# Entity-Relationship (ER) Modeling

The first step to develop any application is to understand what is the problem the application is supposed to solve and what functionality the application should provide. We call this step analysis, or requirements gathering. For database applications, a big part of the analysis is conceptual data modeling; where you are trying to analyze what data needs to be stored in the database.

In this chapter, we cover conceptual data modeling; more precisely, we cover Entity-Relationship (ER) modeling, which uses a particular kind of model, an Entity-Relationship model.

Notice in this book we do *not* cover how to actually get the requirements from the domain experts (users), but just how to express them as an Entity-Relationship model. We do not cover how to get or express the functionality requirements either, just how to model the data requirements as an ER model.

Within the database community, we tend to express requirements in terms of **business**[1] **rules**. A business rule is any statement that constrains any aspect of the business; in a way, business rules are the requirements for the business process, rather than just for the computer system. Business rules normally either assert business structure or somehow control business behavior. We expect most of the business rules will ultimately be automated through the DBMS and/or the computer system being built.

Since business rules are *business* requirements, not computer ones, they will be expressed in terms familiar to the end users, not to software developers. One of the most challenging (and most interesting) parts of software development is that you are required to understand the business domain and learn to talk in *their* language.

Other characteristics good business rules should have (besides being expressed in terms familiar to end users) are shared with good requirements in general. Good business rules should be **declarative** (that is, state what is done, not how), **atomic** (express only one thing), **precise** (have just one meaning), **consistent** (both with itself and with all other business rules for the organization) and **distinct** or **non-redundant** (different from all the other business rules).

In order to create a database for an application, the most important issue is the definition of the data; *what* data needs to be stored in the database. In order to specify what goes in the database, we create a *conceptual* model of the data; this is a high-level, technology independent model of the data.

While creating our data model, we will be choosing names for many data elements. Since the names we choose for all the data elements will be used throughout the application, it is extremely important to choose good names for all your data elements. Again, the names for data elements should be business-oriented rather than computer oriented; the other important issue is to be consistent in your naming conventions; mostl other issues are similar to variable naming conventions in programming languages, except that the consequences of a bad name are worse, since they will be used throughout the application and even outside of it (for ad-hoc reports or other applications that access the same database).



1 Here the term business refers to the organization that will use the software, whether it is designed to make money or not.

In the following sections we will be discussing the Entity-Relationship (ER) model, and the ER diagrams that we use to represent the model. Besides this diagrams, we normally have other information in textual form; at least data definitions and comments expressing constraints that are hard to express with the diagram.

When we express the requirements in English[2] we use **terms**, which are words or phrases with a specific meaning, and **facts** which express relationships among those terms. Notice that eventually, our database will also be a repository of terms and facts.

A **term**, is a word or phrase with a specific meaning.

A **fact** is a statement that expresses a relationship among two or more terms

Since we will be creating this model from user requirements, and so discussing the model with the users, we want a graphical notation, and one that's relatively easy to understand for 'normal' people. One of the most common models used in databases is the *entity-relationship* model (or ER model).

## *1 Basic Components of an ER model*

The term ER model can be used in two ways; we can use it in the general sense to refer to any model that uses entities and relationships, or to refer to a specific ER model for a specific situation. ER models are usually represented as ER diagrams, following some standard conventions. Although there is much variation in how to represent the ER model as an ER diagram, here we settle in one specific notation.

The basic components of an ER model are **entities** and their **relationships**. Both entities and relationships can have a**ttributes**.

## *2 Entities and Attributes*

Entities are the things in the real world that we want to model. We distinguish between **entity types** and **entity instances**; an **entity type** is a set of entities that share common characteristics, whereas an **entity instance** is each one of those entities. This is again the distinction intension and extension of a database, and is similar to the distinction between class and object in object-oriented programming.

An **entity type** is a set of entities with common characteristics.

An **entity instance** is one of the members of an entity type.

Notice that we are always representing the particular definition one organization gives to an entity for a particular problem, not necessarily any general concept.

We represent entity types in an ER diagram, by using a rectangle, with the name of the entity type inside it. As a convention, we use singular nouns for naming entity types (so we would use Person rather than People).

Entities have properties, which we call **attributes**. We represent attributes by an oval, with the name of the attribute inside it. We use a line to attach attributes to the entity type they belong to. Figure 1 shows a simple representation of a Person entity type.

2 or any other human language

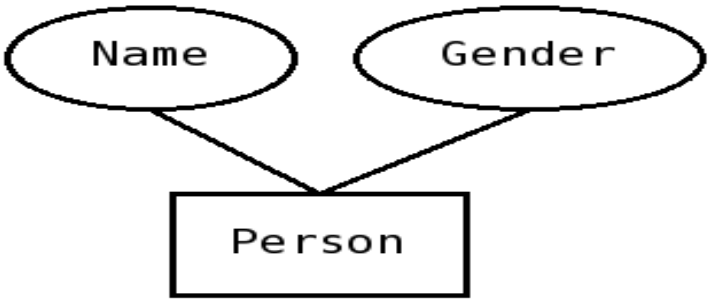


*Figure 1: Simple ER diagram for Person. Notice the rectangle represents the entity type (person) and the ovals the attributes.*

We classify attributes according to four independent binary categories:

- Optional vs Mandatory
- Simple vs Composite
- Single-valued vs Multi-valued
- Stored vs Derived

An attribute is **mandatory** if *all* entity instances of a given entity type *must* have a value for the attribute, and it is optional otherwise. For example, for a particular application we may require that all students in the database *must* have a first and last name (which would make those mandatory), whereas the middle name would *usually* be considered optional. We do not use any specific markings in our drawing to denote whether an attribute is mandatory or optional.

An attribute is **simple** if we cannot divide it into meaningful parts within our application; an attribute is **composite** if we can meaningfully divide it into parts. For a composite attribute, we will sometimes need to see it as a whole, while sometimes we will need to access its component parts.

Notice that at the most basic levels, any attribute other than a bit can be decomposed; for example, a string can be decomposed into its individual characters or letters, and an integer could be divided into its individual bits; the important distinction is whether it is meaningful to do so within the application.

We will distinguish the *parts* of a composite attribute by linking them to the composite attribute rather than to the entity type (or relationship), as shown in Figure 2.

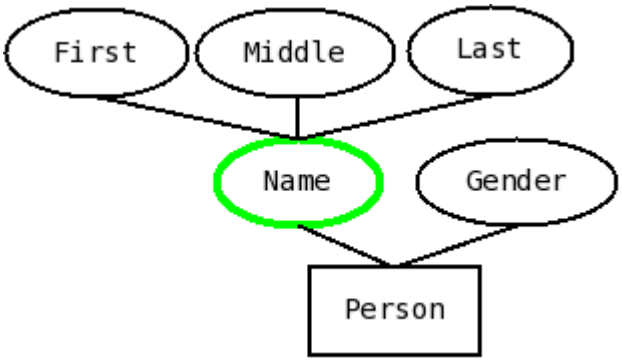


*Figure 2: Name is a composite attribute of Person, with three parts; first, middle and last*

Many times, a single entity may have a set of values for an attribute rather than a single value. We call those attributes **multi-valued**, with attributes that have at most one value being called **single-valued**. Email addresses and phone numbers are common examples of multi-valued attributes, since people many times have more than one email or phone number. We use a double line for the oval to denote multi-valued attributes, as in Figure 3.

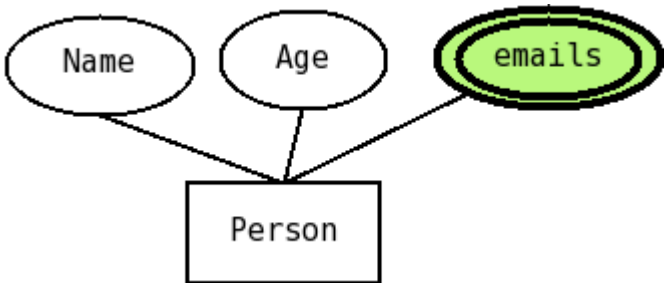


*Figure 3:* Email *is a multi-valued attribute, denoted by double lines.*

Another important distinction is between stored and derived attributes. Most attributes will be stored in the database, but a few can be obtained from other fields in the database; these attributes are called **derived**. We denote derived attributes by using a dashed line in the oval, rather than the normal line. The classical example would be age (which we have actually represented as a stored attribute in figures 1 through 3); we normally do not store a person's age, since the age changes; we would rather store the date of birth and calculate the age from there. If we consider age to be important enough to include in the diagram, we would add it with dashed lines. Notice we would NOT mark date-of-birth in any special way, but would add a note of some sort in our textual documentation with the formula for calculating the age. Figure 4 illustrates representing Age as a derived attribute.

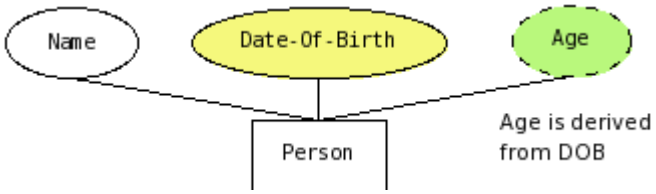


*Figure 4: Age is a derived attribute (dashed line), notice date-of-birth is not especially marked; there would be a note on the definition of age detailing how to calculate it*

In the real world, different entities can always be distinguished from each other. When modeling entities for a database, we do not (normally) store all its attributes (which for a physical entity are arguably infinite); if we are not careful, we may not store enough attributes to distinguish among different entities, leading to confusion. To ensure that doesn't happen, we always mark an attribute (or a set of attributes) as the **identifier** for an entity type, meaning that no two entities will have the same value for that set of attributes. We denote the identifier by underlining its name.

Notice that in many cases we may have more than one attribute that uniquely identifies entities; in that case we will just mark one of the possibilities as the identifier (by underlining it) and we will underline several attributes only when they are *all, together* needed to identify the entities of that type.

Figure 5 shows the full diagram for a (possible) person entity type, including the use of SSN as an identifier.

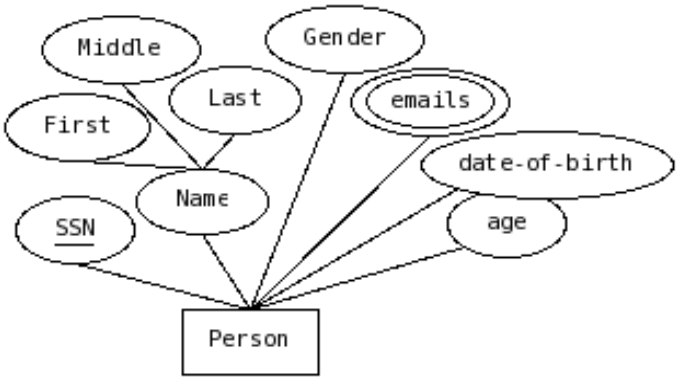


*Figure 5: Complete ER diagram for person attributes*

## *3 Relationship Basics*

Normally, entities by themselves are not terribly useful; we need to somehow relate the entities to one another. In the ER model, **relationships** are used to represent associations between entities. Notice that they are the *only* way to represent associations among entities in the relational model[3]. In ER diagrams, we use *diamonds* to represent relationships among entities.

When modeling relationships, we again can make the distinction between types and entities; a **relationship type** is a meaningful association that can occur between different entities of specified entity types, whereas a **relationship instance** is each one of the actual associations between entity instances. As with entities, we normally model relationship types, since the instances will eventually be stored in the database.

Going back to English, entities are *nouns*, which make up most of the terms; relationships allow you to define *phrases*; the relationship name is the *verb* in the phrase, which is why relationships are verbs, or verb phrases. A relationship type is a *verb* we can use, it defines one kind of phrase we're going to have in our database, while a relationship instance would be each one of the specific phrases of that kind.

For example, if we have an entity type person, with instances Jane and Joe, and an entity type Drink with instances Coffe and Tea, then we can define a relationship type Likes, which links a Person with a Drink, and the instances could be that Joe likes Tea, or Jane likes Coffee.

Figure 6 shows an incomplete ER diagram for a relationship. The diagram does not illustrate cardinality constraints, which we will discuss shortly. Notice that we use verb phrases to name relationships (in this example, lives in); also, although relationships are bidirectional, and could be navigated in either direction, the name we give to it assumes

3 In the relational model, which we will discuss in other chapters, we use foreign keys to represent associations among rows in a table; in the ER model, attributes cannot be used to associate entities, only relationships are used to do that.

one specific direction (in this example, it would be understood that a person lives in a country, rather than a country living in a person).

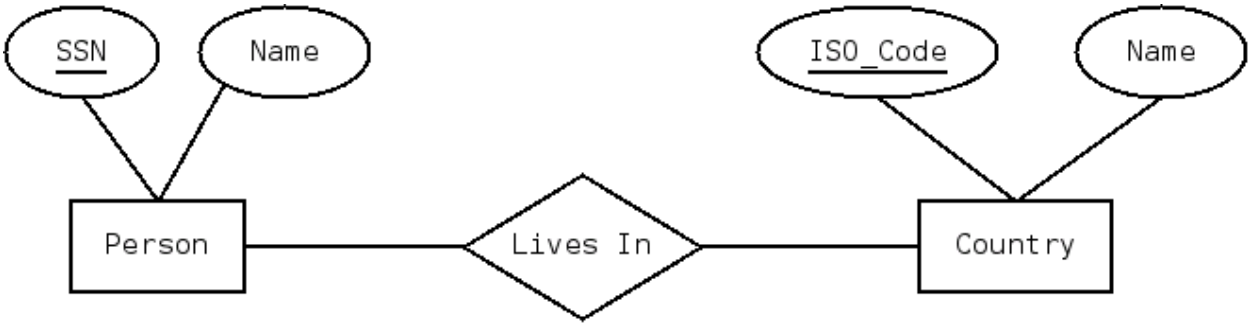


*Figure 6: A person lives in a country (missing cardinality constraints)*

When modeling relationships, one of the most important issues is the **degree** of a relationship, that is, the number of entities that participate on it. The simplest relationships are **binary**, since they relate two entities (later we will discuss relationships of different degrees).

Another important issue is the **cardinality** of a relationship. For each side of a relationship we identify constraints on the number of instances of the opposite side an entity could be related to. For example, we could specify that a person, at any given time, lives in exactly one country, and a country may have zero or more people living in a country at any given time. Figure 7 illustrates how we would represent such a situation:

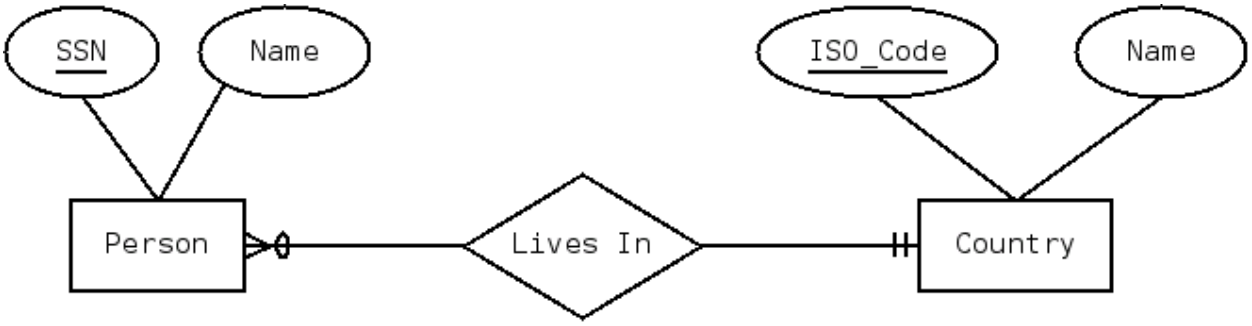


*Figure 7: Cardinality constrains; a person lives in exactly one country (minimum of 1, maximum of 1, represented with ||) and a country can have zero or more people living on it ( represented with 0<)*

On the side opposite to person (next to country) we specify the minimum and maximum countries a person lives in; for the minimum we would use either a 0 or a 1(represented by a circle or a vertical line) and for the maximum either 1 or more than one (represented by < or >); in this case we use || (minimum 1 maximum 1) for person; on the side opposite to country (next to person) we specify how many people could live in the same country; we specify that the minimum is zero (0) while the maximum is more than one; the full constraint is represented as >0.

The ER notation can be extended to represent specific limits if necessary, but in most situations there are no specific limits other than 0,1 or more than one.

For binary relationships, we combine the *maximum* cardinality on both sides, and we say a relationship is one-to-one, one-to-many (or many-to-one) or many-to-many. The relationship in Figure 7 would be one-to-many.

Notice that relationships may also have attributes; for example, if we wanted to include in our model the date a person started living in a country, we could add an attribute (say, since) to the lives in relationship, as illustrated in Figure 8.

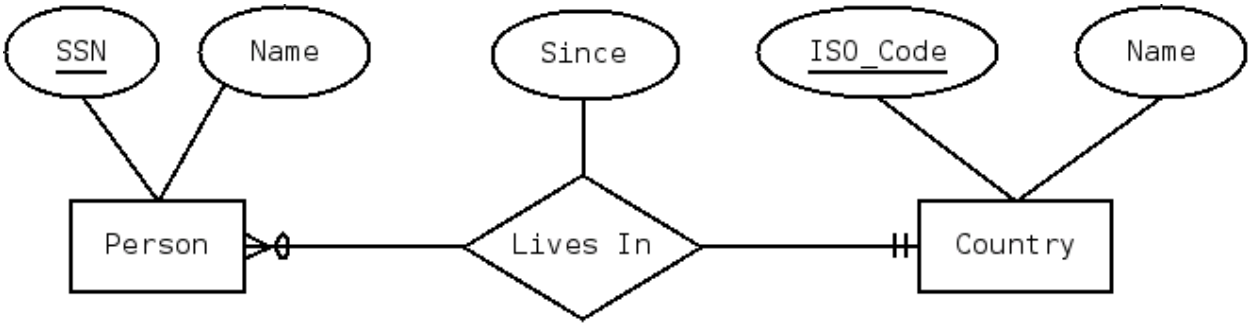


*Figure 8: Relationships can also have attributes*

## *4 Recursive Relationships*

Recursive relationships are those that relate two or more entities of the same type. Normally, we use the entity type to distinguish among the sides of a relationship, but with recursive relationships there are at least two sides that are the same, so we need to introduce the concept of **role** to distinguish among them. A role is just a name given to one of the sides of a relationship.

Notice that the book calls this kind of relationships unary, but this idea does not generalize. If we actually used the book's terminology, a relationship that relates three entity instances of two different entity types would be considered binary (and so, normal an easy :). Although I accept the book's definitions as answers to exams, I prefer to define degree as the number of entity **instances** that participate in a relationship instance.

Recursive relationships are not terribly uncommon, since, besides other uses, they are useful to represent hierarchies.

One of the best examples is the supervises relationship. In many companies, people supervise other people. So, if we want to show that relationship we could start thinking about having two kinds of entities, say Boss and Peon, with the supervising relationship as follows.

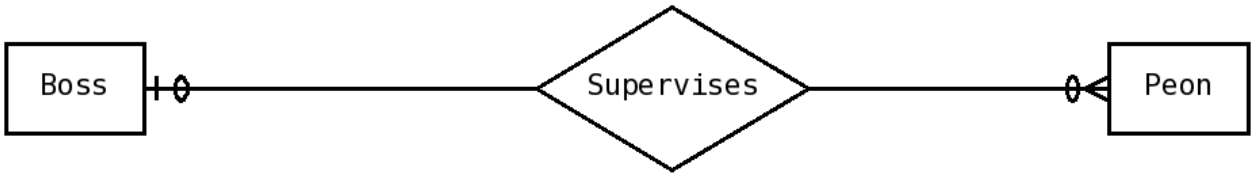


*Figure 9: Conceptual idea of Supervises. WRONG*

And we would say a boss supervises zero or more peons, and a peon is supervised by zero or one bosses.

Of course, we would realize this is not exactly right, since this reflects only a two-level hierarchy, and what we want is one with unlimited levels. In order to do that, we need to introduce recursion (just like in programming), so we need to fold this diagram so the two entities become one.

So, our diagram would look like this:


*Figure 10: Supervises, roles missing, INCOMPLETE*

Now the problem would be how to distinguish among the two sides; that is whether you can have more than one peon or more than one boss. We do that by including the **role** each entity instance plays in the relationship instance. Basically, we assign a name to each side, and use that to distinguish among the sides.

So, the full diagram, specifying that each employee has zero or one bosses (assuming the CEO doesn't have any boss), and that each employee may supervise zero or more other employees would look like this:

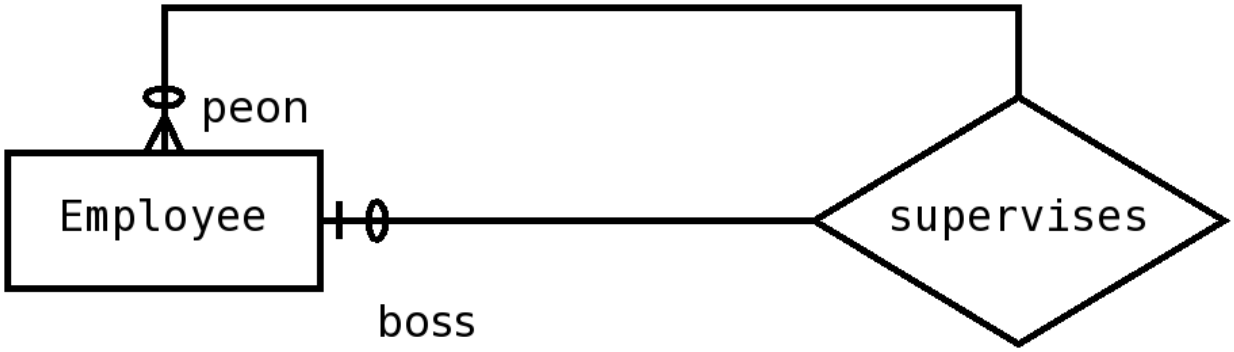


*Figure 11: Supervises, final and right version*

Notice that the cardinality constraints apply to the full entity type, not just to the roles. It may look funny to allow for a boss with no subordinates, but we need to remember this cardinality constraint applies to all Employees, and not all Employees will have subordinates.

## Bill of Materials – Managing parts

Another example is the part-of relationship, also called the bill of materials problem. This involves representing the fact that an item is made from other items; these sub-items may in turn be made from other items, in a hierarchy.

So, we can follow the same kind of reasoning. From a diagram that considers them to be two separate items:

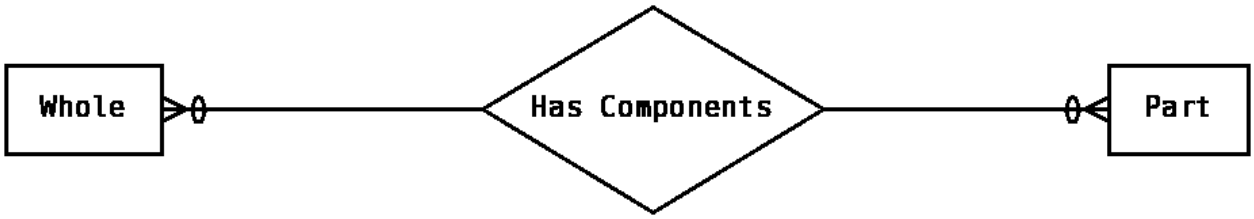


*Figure 12: Bill of materials (wholes and parts), conceptual idea. WRONG*

And then fold them into one recursive relationship as follows:

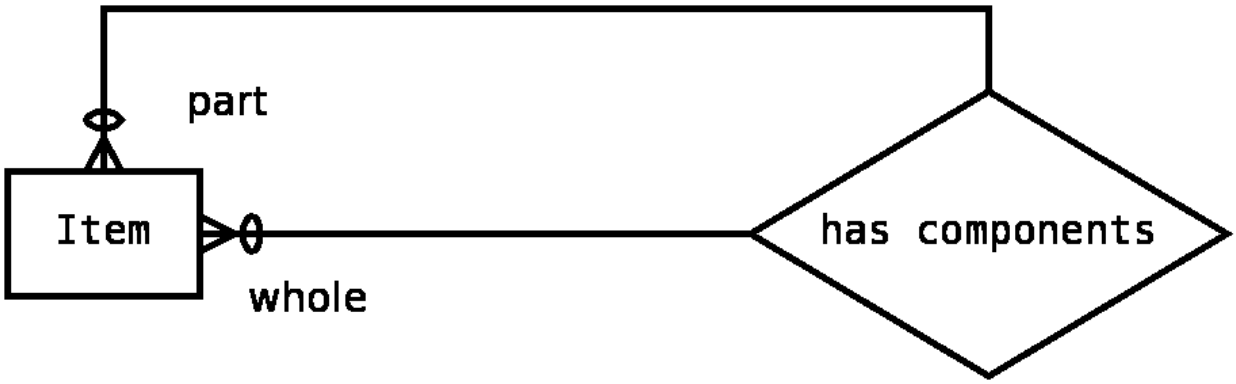


*Figure 13: part-whole recursive relationship.*

Now, we could want to improve this diagram by adding how many of each part each item uses. The diagram would look as follows:

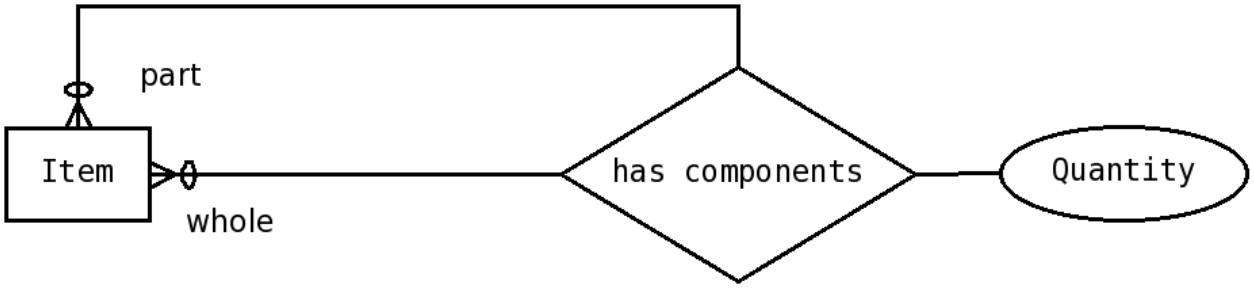


*Figure 14: Bill of materials, attribute on relationship*

## Additional Examples of Recursive Relationships

There are a few other examples that always come to my mind when talking about recursive relationships.

One example that is easy to understand and is NOT used to represent a hierarchy, is that of marriages. Since each marriage is between two people, this would be a recursive relationship; however, it is one-to-one and does not form a hierarchy.

The diagram would look like this, with the roles identified as husband and wife.

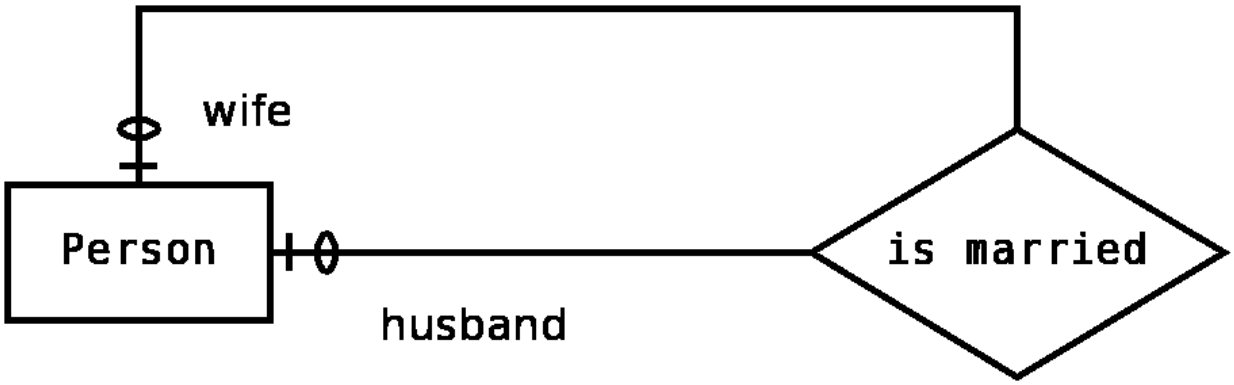


*Figure 15: Marriage is a one-to-one recursive relationship, that does not form a hierarchy.*

Notice that there may be constraints other than cardinalities (such as gender requirements for each role), and those constraints should be added as annotations outside the diagram.

Another example, common in the academic domain, is that of course prerequisites and co-requisites. In many academic programs you are required to take certain courses before others (prerequisites); sometimes, you are allowed to take the courses concurrently (co-requisites). We can express this situation as two different recursive relationships among courses, as in figure 16.

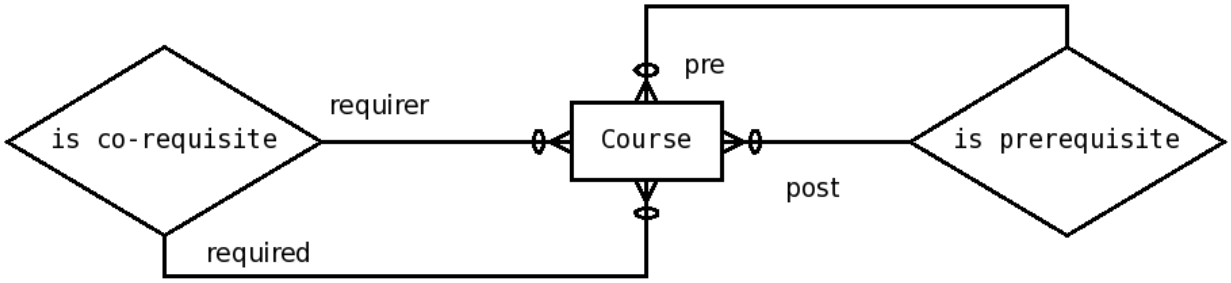


*Figure 16: Course prerequisites and correquisites as two separate relationships*

Another way to model this situation would be to represent this as a single relationship, with the kind of requisite (either pre or co) as an attribute. The diagram would then look like this:

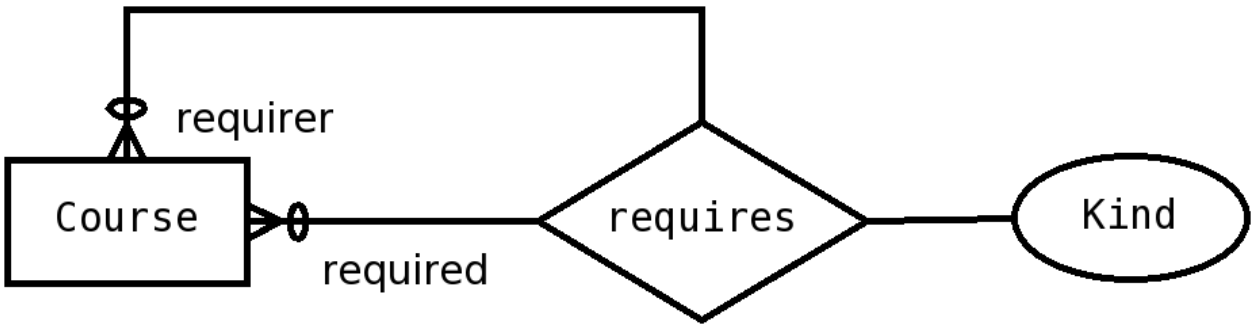


*Figure 17: Course prerequisites and corequisites as one combined relationship*

## *5 Exercises (Starred ones have solutions)*

1. * We have only one entity, called **Person** (everything else is represented as attributes), with the following attributes: Id (the identifier); Name, which is composed of one or more given names and one or more family names; One or more aliases, an address (composed of street, city, state,zip); date-of-birth; and age, which can be calculated from the date of birth.
2. * We want to model a CD as an entity. The only entity is the CD and everything else is modeled as attributes.
   - A CD has a number, which is its identifier.
   - A CD has a title.
   - A CD has zero or more performers; for each performer we keep their first and last name.
   - A CD has zero or more songs. For each song we keep its title and the name of its author (with name divided into first and last)
3. * Now model a CD, but with Person and Song as entities. So the requirements are as follows:
   - A CD has a number, which is its identifier, and a title.
   - A person has an id, and a name, divided in first and last.
   - A song has an id, a title and a length.
   - We keep track of which person is a song's author. A person can author many songs and a song has exactly one author.
   - We keep track of which people perform on a CD. Zero or more people perform on a CD and people can perform on zero or more CDs.
   - We keep track of which songs are included on a CD. One or more songs are included on a CD, and a song is included in one or more CDs.
4. * We want to keep track of **courses**, **sections** and **professors**. A course has a course number (identifier), and a title. A section has a crn (identifier), semester, and year. A section belongs to exactly one course, and a course may have zero or more sections. A Professor has ssn (identifier) and name. A professor teaches one or more sections (a section is taught by exactly one professor). A professor is qualified to teach one or more courses (a course may have one or more qualified professors). We also want to keep track of the date a professor became qualified to teach the course.

5. We want to model information about rental properties and the people who lease them or occupy them.
   - For each property, we keep its id, address (divided into number, street, city, zip and apartment number), the number of square feet, and the maximum number of occupants.
   - For each person we keep its id, name (divided into first, middle, last) and date of birth.
   - We keep track of which person leases a given property. Each property is leased by zero or one person, and each person leases zero or more properties. For properties which are leased, we keep track of the date the lease started.
   - We keep track of which people occupy each property. A person can occupy zero or one properties and a property can be occupied by zero or more people (notice the occupants may or may not be the leaser).

6. We want to represent marketing information about Bars, their Customers ,the Beers they serve, and the Companies that make those beers.
   - For each bar, we keep its id, name, and address.
   - For each Beer, we keep its id, name, a set of nicknames, and its kind.
   - For each Company we keep its id, name, and the number of beers it makes (which can be obtained from other info on the database).
   - For each Customer, we keep its id, name, gender, date of birth and age.
   - We keep track of which company makes each beer. A beer is made by exactly one company, and a company can make zero or more beers.
   - We keep track of which customers go to which bars. A customer goes to one or more bars and a bar may have zero or more customers. We also record the date a person started going to the bar.
   - We keep track of which customers like which beers. A customer likes zero or more beers, and a beer may be liked by zero or more customers.
   - We keep track of which bars serve which beers. A bar serves one or more beers, and a beer is served in zero or more bars.
7. * We have two kinds of entities: **Products** and **Categories**, both with attributes id (the identifier), and name. Each product belongs to zero or more categories and each category can have zero or more products. Categories are organized in a hierarchy; each category belongs to zero or one categories, and may have zero or more sub-categories.
8. * We want to model **course** and **program** information for a university. For each course we keep its id, its title, the number of lecture hours, the number of lab hours, the number of credit hours (which is always equal to the number of lecture hours plus one-half of the number of lab hours). Courses may have other courses as prerequisites. We also want to keep information about programs of study. For each program we want to keep its id and its title. We also want to keep track of which courses are required for a program of study, and which courses are allowed as electives for a program of study.