Identifying Entities

Houston we have a problem

One of the tricky topics to fit into this module has been the topic of database design. This has been tricky for a number of reasons.

Leaving database design this late in the work in my view gives out the message that somehow this aspect of the development is somehow less important than the other parts of the system.

This is in part due to the fact that this is a new module and the material next year will be already written thus available more promptly. In part it is also where the text books tend to place database design within this part of the development life cycle.

Also the way that we have split our analysis of the system is that we have pretty much moved from one layer of the system to the other. We started thinking about the presentation layer by drawing up use cases, next we designed class diagrams to plan our class library it makes sense to finish with designing the data layer using our Entity Relationship Diagrams (ERDs).

We really could however do with more time thinking about the subtleties of the design of the data layer rather than treating it as an after thought before pressing on to the development stage.

Document Standards Chen v UML

There is also another more important problem with designing databases with UML.

In my personal opinion database design was something of an afterthought as the UML was developed.

Entity Relationship Modelling dates back to 1976 when Peter Chen published a paper on the subject. He identified a system of documentation and a set of rules that really help in analysing and designing our entities.

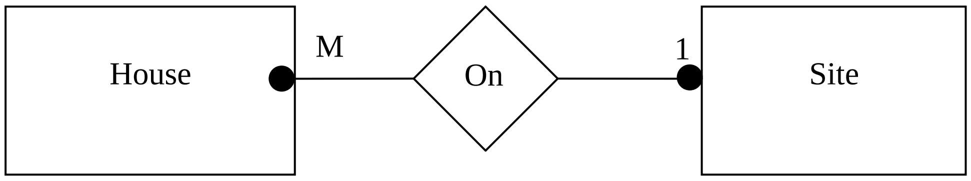
Twenty years later UML comes along to try and rationalise the documentation standards available at the time and produce more standardisation.

This creates three problems, firstly we have multiple documentation standards for modelling entities, secondly I am never personally convinced that the UML standards are as effective in modelling databases and lastly some of you on the first year will have been taught one approach and not the other.

There are three documentation standards currently to be aware of.

The Chen Notation

An example of the Chen notation is as follows.

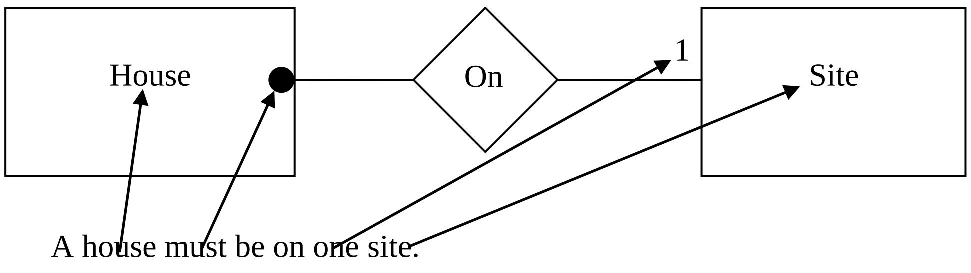


In this example we are stating that a house is on one site.
A house must be on a site.
A site may or may not have many houses.

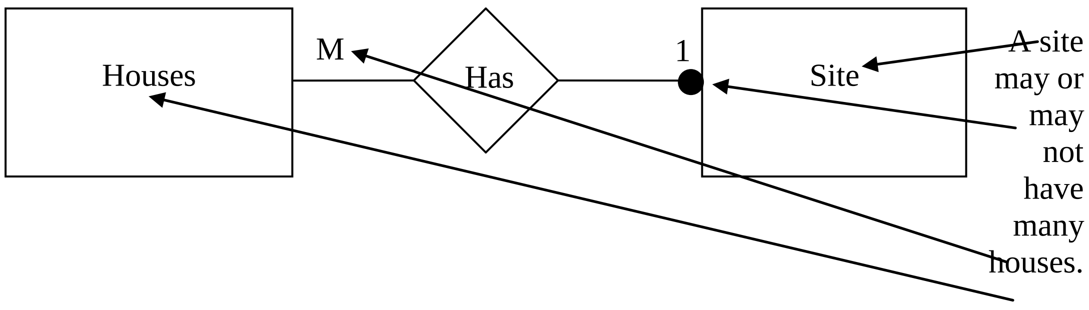
This form of notation is a powerful tool in describing the relationship of one entity to another. The main difficulty people new to the notation find is how to read and write in this way.

The dots indicate the membership class of related entities.

They are read like so…



And the return journey…



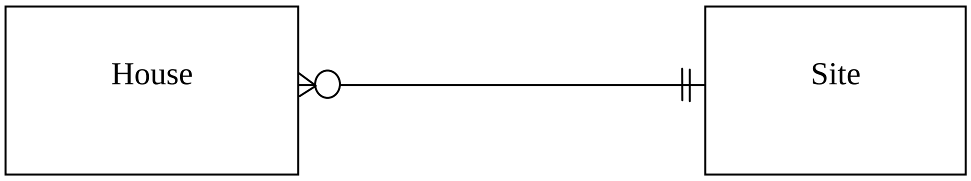
This seems pretty straightforward however when new to the notation there is a tendency to get the structure confused.

The Crows Foot Notation

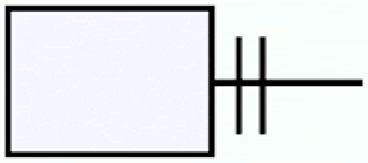
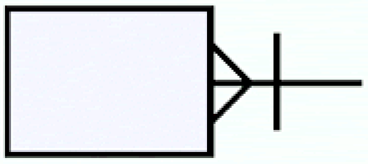
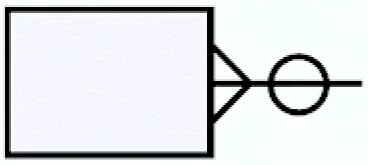
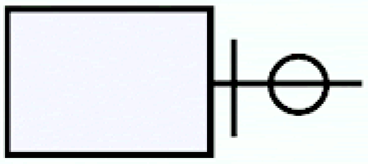
The Crows Foot Notation is not dissimilar to the Chen Notation however it doesn't allow you to specify the membership class of associated entities.

(The membership class is represented in Chen by the dots bolted on to the entities—●)

Here our house - site relationship would be represented as follows.



The degrees of association are represented like so.

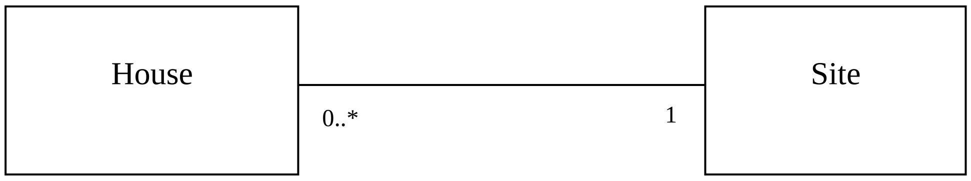
SYMBOL	MEANING	UML REPRESENTATION
	One and only one	1
	One or many	1..*
	Zero, or one, or many	0..*
	Zero, or one	0..1

UML Notation

The last notation style you might come across is the UML notation.

This is pretty much the same as the notation we use for describing classes.

Here our house - site relationship would be represented as follows.



Possibly easier to read than the crows foot notation however it also lacks the opportunity to specify membership class as per the Chen notation.

Which is best?

To some degree it comes down to familiarity and personal preference.

It also comes down to what standard you are told to use and in our case which standard the software we are using i.e. Enterprise Architect “encourages” us to use.

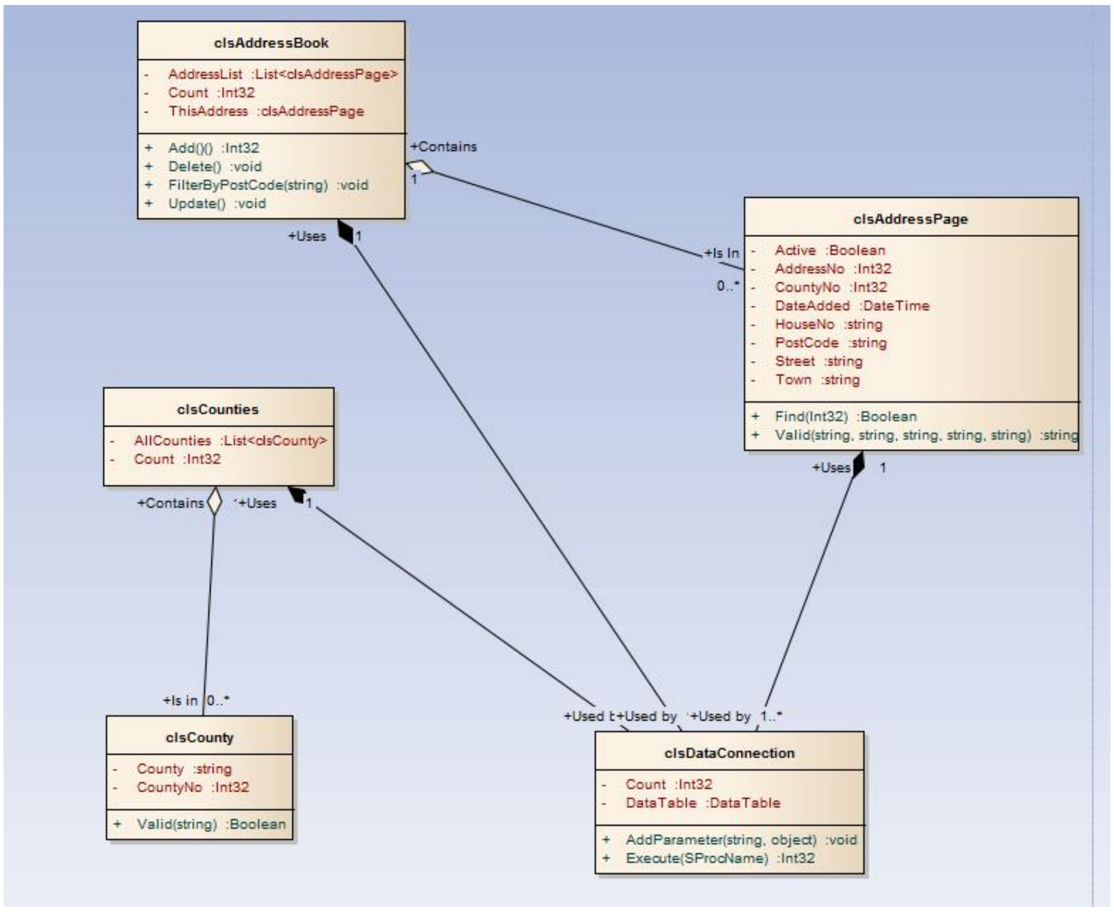
For this module we will be using the Crows Foot notation for the main reason that it is supported by Enterprise Architect.

This brings us on to another point of confusion and misunderstanding in designing our database.

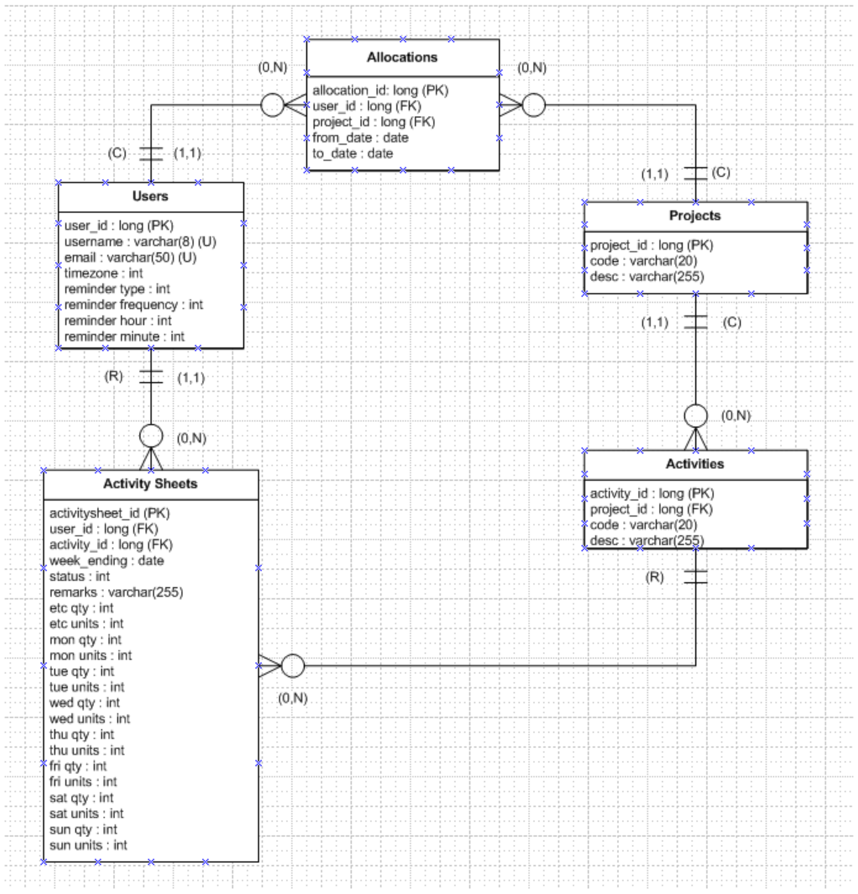
What is the Difference between a Class and an Entity?

It really doesn’t help much when we have two documentation standards that look so similar.

The class diagram…



The ER Diagram…



The rectangular boxes are inevitably going to look pretty similar until the subtleties are fully appreciated.

So what are the differences?

Classes do not persist

Classes are ultimately implemented as objects.

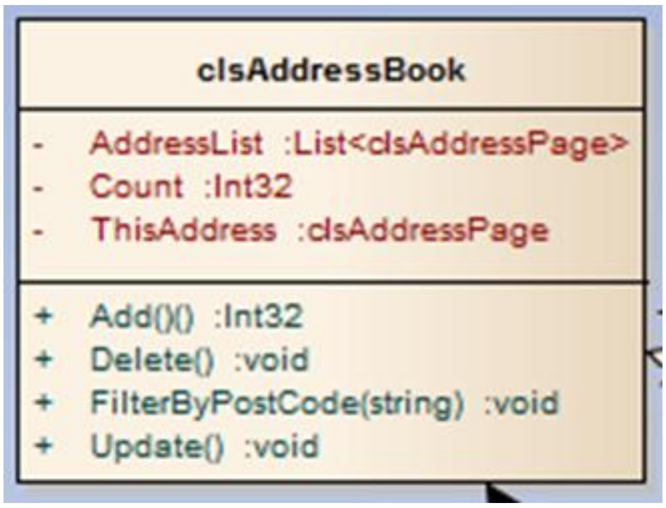
Objects exist in RAM for the duration of the function or program where they have scope. When the function or program ends the object ceases to exist.

Entities are ultimately implemented as tables.

Tables exist in the database persist and do not vanish once the program/function has completed.

Classes have Functionality Entities do not

When we draw up a class in EA we will give it attributes and operations.


clsAddressBook
- AddressList :List<clsAddressPage>
- Count :Int32
- ThisAddress :clsAddressPage
+ Add()() :Int32
+ Delete() :void
+ FilterByPostCode(string) :void
+ Update() :void

When an object is created using the class these become its methods and properties.

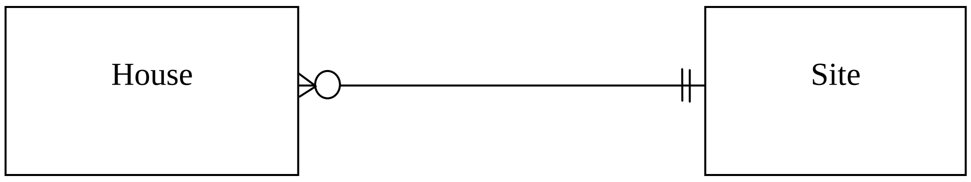
For example…

```
clsAddressBook MyAddressBook = new clsAddressBook();
MyAddressBook.FilterByPostCode("LE1");
```

The methods implement the functionality for that object.

In the case of entities we often don't even bother with identifying the attributes, it's often just a rectangle with the entity name inserted.

For example…



As we make more use of Enterprise Architect we will add the attributes to the entities so that we may use EA's auto code generation features.

The entities will ultimately by implemented as tables in the database with the attributes implemented as the columns.

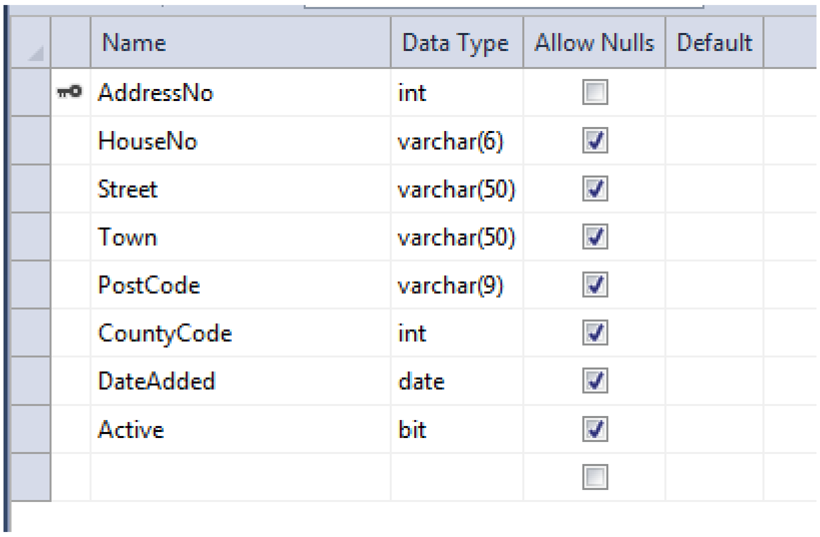
The point is that the tables lack the functionality of objects. An entity will may have attributes but it won't ever have operations that become methods.

"But really – aren't they basically the same thing?"

At this level it may appear that way.

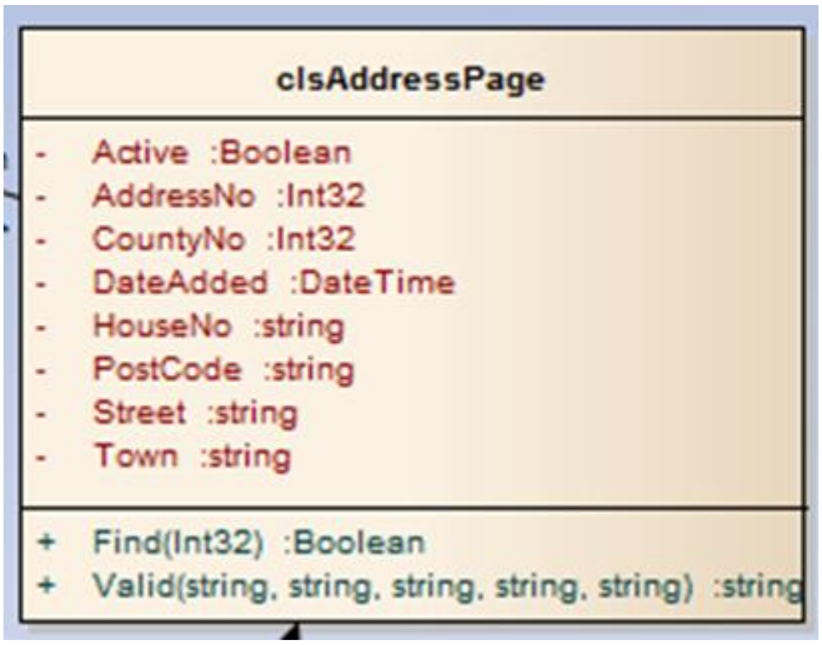
For example we might look at the following class and table definitions.

Out table tblAddress looks something like this…



	Name	Data Type	Allow Nulls	Default
🔑	AddressNo	int	☐	
	HouseNo	varchar(6)	☑	
	Street	varchar(50)	☑	
	Town	varchar(50)	☑	
	PostCode	varchar(9)	☑	
	CountyCode	int	☑	
	DateAdded	date	☑	
	Active	bit	☑	
			☐	

Whilst the class definition for clsAddressPage looks like this…



Are they not basically the same?

Well, at this level of work they probably are.

What happens is that each table ends up being wrapped in an object so that we may manage the rows in that table.

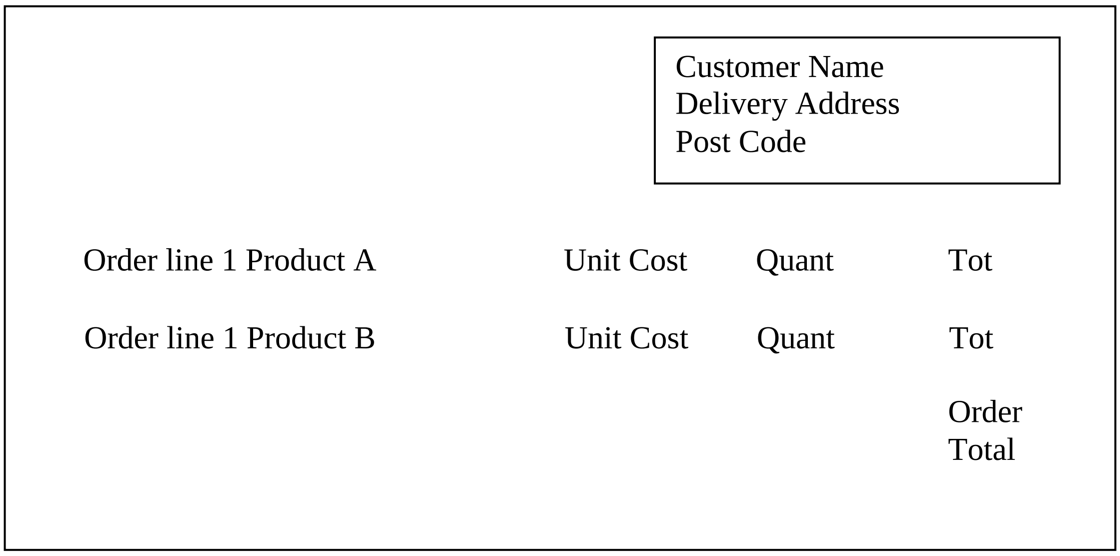
At this level there is a strong correlation between the structure of the class and the structure of the table.

Assuming a one to one mapping between classes and tables is often a good starting point for informing both the design of the classes and the entities.

It is important to remember that this relationship does not always hold true.

A good example of this might be a class to model an invoice.

The order may take the following format…



Customer Name
Delivery Address
Post Code

Order line 1 Product A	Unit Cost	Quant	Tot
Order line 1 Product B	Unit Cost	Quant	Tot
			Order Total

We could have a class defined in the system called clsInvoice giving us access to the features of the invoice.

The data for the class would be derived from multiple tables.

For example
Customer details would come from tblCustomer
Product details would come from tblProduct
There may be a stock check facility making use of tblStock

Some elements of the invoice would never ever be stored in the system.

For example
Order line totals and the invoice total would be calculated on the fly.

When the invoice is stored in the system the data contained in a single object would be distributed across multiple tables.

Which brings is nicely onto the next topic.

Normalisation/De-Normalisation

I imagine that you will have encountered the idea of normalisation already.

Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy. Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them. The objective is to isolate data so that *additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database using the defined relationships.*

(Wikipedea)

Things to Consider when Designing your Tables

<u>There must be no row order significance:</u>

shelf	**product**	**price**
A	butter	89
	lard	37
B	bread	62
	milk	32

In this example we are insisting that the rows must be sorted as above otherwise we won’t know what shelf a product is on.

<u>No column order significance:</u>

student#	name		
0427h	smith	david	ivan

As above we are assuming that the columns must always be in this order otherwise we won’t know the person’s name.

<u>Each attribute value must have only a single value:</u>

car	**make**	**engine-size**
astra	vauxhall	1100,1300

No duplication:

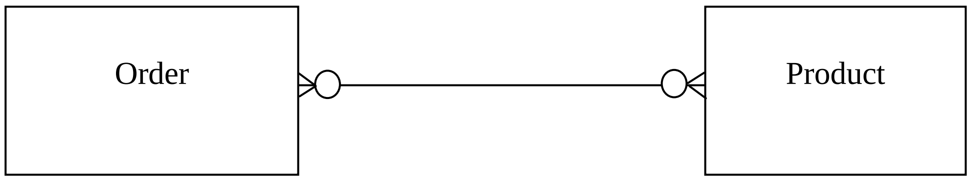
box#	**contents**	**colour**
31	sugar	brown
47	flour	white
9	rice	white
47	flour	white
103	sugar	white

As few null values as possible: (!)

vehicle	**wheels**
car	4
hovercraft	

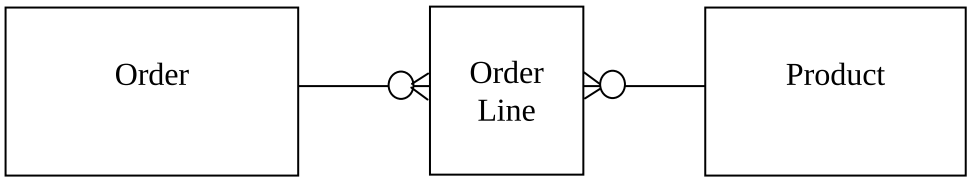
Null values may arise because they are not known or not applicable. (Or is our design incorrect?)

***Always* Decompose Many to Many Relationships**



Many to many relationships are normally a sign that you have missed an important entity.

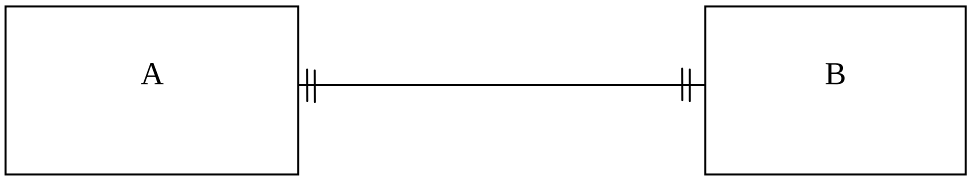
You may not know what the missing entity is but they decompose along the following lines…



One to One Relationships

Do not exist (probably).

As rule if you end up with the following in your ERD…



You can be reasonably sure that you have misunderstood something.

(They do exist but they are very rare!)

De-Normalisation

One last word on the subject of normalisation!

Often when teaching database design this subject gets forgotten.

The truth is that if you create a perfectly normalised database it may actually get in the way of efficiency.

There are a number of measures of a "good" system. One of those measures is how the data is organised. Another measure for example is user response times.

It is possible to create a system that is fully normalised but then need to de-normalise it to some extent in order to increase how fast the system works.

There are other places in your design where you will find clues as to the nature of your entities and their attributes.

- Smoke and mirrors prototype
- Class diagram
- Sequence diagram