

Persistence Brings Success

by Joanna Carter

'I am doing some serious research into the area of object storage mechanisms with the idea of creating a TObjectStore component that can be used in much the same way as a TTable or TQuery. I hope to bring you some results soon.' – BUG News May/June 1998.

Thank you for your patience. It has been two and a half years since that article and I have just about completed an Object Store that does not work at design-time. Since that article, my research has prompted me to move further and further away from using data-aware controls, but is that a bad thing? I have found many dissenting voices discussing the pros and cons of using data-aware controls, and would like to take a moment to discuss some of those views.

The main advantages of data-aware controls are said to be:

- You can see data in your forms at design-time.
- You get character-by-character validation of entries.
- They are easy to use.

However some of the disadvantages are:

- Moving between data-aware controls can trigger unwanted events in the underlying data.
- It can be difficult to validate data entered, before it is sent to the database.
- They tie the form directly to the data – changing a field name or type means changing every form that field appears on.

If we were to discard data-aware controls, what would it cost to change?

How important is seeing data at design-time – surely we can tell how much room text is going to take up; and with the speed of Delphi's compiler, is it that time consuming to run up the form in a test program?

There is a wealth of third party controls out there that will look after character validation, entry masks, etc.

How many times have you had to spend time working out how to validate data; and how many extra datasets do you have to manage to do lookups, etc?

How much SQL do you have to write to bring data into a particular form in yet another arrangement?

What I am suggesting is not intended just to rubbish the idea of using data-aware controls, but to cause you to think about just how much work you are doing to use 'time-saving' components.

Gerald Nunn has designed an excellent custom dataset that can be hooked up to a TCollection; and there is a thread running in one of the newsgroups, all of which seems to be in pursuit of the holy grail of an IDE based object-oriented framework. However, I do sometimes wonder if we are in danger of forgetting that we are meant to be programmers; and programmers write code... Don't they?

There is, however, an alternative to using data-aware controls that I believe can simplify the whole process of writing and maintaining applications.

The Object of the Exercise

Let me describe what using my Object Store is like.

I have designed an Object Store that is capable of looking after the creation, retrieval, updating and deletion of objects. Objects have no knowledge of their ultimate source or destination and can be used without any thought of whether they are being retrieved from a local database, SQL, Paradox, Text file, or from another computer anywhere else in the world. Different objects can be retrieved from different databases on different computers and referential integrity will be maintained. Should a database have to move from one flavour of SQL to another, the whole process can usually be done in an afternoon, without changing one line of code in the application objects. Once your particular flavour of storage mechanism has been coded, all that is necessary is to register the classes and properties you will be using, together with the columns and tables that they are stored in.

The Object Store will even store inherited classes, by storing the differences from the base class in related tables. This arrangement enables me to browse the common properties of different classes that derive from the same base class. In other words, I can browse through a list of Customer, Supplier and Employee objects within the same ListView, and yet when I want to edit any of these Person objects, the correct type of form will be created for the appropriate class of Person.

To help with the choice of form for editing, I use a Form Factory where classes of forms can be associated with classes of objects without the objects knowing anything about the forms that will be used to edit them. If I want to change the style of form that edits a particular class, I just change a register entry and the whole application carries on working.

Most of my forms are derived from a few standard forms that have all the basic OK/ Cancel/Apply logic written once and for all. All that is required in a derived form is to override a method to read properties into the edits of the form, another method to write the values from the edits to the properties; and to use the IDE to design the layout of the edits and hook the OnChange event of all edits to a special event handler already written in the base form.

From the Drawing Board to the Workshop

First of all, Let me start by writing a description of a type of object that I wish to work with:

```
type
  TStockItem = class(TPDBObject)
  private
    fCode: String;
    fCurrentPrice: Currency;
    fDateLastIssued: TDate;
    fDescription: TStrings;
    fListPrice: Currency;
    fUnitOfMeasure: TUnitOfMeasure;
    function GetAveragePrice: Currency;
    function GetDescription: string;
    function GetStockLevel;
    procedure SetCode(const Value: String);
    procedure SetCurrentPrice(const Value:
      Currency);
```

```

procedure SetDateLastIssued(const Value:
                           TDate);
procedure SetDescription(const Value:
                           string);
procedure SetListPrice(const Value:
                       Currency);
procedure SetUnitOfMeasure(const Value:
                           TUnitOfMeasure);

protected
function GetDescriptor: string; override;
public
constructor Create; override;
destructor Destroy; override;
procedure Delete; override;
published
property AveragePrice: Currency
  read GetAveragePrice;
property Code: String
  read fCode
  write SetCode;
property CurrentPrice: Currency
  read fCurrentPrice
  write SetCurrentPrice;
property DateLastIssued: TDate
  read fDateLastIssued
  write SetDateLastIssued;
property Description: string
  read GetDescription
  write SetDescription;
property ListPrice: Currency
  read fListPrice
  write SetListPrice;
property StockLevel: Integer
  read GetStockLevel;
property UnitOfMeasure: TUnitOfMeasure
  read fUnitOfMeasure
  write SetUnitOfMeasure;

end;

```

You will notice that TStockItem derives from TPDObject, which is a base class that knows how to CRUD (Create, Retrieve, Update and Delete) instances of itself. If I want to save an instance of TStockItem, all I have to do, in my program code, is to say MyStockItem.Store. It will even know if it has been retrieved from the database and needs updating, or if it is a newly created object that has to be inserted into the database.

Most of the properties of TStockItem read directly from the underlying private fields and write via setter methods. However, there are some properties that behave in a non-standard fashion:

Description is meant to be a multi-line memo style property and therefore the private field is a TStrings that can maintain line breaks, etc. In order to make life easier for storing, retrieving and displaying, I have made the published property a string; this means that I need both a Get and a Set method to translate the contents of the TStrings to and from a string.

```

function TStockItem.GetDescription: string;
begin
  Result := fDescription.Text;
end;

procedure TStockItem.SetDescription(const
                                   Value: string);
begin
  fDescription.Text := Value;
end;

```

This technique also allows you to store any properties in one format or type and display them in another.

Of course, using a TStringList means that it has to be created as does the TUnitOfMeasure we are going to look at next. To do this is a simple matter of overriding the constructor and destructor:

```

constructor TStockItem.Create;
begin
  inherited Create;
  fDescription := TStringList.Create;
  fUnitOfMeasure := TUnitOfMeasure.Create;
end;

destructor TStockItem.Destroy;
begin
  fUnitOfMeasure.Free;
  fDescription.Free;
  inherited Destroy;
end;

```

The UnitOfMeasure property can be read directly from the private field as usual, but when it comes to writing, we have to use the Assign method of TPDOject which will copy the contents of one object into another rather than just copying the address.

```

procedure TStockItem.SetUnitOfMeasure(const
                                       Value: TUnitOfMeasure);
begin
  fUnitOfMeasure.Assign(Value);
end;

```

This is how all object type properties should be handled, otherwise you can end up freeing an object that is still referenced somewhere else.

Using a Getter method allows us to calculate the value of a property without holding that value in a private field; this is the same as using a Lookup field in a TDataset, but can be a lot more powerful :

```

function TStockItem.GetAveragePrice: Currency;
var
  ItemCount: Integer;
begin
  ItemCount := 0;
  with TBatchCollection.Create do
  try
    AddCriteria(TEqualsCriteria, 'StockItem',
               [self]);

    Connect;
    while not IsDone do
    begin
      Result := Result +
                CurrentItem.ShelfPrice;
      ItemCount := ItemCount +
                CurrentItem.Quantity;

      Next;
    end;
    Result := Result / ItemCount;
  finally
    Free;
  end;
end;

function TStockItem.GetStockLevel: Integer;
begin
  with TBatchCollection.Create do
  try
    AddCriteria(TEqualsCriteria, 'StockItem',
               [self]);

    Connect;
    while not IsDone do
    begin
      Result := Result + CurrentItem.Quantity;

      Next;
    end;
  finally
    Free;
  end;
end;

```

Whenever these properties are required, their values will be dynamically calculated, based on whatever batches are currently in stock. Likewise, we can use a Setter method to determine what action to take should a value be invalid. Here I check to see if the Code assigned to this new object has already been used:

```
procedure TStockItem.SetCode(const Value:
                               String);
begin
  if not IsPersistent then
    with TStockCollection.Create do
      try
        AddCriteria(TEqualsCriteria, 'Code',
                    [Value]);

        Connect;
        if StoredCount > 0 then
          raise
EStockException.Create('Validation(Code)',
                        'Stock Code already exists');

      finally
        Free;
      end;
    fCode := Value;
  end;
end;
```

Likewise, we can enforce referential integrity with this class simply by overriding the Delete method and carrying out our validation:

```
procedure TStockItem.Delete;
begin
  if StockLevel > 0 then
    raise EStockException.CreateFmt('Delete
%s', 'Cannot delete, quantity still in stock',
                                    [fCode]);

  inherited Delete;
end;
```

Of course, you might say that this seems like a lot of code to write, especially if you have got a large number of classes; but this code only has to be written once. No matter where you are in your program, if you want to Delete a TStockItem, you will be sure that all you have to write is MyStockItem.Delete and, if there is any stock left, you will not be allowed to proceed.

Having looked at what is involved in writing an object that knows how to CRUD itself, we will go on to look at the idea of a collection of objects that we have already used to validate properties.

```
type
  TStockCollection = class(TPDCollection)
  private
    function GetItems(Idx: Integer):
                                   TStockItem;

  public
    constructor Create(AOwner: TPDObject =
                       nil); override;
    function ItemClass: TPDObjectClass;
                                   override;
    function AddItem(AOID: TOID) : TStockItem;
    function CurrentItem: TStockItem;
    property Items[Idx: Integer]: TStockItem
      read GetItems; default;
  end;
```

All collections derive from TPDCollection, which like TPDOject knows how to get its own data. What is more TPDCollection gets data in chunks, the size of which can be determined to suit the anticipated usage. TPDCollection supports the concept of an Iterator, which can be defined as:

```
IPDIterator = interface
  ['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx}']
  function CurrentItem: TPDObject;
  procedure First;
  function IsDone: Boolean;
  procedure Next;
end;
```

Because some of the TPDCollection methods like CurrentItem and AddItem only return a TPDOject, it is more convenient to work with instances of the class of the collection than having to continually cast these to the desired type. We therefore write methods that simply convert the result of the TPDCollection methods. Note that these methods cannot override the base methods as they must return the different type:

```
function TStockCollection.AddItem: TStockItem;
begin
  Result := TStockItem(inherited AddItem);
end;

function TStockCollection.CurrentItem:
                                   TStockItem;
begin
  Result := TStockItem(inherited CurrentItem);
end;

function TStockCollection.GetItems(Idx:
                                   Integer): TStockItem;
begin
  Result := TStockItem(inherited Items[Idx]);
end;

constructor TStockCollection.Create(AOwner:
                                   TPDObject = nil);
begin
  inherited Create(AOwner);
  Chunksize := 15;
end;
```

The primary reason for overriding the constructor is to allow us to set how many records will be retrieved in a chunk. The underlying data broker will add this number of records to the collection every time Next is called, spreading the network traffic and allowing the broker to close the database transaction as soon as the collection is fully populated, even if you have not read to the end of the collection. I have also implemented a full Cursor class for PDCollection, which allows you to go to the Last object and to step backwards through the collection by calling Prior, but as SQL only returns a Unidirectional cursor, it means the entire table would have to be retrieved before you would get to the last record. If you really need to go to the last object in a collection, it is better to reverse the order of the objects in the collection and go to the first one.

And that is all the extra code you have to write to create a collection of any type of object.

How Much, How Many and Which Way Round?

When it comes to using these collections in a program, you need to specify the range of objects that you wish to retrieve and also the order that the objects should be sorted by.

TPDOject supports the idea of a Proxy, or minimal object. When you register a class, you specify which properties you are most likely to use when browsing the collection, and these will be the only values that will be retrieved from the database, thus significantly reducing the load on the

database as well as network traffic. Whenever you want to edit an object, you will set the object's IsProxy property to False and then call Retrieve, thus executing a retrieval of all the properties of the object.

To give you some idea of the benefits of setting up this system of paging and proxy objects, I can make it appear that I have filled a ListView with 15,000 objects in less than a second. As long as the data is moved through a page at a time, the user should not be aware that all the data is not already there. On the other hand, if the user were to press Ctrl-End, they would have to wait between 5 and 10 seconds for the list to fill the first time but, thereafter, the entire list can be navigated virtually instantly.

```
procedure TStockForm.RetrieveStock(Value:
                                string);
begin
  StockCollection.ProxyObjects := False;
  StockCollection.AddCriteria(TStartingCriteria,
                              'Code', [Value]);
  StockCollection.AddOrderBy('Code',
                              obdDescending);
  StockCollection.Connect;
end;
```

This example method starts off by saying that we want to retrieve the full objects (because we are only hoping for a small result set!), by setting ProxyObjects to False. Then we add a Selection Criteria that stipulates that only objects whose Code starts with the Value passed in will be retrieved. We want to order the collection on the Code property in descending order and finally we call Connect to enable the Broker and retrieve the first object.

There is a great deal more to discuss on this subject, and I will continue to reveal more of the Object Store in subsequent articles.

Conclusion

We started off by taking a quick look at some of the pros and cons of data-aware controls, and then went on to describe some of the benefits of using an Object Store. Next we looked at the design of a class that derives from TPDOObject, TStockItem. We looked at the use of property Get and Set methods to provide data translation, calculated values, and a means of validating values. We said that it was possible to override methods like Delete to provide a means of enforcing referential integrity.

The concept of a collection of objects is encapsulated in the TPDCollection class and we looked at the few, minor, methods that need to be overridden to enhance the usability of the collection. A Cursor is available for use with the collection, but this can cause speed problems if a call is made to Last on a large collection. A better alternative was to reverse the order of the collection and go to the First object.

Finally, we looked at how TPDOObject supports the idea of a Proxy version, which allows us to retrieve only sufficient data to browse, thereby reducing database and network load, with the use of Selection Criteria to limit the range of objects returned and specifying the order of the objects returned.



Joanna Carter is a writer, developer and trainer specialising in requirements-driven training and consultancy. Joanna is involved with mentoring several companies in the setting up of mapping object-oriented systems into relational databases. You can e-mail her at JoannaC@btinternet.com and her web site is at joannac.btinternet.co.uk.