Life Without Data-Aware Controls

by Philip Brown

Last year at *DCon 99*, I presented a session *Design Patterns for the Real World* and amongst the patterns discussed was the Bridge Pattern. The example shown was a class that could populate a `TTreeView` control given a concrete implementation of an abstract class designed to expose hierarchical data. It was shown that this design was ultimately more flexible and elegant than the established way of using a data-aware `TTreeView` variant. I passed comment that I never use data-aware controls, and that I was aware that this put me in the minority!

Shortly afterwards, this session was commented upon in *One Last Compile...* (Issue 51, November 1999), and the author noted that my assertions caused a sharp intake of breath around the room. This is not too surprising, as I was challenging a very accepted, even 'normal', way of writing Delphi applications, one which is fully documented and encouraged in many books, and supported by many third-party components. Fortunately, the author of the *One Last Compile...* column noted that by the end of the session I had gone some way to justifying my comments and that I had stirred up some deep re-appraisals of how Delphi programs could be implemented. Such is the benefit of attending conferences, and I'm sure the Editor won't object to me noting that *DCon 2000* is but a few months away! *[Not at all: don't forget to book your place soon! Ed.]*

The assertion that data-aware controls are flawed can be contentious. Many people have used them to write fully functional programs that satisfy the requirements of their users. If a development principle works, why bother changing? Many Delphi programmers who have been tutored to rely upon data-aware controls seem to be suspicious of other possibilities. A common reaction when the validity of relying upon data-aware controls is questioned is a knee-jerk response that it is in the manuals, it's quick and easy, and everyone else does it, so it must be the best way, right?

Data-aware controls *are* covered in the Delphi manuals, but this doesn't mean that they're necessarily the best way of representing database field values to the user. After all, early versions of Delphi promoted the use of ReportSmith as a report generator, but I'm sure that few developers still view this as the 'right' way to do reports. It may be worthwhile to consider *why* data-aware controls exist. My first experience of data-aware controls as a concept was with Visual Basic, before the advent of Delphi 1. Very quickly I (and many other VB developers) grew to dislike them intensely and forswore them for all developments. Initially, this was based on their extremely buggy nature and lack of control, but later on more subtle issues became evident. When Delphi was released I was very keen to see how it interfaced the visual display to the database, and was disappointed to see that the concepts had been slavishly copied from VB. In truth, this may have had nothing to do with the desires of the original Delphi design team but rather more to do with the target market for Delphi, to win over developers from VB (remember when Delphi was marketed as 'the VB killer'?) In order to be instantly familiar to VB developers, the basic concepts of data-aware controls were implemented wholesale in Delphi, substituting the BDE for the JET engine as the database API layer with which the controls interacted.

It is also possibly true that a need to appeal to simplistic competitive language 'reviews' also drove Borland to mimic VB in this way. At the time magazines were very fond of putting up a number of similar software products in a 'group test' and deciding upon a winner. These very often boiled down to a feature point checklist, and if Delphi had lacked the data-aware controls of VB it would have been correspondingly marked down. Delphi was being sold as a Rapid Application Development tool, and to the average reviewer this basically meant 'how many mouse clicks does it take me to get the data from my one database table displayed on-screen'. Assessing the quality, robustness and durability of an application produced with the tool had nothing to do with the evaluation: expedience was everything.

Although fundamentally better implemented in Delphi, data-aware controls in this language share many of the disadvantages of their VB forebears. Originally there was one overriding reason against using them, a rigid reliance upon the BDE. Readers will remember the obscure chicanery (such as patching DCU files) required to use a non-BDE database with Delphi 2. This was in stark contrast to the rest of the elegant Delphi architecture, and it was a very welcome improvement in Delphi 3 when a decent database-related class hierarchy was introduced with `TDataSet`. Despite the removal of the reliance upon the BDE, there are still a number of practical and theoretical reasons why I prefer not to use data-aware controls.

Taking A Stand

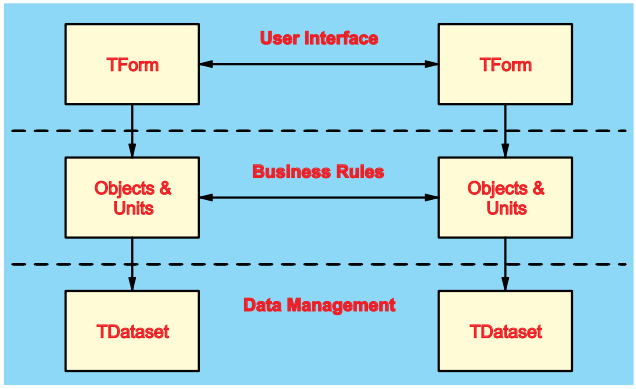
Data-aware controls basically sacrifice control for expedience. By wholeheartedly accepting the functionality that they provide it is possible to get data in front of the user very quickly. The strength of this approach (getting data in front of the user with minimum effort) is also its weakness: it is very hard to modify the behaviour and look and feel of the application from that which data-aware controls provide. Sometimes, as with very small throwaway in-house systems, this is acceptable. For

systems that must be deployed to client sites there are a number of drawbacks.

At a purist level, there are many theoretical disadvantages to data-aware controls. All of the proponents of advanced Object Oriented principles such as Grady Booch, Peter Coad, Ed Yourdon *et al* agree that a well-designed OO system has a strong separation between the user interface, business logic and database interaction layers. Many design patterns centre around the proposition of separating data representation objects from a presentation, keeping these elements firmly decoupled. Figure 1 shows the recommended communication pathways between these layers: an arrow means that an object in one layer can interrogate a property or call a method on an object in another layer. A brief analysis of this diagram shows that the user interface (here represented by `TForm` objects) can call methods on other forms and on the objects representing the business rules, but that objects representing business rules cannot call methods on interface elements. Business rules can be represented as objects or as standard procedural code within units.

The implications of this diagram are wide-ranging and encompass all aspects of application development. You do not need to be doing 'purist' OO development to benefit from the concepts represented within it. For example, strongly separating business rules and user interface means that the *only* code that can go behind buttons, menu options and the like is code that calls methods implemented in the business rules and then updates the user interface by interrogating the state (properties) of objects representing business. This rules out forms that have massive amounts of calculations and database updates behind buttons. In fact, by following the principles of this diagram, the code within forms becomes much more focused on representing data visually and controlling interaction with the user, and becomes correspondingly easier to understand and maintain. Applications that consist of a couple of enormous form units are a maintenance nightmare waiting to happen. Unfortunately, a complete discussion of application architecture will have to be the topic of future articles!

➤ *Figure 1: Interactions between application functional layers.*



Returning to the topic of this article, there is an immediate implication in the diagram for data-aware controls, the user interface *cannot* communicate directly with the data management layer, and this is exactly what data-aware controls do! Aside from the purist arguments against this, there are many practical reasons for not wanting to do so. Simply exposing data fields on-screen forces the interface to enforce business rules (such as two fields having related values), and this logic is better represented elsewhere as it must be enforced within any interface element that exposes such fields. It is true that for much of the time a data field will store a value that can be easily represented on-screen (such as a customer name), but as soon as any degree of control over data entry is required the data-aware world starts to work against you rather than for you.

Trading control for expedience has another side effect: your interface (and maybe database design) will be guided towards offering features that map well onto a data-aware control, rather than the most appropriate interface element or database type. An example of this is the commonly encountered `Address` element. This is very often coded as a number (usually 4 or 5) of separate data-aware edit boxes named `Address1` .. `Address5` or things like `House`, `Street`, `Town`, `County`, etc. The disadvantages of this approach are that it doesn't cater well for exceptional addresses (with many lines or with individually long lines), and that users rarely enter the data in such a well-defined fashion (a 3-line address may end up with the county in the `Town` field). A better approach is to design the address component as a multi-line type control (`TMemo`) and to store the data in a database-friendly form (such as `TMemo.CommaText`). This allows the presentation to vary from how it is stored, maximising flexibility and control. At this stage, data-aware proponents will be thinking `DBMemo`. For simple cases this can be utilised, but you still have little control over how the data is stored, and any control you have must be replicated every time the control is used. Another example of a datatype with poor support is sets. Sets can often be a natural, expressive and compact representation. Consider a database that stores data about people, in particular the pets they own. Assuming we don't need to know anything about the pets other than whether or not the person owns one, this could be represented with the type in Listing 1.

This representation is the most natural to use within business logic, but few databases support set type fields and there is no set-oriented data-aware control. A

➤ *Listing 1*

```
TPet = (Dog, Cat, Rabbit, Mouse,
        Gerbil, Budgie, ...);
TPets = set of TPet;
type
  TPerson = class
  private
    FPets: TPets;
  public
    property Pets: TPets
      read FPets write FPets;
  end;
```

small amount of work is required to translate this type into a standard database field, but this should only need to be implemented once and called from wherever it is required. In the data-aware world, lacking a set control, the developer would typically shy away from the natural representation and probably use a more cumbersome array of `Boolean` fields (represented using `TDBCheckBox`).

This last example shows another aspect of using data-aware controls: if not all data types can be easily represented using standard data-aware versions then the interface becomes an eclectic mix of standard and data-aware variants. The population and handling of these controls becomes quite distinct (one being implicit within property values and the other explicit in code) and co-ordinating the two requires complex event handlers to intercept, and if necessary interrupt, standard data-aware mechanisms. Sacrificing control for instant programmer gratification is beginning to become inconvenient.

Ugly Applications

Another objection to data-aware controls is their appearance and usability, both of which are somewhat lacking. The nature of most data-aware applications is immediately apparent due to the proliferation of `DBNavigator` controls and data-aware grids. The simple fact is that these are interface elements only seen in these type of applications, there are extremely few examples of grids of any type appearing in the Windows OS, Microsoft products and most commercial applications. Simply by using data-aware controls you are making your application different from most examples in the marketplace.

The `DBNavigator` control is also not very user-friendly because of its intensely record-based nature. Admittedly, the learning curve associated with it for the user is not immense, but why not use an interface style that is more descriptive, natural, and in keeping with other Windows applications? Generally, the interface paradigm that is used throughout Windows is for the user to select to `Add` or `Edit` a new object. This brings up a dialog showing the properties for that object, to which changes are made before finally confirming or cancelling the action. Although this interface style is possible using data-aware controls, most developers go for the simple option of slapping a data-aware grid in the client area of a form and letting the user make in-place edits, controlling the whole process with the `DBNavigator`. This is in direct contrast with the rest of Windows, and most commercial applications.

Once this approach is taken, most developers quickly wish for more functionality from their data-aware grids, as the one provided with Delphi offers little support for in-place editing of any data types other than strings. In these situations they hunt around for a third party alternative, for which they do not have to look too far. Virtually every single component pack contains an all-singing, all-dancing data-aware grid, and the Delphi component websites are awash with freeware or shareware alternatives. Rather than exhibit the strengths of data-awareness, this shows the major limitation. When the standard functionality is insufficient, it is necessary to search for a complete replacement. By sacrificing control, you have placed yourself entirely beholden to the features that your choice of component offers. When these are inadequate, there are few options available without wholesale changes to the application.

Another area in which an application that uses data-aware controls has little control is in direct interaction with the database. The data access pattern will be dictated by the implementation of the control and this may have a large impact on database performance. Some implementations of data-aware grids effectively prepare a recordset based on selecting the entire contents of a table, even if these records are sent to the client a block at a time. At the server database level, having a 'live' (updatable) query open can affect the performance and concurrency of future transactions. As the numbers of users increases, this can impact noticeably.

A common approach to a property sheet type dialog with data-aware controls is to allow the components to make updates to database fields as the user changes the displayed values, and then to commit the transaction when the `OK` button is pressed, or to force a rollback if `Cancel` is pressed. Although this might seem like 'free' functionality (it takes little code on behalf of the programmer, and the net result is effective), the actual impact at the database level is that the server will be performing unnecessary transactions. All this extra network and database traffic can be avoided by assuming control over the database updates: this requires a little more effort on behalf of developers, but the benefits are far greater concurrency and efficiency of bandwidth utilisation.

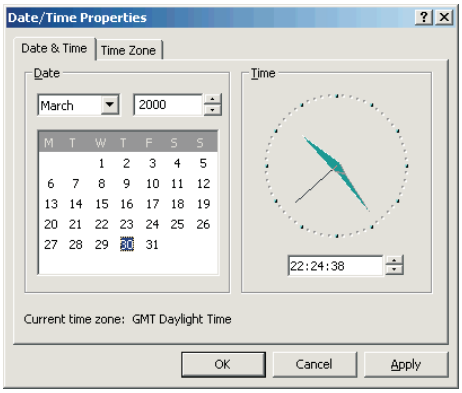
One of the biggest drawbacks to data-aware controls is the implicit nature of data access: a `DBEdit` box will contain the name of a database field that will be displayed. This is little more than a string value and should the name, type or size of the field change then the reference will be out of date, and will most likely cause a runtime error. In the event of such a field change, the only way to correct *all* occurrences is to check every single form for such a reference. Over the lifetime of even a medium sized application, such oversights are bound to occur.

What's The Alternative

To implement an application without using data-aware controls it is necessary to consider how they are used, and sometimes to re-evaluate how to design the system. Data-aware controls can be categorised into two distinct types: ones that handle a set (list) of data and ones that handle a single element. Examples of the former would be grids and

listboxes, the latter would include edit and checkbox components. So that applications can conform to the user interface style within Windows, I tend to have a property sheet type dialog for each main element in the system. An 'element' in this situation could be an object representing some part of the business rules (a 'problem domain' or business object), or in a less object-oriented application it could be something like a `TDataSet` descendant. Implementing these property sheets is a one-off process that happens quite early on in the lifetime of the application. Obviously, these dialogs will be updated and maintained as properties or fields are added to the elements they represent, but the essential factor is that there is a single property sheet dialog for each main element within the system. These dialogs will be referenced from many places throughout the application, and provide a standard way of viewing and updating the details for an object.

➤ *Figure 2: Standard Windows property sheet dialog.*



To demonstrate how such a dialog can be implemented we shall consider a worked example for a database record representing a customer, and details about the orders they have placed. A quick examination of any Windows property type dialog shows a standard dialog as shown in Figure 2, this example is the date/time Control Panel applet. The general layout is a page control with `OK`, `Cancel` and `Apply` buttons aligned with the right margin of the page control. Many more such examples can be found within Windows, and indeed other applications.

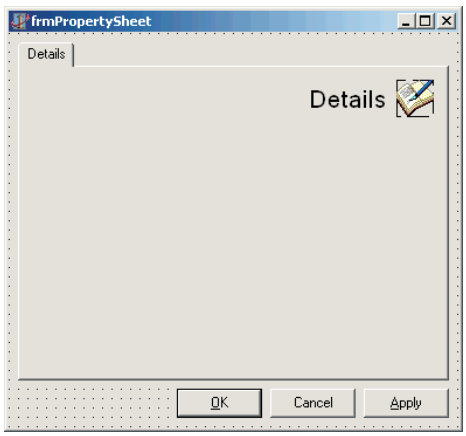
We will use a similar approach for all of our property sheet dialogs. In addition, I like to add a largish label and a 32x32 icon to each tab on the page control. This adds a pleasant graphical embellishment to the form (without descending into the anarchy of gradient controls and customised colour schemes), and if the same icon image is used to represent such objects throughout the system then it can assist the user to understanding what they are looking at. Some Microsoft BackOffice components use a similar approach to their property sheets.

As all of our property sheets will have a similar presentation, it seems sensible to use Delphi's visual form inheritance (VFI). VFI is a real boost to application consistency and productivity, and no other Windows development language does it anything like as well. For more information on this

feature, see Guy Smith-Ferrier's articles in Issues 55 and 56. Using VFI has the benefit of enforcing a consistent way of programming and, as Guy noted, if you decide to change some of the fundamentals of how the form behaves (such as advancing to the next control when `Enter` is pressed), applying this once to the ancestor form causes all descendants to behave similarly.

In order to create a new base property sheet dialog, create a new form and drop a page control onto it. Add a new page to this control, and on this new page add a `TImage` resized to 32x32 and a `TLabel` alongside it that should be right aligned and have the font changed to something a little larger, say Arial or Tahoma 16 point. I've added a default icon to the image control but in general a descendant form would update it with something more appropriate. The form could also have the `BorderStyle` changed to `bsSingle` and the `BorderIcons` property to `[biSystemMenu, biHelp]` to make it similar to other such forms within Windows. Note that there's nothing stopping us increasing (or decreasing) the size of this form for our descendants as best suits the particular object being edited.

We now need to consider exactly how we are going to support database operations, now that we are going to avoid using data-aware controls. An obvious way of doing this is to populate the visual components once, just before the form is displayed, and to update the database if the `OK` button is pressed. The advantage of this approach is that there are two relatively simple database operations going on (a `SELECT` to populate the form and an `UPDATE` to populate the database), without keeping a live recordset open in the meantime. This benefits database concurrency and availability. There are very many variations on this situation (the dialog could be passed a recordset as a parameter and it could assume the 'current' record, or the dialog could be passed an object which can be assumed to be pre-populated). The essence is that there is a single process to populate visual components from a supplied data source. The disadvantage of this approach is that there is (currently) no control over other users editing or updating the same object. Note, however, that by using the form inheritance we can implement such features ourselves relatively easily. Using this approach it is possible to lock the database record using database locks or a field reserved for this purpose, or to do read-around-write type data control when the `OK` or `Apply` buttons are pressed. This is a small amount of code to write but we have been able to take complete control over arbitration to data and *know* that it is the same throughout the application. At this stage it's worth pointing out that it is accepted that the current design is breaking the theoretical rules against the interface interacting with the data management layer directly. To overcome this it is necessary to move towards a more advanced object oriented design in which the concept of recordsets is totally encapsulated within the data management layer. In this situation we would pass in an object representing a business object (conceptually a class-oriented wrapper around a database record) to our property sheet dialog. Sadly, such a purist approach is beyond the scope of this article.



➤ *Figure 3: A base property sheet dialog.*

It is actually possible to provide a base property sheet dialog that is completely neutral to the object that is being edited. Our dialog will simply be passed a `TObject` that needs to be edited; because the base dialog provides only interface functionality it does not actually need to know more than this about the class(es) it will edit. Of course, any descendant forms will need to have a very exact knowledge about what they are editing (whether it is a `TDataSet` type object, or a custom instance representing an application-specific business object) but they will simply be able to typecast the object appropriately.

Rather than use a single instance of the property sheet dialog, each time we attempt to edit an object, a new form will be dynamically constructed. To facilitate this process our base dialog exposes a new `Edit` class function:

```
class function Edit(
  ObjectToEdit: TObject):
  TModalResult;
```

It can be seen that as far as our base property sheet dialog is concerned, *anything* can be passed in to it. The customised handling of this is left entirely to the descendant forms, a powerful example of polymorphism. This routine does little more than create a form of its own type and returns the result from `ShowModal` to determine if the `OK` or `Cancel` buttons were ultimately pressed. A form constructor is virtual and therefore calling `Self.Create` results in the descendant form being constructed rather than our base property sheet dialog itself, remember that in class methods `Self` refers to the class *type* rather than the object *instance*.

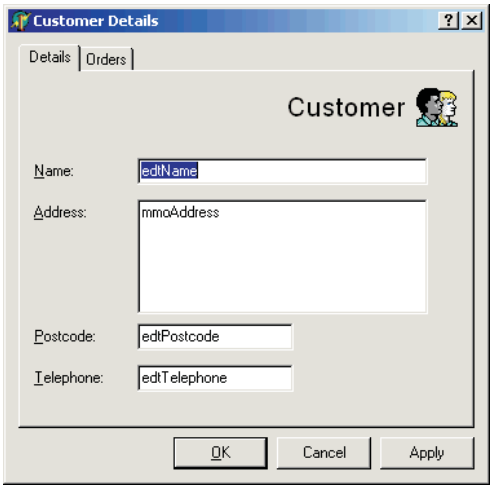
At this stage our base property sheet dialog looks something like Figure 3. Not very exciting, but we'll soon change that. Let's look at the implementation of how the form is populated from the data. The first thing to consider is how this is going to happen. A simple dialog would have an appropriate component for each property we want to update, and these could be populated from the object passed to the form (be it a recordset or

business object). In more complex examples, however, we might have a number of tabs on the page control, and some of these might be quite complicated to update. A very common example would be to show related data in the classic master/detail type approach. In our customer example this would be to show the list of orders that the customer has ever made. Rather than clutter up the dialog and create a very complex tab with both the customer details (name, address etc.) and the master/detail order information, it is better to split the data across two tabs. The first will be simple data showing the customer name and the second will display the lists of orders the customer has ever placed. This has a number of benefits. Firstly, it keeps the dialog simple to use and understand for the user. If they are not bombarded by information they will find it easier to comprehend the dialog. Secondly, it has the advantage that we can delay populating the second tab until the first time that the user looks at it. This can have a major performance benefit: nine times out of ten, when a property sheet is displayed the user will only want to view or edit the basic information. The query to select the data to populate the order listing for the customer might be complicated, or might return a lot of rows (or both): far better only to issue this query when we know it is needed. By contrast, the classic master/detail form incurs this overhead each time it is displayed.

Implementing this populate-on-demand approach is very simple. The page control generates an `OnChange` event just before the page is displayed, so we can intercept this event and make a call to a virtual function (overridden by our descendants) that causes the page to be populated. We only want to do this the first time each page is accessed so our base property sheet dialog provides an `IsLoaded` property indexed by `TTabSheet` that returns a `Boolean`. This is made available to descendant classes in the `protected` section in case they are interested in knowing which tabs have been displayed or not. The implementation of `IsLoaded` is relatively trivial and simply requires the base dialog to know which forms have been loaded. The implementation presented here uses a bit in a 4-byte Integer to represent each tab's `PageIndex` property. This has the capacity to handle up to 32 separate tabs on each property sheet; if you need more than this then the implementation must be changed but I really can't imagine what a

➤ *Listing 2: Implementation of a base property sheet dialog.*

```
unit PropertySheet;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, ComCtrls;
type
  TfrmPropertySheet = class(TForm)
    pgcDetails: TPageControl;
    tabDetails: TTabSheet;
    btnApply: TButton;
    btnCancel: TButton;
    btnOK: TButton;
    imgDetails: TImage;
    lblDetails: TLabel;
    procedure pgcDetailsChange(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure btnOKClick(Sender: TObject);
    procedure btnApplyClick(Sender: TObject);
  private
    LoadedTabs: Integer;
    EditObject: TObject;
    function GetIsLoaded (TabSheet: TTabSheet): Boolean;
    procedure SetIsLoaded (TabSheet: TTabSheet;
      Value: Boolean);
    procedure SaveDetails;
  protected
    // Allow control over page loading
    property IsLoaded [TabSheet: TTabSheet]: Boolean
      read GetIsLoaded write SetIsLoaded default False;
    // Override these methods to populate pages
    procedure LoadTab (TabSheet: TTabSheet; ObjectToLoad:
      TObject); virtual; abstract;
    procedure SaveTab (TabSheet: TTabSheet; ObjectToSave:
      TObject); virtual; abstract;
  public
    class function Edit (ObjectToEdit: TObject):
      TModalResult;
  end;
implementation
{$R *.DFM}
function TfrmPropertySheet.GetIsLoaded (TabSheet:
  TTabSheet): Boolean;
begin
  // Use bitwise operator for speed and compactness
  Result :=
    (LoadedTabs and (1 shl TabSheet.PageIndex) <> 0);
end;
procedure TfrmPropertySheet.SetIsLoaded (TabSheet:
  TTabSheet; Value: Boolean);
begin
  if Value <> IsLoaded[TabSheet] then
    LoadedTabs := LoadedTabs xor (1 shl TabSheet.PageIndex);
end;
class function TfrmPropertySheet.Edit(ObjectToEdit:
  TObject): TModalResult;
var
  dlgPropertySheet: TfrmPropertySheet;
begin
  dlgPropertySheet := Self.Create (nil);
  try
    dlgPropertySheet.Icon.Assign(
      dlgPropertySheet.imgDetails.Picture.Icon);
    dlgPropertySheet.EditObject := ObjectToEdit;
    Result := dlgPropertySheet.ShowModal;
  finally
    dlgPropertySheet.Free;
  end;
end;
procedure TfrmPropertySheet.SaveDetails;
var
  ThisPage: Integer;
begin
  for ThisPage := 0 to pgcDetails.PageCount - 1 do begin
    if IsLoaded [pgcDetails.Pages[ThisPage]] then begin
      SaveTab (pgcDetails.Pages[ThisPage], EditObject);
    end;
  end;
end;
// Events
procedure TfrmPropertySheet.pgcDetailsChange(Sender:
  TObject);
begin
  if not IsLoaded [pgcDetails.ActivePage] then begin
    // Populate controls
    LoadTab (pgcDetails.ActivePage, EditObject);
    IsLoaded[pgcDetails.ActivePage] := True;
    // Reset focus to first control on tab
    pgcDetails.ActivePage.SetFocus;
    SendMessage (Handle, WM_NEXTDLGCTL, 0, 0);
  end;
end;
procedure TfrmPropertySheet.FormShow (Sender: TObject);
begin
  pgcDetailsChange (Sender);
end;
procedure TfrmPropertySheet.btnOKClick(Sender: TObject);
begin
  SaveDetails;
  ModalResult := mrOK;
end;
procedure TfrmPropertySheet.btnApplyClick(Sender: TObject);
begin
  SaveDetails;
  btnApply.Enabled := False;
  btnOK.Enabled := True;
  btnCancel.Enabled := False;
  btnCancel.Cancel := False;
  btnOK.Caption := 'Close';
end;
end.
```



➤ *Figure 4: Customer property sheet dialog.*

property sheet with 32 tabs would look like!

The `OnChange` event handler ensures that the first time each tab is displayed a call is made to an abstract method that requires the custom dialog to populate the appropriate tab. Each descendant form of our property sheet dialog must therefore override this method and perform any task necessary. The general format of this method is a set of statements that detect the specific tab passed in (remember, this routine will be called a number of times at non-determinate intervals) and then react accordingly. In order to populate the visual components, the descendant form will need to typecast the object passed to the `Edit` method to a known type, and then update the visual components from the object. If the object to be edited was a `TDataSet` type then this would take the form of accessing named field values. If the object was something better qualified, like a business object (say `TCustomer`), then the visual components would be updated directly from the object properties. A similar process happens when the `OK` or `Apply` buttons are pressed: for each tab that has been loaded a call is made to another abstract method that requires the descendant form to update the object from the visual components. Listing 2 shows the complete implementation of our base property sheet dialog.

Let us consider our twin-tabbed `TCustomer` example. First, add the base property sheet dialog to a new project (or ensure that it has been added to your Repository). Then, select `New` from the `Delphi File` menu, and select the property sheet form. On the form that results, add a new tab to the page control, and copy and paste the label and image from the first tab to the second. Rename these to reflect the fact that this tab will be displaying a list of orders for the customer. On the first tab, add a series of ordinary edit boxes, labels and a memo to represent the basic customer details. When you have finished, the first tab will look something like that shown in Figure 4.

In order to populate these controls from the object we must override the `LoadTab` method as shown in Listing 3. Here the example chosen was for a `TDataSet` type object, note that the example given works equally well irrespective of the specific type of `TDataSet` passed.

Calling the `Edit` method on the customer details class now displays the dialog, and the user can change any values to their heart's content. I have chosen to use the standard visual components in this example, but more functional variants can of course be used, as long as they are not data-aware!

It is easy to see how we have full control over all aspects of data entry in a consistent manner, and that this display and control is completely independent of the data source. This can be demonstrated by upgrading the customer example to use a specific `TCustomer` type object rather than a `TDataSet`. In this instance we would pass in a `TCustomer` instance to the `Edit` method on the form, but this time rather than populate the visual components from named fields, we would use named properties. This has the particular advantage that the names and the types of the `TCustomer` properties are known at compile time, and therefore any attempt to compile an application with invalid references to properties will fail. This has huge ramifications for application quality: simply by compiling an application you know that you will not get *any* runtime errors due to incorrect property names or types embedded in any forms.

Whenever the user presses the `Apply` or `OK` buttons a routine is called in our base dialog that makes a series of calls to another abstract method requiring the dialog to save the details, again once for each tab that has been loaded. The implementation of this routine is easy, it's a question of copying and pasting the code from the `LoadTab` method and reversing the assignments. This process will either update the recordset field values, or the object properties (depending on what type of object is being edited). In order to force the values through to the database it will be necessary to do a `Post` on the `TDataSet`, save the business object using whatever mechanism it provides, or construct an SQL query to perform the update. If your application can make assumptions about the type of object being edited then it is possible to create an intermediate descendant of our property sheet dialog that is customised to handle objects of a known generic type, performing these updates automatically. This has obvious

➤ *Listing 3*

```
procedure TdlgCustomerDetails.LoadTab(TabSheet: TTabSheet;
  ObjectToLoad: TObject);
begin
  with ObjectToLoad as TDataSet do begin
    if TabSheet = tabDetails then begin
      edtName.Text := FieldByName ('Name').AsString;
      mmoAddress.CommaText := FieldByName ('Address').AsString;
      edtPostcode.Text := FieldByName ('Postcode').AsString;
      edtTelephone.Text := FieldByName ('Telephone').AsString;
    end else if TabSheet = tabOrders then begin
      // Populate orders tab…
    end;
  end;
end;
```

benefits from a code centralisation point of view. Using this approach it is possible to reduce each specific details dialog to contain just the code necessary to populate the components from the object and vice versa, and to control any input from the user.

Avoiding Gridlock

We have seen how to implement a property sheet type dialog that allows the user to view or edit a number of values related to a single object. In order to be able to view these properties, the user must have had some way of selecting the specific object, and typically this will happen by choosing one from a list of many.

The traditional data-aware solution is to use a grid and, as previously mentioned, these very often support in-place editing. Personally, I avoid using grids on two counts: they are not used within standard Windows or Microsoft applications and they are not intuitive for a user to use. The editing options (and control) they support are weak, and the scroll bars are typically not proportional to the amount of data displayed on screen, compared with the amount of data available.

Instead, we could use the component that Windows uses extensively for displaying lists of values: the `TListView` with `ViewStyle` set to `vsReport`. This highly functional component has a very pleasing appearance, and has become ubiquitous within Windows and other applications, virtually replacing the listbox (a listbox can be viewed as a single column listview, with the column headers hidden). The one advantage that the listbox has over a listview is that it can be populated with data much more quickly than a listview, although this becomes apparent only when dealing with hundreds of items. Generally this is not a major issue, as few users will appreciate being required to hunt for a specific entry within a list of a thousand: needles and haystacks spring to mind. I'm not sure whether the overheads associated with the listview are related to the Delphi wrapper or the underlying Win32 common control, but Delphi 5 made some welcome improvements in listview handling performance over previous versions. When populating the listview with a number of items it's always worthwhile surrounding the task with calls to `BeginUpdate` and `EndUpdate` on the `TListView.Items` property: this suppresses many costly internal events.

The listview can be customised at design-time, defining the columns and so on in the usual way. Of course, you can use more functional listview variants if you choose: I generally use one that does intelligent sorts when a column header is clicked, and changes a particular column width as the component is resized. Useful defaults for such listviews are to enable `RowSelect` and `ReadOnly`.

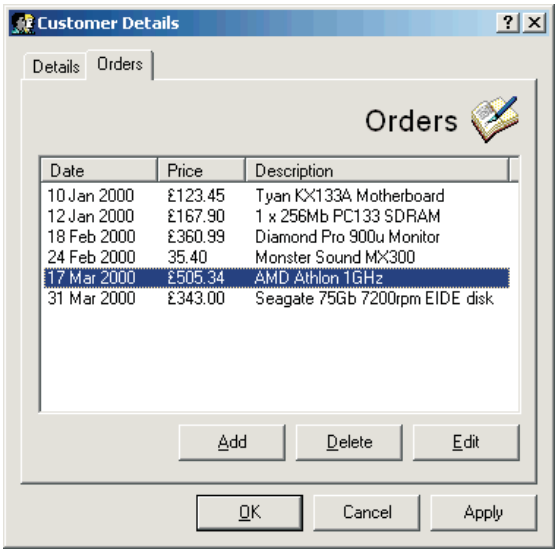
➤ *Listing 4: A generic routine to populate a listview.*

```
procedure LoadList(ListView: TListView; DataSet: TDataSet;
  Fields: array of String);
var Index: Integer;
begin
  Screen.Cursor := crHourGlass;
  ListView.Items.BeginUpdate;
  try
    ListView.Items.Clear;
    DataSet.First;
    while not DataSet.EOF do begin
      with ListView.Items.Add do begin
        Caption := Fields[0];
        for Index := 1 to High (Fields) do begin
          SubItems.Add (DataSet.FieldByName (Index).AsString);
        end;
      end;
      DataSet.Next;
    end;
  finally
    Screen.Cursor := crDefault;
    ListView.Items.EndUpdate;
  end;
end;
...
LoadList (lvwOrders, OrderDataSet, ['Date', 'Price', 'Description']);
```

At runtime the listview must be populated with data. In a similar way to how the property sheet dialog works, I favour a static population of the listview as a one-off process, and then allow the user to interact with this list. This will typically be to select an entry to edit (which brings up an appropriate property sheet dialog), to delete an entry from the list or to add a new entry to the list. Deleting an entry is obviously simple, and new entries are catered for by creating a new blank object (this could be a new record in the dataset, or a freshly created business object) and then editing it using the property sheet dialog. A `ModalResult` of `mrOK` causes the object to be saved (inserted into the database).

There are a number of ways to populate the listview. The essence will be a loop through each available object in a given list, this could be records in a dataset or a series of business objects. It is very simple to provide a standard method, possibly on the listview itself, that adds items incrementally from a given source. This could look something like that shown in Listing 4, although many variations are possible. One particularly elegant variation would be to have a generic `LoadList` routine that can populate a listview from *any* source of data, be it a `TDataSet`, a list of business objects or any other set of entities. This would be achieved by applying the Bridge design pattern. In this instance we would define an abstract class to allow navigation through a set of objects, and then provide concrete implementations to handle specific examples of `TDataSets` and so on. These advanced concepts may need to be the topic of a future article!

This may seem like a lot of work, but in essence it only needs to be done once. All of these concepts are application-independent and can be applied many times over. These approaches are certainly more powerful and flexible than the standard data-aware variants. With the rest of the Delphi VCL and RTL being so well designed, I suspect that had Visual Basic not



➤ *Figure 5: An example of an alternative to a data-aware grid.*

prejudiced the world with data-aware controls, the actual implementation of data and interface integration within Delphi would have been more along these lines.

Figure 5 shows the Orders tab of our customer property sheet dialog, together with some example purchases. This interface style is much more consistent with Windows as a whole, and is more immediately accessible to a user, as there are explicit buttons to assist with each maintenance task (adding, editing and deleting entries).

Conclusion

We've discussed data-aware controls and their limitations. An alternative way of allowing users to interact with data has been explored and a base property sheet type dialog has been demonstrated. All record-oriented database updates can now be handled in a more consistent manner. The application is now more similar to other Windows applications in look and feel, and centralised code offers opportunities to significantly enhance the behaviour of all such forms. In particular, we've wrested back control over data access and visual representation, and we've provided some significant benefits in database utilisation and concurrency, increasing the responsiveness of complex forms by only populating those components that the user wishes to view. The amount of code required for specific forms is small, and is much clearer and easier to maintain than the interdependent event handlers necessary when working in the traditional data-aware fashion.

Data-aware controls are a very expedient way to start writing an application, but for systems of any size the alternatives offer significant benefits that should be fully evaluated.

Philip Brown is a senior consultant with Informatica Consultancy & Development, specialising in OO systems design and training. When not orienting objects he enjoys sampling fine wine. Contact him at phil@informatica.uk.com