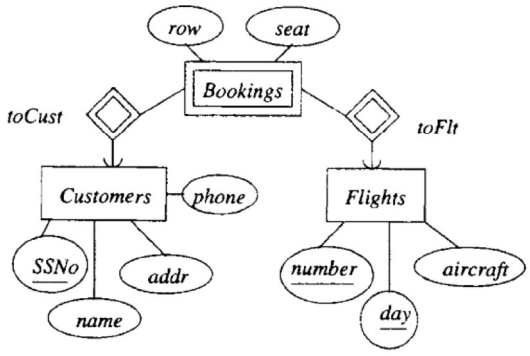

Figure 4.29: An E/R diagram about airlines

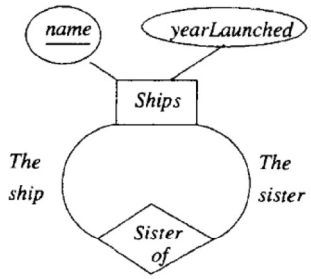

Figure 4.30: An E/R diagram about sister ships

4.6 Converting Subclass Structures to Relations

When we have an isa-hierarchy of entity sets, we are presented with several choices of strategy for conversion to relations. Recall we assume that:

- There is a root entity set for the hierarchy,
- This entity set has a key that serves to identify every entity represented by the hierarchy, and
- A given entity may have *components* that belong to the entity sets of any subtree of the hierarchy, as long as that subtree includes the root.

The principal conversion strategies are:

1. *Follow the E/R viewpoint.* For each entity set E in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to E.

2. *Treat entities as objects belonging to a single class.* For each possible subtree that includes the root, create one relation, whose schema includes all the attributes of all the entity sets in the subtree.

3. *Use null values.* Create one relation with all the attributes of all the entity sets in the hierarchy. Each entity is represented by one tuple, and that tuple has a null value for whatever attributes the entity does not have.

We shall consider each approach in turn.

4.6.1 E/R-Style Conversion

Our first approach is to create a relation for each entity set, as usual. If the entity set E is not the root of the hierarchy, then the relation for E will include the key attributes at the root, to identify the entity represented by each tuple, plus all the attributes of E. In addition, if E is involved in a relationship, then we use these key attributes to identify entities of E in the relation corresponding to that relationship.

Note, however, that although we spoke of "isa" as a relationship, it is unlike other relationships, in that it connects components of a single entity, not distinct entities. Thus, we do not create a relation for "isa."

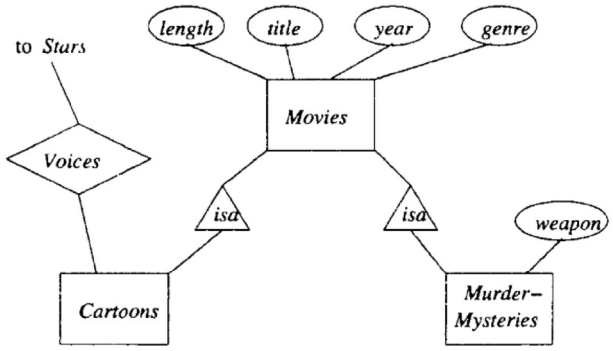

Figure 4.31: The movie hierarchy

Example 4.31: Consider the hierarchy of Fig. 4.10, which we reproduce here as Fig. 4.31. The relations needed to represent the entity sets in this hierarchy are:

1. `Movies(title, year, length, genre)`. This relation was discussed in Example 4.24, and every movie is represented by a tuple here.

2. `MurderMysteries(title, year, weapon)`. The first two attributes are the key for all movies, and the last is the lone attribute for the corresponding entity set. Those movies that are murder mysteries have a tuple here as well as in `Movies`.

3. `Cartoons(title, year)`. This relation is the set of cartoons. It has no attributes other than the key for movies, since the extra information about cartoons is contained in the relationship *Voices*. Movies that are cartoons have a tuple here as well as in `Movies`.

Note that the fourth kind of movie — those that are both cartoons and murder mysteries — have tuples in all three relations.

In addition, we shall need the relation `Voices(title, year, starName)` that corresponds to the relationship *Voices* between *Stars* and *Cartoons*. The last attribute is the key for *Stars* and the first two form the key for *Cartoons*.

For instance, the movie *Roger Rabbit* would have tuples in all four relations. Its basic information would be in `Movies`, the murder weapon would appear in `MurderMysteries`, and the stars that provided voices for the movie would appear in `Voices`.

Notice that the relation `Cartoons` has a schema that is a subset of the schema for the relation `Voices`. In many situations, we would be content to eliminate a relation such as `Cartoons`, since it appears not to contain any information beyond what is in `Voices`. However, there may be silent cartoons in our database. Those cartoons would have no voices, and we would therefore lose information should we eliminate relation `Cartoons`. □

4.6.2 An Object-Oriented Approach

An alternative strategy for converting isa-hierarchies to relations is to enumerate all the possible subtrees of the hierarchy. For each, create one relation that represents entities having components in exactly those subtrees. The schema for this relation has all the attributes of any entity set in the subtree. We refer to this approach as "object-oriented," since it is motivated by the assumption that entities are "objects" that belong to one and only one class.

Example 4.32: Consider the hierarchy of Fig. 4.31. There are four possible subtrees including the root:

1. *Movies* alone.
2. *Movies* and *Cartoons* only.
3. *Movies* and *Murder-Mysteries* only.
4. All three entity sets.

We must construct relations for all four "classes." Since only Murder-Mysteries contributes an attribute that is unique to its entities, there is actually some repetition, and these four relations are:

```
Movies(title, year, length, genre)
MoviesC(title, year, length, genre)
MoviesMM(title, year, length, genre, weapon)
MoviesCMM(title, year, length, genre, weapon)
```

If *Cartoons* had attributes unique to that entity set, then all four relations would have different sets of attributes. As that is not the case here, we could combine `Movies` with `MoviesC` (i.e., create one relation for non-murder-mysteries) and combine `MoviesMM` with `MoviesCMM` (i.e., create one relation for all murder mysteries), although doing so loses some information — which movies are cartoons.

We also need to consider how to handle the relationship *Voices* from *Cartoons* to *Stars*. If *Voices* were many-one from *Cartoons*, then we could add a `voice` attribute to `MoviesC` and `MoviesCMM`, which would represent the *Voices* relationship and would have the side-effect of making all four relations different. However, *Voices* is many-many, so we need to create a separate relation for this relationship. As always, its schema has the key attributes from the entity sets connected; in this case

```
Voices(title, year, starName)
```

would be an appropriate schema.

One might consider whether it was necessary to create two such relations, one connecting cartoons that are not murder mysteries to their voices, and the other for cartoons that *are* murder mysteries. However, there does not appear to be any benefit to doing so in this case. □

4.6.3 Using Null Values to Combine Relations

There is one more approach to representing information about a hierarchy of entity sets. If we are allowed to use `NULL` (the null value as in SQL) as a value in tuples, we can handle a hierarchy of entity sets with a single relation. This relation has all the attributes belonging to any entity set of the hierarchy. An entity is then represented by a single tuple. This tuple has `NULL` in each attribute that is not defined for that entity.

Example 4.33: If we applied this approach to the diagram of Fig. 4.31, we would create a single relation whose schema is:

```
Movie(title, year, length, genre, weapon)
```

Those movies that are not murder mysteries would have `NULL` in the `weapon` component of their tuple. It would also be necessary to have a relation `Voices` to connect those movies that are cartoons to the stars performing the voices. as in Example 4.32. □

4.6.4 Comparison of Approaches

Each of the three approaches, which we shall refer to as "straight-E/R," "object-oriented," and "nulls," respectively, have advantages and disadvantages. Here is a list of the principal issues.

1. It can be expensive to answer queries involving several relations, so we would prefer to find all the attributes we needed to answer a query in one relation. The nulls approach uses only one relation for all the attributes, so it has an advantage in this regard. The other two approaches have advantages for different kinds of queries. For instance:

   (a) A query like "what films of 2008 were longer than 150 minutes?" can be answered directly from the relation `Movies` in the straight-E/R approach of Example 4.31. However, in the object-oriented approach of Example 4.32, we need to examine `Movies`, `MoviesC`, `MoviesMM`, and `MoviesCMM`, since a long movie may be in any of these four relations.

   (b) On the other hand, a query like "what weapons were used in cartoons of over 150 minutes in length?" gives us trouble in the straight-E/R approach. We must access `Movies` to find those movies of over 150 minutes. We must access `Cartoons` to verify that a movie is a cartoon, and we must access `MurderMysteries` to find the murder weapon. In the object-oriented approach, we have only to access the relation `MoviesCMM`, where all the information we need will be found.

2. We would like not to use too many relations. Here again, the nulls method shines, since it requires only one relation. However, there is a difference between the other two methods, since in the straight-E/R approach, we use only one relation per entity set in the hierarchy. In the object-oriented approach, if we have a root and n children ($n + 1$ entity sets in all), then there are 2^n different classes of entities, and we need that many relations.

3. We would like to minimize space and avoid repeating information. Since the object-oriented method uses only one tuple per entity, and that tuple has components for only those attributes that make sense for the entity, this approach offers the minimum possible space usage. The nulls approach also has only one tuple per entity, but these tuples are "long"; i.e., they have components for all attributes, whether or not they are appropriate for a given entity. If there are many entity sets in the hierarchy, and there are many attributes among those entity sets, then a large fraction of the space could be wasted in the nulls approach. The straight-E/R method has several tuples for each entity, but only the key attributes are repeated. Thus, this method could use either more or less space than the nulls method.

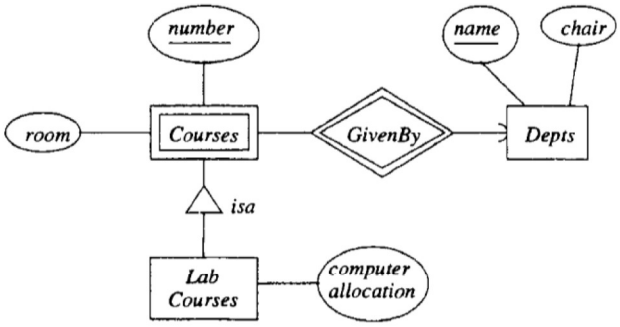

Figure 4.32: E/R diagram for Exercise 4.6.1

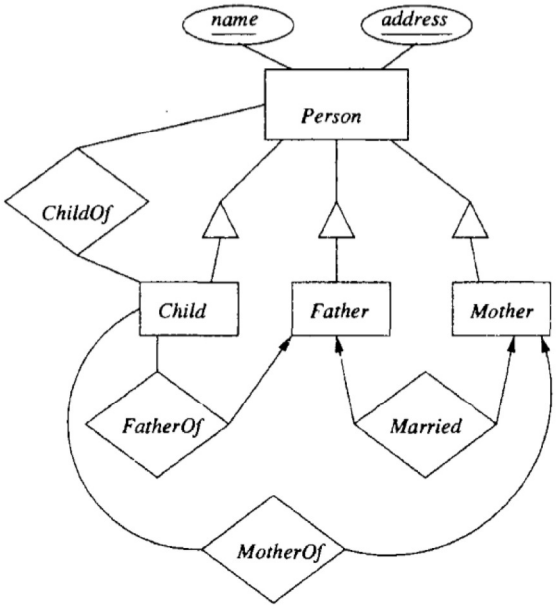

Figure 4.33: E/R diagram for Exercise 4.6.2

4.6.5 Exercises for Section 4.6

Exercise 4.6.1: Convert the E/R diagram of Fig. 4.32 to a relational database schema, using each of the following approaches:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! **Exercise 4.6.2:** Convert the E/R diagram of Fig. 4.33 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

Exercise 4.6.3: Convert your E/R design from Exercise 4.1.7 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! **Exercise 4.6.4:** Suppose that we have an isa-hierarchy involving e entity sets. Each entity set has a attributes, and k of those at the root form the key for all these entity sets. Give formulas for (i) the minimum and maximum number of relations used, and (ii) the minimum and maximum number of components that the tuple(s) for a single entity have all together, when the method of conversion to relations is:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

4.7 Unified Modeling Language

UML (*Unified Modeling Language*) was developed originally as a graphical notation for describing software designs in an object-oriented style. It has been extended, with some modifications, to be a popular notation for describing database designs, and it is this portion of UML that we shall study here. UML offers much the same capabilities as the E/R model, with the exception of multiway relationships. UML also offers the ability to treat entity sets as true classes, with methods as well as data. Figure 4.34 summarizes the common concepts, with different terminology, used by E/R and UML.

UML	E/R Model
Class	Entity set
Association	Binary relationship
Association Class	Attributes on a relationship
Subclass	Isa hierarchy
Aggregation	Many-one relationship
Composition	Many-one relationship with referential integrity

Figure 4.34: Comparison between UML and E/R terminology

4.7.1 UML Classes

A class in UML is similar to an entity set in the E/R model. The notation for a class is rather different, however. Figure 4.35 shows the class that corresponds to the E/R entity set *Movies* from our running example of this chapter.

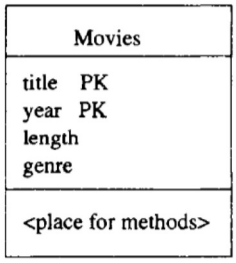

Figure 4.35: The *Movies* class in UML

The box for a class is divided into three parts. At the top is the name of the class. The middle has the attributes, which are like instance variables of a class. In our *Movies* class, we use the attributes *title*, *year*, *length*, and *genre*.

The bottom portion is for methods. Neither the E/R model nor the relational model provides methods. However, they are an important concept, and one that actually appears in modern relational systems, called "object-relational" DBMS's (see Section 10.3).

Example 4.34: We might have added an instance method *lengthInHours*(). The UML specification doesn't tell anything more about a method than the types of any arguments and the type of its return-value. Perhaps this method returns *length*/60.0, but we cannot know from the design. □

In this section, we shall not use methods in our design. Thus, in the future, UML class boxes will have only two sections, for the class name and the attributes.

4.7.2 Keys for UML classes

As for entity sets, we can declare one key for a UML class. To do so, we follow each attribute in the key by the letters PK, standing for "primary key." There is no convenient way to stipulate that several attributes or sets of attributes are each keys.

Example 4.35: In Fig. 4.35, we have made our standard assumption that *title* and *year* together form the key for *Movies*. Notice that PK appears on the lines for these attributes and not for the others. □

4.7.3 Associations

A binary relationship between classes is called an *association*. There is no analog of multiway relationships in UML. Rather, a multiway relationship has to be broken into binary relationships, which as we suggested in Section 4.1.10, can always be done. The interpretation of an association is exactly what we described for relationships in Section 4.1.5 on relationship sets. The association is a set of pairs of objects, one from each of the classes it connects.

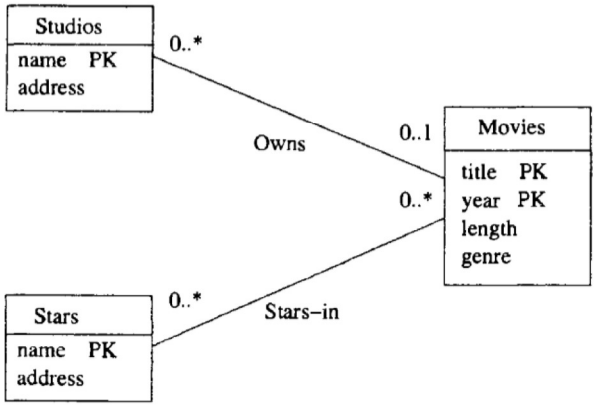

Figure 4.36: Movies, stars, and studios in UML

We draw a UML association between two classes simply by drawing a line between them, and giving the line a name. Usually, we'll place the name below the line. For example, Fig. 4.36 is the UML analog of the E/R diagram of Fig. 4.17. There are two associations, *Stars-in* and *Owns*; the first connects *Movies* with *Stars* and the second connects *Movies* with *Studios*.

Every association has constraints on the number of objects from each of its classes that can be connected to an object of the other class. We indicate these constraints by a label of the form $m..n$ at each end. The meaning of this label is that each object at the other end is connected to at least m and at most n objects at this end. In addition:

- A $*$ in place of n, as in $m..*$, stands for "infinity." That is, there is no upper limit.
- A $*$ alone, in place of $m..n$, stands for the range $0..*$, that is, no constraint at all on the number of objects.
- If there is no label at all at an end of an association edge, then the label is taken to be 1..1, i.e., "exactly one."

Example 4.36: In Fig. 4.36 we see $0..*$ at the *Movies* end of both associations. That says that a star appears in zero or more movies, and a studio owns zero or more movies; i.e., there is no constraint for either. There is also a $0..*$ at the *Stars* end of association *Stars-in*, telling us that a movie has any number of stars. However, the label on the *Studios* end of association *Owns* is 0..1, which means either 0 or 1 studio. That is, a given movie can either be owned by one studio, or not be owned by any studio in the database. Notice that this constraint is exactly what is said by the pointed arrow entering *Studios* in the E/R diagram of Fig. 4.17. □

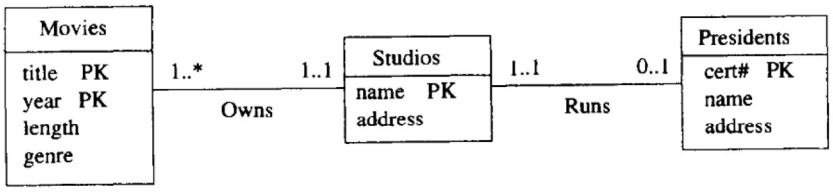

Figure 4.37: Expressing referential integrity in UML

Example 4.37: The UML diagram of Fig. 4.37 is intended to mirror the E/R diagram of Fig. 4.18. Here, we see assumptions somewhat different from those in Example 4.36, about the numbers of movies and studios that can be associated. The label $1..*$ at the *Movies* end of *Owns* says that each studio must own at least one movie (or else it isn't really a studio). There is still no upper limit on how many movies a studio can own.

At the *Studios* end of *Owns*, we see the label 1..1. That label says that a movie must be owned by one studio and only one studio. It is not possible for a movie not to be owned by any studio, as was possible in Fig. 4.36. The label 1..1 says exactly what the rounded arrow in E/R diagrams says.

We also see the association *Runs* between studios and presidents. At the *Studios* end we see label 1..1. That is, a president must be the president of one and only one studio. That label reflects the same constraint as the rounded arrow from *Presidents* to *Studios* in Fig. 4.18. At the other end of association *Runs* is the label 0..1. That label says that a studio can have at most one president, but it could not have a president at some time. This constraint is exactly the constraint of a pointed arrow. □

4.7.4 Self-Associations

An association can have both ends at the same class; such an association is called a *self-association*. To distinguish the two roles played by one class in a self-association, we give the association two names, one for each end.

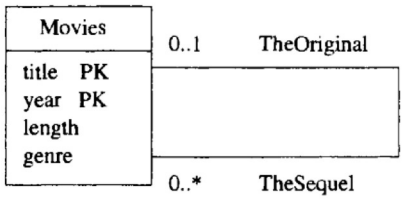

Figure 4.38: A self-association representing sequels of movies

Example 4.38: Figure 4.38 represents the relationship "sequel-of" on movies. We see one association with each end at the class *Movies*. The end with role *TheOriginal* points to the original movie, and it has label 0..1. That is, for a movie to be a sequel, there has to be exactly one movie that was the original. However, some movies are not sequels of any movie. The other role, *TheSequel* has label 0..*. The reasoning is that an original can have any number of sequels. Note we take the point of view that there is an original movie for any sequence of sequels, and a sequel is a sequel of the original, not of the previous movie in the sequence. For instance, *Rocky II* through *Rocky V* are sequels of *Rocky*. We do not assume *Rocky IV* is a sequel of *Rocky III*, and so on. □

4.7.5 Association Classes

We can attach attributes to an association in much the way we did in the E/R model, in Section 4.1.9.[5] In UML, we create a new class, called an *association class*, and attach it to the middle of the association. The association class has its own name, but its attributes may be thought of as attributes of the association to which it attaches.

Example 4.39: Suppose we want to add to the association *Stars-in* between *Movies* and *Stars* some information about the compensation the star received for the movie. This information is not associated with the movie (different stars get different salaries) nor with the star (stars can get different salaries for different movies). Thus, we must attach this information with the association itself. That is, every movie-star pair has its own salary information.

Figure 4.39 shows the association *Stars-in* with an association class called *Compensation*. This class has two attributes, *salary* and *residuals*. Notice

[5] However, the example there in Fig. 4.7 will not carry over directly, because the relationship there is 3-way.

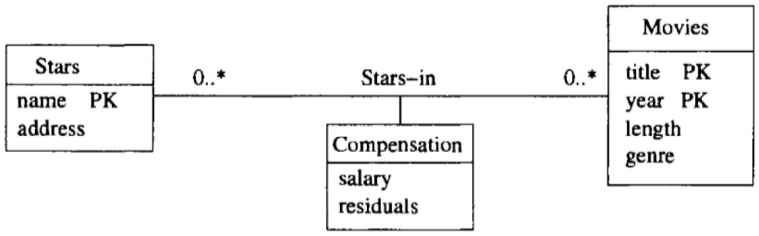

Figure 4.39: *Compensation* is an association class for the association *Stars-in*

that there is no primary key marked for *Compensation*. When we convert a diagram such as Fig. 4.39 to relations, the attributes of *Compensation* will attach to tuples created for movie-star pairs, as was described for relationships in Section 4.5.2. □

4.7.6 Subclasses in UML

Any UML class can have a hierarchy of subclasses below it. The primary key comes from the root of the hierarchy, just as with E/R hierarchies. UML permits a class C to have four different kinds of subclasses below it, depending on our choices of answer to two questions:

1. *Complete versus Partial.* Is every object in the class C a member of some subclass? If so, the subclasses are *complete*; otherwise they are *partial* or *incomplete.*

2. *Disjoint versus Overlapping.* Are the subclasses *disjoint* (an object cannot be in two of the subclasses)? If an object can be in two or more of the subclasses, then the subclasses are said to be *overlapping.*

Note that these decisions are taken at each level of a hierarchy, and the decisions may be made independently at each point.

There are several interesting relationships between the classification of UML subclasses given above, the standard notion of subclasses in object-oriented systems, and the E/R notion of subclasses.

- In a typical object-oriented system, subclasses are disjoint. That is, no object can be in two classes. Of course they inherit properties from their parent class, so in a sense, an object also "belongs" in the parent class. However, the object may not also be in a sibling class.

- The E/R model automatically allows overlapping subclasses.

- Both the E/R model and object-oriented systems allow either complete or partial subclasses. That is, there is no requirement that a member of the superclass be in any subclass.

Subclasses are represented by rectangles, like any class. We assume a subclass inherits the properties (attributes and associations) from its superclass. However, any additional attributes belonging to the subclass are shown in the box for that subclass, and the subclass may have its own, additional, associations to other classes. To represent the class/subclass relationship in UML diagrams, we use a triangular, open arrow pointing to the superclass. The subclasses are usually connected by a horizontal line, feeding into the arrow.

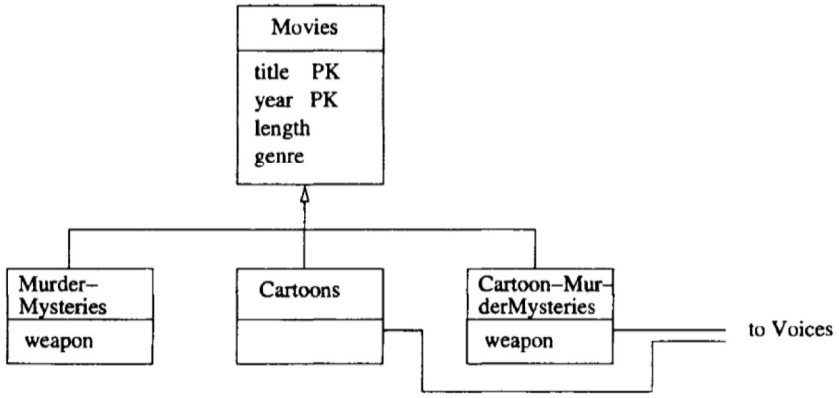

Figure 4.40: Cartoons and murder mysteries as disjoint subclasses of movies

Example 4.40: Figure 4.40 shows a UML variant of the subclass example from Section 4.1.11. However, unlike the E/R subclasses, which are of necessity overlapping, we have chosen here to make the subclasses disjoint. They are partial, of course, since many movies are neither cartoons nor murder mysteries.

Because the subclasses were chosen disjoint, there must be a third subclass for movies like *Roger Rabbit* that are both cartoons and murder mysteries. Notice that both the classes *MurderMysteries* and *Cartoon-MurderMysteries* have additional attribute *weapon*, while the two subclasses *MurderMysteries* and *Cartoon-MurderMysteries* have associations with the unseen class *Voices*. □

4.7.7 Aggregations and Compositions

There are two special notations for many-one associations whose implications are rather subtle. In one sense, they reflect the object-oriented style of programming, where it is common for one class to have references to other classes among its attributes. In another sense, these special notations are really stipulations about how the diagram should be converted to relations; we discuss this aspect of the matter in Section 4.8.3.

An *aggregation* is a line between two classes that ends in an open diamond at one end. The implication of the diamond is that the label at that end must

be 0..1, i.e., the aggregation is a many-one association from the class at the opposite end to the class at the diamond end. Although the aggregation is an association, we do not need to name it, since in practice that name will never be used in a relational implementation.

A *composition* is similar to an association, but the label at the diamond end must be 1..1. That is, every object at the opposite end from the diamond must be connected to exactly one object at the diamond end. Compositions are distinguished by making the diamond be solid black.

Example 4.41: In Fig. 4.41 we see examples of both an aggregation and a composition. It both modifies and elaborates on the situation of Fig. 4.37. We see an association from *Movies* to *Studios*. The label 1..* at the *Movies* end says that a studio has to own at least one movie. We do not need a label at the diamond end, since the open diamond implies a 0..1 label. That is, a movie may or may not be associated with a studio, but cannot be associated with more than one studio. There is also the implication that *Movies* objects will contain a reference to their owning *Studios* object; that reference may be null if the movie is not owned by a studio.

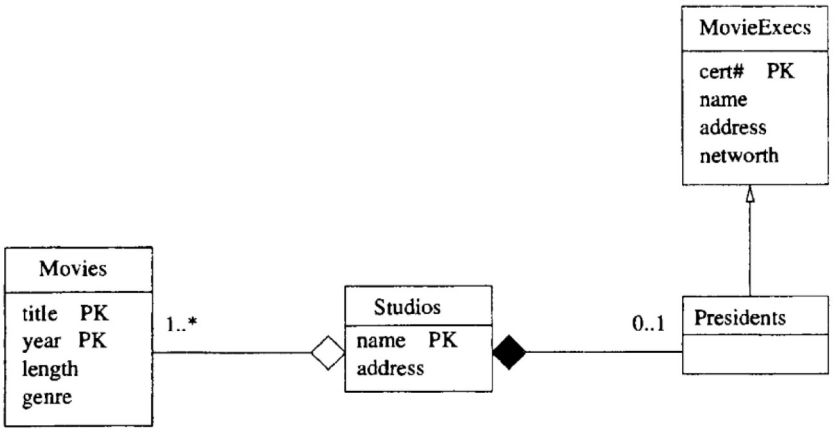

Figure 4.41: An aggregation from *Movies* to *Studios* and a composition from *Presidents* to *Studios*

At the right, we see the class *MovieExecs* with a subclass *Presidents*. There is a composition from *Presidents* to *Studios*. meaning that every president is the president of exactly one studio. A label 1..1 at the *Studios* end is implied by the solid diamond. The implication of the composition is that *Presidents* objects will contain a reference to a *Studios* object, and that this reference cannot be null. □

4.7.8 Exercises for Section 4.7

Exercise 4.7.1: Draw a UML diagram for the problem of Exercise 4.1.1.

Exercise 4.7.2: Modify your diagram from Exercise 4.7.1 in accordance with the requirements of Exercise 4.1.2.

Exercise 4.7.3: Repeat Exercise 4.1.3 using UML.

Exercise 4.7.4: Repeat Exercise 4.1.6 using UML.

Exercise 4.7.5: Repeat Exercise 4.1.7 using UML. Are your subclasses disjoint or overlapping? Are they complete or partial?

Exercise 4.7.6: Repeat Exercise 4.1.9 using UML.

Exercise 4.7.7: Convert the E/R diagram of Fig. 4.30 to a UML diagram.

! **Exercise 4.7.8:** How would you represent the 3-way relationship of *Contracts* among movies, stars, and studios (see Fig. 4.4) in UML?

! **Exercise 4.7.9:** Repeat Exercise 4.2.5 using UML.

Exercise 4.7.10: Usually, when we constrain associations with a label of the form $m..n$, we find that m and n are each either 0, 1, or $*$. Give some examples of associations where it would make sense for at least one of m and n to be something different.

4.8 From UML Diagrams to Relations

Many of the ideas needed to turn E/R diagrams into relations work for UML diagrams as well. We shall therefore briefly review the important techniques, dwelling only on points where the two modeling methods diverge.

4.8.1 UML-to-Relations Basics

Here is an outline of the points that should be familiar from our discussion in Section 4.5:

- *Classes to Relations.* For each class, create a relation whose name is the name of the class, and whose attributes are the attributes of the class.
- *Associations to Relations.* For each association, create a relation with the name of that association. The attributes of the relation are the key attributes of the two connected classes. If there is a coincidence of attributes between the two classes, rename them appropriately. If there is an association class attached to the association, include the attributes of the association class among the attributes of the relation.

Example 4.42: Consider the UML diagram of Fig. 4.36. For the three classes we create relations:

```
Movies(title, year, length genre)
Stars(name, address)
Studios(name, address)
```

For the two associations, we create relations

```
Stars-In(movieTitle, movieYear, starName)
Owns(movieTitle, movieYear, studioName)
```

Note that we have taken some liberties with the names of attributes, for clarity of intention, even though we were not required to do so.

For another example, consider the UML diagram of Fig. 4.39, which shows an association class. The relations for the classes *Movies* and *Stars* would be the same as above. However, for the association, we would have a relation

```
Stars-In(movieTitle, movieYear, starName, salary, residuals)
```

That is, we add to the key attributes of the associated classes, the two attributes of the association class *Compensation*. Note that there is no relation created for *Compensation* itself. □

4.8.2 From UML Subclasses to Relations

The three options we enumerated in Section 4.6 apply to UML subclass hierarchies as well. Recall these options are "E/R style" (relations for each subclass have only the key attributes and attributes of that subclass), "object-oriented" (each entity is represented in the relation for only one subclass), and "use nulls" (one relation for all subclasses). However, if we have information about whether subclasses are disjoint or overlapping, and complete or partial, then we may find one or another method more appropriate. Here are some considerations:

1. If a hierarchy is disjoint at every level, then an object-oriented representation is suggested. We do not have to consider each possible tree of subclasses when forming relations, since we know that each object can belong to only one class and its ancestors in the hierarchy. Thus, there is no possibility of an exponentially exploding number of relations being created.

2. If the hierarchy is both complete and disjoint at every level, then the task is even simpler. If we use the object-oriented approach, then we have only to construct relations for the classes at the leaves of the hierarchy.

3. If the hierarchy is large and overlapping at some or all levels, then the E/R approach is indicated. We are likely to need so many relations that the relational database schema becomes unwieldy.

4.8.3 From Aggregations and Compositions to Relations

Aggregations and compositions are really types of many-one associations. Thus, one approach to their representation in a relational database schema is to convert them as we do for any association in Section 4.8.1. Since these elements are not necessarily named in the UML diagram, we need to invent a name for the corresponding relation.

However, there is a hidden assumption that this implementation of aggregations and compositions is undesirable. Recall from Section 4.5.3 that when we have an entity set E and a many-one relationship R from E to another entity set F, we have the option — some would say the obligation — to combine the relation for E with the relation for R. That is, the one relation constructed from E and R has all the attributes of E plus the key attributes of F.

We suggest that aggregations and compositions be treated routinely in this manner. Construct no relation for the aggregation or composition. Rather, add to the relation for the class at the nondiamond end the key attribute(s) of the class at the diamond end. In the case of an aggregation (but not a composition), it is possible that these attributes can be null.

Example 4.43: Consider the UML diagram of Fig. 4.41. Since there is a small hierarchy, we need to decide how *MovieExecs* and *Presidents* will be translated. Let us adopt the E/R approach, so the `Presidents` relation has only the `cert#` attribute from *MovieExecs*.

The aggregation from *Movies* to *Studios* is represented by putting the key *name* for *Studios* among the attributes for the relation `Movies`. The composition from *Presidents* to *Studios* is represented by adding the key for *Studios* to the relation `Presidents` as well. No relations are constructed for the aggregation or the composition. The following are all the relations we construct from this UML diagram.

```
MovieExecs(cert#, name, address, netWorth)
Presidents(cert#, studioName)
Movies(title, year, length, genre, studioName)
Studios(name, address)
```

As before, we take some liberties with names of attributes to make our intentions clear. □

4.8.4 The UML Analog of Weak Entity Sets

We have not mentioned a UML notation that corresponds to the double-border notation for weak entity sets in the E/R model. There is a sense in which none is needed. The reason is that UML, unlike E/R, draws on the tradition of object-oriented systems, which takes the point of view that each object has its own *object-identity*. That is, we can distinguish two objects, even if they have the same values for each of their attributes and other properties. That object-identity is typically viewed as a reference or pointer to the object.

In UML, we can take the point of view that the objects belonging to a class likewise have object-identity. Thus, even if the stated attributes for a class do not serve to identify a unique object of the class, we can create a new attribute that serves as a key for the corresponding relation and represents the object-identity of the object.

However, it is also possible, in UML, to use a composition as we used supporting relationships for weak entity sets in the E/R model. This composition goes from the "weak" class (the class whose attributes do not provide its key) to the "supporting" class. If there are several "supporting" classes, then several compositions can be used. We shall use a special notation for a *supporting* composition: a small box attached to the *weak* class with "PK" in it will serve as the anchor for the supporting composition. The implication is that the key attribute(s) for the *supporting* class at the other end of the composition is part of the key of the weak class, along with any of the attributes of the weak class that are marked "PK." As with weak entity sets, there can be several supporting compositions and classes, and those supporting classes could themselves be weak, in which case the rule just described is applied recursively.

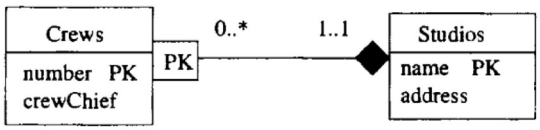

Figure 4.42: Weak class *Crews* supported by a composition and the class *Studios*

Example 4.44: Figure 4.42 shows the analog of the weak entity set *Crews* of Example 4.20. There is a composition from *Crews* to *Studios* anchored by a box labeled "PK" to indicate that this composition provides part of the key for *Crews*. □

We convert weak structures such as Fig. 4.42 to relations exactly as we did in Section 4.5.4. There is a relation for class *Studios* as usual. There is no relation for the composition, again as usual. The relation for class *Crews* includes not only its own attribute *number*, but the key for the class at the end of the composition, which is *Studios*.

Example 4.45: The relations for Example 4.44 are thus:

```
Studios(name, address)
Crews(number, crewChief, studioName)
```

As before, we renamed the attribute *name* of *Studios* in the `Crews` relation, for clarity. □

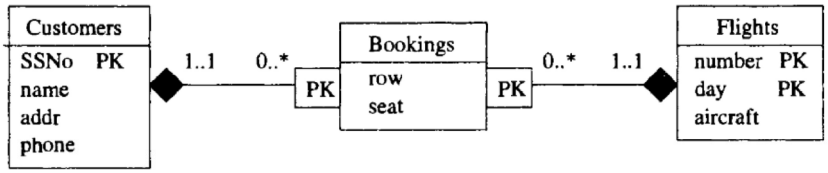

Figure 4.43: A UML diagram analogous to the E/R diagram of Fig. 4.29

4.8.5 Exercises for Section 4.8

Exercise 4.8.1: Convert the UML diagram of Fig. 4.43 to relations.

Exercise 4.8.2: Convert the following UML diagrams to relations:

a) Figure 4.37.

b) Figure 4.40.

c) Your solution to Exercise 4.7.1.

d) Your solution to Exercise 4.7.3.

e) Your solution to Exercise 4.7.4.

f) Your solution to Exercise 4.7.6.

! **Exercise 4.8.3:** How many relations do we create, using the object-oriented approach, if we have a three-level hierarchy with three subclasses of each class at the first and second levels, and that hierarchy is:

a) Disjoint and complete at each level.

b) Disjoint but not complete at each level.

c) Neither disjoint nor complete.

4.9 Object Definition Language

ODL (Object Definition Language) is a text-based language for specifying the structure of databases in object-oriented terms. Like UML, the class is the central concept in ODL. Classes in ODL have a name, attributes, and methods, just as UML classes do. Relationships, which are analogous to UML's associations, are not an independent concept in ODL, but are embedded within classes as an additional family of properties.