8. Developing a Data Model

Fall 2018

We'll introduce to a few problems that frequently occur in order to enlarge your armory for attacking tricky situations.

Today's Lecture

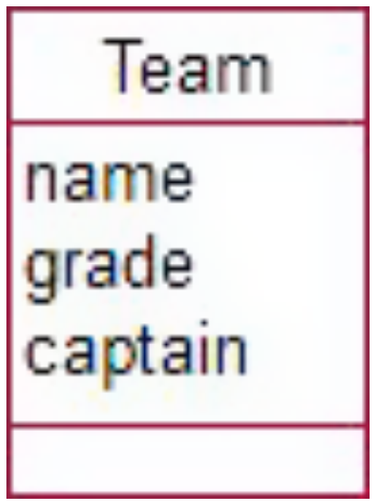
- Attribute, Class, or Relationship?
- Two or More Relationships Between Classes
- Different Routes Between Classes
- Relationships Between Objects of the Same Class
- Relationships Involving More Than Two Classes
- Exercises
- Summary

Attribute, Class, or Relationship?

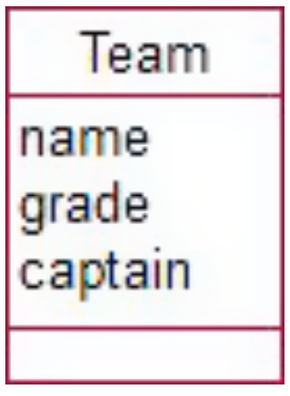
- It is never possible to say that a given data model is the correct one. We can only say that it meets the requirements of a problem within a given scope, and subject to certain assumptions or approximations.
- Which various pieces of data may be represented as an attribute, class, or relationship depending on the overall requirements of the problem.

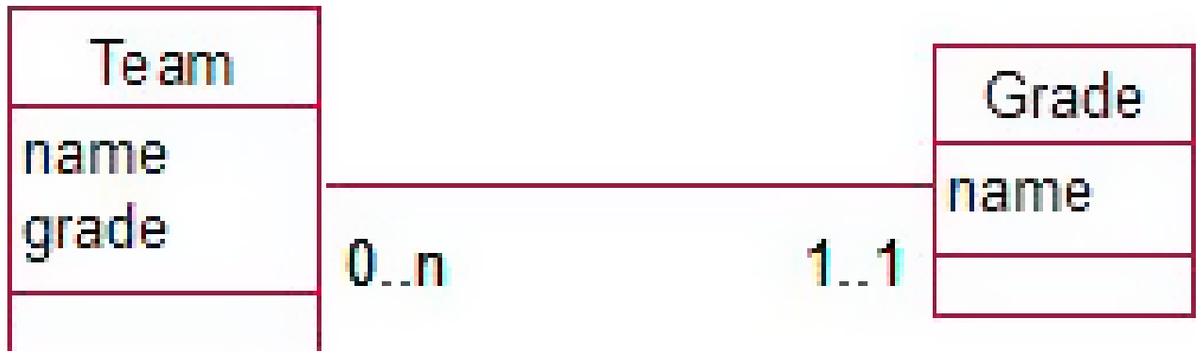
Example 8-1: Sports Club

Let's say we are keeping information about current teams for a sports club. The club wishes to keep very simple records of the team name, its grade, and the captain. As a start we could have a class to contain this information as shown in the figure.

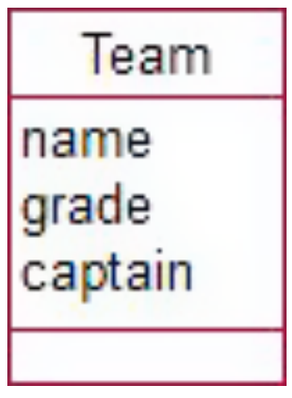


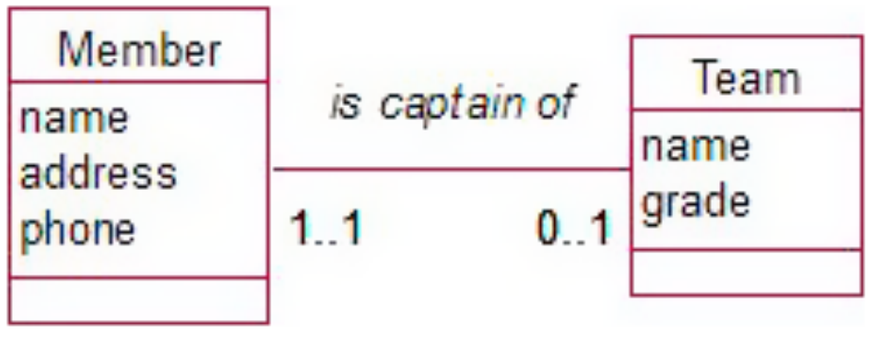
- Each of the pieces of information we are capturing about a team, the name, grade, and the name of the captain is represented by an attribute.
- With this model, we can find the values of the attributes for any given team.
- It is important to consider how the data being stored might be used in the future.
- We may want to find all the teams in a given grade. Will the simple data model allow this?
- It is certainly possible to find all the Team objects with a given value for the grade attribute; however, to obtain reliable data, we would require the data entry to be exact.
- Depending on the requirements of the project, we might choose to represent the grade as an attribute of Team (if the consistency of the spelling is not important) or as a class of its own (if we think we may want to find all the teams belonging to the same grade, for example).





- Consider the captain attribute
- It's unlikely that a person will captain more than one team at a time, so a query analogous to the one in the previous section is unlikely to be a high priority.
- However, there may be some additional data about a captain that we might like to keep: perhaps her phone number and address.
- It is highly likely that this information already exists in some membership list. We very possibly have another class, Member, that keeps contact information about all the members of the sports club
- A particular object of the Member class is the captain of an object of the Team class.
- Depending on the problem, we have different ways to represent a team's captain. We might choose to represent the captain as an attribute of Team or as a relationship between Member and Team.



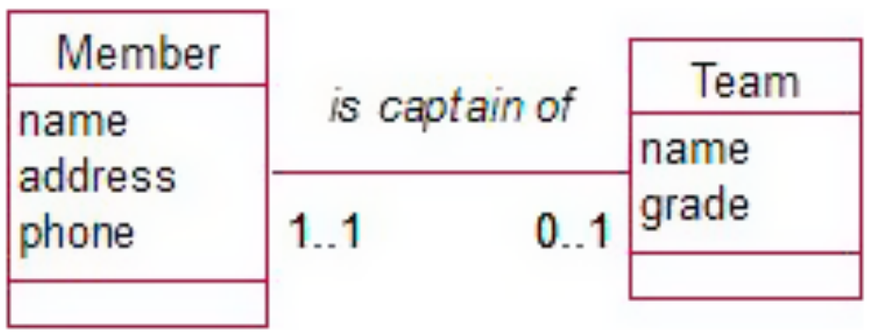


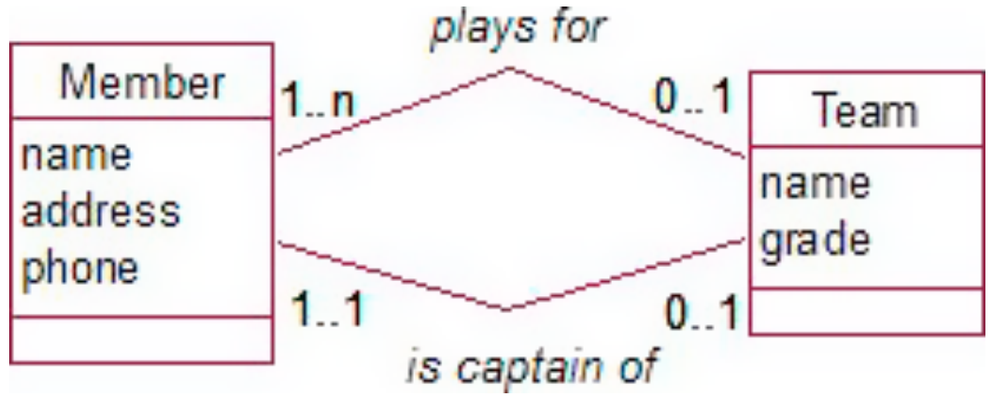
Some useful questions to ask when considering whether to represent information as an attribute, class, or relationship are summarized here:

- “Am I likely to want to summarize, group, or select using a given piece of information?” For example, might you want to select teams based on grade? If so, consider making the piece of information into a class.
- “Am I likely now or in the future to store other data about this piece of information?” For example, might you want to keep information such as phone and address about a captain? Does (or should) this information already exist in another class? If so, consider representing the piece of information as a relationship between the classes.

Two or More Relationships Between Classes

- If we also wanted to keep information about the people playing for a team? We may need to know their names and phone numbers.
- We would need attributes such as player_Name, player_Phone, and so on.
- we probably have the information we require about players already in a Member class. Particular members play for a particular team can therefore be represented as a relationship between the classes.
- We now have two relationships between our Member and Team classes. One is about which members play for the team (could be many). The other is about which member is the captain of the team (just one).



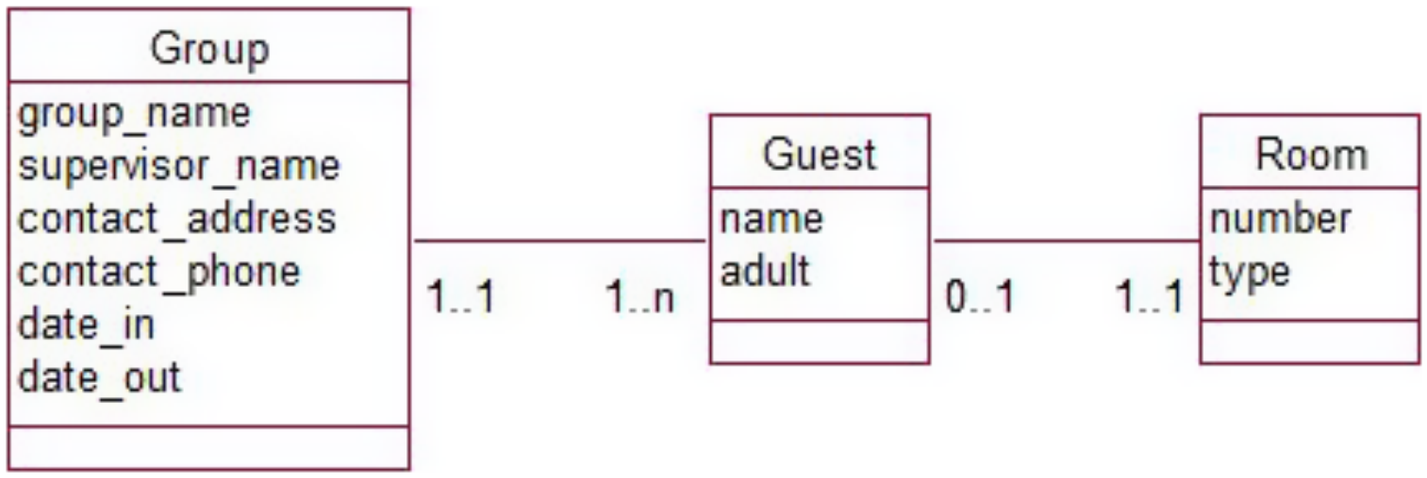


- We may be wondering whether a captain of the team should always be one of the players in that team. The model does not have anything to tell us about such a constraint.
- There are a number of ways to represent constraints.
- It is possible to make the constraints part of the relevant use case. For the use case describing entering information about a team, we would say that the captain has to be one of the players.
- The *Object Constraint Language (OCL)*, developed by the Object Management Group, to supplement the Unified Modeling Language, UML, provides a formal specification for constraints.

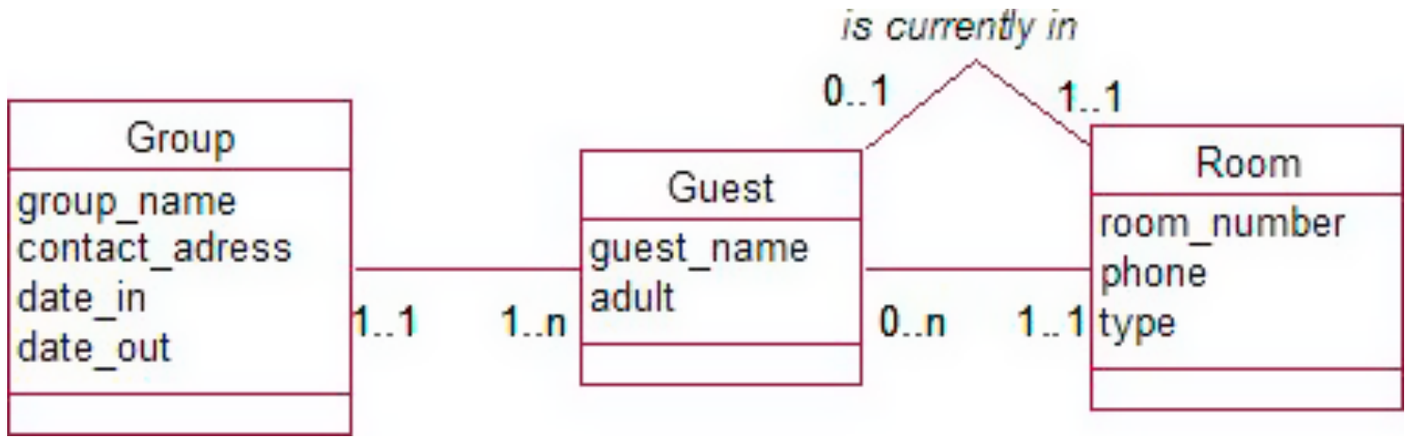
Another situation in which it is possible to consider two relationships between classes is when we have historical data.

Example 8-2: Small Hostel

A small hostel consists of single-occupancy rooms. Typically, groups of people (e.g. school classes) stay at the hostel. We will expand the problem from Chapter 7 by keeping information about previous guests as well as current guests. A room will have many guests over time.(For simplification, we will assume that a guest only stays once and in one room.)



- The hostel primarily caters to groups of visitors and so the check-in and check-out dates belong with the group rather than each individual guest.
- If we have some query about a guest, we can easily find his room number, and we can also find the length of the stay by checking the dates of the related group object.
- There are an increasing number of guests associated with each room over time, so how do we go about finding the current one?
- One way would be to search through all the guests associated with the room and check their associated group information to find one with a date_out value in the future.
- Another likely task is to retrieve a list of empty rooms. To do this, we would have to find those rooms without a guest belonging to a group with a date_out in the future.
- These solutions are quite feasible, but for tasks that are likely to be required regularly, they are complicated and tedious.
- A different option is to consider having a second relationship between Room and Guest for the current guest.



- We can find the current guest with reference to objects of only two classes (Room and Guest). With the model, we needed to inspect date attribute values of the Group object as well.
- To find empty rooms, we can now simply look for all rooms that have no current guest.
- There are a few problems with modeling the data in this way as some extra updating is required to keep the data consistent. For example, when a group checks out, we will have to update the date_out in Group, and we will also have to remove each *is currently in* relationship instance to reflect that the room is now empty.
- This extra maintenance step is caused because we are in effect storing the same piece of information in more than one way.
- While the retrieval of information about empty rooms and current guests is simpler the updating of data is more complex.

	With Data Model in Figure 5-5	With Data Model in Figure 5-6
Check out a group	Update **date_out** for appropriate **Group** object.	Update **date_out** for appropriate **Group** object. Find all associated **Guest** objects and remove the *currently in* association with the **Room** object.
List all currently empty rooms	First find the occupied rooms: find all **Group** objects with **date_out** in the future. Find all the associated **Guest** objects for these groups, and the set of all **Room** objects associated with these guests. List the **room_number** for all **Room** objects *not* included in this set.	Find all **Room** objects that do not have an associated current **Guest** object.

- It is so at the expense of making the maintenance more difficult and therefore the reliability more likely to be compromised.
- We realize that neither of the models is particularly good for the hostel problem if we want to keep historical data about room occupancy.

Different Routes Between Classes

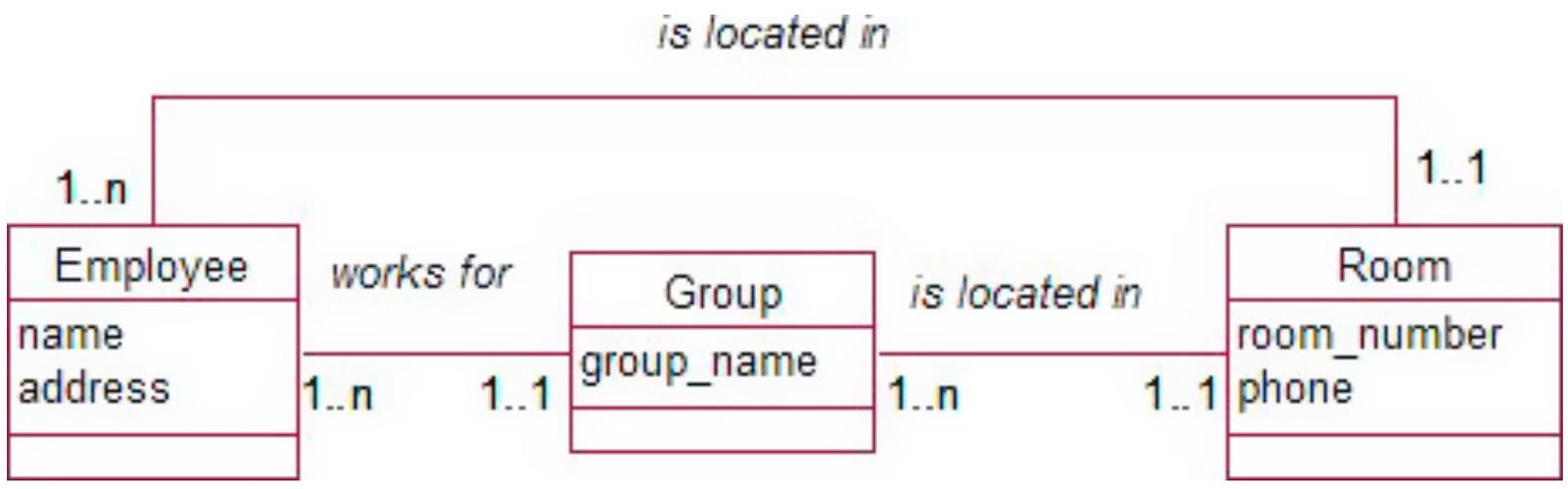
- We can find the current guest in a room by two routes: via the relationship *currently in* or by checking the date_out for each guest who has occupied the room.
- The problem here is that if the data are not carefully maintained, we might find that we come up with two different answers.
- What we should avoid at all costs is having alternative routes for a piece of information when there is no associated reduction in complexity.

Redundant Information

Having what should be the same piece of information available by two different routes can be referred to as redundant information.

Example 5-3: Startup Incubator

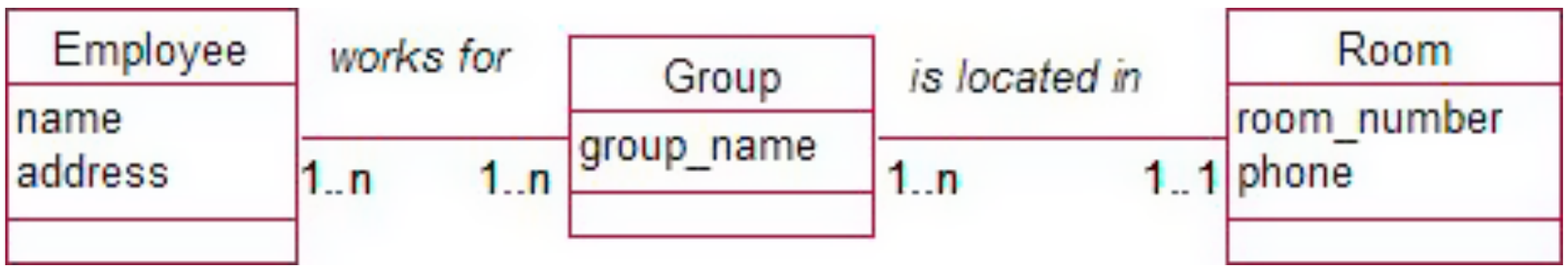
A startup incubator has employees who each work for one of a number of different small project groups. Each group and all its employees are housed in one particular room, with larger rooms housing several groups. We may require information such as where each employee is located, a particular employee's phone number, where to find a particular group, which employees work in each group, who is in each room, and so on. One possible data model is shown in the figure. The model has redundant information. Can you see what it is?



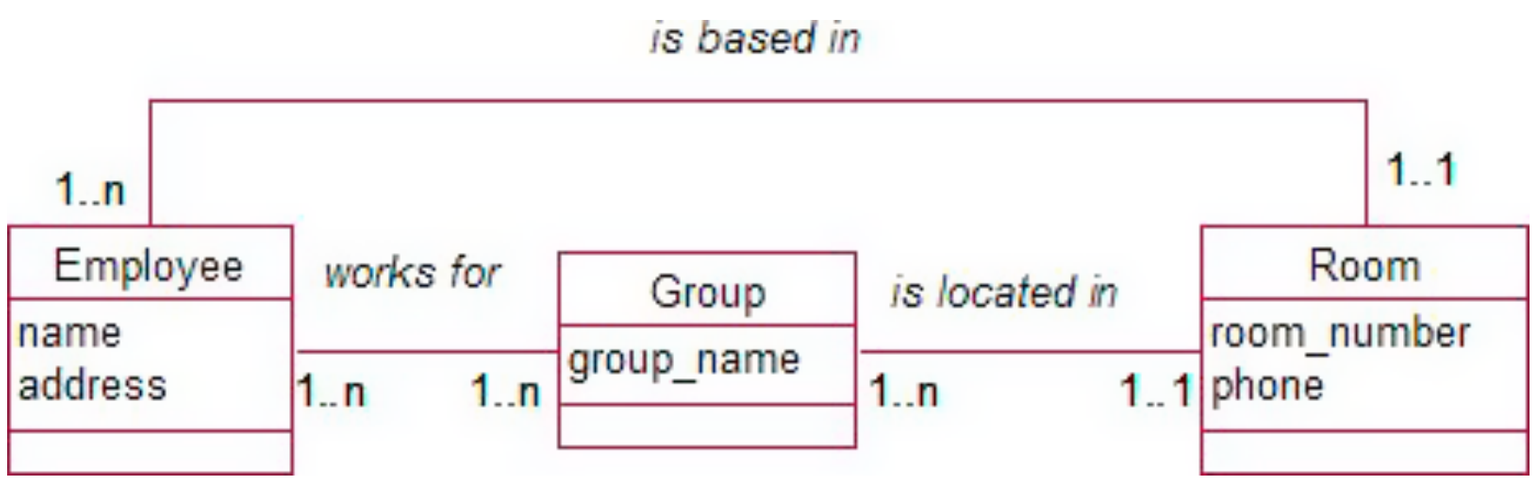
- If we regularly want to find an employee's phone number, we might think that the top relationship between Employee and Room would be a useful direct route.
- However, this same information is very easily available by an alternative route through Group. We can find the employee's (one only) group and then find that group's (one only) room. This is a very simple retrieval.
- The extra relationship is not just unnecessary, it is dangerous. With two routes for the same information, we risk getting two different answers unless the data is very carefully maintained.
- Whenever an employee changes groups or a group shifts rooms, there will be two relationship instances to update.
- Redundant information is prone to inconsistencies and should always be removed.
- Whenever there is a closed path in a data model, it is worth checking carefully to ensure that none of the relationships are redundant.

Routes Providing Different Information

- Not all closed paths necessarily mean redundant data.
- One of the routes may contain different information. Alter the problem slightly to allow an employee to work for more than one of the small project groups.



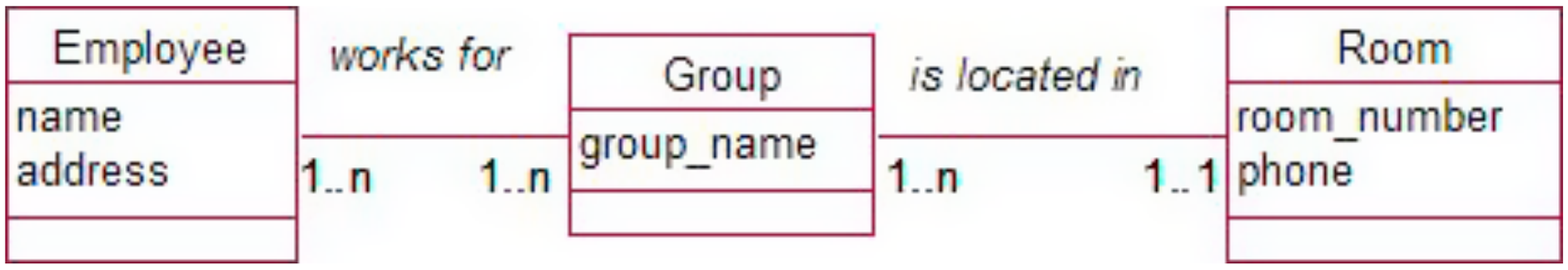
- There is no certain clear route between an employee and a particular room. For example, Group A may be in Room 12, Group B in Room 16, and Jim may work for both groups. Thus, Jim could be in either Room 12 or Room 16.
- If, each employee has a home room and we wish to record that information, we will need an additional relationship between employee and room



- For real-life problems, this may be exactly what is required. The size of a room and the number of employees in a group are unlikely to always match.
- The important thing is to ensure that two routes do not contain what should be identical information so we do not introduce avoidable inconsistencies.

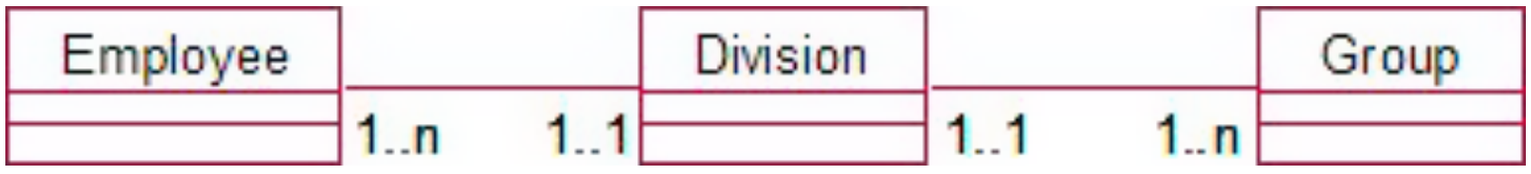
False Information from a Route (Fan Trap)

- Not being able to deduce an employee's room is an example of a more general problem.

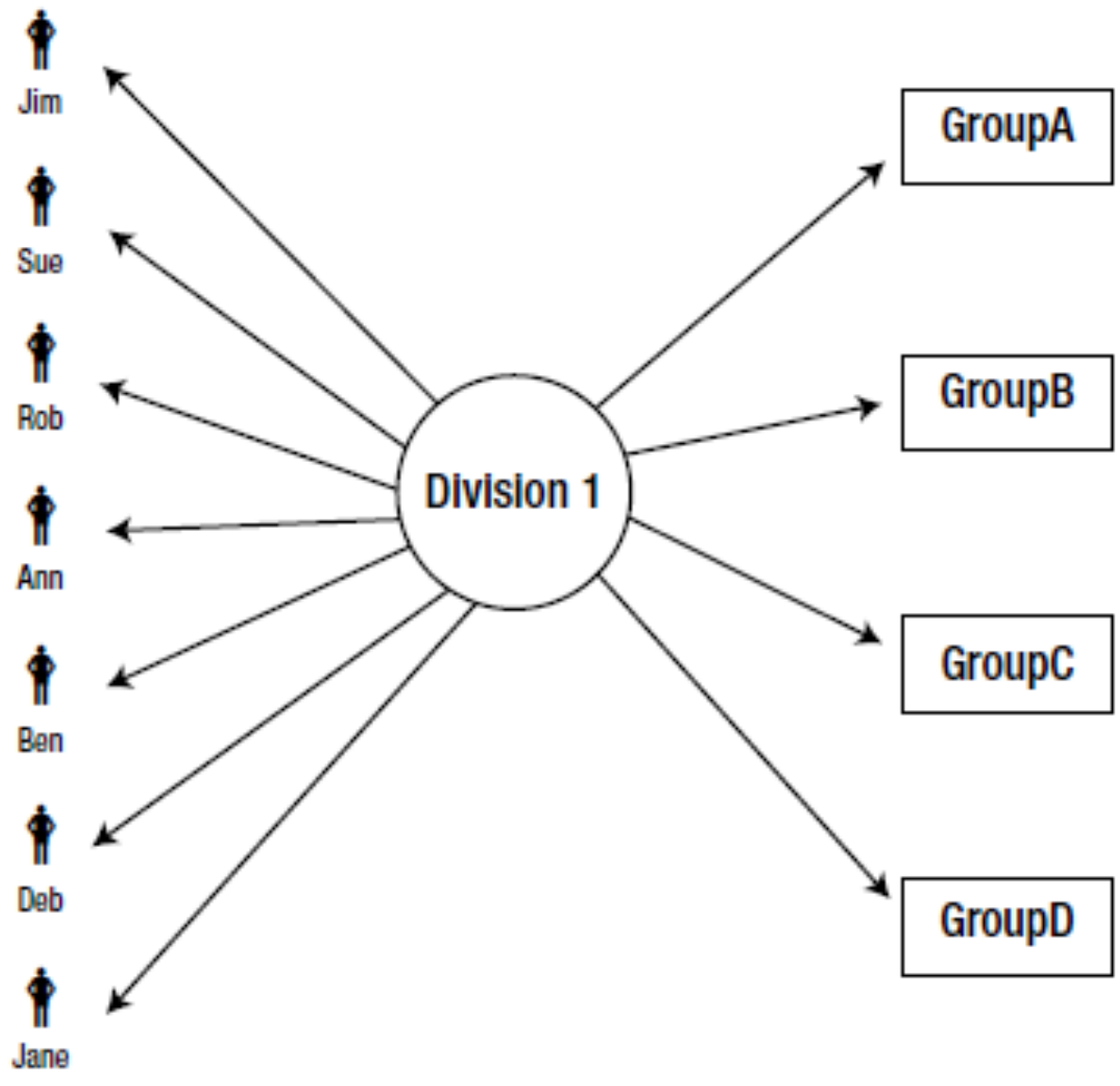


Example 5-4: larger organIzatIon

An organization has several divisions. Each of these divisions has many employees and is broken down into a number of groups. We might model this as in the figure. Have a look at the model. What can we deduce about which group or groups a particular employee is associated with?



- The figure represents a very common problem often referred to as a fan trap.
- The danger here is to take a route between employee and group and infer something that was not intended.



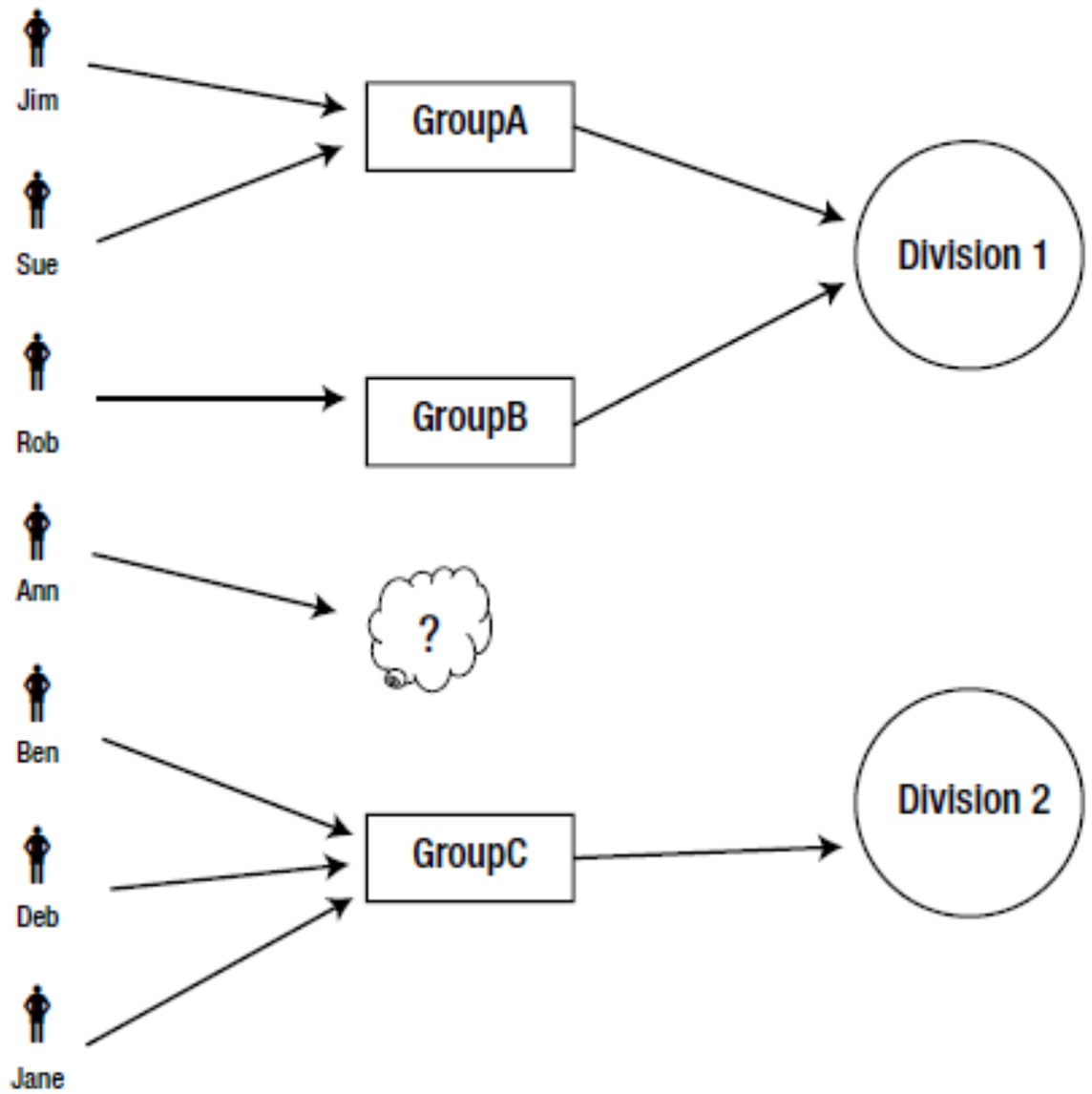
- Consider employees.
- It is not possible to infer anything about which groups Jim or Sue work for.
- It is only possible to get many combinations of a Group object and an Employee object that have a Division in common.
- We must not mistake these combinations for the information we require—for example, which group or groups does Jane belong to?

- The feature that alerts us to a fan trap is a class with two relationships with a Many cardinality at the outside ends.
- What can we do about it?
- If it is important for our system to be able to show the groups for which an employee works, we will need another relationship between Group and Employee, or we may need to model the problem quite differently.

Gaps in a Route Between Classes (Chasm Trap)

- We might choose to model the relationships between divisions, groups, and employees in a hierarchical way
- The optionality at one end of the employee-group relationship has not been specified.

- We have a direct connection between an employee and a single group (Jim works for Group A) and another between a group and its one division (Group A is in Division 1).
- We can therefore make a confident and unique connection between Jim and Division 1.

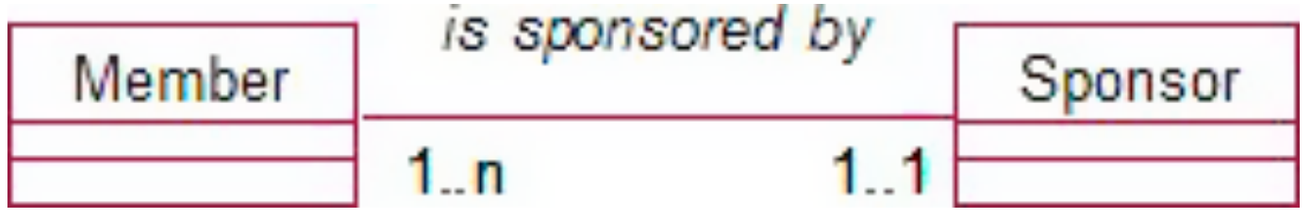


- It is always useful to check that that connection is always there.
- What if Ann is not attached to a specific group? Maybe she is a general administrator for Division 1 and serves all groups.
- If an employee does not necessarily belong to a group, the model does not provide a link between Ann and her division. To find the appropriate Division object, we need to know the Group, and Ann has no related Group object. If we need to know this information, we have a problem.
- This is sometimes referred to as a *chasm trap*.
- This is yet another case where careful study of the data model provides quite interesting questions about the problem.
- We should always check for the exceptional case of an employee who may not be attached to any group.

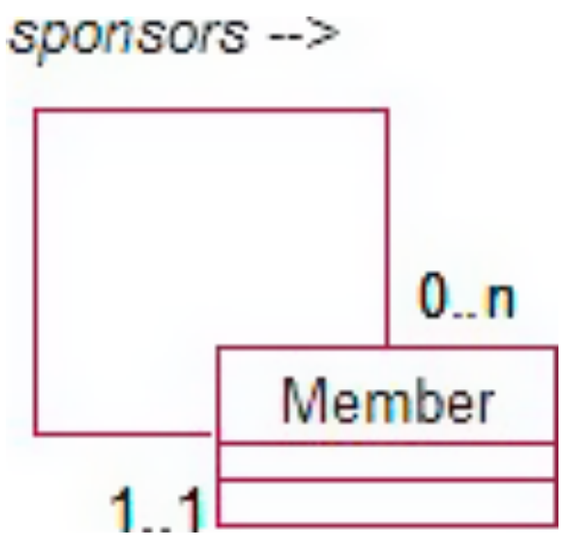
- How we solve the problem of a chasm trap depends on the situation we are modeling.
- One possibility is to add another relationship between division and employee so we can always make that connection.
- However, this extra relationship is going to cause redundant information.
- A different way to get around the problem is to introduce another group object (administration or ancillary staff).
- Ann could belong to this group, and we can then insist that every employee must be in a group. However, it may be that the problem needs to be remodeled entirely.
- It is often best to go back to the use cases and reconsider what information is the most important for the problem. It is never possible to capture every detail in a project with finite resources, so pragmatism becomes very important.

Relationships Between Objects of the Same Class

- Let's return to our sports club.
- Many clubs require a new member to be introduced or sponsored by an existing member.
- If it is necessary to store sponsorship information, a first attempt at a data model might be as shown in the figure.



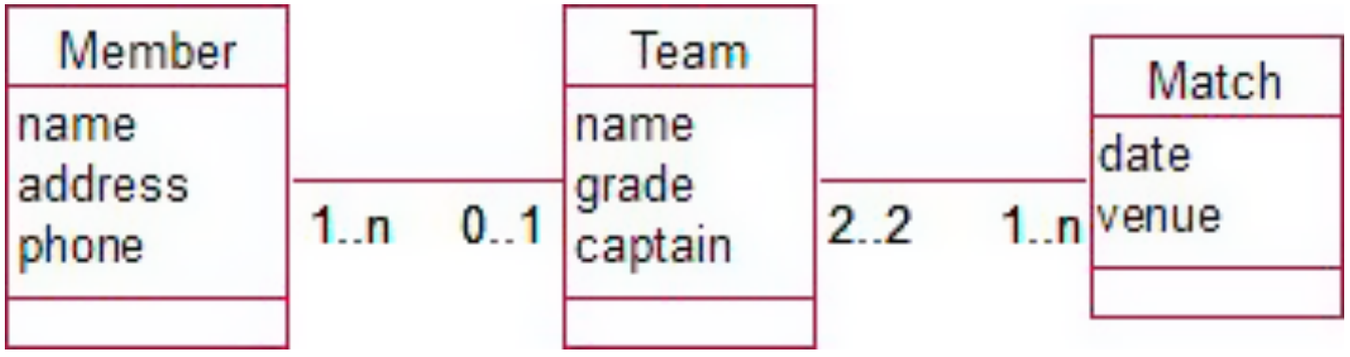
- The problem with the model is that (by definition) a sponsor is a member.
- The model will mean that if Jim sponsors a new member of the club, we will be storing two objects for him (one in the Member class and one in the Sponsor class), both probably containing the same information (until it inevitably becomes inconsistent).
- What is really happening here is that members sponsor each other.
- This can be represented by a self relationship.



- The relationship is read exactly the same as a relationship between two different classes.
- Reading clockwise, we have *a particular member may sponsor many members*, while counterclockwise we have *a particular member is sponsored by exactly one member*.
- Nothing in this data model prevents members from sponsoring themselves. Such constraints need to be noted, most usefully by mentioning them in the appropriate use case (e.g., adding a member).
- Self relationships appear in many situations. This is certainly true for data pertaining to genealogy or animal breeding.

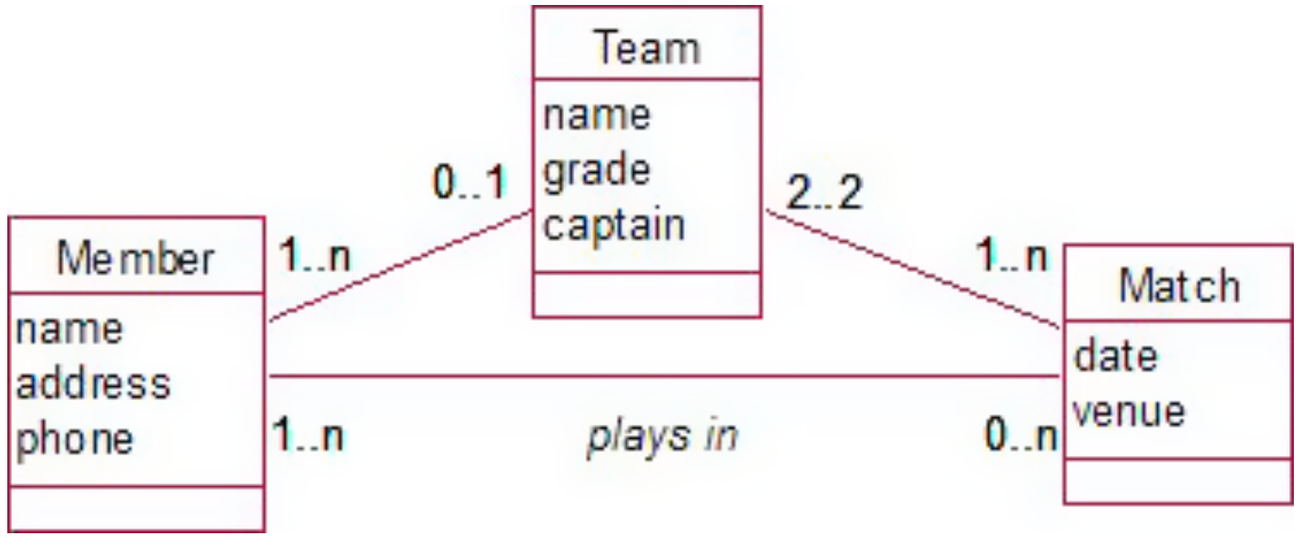
Relationships Involving More Than Two Classes

- Sometimes we have data that depend on objects of more than two classes. Let's reconsider a sports club.
- As well as keeping data about members and their current team, we might also want to keep information about games or matches between teams.



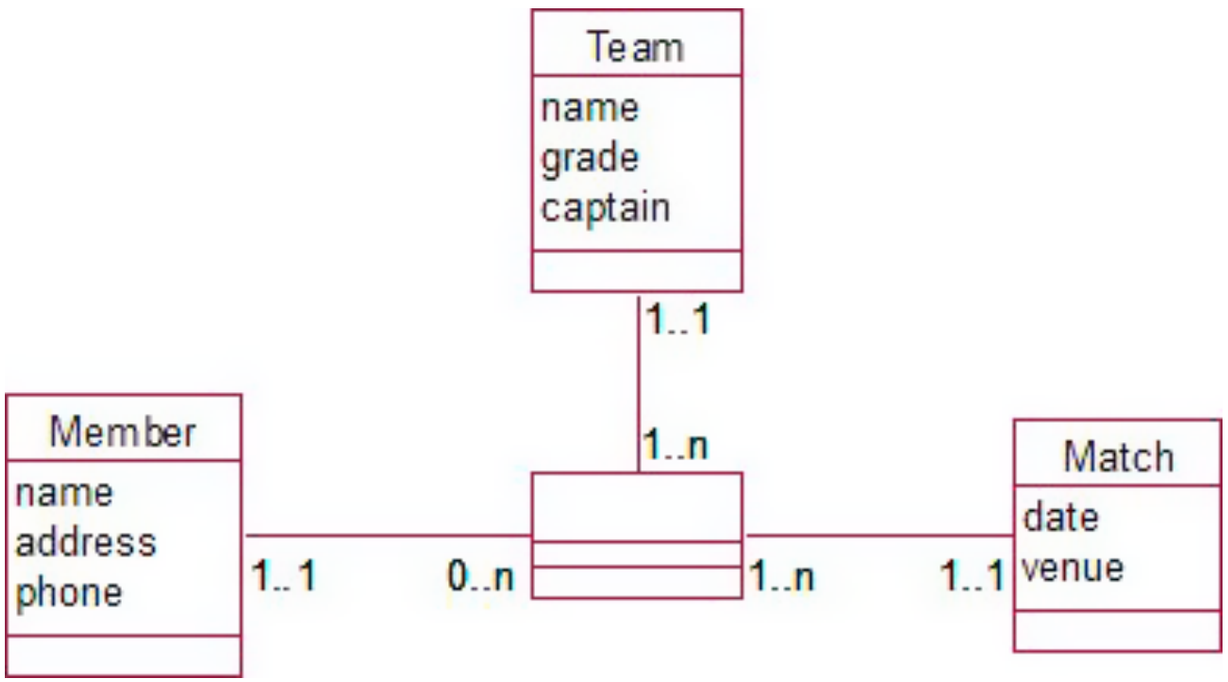
- The model allows us to record a player's current or main team, the current members of a particular team, and the matches in which teams are involved.

- We cannot deduce that a particular player played in any given match (he may have been sick or injured).
- This is an example of the fan trap described earlier.
- A team has many players and is involved in many matches, but we cannot say any more about which players were involved in particular matches.
- We could attempt to address this by adding a relationship between Member and Match.
- Can you see where there is a possibility that the data might become inconsistent?



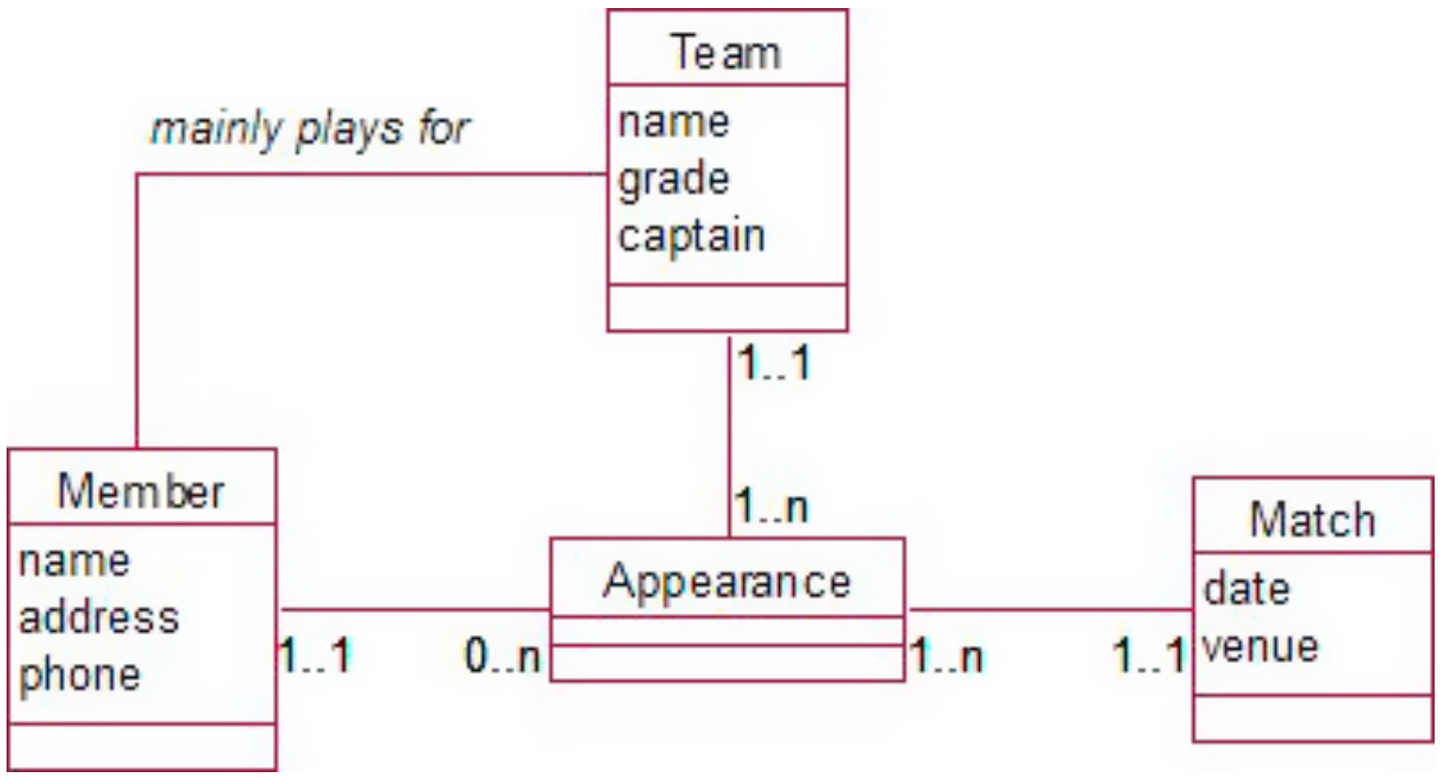
- It is possible to have the following relationship instances:
  - John plays for Team A.
  - John plays in the match on Tuesday.
- The match on Tuesday is between Teams B and C.
- If John plays for only one team (as the model indicates), then something weird is going on here.
- If we want to keep track of who plays in which matches, our problem has some intricacies that the model does not adequately represent.
- If we are allowing for people being injured and not taking part, we also need to account for the situation in which someone from another team may need to fill in as a replacement.
- We still have a problem. the model doesn't tell us which team he was playing for.

- We need to revisit the use cases, and figure out exactly what it is we want to know.
- If we want to know exactly which players played on each team for each match, then no combination of the relationships will tell us that.
- The crucial point is that who played for which team in which match requires simultaneous knowledge of objects from three classes: which Member, which Team, and which Match.
- This is sometimes referred to as a ternary relationship (and, similarly, quaternary for four classes and so on).



- We might be able to think of an appropriate name for this class; in this case Appearance would be sensible. If not, concatenating the other three class names will suffice (e.g., Team/Member/Match).
- The new class may or may not have attributes. It may just be a holding place for valid combinations of Member, Team, and Match objects. If there are attributes for the new class, they must be something that involves all three classes. For example, what do we need to know about a particular player playing for a particular team in a particular match? Possibly the position.
- The model clearly has additional information that is impossible to deduce from the before.
- There may be other information about each pair of classes that we would like to keep. For instance, we know all the teams Jim played for but we don't know the team with which he regularly trains. Some binary relationships between each of the three classes may be required in addition to the relationships with the new class.

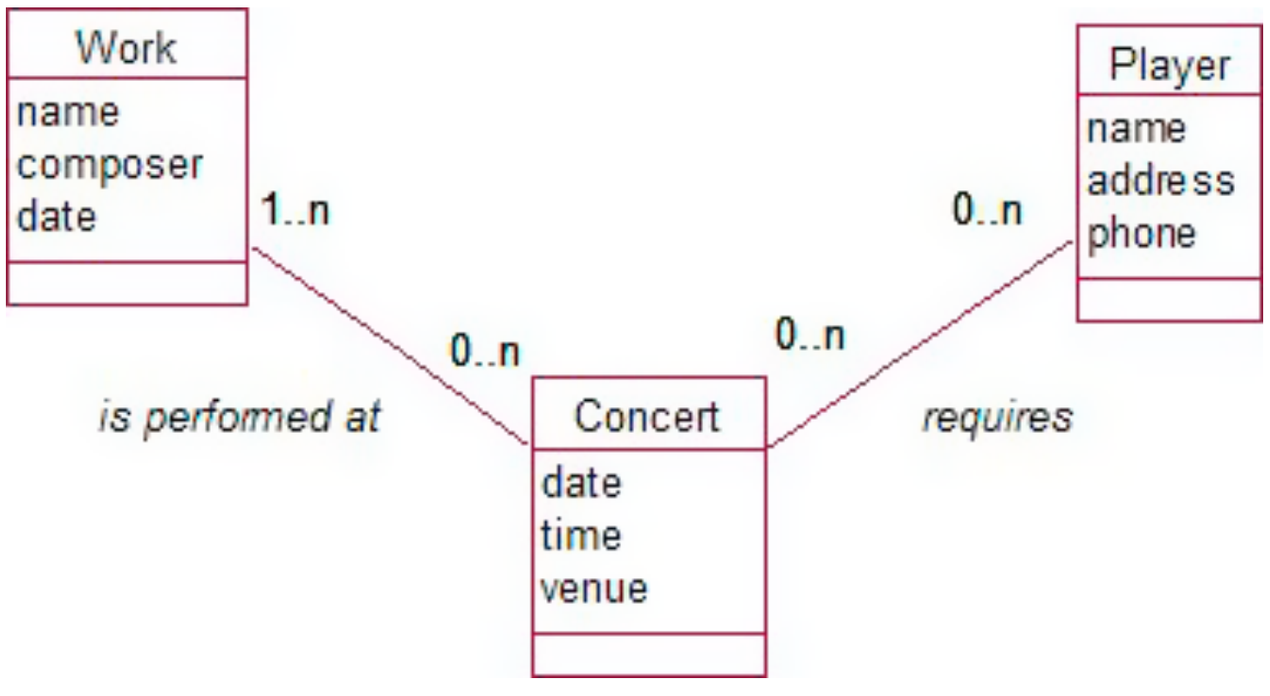
- If we have classes A, B, and C connected to a third class, for each pair of classes we should ask a question like, “Is there something I need to know about a relationship between A and B that is independent of C?”



- We need to ask a similar question for each of the other combinations; for example, “What do I need to know about a particular team and a particular match independent of the members?”

Exercise 8-4

An orchestra keeps information about its musicians, its repertoire and concerts. A partial data model is shown in the figure. The relationships store information such as that Joe Smith is required for Saturday's concert and that Beethoven's violin sonata is to be performed at Saturday's concert.



What false information could be deduced from this initial model?

Amend the model so that it can maintain the following information correctly:

- *Which players are involved in particular works in a concert*
- *The works being presented at a concert*
- *The fee a player receives for appearing in a particular concert*

Summary

Attribute, class, or relationship?

- Here are some examples of questions to help you decide. Might I want to select objects based on the value of an attribute? For example, might you want to select teams based on grade? If the answer is yes, consider introducing a class for that information (create a grade class).
- Am I likely now or in the future to store other data about this information? For example, might I want to keep additional information about the captain: phone, address, and so on? If yes, consider introducing a class.
- Am I storing (or should I be storing) such information already? For example, the information about a captain is the same or similar to information about members. Consider a relationship between existing classes.

More than one relationship between two classes:

- Consider more than one relationship between two classes if there is different information involving both classes. For example, a member might play for a team, captain a team, manage a team, and so on.

Consider self relationships:

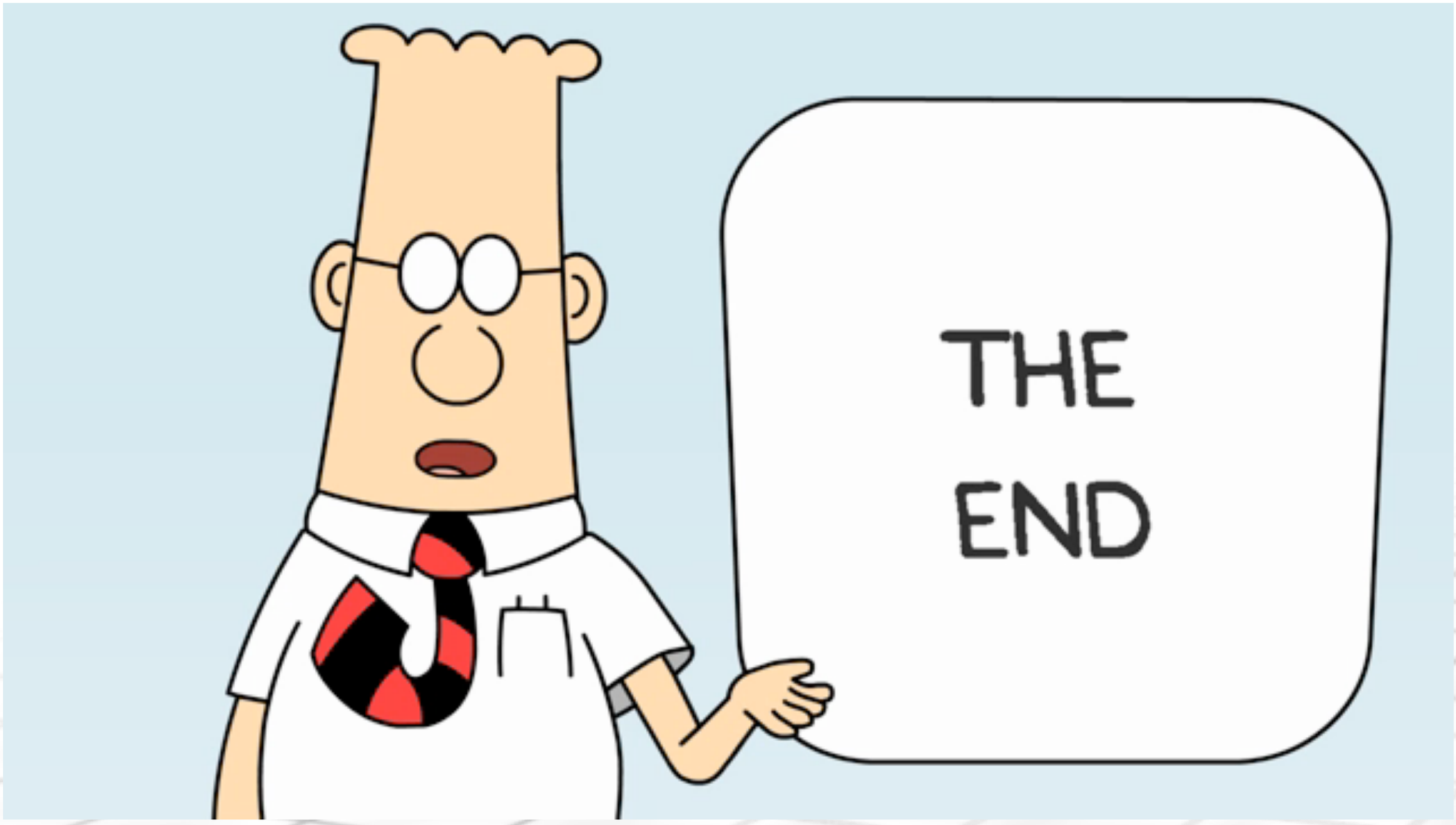
- Objects of a class c an be related to each other. For example, members sponsor other members, people are parents of other people.

Different routes between classes:

- Check wherever there is a closed loop to see whether the same information is being stored more than once.
- Check to ensure you are not inferring more than you should from a route; that is, look out for fan traps where a class is related to two other classes and there is a cardinality of Many at both outer ends.
- Check to ensure a path is available for all objects; that is, look out for chasm traps (are there optional relationships along the route?).

Information dependent on objects of more than two classes:

- Consider introducing a new class where you need to know about combinations of objects from three or more classes simultaneously; for example, which member played for which team in which match?
- Any attributes in the new class must depend on a particular combination of objects from each of the participating classes; such as, what do I need to know about a particular member playing on a particular team in a particular match?
- Consider what information might be pertinent to two objects from pairs of the contributing classes; for example, what do I need to know about a particular member and a particular team independent of any match?



출처: metachannels.com

Thank you!