


SPSU

Conceptual Modeling

Entity Relationship Diagrams

Orlando Karam
okaram@spsu.edu

 SOUTHERN POLYTECHNIC STATE UNIVERSITY

Within the database community, we tend to express requirements in terms of **business¹ rules**. A business rule is a statement that constrains some aspect of the organization; 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.



SPSU

Business Rules

- **Business Rule: Statement that constrains certain aspects of the business**
 - Assert business structure or
 - Control business processes
 - Basically, Requirements from Business
- **Our DB App will (hopefully) automate business rules**

2

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

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.

We usually represent our conceptual model by using an Entity-Relationship or ER diagram. An ER diagram is a graphical representation of an ER Model.

An ER Model represents a real-world situation by describing the Entities involved, the Relationships among those entities, and the attributes of both entities and relationships. In the following slides we define ER models in more detail.

ER modeling

SPSU

- An ER Model is based on Entities, Relationships among entities, and attributes of entities and relationships
- An ER Diagram is a graphical representation of an ER Model

3

Entities represent *things* in the real world, whether physical things, like, say, chairs or conceptual things, like departments, or courses.

In a database, we will store many many things, so we want to reduce the number of concepts we deal with by grouping things with the same properties, so during analysis and design we can focus on just a few *kinds* of things, rather than on the actual million of things that will go in the database.

So, during conceptual modeling we define *entity types*, sets of things with common characteristics. Eventually, the database will contain many *entity instances*, but just from a few different types. In an ER diagram, we use rectangles to represent entity types.

Entities

SPSU

- Things in the real world, physical or not
- Entity type is set of entities that share properties or characteristics
- Entity Instance is each of the instances of an entity type
- Represented by rectangle

Student

4

For each entity, we want to record many properties; we call those properties attributes. Later on we'll see relationships also can have attributes, but right now we'll focus on attributes of entities. In an ER diagram, we use ovals to represent attributes.

Since attributes are so important for databases, we will spend some time studying them. We classify attributes according to four independent categories. Attributes can be optional or mandatory; for mandatory attributes, every entity in the database needs to have a value.

Attributes can be simple, or composite. Composite attributes can be broken down into smaller pieces.

Most attributes can only have one value for each entity instance; however, multi-valued attributes can have more than one value for the same instance.

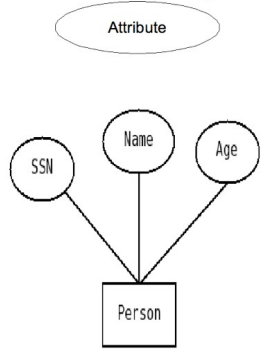
Most attributes will be stored in our database; however, derived attributes can be calculated from other attributes and so would not need to be stored.

Finally, we need to find a set of attributes that will uniquely identify each entity instance of the same entity type; we call this set of attributes the identifier of the entity type, and we underline the attributes that are members of the identifier.

Attributes

SPSU

- Properties of entities or relationships
- Represented by oval
- Can be (at the same time)
 - Optional or mandatory
 - Simple or composite
 - Single- or multivalued
 - Stored or Derived
- We also need to define an *identifier* which is a set of attributes for which no two entities of a given type will have the same value



5

As mentioned before, some attributes are optional, that is, some entity instances may have no value for the attribute, whereas others will be mandatory, each instance in the database will have a value.

Notice this distinction is within the database, not necessarily for the real world; for example, if we're representing an Employee entity, its SSN might be mandatory, but a gender attribute could be optional; even though every employee will have a gender in the real world, we may not necessarily have it in the database for all employees.

Optional vs Mandatory

SPSU

- For optional attributes, some entities may have no value at all.
- For mandatory attributes, a value has to be entered in DB.
- Not represented in ER diagram.
- By default, consider all attributes optional
- Examples of Mandatory: SSN, Name (for many applications)

6

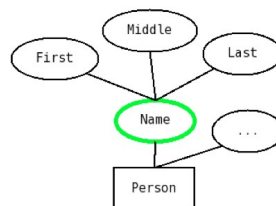
Most attributes in a database are simple; they cannot be broken up into smaller pieces without losing their meaning; however, some attributes are composite; they have meaningful pieces.

For example, in the US most people divide their full name into their first, middle and last names, which we would represent in an ER diagram by having the name attribute linked to its three pieces.

Simple vs Composite

SPSU

- Composite attributes can be broken-up into meaningful parts.
- Each part has a name, and they could be of different types.



7

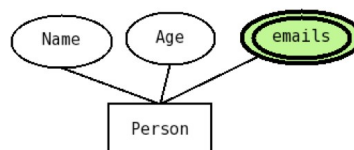
Most attributes are single-valued; each entity instance will have at most one value in the database; for example, you have *one* age, and one name, even though it is broken up in pieces.

However, some attributes in a database may have more than one value; for example, many people will have more than one email address, so we would say that emails is a multivalued attribute, which we represent with double lines.

Single-valued vs multi-valued

SPSU

- For some attributes, we may have more than one value for the same entity.
- Here, all values are of the same type, and they are not identified by name; similar to an array or a set.
- Represented by double lines



8

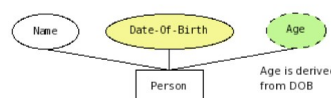
Most attributes of interest will actually be stored in the database; after all, that's why we're building it; however, there are some attributes which can be obtained from other attributes; we call those derived.

For example, although we're many times interested in knowing somebody's age, we probably wouldn't store it in the database, since it changes every day !, we would store the person's date of birth, and represent age as a derived attribute (and somewhere else we'll document how to calculate the age from the date of birth). Derived attributes are one way we get information rather than raw data from a database.

Derived vs Stored

SPSU

- Many times we can calculate an attribute from some other attributes stored in the database
- We call those *derived* attributes (and use dash-lines around them).
- Sometimes you can choose which ones to store or derive



9

We represent derived attributes in a diagram by drawing the oval using a dashed line.

Ok, so now you try it. Don't look at the solutions yet, just get a piece of paper and try the exercise yourself. When you're done, the solution is on the next slide.

Practice: Person

SPSU

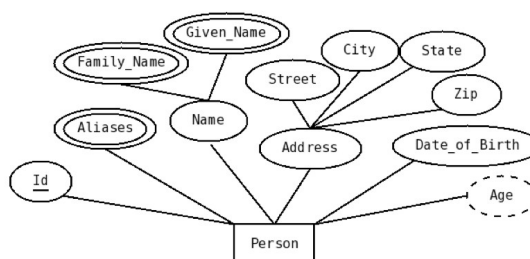
- Produce an ER diagram for the following situation:
- We have only one entity, called **Person** (everything else is represented as attributes), with the following attributes: **Id** (the identifier); **Name**, which is composed of one or more **given names** and one or more **family names**; One or more **aliases**, an **address** (composed of street, city, state, zip); **date-of-birth**; and **age**, which can be calculated from the date of birth.

10

Ok, let's see. Person is the entity, so we put it in a rectangle. It has Id, the identifier, so we underline it. It has a name, which is composed of two attributes, given names and family names, and each one of these is multivalued, so they have double lines. Aliases is multivalued, so it gets double lines too. We have address, with its pieces, then date of birth, and age, which is derived and so drawn with a dashed line.

Solution: Person

SPSU



11

Although attributes allow us to represent many aspects of the real world, they do not allow us to represent associations between entities; to do so, we need a *relationship*, which appears as a diamond in an ER diagram.

Relationships themselves may have attributes, and we'll see some examples of that later; however, they cannot have an identifier, since there can only be one relationship instance for a given collection of entity instances.

Relationships

SPSU

- Association between two or more entities
- Represented by diamond
- Degree: number of entity types that participate (mostly binary)
- Cardinality constraints

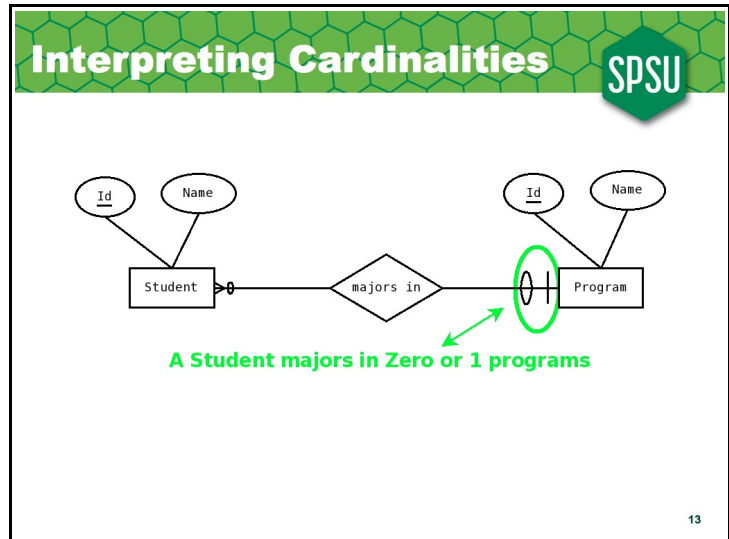


12

We can define the Degree of a relationship as the number of entity types that relationship associates; most relationships will be binary, associating only two entities.

We also need to define, for each side of a relationship, cardinality constraints, that define the number of entities (on the other side) that an entity will be associated with. For each side, we define a minimum cardinality (that is usually either 0 or 1) and a maximum cardinality (which is usually either one or more than one, which we also call many). For the minimum cardinality, we use a circle for 0 and a vertical line for 1. For the maximum, we use a vertical line for one, and a less-than or greater-than symbol for many.

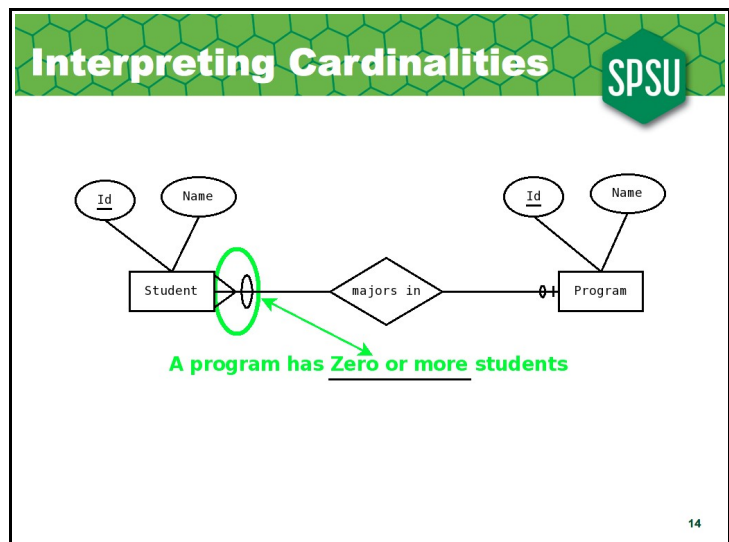
Now let's see how we interpret those cardinality constraints; We would interpret the squiggles next to the Program entity as representing constraints on how many programs can a student be related to; in this case, the diagram says that a student majors in zero or 1 programs.



And the squiggles next to the Student entity tell us with how many students can each program be related; in this case, a program may have zero or more students majoring on it.

Notice also that the relationships are designed to be read one way, and it is expected you can understand it by the name; for example, here we assume you realize a Student majors in a program and not the other way.

Although we name the relationship in one way, when we put into the database we will be able to access the relationship either way; that is, given a student we can find its major, and given a program we can find all the students majoring in the program.



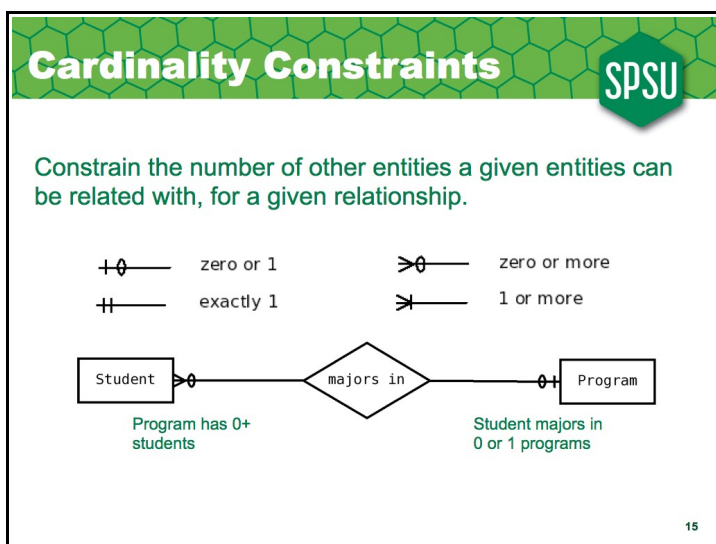
Ok, so let's review. We have four possible values for the cardinality constraints, and we put constraints on each side.

The cardinality constraints express the minimum and maximum number of entities that entities of a given type will be related to through that relationship. The minimum is either zero or 1, and the maximum is either 1 or more than one.

So, the possible combinations are:

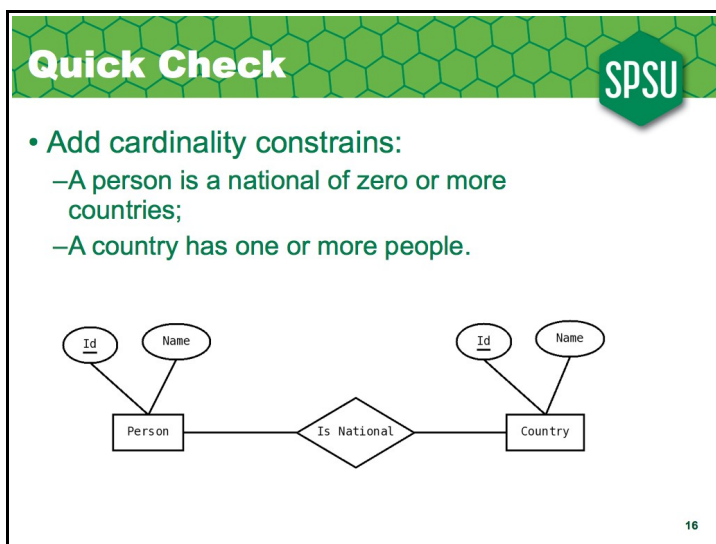
- zero or one
- zero or more
- exactly one
- one or more

Notice that the constraints go on the side *opposite* the entity they constrain, so in the diagram above, the zero-or-1 constraint, which says that a student majors in zero-or-one programs is *opposite* the student entity, so next to the program.

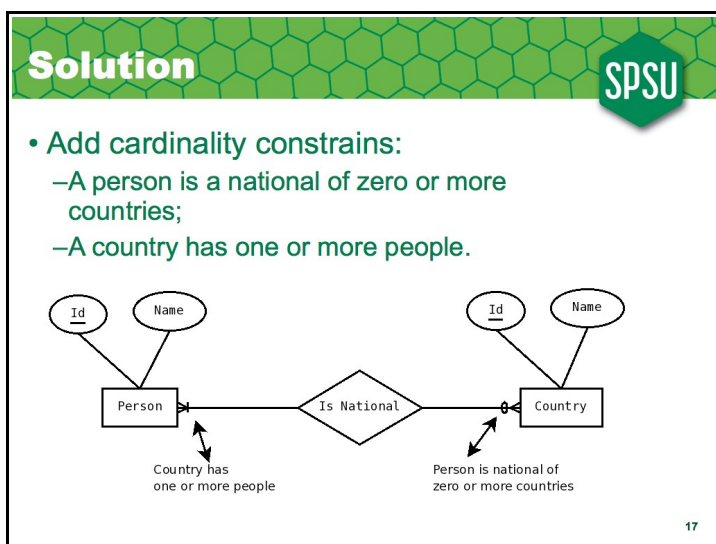


So now you try it. Add the cardinality constraints to the following diagram; a person is national of zero or more countries

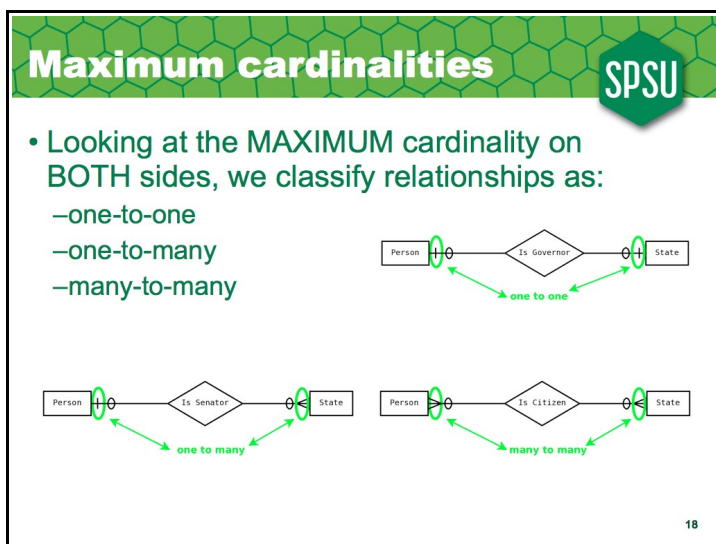
A country has one or more people



And here's the solution. Since a person is a national of zero or more countries, we write the 'zero-or-more' constraint on the side opposite to person; similarly, we write the one -or-more constraint opposite country.

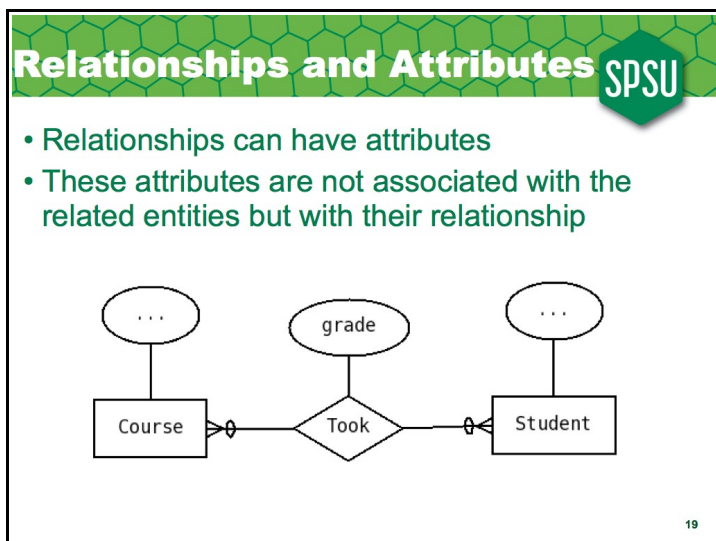


Although this is more useful when comparing the ER with the relational model, which we'll see later, it is convenient to classify binary relationships by their MAXIMUM cardinality on both sides; if the maximum is one on both sides, we call it a one-to-one relationship; if the maximum is one on one side and more-than-one on the other we call it a one-to-many relationship (or a many-to-one, depending on our viewpoint :), and if it's more-than-one on both sides we call it a many-to-many relationship.



Notice that relationships may have attributes; most of the cases, if a relationship has attributes it will a many-to-many relationship.

For example, if we're representing the fact that a student takes a course, we may want to keep track of the grade a student got in that course; in this case, the grade, say A, cannot be an attribute of the student alone, since we wouldn't know on which course did the student got an A; similarly, it cannot be an attribute of the course alone; it has to be an attribute of the relationship.

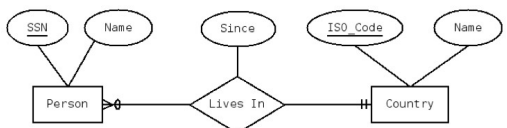


So, let's review. Relationships represent associations between entities, and we use a diamond for them. The degree of a relationship is the number of entity types it associates. Relationships may have attributes, and they do need to have cardinality constraints on both sides (these constraints are on the entity instances). Depending on the maximum cardinality on both sides, we classify relationships as one-to-one, one-to-many or many-to-many.

Relationships recap

SPSU

- Association between two or more entities
- Represented by diamond
- Degree
- May have attributes
- Cardinality constraints
- 1 to Many etc (use MAX on both sides)



20

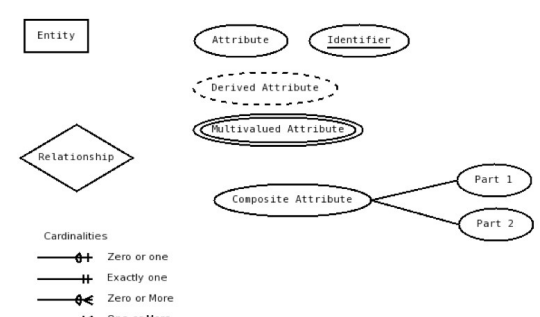
And another review of notation; we use rectangles for entities, ovals for attributes and diamonds for relationships.

We mark the identifier of an entity type by underlining it (relationships cannot have identifiers, since there can only be one relationship instances of a given type for a given collection of relationship instances).

We denote derived attributes by drawing their oval with dashed lines; we denote multivalued attributes by drawing their oval with double lines, and we denote composite attributes by linking their pieces to the composite attribute instead of to the entity of relationship.

What we've covered

SPSU



21

Relationships have cardinality constraints on each side; the constraints are on the minimum and maximum number of instances of the other type that an instance can be related to. The minimum is either zero or one, and the maximum either one or more, leaving to four possible combinations; zero or one, exactly one, zero-or-more, and one-or-more.

So, try this one; no peeking, try the solution first.

Practice: Products

SPSU

- We have two kinds of entities: **Products** and **Categories**. For each product we keep its id (the identifier), name, price, wholesale price, and profit margin, which is calculated from the price and wholesale price. For each category we keep its id (the identifier) and its name. Each product belongs to zero or more categories and each category can have zero or more products.

22

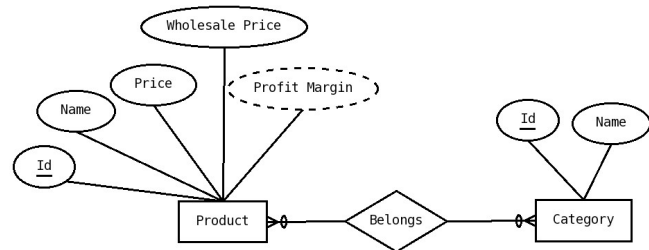
And here's the solution; we have two entity types, products and categories; for product, we underline its id, which is the identifier, add the other attributes, and use dashed lines for the profit margin, since it is derived; it can be calculated from the price and wholesale price.

Category is even simpler, since it has only id and name; once again, id, the identifier is underlined.

Now, we also need to keep track of which products belong to which categories; in order to do this, we need to use relationships; we cannot use attributes. A decent name for the relationship would be belongs, and we put cardinality constraints denoting that a product belongs to zero or more categories and that a category can have zero or more products.

Solution: Products

SPSU



23

Now, let's try this one; for this example, you will model CD as an entity and you're not allowed to have other entities or relationships, just attributes.

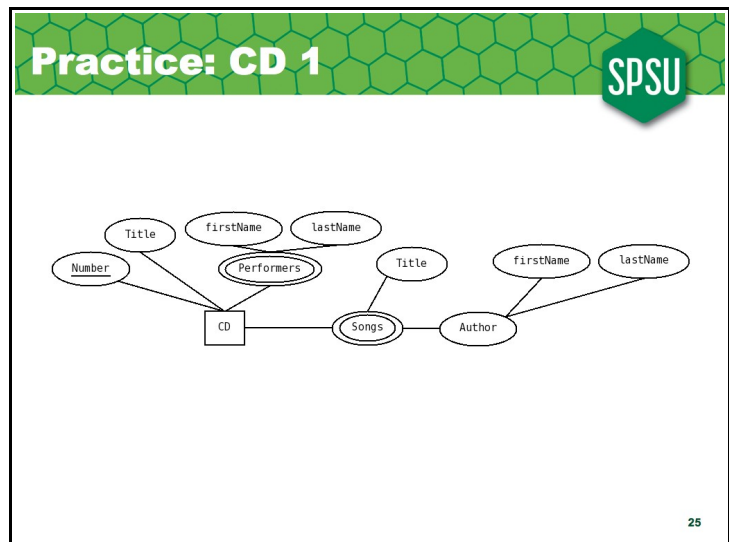
Practice: CD 1

SPSU

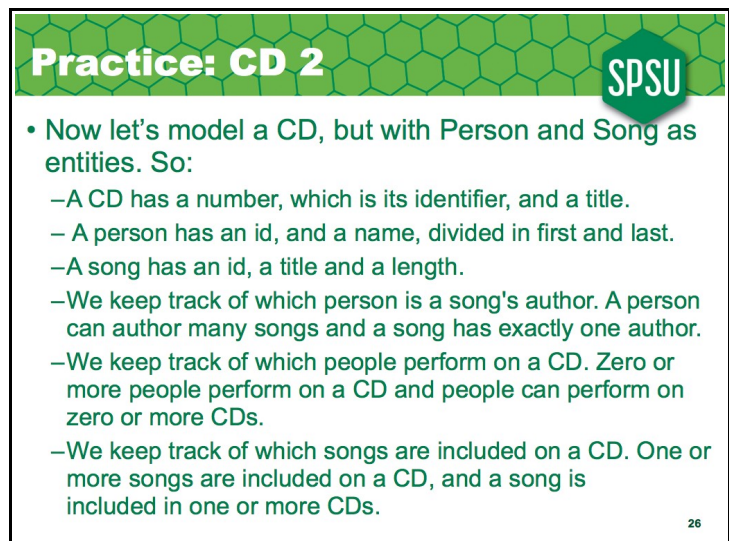
- We want to model a CD as an entity. The only entity is the CD and everything else is modeled as attributes.
 - a) A CD has a number, which is its identifier.
 - b) A CD has a title.
 - c) A CD has zero or more performers; for each performer we keep their first and last name.
 - d) A CD has zero or more songs. For each song we keep its title and the name of its author (with name divided into first and last)

24

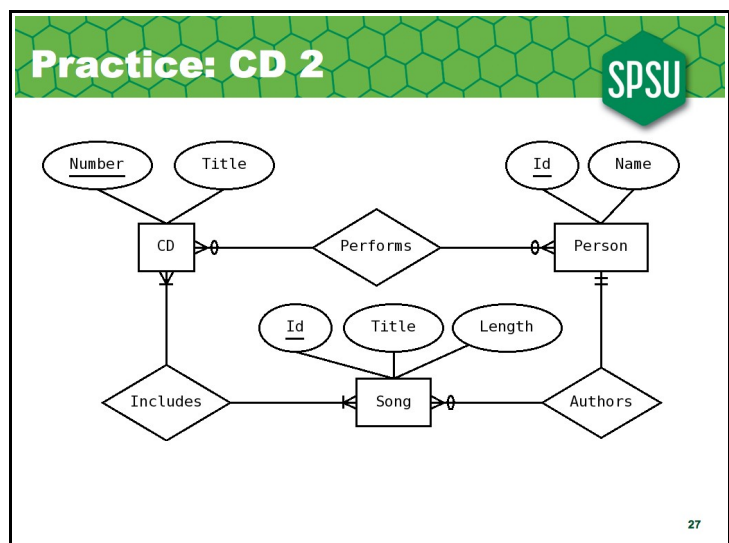
And here's the solution.



Now let's represent the same situation, but with more entities and relationships.



And here's the solution.



Sometimes, we need to define relationships that associate two entities of the same entity type; we call those *recursive* relationships. Most relationships are binary, and most recursive relationships are too; they associate two entities; recursive relationships are often used to represent hierarchies and graphs.

Although they are not that complicated, we need to make explicit the concept of a *role* an entity instance plays in a relationship instance, since we cannot use the entity types to distinguish.

Recursive Relationships

SPSU

- Two or more of the entities in the relationship are of the same type.
- Simplest is binary, two entities of the same type.
- Some books call them unary which doesn't make much sense
- Can be used to represent hierarchies (ie. supervises relationship)
-

28

So, let's do an example. Imagine we want to represent when an employee supervises another employee; we could define a relationship called supervises; we could initially think there are two kinds of employees, supervisor and supervised, and tentatively start with a diagram like this.

However, we'll soon realize that the supervisors can also have supervisors of their own, forming a hierarchy, and we cannot use a different entity type to represent each level on the hierarchy, so we need to do something different.

Example: Supervises

SPSU

- Want to represent supervises relationship, an employee supervises other employees.
- We could *start* with something like this:



29

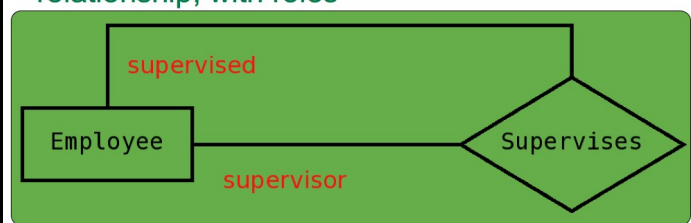
So, we need to turn the relationship onto itself, so it associates two instances of the employee entity type; now, to distinguish those instances, we need to add *role names* to each side of the relationship; good names would be supervisor and supervised.

Example: Supervises

SPSU

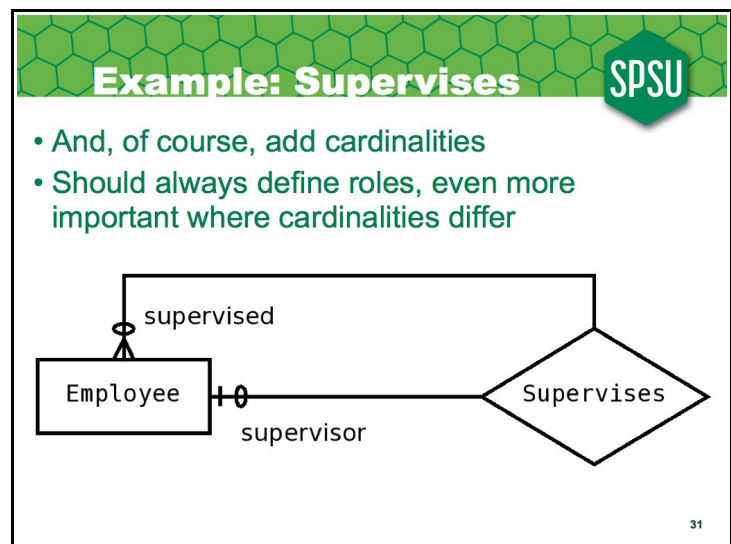


- but both supervisor and supervisee are employees, so we need a recursive relationship, with roles

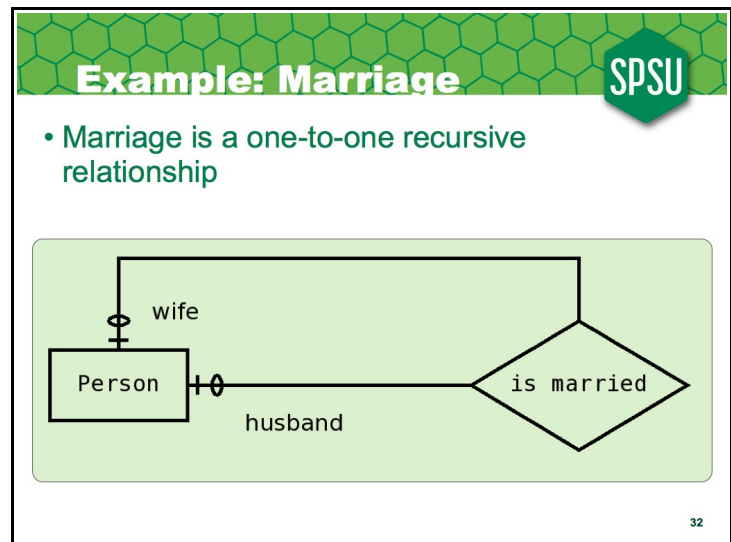


30

And finally we add the cardinality constraints; notice the constraints constrain employees, so in this case we would say an Employee supervises zero or more employees, and has zero or one supervisors.



Although recursive relationships are used to represent hierarchies, not all recursive relationships represent hierarchies; for example, we could represent marriages by having a relationship, *is married*, associating the husband and wife; this would be a one-to-one relationship, and so does not represent a hierarchy.



Another example, which represents a hierarchy, is the part-whole relationship, or bill of materials, commonly used in manufacturing; an item is composed of many parts, and may, in turn, be a part of many other items.

