

Keeping Hold of Your Things

by
Joanna Carter

Part 1 - Finding out what your things look like

During my recent articles on OO analysis, I have referred to the concept of an Object Store, a database specifically designed for the storage of objects. Then a client came to me with the requirement for using standard Delphi data-aware controls to edit data from a UNIX 4GL. The easiest way, in theory, to allow such connectivity is to create a component that inherits from TDataSet. This allows us to place the component on a form and connect a TDataSource component to it thus enabling data-aware controls connected to the TDataSource to connect to the underlying data.

It is my intention in this and further articles to look further at the concept of an Object Store and how it would be possible to implement that concept in a component that can be connected to Delphi's existing user interface controls. This is not a light undertaking, as there are a great many considerations involved, but I will start by looking at implementing a TDataSet derivative that is capable of reading binary data and connecting to a TDataSource component.

Strategy

I am going to handle the construction of this component a step at a time in the following order:

1. A Class that is capable of reading a schema.
2. A Class derived from TDataSet that connects a binary file to a TDataSource (the component).
3. Classes to facilitate connection to indexed data.

My aim is to create a series of classes that will serve to demonstrate just some of the principles of implementing OO design using Delphi. All code has been written using Delphi v4.02.

The Schema Class

The eventual idea behind this class is to enable us to read the schema of an Object Store, but at this stage I am simply using it to read the structure of a single table. This structure is to be stored in a separate file from the main binary data, although, of course, it could also be stored at the top of the binary data file.

We will start by defining the interface of a Field Definition:

```
ISchemaFieldDef = interface
  ['{<<your own GUID>>}']
  function DisplayName: string;
  function FieldName: string;
  function FieldSize: Integer;
  function FieldType: TFieldType;
end;
```

It might seem a bit excessive to use an interface for such an insignificant class but there is a reason which I will get to. Next we need to create a class to implement that interface:

```
TSchemaFieldDef = class(TInterfacedObject,
  ISchemaFieldDef)
private
  fDisplayName: string;
  fFieldName: string;
  fFieldSize: Integer;
  fFieldType: TFieldType;
public
  constructor Create(AOwner: TSchemaFile);
  virtual;
  function DisplayName: string; virtual;
  function FieldName: string; virtual;
  function FieldSize: Integer; virtual;
  function FieldType: TFieldType; virtual;
end;
```

Complicated, isn't it? So far, all we have is an interface declaration and a class declaration that virtually repeats the interface. Surely we only need to use the class definition on its own? Well, the idea behind deriving TSchemaFieldDef from TInterfacedObject is that it allows us to maintain an internal list of TSchemaFieldDefs in the TSchemaFile class that does not require the items in the list freeing back to the heap when it is destroyed. The four functions in the class are purely accessor methods for the four private fields. The constructor is based on the TComponent idea of an Owner being responsible for the memory of those components that it owns. The AddDef method will be explained shortly.

```
constructor TSchemaFieldDef.Create(AOwner:
  TSchemaFile);
begin
  inherited Create;
  AOwner.AddDef(self);
end;
```

Now for the class that is going to be responsible for managing the file that holds the field definitions:

```

TSchemaFile = class
private
  fFieldList: TInterfaceList;
protected
  function GetFieldCount: Integer; virtual;
  function GetFieldDef(Idx: Integer):
    ISchemaFieldDef; virtual;
public
  constructor Create(FileName: TFileName);
    virtual;
  destructor Destroy; override;
  class function FileExt: string; virtual;
  procedure AddDef(ADef: TSchemaFieldDef);
    virtual;
  property FieldCount: Integer
    read GetFieldCount;
  property FieldDef[Idx: Integer]:
    ISchemaFieldDef
    read GetFieldDef;
end;

```

As you can see, I have used a TInterfaceList for the field list. This is the partner to items derived from TInterfacedObject as it will hold ISchemaFieldDef references. I will go through the constructor for TSchemaFile in some detail in order to explain what is going:

```

constructor TSchemaFile.Create(FileName:
  TFileName);
var
  tmpFieldDef: TSchemaFieldDef;
  i: Integer;
begin
  inherited Create;

  with TStringList.Create do
  try
    LoadFromFile(FileName + FileExt);
    if Count = 0 then
      raise
        ESchemaFileError.Create('FieldDef
          file empty');
    if (Count mod 4) <> 0 then
      raise
        ESchemaFileError.Create('FieldDef
          file corrupted');

```

In this particular implementation each field in the schema file, which is just a text file, takes the form of these examples:

```

CustFirstName
ftString
25
First Name
CustAge
ftInteger
0
Age

```

There are four attributes for each field: the FieldName, the FieldType, the FieldSize and the DisplayName. These attributes follow the same rules as those for a TField or TFieldDef. First we read the contents of the file into a StringList which allows us to read each line in turn and also readily count how many lines were in the file. Then we check to see if the file was empty and if not, find out if the number of lines is divisible by four. Next we need to initialise the list and a temporary pointer to a TSchemaFieldDef:

```

fFieldList := TInterfaceList.Create;
tmpFieldDef := nil;

```

There may be other ways of stepping through a StringList four lines at a time but I thought this was rather elegant:

```

for i := 0 to Pred(Count) do
  case (i mod 4) of
  0:
  begin
    tmpFieldDef :=
      TSchemaFieldDef.Create(self);
    tmpFieldDef.fFieldName :=
      Strings[i];
  end;
  1:
    tmpFieldDef.fFieldType :=
      TFieldType(GetEnumValue(TypeInfo(TFieldType),
        Strings[i]));
  2:
    tmpFieldDef.fFieldSize :=
      StrToInt(Strings[i]);
  3:
    tmpFieldDef.fDisplayName :=
      Strings[i];
  end;
  finally
    Free;
  end;
end;

```

You will notice that I am referencing the private fields of the TSchemaFieldDef instance rather than having implemented public mutator methods in the class. What is happening here is a Delphi implementation of the Friend construct found in C++. A Friend is a class or other external entity that has privileged access to the inner sanctum of another class. In C++ friends are explicitly defined within a class, but in Delphi we have to place the two classes within the same unit and use the unit scope to gain access to the, normally hidden, private data fields of the friendly class. However, unlike C++, not only can the 'friend' see the 'friendly' class, the opposite is also true, which may not always be desirable.

Now comes the easy bit, when we need to destroy the TSchemaFile all we need to do with a TInterfaceList is to free it without freeing the items, as they are interface references and automatically dispose of themselves.

```
destructor TSchemaFile.Destroy;
begin
  fFieldList.Free;
  inherited Destroy;
end;
```

I have used a class function to return the extension that we are going to use for the schema file as this allows us to connect the file extension string with that particular class rather than any one instance or object.

```
class function TSchemaFile.FileExt:
  string;
begin
  Result := '.def';
end;
```

The last three methods are responsible for managing the internal list of ISchemaFieldDef references:

```
procedure TSchemaFile.AddDef(ADef:
  TSchemaFieldDef);
begin
  fFieldList.Add(ADef);
end;

function TSchemaFile.GetFieldCount: Integer;
begin
  Result := fFieldList.Count;
end;

function TSchemaFile.GetFieldDef(Idx:
  Integer): ISchemaFieldDef;
begin
  fFieldList[Idx].QueryInterface(ISchemaFieldDef,
  Result);
end;
```

Conclusion

We started this section by looking at one way of setting up an 'Owner' class that is responsible for looking after an internal list of owned objects. We wanted the list to automatically return the memory, used by its members to the heap, when it was destroyed and accomplished this by the use of a TInterfaceList within the owner class and deriving the list member class from TInterfacedObject, thus utilising automatic garbage collection.

We used a TStringList to read in the schema file and used modulo with the of the number lines in the StringList to assign the correct attribute from the file to the schema field definitions.

Finally we used the implicit 'friend' construct that Delphi provides in its unit scoping to access the private variables of another class.

In the next article I will go through the process of creating a TDataset derivative in some detail.

Copyright © 1999 Joanna Carter

First published in [UK-BUG](#) News March/April 1999.

Not to be reproduced without consent; please contact [Joanna Carter](#)