Surviving Client/Server: ODBMS in Practice, Part 2

by Steve Troxell

Back in July, we started looking into the Jasmine object oriented database system from Computer Associates. In that issue we went through the motions of defining class structures and methods, and basically made ourselves familiar with the world of object databases. Having got a feel for how we would set up a database in object-land, let's now turn our attention to accessing that database from a Delphi application.

My apologies for making you wait an extra month for this follow-up. The pressures of a major release for our company's product interrupted my research into Jasmine. Since the Editor has already asked me to pay for his increased medical insurance premiums from the heart attack I gave him, I'd better make good with this issue.

Before we start, let me say again that this is not intended to be a product review nor a tutorial for the Jasmine system. My goal here is to show you what it's like from a programmer's point of view to use an object oriented database, using Jasmine as the guinea pig platform.

Into The Breach

As you will find with most ODBMS systems, the Jasmine API you'll need to talk to the database from Delphi is provided in the form of an ActiveX control. There is nothing special about installing the ActiveX control, at least as far as installing any ActiveX control is special. The `TJasmine` control is installed on the component palette on the ActiveX page. You drop this visual control on a form in your project. The `TJasmine` control takes the form of a button which, when clicked at runtime, establishes the connection to a Jasmine database. Normally, you would set the `Visible` property of this control to False to hide it at runtime and simply connect to the database by calling the `Connect` method in code.

Reading The Class Hierarchy

One of the most basic things we could do from a client application is obtain a list of the classes (tables). Remember that Jasmine organizes a set of related classes into a class family; similar to how we might organize a set of related tables into a database. Figure 1 shows the Jasmine class browser's view of the `CAStore` class family. Since classes can inherit their base structure from other classes, this illustration clearly shows the class hierarchy.

How would we get this same information in our application? The `TJasmine` control provides a method called `ClassFamilyFromName` which returns a `ClassFamily` object, from which we can get details about the class family. Actually, what is returned is an OLE interface, `IClassFamily`, but from a coding standpoint, we can treat as an object.

`ClassFamily` itself has a property called `Classes`, which is a collection of objects for each class in the class family. Each item in the `Classes` collection is a `Class` object which gives us a few details about the class's metadata. As Listing 1 shows, we can loop through the `ClassFamily.Classes` collection and see all the classes available to us (I don't think I've ever in my life written a paragraph with more uses of the word "class").

Notice that the `Classes` collection is one-dimensional and doesn't really reflect the hierarchy of the class family as it's shown in Figure 1. Most of the time this is fine since we don't really need to know a class's ancestry just to see if the class itself exists. However, we can get to the ancestry information if we wanted to. The objects in the `Classes` collection contain a `SubClasses` property which is a collection of all the subclasses of the current class.

But we don't want to just take our existing code in Listing 1 and list the subclasses of each class as we come to it. That would make a wildly redundant list since every class would be listed twice: once in the main list we already have, and then again when it is listed as a subclass under its ancestor class. What we want is an indented outline like that shown in Figure 1. To do that we need to know what the 'root' class is. In the `CAStore` class family, all classes descend from `CAComposite`. So what we need to do is identify the 'root' class of the class family (`CAComposite`) and then list all of its subclasses. As we hit each subclass, we want to list all of *its* subclasses, and so on; recursively unwinding the class hierarchy.

So how do we find the 'root' class? Fortunately, the `ClassFamily`

➤ *Listing 1*

```
procedure ShowClasses;
var
  CF: OleVariant;
  I: Integer;
begin
  CF :=
    Jasmine1.ClassFamilyFromName(
      'CAStore');
  WriteLn('Class Family: ' +
    CF.Name);
  WriteLn('Classes: ');
  for I := 1 to CF.Classes.Count do
    WriteLn(Format('  %s',
      [CF.Classes[I].Name);
end;

output:
Class Family: CAStore
Classes:
  Accessory
  ActivationObject
  ActiveObject
  Background
  Belt
  Boutique
  ...
  Shirt
  Shoes
  Skirt
  Sounds
  Suit
  Supplier
  Top
```

```
procedure ShowClassHierarchy;
var
  CF: OleVariant;
  I: Integer;
  procedure ShowSubClasses(aSubClasses: OleVariant;
    aIndent: Integer);
  var
    I: Integer;
  begin
    for I := 1 to aSubClasses.Count do begin
      Write(StringOfChar(' ', (aIndent * 2) + 2) +
        'Subclass: ' + aSubClasses.Item[I].Name);
      ShowSubClasses(aSubClasses.Item[I].SubClasses,
        aIndent + 1);
    end;
  end;
begin
  CF := Jasmine1.ClassFamilyFromName('CAStore');
  WriteLn('Class Family: ' + CF.Name);
  for I := 1 to CF.TopClasses.Count do begin
    WriteLn('Class: ' + CF.TopClasses[I].Name);
    ShowSubclasses(CF.TopClasses[I].SubClasses, 0);
  end;
End;
```

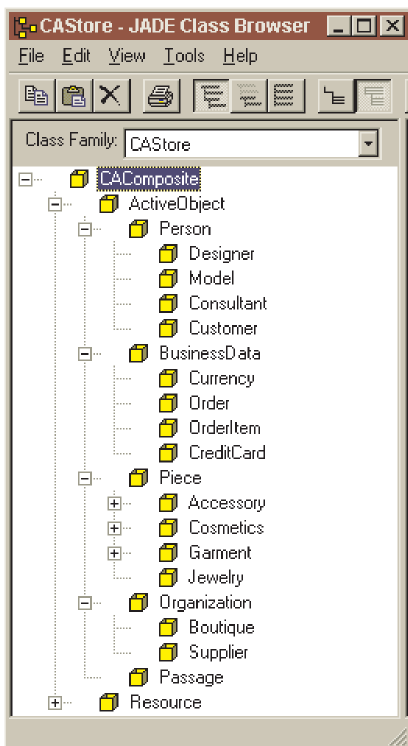
```
output:

Class Family: CAStore
Class: CAComposite
  Subclass: ActiveObject
    Subclass: Person
      Subclass: Designer
      Subclass: Model
      Subclass: Consultant
      Subclass: Customer
    Subclass: BusinessData
      Subclass: Currency
      Subclass: Order
      Subclass: OrderItem
      Subclass: CreditCard
    Subclass: Piece
      Subclass: Accessory
        Subclass: Belt
        Subclass: Handbag
        Subclass: Hat
        Subclass: Shoes
      Subclass: Cosmetics
        Subclass: Eyeliner
        Subclass: Lipstick
        Subclass: Perfume
```

➤ *Listing 2*

object gives us a collection called `TopClasses` which is a collection of all the top-level classes in the family (there is no particular reason to assume there would only be one top-level class, although that would make sense in most designs). In effect, the `TopClasses` collection is a subset of the `Classes` collection, where `Classes` contains all classes in the family, and `TopClasses` contains only those having no ancestor class defined in the family (their ancestor class, or superclass, would be the Jasmine system class `Composite`).

➤ *Figure 1*



Listing 2 shows how we would use the `TopClasses` collection and recurse through the `SubClasses` collections to get a hierarchical view of the class family like that shown in Figure 1.

Not only can we use these class collections to get the names and organization of the classes, we can also get metadata information about the properties (columns) and methods of each class. Listing 3 shows how we can see the property names and datatypes as well as the method names for the `Customer` class.

Notice that there are some out of the ordinary datatypes in this structure. The `address` and `shippingaddress` properties are ‘multi-valued’ properties. In this case, an address is one or more strings of text, like on a mailing label, with different pieces of the address on different lines. In this sense, the `address` property acts like the `Params` property of a `TDatabase` component: a collection of strings.

Also notice that the `creditcard` property is not a regular datatype at all, but is an instance of type `CreditCard`. This means that the `creditcard` property is a direct link to an instance in the `CreditCard` class (table). When we access `Customer.creditcard`, we are directly accessing a class of type `CreditCard` (made more confusing because the property name and its classname are the same thing, distinguished only by the capitalization in the classname and the lack thereof in the property name). In concept, this is no different than a `TEdit`’s `Font` property being an instance of `TFont`.

Retrieving Data

Everything we’ve looked at so far has only given us metadata information about the classes, not the actual data within them. So how do we do that? Just as with SQL databases, where the only way to get at the data is through an SQL query, the only way to get at the Jasmine data is through an ODQL (Object Database Query Language) query. To run such a query we use the `RunQueryExpression` method. Listing 4 shows us how we could get the name and address of the first customer in the `Customer` class. As we can see from our output, the `address` collection of strings lists each piece of the address on a separate line. Not my idea of a brilliant database design, but we take what we are given here.

The string we pass into `RunQueryExpression` is the ODQL query we wish to execute. In this case we are asking for all the `Customer` instances from the `Customer` class. What we get back from `RunQueryExpression` for this query is a collection of `Customer` class instances, so the `OurCustomers` variable represents a collection of `Customer` objects. We grab the first instance from the collection and stick it into the `FirstCustomer` variable. Since `FirstCustomer` is an instance of the `Customer` class, we access its properties and methods like any other class reference. So you see that we simply refer to the name and

```
procedure ShowClassStructure;
var
  CF, AClass: OleVariant;
  Properties, AProperty: OleVariant;
  Methods: OleVariant;
  I: Integer;
  DataType: string;
begin
  CF := Jasmine1.ClassFamilyFromName('CAStore');
  AClass := CF.ClassFromName('Customer');
  Properties := AClass.Properties(True, jAllProperties);
  Methods := AClass.Methods(True, jAllMethods);
  WriteLn('Class Family: ' + CF.Name);
  WriteLn('Class: ' + AClass.Name);
  WriteLn('  Properties:');
  for I := 1 to Properties.Count do begin
    AProperty := Properties.Item[I];
    DataType := AProperty.ClassName;
    if AProperty.IsSet then
      DataType := 'collection of ' + DataType
    else if AProperty.PropertyType = jString then
      DataType := DataType + '[' +
        IntToStr(AProperty.Precision) + ']';
    WriteLn(Format('    %-20.20s  (%s)', [AProperty.Name,
      DataType]));
  end;
  Write('  Methods:');
  for I := 1 to Methods.Count do
    Write('    ' + Methods.Item[I].Name);
end;
```

```
output:

Class Family: CAStore
Class: Customer
  Properties:
    customernumber        (Integer)
    address               (collection of String)
    shippingaddress       (collection of String)
    phonenumber           (String[65536])
    creditcard            (CreditCard)
    shoesize              (String[65536])
    waist                 (String[65536])
    leg                   (String[65536])
    neck                  (String[65536])
    hat                   (String[65536])
    currentOrder          (Order)
    orders                (collection of Order)
    password              (String[65536])
    nextCustomerNumber    (Integer)
    thumbnail             (CABitmap)
    photo                 (CABitmap)
    tag                   (CABitmap)
    video                 (CAVideo)
    audio                 (CAAudio)
    name                  (String[65536])
  Methods:
    addOrder
    addCustomer
```

➤ *Listing 3*

```
procedure ShowFirstCustomer;
var
  OurCustomers: OleVariant;
  FirstCustomer: OleVariant;
  I,J : Integer;
begin
  OurCustomers := Jasmine1.RunQueryExpression(
    'CAStore::Customer from CAStore::Customer');
  FirstCustomer := OurCustomers.Item(1);
  WriteLn(FirstCustomer.name);
  for J := 1 to FirstCustomer.address.Count do
    WriteLn(FirstCustomer.address.Item(J));
End;

output:
DE JAMES
163 Twisted Rd
Old Castle
WA
99910
```

➤ *Listing 4*

```
procedure ShowCreditCards;
var
  OurCustomers: OleVariant;
  Customer: OleVariant;
  I : Integer;
  CC: string;
begin
  OurCustomers := Jasmine1.RunQueryExpression(
    'CAStore::Customer from CAStore::Customer');
  for I := 1 to OurCustomers.Count do begin
    Customer := OurCustomers.Item(I);
    CC := '';
    if not VarIsEmpty(Customer.creditcard) then
      CC := Format('%-11.11s %s',
        [Customer.creditcard.type, Customer.creditcard.creditcardnumber]);
    WriteLn(Format('%-20.20s %s', [Customer.name, CC]));
  end;
end;

output:
DE JAMES            Visa        7854 753 954 1300
LYDIA CHASE         Discover    7854 753 954 1304
ANGELA EASA         Visa        7854 753 954 1306
MARION AKERLUND     Master Card 7854 753 954 1310
```

➤ *Listing 5*

address properties directly. Note that property names are case-sensitive and as the class in the Jasmine database was defined with lowercase property names, we must have lowercase property names when we reference them.

Remember the `creditcard` property in `Customer`? It was a `Customer` property that represented an instance in the `CreditCard` class. So how do we access that? We access properties of the `CreditCard` class directly, just like you would expect in an object world. Listing 5 shows a roster of all customers and their credit card numbers. If it so happened that one of our customers didn't have a credit card assigned to them, we would detect that by testing to see if the `creditcard` property was unassigned by using Delphi's `VarIsEmpty` function.

Modifying Data

How do we change values in the class? Just assign a new value to the property like this:

```
Customer.name := 'Joe Smith';
```

The new value is immediately saved in the database. You can also set up transactions so that all changes to a class instance are not recorded in the database until you commit the transaction.

What about adding a new instance to the class, like SQL's `INSERT` statement? All classes defined in Jasmine include the system method `NewObject`, which we call to create a new instance of that class. Once a new instance is created, we simply assign values to the instance properties. At this point we are simply modifying an existing instance like we described above. Any properties we don't explicitly assign a value become nil when stored in the database (equivalent to SQL's `null`). Listing 6 shows an example.

By the same token, the DeleteObject method can be used to delete an instance from a class, as shown in Listing 7.

Calling Object Methods

The classes in our object database can have methods tied to them. Listing 3 shows that our Customer class has two methods: addOrder and addCustomer. addOrder happens to be an instance-level method, meaning it operates on an instance of the class. AddCustomer happens to be a class-level method, meaning it operates independently of any particular instance in the class. You wouldn't know this without examining the method definitions in greater detail than I've shown here, so just trust me on this one.

Calling the methods of a class is just as transparent as accessing its properties. Let's start with the addCustomer method. To call a class-level method, we qualify the method name with the class name. This is similar in concept to how we call the Create method to instantiate any Delphi object by qualifying it with the name of the class we are instantiating (eg, List := TStringList.Create). So for our purposes, we need to get a reference to the Customer class itself in order to call any class-level methods. Listing 8 shows how we call addCustomer.

Instance-level methods must be called from an instance of the class. We've already seen plenty of examples of how we get class instances, so Listing 9 should be fairly straightforward. In this case, we are passing a null value for the single parameter. Note that the keyword "null" we are using is the Null predefined variant supplied by Delphi.

```
procedure CreateNewInstance;
var
  AClass: OleVariant;
  AInstance: OleVariant;
Begin
  { GetClassObject is a shortcut to getting the class metadata.
    In Listing 3 we used ClassFamilyFromName and ClassFromName
    to do the same thing. }
  AClass := Jasmine1.GetClassObject('CAStore', 'Customer');
  AInstance := AClass.NewObject;
  AInstance.name := 'Steve Troxell';
  AInstance.phonenumber := '111-555-1234';
end;
```

➤ Listing 6

```
procedure DeleteAnInstance;
var
  OurCustomers: OleVariant;
begin
  OurCustomers := Jasmine1.RunQueryExpression(
    'CAStore::Customer from CAStore::Customer' +
    'where CAStore::Customer.name == "Steve Troxell"');
  for I := 1 to OurCustomers.Count do
    OurCustomers.Item(I).DeleteObject;
end;
```

➤ Listing 7

```
procedure AddCustomer;
var
  AClass: OleVariant;
begin
  AClass := Jasmine1.GetClassObject('CAStore', 'Customer');
  AClass.addCustomer(Steve Troxell', '');
end;
```

➤ Listing 8

```
procedure TForm1.Button15Click(Sender: TObject);
var
  Customers,
  Customer: OleVariant;
begin
  Customers := Jasmine1.RunQueryExpression(
    'CAStore::Customer from CAStore::Customer ' +
    ' where CAStore::Customer.name == "DE JAMES"');
  Customer := Customers.Item(1);
  Customer.addOrder(null);
end;
```

➤ Listing 9

Conclusion

Through the magic of OLE automation (you have been reading Dave Jewell's series haven't you?) the objects in our object database can be more or less seamlessly accessed within a Delphi application. In many ways, this may make your code considerably easier to develop and maintain than with traditional databases. I can think of an endless stream of questions this article poses. How do you do this or that in an object database? My purpose here was not to illustrate every conceivable thing you would want to do with an object database, but rather to show you what the basic operations would be like to give you a taste for the technology.

One more point to bear in mind. Obviously since we are bypassing all the normal data access techniques, we cannot use Delphi's data-aware controls without creating a custom TDataset descendant. Now you know why I bothered to show you how to read the metadata as well as the actual data.

Next Month...

Next time we meet, we'll go back to the world of SQL and look at a technique to add a sort of visual query builder to an existing system. This would allow users to define filters for their data based on the plain text labels and descriptions they understand. Then we the programmers take on the burden of turning that into a dynamic SQL statement to fetch the records they are interested in.

Steve Troxell is a software engineer with Utimate Software Group in the USA. He can be contacted via email at Steve_Troxell@USGroup.com