You can make up constraints on the attributes on the issue
Must put a limit on the size of strings so that the records in data base are of fixed length

# 5. Basic Object/Data Modelling

Last Mod: 2024-05-12

© 2024 Russ Tront

Part of Requirements Analysis is distilling the gathered info into forms more suitable for designers and programmers.

For the first assignment, in this section we're first going to study a subset of entity modelling and object-oriented analysis:

- Entity/Object Modelling.
- Entity/Object Normalization
- Formalizing Many-to-Many relationships

Midterm will be announced ten days in advance.

We’ll first briefly overview traditional information systems data ‘entity’ modelling.

- Provides a super-logical way to organize our data.
- Maps nicely later into object-oriented analysis, design, and programming.

# Table of Contents

## 5.1 Required Readings

- Appendix A: Why Organize Data Properly?

## 5.2 Object Orientation

Object orientation means something different to each team member:

- To business system analysts it means determining and focusing on the business entities.
  - E.g. sales item, customer, invoice, et cetera, about which information must be processed or recorded.

A software designer is a programmer with more experience.

- To a software designer, it's the architectural view that a system satisfies each external command or event, by the set of actions resulting from the trace of calls/messages sent among various reactive software components to implement that request.

- To a programmer, it means a programming language that allows the programmer to easily:
  - view data as having reactive abilities, and
  - re-use code via inheritance hierarchies, and
  - have both type flexibility and ease of maintenance via polymorphism.

We'll initially de-emphasize the OO language view to concentrate on entity/object data modelling.

Humans naturally understand the object-oriented paradigm. For instance, a car is an object. It has:

- a stored identity (licence or serial number),
- stored data about itself (odometer reading).
- the abilities to respond to requests (start, turn on its left turn signal, etc.).

Humans naturally simplify and bring organization to their life by categorizing objects. The two most common ways of categorizing objects are:

- membership in a class of similar instances.
  - E.g. all Honda Rav4's, all black bears, all personnel records.
- composition or ownership.
  - E.g.  A truck is composed of its parts.

These categorizations are simplifying abstractions, so we need **not** mention, or be distracted by, the full details.

- We don’t have to enumerate the identity of every Honda Rav4 in the world, we just say “Honda Preludes”.
- We needn’t list a truck’s parts when we refer to it, we just say “the truck”.

Abstraction is the **major** way to simplify a complex systems and world.

In fact, we form abstraction hierarchies.

- E.g. Toyota Rav4 is a sub-class of the larger abstraction we call passenger vehicles.
- And a truck fleet can be made up of a number of trucks, which in turn are composed of parts.

**Don't get the two hierarchies above mixed up!** They are orthogonal!

- One is categorization by "type class".
- Other is categorization by "composition" (by construction or aggregation).

Both kinds of hierarchies are smoothly modeled and implemented with object-oriented software designs and implementations.

## 5.3 Introduction to Modelling in General

A model is a representation of an actual thing.

- To a child, a model is something created which is a 'smaller' but adequate likeness of the real thing.
- To a car dealer, a model is a bunch of cars which are near identical (cf. object 'class').
- In systems analysis, a model captures the essential nature of something by indicating the essential details that need to be **stored** about things of that 'class'.

**Definition: A model is an alternate representation with an 'adequate likeness' of the real thing.**

Memorize this

Some of the alternate representations systems design may use for actual things are:

- diagram or picture
- form or computer record
- process description, data flow diagram, or
- finite state machine

The purpose of creating a model is to represent **only the essential characteristics** of the thing so that:

- we may understand and clearly document the nature of the thing,
- we may store the essence of the thing for later retrieval,
- we may communicate the nature of the thing to someone else,
- they can think and/or reason about the correctness of the model without:
  - being distracted by the complexities of the complete real thing (i.e. abstraction).

– having to travel to where the real thing is located.

– having to see the function of a real thing while it is operating very fast. Ask professor about this point

– we needn't waste space storing useless information about the thing,

– we may write a program to implement a system which allows humans to better administrate the processes in which the 'thing' participates.

## 5.4 Entities vs. Objects

The data persistently stored (across a reboot) are usually computer **records** of instances of various entity classes in the application domain.

- E.g. orders, customers.

Traditionally these were called **entities**.

They're the 'application' info we store in databases.

E.g.

- The fields of a disk record, or data members in **`struct`**s in RAM.
- The data members of an **`object`**s in RAM.

Each entity class has a record/structure type with a different layout of attribute fields.  E.g.

- Customer records have name, address, and phone number record fields.
- Order instances have order ID number, part ID designator, and ‘quantity of order’ fields.

They often become the *data member* portion of objects we’ll use in programming.

E.g.

Non-OO in C language:

```
struct CustomerType custRecord;
printRec(custRecord, theFastPrinter);
```

More object oriented in C++:

```
CustomerType custInstance;
custInstance.print(theFastPrinter);
```

## 5.5 Advantages of Encapsulation

The data and behavioral abilities of an object class are said to be **encapsulated** together. Contains both:

- Data representation (including state-like),
- Operations that describe what can be done to an object of that class.
  - I.e. what functions can be invoked on it, what messages can be sent it).
- The internal (state) data of an instance can be used to control how it reacts/behaves when:
  - operations (e.g. 'withdraw)

  - under various conditions/modes (e.g. ‘frozen’ bank account)

are invoked on/done to it.

Objects are **natural for human beings**.

- It’s our nature to put together things which belong together.
- We make **less mistakes** when we’re manipulating and designing with natural-feeling things.

Another reason for encapsulation it’s the **correct way to group program details** about things so as to **make maintenance easier**!

We prefer during maintenance to avoid tearing stuff apart and re-arranging it.

- If the entire nature of each application object is encapsulated, we’re less likely to have to rip them apart during maintenance (since they inherently ‘belong’ together).

Oh, we may need to:

- add extra data attributes to an object, or
- change and add operations, or
- even add or delete whole classes of objects to/from our design.

But if we've done our analysis and design right, then we are unlikely to have to **tear objects apart**, nor **merge partial aspects of two objects**.

E.g. Think of an airline reservation system.

No matter what kind of maintenance needs to be done on the application code, there will always need to be:

- aircraft objects (to characterize the nature of the aircraft),
- flight objects (to store time and load on particular trips), and
- passenger objects which can be added to one or more (in the case of a return trip) flights.

We usually start requirements gathering and analysis by first determining the stored object data and relationships needed by the application, and leaving functional abilities to later.

This is done because:

- In ‘information’ systems, this data and its logical organization is key.
- And even for systems that don’t retain data, objects are just records in RAM, and their identification and attributes (data members) are key.

Examples of the object classes needing to be modeled within an application might be:

- a physical object (e.g. person, aircraft, robot, printer).
- an incident or transaction that needs to be recorded either for:
  - immediate use,
  - for transmission to someone else, or
  - for a historical log (e.g. order, purchase, sale, boarding an airplane, graduation, marriage, phone call).
- a role (e.g. student, client, customer, manager, spouse).

- an intangible concept (e.g. bank account, time delay, date, sound recording).
- a place (e.g. parking space, warehouse #3, the 13th floor heat control).
- a relationship (e.g. customer's sales representative, a flight's captain).
- a structure - e.g. the list of an airplane's component part numbers (body, wings, engines, tail), possibly even a hierarchy. Or a container/list of things.
- an organization or organizational unit (e.g. university, department, corporation, submarine crew, sports team).

- a displayable field (e.g. string, icon, image) or printed report, or an I/O signal.
- Specifications or procedures- e.g. organic compound or recipe.

## 5.6 Object Attributes and Attribute Values

It is common for entities of a single class to be modelled as a table of fixed-length records:

STUDENT TABLE

| **student-id** | **student-name** | **student-address** | **student-phone** | **high-school** |
|---|---|---|---|---|
| 93010-1234 | Smith, Bill | 123 Second St. | 420-1234 | Mt. Douglas |
| 92010-4321 | Jones, Jane | 234 Third St. | 123-4567 | Burnaby |
| 91111-1056 | Able, Jim | 345 Fourth Rd. | 822-9876 | John Oliver |

Each column represents an **attribute** of the type 'student' (i.e. a field of a student record, or a data member of an object).

The legal set of values that an attribute may take on is called the **domain** of the attribute. Examples are:

- date = (1..31), and
- day= (Sunday..Saturday).

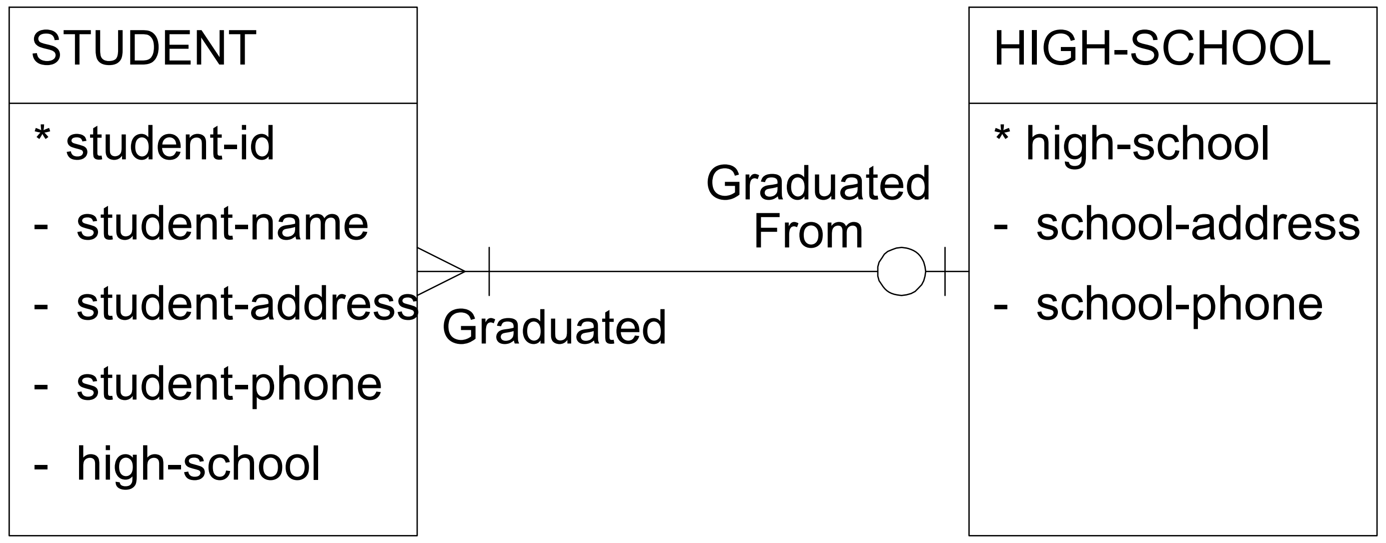
Each row represents a particular **instance** of a student.

Often the rows are sorted in order by a particular column or columns. That column(s) is called the **primary key**.

## 5.7 Entity/Object Relationship Diagrams

## 5.7.1 *Object Icons*

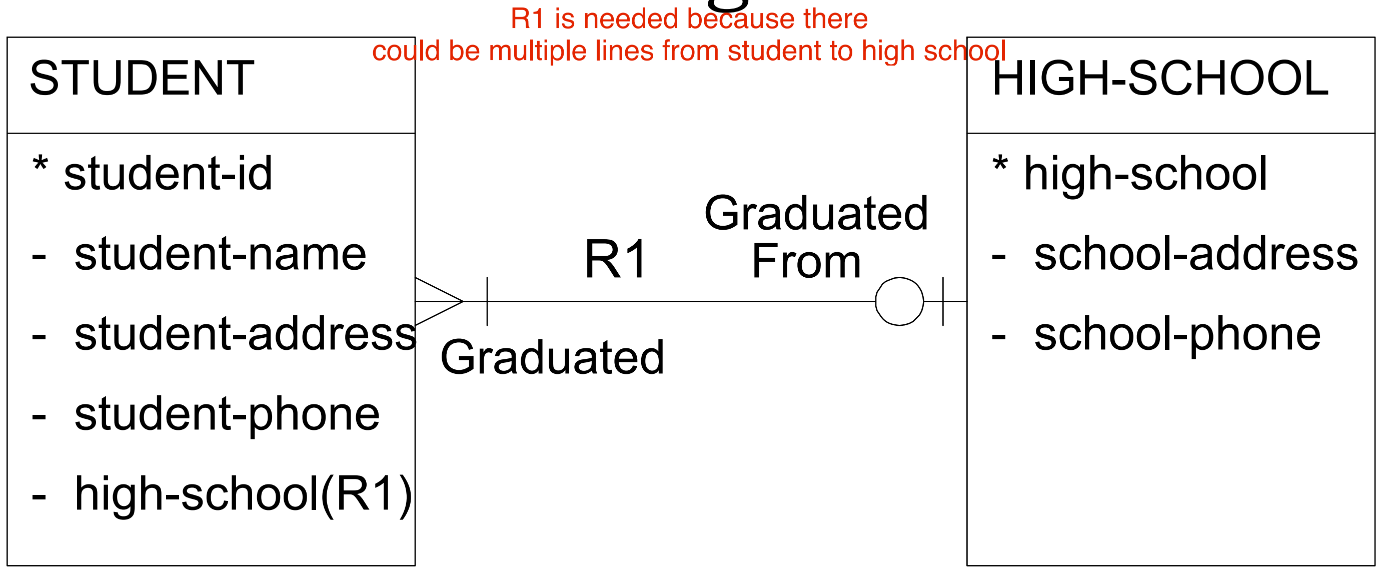
Let’s examine an example of an Entity/Object Relationship Diagram (ERD or ORD).



Note that the attribute on which the records are sorted is called the **primary key** of the entity, and are labelled with a ‘*’.

## 5.8 Relationships

An example of a relationship line is that between a student and a high school.



These are the referential routes that can be traversed at run time by the application code to find other related data.

– E.g. The address of the high school a student went to.

Note the high school attribute in the student class is a **foreign key** which provides the information needed to traverse relationship R1.

A foreign key is a **value- or pointer-based reference** to a particular, related instance.

- E.g. particular high school.

Value-based foreign keys refer to the primary key of the other related (i.e. foreign) object.

ORDs provide a **map** showing **all possible** 'routes' over which the application can navigate around the data!

Question: Given a student object, how does the application code find out details of what high school she went to?

Answer: Look in the High School attribute of that student.

Question: How can application code find which students went to a particular high school?

Answer: Search the student objects and select **all** students that went to that particular high school.

You can sometimes during requirements analysis be alerted to a relationship when seeing **possessive grammar** used. E.g.

- The student’s high school, and
- The high school’s students.

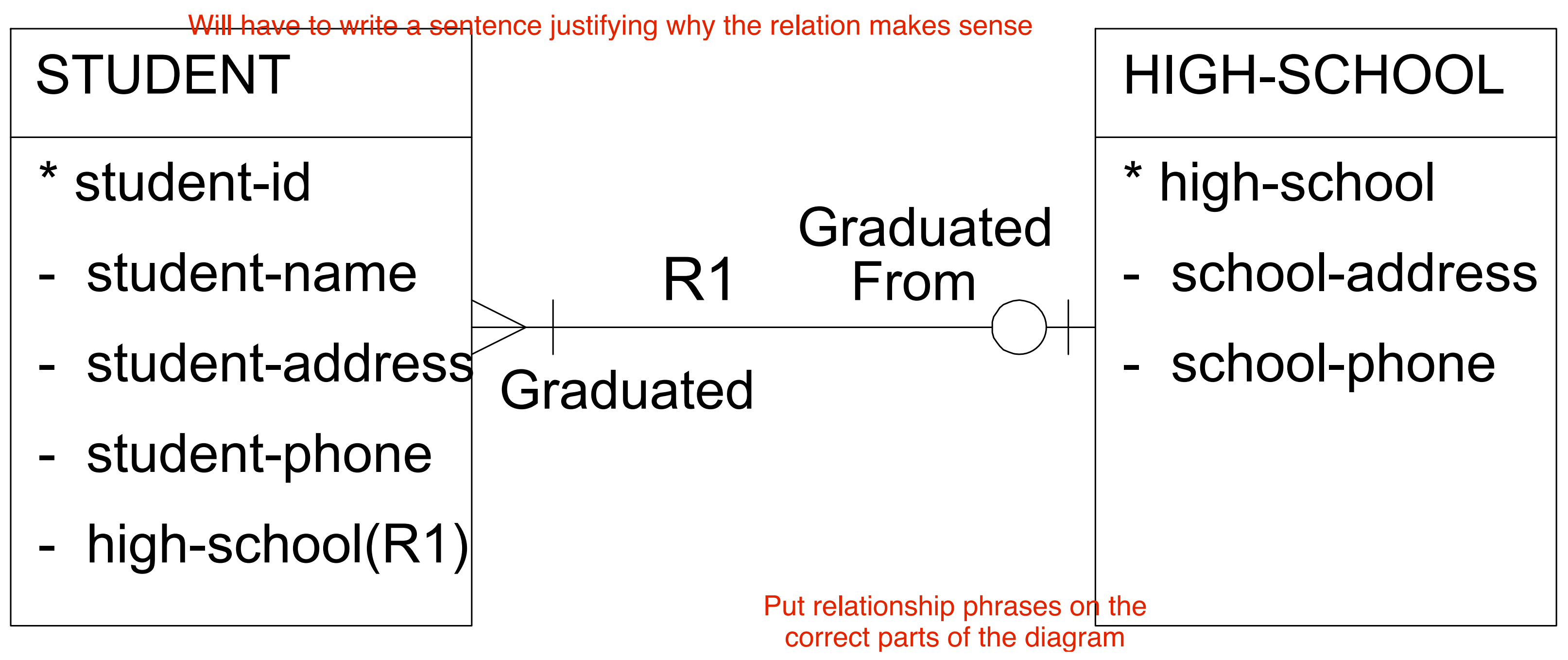
*Careful though as possessiveness is alternately sometimes just an indicator of an attribute (e.g. Student’s name)!*

The details of a relationship provide information necessary for the design phase. The details include:

- multiplicity,
- optionality,
- relationship sparseness, and
- traversal frequencies.

In Cmpt 276, we'll only cover **multiplicity and optionality (which together we call cardinality**).

- The others are topics for an advanced course.

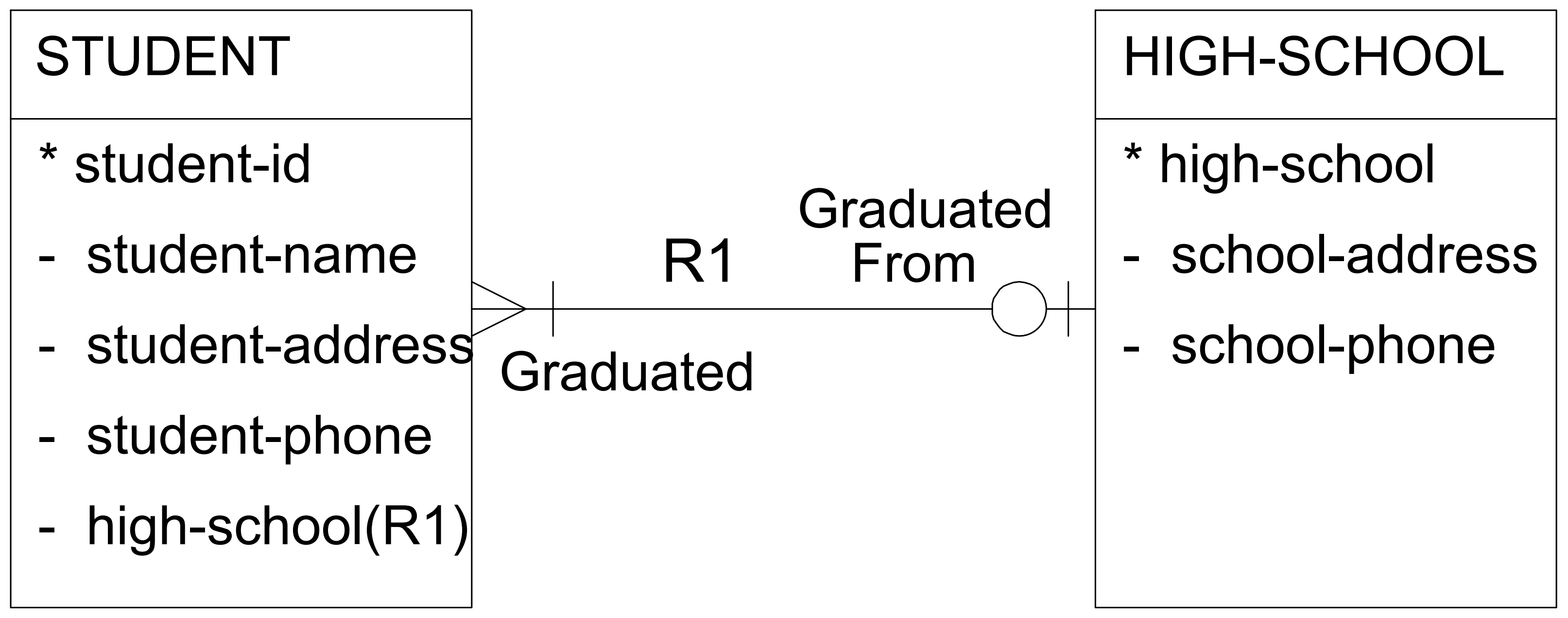


A student can graduate from only one high school. But a high school has likely graduated many students. The last sentence is indicative of a 1-to-many (1:M) **multiplicity**.

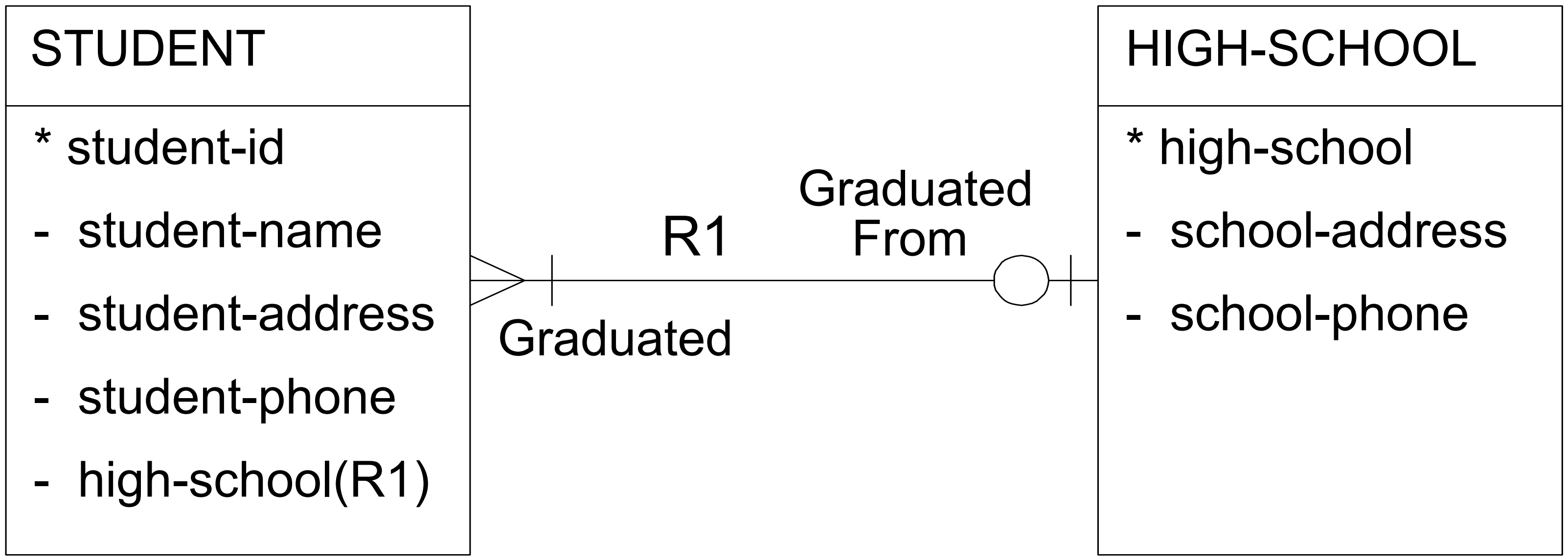
- You will see shortly, that both the multiplicity and optionality of the high school-to-student relationship is important to database design.

The two objects in the ORD are joined by a line indicating the ‘graduated-from/graduated’ relationship. Relationships are always two-way:

- The student **Graduated From** the high school
- The high school **Graduated** the student

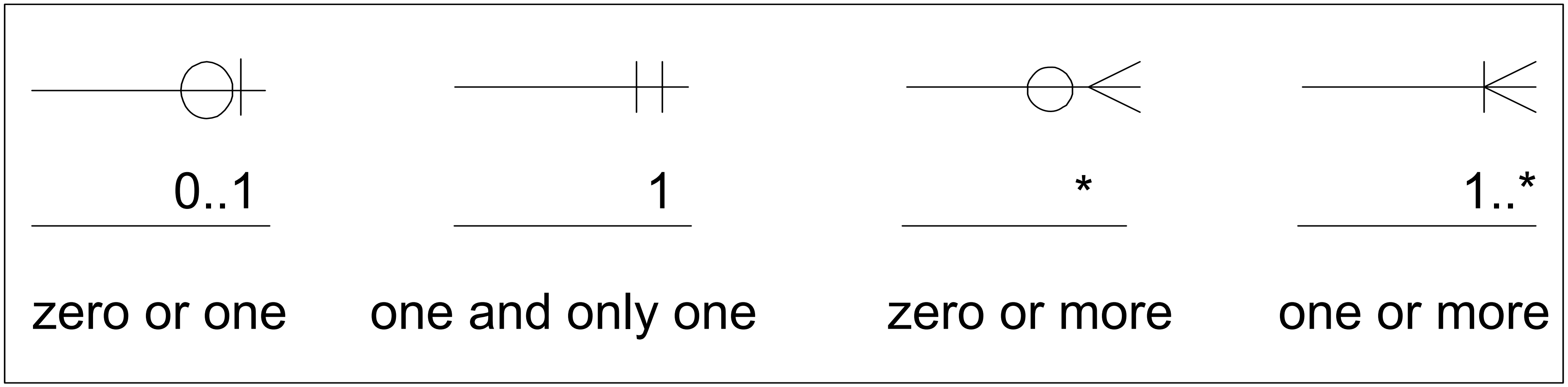


The relationship phrases are by convention usually put near the “end” of the relationship line for the direction that version of the relationship name applies.



Relationship lines have **cardinality symbols** or more modernly, UML number ranges on them.

We can use either convention.



In the old form, the symbols closest to the center of the line portrait the **optionality** as either a 0 or 1. If 0, that means that some students in our database may never have graduated from a high school, and thus don’t have any relationship with a high school. Non-graduates might be:

- mature students let in by special permission, or
- have finished their high school qualification by attending a college high-school-equivalency program.

In that latter case, the high-school attribute of such a student would be blank or null.

On the other hand, maybe we should insist that all university students first have high school equivalency, and just allow colleges to be listed in the high school file.

These policies are called ‘**BUSINESS RULES**’.

The cardinality is not determined from some magic database theory, but from actually asking an application area specialist/customer what the case is: optional (0) or mandatory (1)?

Remember, cardinality is a *combination* of optionality and multiplicity. I.e. For each end,

Memorize this

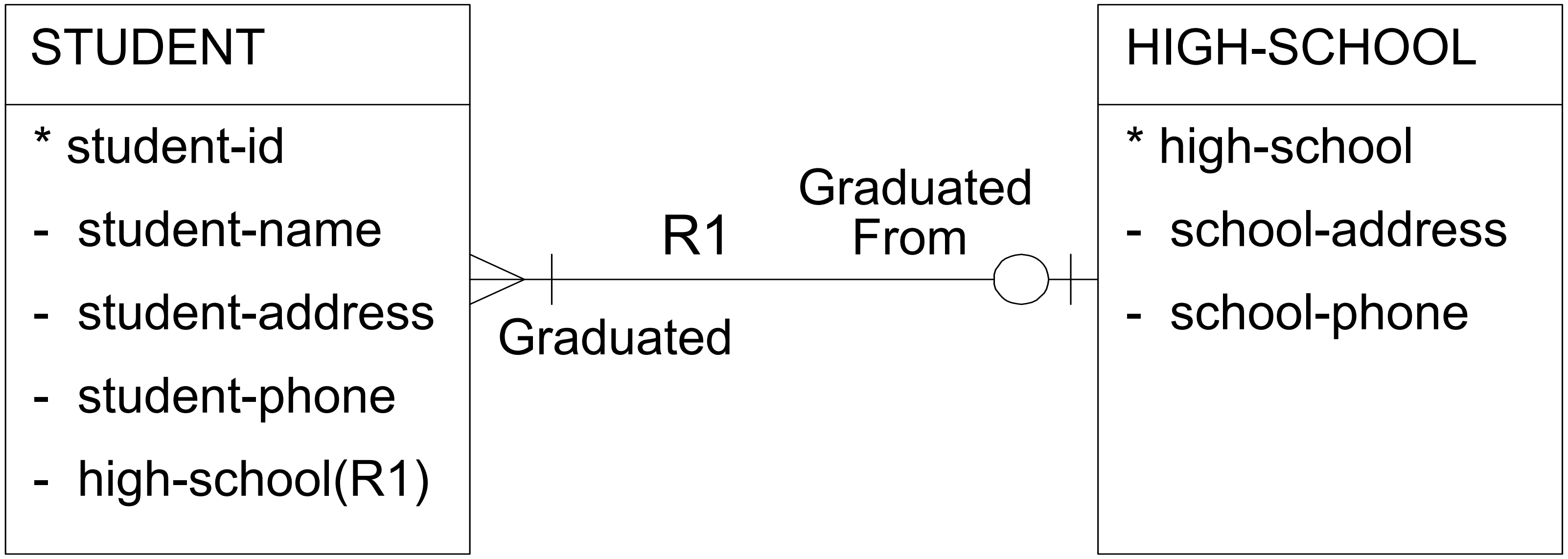
**cardinality = optionality + multiplicity**

So you'll also have to determine the optionality for the other direction:

- Is it possible to have a high school in the high school file which has never sent a student to SFU?
  - Perhaps it's a new high school who's never graduated a student yet. Would we want such a high school in SFU's database of high schools. Perhaps in anticipation?

You also, through research or interviews, must determine the <u>**multiplicity**</u> for each direction.

The multiplicity symbols are located at both extreme ends of the relationship line.



The multiplicity symbol is usually either:

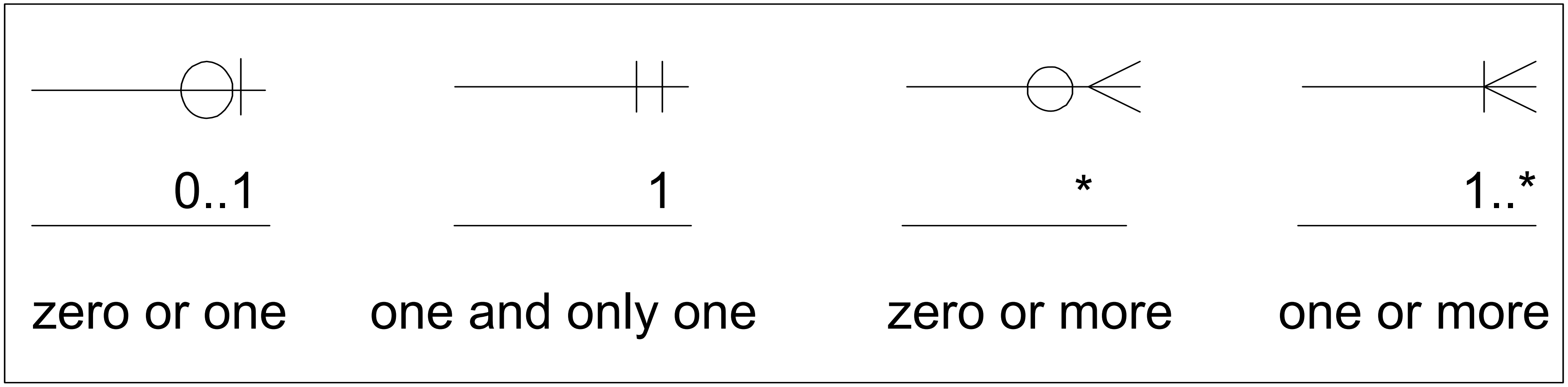
- A vertical bar, or in UML a max range of 1.
- Or ⋖ , or in UML a “*”.  (Or less commonly the top of a number range.)

This instructor calls the ⋖ a “crow’s foot” as it looks like a bird’s foot.

It indicates that a high school could have (be related to) more than one student instance.

An important database design factor is whether, for a **single** particular (high school) instance at **one end** of a relationship, does there exist multiple related (student) instances of the other entity.  This must be determined by the analyst and documented on the ORD.

There are thus 4 possible cardinalities for one end.  And 4 for the other end.



(Actually in UML there are other possible max numbers.)

**Important note**: Just because a relationship is optional, or multiple, in one direction doesn't mean it is in the other. **The two directions are entirely separate.**

- They must be carefully researched in both directions (what questions would you ask of the customer?).

The fact that an object icon exists on an ORD means that at least one instance can exist.

- The question is, given its existence, how many instances of another **related** class of entity can there be that are related to the first?

An important aspect of relationships is that they **must be recorded** (stored, remembered) somehow.

- Adding a foreign key is one way.

Important advice: **To determine the cardinality of both ends requires 4 questions to the customer!**

- Is the left end optionality 0 or 1?
- Is the left end multiplicity 1 or * ?
- Is the right end optionality 0 or 1?
- Is the right end multiplicity 1 or * ?

Story: One semester my Cmpt 370 students did terrible on cardinality requirements in the project. The next time I taught the course, I insisted they write a sentence in application domain language answering each of those 4 questions. Everybody did great!

NOTE: You'll sometimes see the term "one-to-many". That means the multiplicity is one on (say) the left and many on the right.

In contrast, "many-to-many" is where there's a crowfoot (or *) on both ends!

You must justify why your relationship can be the only one which can be used to model this

## 5.9 Normalization

The main goal of normalization is modifying the data model to get: This will be tested on the midterm

- fixed length records (for better performance),
- with little redundancy, and
- with few optional attribute fields (to save space).

Recall **`struct`**s in RAM are just fixed length records, too.

- However, it’s easier and faster in RAM to have an attribute of a structure point at a linked RAM list to implement the effect of a variable length record.

Nonetheless, learning how to implement a system without variable length records will **teach you a lot about your data**.

In a familiar application, normalization is easily done.

- It doesn’t even have to depend on weird relational algebra from a database course.

However, in an unfamiliar application area it’s more difficult.

Normalization is an important method to distill the huge pile of info we get during requirements gathering.

Before beginning normalization, determine the dependencies and cardinalities by asking about the business rules.  E.g.

- Each airliner has a licence number painted on its tail.
- When storing information about a particular flight, is the flight number or licence number the key?
- Can a particular plane be used on different flight numbers? Yes
- Can a flight have many aircraft licence numbers?  Maybe, but only on different days.

Let’s say our database will need to hold a bunch of info about:

- students,
- the courses and semesters, and
- instructors.

(This is a easy example as it is familiar to you, but a poor example as the relationships and normalizations are easy since you’re familiar with the application subject area.).

## Assume the following attributes must be retained about a student.

stud-id

stud-name

address

phone

course

credit

semester

grade

course-room

instructor

instructor-office

Exam question:
difference between unnormalized and first normal form?
First normal form represents a table which has additional columns which has redundant information
Unnormalized form has repeated information
Difference: remove repeating groups by making each record contain more information

Repeated for each time a particular course is attempted

Repeated for each course taken.

We gathered this by interviews, looking at the university calendar, course timetable, telereg instructions, forms, examples of transcripts, and watching the current system in operation.

In addition to knowing the attributes, we have picked up an understanding of some of the business rules.

- Students take courses.
- Students typically take more than one course in their life.
- Students can fail courses, and can repeat the same course later in a different semester. Students can thus take the same course more than once.
- Each time a student takes a course, they are assigned a grade.

Notice I have underlined some **nouns** and **cardinality** info.

– This is a sometimes helpful technique.

## *5.9.1 First Normal Form*

The above set of attributes are in unnormalized form.

- They're in unnormalized form as the records for different students NOT the same length.
- This is an unacceptable situation, as it will make fast disk record access and fast insert/deletes impossible.
- And waste lots of space due to disk fragmentation.

The solution is to (unfortunately) add redundancy, by creating a separate row **for each occurrence of a student taking a course**.

(Note: we will later be able to remove this redundancy, but it is a necessary starting step).

**CLUE**: The data that needs to be retained has repeating groups.

PROCEDURE: Remove repeating groups by adding extra rows to hold the repeated attributes.

FIRST NORMAL FORM (1NF): Tables should have no repeating groups.

The result is a single table *with a compound (i.e. multi-attribute) primary key*.

Note below that to select/specify a row you need to specify all three of: the student ID, the course, and semester (since students can repeat a course for a different grade).

## <u>1NF STUDENT TABLE</u>

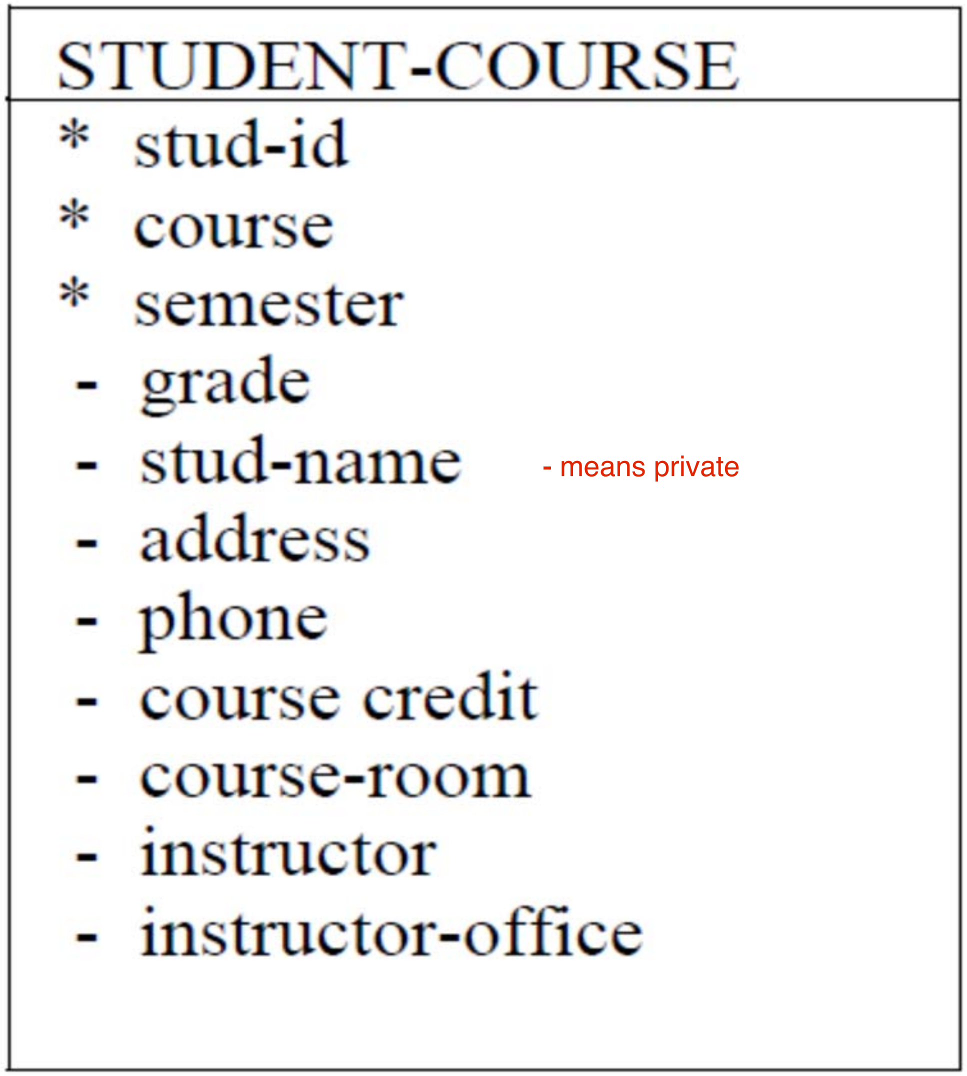
| <u>student-id</u> | <u>course</u> | <u>semester</u> | grade | stud-name |
|---|---|---|---|---|
| 93010-1234 | Cmpt 105 | 95-3 | B | John Smith |
| 94444-9999 | Cmpt 201 | 95-3 | D | Bill Jones |
| 94444-9999 | Cmpt 201 | 96-1 | C+ | Bill Jones |

Most significant key is to the very left of the table

I have underlined the columns that make up the compound primary key.

I've put the 3 attributes that make up the primary key on the left, and sorted the rows according to the resultant compound key.

## This in effect creates a single entity class with a large number of attributes:



- means private

Storing the same information many times is not a good idea

Parts of the primary key are denoted with "*".

So now we have fixed length records!

## But, there are several problems with this data model:

- There's **too much redundancy**.
  - If you think of each entity instance as a row in a table, we'll be storing both a student's id, name, address, and phone number each time he enrolls in a course. This is a ridiculous!
    - We need to instead have a separate file of students and the *attributes that depend on only the stud-id*.
- There are **insert anomalies** that can occur.
  - It's impossible to admit a student to the university and enter him in this table, unless he has enrolled in a course!

You do not need the three key attributes in order to be inserted to the table

- This is because the table uses a compound key composed of three different attribute fields.

Memorize the different types of anomalies

- None of these three fields can be null, as they are the data we ask the database access software (ISAM B+ tree) to use in its search.

• There are **delete anomalies** that can occur.
  - If this is the only file that stores which room a course is held in, in a particular semester, then if you deleted the row containing the only student who had registered in the course so far, you would also delete the only record of which classroom that course will be take place in.

• There are **update problems** which can occur.
  - Because of the redundant data, to change a woman student's name because she just got married, you

have to change it in *all the rows corresponding to every instance of every course* she had ever taken! .

- Though marriage is not that frequent an occurrence for students (compared to the frequency with which they take courses), this is nonetheless a computationally burdensome task.

– And though many women keep their maiden name, the university must be able to respond to those who do change their names, no matter how many or few occur, because this is provincial law and outside of the university’s jurisdiction to control.

– So, this is a <u>business rule</u> that has been imposed on the university’s business.

– The analyst must consider ‘external’ rules that are elicited during customer requirements interviews.

## 5.9.2 *Second Normal Form*

**CLUE**: The problem in 1NF is non-key attributes which depend on (i.e. are functions of) **only part of** the compound key. E.g.

- Address is a function of only the stud-id, and
- credit is a function of only the course, not of the course + student + semester.

Ask professor to explain what he means here.
Ask for an example

Note: If the 1NF key is not compound, there cannot be partial key dependencies, and the table will *already be in 2NF*!

-

**PROCEDURE:** **Remove partial key dependencies**. Break the 1NF table into several tables such that in each table, each non-key attribute is dependent on only the (lesser)primary key of that table.

Remember partial key dependencies

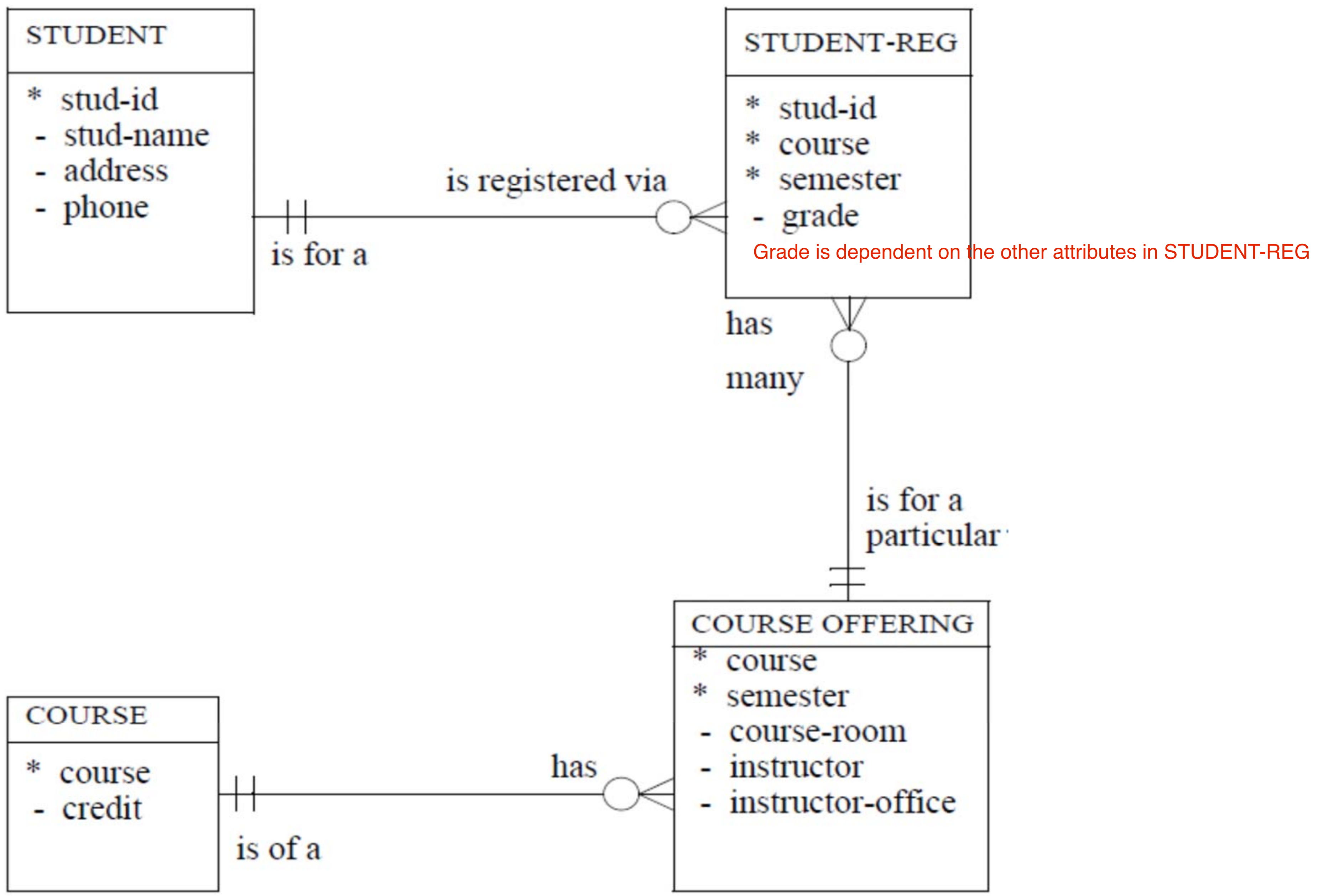
**SECOND NORMAL FORM: There are no non-key attributes with partial key dependencies in any table**.

## For the university application, the business rules will tell us something about these dependencies.

- stud-name, address, and phone number are a function of only the stud-id. Thus we can create a student file of only this information.
- credit is a function of only the course, and is independent of which semester it is offered in, and which student is taking it. We therefore must separate the concept of a ‘course’ and that of a ‘course offering’ in a particular semester.
- course-room, course-instructor, and instructor office is a function of only the course and semester, and is independent of which student is taking it.
- only course grade is necessarily a function of all three parts of the original primary key.

To get the university application into 2NF, we separate the data into 4 files!

## Second Normal Form ERD:



That was quite a big leap.

- We could have first pulled out the stuff that was just dependent only on course and semester.
- This would have resulted in two entities, student and course.
- Then we would have still found partial key dependencies in each of those.
- This is quite common. When you do break a file up, make sure you look again for partial key dependencies in each, as it is often easier to see even more that you might not have noticed in the big original 1NF organization!

In going to 2NF we also had to ask questions about the cardinalities to show on the resultant ORD.

- These cardinalities did not come from the normalization process.
  - We had to specifically understand or ask about the application domain rules.
  - I have shown reasonable assumptions for the (business) rules regarding cardinality in a university application.

## With the data model in 2NF, notice the following:

- Now far less redundancy in the organization.
  - A student's name doesn't have to be stored with each of his many registrations, and the course credit needn't be stored for every course offering.
- A student doesn't have to be registered in a course for the student to be initially admitted to the university.
- A course, and the room that a course offering will be located in, can be entered even when there are no students registered in that offering yet.
- Most of the update problems have been solved.

## 5.10 Third Normal Form

There's still problems with the 2NF data model.

- The course instructor's office is stored for every occurrence of a course offering.
  - If a professor teaches 4 courses over a year, why do we need to stored his office 4 times?.
  - Surely, offices are only a function of the prof's name, not his *name* + *course teaching assignments*! This causes insert/delete/update anomalies.

What  if we consider the case where we have sessional professors?
Their office hours may change depending on the available classrooms they can access

**CLUE**: The problem in our 2NF is a class with non-key attributes that depend on (i.e. are functions of) other **non**-key attributes.

- Instructor-office is a function of the instructor name, and not both of, or either of, course + semester!

Remember this definition

The office of an instructor depends on the instructor, but neither is a key

**PROCEDURE: Remove non-key dependencies.** Split the table so that the functional dependency is enumerated in its own separate table.

**THIRD NORMAL FORM: Every table is in 2NF and additionally, there are no non-key table attributes with dependencies on other non-key attributes.**

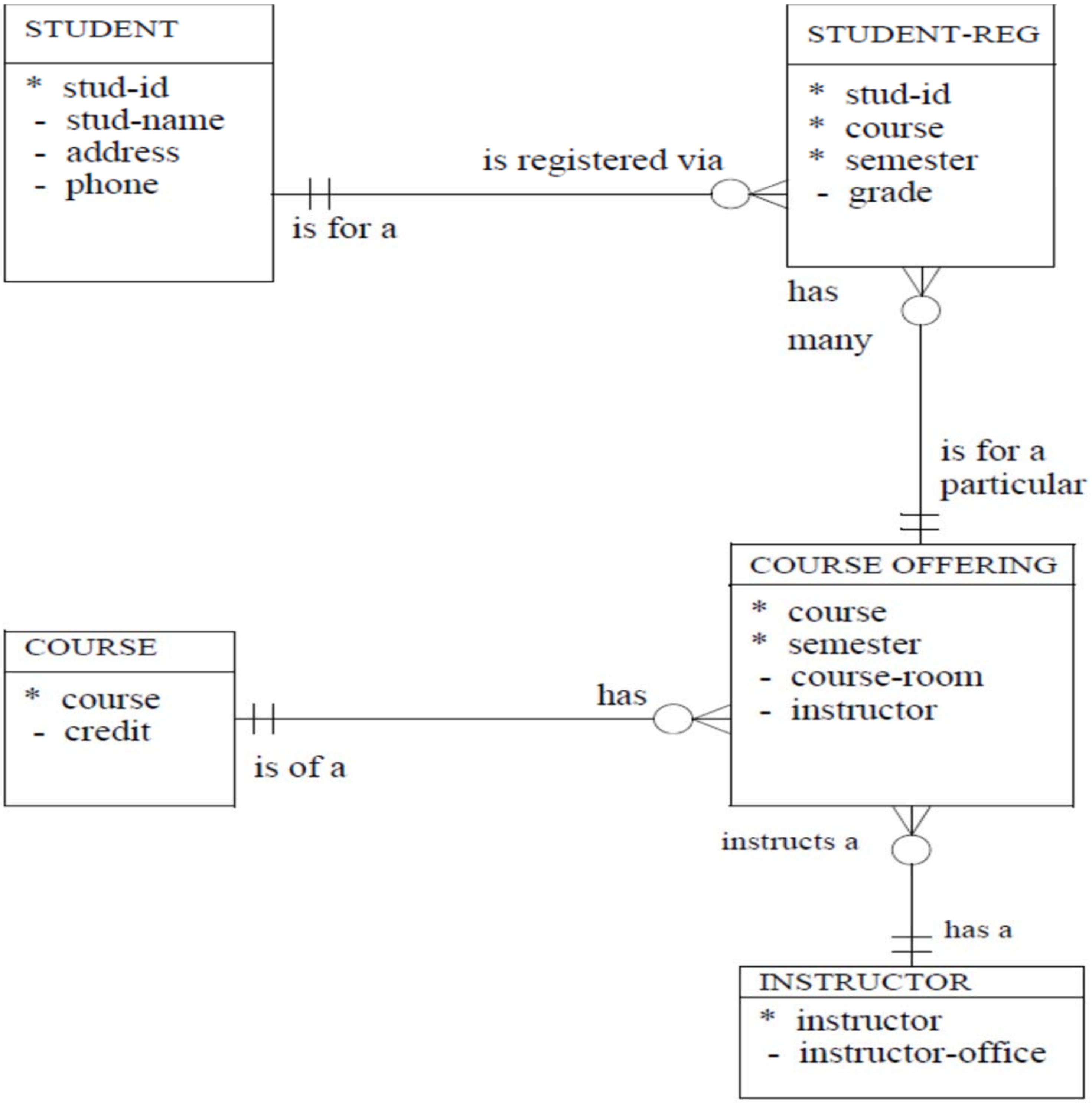
(Exception:  Dependencies on columns which are also ‘candidate’ keys are allowed; this is a subtle issue covered in database courses.)

The one object class in the previous ORD that does have a non-key attribute with a dependency on another non-key, non-candidate-key field is instructor-office.

- Instructor-office is purely a function of only the instructor's name.
- We can't move instructor and instructor-office to the course file, as a course can have several instructors teach it over the span of several semesters, or even during one semester.

The solution is illustrated on the next diagram.

**Third Normal Form ERD:**


STUDENT
* stud-id
- stud-name
- address
- phone
is registered via
is for a
STUDENT-REG
* stud-id
* course
* semester
- grade
has
many
is for a
particular
COURSE OFFERING
* course
* semester
- course-room
- instructor
COURSE
* course
- credit
has
is of a
instructs a
has a
INSTRUCTOR
* instructor
- instructor-office

Note that each time we go to a higher normal form, things get more and more *logically organized*!

Question: What if a course is offered twice in the same semester?

Question: Will making instructor a part of course offering's key solve this? Answer: No. Why?

If you make the instructor part of the key, would it not allow you to search for that particular offering?
It would not since the same professor can teach two different sections. You would not be able to distinguish the sections from each other with the new key

## 5.10.1 Normalization Summary

In normalization, we are seeking to make sure that attributes depend:

- on the key (1NF)
- on the whole key (2NF), and
- on nothing but the key (3NF).

There are higher levels of normal form, but you will only be expected to go into 3NF

When fully normalized:

- all records are fixed length,
- no insert/delete anomalies to mess up the recording of certain information,
- no update anomalies to be constantly handled,
- little redundancy (other than needed to handle the above factors).

Re-organizing data into a higher normal form usually, but not always, saves space and improves overall performance. You can study more about this in a database course. You will also study even higher forms of normalization (e.g. 5NF).

Exam question:
If there are 100 courses and 27 instructors, what would take up more space, 2NF or 3NF?

## 5.11 Formalization

Some relationships are optional.

E.g. For an application that records all persons and marriages, we must efficiently store data indicating *whether* a particular person is married or not, and if so *to whom*).

**Formalization** is determining the appropriate storage representation needed for an application to 'remember' that a particular relationship between two instances actually *exists*, and if so between *which* instances.

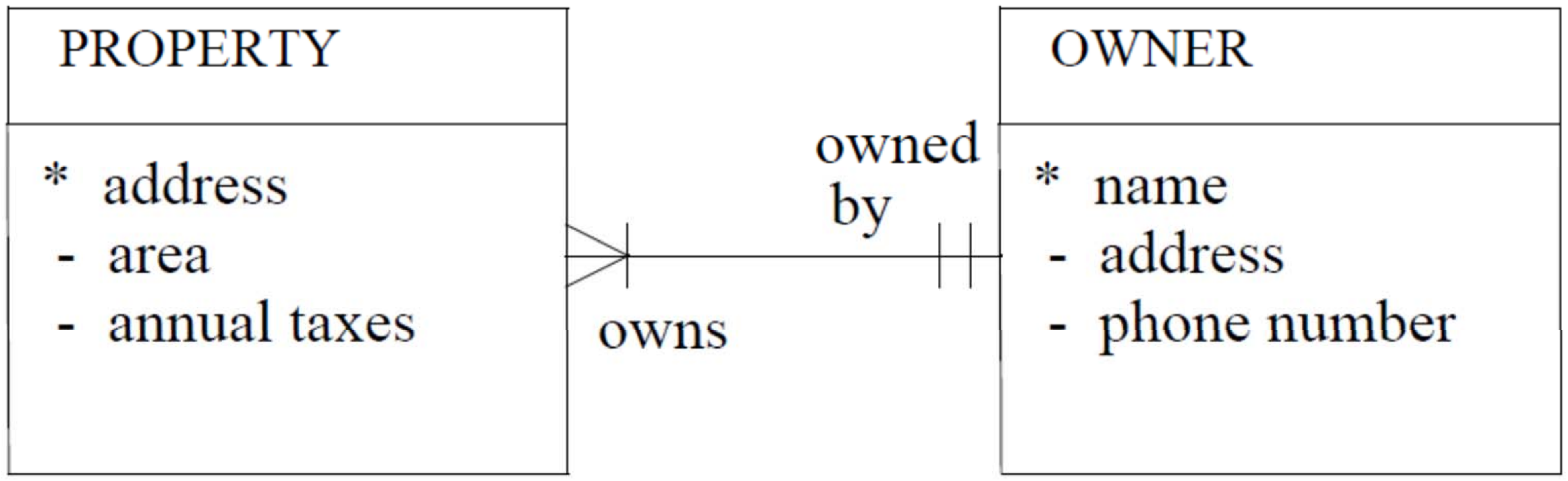
Read more about formalization

Often, the process of normalization will automatically accomplish formalization for you.

In large projects with multiple programmers it’s sometimes not clear which attributes of a record are foreign keys.

- And which ones are for which relationships leaving the entity.
- Sometimes between project team members there are homonyms and synonyms that confuse.

E.g. Here's an ORD for properties and property owners. Assume a person may own several properties.



It's a nice diagram but doesn't provide for the **storage of the relationships** regarding which properties are owned by which owners. This is what is meant by 'formalization'.

## 5.11.1 Foreign Keys

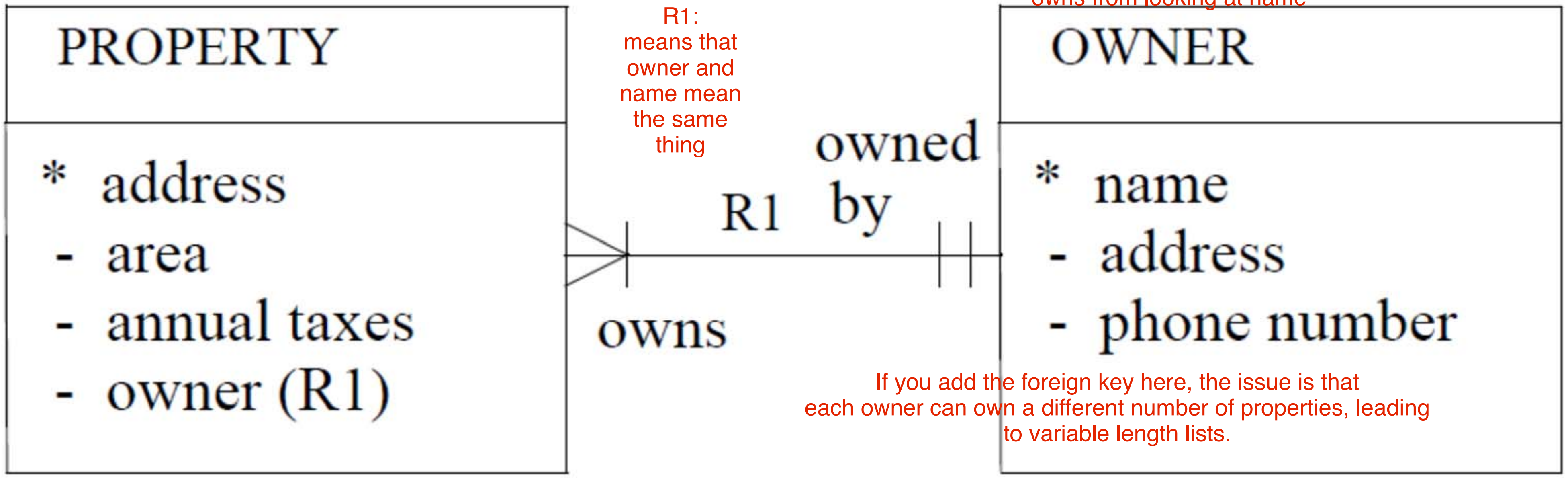
To formalize relationships you typically have/add a foreign key to one end of each relationship according to the following rules:

- If the *multiplicity* on one end of the relationship line is “many”, and on the other end “one”, then put the foreign key in the “many” end.
- If the *multiplicity* on both ends of the relationship line is both “one”, then put the foreign key in the optional end.  If both ends are optional, then put the foreign key in either end.
- If the multiplicity on the two ends of the relationship line is both “many”, the you’ll need to form a new object called an ‘association’.

Does the client need to delete/add employees?

In the above case, we use the first rule.

Can find all the property a person owns from looking at name



R1: means that owner and name mean the same thing

If you add the foreign key here, the issue is that each owner can own a different number of properties, leading to variable length lists.

Find the owner associated with this property by looking at the foreign key “owner”

I have added an attribute called “owner” to the ‘many’ end of the above relationship. This attribute, when stored on disk as part of each property entity, **records/stores** the relationship!

In addition, I’ve given the relationship line a designator (R1), and written this designator beside the foreign key attribute.

- Prevents misunderstanding since PROPERTY “owner” and the OWNER’s “name” attributes are ***synonyms***.
- They are two names for the same thing.  I.e.
  - the value of an owner foreign key attribute in a PROPERTY entity **is** the name of the property owner as recorded in the primary key of the OWNER entity.
  - By this I am saying that owner is a *reference* to that property’s owner’s OWNER record.

We can use foreign keys to navigate the ORD for application operations. E.g. Given the address of a property, I can find the owner's phone number by:

- first, searching the PROPERTY file for a PROPERTY record whose address attribute value equals the address of the property.
- second, in that record, get the owner's name by looking at the character string stored in the owner attribute,
- third, look in the PROPERTY_OWNER file for a record whose name = that string.
- finally, look at that PROPERTY_OWNER's record to get the correct phone number.

An ORD was a *referential map*.

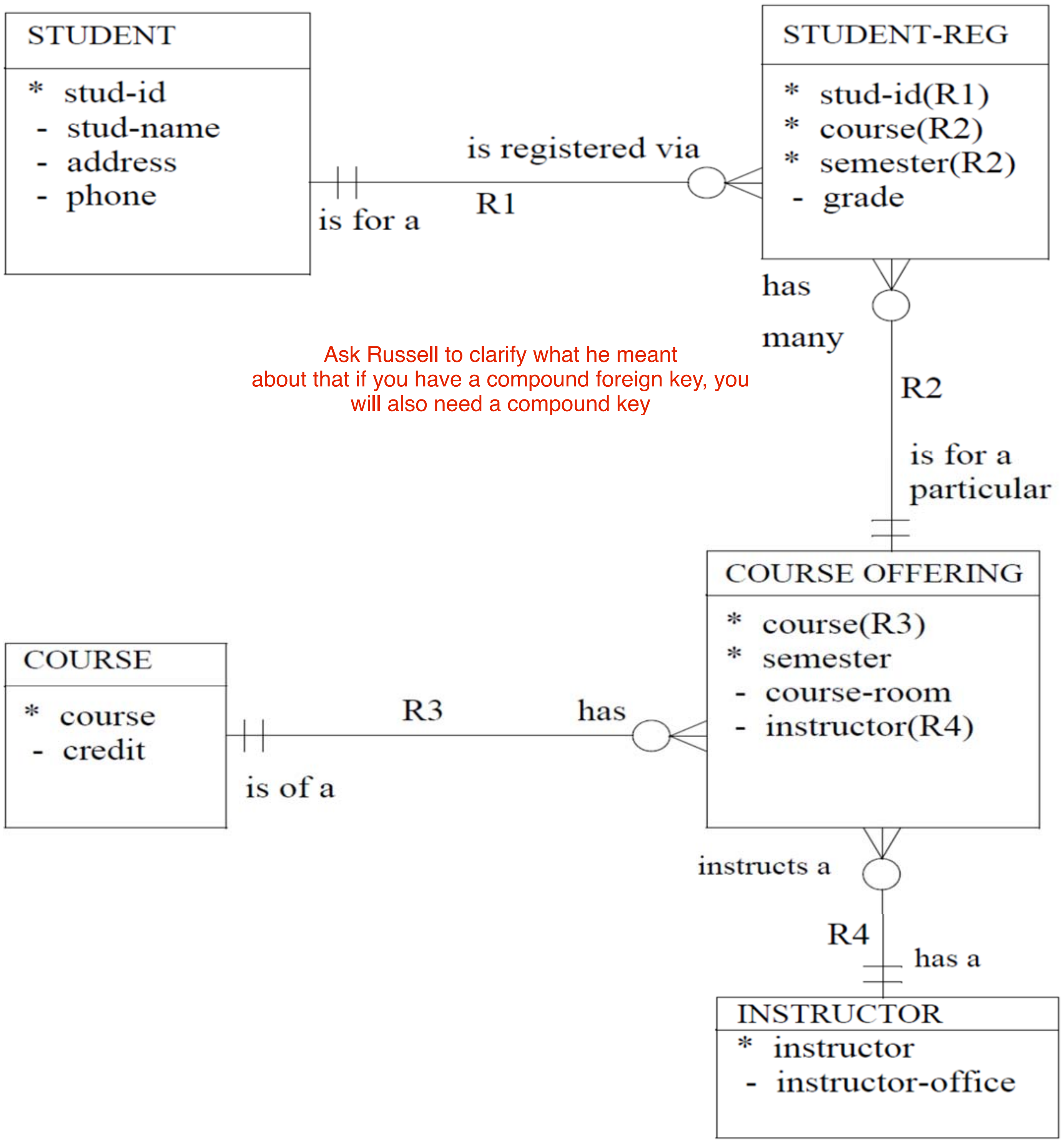
- It shows you all the possible journey directions you can take when referring from one object instance to any related ones.

I can also **travel a relationship the other way**. Given an owner, I can search the property file for all (i.e. the set of) properties owned by a particular owner.

Note: If we'd wrongly added the foreign key to the wrong end, the property owner records would have repeated groups since an owner can own any number of properties.

- Clearly this leads to variable length records.
- The best thing is to, as suggested above, add the foreign key to the relationship end with the higher multiplicity.

Here is the student registration system with all the foreign keys identified:



Course and semester is a compound foreign key

Ask Russell to clarify what he meant about that if you have a compound foreign key, you will also need a compound key

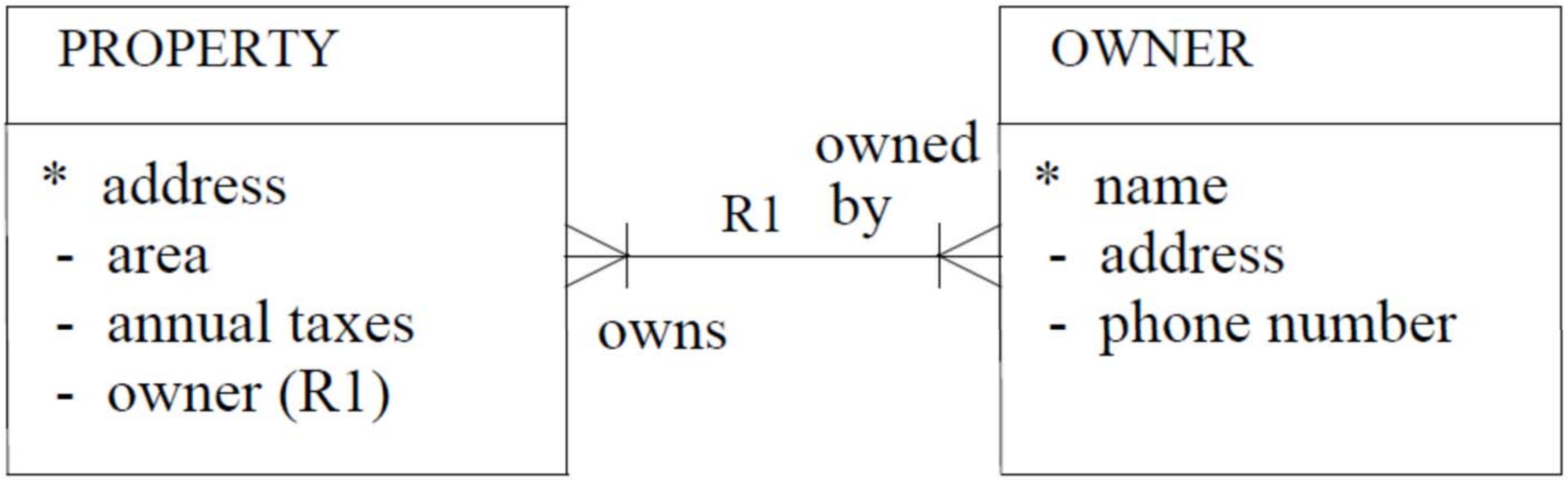
### *5.11.2 Many-To-Many Associations*

When the multiplicity is ‘many-to-many’, simply adding a foreign key to one, nor even both ends, is *not enough to formalize the relationship*.

Instead, a **whole new object class** called an “association” is needed to keep fixed length records.

Continuing with our property registration system, let’s now assume that we have recently been told that we have to adjust our system to handle the business rule that a property may be owned by several owners (i.e. partners).

Previously:

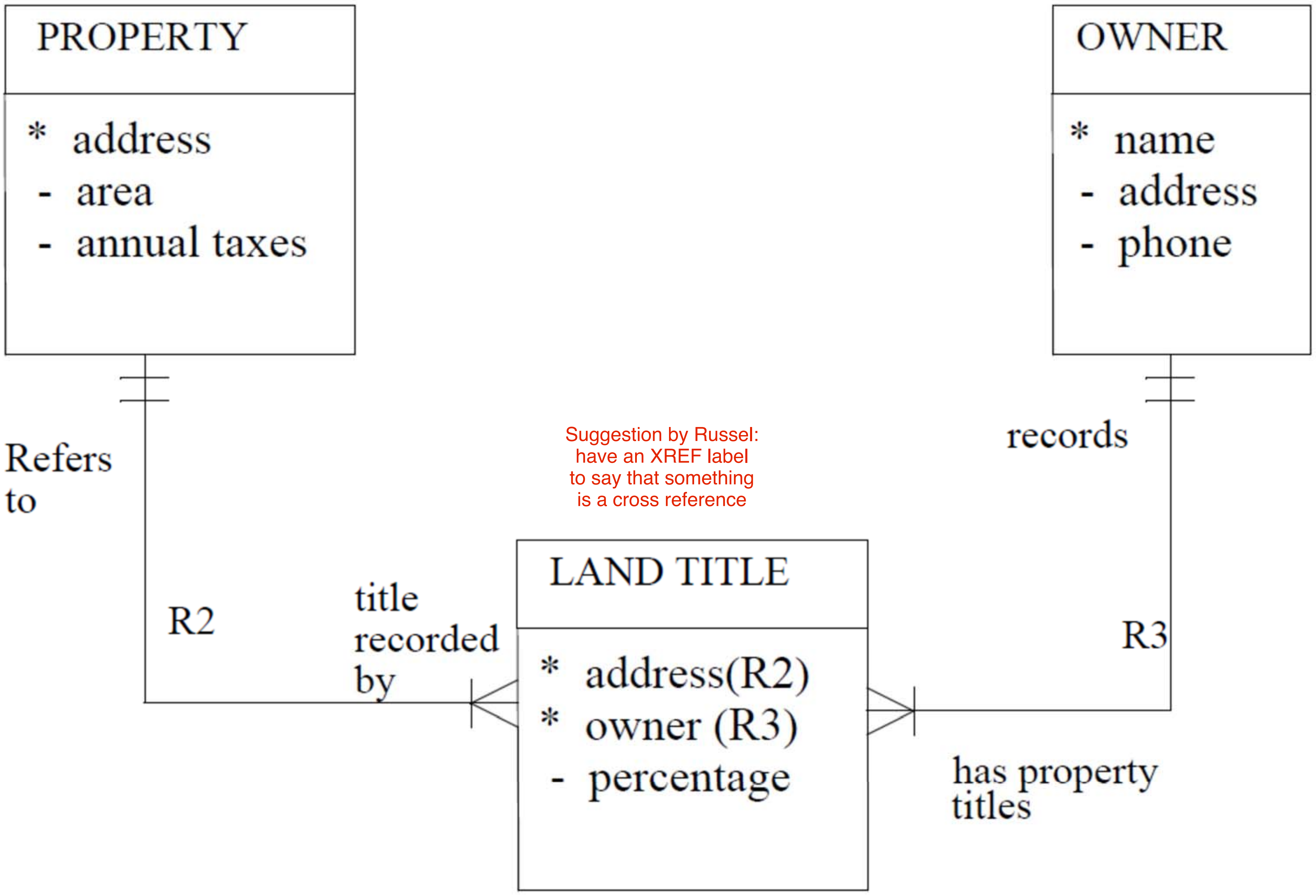


But we now need a PROPERTY to have several owner attributes (i.e. repeated group, ---> variable length records).

There's no way to formalize this with a single foreign key. Or with a foreign key in both ends!

We must create a new associative object to store the information about which properties are owned by which and how many owners.

- The need for this becomes even more obvious when you consider that you should also store what percentage of each property each owner owns.



Suggestion by Russel:
have an XREF label
to say that something
is a cross reference

The new object will have as a primary key **at least** the **UNION** of the primary keys of the original two objects! This means the primary key of the associative object will be **compound**!

This allows all possible pairs of PROPERTY instances and OWNER instances to be recorded via the association.

| address | owner | percentage |
|---|---|---|
| 123 9th Ave. | Smith, Bill | 49.5 |
| 123 9th Ave. | Jones, Jane | 50.5 |
| 500 First St. | Able, Jim | 100 |
| 999 3rd Ave. | Able, Jim | 100 |

Here we see that such an association can both:

- record that 123  9th Ave. is owned by two partners, and that
- Jim Able can own two different properties.

**Note**:  *We know the address attribute in the LAND TITLE entity is the address of a property and not of an owner, because the analyst annotated the address attribute to indicate it is a foreign key which formalizes R2.*

Thus in addition to clarifying some synonym problems, annotating foreign keys can also clarify some **homonym** problems.

Often the normalization process will generate associative objects automatically for you, as in the university student database (can you find the association?).

Other times you do it yourself as part of the design phase.

And you may have to manually resolve synonym and homonym problems.

Note that an association might not be an application domain object.

- It might just be a creation of the *implementation* needed by the application.
- Or, sometimes it's a physical object in the application domain.
  - An example would be a property deed, which is a legal document stating property ownership.

Our formalization recommendations may not be optimum (memory or performance-wise) for relationships which are:

- very sparse (not many instances participate in the relationship), or

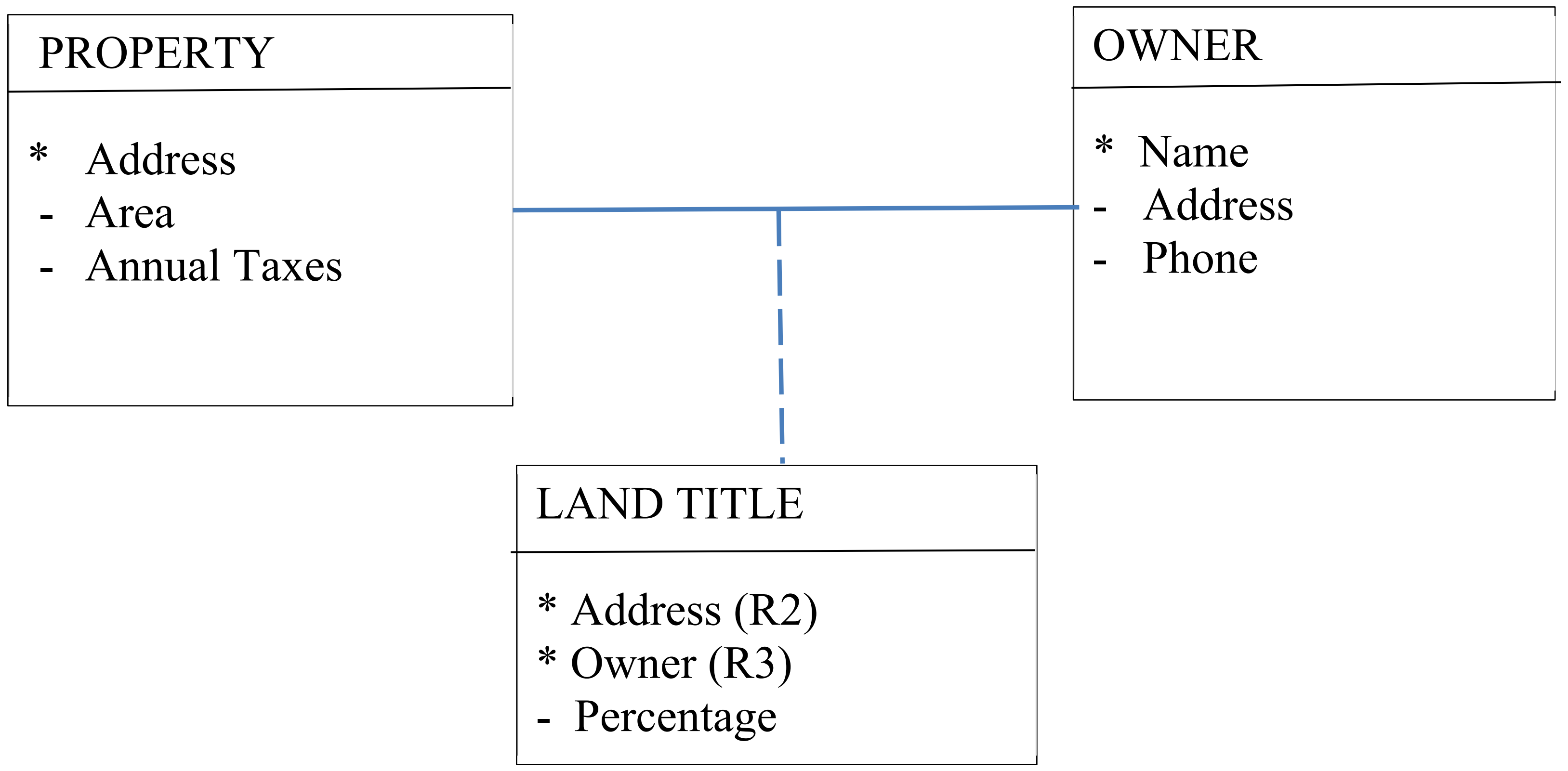
Ask Russell about this

Know what an association class is:
it joins two objects in a many to many relationship

- that are traversed during execution very frequently or infrequently.

## 5.11.3 UML Association Class

Sometimes an association class is denoted this way.



But I don’t think it’s as clear.

## 5.12 Appendix A: Why Organize Data Properly?

Information systems retain **persistent data**.

- I.e. Data retained overnight, between program executions, or over a power shutdown.

The problems of information systems are:

- storing huge amounts of inter-related data,
- for long periods of time, and
- rapidly creating outputs composed from the retained data, or
- to be able to easily and rapidly modify particular data records!

**As we’ve seen, learning how to organize persistent data also provides insights on to how to organize even our non-persistent (RAM) objects!**

Hard disk drives have access times of approximately 0.010 seconds (10 ms).

This is a million times slower than the instruction time of processors!

To obtain tolerably fast random access times to disk data requires very special techniques to be used in data bases.

- Invariably, this requires that retained records in a file all be the same length.

With fixed length records, to read the 1000th record, you simply seek to 999 x Record_Size from the beginning of the file.

- This avoids reading all the records earlier in the file.
- It also means that when a record is deleted, then another one added, the added one can fit in the same length space as the one deleted!
  - If the added record were shorter, some space would be wasted.
  - If it was longer, it wouldn't fit and would have to be placed elsewhere and the entire space from the deletion would be wasted until a smaller record needed to be added.

With proper analysis you can construct a data model. You get:

- fixed length records.
- no wasted space due to empty or duplicated data.
- fast random and sequential access.
- fast insert and delete operations (why not possible with simple sorted records?)

Should it not be possible?

## *5.12.1 <u>B+ Trees</u>*

This sub-section gives a wonderful introduction to basic database implementation via index trees and links.

Once you have fixed length records, fast random and sequential access is achieved by writing or purchasing database software that uses the Indexed Sequential Access Method (ISAM).

Not expected to remember this

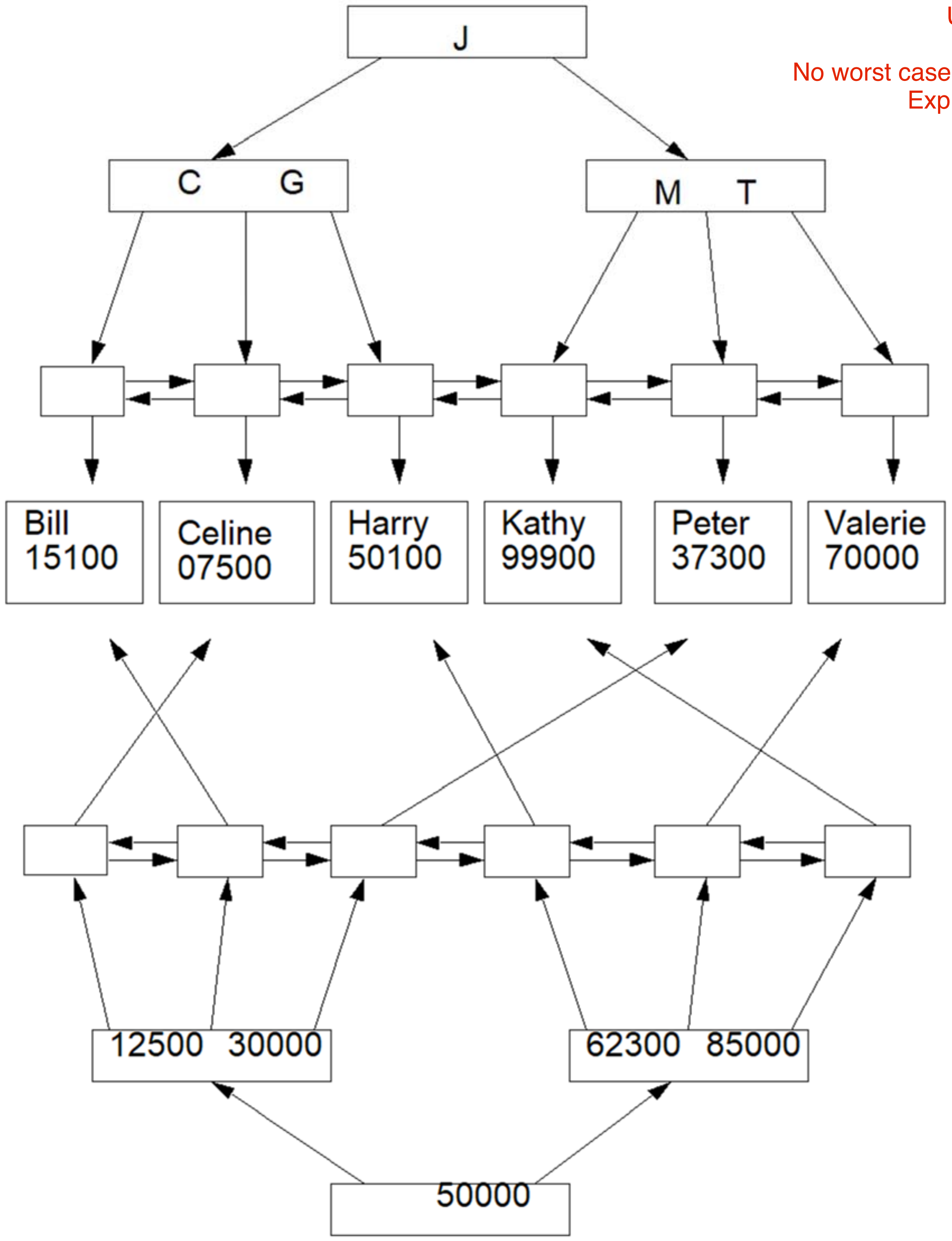
ISAM is based on B+ tree index structures on disk.

- This special kind of tree is very short in height, thus requiring very few disk accesses to traverse down to any randomly-chosen leaf.
- The tree is always balanced so that no branches are very deep.
- The use of a tree makes random disk searches, inserts, and deletes as fast as is possible, which is very important as disks are orders of magnitude slower than RAM.

- Additionally, the leaves are sequentially accessible via a linear, doubly-linked (disk) list. The sequential links make access of the `next' and `previous' items in the sorted structure very fast.

Here is a diagram of a database accessible via 2 different B+ trees, one for each of two different search keys.

Unsorted database
never delete files
No worst case runtime for database operations
Explain language features



Everything above the data is stored in an index file (research more about this)

Here are 3 advantages of B+ trees:

- B+ trees have an upper tree part which allows fast (O(log N)) random access.
  - Additionally, the nodes in the tree part are a full disk sector in size (512 bytes).
    - Results in a high-order tree (many arrows coming out of each node).
    - This makes the tree height short, to reduce the number of disk accesses. :<)
- In B+ trees (not to be confused with B trees or binary trees), the *leaves* of the tree are special nodes that are doubly-linked to additionally provide rapid sequential access to the `next' or `previous' record. :<)

- Third, the data is neither in the nodes or leaves of the tree.
  - The nodes and leaves are in a special, separate “index” file.
  - The leaf nodes contain the forward and backward links, and a record or byte number of the particular record over in the data file.
    - The advantage of this is you can have a single data file of persons as shown above, and TWO index files/trees pointing into it.
    - One index file is for fast access by name.
    - The second index tree is for fast access by driver's licence number!

## This is the fundamental disk data structure underlying almost every database management system!

When modelling inheritance, you can have either a square be a type of rectangle or vice versa. It depends on how you model the scenario