COP 4710: Database Systems
Fall 2010

Chapter 2 – Introduction to Data Modeling

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4710/fall2010

Department of Electrical Engineering and Computer Science
University of Central Florida

Introduction to Data Modeling

- Semantic data models attempt to capture the "meaning" of a database. Practically, they provide an approach for conceptual data modeling.
- Over the years there have been several different semantic data models that have been proposed.
- By far the most common is the *entity-relationship data model*, most often referred to as simply the *E-R data model*.
- The E-R model is often used as a form of communication between database designers and the end users during the developmental stages of a database.

Introduction to Data Modeling (cont.)

- The E-R model contains an extensive set of modeling tools, some of which we will not be concerned with as our primary objective is to give you some insight into conceptual database design and not learning all of the ins and outs of the E-R model.
- Another conceptual modeling which is becoming more common is the *Object Definition Language* (ODL) which is an object-oriented approach to database design that is emerging as a standard for object-oriented database systems.

Database Design

- The database design process can be divided into six basic steps. Semantic data models are most relevant to only the first three of these steps.

1. *Requirements Analysis:* The first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. Often this is an informal process involving discussions with user groups and studying the current environment. Examining existing applications expected to be replaced or complemented by the database system.

Database Design (cont.)

2. *Conceptual Database Design:* The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold on this data.

3. *Logical Database Design:* A DBMS must be selected to implement the database and to convert the conceptual database design into a database schema within the data model of the chosen DBMS.

Database Design (cont.)

4. *Schema Refinement*: In this step the schemas developed in step 3 above are analyzed for potential problems. It is in this step that the database is *normalized*. Normalization of a database is based upon some elegant and powerful mathematical theory. We will discuss normalization later in the term.

5. *Physical Database Design:* At this stage in the design of a database, potential workloads and access patterns are simulated to identify potential weaknesses in the conceptual database. This will often cause the creation of additional indices and/or clustering relations. In critical situations, the entire conceptual model will need restructuring.

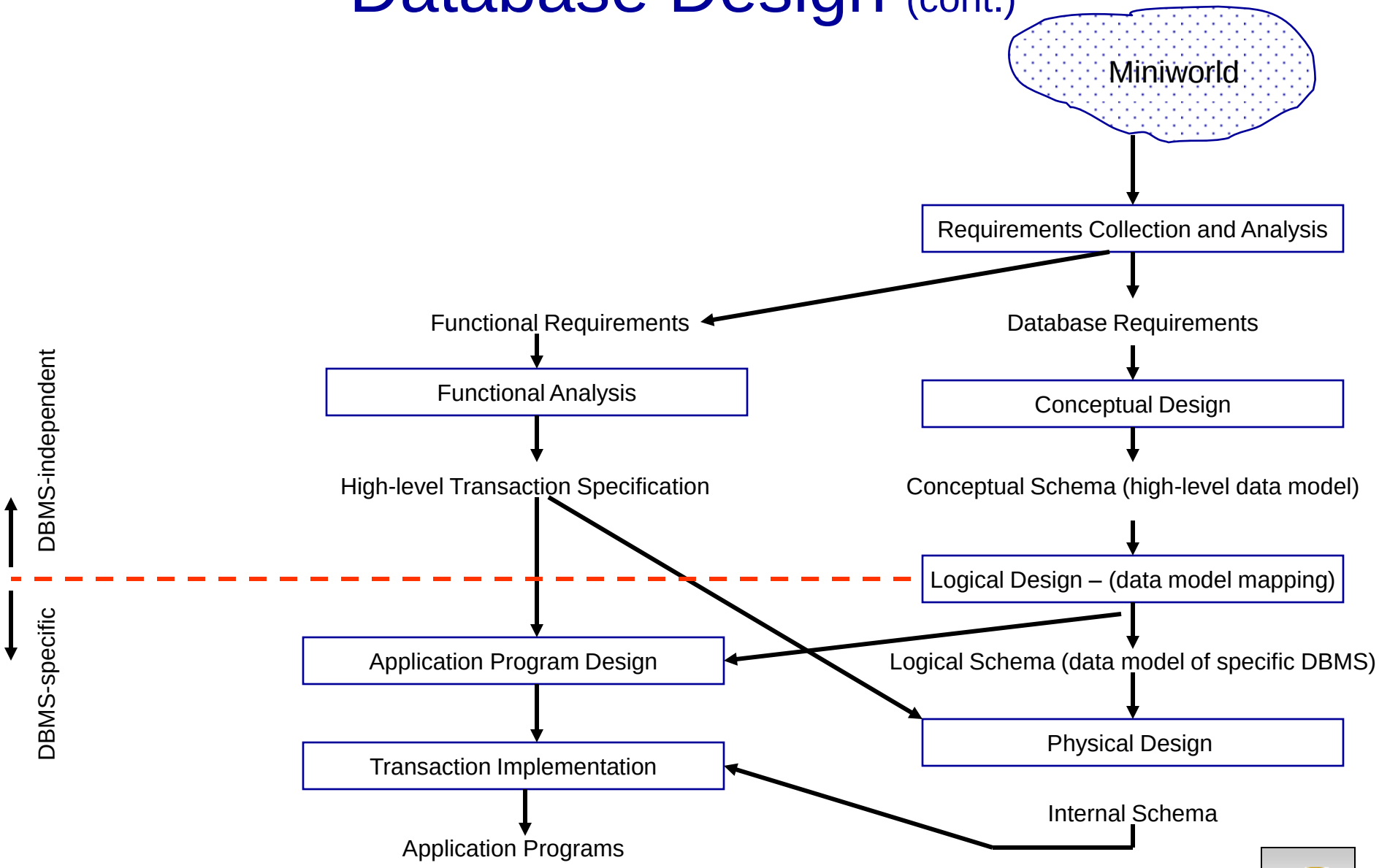
Database Design (cont.)

6. *Security Design:* Different user groups are identified and their different roles are analyzed so that access patterns to the data can be defined.

- There is often a seventh step in this process with the last step being a *tuning phase*, during which the database is made operational (although it may be through a simulation) and further refinements are made as the system is "tweaked" to provide the expected environment.

- The illustration on the following page summarizes the main phases of database design.

Database Design (cont.)



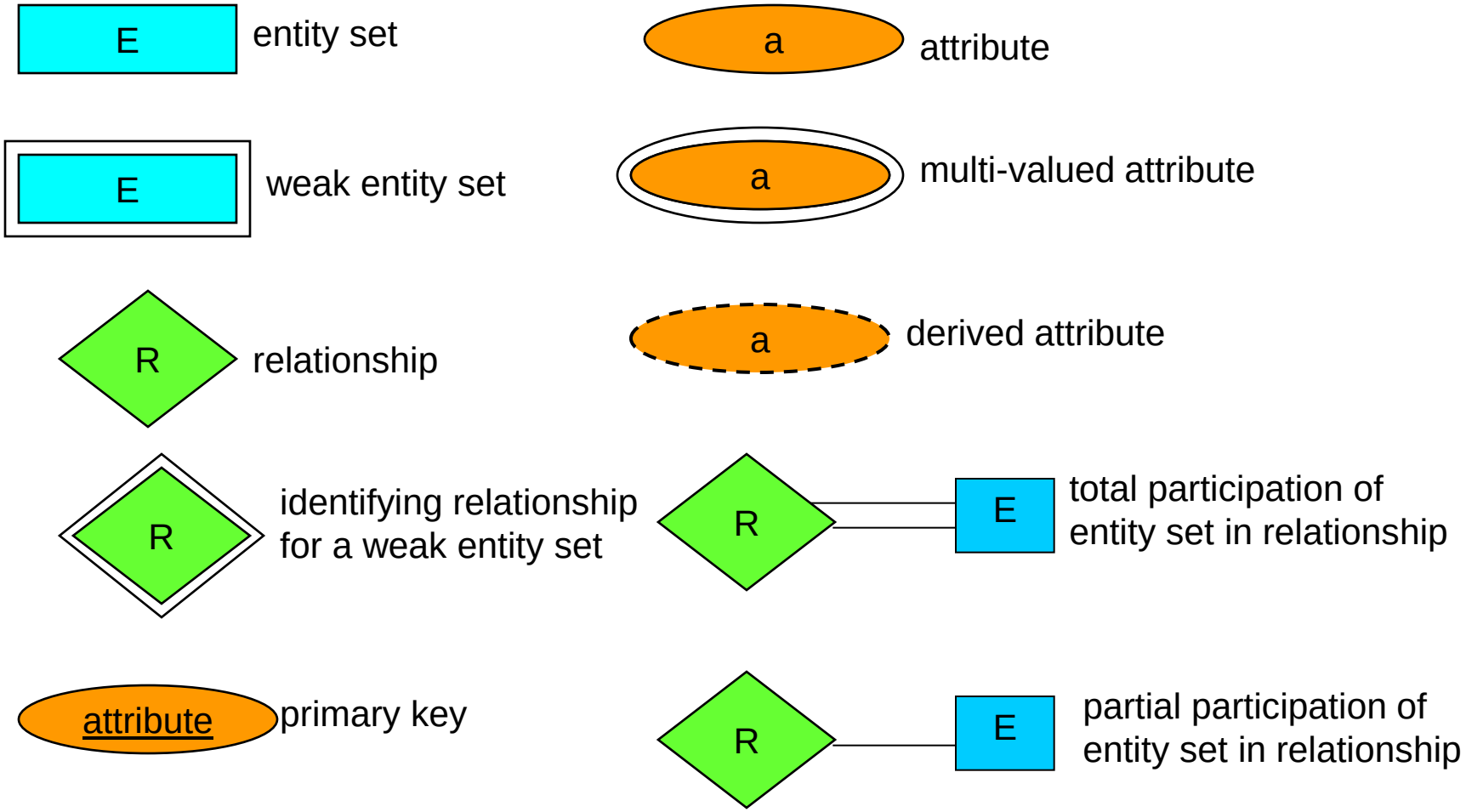
The Entity-Relationship Model

- The E-R model employs three basic notions: *entity sets, relationship sets*, and *attributes*.
- An *entity* is a "thing" or "object" in the real world that is distinguishable from all other objects. An entity may be either concrete, such as a person or a book, or it may be abstract, such as a bank loan, or a holiday, or a concept.
- An entity is represented by a set of *attributes*. Attributes are descriptive properties or characteristics possessed by an entity.
- An *entity set* is a set of entities of the same type that share the same attributes. For example, the set of all persons who are customers at a particular bank can be defined as the entity set *customers*.

The Entity-Relationship Model (cont.)

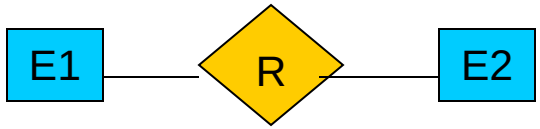
- Entity sets do not need to be disjoint. For example, we could define the entity set of all persons who work for a bank (*employee*) and the entity set of all persons who are customers of the bank (*customers*). A given person entity might be an *employee*, a *customer*, both, or neither.
- For each attribute, there is a permitted set of values, called the *domain* (sometimes called the *value set*), of that attribute. More formally, an attribute of an entity set is a function that maps from the entity set into a domain. Since an entity set may have several attributes, each entity in the set can be described by a set of <attribute, data-value> pairs, one for each attribute of the entity set.
- A database contains a collection of entity sets.

E-R Model Notation

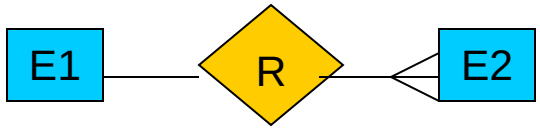


E-R Model Notation (cont.)

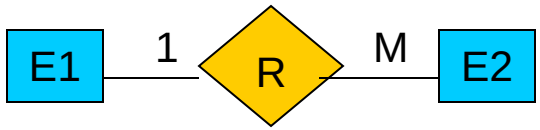
discriminating attribute of
a weak entity set



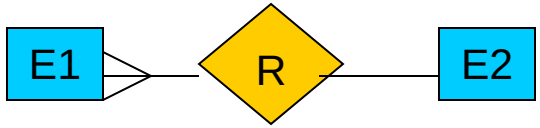
1:1 cardinality from E1 to E2



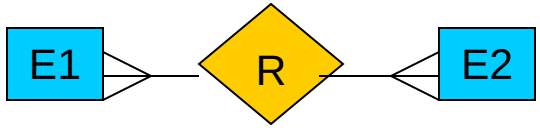
1:M cardinality from E1 to E2



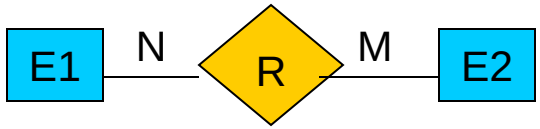
alternate form for 1:M cardinality from E1 to E2



M:1 cardinality from E1 to E2

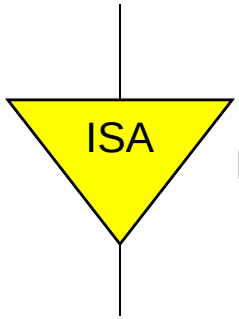


M:M cardinality from E1 to E2

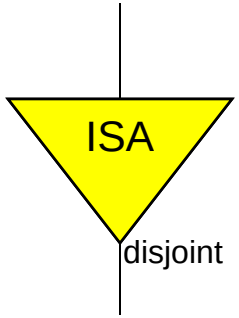


alternate form for M:M cardinality from E1 to E2

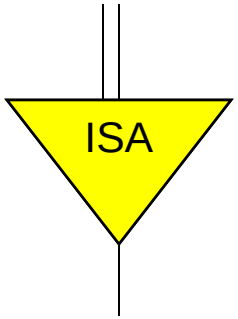
E-R Model Notation (cont.)



ISA (specialization or generalization)(partial participation)

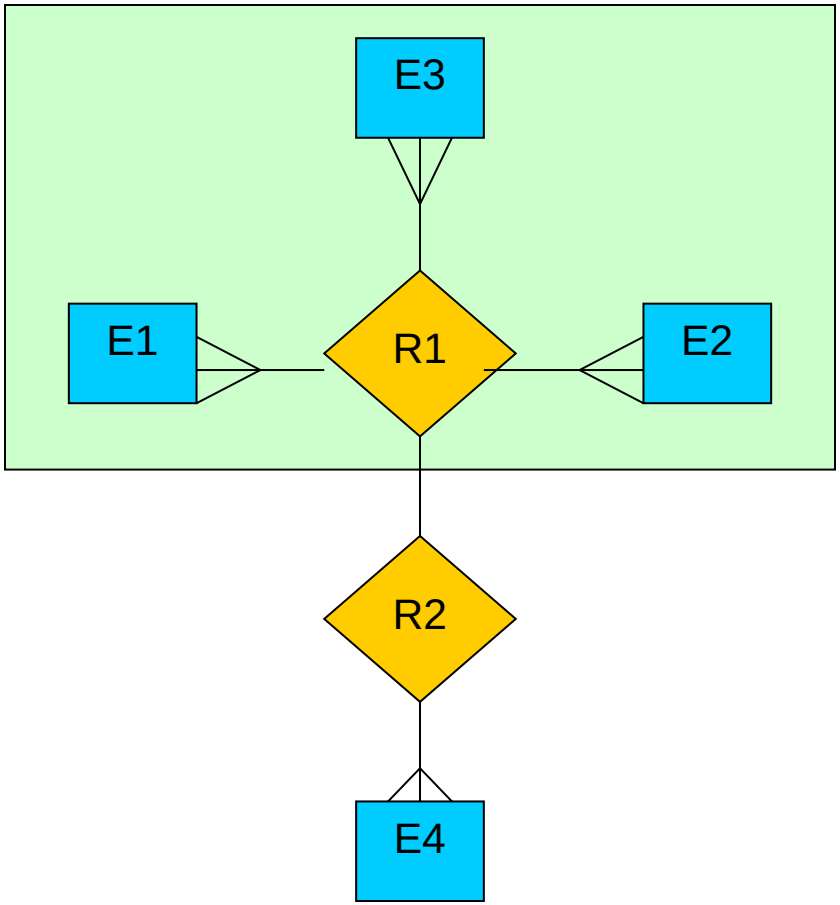


Disjoint ISA (specialization or generalization)

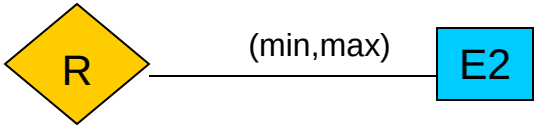


Total generalization

E-R Model Notation (cont.)

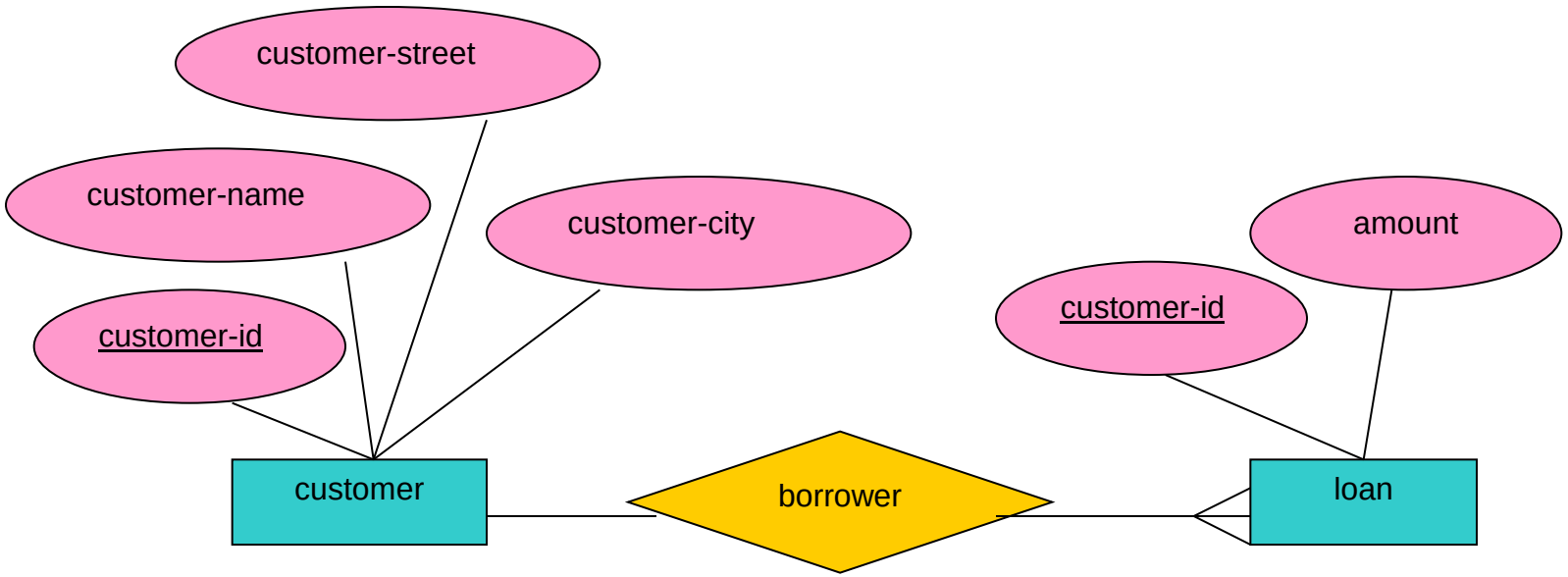


Aggregation: box drawn around relationship which is treated as an entity



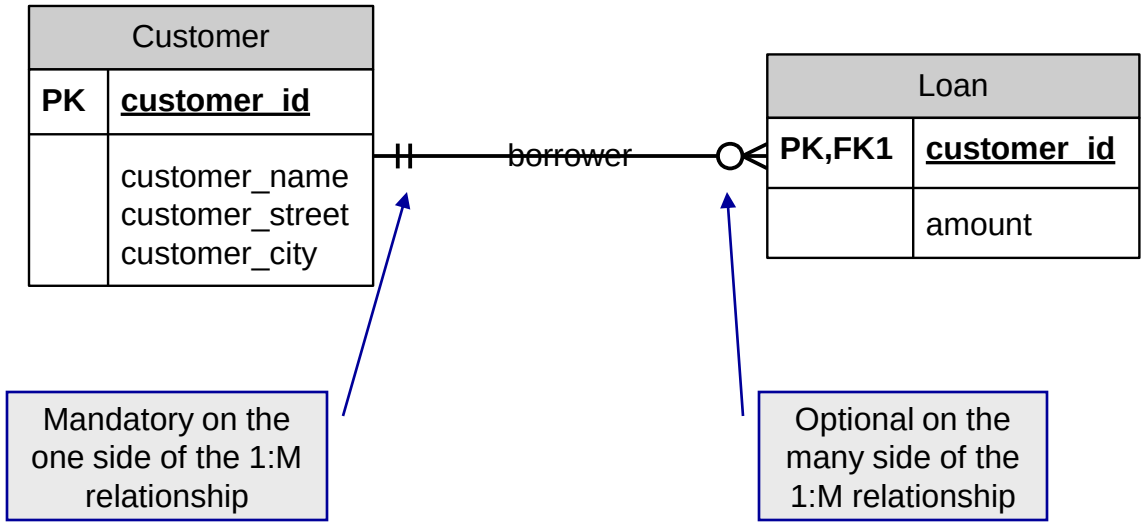
Structural constraint: (min,max) on the participation of an entity in a relationship

Example E-R Diagram (ERD)

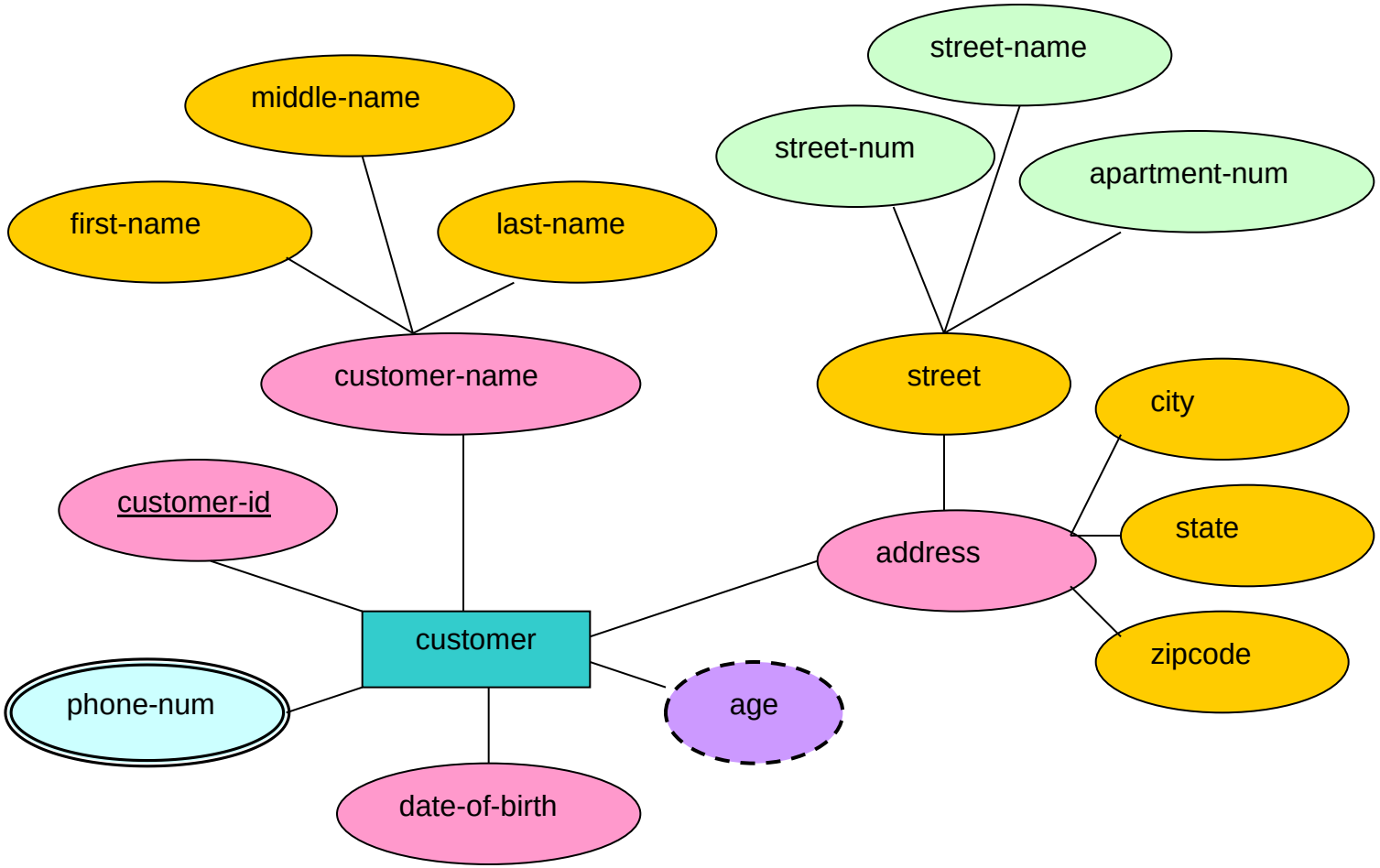


Example E-R Diagram

Visio Pro 2003 Version



Another Example ERD



Attributes in the E-R Model

- As used in the E-R model, an attribute can be characterized by the following attribute types:
- *Simple* or *Composite*: A simple attribute contains no subparts while a composite attribute will contain subparts. For example, consider the attribute *name*. If *name* represents a simple attribute then we must treat the first name, middle name, and last name as an atomic, indivisible attribute. On the other hand, if *name* represents a composite attribute then we have the option of dealing with the entire name as a whole or dealing only with one of the subparts. For example, we could look only at last names, something that we could not do with a simple attribute.

Attributes in the E-R Model (cont.)

- *Single-valued* or *Multi-valued*: A single-valued attribute may have at most one value at any particular time instance. A multiple-valued attribute may have several different values at any particular time instance.
  - For example, consider a particular course at UCF. At any given moment the number of students enrolled in that course is a single value, say 100, but not 100, 80, and 45! On the other hand, some attributes may contain different values at the same time instant. For example, consider an attribute of the entity set student which might be phone-number. At any given time instant a student may have several different phone numbers and thus a multi-valued attribute would be best to accurately model the student. It is also common to place lower and upper bounds on the number of different values that a multi-valued attribute may have at any given time.

Attributes in the E-R Model (cont.)

- *Derived*: This is an attribute whose value is derived (computed) from the values of other related attributes or entities.
  - For example, suppose that the bank customer entity set contains an attribute loans-held, which represents the number of loans a customer has from the bank. The value of this attribute can be computed for each customer by counting the number of loan entities associated with that customer.

Attributes in the E-R Model (cont.)

- *Null*: An attribute takes a null value when an entity does not have a value for it. Null values are usually special cases that can be handled in a number of different ways depending on the situation.
  - For example, it could be interpreted to mean that the attribute is "not applicable" to this entity, or it could mean that the entity has a value for this attribute but we don't know what it is. We will see later in the term how different systems handle null values and the different interpretations that may be associated with this special value.

Relationships in the E-R Model

- A *relationship* is an association among several entities.
  - For example, we can define a relationship that associates you as a student in COP 4710. This relationship might specify that you are *enrolled* in this course.

A *relationship set* is a set of relationships of the same type.
More formally, it is a mathematical relation on n ≥ 2 (possibly non distinct) entity sets.

If E_1, E_2, …, E_n are entity sets, then a relationship set *R* is a subset of:

$$\{(e_1, e_2, \cdots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \cdots, e_n \in E_n\}$$

where $(e_1, e_2, \cdots, e_n)$ is the relationship.

Relationships in the E-R Model (cont.)

- The association between entity sets is referred to as participation; that is, the entity sets E_1, E_2, …, E_n participate in relationship *R*.
- A *relationship instance* in an E-R schema represents an association between named entities in the real world enterprise which is being modeled.
- A relationship may also have attributes which are called *descriptive attributes*. For example, considering the bank scenario again, suppose that we have a relationship set *depositor* with entity sets *customer* and *account*. We might want to associate with the *depositor* relationship set a descriptive attribute called *access-date* to indicate the most recent date that a customer accessed their account.
-

Constraints in the E-R Model

- As we have mentioned earlier, a the values contained within a given database often have constraints placed upon them to ensure that they accurately model the real world enterprise captured in the database.
- The E-R model has the capability of modeling certain types of these constraints.
- We will focus on two types of constraints: mapping cardinalities and participation constraints, which are two of the more important types of constraints.

Constraints in the E-R Model (cont.)

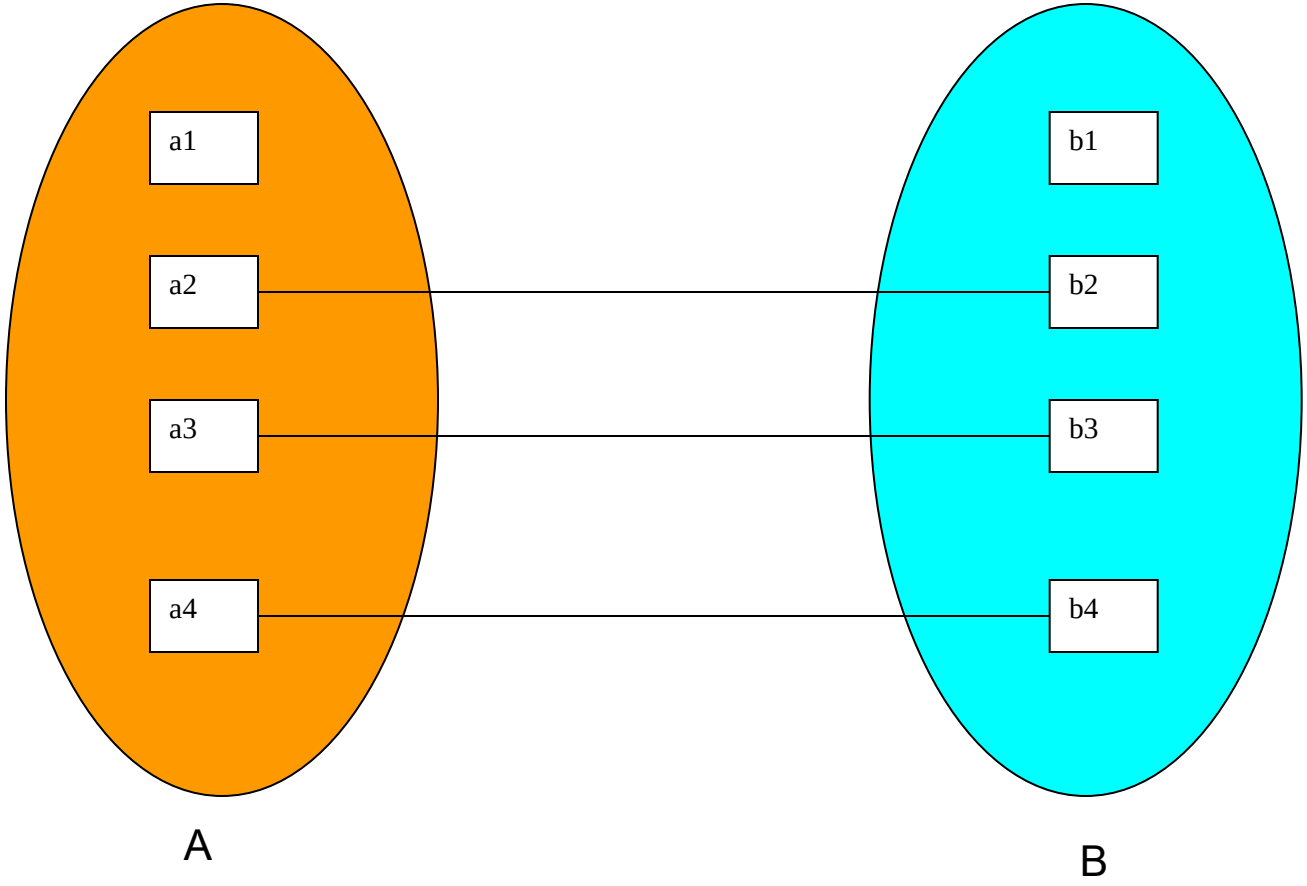
- Mapping cardinalities (also called *cardinality ratios*), express the number of entities to which another entity can be associated via a relationship set.

- Mapping cardinalities are most useful in describing binary relationships, although they can be helpful in describing relationship sets that involve more than two entity sets. We will focus only on binary relationships for now.

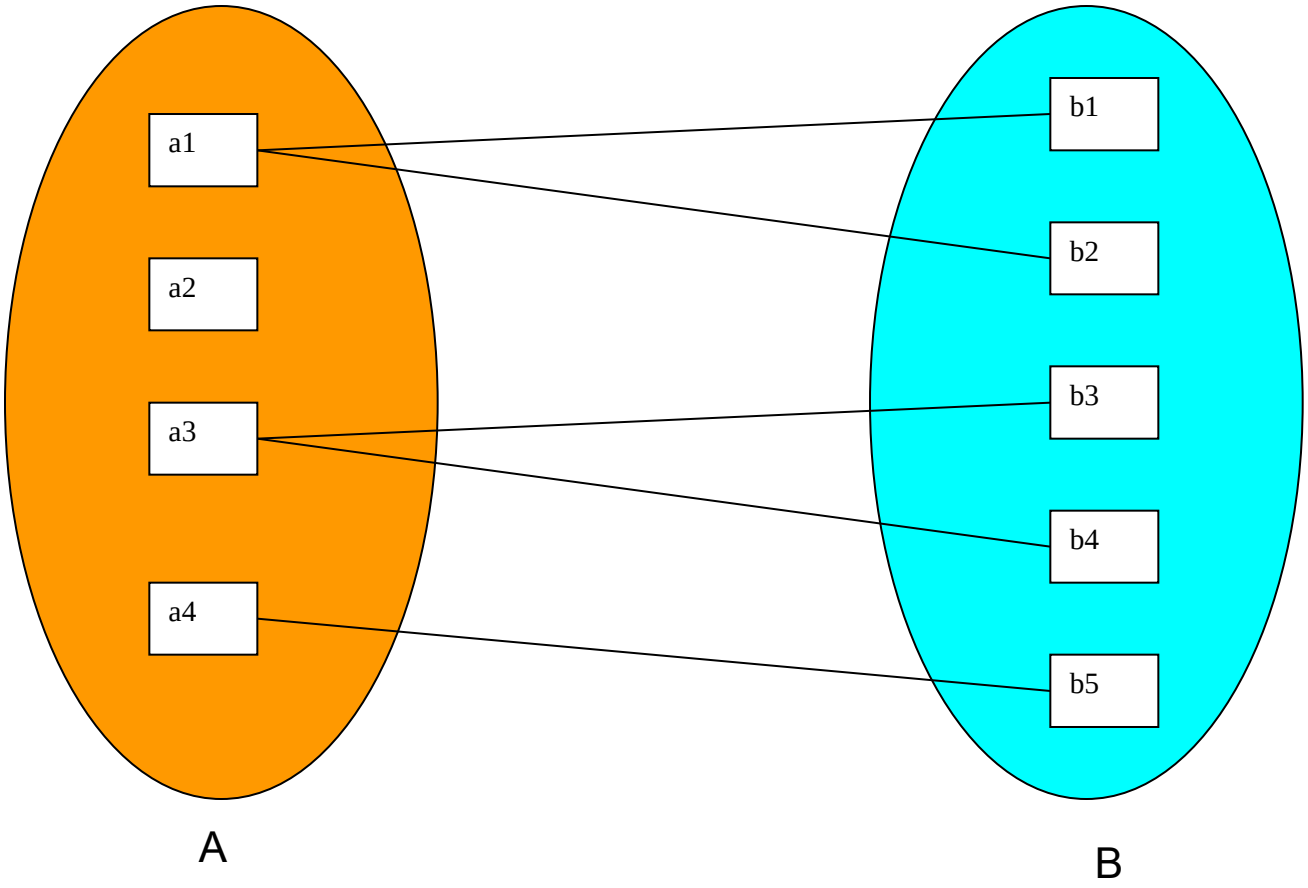
- For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:

- (1:1) *one to one from A to B*
- (1:M) *one to many from A to B*
- (M:1) *many to 1 from A to B*
- (M:M) *many to many from A to B*

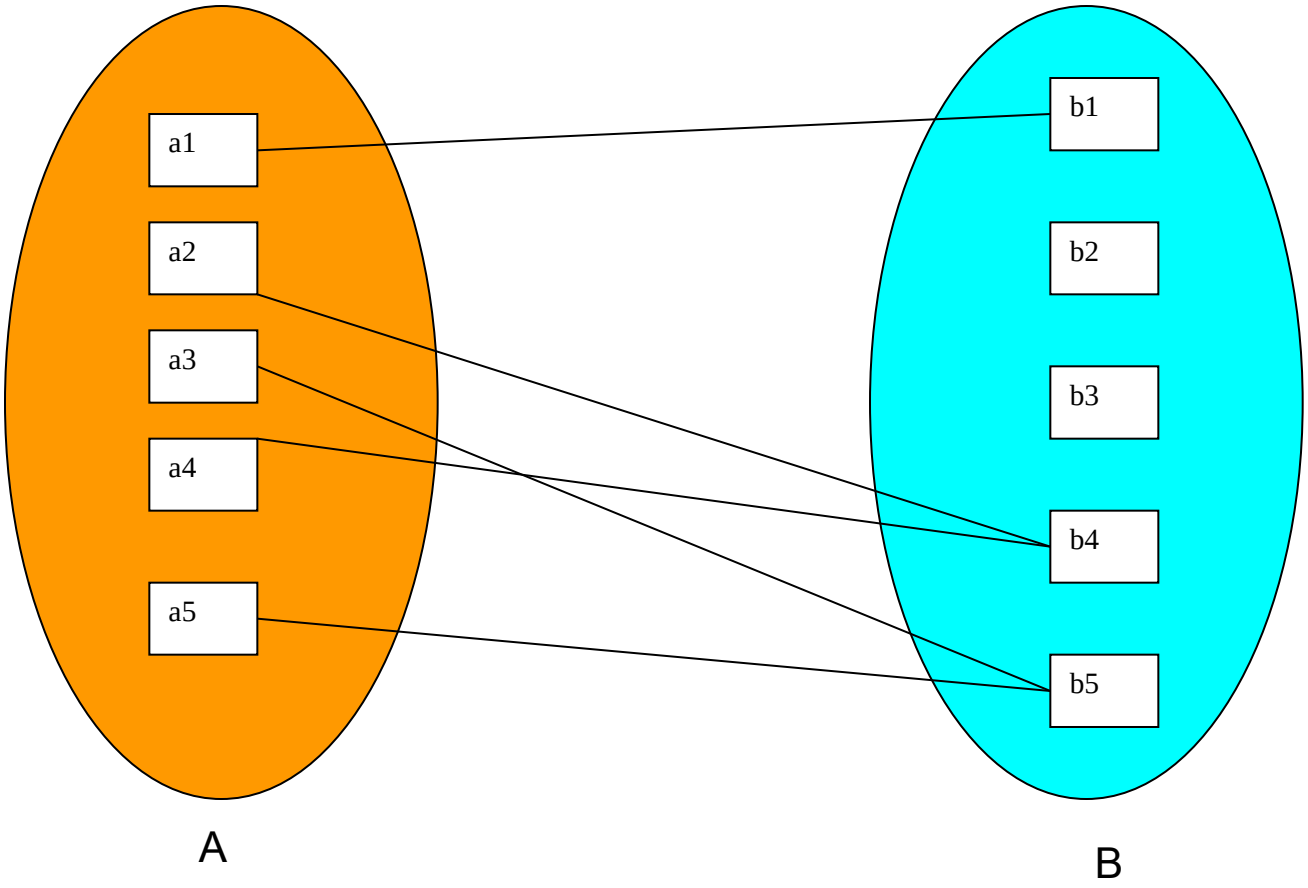
Mapping Cardinality: 1:1 from A to B



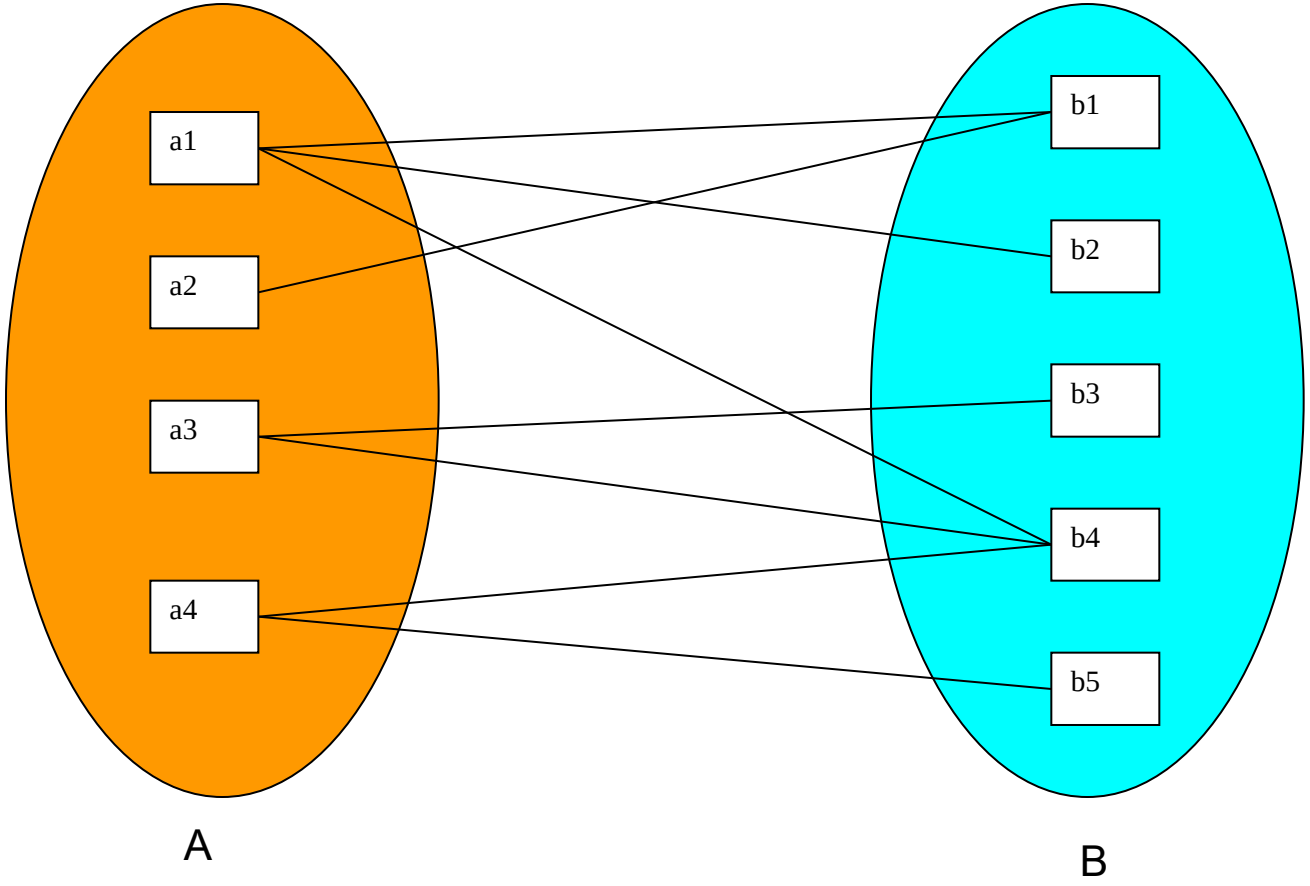
Mapping Cardinality: 1:M from A to B



Mapping Cardinality: M:1 from A to B



Mapping Cardinality: M:M from A to B



Participation Constraints in the E-R Model

- The participation of an entity set *E* in a relationship set *R* is said to be *total* if every entity in *E* participates in at least one relationship in *R*. If only some of the entities in *E* participate in a relationship in *R*, the participation of entity set *E* is relationship *R* is said to be *partial*.

- As examples, consider the banking example again. We would expect that every loan entity be related to at least one customer through a *borrower* relationship. Therefore the participation of *loan* in the relationship set *borrower* is total. In contrast, an individual can be a bank customer whether or not they have a loan with the bank. Thus, it is possible that only some of the customer entities will be related to a loan entity through the borrowers relationship. Therefore, the participation of the customer entity set in the borrower relationship is partial.

Keys of an Entity Set

- We must have some mechanism for specifying how entities within a given entity set are distinguished.
- Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes. Therefore, the values of the attribute values of an entity must be such that they can *uniquely identify* the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.
- A *key* allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from one another.

Primary Keys, SuperKeys and Candidate Keys

- A *superkey* is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set. Suppose that we have an entity set modeling the students in COP 4710. Suppose that we have the following schema for this entity set:

  Students(SS#, name, address, age, major, minor, gpa, spring-sch)

- Among the attributes which we have associated with each student must be a set of attributes which will uniquely distinguish each student. Suppose that we define this set of attributes to be:

  (SS#, name, major, minor)

Primary Keys, SuperKeys and Candidate Keys
(cont.)

- This set of attributes (SS#, name, major, minor) defines a superkey for the entity set Students. Notice that the set of attributes (SS#, name) also defines a superkey for this entity set, because given this second set of attributes we can still uniquely distinguish each student in the set. The concept of a superkey is not a sufficient definition of a key because the superkey, as we can see from this example, may contain extraneous attributes.

Primary Keys, SuperKeys and Candidate Keys (cont.)

- If the set *K* is a superkey of entity set *E*, then so too is any superset of *K*. We are interested only in superkeys for which no proper subset of *K* is a superkey. Such a minimal superkey is called a *candidate key*.
- For a given entity set *E* it is possible that there may be several distinct sets of attributes which are candidate keys.
- Either there is only a single such set of attributes or there are several distinct sets from which only one is selected by the database designer and this set of attributes defines the *primary key* which is typically referred to simply as the *key* of the entity set.

Primary Keys, SuperKeys and Candidate Keys
(cont.)

- A key (primary, candidate, and super) is a property of the entity set, rather than of the individual entities. Any two individual entities in the set are prohibited from having the same value on all attributes which comprise the key attributes at the same time. This constraint on the allowed values of an entity within the set is a *key constraint*.

- The database designer must use care in the selection of the set of attributes which comprise the key of an entity set to: (1) be certain that the set of attributes guarantees the uniqueness property, and (2) be certain that the set of key attributes are never, or very rarely, changed.

Relationship Sets

- The primary key of an entity set allows us to distinguish among the various entities in the set. There must be a similar mechanism which allows us to distinguish among the various relationships in a relationship set.

- Let R be a relationship set involving entity sets E_1, E_2, …, E_n. Let K_i denote the set of attributes which comprise the primary key of entity set E_i. For now lets assume that

  - (1) all attributes names in all primary keys are unique, it will make the notation easier to understand and it really isn't a problem if the names aren't unique anyway, and

  - (2) each entity set participates only once in the relationship.

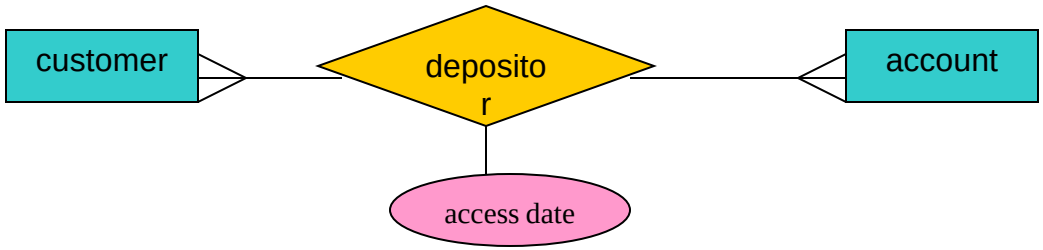
- Then the composition of the primary key for the relationship set depends on the set of attributes associated with the relationship set R in the following ways:

Relationship Sets (cont.)

- (a) If the relationship set *R* has no attributes associated with it, then the set of attributes: $K_1 \cup K_2 \cup \ldots \cup K_n$ describes an individual relationship in set *R*.
- (b) If the relationship set *R* has attributes $a_1, a_2, \ldots, a_m$ associated with it, then the set of attributes: $K_1 \cup K_2 \cup \ldots \cup K_n \cup \{ a_1, a_2, \ldots, a_m \}$ describes an individual relationship in set *R*.
- In **both** of these cases, the set of attributes: $K_1 \cup K_2 \cup \ldots \cup K_n$ forms a superkey for the relationship set.

Effect of Cardinality Constraints on Keys

- The structure of the primary key for the relationship set depends upon the mapping cardinality of the relationship set. Consider the following case:



- This E-R diagram represents a many to many cardinality for the relationship *depositor* with an attribute of *access date* associated with the relationship set with two entities *customer* and *account* participating in the relationship. The primary key of the relationship *depositor* will consist of the union of the primary keys of *customer* and *account*.

Effect of Cardinality Constraints on Keys

- To further clarify this situation consider for a moment the schemas of these two entity sets:

  Customer (customer-id, customer-name, address, city)

  Account (account-number, balance)

- A many-to-many relationship between these two sets means that it is possible for one customer to have several accounts and similarly for a given account to be held by several customers.

- To uniquely identify a relationship between two entities in *customers* and *accounts* will require the union of the primary keys in both entity sets.

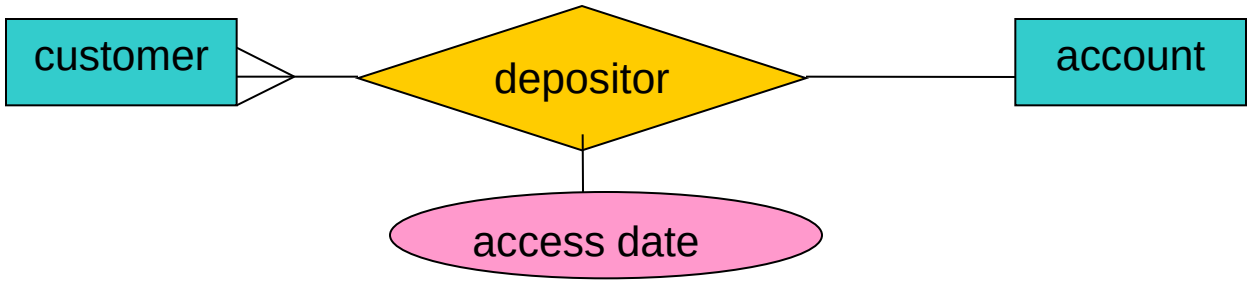
Effect of Cardinality Constraints on Keys (cont.)

- In order to "see" the last deposit made to specific account number requires that we specify by whom the deposit was made since several account holders may have made deposits to the same account.
- The schema for the *depositor* relationship is then:

Depositor (customer-id, account-number, access-date)

Effect of Cardinality Constraints on Keys (cont.)

- Now consider the case when a customer is only allowed to have one account. This means that the *depositor* relationship is many-to-one from *customer* to *account* as shown in the following diagram.



- In this case the primary key of the *depositor* relationship is simply the primary key of the *customer* entity set. To clarify this, again look at the schemas of the entity sets:

Customer (customer-id, customer-name, address, city)

Account (account-number, balance)

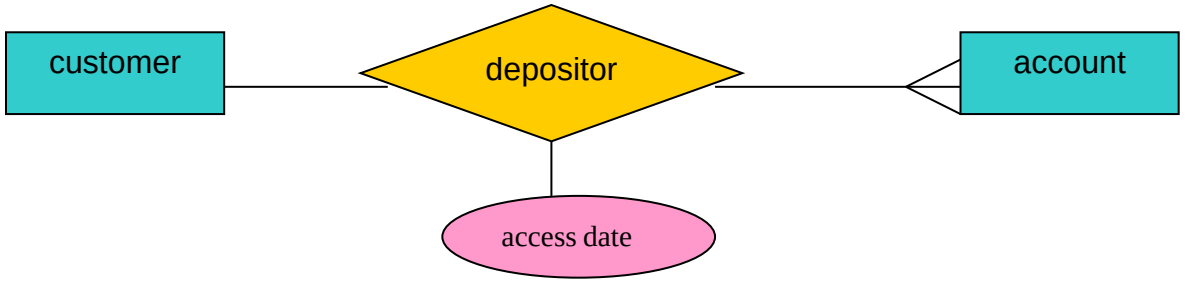
Effect of Cardinality Constraints on Keys (cont.)

- A many-to-one relationship means that a given customer can have only a single account then the primary key of the *depositor* relationship is simply the primary key of the *customer* set since for a given customer they could only make a single most recent deposit since they only "own" one account, so specifying the account number is not necessary to identify a unique deposit by a given customer.
- The schema for the *depositor* relationship set is then:

  Depositor (customer-id, access-date)

Effect of Cardinality Constraints on Keys (cont.)

- Now consider the case when the *depositor* relationship is many-to-one from *account* to *customer*.



- A many-to-one relationship from account to customer means that each account is owned by at most one customer but each customer may have more than one account. In this situation the primary key of the *depositor* relationship is simply the primary key of the *account* entity set since there can be at most one most recent deposit to a given account because at most one customer could make the deposit. We do not need to uniquely identify which customer made the deposit in question because there could only be one.

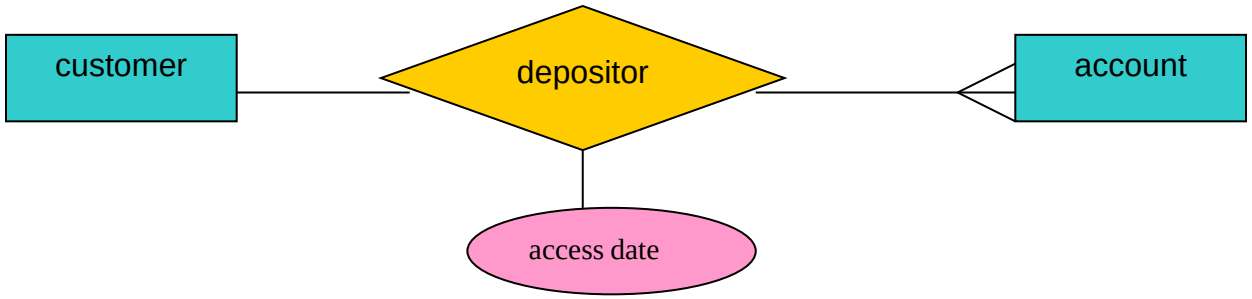
- The schema for the *depositor* relationship is then:

  Depositor (<u>account-id</u>, access-date)

Placement of Relationship Attributes

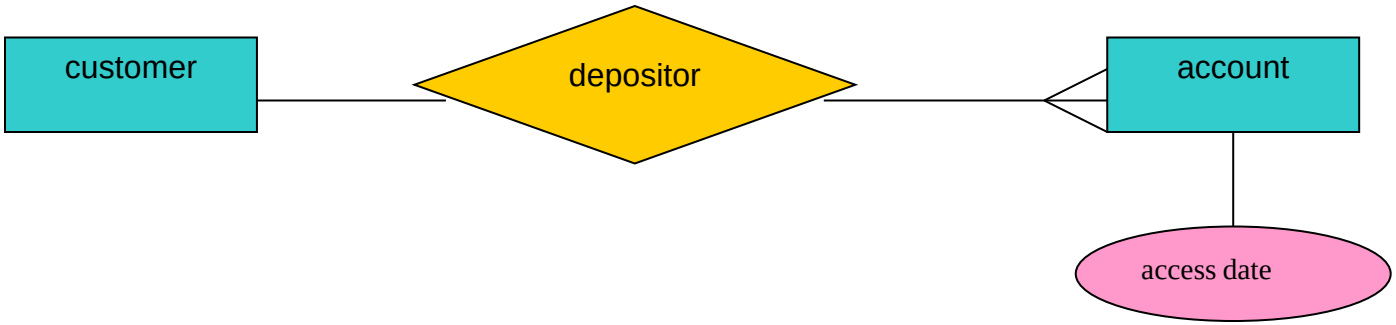
- Just as the cardinality of a relationship set affects the set of attributes which comprise the primary key of the relationship set, so too does it affect the placement of the attributes.

- The attributes of a one-to-one or one-to-many relationship set can be associated with one of the participating entity sets, rather than with the relationship set itself. For example consider the following case:



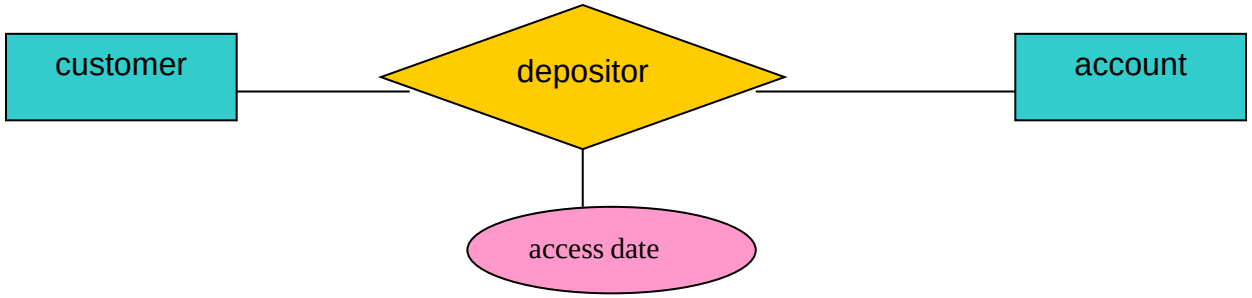
Placement of Relationship Attributes (cont.)

- The attribute *access-date* could be associated with the *account* set without loss of information. Since a given account can be owned by at most one customer it could have at most one access-date which could be stored in the *account*



Placement of Relationship Attributes (cont.)

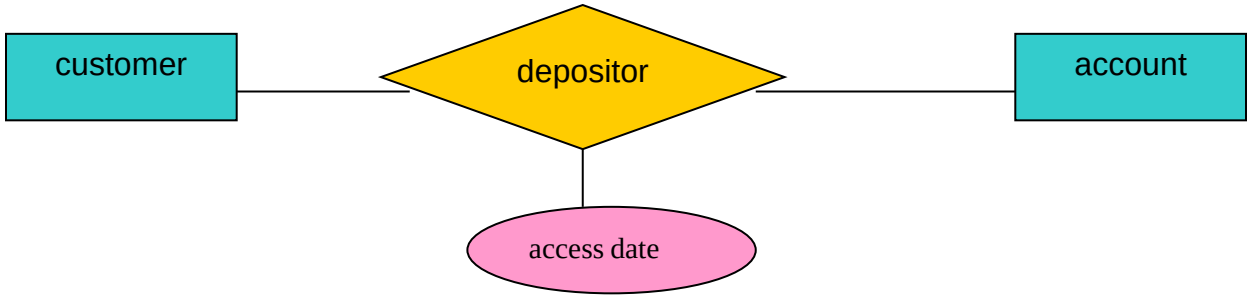
Now consider the following case:



- The attribute *access-date* could be associated with either the *customer* set or the *account* set without loss of information. In this case a given account can be owned by at most one customer and a given customer can own at most one account. Therefore, if the *access-date* attribute is stored with the *customer* set then it must refer to the last access by this customer on the only account they can have. Similarly, if the *access-date* attribute is stored with the *account* set, then it must refer to the last access on this account by the only customer who owns this account.

Placement of Relationship Attributes (cont.)

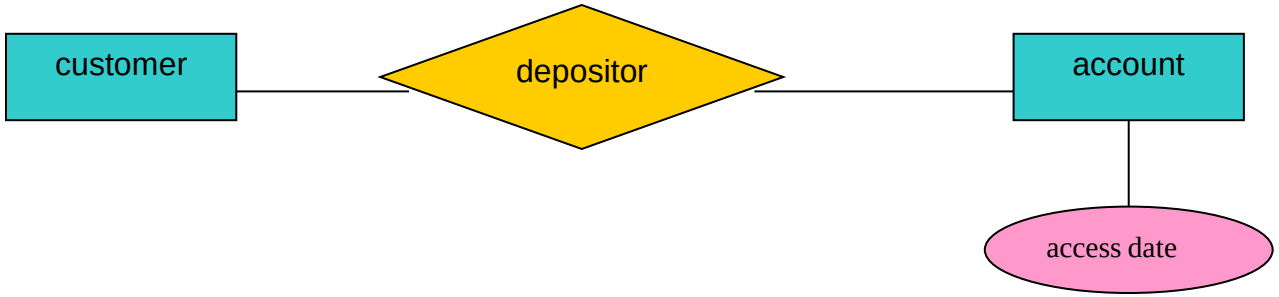
Now consider the following case:

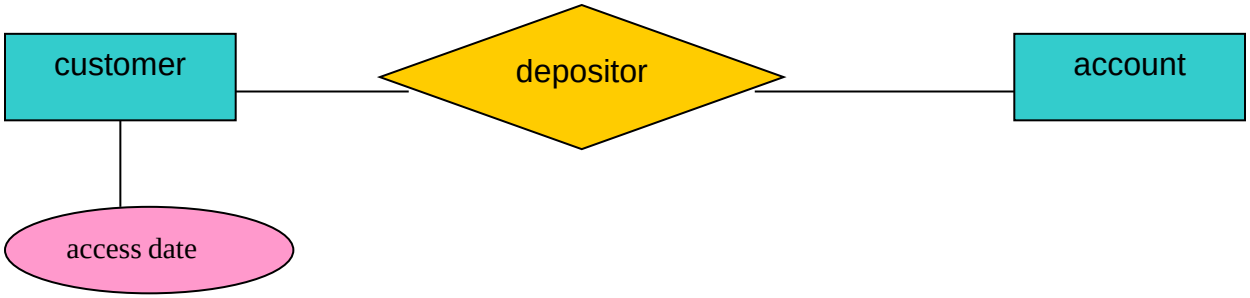


- The attribute *access-date* could be associated with either the *customer* set or the *account* set without loss of information. In this case a given account can be owned by at most one customer and a given customer can own at most one account. Therefore, if the *access-date* attribute is stored with the *customer* set then it must refer to the last access by this customer on the only account they can have. Similarly, if the *access-date* attribute is stored with the *account* set, then it must refer to the last access on this account by the only customer who owns this account.

Placement of Relationship Attributes (cont.)

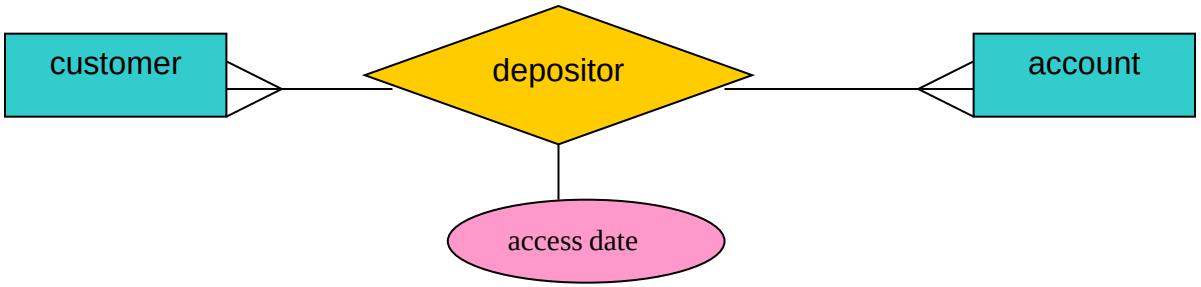
- Therefore, either diagram below would be a correct representation of this situation:





Placement of Relationship Attributes (cont.)

- When the relationship set has a cardinality constraint of many-to-many, the situation is much clearer. Consider the following situation:



- account may be owned by several customers, we see that associating the *access-date* attribute with either entity set will not properly model this situation without the loss of information. If we need to model the date that a specific customer last accessed a specific account the *access-date* attribute must be an attributed of the *depositor* relationship set, rather than one of the participating entities. For example, if *access-date* were an attribute of *account* we could not determine which customer made the last access to the account. If *access-date* were an attribute of *customer* we could not determine which account the customer last accessed.

Further Design Issues

- The notions of an entity set and a relationship set are not precise.
- It is possible to define a set of entities and the relationships among them in a number of different ways. We'll look briefly at some of these different approaches to the modeling of the data.
- To some extent this is where the "art" of database design becomes tricky. Sometimes several different design scenarios may all look equally plausible and even after refinement may still be suitable, sometimes not. Only a careful design will eliminate some of the problems we've discussed earlier.

Entity Sets vs. Attributes

- Consider the entity set: *Employee(emp-name, telephone-number, age)*
- It could easily be argued that a telephone is an entity in its own right with attributes of say, *telephone-number*, *location*, *manufacturer*, *serial-num*, and so on. If we take this point of view, then:

1. The *Employee* entity set must be redefined as:

   Employee (emp-name, age)

2. Must create a new entity set:

   Telephone(telephone-number, location, manufacturer, serial-num,…)

3. A relationship set must be created to denote the association between employees and the telephones that they have.

   Emp-Phone(emp-name, telephone-number, age, location, manufacturer, serial-num)

Entity Sets vs. Attributes (cont.)

- Now we must consider what it the main difference between these two definitions of an employee?
- Treating the telephone as an attribute *telephone-number* implies that employees have precisely one telephone number each. (Note that this must be true or otherwise the telephone-number attribute would need to be a part of the key for an employee and it isn't here – not considering multiple-valued attributes).
- Treating a telephone as an entity permits employees to have several phones (including zero) associated with them. However, we could easily make the *telephone-number* attribute be a multi-valued one to allow multiple phones per employee. So clearly, this is **not** the main difference in the two representations.

Entity Sets vs. Attributes (cont.)

- The main difference then is that treating a telephone as an entity better models a situation where one might want to keep additional information about a telephone, as we have indicated with our example above.

- If we used the original approach and wished to make the telephone an attribute of an employee and we wished to maintain this additional information about their phone, then the *Employee* entity set would look like:

  Employee(emp-name, telephone-number, age, location, manufacturer,...)

- This is clearly not a good schema, for example, is the *age* attribute associated with the employee or the telephone? In this situation we are attempting to model two different entity sets inside a single entity set.

Entity Sets vs. Attributes (cont.)

- Conversely, it would not be appropriate to treat the attribute *emp-name* as an entity; it is difficult to argue that an employee name is an entity in its own right (in contrast to the telephone). Thus, it is entirely appropriate to have *emp-name* as an attribute of the *Employee* entity set.

- So, what constitutes and attribute and what constitutes an entity?

  – Unfortunately, there are no simple answers. The distinctions depend mainly upon the structure of the real-world scenario which is being modeled, and on the semantics associated with the attribute in question.

Entity Sets vs. Attributes (cont.)

- A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For example, given our bank example again, it would not be appropriate to model *customer-id* as an attribute of *loan* even if each loan had only one customer associated to it. The relationship *borrower* is the correct way of representing the relationship between a loan and a customer, since it makes their connection explicit rather than implicit via an attribute.

Associative Entities

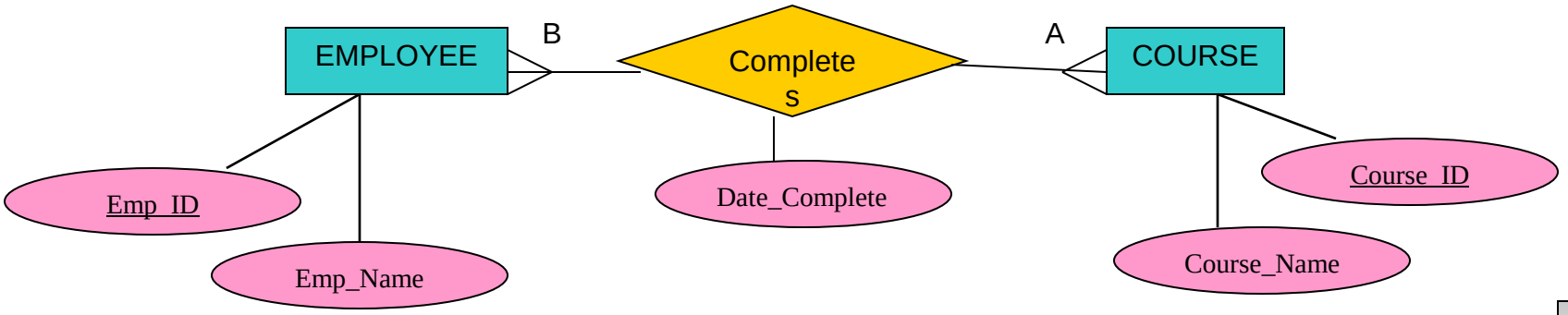
- The presence of one or more attributes on a relationship suggests to the designer that the relationship should perhaps instead be represented as an entity type.
- As associative entity is an entity type that associates the instances of one or more entity types and contains attributes that are peculiar to the relationship between those entity instances.
- For example, (see ER diagram on next page) consider an organization that wishes to record the date (month and year) when an employee completes each certification course. The date completed cannot be associated with either entity sets EMPLOYEE or COURSE, because Date_Completed is a property of the relationship Completes.

Associative Entities (cont.)

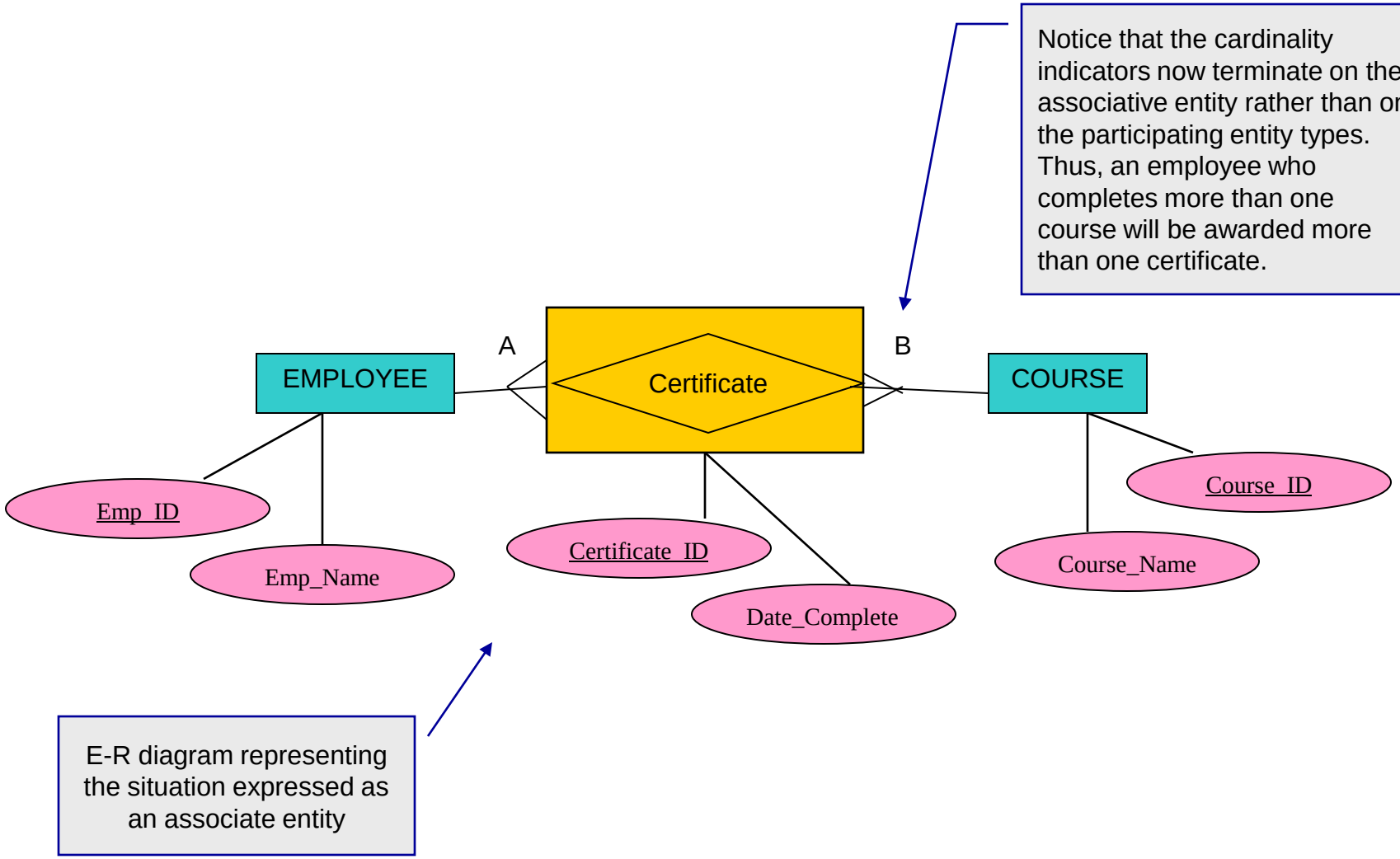
Employee_Name	Course_Title	Date_Completed
Kristi	C++	6/2009
Kristi	Java	12/2008
Debi	SQL	11/2009
Angela	SQL	10/2009
Angela	Perl	1/2010

Some sample data

E-R diagram representing the situation



Associative Entities (cont.)



Associative Entities (cont.)

- How do you know whether to convert a relationship into an associative entity type?
- There are four conditions that should exist:

1. All of the relationships for the participating entity types are "many" relationships.
2. The resulting associative entity type has independent meaning to end users, and preferably can be identified with a single-attribute identifier.
3. The associative entity has one or more attributes, in addition to the identifier.
4. The associative entity participates in one or more relationships independent of the entities related in the associated relationship.

Entity Sets vs. Relationship Sets

- It is not always clear whether an object is best expressed by an entity set or a relationship set.
- Consider the banking example. We have been modeling a loan as an entity. An alternative is to model a loan as a relationship between customers and say branches of the bank, with *loan-number* and *amount* as descriptive attributes. Each loan is then represented as a relationship between a customer and a branch.

Entity Sets vs. Relationship Sets (cont.)

- If every loan is owned by exactly one customer and is associated with exactly one branch, then it may be satisfactory to model the loan as a relationship.
- However, with this design we cannot represent in a convenient way the situation in which several customers jointly own a single loan.
  - To handle this type of situation, we would need to define a separate relationship for each holder of the joint loan.
  - Then we would replicate all of the values for the descriptive attributes *loan-number* and *amount* in each such relationship. Each such relationship must, of course, have the same value for the descriptive attributes.

Entity Sets vs. Relationship Sets (cont.)

- Two problems arise as a result of the replication:

  1. The data are stored in multiple locations (the very meaning of replication).

  2. Updates potentially leave the data in an inconsistent state, where the values in two different sets differ when they should be identical. We'll look at the complications that this replication causes as well as solution techniques (normalization theory) later in the course. Notice that the problem of replication is absent in our original version because *loan* is represented by an entity set in that case.

- One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

Weak Entity Sets vs. Strong Entity Sets

- An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a *weak entity set*. An entity set that has a primary key is termed a *strong entity set*.
  - As an example, consider an entity set *payment*, which has three attributes: *payment-number*, *payment date*, and *payment amount*. Payment numbers are typically just sequential numbers, starting at 1 and are generated separately for each loan. Thus, although each *payment* entity is distinct, payments for different loans may share the same payment number, thus the set does not have a primary key and is a weak entity set.

Weak Entity Sets vs. Strong Entity Sets (cont.)

- For a weak entity set to be meaningful, it must be associated with another entity set which is called the *identifying* or *owner entity set*. Every weak entity must be associated with such an identifying entity set.
- The weak entity is said to be *existence dependent* on the identifying set. The identifying set is said to *own* the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the *identifying relationship*. The identifying relationship is many-to-one from the weak entity set to the identifying set and the participation of the weak entity set in the relationship is total.

Weak Entity Sets vs. Strong Entity Sets (cont.)

- Although a weak entity set does not have a primary key, we nevertheless need a means of distinguishing among all those entities in the weak entity set that depend upon one particular strong entity.

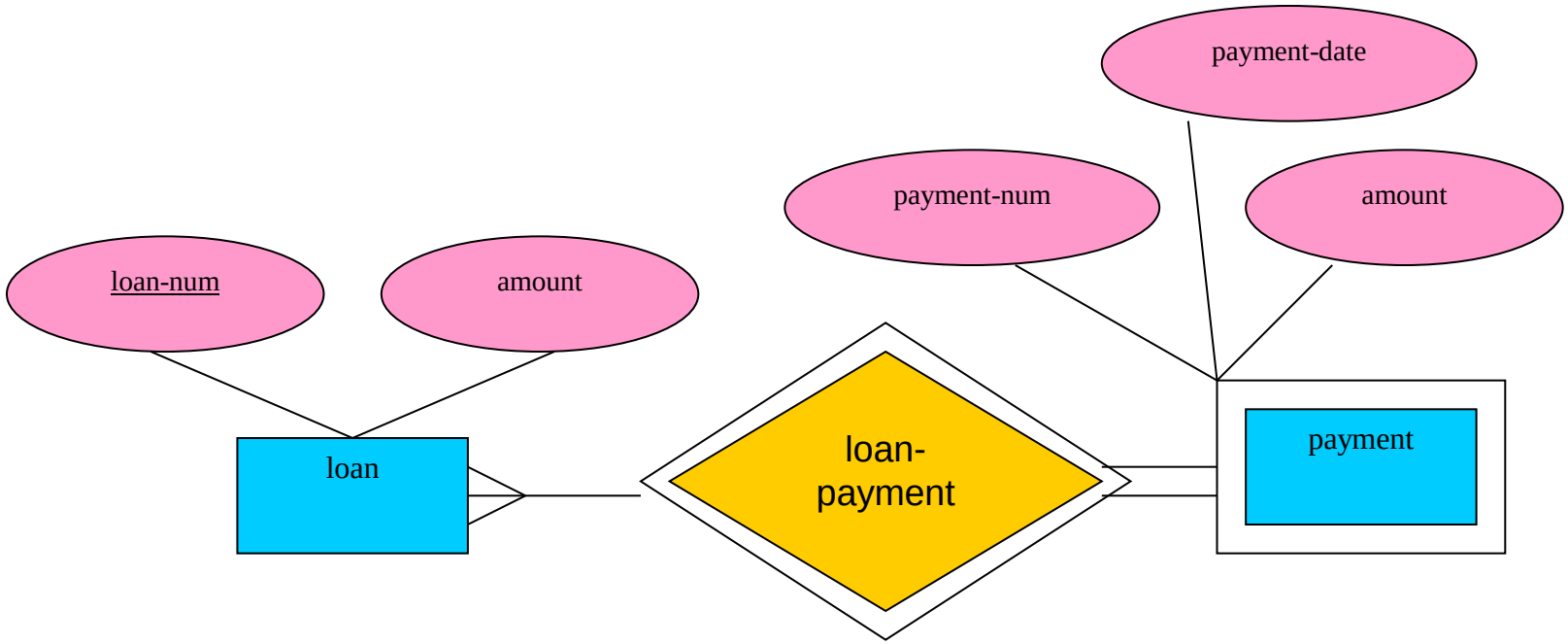
- The set of attributes of a weak entity that allows this distinction to be made is called the *discriminator* (sometimes also called the *partial key*). For example, the discriminator of the weak entity set *payment* from above is the attribute *payment-number*, since for each loan, a payment number uniquely identifies one single payment for that loan.

Weak Entity Sets vs. Strong Entity Sets (cont.)

- The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.
  - For the case above, the primary key of the entity set *payment* would be: {*loan-number*, *payment-number*}, where *loan-number* would be the primary key of the identifying entity set *loan* and *payment-number* is the discriminator of the weak entity set *payment*.
- Within the E-R diagram, a weak entity set is represented by a rectangle with double lines and the identifying relationship for a weak entity set is represented by a diamond with double lines.

Weak Entity Sets vs. Strong Entity Sets (cont.)

Example of a weak entity set.



Extensions of the E-R Model

- Some features of a real world situation can be difficult to model using only the features of the E-R model that we have seen so far.
- Some quite common concepts require extending the E-R model to incorporate mechanisms for modeling these features. Again, we won't look at all of them, but rather an overview of some of the more important extensions.

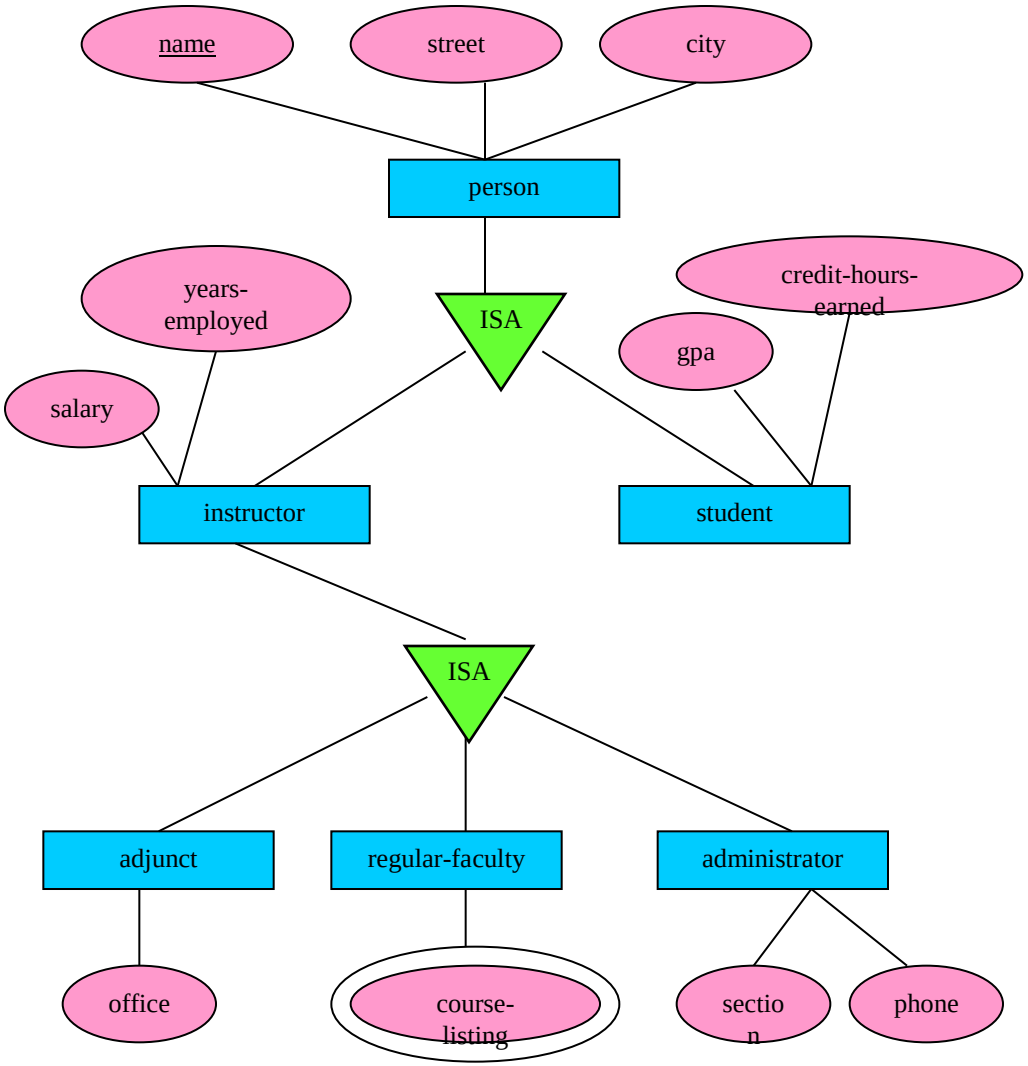
Specialization

- An entity set may include sub-groupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the set.
  - As an example, consider the entity set *person*, with attributes *name*, *street*, and *city*. A person could further be classified as one of the following: *student* or *instructor*. Each of these person types is described by a set of attributes that includes all of the attributes of the entity set *person*, plus possibly some additional attributes. For example, *student* entities may be further described by the attributes *gpa*, and *credit-hours-earned*, whereas, *instructor* entities are not characterized by these attributes, but rather a different set such as, *salary*, and *years-employed*.
- The process of designating sub-groupings within an entity set is called *specialization*.

Specialization (cont.)

- The specialization of *person* allows us to distinguish among persons according to whether they are students or instructors.
- Specialization can be repeatedly applied so that there may be specializations within specializations.
- In terms of an E-R diagram, specialization is depicted by a triangle shaped component which is labeled ISA, which is a shorthand form of the "is-a" superclass-subclass relationship.
- The ISA relationship is illustrated in the diagram in the next slide.

Specialization (cont.)



Generalization

- The refinement from an initial entity set into successive levels of entity sub-groupings represents a top-down design approach in which distinctions are made explicit.

- This same design process could also proceed in a bottom-up approach, in which multiple entity sets are synthesized into a higher-level entity on the basis of common attributes. In other words, we might have first identified the entity set *students(name, address, city, gpa, credit-hours-earned)* and an entity set *instructors(name, address, city, salary, years-employed).*

- This commonality of attributes is expressed by *generalization*, which is a containment relationship that exists between a higher-level entity set and one or more lower level entity sets.

Generalization (cont.)

- In our example, *person* is the higher-level entity set and *instructor* and *student* are the lower-level entity sets.
- The higher-level entity set represents the *superclass* and the lower-level entity represents the *subclass*. Thus, *person* is the superclass of the *instructor* and *student* subclasses.
- For all practical purposes, generalization is just the inverse of specialization and both processes can be applied (almost interchangeably) in designing the schema for some real-world scenario. Notice in the E-R diagram on page 70 that there is no difference specified between generalization and specialization other that how you view the picture (reading from the top down or from the bottom up).

Specialization vs. Generalization

- Differences in the two approaches are normally characterized by their starting points and overall goal:
- Specialization arises from a single entity set; it emphasizes differences among the entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes or participate in relationships, that do not apply to all the entities in the higher-level entity set.
- In fact, the reason that a designer may need to use specialization is to represent such distinctive features of the real world scenario.
  - For example, if *instructor* and *student* neither have attributes that *person* entities do not have nor participate in relationships different than those in which *person* entities participate, there would be no need to specialize the *person* entity set.

Specialization vs. Generalization (cont.)

- Generalization arises from the recognition that a number of entity sets share some common characteristics (namely, they are described by the same attributes and participate in the same relationship sets).
- On the basis of these commonalities, generalization synthesizes these entity sets into a single, higher-level entity set.
- Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences. It also permits an economy of representation in that the shared attributes are not replicated.

Attribute Inheritance

- A crucial property of the higher and lower level entities that are created by specialization and generalization is *attribute inheritance.*
- The attributes of the higher-level entity sets are said to be *inherited* by the lower-level entity sets.
  - In our example above, *instructor* and *student* both inherit all the attributes of *person* (recall that *person* is the superclass for both *instructor* and *student*).
- A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity set (its superclass) participates.
- A lower-level entity (subclass) inherits all attributes and relationships which belong to the higher-level entity set (superclass) which defines it.

Attribute Inheritance (cont.)

- Higher-level entity sets do not inherit any attribute or relationship which is defined within the lower-level entity set.
- Typically, what is developed will be a *hierarchy* of entity sets in which the highest-level entity appears at the top of the hierarchy.
- If, in such a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship, then the inheritance is said to be *single-inheritance*.
- If, on the other hand, a given entity set is involved as a lower-level entity set in more than one ISA relationship, then the inheritance is said to be *multiple-inheritance* (then the resulting structure is called a *lattice*).

Constraints on Generalization

- In order to more accurately model a real-world situation, a data designer may choose to place constraints on a generalization (or specialization).

- The first type of constraint involves determining which entities can be members of a given lower-level entity set. This membership can be defined in one of the following two ways:

Predicate-defined: In predicate-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit predicate (a condition).

  – For example, assume that the higher-level entity set *account* has the attribute *account-type*. All account entities are evaluated on the defining *account-type* attribute. Only those entities that satisfy the predicate *account-type = "savings account"* would be allowed to belong to the lower-level entity set *savings-account*. Since all the lower-level entities are evaluated on the basis of the same attribute, this type of generalization is said to be *attribute-defined*.

Constraints on Generalization (cont.)

User-defined: User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set.

- For instance, suppose that after working 3 months at a bank, the employee is assigned to one of five different work groups. The teams would be represented as five lower-level entity sets of the higher-level entity set *employee*. A given employee is not assigned to a specific work group automatically on the basis of an explicit defining condition. Instead, the user responsible for making the group assignment does so on an individual basis, which may be arbitrary.

Constraints on Generalization (cont.)

- A second type of generalization constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower-level entity sets may be one of the following:

Disjoint: A disjointness constraint requires that an entity belong to no more than one lower-level entity set. In the example from above, an *account* entity can satisfy only one condition for the *account-type* attribute at any given time.

  – For example, an account-type might be either a checking account or a savings account, but it cannot be both.

Constraints on Generalization (cont.)

Overlapping: In *overlapping generalizations*, the same entity may belong to more than one lower-level entity set within a single generalization. For example, consider the banking work group from the previous section. Suppose that certain managers may participate in more than one work team. A given employee (a manager) may therefore appear in more than one of the group entity sets that are lower-level entity sets of *employee*.

- Note: lower-level entity overlap is the default case; a disjointness constraint must be placed explicitly on a generalization (or specialization). Within the E-R model a disjointness constraint is modeled by placing the word "disjoint" next to the triangle symbol as shown in the example below. The meaning of this diagram should now be clear: employees and customers are specializations of the set persons and the disjointness constraint implies that an employee is not also a customer. If the disjoint constraint is removed, then it is possible for an employee to also be a customer (or viewed from the other direction, it is possible for a person to be both a customer as well as an employee).

Constraints on Generalization (cont.)

- A final type of constraint, the *completeness constraint* on a generalization or specialization, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This type of constraint can assume one of the following two forms:

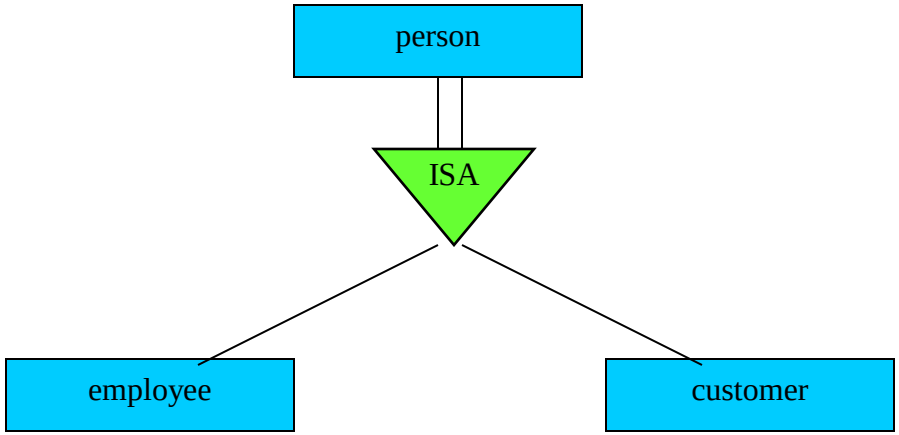
Total generalization/specialization: Each higher-level entity must belong to a lower-level entity.

Partial generalization/specialization: Some higher-level entities may not belong to any lower-level entity set.

  - Partial generalization is the default case. (Recall that total participation in a relationship is represented in the E-R model by a double line – so too will it be used to represent a total generalization. In the example shown below the generalization is total and overlapping which means that every person must appear as either an employee or a customer and it is possible for a person to be both.

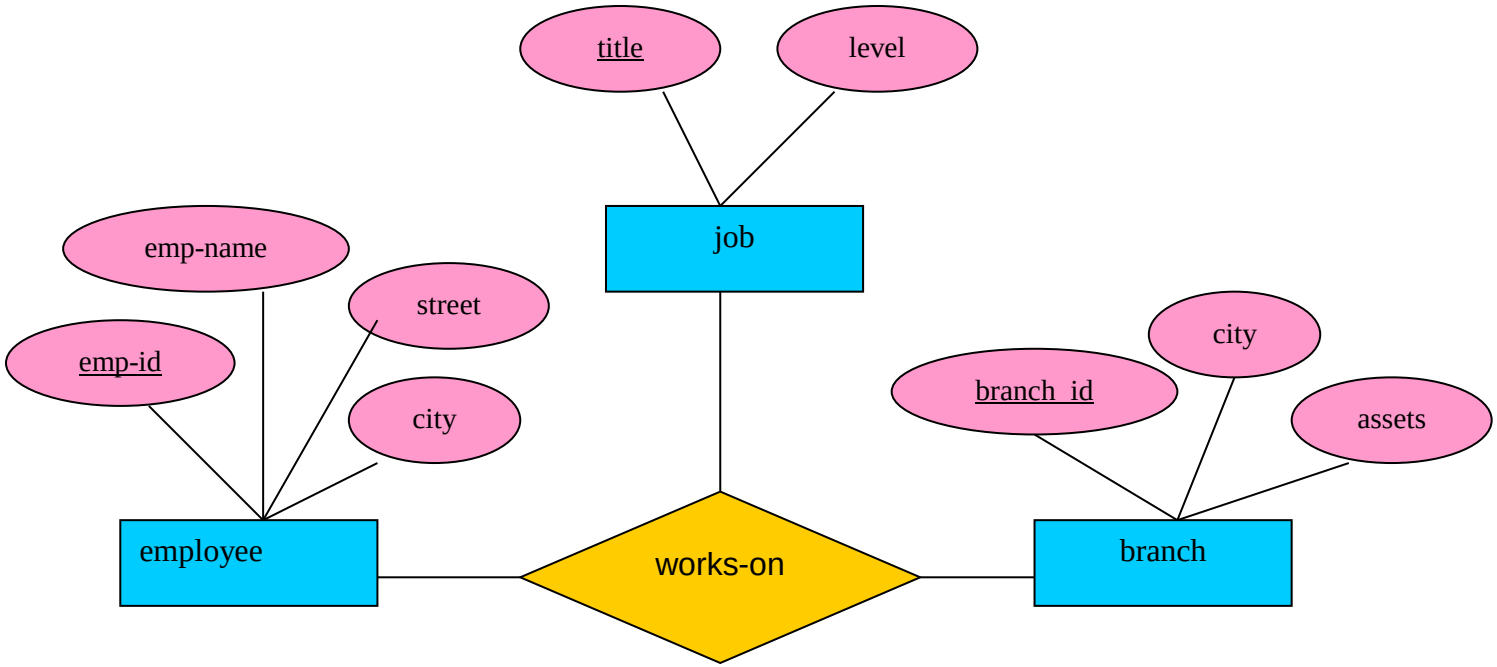
Example ERDs with Constraints

A total overlapping generalization/specialization



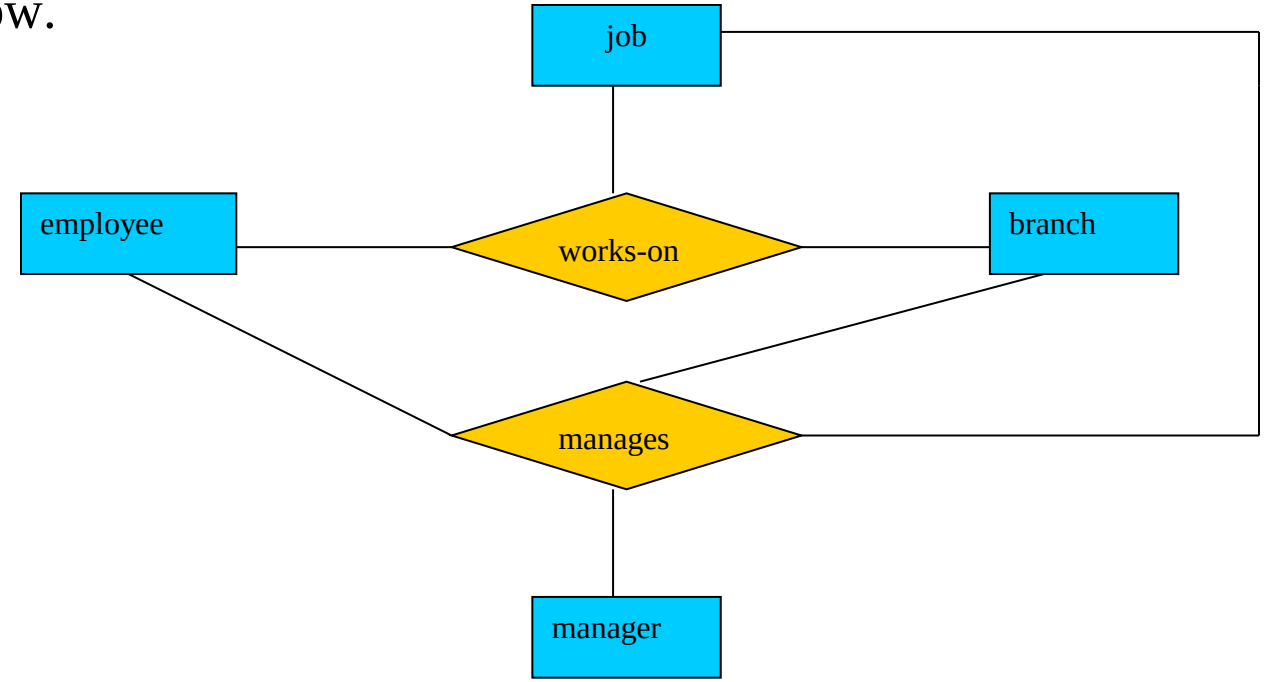
Aggregation

- One of the limitations of the E-R model is that it cannot express relationships among relationships. To understand why this is important consider the ternary relationship (3-way relationship) *works-on* between *employee*, *branch*, and *job* shown in the following E-R diagram.



Aggregation (cont.)

- Given this scenario, now suppose that we want to record the managers for tasks performed by an employee at a branch office; that is, we want to keep track of managers for *(employee*, *branch*, *job)* combinations. Let's assume that there is an entity set *manager*.
- One way to handle this is to create a quaternary relationship as shown below.



Aggregation (cont.)

Question: Why wouldn't a binary relationship between *manager* and *employee* work?

Answer:

A binary relationship would not permit us to represent which (*branch*, *job*) combinations of an employee are managed by which manager.

Aggregation (cont.)

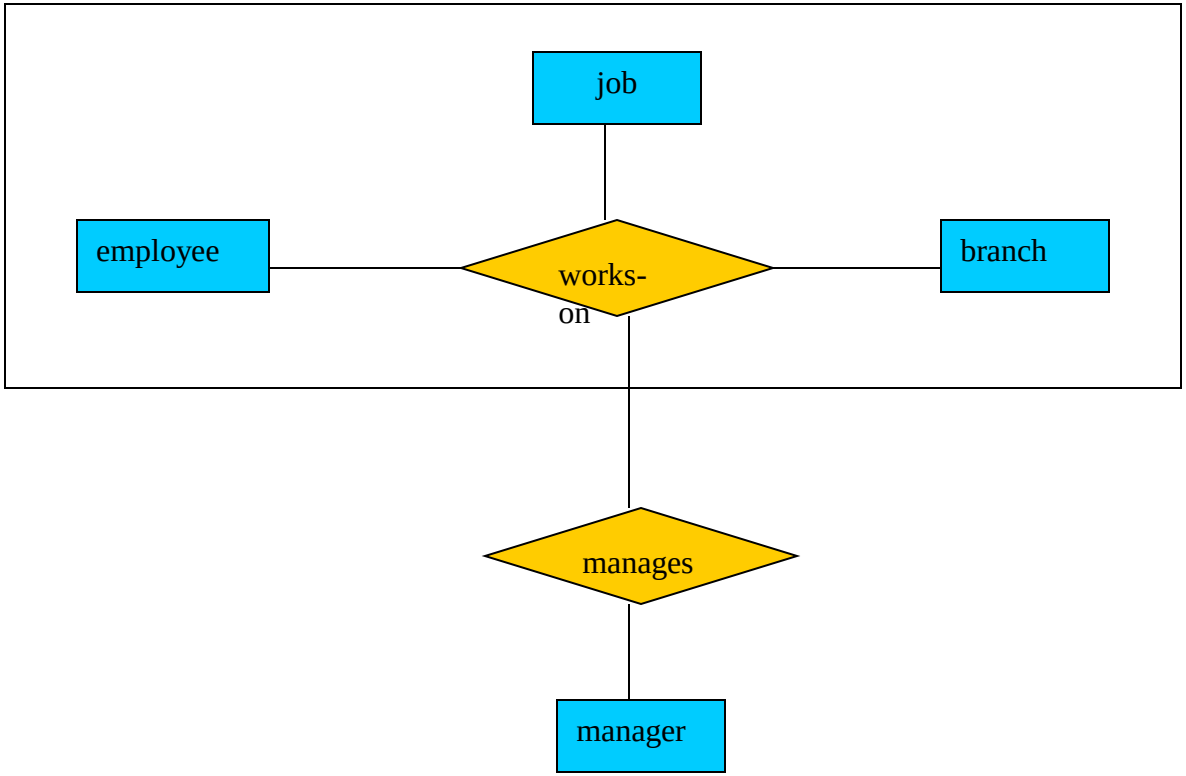
- When you look at the E-R diagram which models this situation, it would appear that the relationships sets *works-on* and *manages* could be combined into a single relationship set. However, we cannot do this since some *employee*, *branch*, *job* combinations may not have a manager.

- There is clearly redundant information in this figure, however, since every *employee*, *branch*, *job* combination in *manages* is also in *works-on*. If the manager were a value rather than an entity, we could make *manager* a multi-valued attribute of the relationship *works-on*. However, doing this would make it more difficult (both logically as well as in execution cost) to find, for example, employee-branch-job triples for which the manager is responsible. However, this option is not available in any case since the manager is a *manager* entity.

Aggregation (cont.)

- The best way to model this type of situation is to use *aggregation*.
- Aggregation is an abstraction through which relationships are treated as higher-level entities.
- Thus, in our example, we would regard the relationship set *works-on* (relating the entity sets *employee*, *branch*, and *job*) as a higher-level entity set called *works-on*. Such an entity set is treated in the same manner as any other entity set. We can then create a binary relationship *manages* between *works-on* and *manager* to represent who manages what tasks.
- The E-R diagram in the next slide illustrates how aggregation is represented in the E-R model.

Aggregation (cont.)

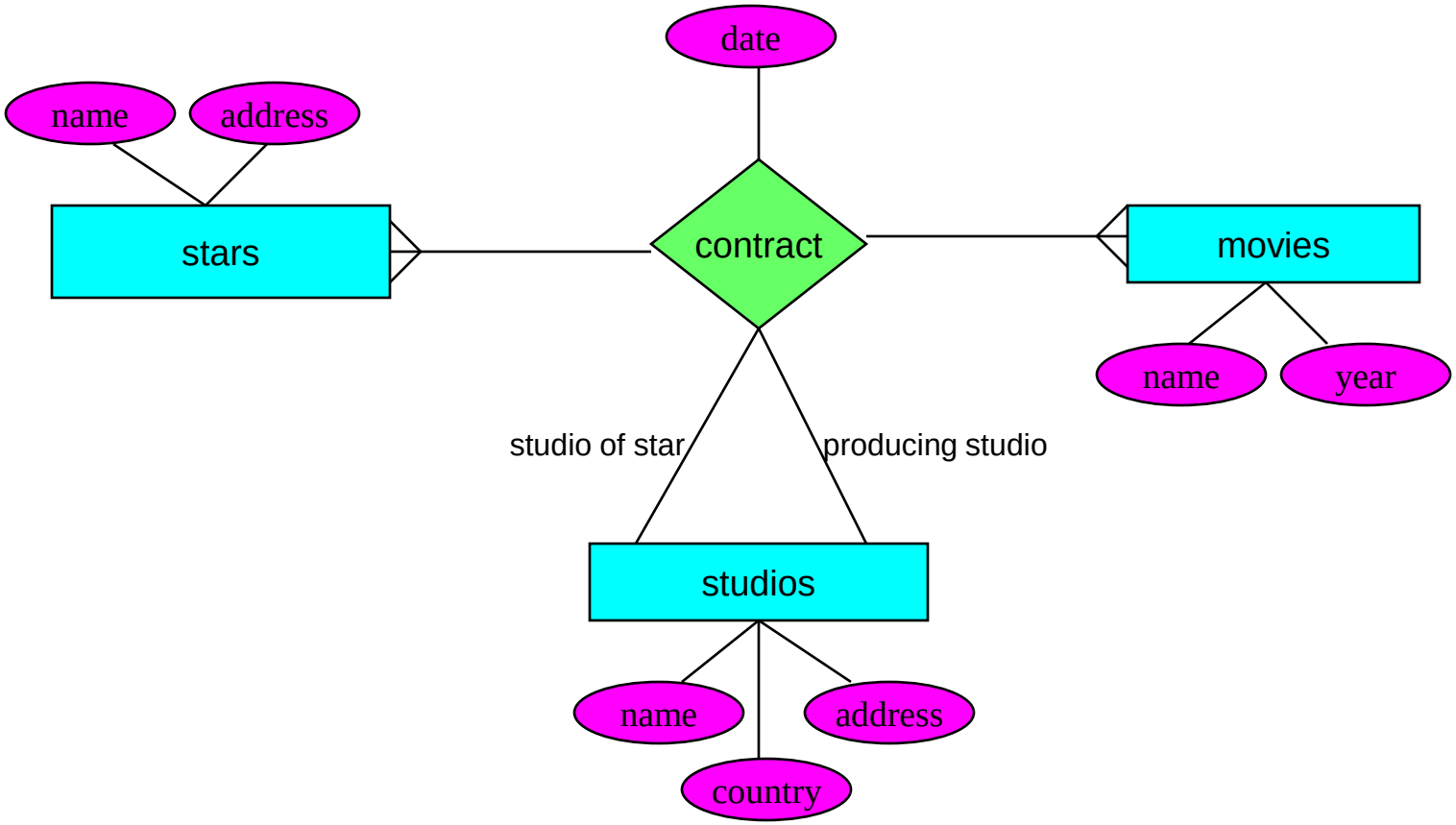


ERD illustrating aggregation

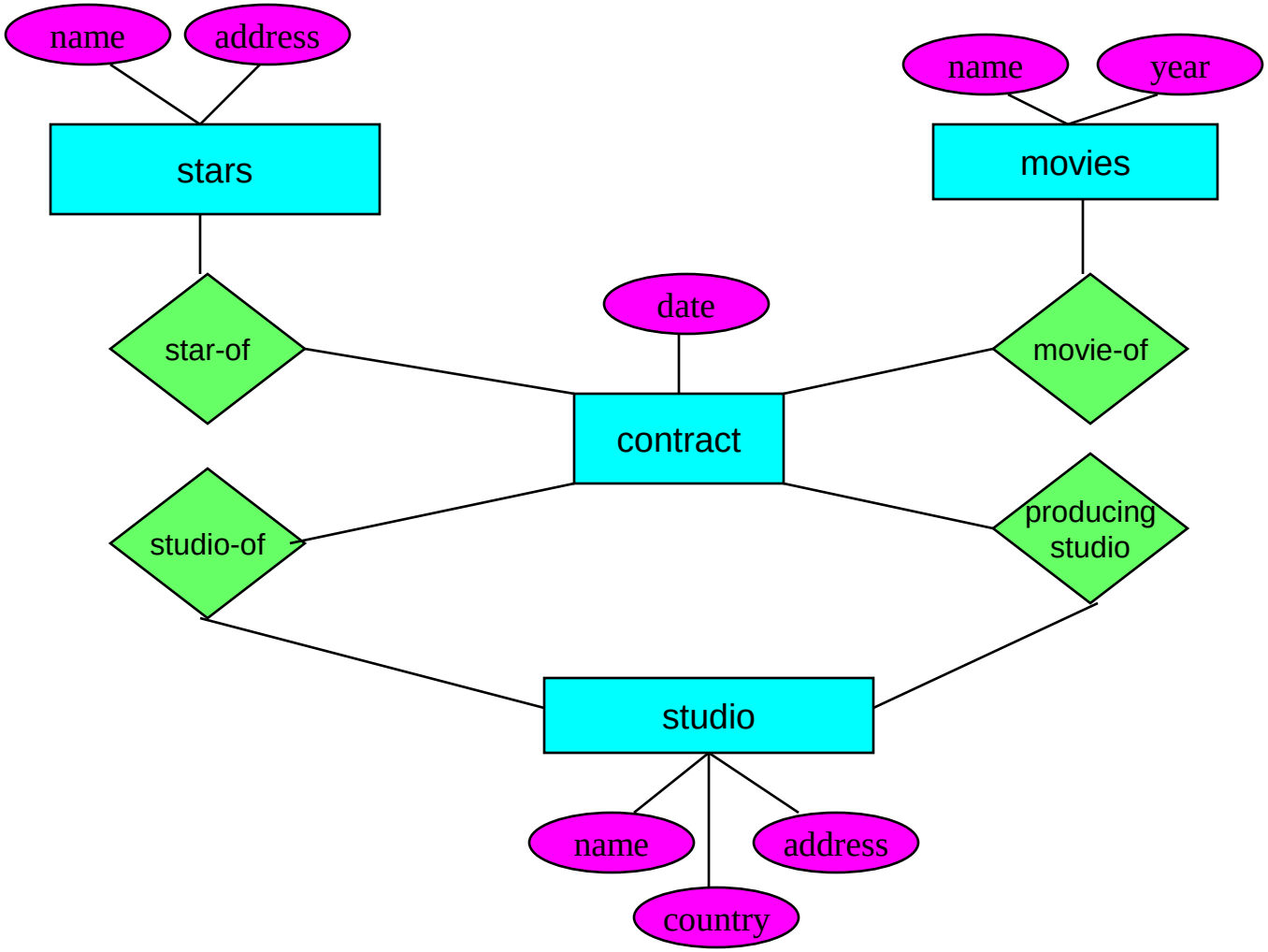
Multiway Relationships

- Most of the relationships that we have examined so far have been binary relationships, i.e., those relationships involving two entity sets.
- Any relationship involving more than two entity sets can be converted to a collection of binary, many-to-one relationships.
  - This is useful because, while the E-R model does not limit relationships to binary, many data models do, such as the Object Definition Language.
- To illustrate the conversion of a multiway relationship into a collection of binary relationships, consider the example E-R diagram on the next page.

Multiway Relationships (cont.)

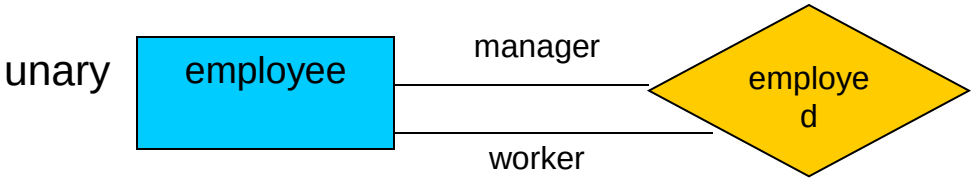


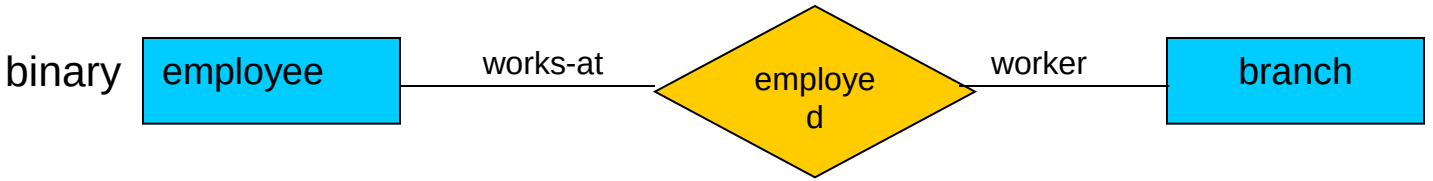
Multiway Relationship Converted to a Collection of Binary Relationships



E-R Diagrams with Role Indicators

- Roles in an E-R diagram are indicated by labeling the lines that connect entity sets to relationship sets.
- Roles can be identified for unary (recursive), binary, and nonbinary relationships.



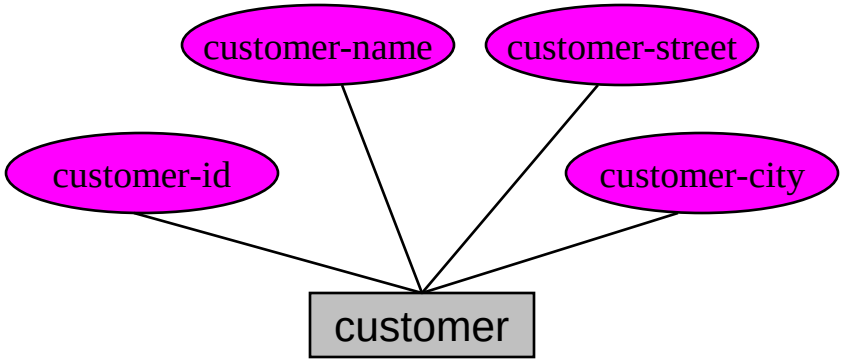


The Unified Modeling Language (UML) (cont.)

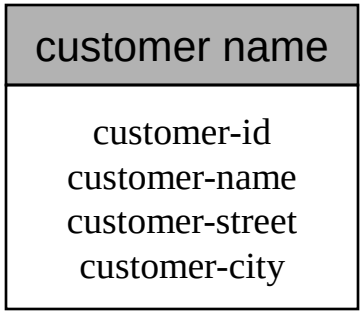
- Some of the parts of UML are:
  1. Class diagram. A class diagram is similar to an E-R diagram. We'll see the correspondence between them shortly.
  2. Use case diagram. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money from a bank account or registering for a course).
  3. Activity diagram. Activity diagrams depict the flow of tasks between various components of the system.
  4. Implementation diagram. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

Correspondence of E-R & UML Class Diagrams

Entity sets and attributes



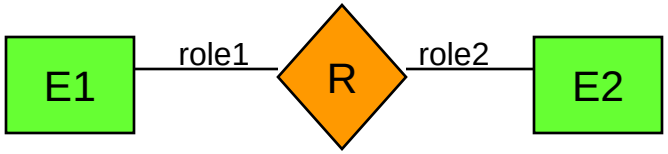
E-R Diagram

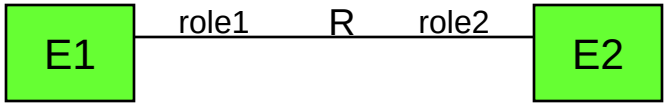


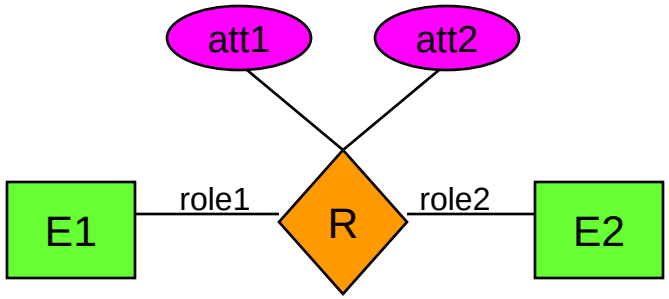
UML Class Diagram

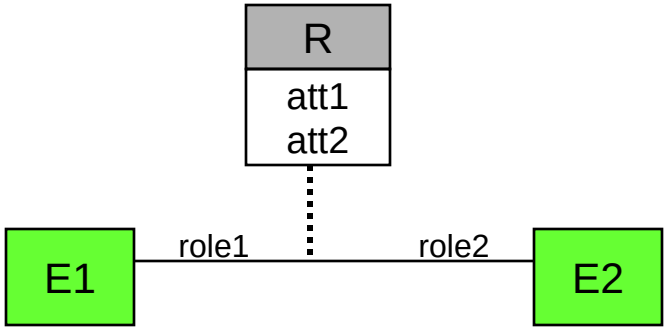
Correspondence of E-R & UML Class Diagrams (cont.)

Relationships







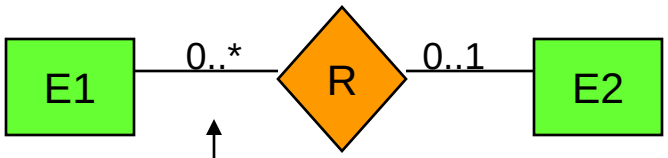


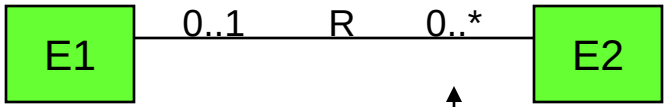
E-R Diagrams

UML Class Diagrams

Correspondence of E-R & UML Diagrams (cont.)

Cardinality Constraints





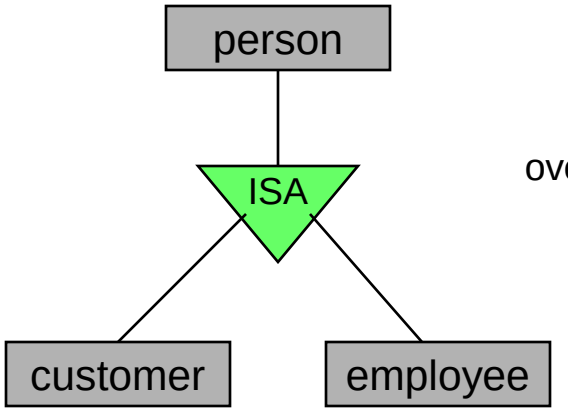
NOTE: Positioning of cardinality constraints is exactly opposite in the two models. In the UML model the constraint 0..1 on the left side means that an E2 entity can participate in at most 1 relationship, whereas each E1 entity can participate in many relationships; in other words, the relationship is many to one from E2 to E1

E-R Diagrams

UML Diagrams

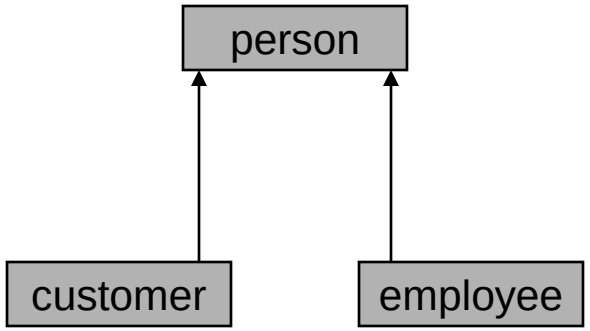
Correspondence of E-R & UML Class Diagrams (cont.)

Generalization & Specialization



E-R Diagrams

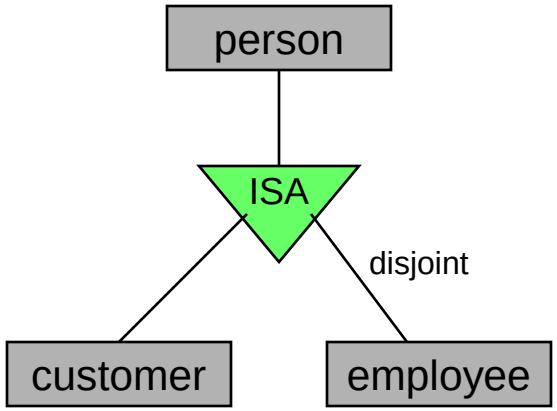
overlapping generalization



UML Class Diagrams

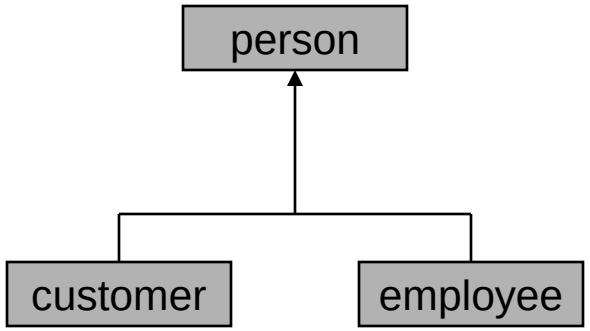
Correspondence of E-R & UML Class Diagrams (cont.)

Generalization & Specialization



E-R Diagrams

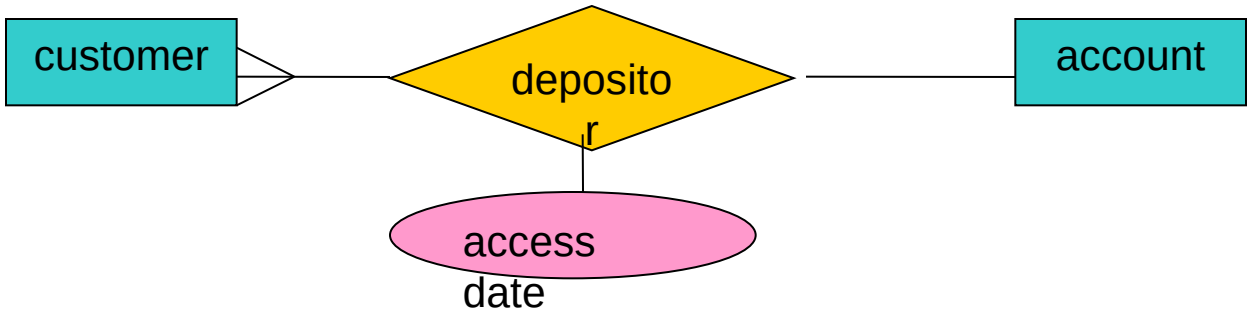
disjoint generalization



UML Class Diagrams

Referential Integrity Constraints

- Referential integrity constraints can be as simple as asserting that a given attribute have a non-null, single value. However, referential integrity constraints most commonly refer to the relationships among entity sets.
- Let's again consider our banking example and the many-to-one relationship between customer and account as shown below:



Referential Integrity Constraints (cont.)

- The many-to-one relationship depositor simply says that no account can be deposited into by more than one customer (and also that a customer can deposit into many different accounts).
- More importantly, it does **not** say that an account must be deposited into by a customer, nor does it say that a customer must make a deposit into an account. Further, it does not say that if an account is deposited into by a customer that the customer be present in the database!
- A referential integrity constraint requires that each entity "referenced" by the relationship must exist in the database.
- There are several methods which can be used to enforce referential integrity constraints:

Referential Integrity Constraints (cont.)

1. Deletion of a referenced entity is not allowed. In other words, if Kristi makes a deposit into account number 456, then subsequently we cannot delete either the information concerning either Kristi or account 456.

2. If a referenced entity is deleted, then all entries that reference the deleted entity also be deleted. In other words, if we delete the information on Kristi, then we must delete all account information for accounts that she (alone) has deposited into. Notice in the specific example we are considering, that the relationship is M:1 which means that if Kristi has deposited into an account, she will be the only customer to do so. This will not be the case for a M:M relationship however.

- Referential integrity constraints can be modeled in the E-R model. Typically, they are depicted with a curved arrow as shown on the next page.

Referential Integrity Constraints (cont.)