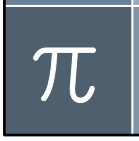
# Structuring Data

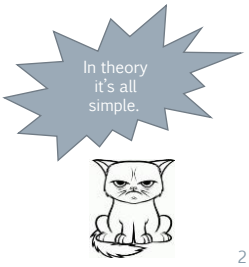
## Relational Modelling, Part II
## Relations and Relationships




This is the second part of three in our discussion of relational modelling.

# Entity Relationship Design

- Steps to build a conceptual design
  1. Identify the entity types
  2. Identify and associate attributes with the entity types
  3. Identify the relationship types
  4. Determine cardinality and participation constraints
  5. Determine primary and foreign keys
  6. Validate the model



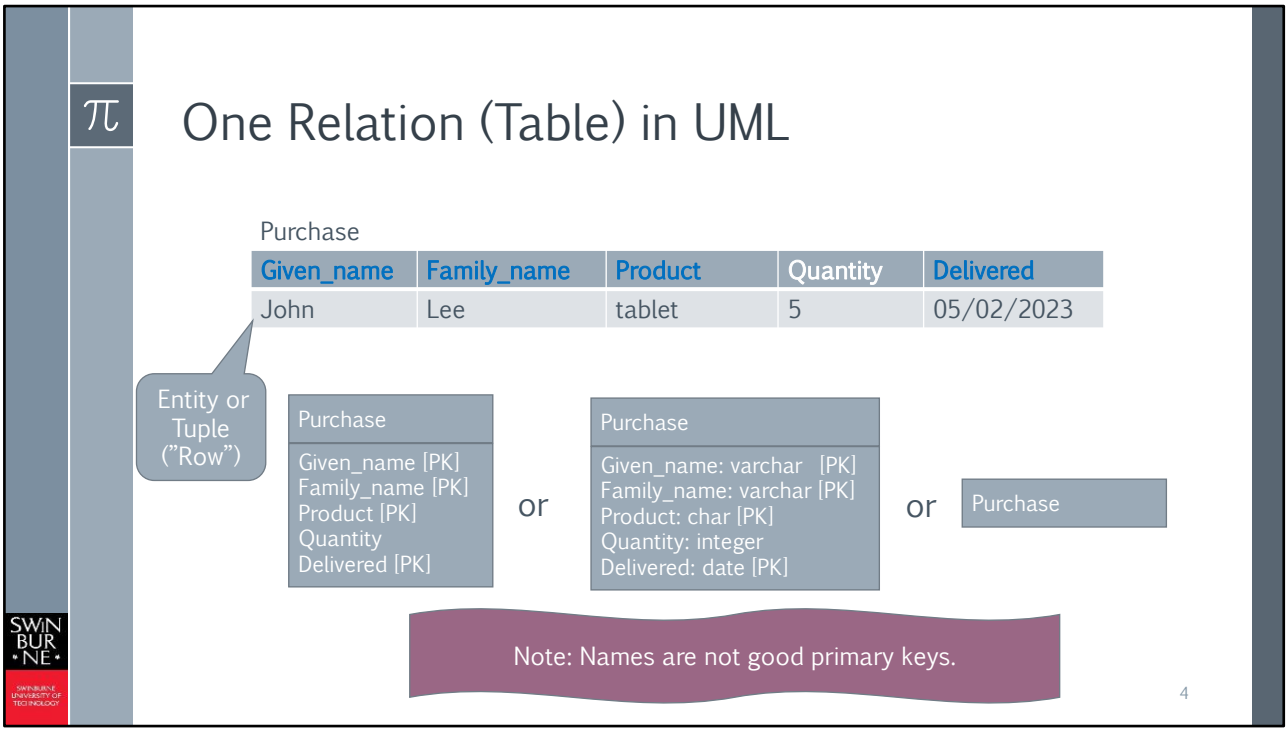

2

We have become familiar with the steps involved in relational modelling in the previous module, where we were looking at how to create entities and attributes, how to define and where to place attributes. We have also already looked into the concept of a primary key, and how it helps us determine where each attribute should be.
The model we looked at so far was very limited in terms of the information it could store. Adding relationships to the model means we can have several entities with meaningful connections between them, which makes our database much more useful.

# But first: Notation (UML)



Discussing designs is easier when we have a visual representation. Before we can discuss relationships, we need a meaningful way to visualised them along with the entities.
In FDM, we use UML notation, because it is clearer than ER diagrams

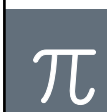



Back to our trusty, although a bit rudimentary, table of purchases. Names are not recommended as primary keys, not even good components of composite keys, but let's keep them for now.
Officially, in relational speak, tables are called relations. Entities, practically table rows, are also called tuples.

Our notation so far has been a table with some columns and data in it. That's helpful for visualisation, but not a real notation that you will find in a company when they tell you what their database looks like.
In our modules, we will use more UML, but also our data tables when it helps explain things.

The UML diagram of an entity is a box where the name of the relation/entity is shown on top, above a dividing line.
Under the line, all attributes are listed. In diagrams of an actual databases, the datatypes are usually included, because the diagram can be used by programmers who need to interface with the database. They need to know the datatypes. I might sometimes omit them for clarity here.
But sometimes, in large diagrams where we focus on the relationships rather than the content of the relation, we omit the attributes altogether.
Another item that should be mentioned is whether the attribute is part of the primary key.

# A Simple Relationship





Let's look at our first relationship.

## Two Relations and their Relationship

Customer

| Given_name | Family_name | Phone_no |
|---|---|---|
| John | Lee | 0405 999 555 |

Purchase

| Given_name | Family_name | Product | Quantity | Delivered |
|---|---|---|---|---|
| John | Lee | tablet | 5 | 05/02/2023 |
| John | Lee | PC | 10 | 10/06/2023 |

Why are we still keeping these?

6

For the sake of the example, let's continue to ignore that a first and last name combination doesn't actually make a very good identifier for a person (with so many names repeating).
For now, we want to just stick with the natural keys, so we assume all our customers differ in at least one of the first or last names.

I'm sure you have wondered how we can add the phone number to our database, because all we said so far is that it doesn't fit into our Purchase table. The problem was that our phone number would keep repeating whenever John Lee buys something, We have to do something different. Typically, in this situation, the relational model creates a new entity i.e. table. If we give the customer a table of their own, each customer will only appear once. All the attributes that are specific to the customer, like phone, address, etc. can go in that table.

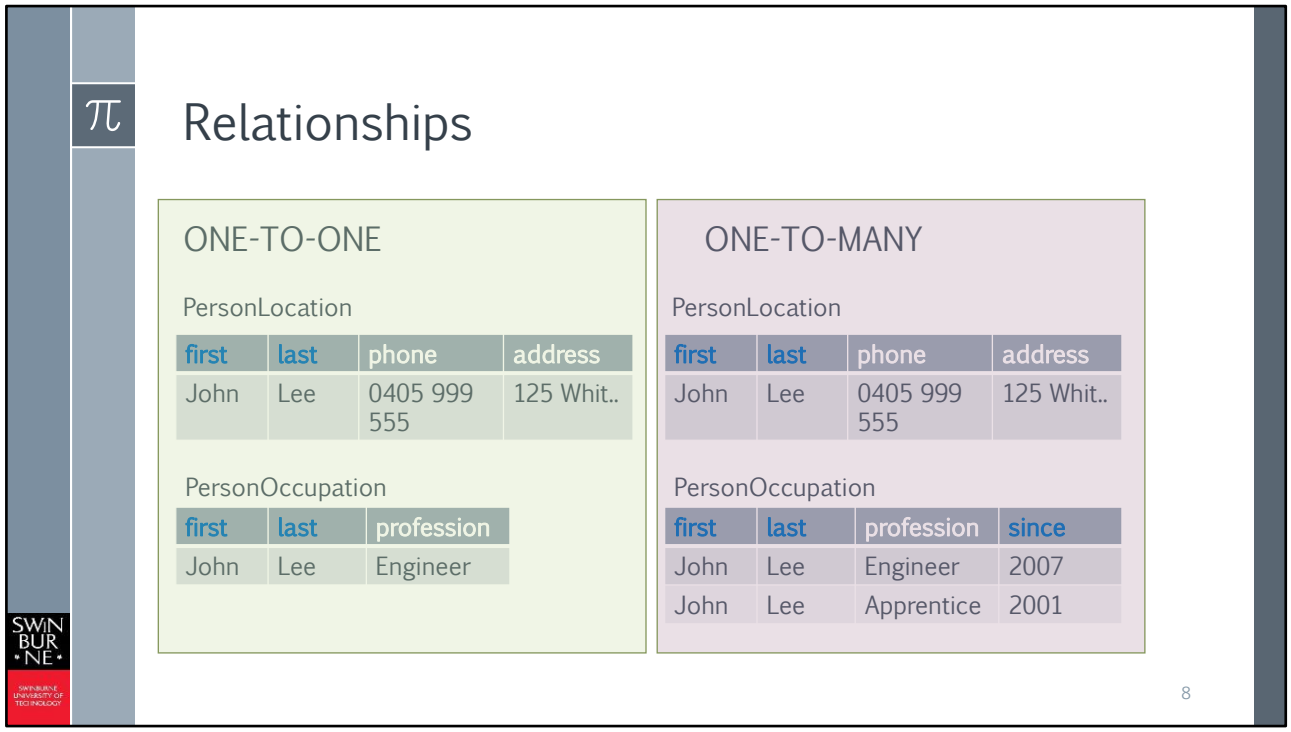
Now that we have shifted customer-specific data into the customer table, we could actually remove the two columns for the name from the Purchase table. Or could we? If we remove Given_name and family_name, we no longer know who bought the products. We also lose part of our primary key. The remaining table doesn't carry meaningful information.

The name of the customer is also the link to the customer table; if we

remove it from Purchase, we lose the information who to call when we run out of stock on a product that is up for delivery. This is because in practice, we need the columns not only as a primary key, but also as a foreign key.

# What is a Foreign Key?

- To keep a connection between two tables (relations)...
  - we have to put the primary key columns of one table into the other table as a foreign key.
- The foreign key goes only into one of the two tables.
  - Which one does it go in?
- To find out, we have to look at cardinalities (degrees) of relationships.
  - One-to-one
  - One-to-many




In a one-to-one relationship, one row (tuple, entity) in one table (relation) matches exactly one row in the other table.
One-to-one relationships are not strictly necessary to use, because the problem of duplication (redundancy) doesn't occur if we combined these tables. The reason for one-to-one relationships is always logical: We know this information is about the same entity that is already describe in the other table, but somehow we think the information in each table is somehow logically different.
For example, we can collect information about the occupation of a person, as well as where they live. We may think that these two topics are not exactly related, so we can put them in different relations. In one-to-one relationships, it is possible for both tables to have the same primary key. Having the same key means there is an automatic foreign key relationship between the tables.
One-to-many relationships are the most common relationship we usually model. Because of the redundancy rule in relational modelling, we MUST create a new table for any entity that has a different 'cardinality' (i.e. different numbers of row per same concept).
In this case, we have just made a small change to the PersonOccupation table: we allow for more than one occupation per person. We just state when the person started working in that profession. Now the database can accommodate the entire working history of a person. Note that we had to

change the primary key – the name of the person will now repeat with every new profession, and we have to add another column to provide a unique ID for the entities.

# One-to-many Relationship

Customer

| Given_name | Family_name | Phone_no |
|---|---|---|
| John | Lee | 0405 999 555 |

"Parent"

Purchase

| Given_name | Family_name | Product | Quantity | Delivered |
|---|---|---|---|---|
| John | Lee | tablet | 5 | 05/02/2023 |
| John | Lee | PC | 10 | 10/06/2023 |

"Child"

Where does the foreign key go?

9

Returning to our original example, we can now see that the relationship between the Purchase and Customer tables is a one-to-many relationship; the customer has one set of personal data, but several purchases, which equate to several tuples in the purchase table.

In a one-to-many relationship, we occasionally use the term parent for the 'one' side of the relationship, and 'child' for the 'many' side of the relationship. This is because the 'many' side depends on the 'one' side for information; without a customer, a purchase is nothing.

We have already seen that in a one-to-one relationship, the primary keys are often identical, and determining where the foreign key goes is easy.

In a one-to-many relationship, it is not much harder to see where the foreign key goes.

Suppose we were to put the foreign key into the customer table. The primary key of the purchase table has 4 components: given_name, family_name, product and delivered. We already have given_name and family_name in the Customer table. Now we would have to add product and delivered. We can see that this would break the relational principle, because we would have several values for product in one line of customer.

However, if we put the foreign key into the Purchase table, we do not need to add any columns – the given_name and family_name columns are already there, and for each row in the table, we already have this information.

This is why the foreign key in the many-to-one (or parent-child) relationship

always has to go in the child table.

# More Notation (UML)



Let's now look how we represent relationships in UML.

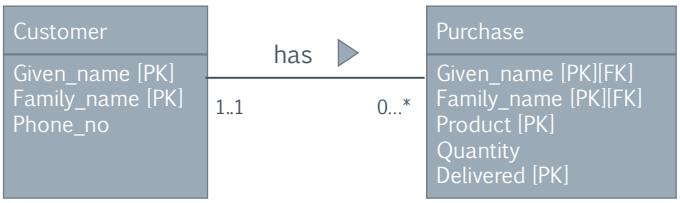
# Two Relations and their Relationship in UML

Customer

| Given_name | Family_name | Phone_no |
|---|---|---|
| John | Lee | 0405 999 555 |

Purchase

| Given_name | Family_name | Product | Quantity | Delivered |
|---|---|---|---|---|
| John | Lee | tablet | 5 | 05/02/2023 |
| John | Lee | PC | 10 | 10/06/2023 |




11

Let's now express this relationship in UML. We can see that the Purchase table now has foreign keys, marked [FK].
We also have a definition of the relationship, the 'has' verb. These are not always used. The numbers, however, are important. They are called cardinality or degree.
How you read this is:
1...1 – for every row in the Purchase table, there is at least one and at most one Customer. 'At least one' means the customer is mandatory: You cannot have a purchase without customer. At most one means, this is the parent (one) side of the relationship.
0...* - for every row in the Customer table, there is possibly no (0) row in the Purchase table. There are possibly an (almost) infinite number of rows in the Purchase table. This means we allow for a customer to be entered in the customer table before they have made a purchase. This is not unusual, but in your modelling endeavours, it is worth thinking about; should you have customers that have never bought anything? Sometimes this is a technical question: If you are building an online outlet for a company, chances are customers have to enter their details before ordering.

# Cardinalities (degrees) of Relationships

UML NOTATION

CROW'S FEET NOTATION

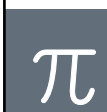

12

There are two ways people show the cardinalities of a relationship, UML, which is sometimes called Chen's notation, and Crow's feet notation.

If we allow no match in the other table, we put a zero in Chen's notation and a circle in Crow's feet notation. If there must be a match, it is a 1 in UML notation and a vertical bar in Crow's feet notation. A star in Chen's notation means many and its equivalent in Crow's feet notation are two bars forking off the main line.

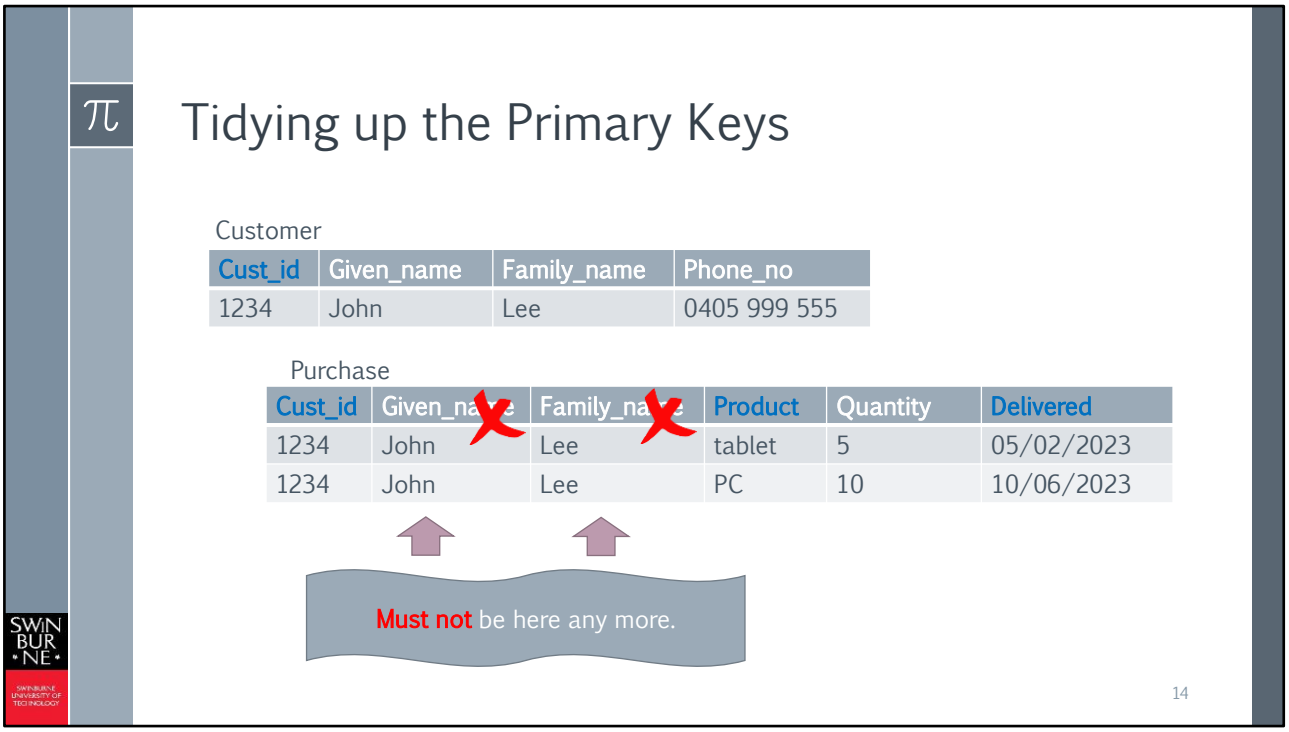
With UML notation, we generally use the numbers, but you might find Crow's feet notation (people mix and match in database diagrams), so now you know what it all means. Pretty simple really, we don't have numbers other than 0 and 1, because relational models don't really specify exactly how many child rows you can have (say, 4). Mainly because DBMSs have no way of implementing the restriction.

# Better Primary Keys





Let's look at a few more examples so that it all becomes clearer.




At long last, let's clean up the primary key of these tables. As we know, people's names duplicate all the time and are terrible candidates for primary keys. Primary keys are used to uniquely define the entity, i.e. row data, and first and last names won't really do that.

We have explained surrogate keys in the last module. For entities that represent people, such as our customers, we usually always use a surrogate key – i.e. a key that means nothing but is unique. Often, we use autonumbers, because most DBMSs have them.

An important consequence of replacing the names with an ID is that now the given and family name are no longer key, just a normal attribute. As you already know, the same data must not repeat in different tables. The foreign key is the only exception to this rule – we tolerate its repeating values because we need it to create the connection between related entities.

A consequence of this is that the primary key of the Purchase table must also change. We now have CustID, Product and Delivered.

# Tidying up more Primary Keys

Customer

| Cust_id | Given_name | Family_name | Phone_no |
|---|---|---|---|
| 1234 | John | Lee | 0405 999 555 |

Purchase

| Cust_id | Prod_id | Quantity | Delivered |
|---|---|---|---|
| 1234 | 2345 | 5 | 05/02/2023 |
| 1234 | 2346 | 10 | 10/06/2023 |

Possible new attribute (would be useful)

Product

| ProdID | Product | Price |
|---|---|---|
| 2345 | tablet | 550.00 |
| 2346 | PC | 450.00 |

15

The product that we have had so far has been a little anaemic. Normally, we know a lot more about our products – make, model, price, suppliers etc. A separate product table is no doubt warranted. And of course, a simple product name should not be key.

However, instead of the surrogate key I put in here, real businesses could use the 'SKU' number. SKU stands for stock keeping unit, and most enterprises use such a number to identify the products they sell. SKUs usually include some meaning, e.g. category of product, so I wouldn't call them surrogate keys (as surrogate keys mean absolutely nothing).

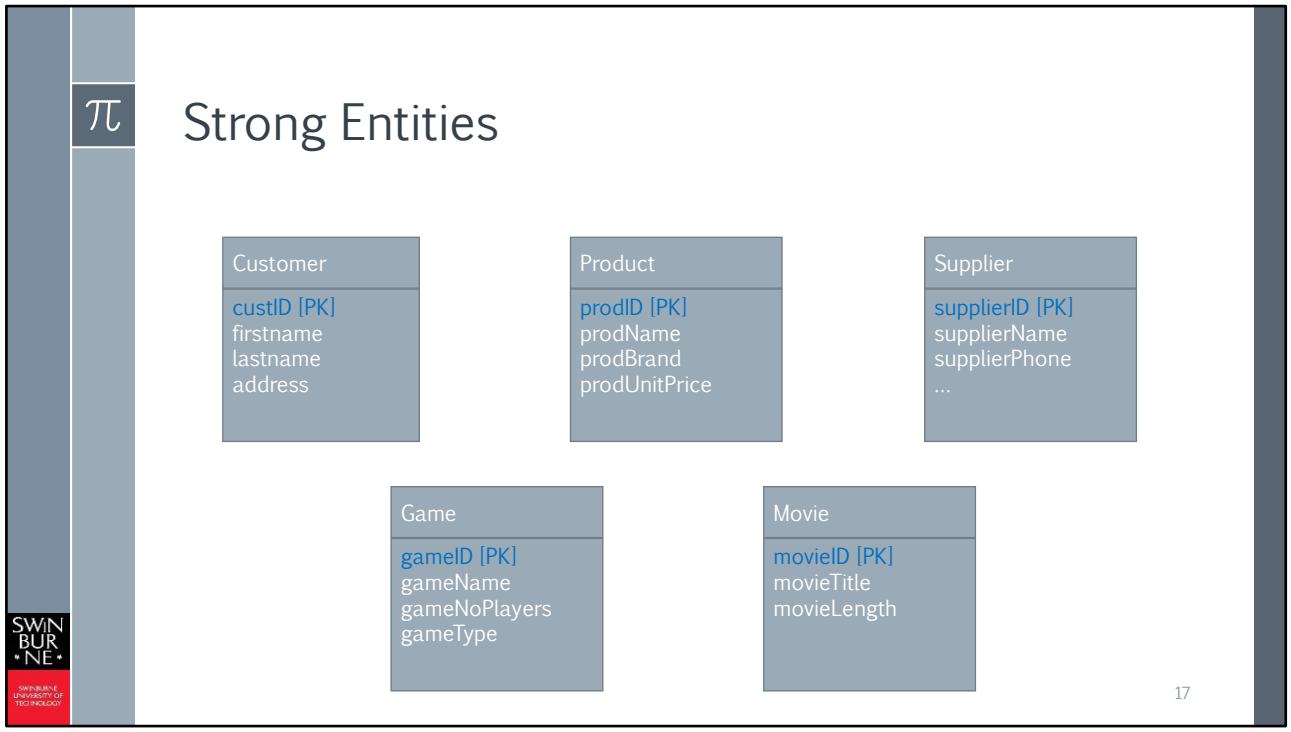
Again, observe how the primary key in the Purchase table also changes.

Now you'll be asking when we put a surrogate key into the Purchase table and get rid of the composite key. To discuss this, we better clarify the concepts of strong and weak entities.

# Strong and Weak Entities




It's a rather simple concept.

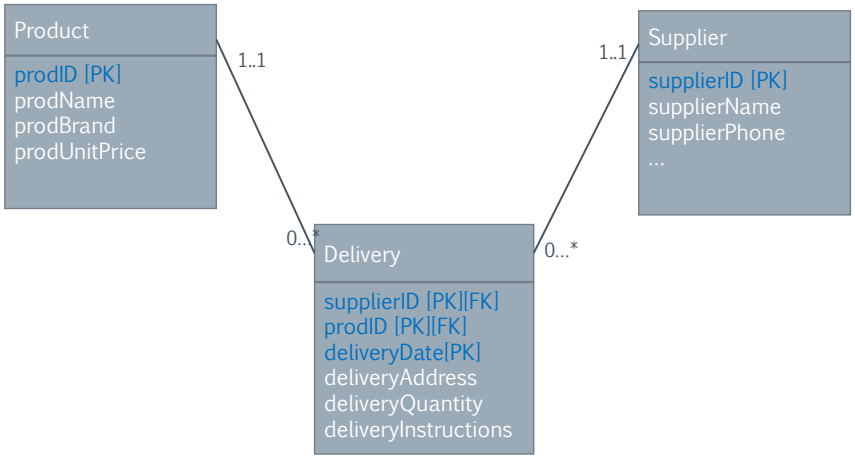



Strong entities are most often tangible things that exist in their own right, such as people or items or movies or houses. They typically do not have a composite key.

You can think of firstname and lastname as a possible natural key for customer, as we have discussed. Let's forget for a moment about the problem of uniqueness (or lack thereof) of people's names. If we choose this composite key for a Customer table, both parts of the key still describe the same strong entity (the person).

With weak entities, this is not the case – weak entities typically have composite keys that describe different aspects of the entity. Let's look at an example.

# Weak Entities




18

Delivery is a typical example of a weak entity. Product and Supplier are strong entities, but Delivery depends on both of them. Without a supplier and a product, there really isn't much of a delivery.

Let's first study the cardinalities. In our simple model, every delivery (each row or tuple of Delivery) has exactly one product. If we assume that these are just-in-time bulk deliveries, that might be appropriate. If we want different products in one delivery, we have to create an even weaker entity, but more of that later.

For each Product tuple, there is a possible 0 or many deliveries. We are allowing no deliveries, because we think that when we first enter a product, it is possible we haven't had a delivery yet.

Similarly, for each Delivery tuple, we have exactly one supplier. The supplier cannot be zero, because there is no way we can enter a delivery without knowing the supplier.

As the weak entity is (always) on the many side, the foreign keys belong in the Delivery table. They also make part of the primary key. However, they are not enough by themselves. How many times can one product be delivered by one supplier if supplierID and prodID are the only parts of the composite key? Exactly, only once. If we want to repeat the delivery, we have to include another attribute, which can be the deliveryDate. It definitely can't be the quantity or address.

## Summary


- UML helps us communicate about database modelling with others.
- Relationship modelling helps us determine how the tables work together.
- Most relationships are one-to-many. One-to-one separates attributes about the same entity by topic.
- A good relation should have no duplication of data in the tuples (except for foreign keys).
- Strong entities can stand alone.
- Weak entities describe interactions between strong entities.

19

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.