

Do you object to objects?

Now object-oriented functions are bolted on to relational databases, Mark Whitehorn explains the ins and outs

Many database engines (SQL Server, DB2, Access) are based on the relational model – hence they use tables, primary keys, foreign keys, joins etc – all the usual suspects. In the 1990s a vociferous group started to argue that the time had come to replace the relational database engine with object-oriented (OO) ones. A reasonably cogent argument from that time was that OO databases were fundamentally better than relational for modelling certain types of data. This was perfectly true and is a great argument for using OO databases for specific applications. However, many OO supporters were zealots; sharing the database market wasn't even on the cards. OO was the 'only true way'; so the total annihilation of the relational database was the only option.

Which was, of course, never going to happen. The relational model has its failings, but it is very effective in many situations and was, by then, well established. To displace an established technology, the new kid on the block must be able to demonstrate some 'killer' advantage. OO is better in certain situations but these advantages are not great enough to cause companies to throw away their huge investment in relational databases. So pure OO database engines have never, and will never, replace the relational ones, but it is a shame that we can't have both types of technology in one database engine – then we could use either or both as appropriate.

Well, that is, of course, what is now happening. Relational database engines are appearing with OO technology bolted on, enabling you to use either or both. In fact, both modelling approaches can even be used in the same database at the same time. Now that we can have the best of both worlds, the major barrier to learning about OO has gone – so let's have a look at it.

To illustrate why object oriented technology has its place, we'll use a database engine called Caché which, rather than a relational database with OO capabilities bolted on, was designed from the start to be capable of handling both relational data and OO data with equal dexterity.

SCREENSHOT 1

SCREENSHOT 2

Top: A big table, full of nulls, is one solution

Bottom: You can use queries and forms to help manage the big table, but it is messy

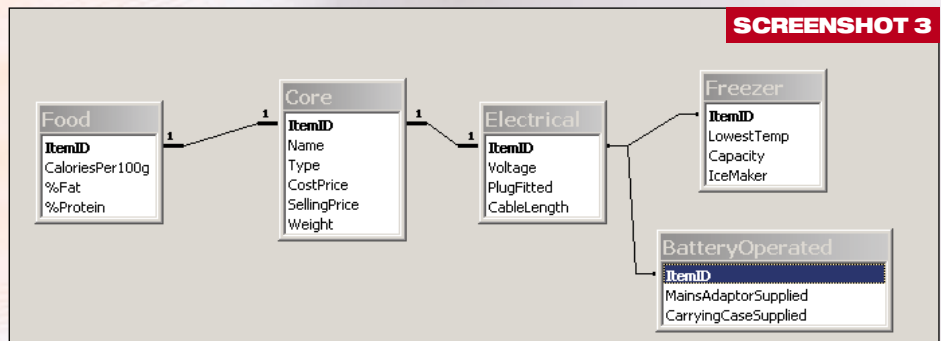
Real-world objects

The easiest way to an appreciation of objects in the database world is to look at a type of data that is handled inelegantly as relational data, but with total composure as a set of objects.

Suppose you work as a relational database designer for a supermarket. Your company sells a vast range of products – electrical items in general,

are your options? Well, you can create one huge table with all the possible attributes represented as fields, and then you can only fill in the ones that are appropriate for a given type of item (see screenshot 1).

To avoid fields being filled in where they shouldn't be, you can create queries that isolate the different types (or classes) of goods and then base forms on those queries to allow data to be entered and retrieved (see screenshot 2). This is designed to ensure that you don't allow users to, for example, enter a voltage for a peanut bar. However, this protection is at the form level only, and users might still open the table directly and enter erroneous data there, so you'll have to put security on the table itself. And you'll still end up with a table that is full of null values which are being used somewhat inappropriately; as a general rule, a table



Some sample tables roughed out as an example solution using sub-typing. This is not a complete solution; as the test suggests, a complete solution will be complex if the problem is solved this way

white goods in particular, food, clothing, and so on. Your job is to design a relational structure to hold data about the items offered for sale. Clearly all the items share some common attributes – cost price, selling price, weight, etc. However, the electrical items have a set of attributes that are unique to them – voltage, plug-fitted, length of cable and so on. Specific classes of electrical items, for example freezers, have attributes such as lowest temperature, capacity, or ice-maker. Some of the battery-powered electrical items will have attributes such as MainsAdaptorSupplied which is inappropriate for a freezer and totally inappropriate for a food item such as pancakes. You get the idea. What

that is planned, right from the start, to contain significant numbers of null values, is of questionable design). OK, let's try another relational approach that gets rid of the nulls – it's called sub-typing. You create a table that lists only the core, shared attributes. Then you create another table that lists all of the attributes that relate to electrical items, then another with the attributes for freezers and so on. Then you use a set of unique identifiers in a system of primary and foreign keys to link the whole set together. The bad news is that, in practice, this is not quite as easy as it sounds. For a start, you have to create a set of one-to-one joins (see screenshot 3), but you also have to try to ensure that

a product can't be entered as both, for example, a white good and as food. So some coding may be necessary, depending upon the database engine you use. Again, you can hide the complexity through the interface so that, when a user of your database selects a certain type of item, the relevant fields appear on the screen.

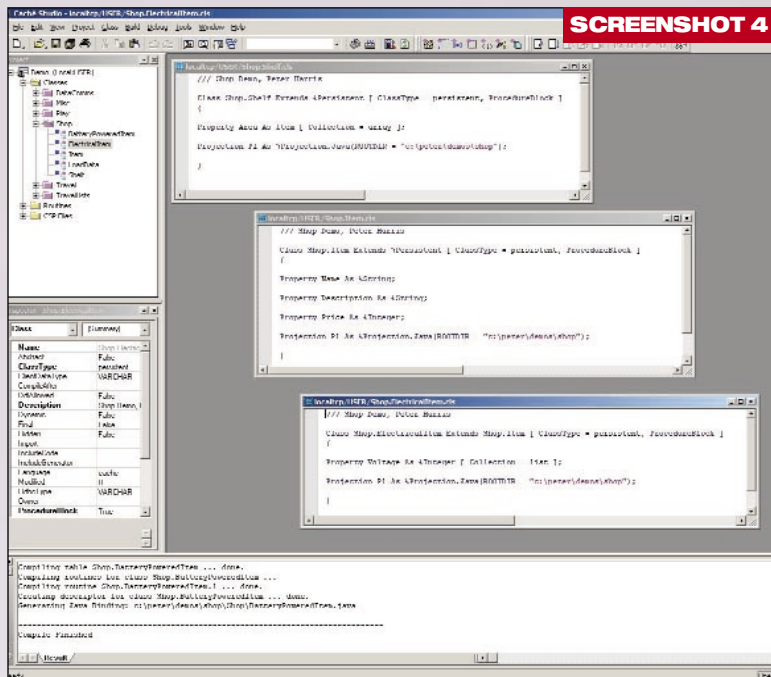
A very important point here is that the argument is not that the relational model cannot model this type of data – it can. However, to do so requires skill and hard work; and the system that you build is likely to be complex and relatively fragile; in other words it is prone to breakage if/when you have to alter it. For example, if you add a new type of item that you sell, you'll have create a whole new set of control mechanisms to make sure that only the correct data can be entered into the correct table. You can't help but be left with the feeling that the data structure you are using is ill-suited to the problem.

The object solution

Now let's try that with objects. First you create an object class called, for example, Item. That will have properties such as Name, CostPrice, SellingPrice, Weight etc. In other words, essentially the same set as the table called Core. Then you create a sub-class of object called, for example, BatteryPoweredItem. This class inherits all the properties of Item and is extended to include properties such as Carrying-CaseSupplied and MainsAdaptorSupplied. You then simply carry on, adding object classes until you have covered all of the item types (see screenshots 4 and 5). So far this sounds as if you are simply doing the same sort of work as in the second relational solution and in some ways you are. However, the difference is that object oriented technology expects objects to share some properties and not others. So you don't have to do all the complex joining and you don't have to write all the controls, they are already inherent in the structure you are building.

So, for example, if you decide to add information about a new battery-powered item, you will be able to fill in properties for CarryingCaseSupplied and MainsAdaptor Supplied; but it will also automatically have properties so you can fill in Name, CostPrice, SellingPrice, Weight and so on.

And it gets better. Suppose that you then write a set of 'Methods' for the object class called Item. These methods define, say, how users are allowed to enter data into the Item objects and retrieve it. These methods are automatically inherited by all the sub-class



Using Caché studio to manage objects (the objects shown here are not identical to those described in the text, but the basic principle is exactly the same)

objects. In other words, code that you write to control data entry only has to be written once rather than once for each sub-class of object.

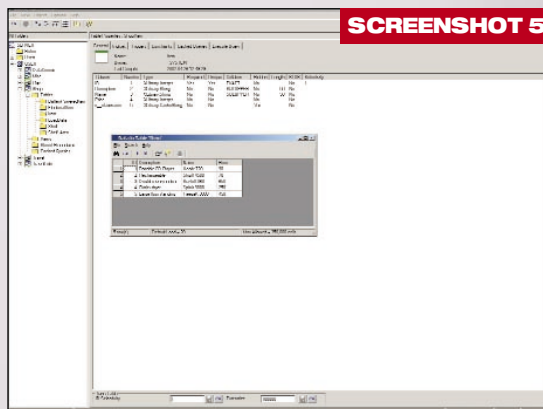
And if, in the future, you add a new sub-class, that sub-class will automatically inherit both the appropriate properties and methods.

The final advantage is that, although these structures were created as objects, Caché will let you look at the data structures as tables, if you so desire, and will also let you use standard SQL to query them. You can have your cake and eat it.

Of course, there is much more to OO databases than this, but even this simple example hopefully shows why people are so interested in the subject, particularly now that we can use objects to supplement relational databases.

If you want to have a play for yourself, visit www.intersystems.com where you can download a fully functional version of Caché. The only restriction is that it is single-user.

Looking at objects and their data from a relational viewpoint



Date programming on a roll?

With a fragment of code, you can get Access to increment and decrement dates (and other datatypes) using the + and – keys.

Simply attach this to the KeyPress event of the text box which holds the date.

```

Select Case KeyAscii
  Case 43
    KeyAscii = 0
    Screen.ActiveControl = Screen.ActiveControl + 1
  Case 45
    KeyAscii = 0
    Screen.ActiveControl = Screen.ActiveControl - 1
End Select
  
```

(Key: ✓ code string continues)

Whenever a key is pressed, Access sends its Ascii value to this code in a variable called KeyAscii. If the Ascii value is 43 (which is the plus key) the code sets the value of KeyAscii back to zero and then increments the date value by one. The minus key, as you have undoubtedly worked out, has an Ascii value of 45.

Given the Ascii code of the rest of the keys you can, of course, get this sort of code to perform whatever other manipulations you fancy.

CONTACTS

Mark Whitehorn welcomes your comments on the Databases column. Contact him via the PCW editorial office or email: database@pcw.co.uk. Please do not send unsolicited file attachments.