Design pattern: many-to-many (order entry)

There are some modeling situations that you will find over and over again as you design real databases. We refer to these as **design patterns**. If you understand the concepts behind each one, you will in effect be adding new “tools” to your design toolbox that you can use in building the model of an enterprise.

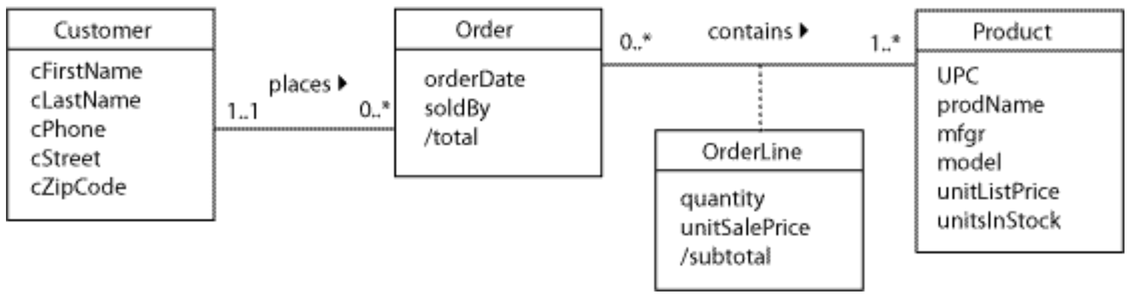
Our sales database represents one of these patterns. So far, we have customers and orders. To finish the pattern, we need products to sell. We’ll first describe what the Product class means, and how it is associated with the Order class:

> “A product is a specific type of item that we have for sale. Each product has a descriptive name; we distinguish similar products by the manufacturer's name and model number. For each product, we need to know its unit list price and how many units of this product we have in stock.”
>
> • It is important to understand exactly what this class means: an example product might be named “Blender, Commercial, 1.25 Qt.”, manufactured by Hamilton Beach, model number 908. This is a type of product, not an individual boxed blender that is sitting on our shelves. The same manufacturer probably has different blender models (909, 918, 919), and there are probably blenders that we stock that are made by other companies. Each would be a different instance of this class.
>
> “Each Order contains *one or more* Products.” (At least one because it doesn't make sense to place an order for no products.)
>
> “Each Product is contained in *zero or more* Orders.” (Zero because we might not have sold any of this product yet.)

Since the maximum multiplicity in each direction is “many,” this is called a **many-to-many** association between Orders and Products.

Each time an order is placed for a product, we need to know how many units of that product are being ordered and what price we are actually selling the product for. (The sale price might vary from the list price by customer discount, special sale, etc.) These attributes are a result of the association between the Order and the Product. We show them in an **association class** that is connected to the association by a dotted line. If there are no attributes that result from a many-to-many association, there is no association class.

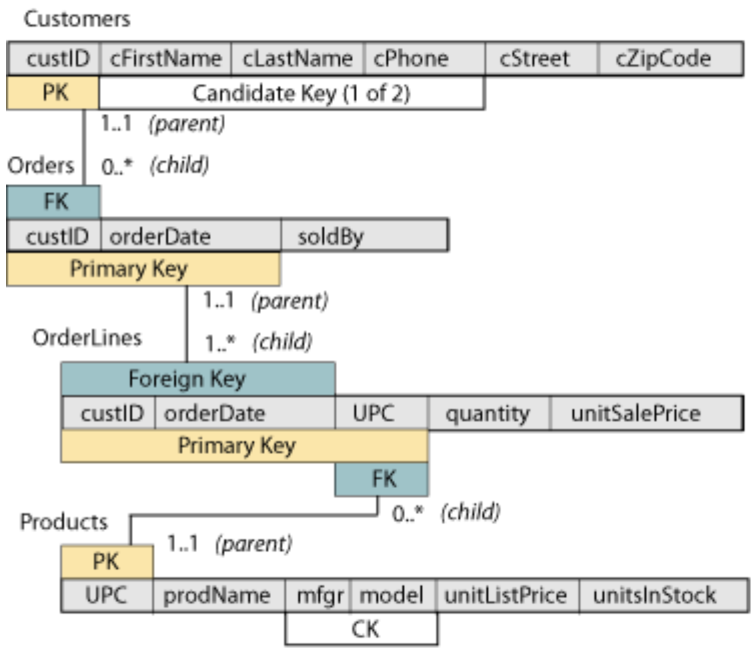
Class diagram



There are two additional attributes shown in the class diagram that we haven’t talked about yet. We need to know the subtotal for each order line (that is, the quantity times the unit sale price) and the total dollar value of each order (the sum of the subtotals for each line in that order). Since these values can be computed, they don’t need to be stored in the database, and they are not included in the relation scheme. Their names in the class diagram are preceeded by a “/” to show that they are **derived attributes**.

Relation scheme diagram

We can’t represent a many-to-many association directly in a relation scheme, because two tables can’t be children of each other—there’s no place to put the foreign keys. So for *every* many-to-many, we will need a **junction table** in the database, and we need to show the scheme of this table in our diagram. If there is an association class (like OrderLines), its attributes will go into the junction table scheme. If there is no association class, the junction table (sometimes also called a join table or linking table) will contain only the FK attributes from each side of the association.



The many-to-many association between Orders and Products has turned into a one-to-many relationship between Orders and Order Lines, plus a many-to-one relationship between Order Lines and Products. You should also describe these in English, to be sure that you have the fk's in the right place:

> "Each Order is associated with *one or more* OrderLines."
>
> "Each OrderLine is associated with *one and only one* Order."
>
> "Each OrderLine is associated with *one and only one* Product."
>
> "Each Product is associated with *zero or more* OrderLines."

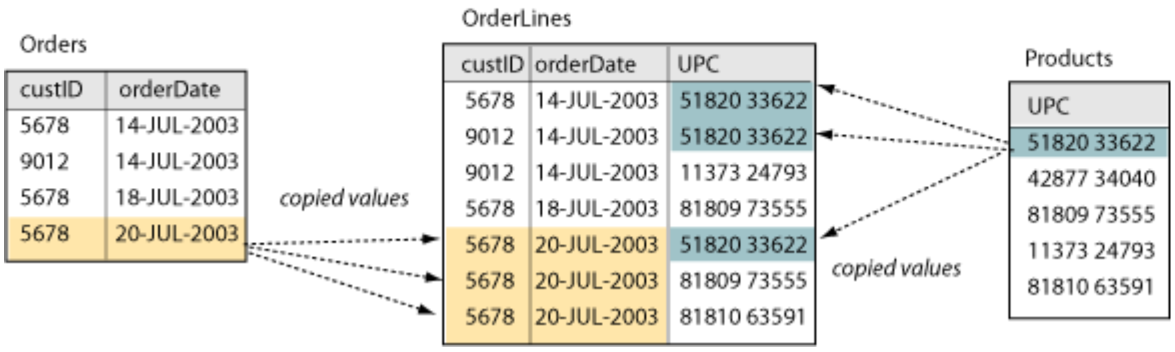
With Orders now a parent of OrderLines, we might have decided that it needs a surrogate key (order number) to be copied into the OrderLines. In fact, most sales systems do this, as you know if you've ever tried to check on the status of something you've ordered from a company. For this example, it seems to be just as easy to stick with the existing pk of Orders, since it already has a surrogate key from Customers, and the order date doesn't add much size.

The UPC (Universal Product Code) is an external key. UPCs are defined for virtually all grocery and manufactured products by a commercial organization called the Uniform Code Council, Inc.® We will use it as the primary key of our Products table, which also has a candidate key here: {mfgr, model}.

To uniquely identify each order line, we need to know both which order this line is contained in, and which product is being ordered on this line. The two fk's, from Orders and Products, together form the only candidate key of this relation and therefore the primary key. There is no need to look for a smaller pk, since OrderLines has no children.

Data representation

The key to understanding how a many-to-many association is represented in the database is to realize that each line of the junction table (in this case, OrderLines) connects *one*line from the left table (Orders) with *one* line from the right table (Products). Each pk of Orders can be copied many times to OrderLines; each pk of Products can also be copied many times to OrderLines. But the *same* pair of fk's in OrderLines can only occur once. In this graphic example, we show only the pk and fk columns for sake of space.



Orders

custID	orderDate
5678	14-JUL-2003
9012	14-JUL-2003
5678	18-JUL-2003
5678	20-JUL-2003

OrderLines

custID	orderDate	UPC
5678	14-JUL-2003	51820 33622
9012	14-JUL-2003	51820 33622
9012	14-JUL-2003	11373 24793
5678	18-JUL-2003	81809 73555
5678	20-JUL-2003	51820 33622
5678	20-JUL-2003	81809 73555
5678	20-JUL-2003	81810 63591

Products

UPC
51820 33622
42877 34040
81809 73555
11373 24793
81810 63591