER and EER. In subsequent chapters, we elaborate on how to proceed to both logical and physical design.

Scenario Conclusion

Figure 3.44 shows the EER model for our Sober scenario case. It has eight entity types. The CAR entity type has been specialized into SOBER CAR and OTHER CAR. Sober cars are owned by Sober, whereas other cars are owned by customers. The RIDE entity type has been specialized into RIDE HAILING and RIDE SHARING. The shared attribute types between both subclasses are put in the superclass: RIDE-NR (which is the key attribute type), PICKUP-DATE-TIME, DROPOFF-DATE-TIME, DURATION, PICKUP-LOC, DROPOFF-LOC, DISTANCE, and FEE. Note that DURATION is a derived attribute type since it can be derived from PICKUP-DATE-TIME and DROPOFF-DATE-TIME. DISTANCE is not a derived attribute type since there could be multiple routes between a pick-up location and a drop-off location. Three attribute types are added to the RIDE HAILING entity type: PASSENGERS (the number of passengers), WAIT-TIME (the effective wait time), and REQUEST-TIME (Sober App request or hand wave). The LEAD_CUSTOMER relationship type is a 1:N relationship type between CUSTOMER and RIDE HAILING, whereas the BOOK relationship type is an N:M relationship type between CUSTOMER and RIDE SHARING. A car (e.g., Sober or other car) can be involved in zero to N accidents, whereas an accident can have one to M cars (e.g., Sober or other car) involved. The DAMAGE AMOUNT attribute type is connected to the relationship type because it is dependent upon the car and the accident.

As discussed in this chapter, our EER model has certain shortcomings. Since the EER model is a snapshot in time, it cannot model temporal constraints. Examples of temporal constraints that cannot be enforced by our EER model are: the pick-up-date-time should always precede the drop-off-date-time; a customer cannot book a ride-hailing and ride-sharing service that overlap in time.

The EER model cannot guarantee the consistency across multiple relationship types. An example of a business rule that cannot be enforced in the EER model is: a customer cannot book a ride-hailing or ride-sharing service with his/her own car.

The EER model does not support domains – for example, we cannot specify that the attribute type PASSENGERS is an integer with a minimum value of 0 and a maximum value of 6.

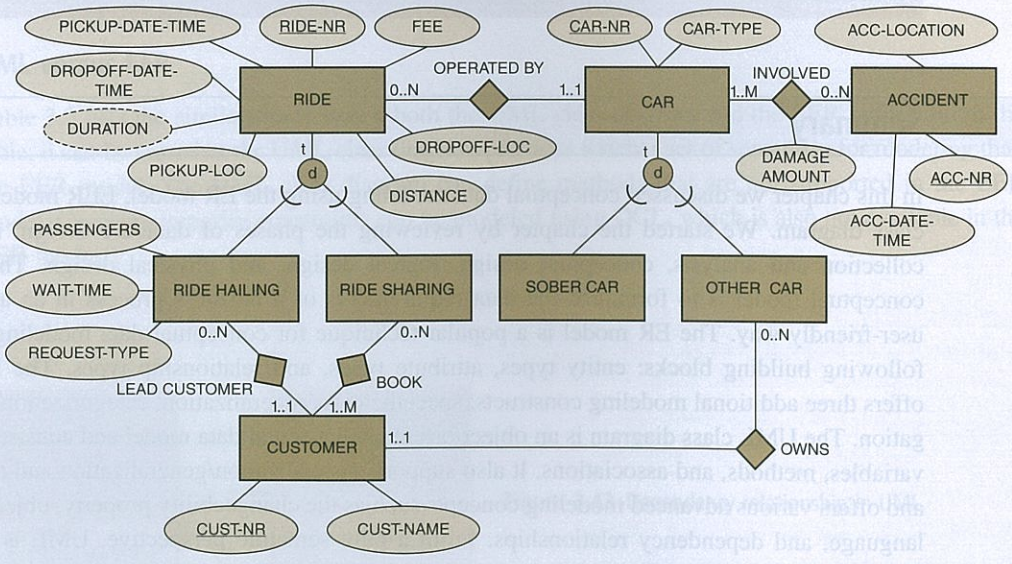


Figure 3.44 EER model for Sober.

Furthermore, our EER model does not specify that the maximum number of passengers for a ride-share service is ten or that the WAIT-TIME is only relevant for Sober App requests and should be zero in case of hand-wave requests.

Figure 3.45 shows the UML class diagram for Sober. It has nine classes. The class INVOLVED is an association class between CAR and ACCIDENT. To enforce information hiding, the access modifiers of all variables have been set to private. Getter and setter methods are used to access them. We also added the following additional methods:

- In the RIDE class: CalcDuration which calculates the derived variable duration.
- In the RIDE-SHARING class: NumberOfCustomers, which returns the number of customers for a ride-share service.
- In the CUSTOMER class: Top5CustomersHail and Top5CustomersShare, which returns the top five customers for ride-hailing and ride-sharing services, respectively.
- In the CAR class: NumberOfRides, which returns the number of rides a car has serviced.
- In the SOBER CAR class: NumberOfSoberCars, which returns the number of Sober cars in the database.
- In the INVOLVED association class: GenerateReport, which returns a report of which cars have been involved in what accident.
- In the ACCIDENT class: Top3AccidentHotSpots and Top3AccidentPeakTimes, which return the top three most common accident locations and timings, respectively.

All associations have been defined as bidirectional, which implies that they can be navigated in both directions. We set the changeability property of the number variables (e.g., RIDE-NR, CAR-NR, etc.) to frozen, which means that once a value has been assigned to any of them, it can no longer be changed.

We can now enrich our UML model by adding OCL constraints. We define a class invariant for the RIDE HAILING class, which specifies that the number of passengers for a ride-hail service should be less than six:

```
RIDE-HAILING: PASSENGERS ≤ 6
```

Remember that the maximum number of passengers for a ride-share service is ten. This can be defined using the following OCL constraint:[5]

```
Context: RIDE SHARING
invariant: self.BOOK→size() ≤ 10
```

The constraint applies to every ride-sharing object, hence the keyword invariant. Other OCL constraints can be added for further semantic refinement.

The UML specification is semantically richer than its EER counterpart. As an example, both passenger constraints cannot be enforced in the EER model. The UML class diagram also specifies the domains (e.g., integer, string, etc.) for each of the variables and includes methods, both of which were not possible in the EER model.

Sober is now ready to proceed to the next stage of database design in which the conceptual data model will be mapped to a logical model.

[5] We could have defined two different role names for the association BOOK to represent its two directions. However, for the sake of simplicity, we use BOOK to refer to the association from the direction of RIDE SHARING to CUSTOMER.

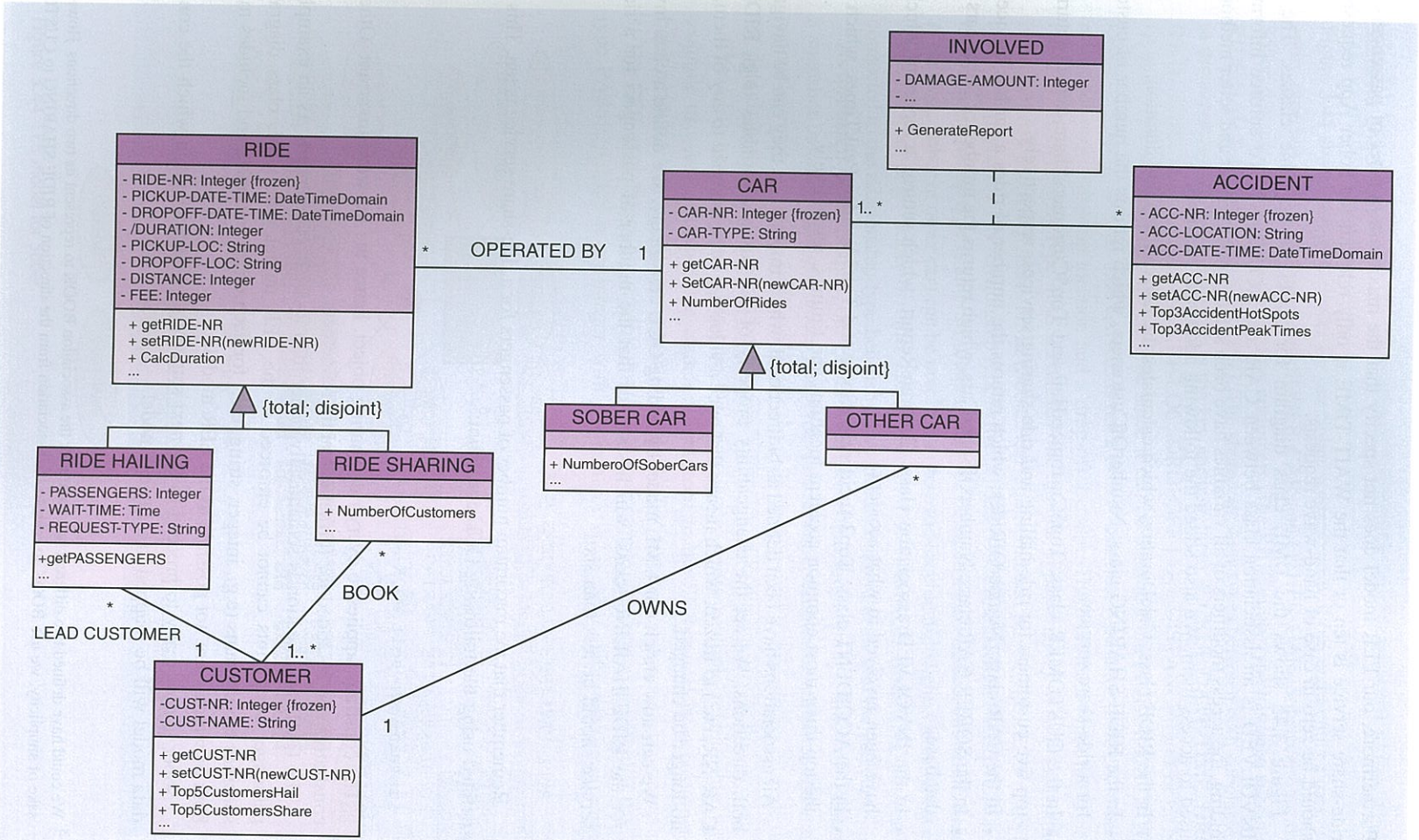


Figure 3.45 UML class diagram for Sober.