Under Construction:
Data-Awareness And Interfaces

by Bob Swart

Five years ago (Issue 7, March 1996) I first wrote about data-aware components in this column. Now I want to share some new insights on the architecture of data-aware components. I want to show you a more elegant way of implementing data-awareness using interfaces.

But first, let's examine the way data-aware components work behind the scenes, and see how we can make our own data-aware components (both the current way, and the proposed new way).

Data Controls

I usually call them data-aware controls, but in Delphi, C++Builder and Kylix you find them on the `Data Controls` tab of the Component Palette. They all display the same characteristic: a `DataSource` property (to connect to a `DataSource`). Most of them also feature a `DataField` property (to connect to a specific `DataField` from the `DataSource`). Some, like the `DBNavigator` and `DBGrid` component, work on the entire dataset that they obtain from the `DataSource`, so they do not have (or need) a `DataField` property. We'll focus on the first category of data-aware components today, and will now start to make one ourselves.

Choose `File | New`, and select the `Component` icon from the Object Repository, which results in the `New Component` dialog. Let's make a useful data-aware component that doesn't exist yet (in Delphi 5), like a data-aware calendar (based on the `TCalendar` component that can be found on the `Samples` page of the Delphi Component Palette). Select `TCalendar` in the `Ancestor type` combobox. The `Class Name` now shows `TCalendar1`, but rename that to `TDMCalendar` instead. For the `Palette Page` I always use `DrBob42`, but you should specify your own logical choice here. Finally, I click on the `OK` button (and not yet on the `Install` button), to generate the `TDMCalendar` component skeleton.

As we noticed before, a data-aware component has a `DataSource` property (of type `TDataSource`) and a `DataField` property (of type `String`), so let's add the following lines to the `published` section of the `TDMCalendar` component:

```
property DataSource:
  TDataSource
  read GetDataSource
  write SetDataSource;
property DataField: String
  read GetDataField
  write SetDataField;
```

Now, hit `Ctrl+Shift+C` to let Delphi generate empty getter (read) and setter (write) routines for these two properties.

TFieldDataLink Delegation

You now have four empty methods inside your unit, and probably wonder where you need to get or set the `DataSource` and `DataField` properties. This will all be taken care of by (or rather: is being delegated to) a private field called `FFieldDataLink` of type `TFieldDataLink` that you need to add to the `private` section of the `TDMCalendar` component. The `TFieldDataLink` type is defined in the `DBCtrls` unit, by the way, and `TDataSource` is defined in the `DB` unit, so add these units to the `uses` clause within your

➤ *Listing 1: TDMCalendar first implementation.*

```
unit DMCalendar;
interface
uses
  Windows, SysUtils, Classes, Controls, Forms,
  Calendar, DB, DBCtrls;
type
  TDMCalendar = class(TCalendar)
  private
    FFieldDataLink: TFieldDataLink;
    function GetDataField: String;
    function GetDataSource: TDataSource;
    procedure SetDataField(const Value: String);
    procedure SetDataSource(const Value: TDataSource);
  protected
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
    property DataField: String read GetDataField
      write SetDataField;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DrBob42', [TDMCalendar]);
end;
constructor TDMCalendar.Create(AOwner: TComponent);
begin
  inherited;
  FFieldDataLink := TFieldDataLink.Create;
end;
destructor TDMCalendar.Destroy;
begin
  FFieldDataLink.Free;
  FFieldDataLink := nil;
  inherited
end;
function TDMCalendar.GetDataField: String;
begin
  Result := FFieldDataLink.FieldName
end;
function TDMCalendar.GetDataSource: TDataSource;
begin
  Result := FFieldDataLink.DataSource
end;
procedure TDMCalendar.SetDataField(const Value: String);
begin
  FFieldDataLink.FieldName := Value
end;
procedure TDMCalendar.SetDataSource(const Value:
  TDataSource);
begin
  FFieldDataLink.DataSource := Value
end;
end.
```

```
unit DMCalendar;
interface
uses
  Windows, SysUtils, Classes, Controls, Forms,
    Calendar, DB, DBCtrls;
type
  TDMCalendar = class(TCalendar)
  private
    FFieldDataLink: TFieldDataLink;
    function GetDataField: String;
    function GetDataSource: TDataSource;
    procedure SetDataField(const Value: String);
    procedure SetDataSource(const Value: TDataSource);
  protected
    // date changed in table
    procedure DataChange(Sender: TObject);
    // date changed by user in calendar
    procedure Change; override;
    // change data in table
    procedure UpdateData(Sender: TObject);
    procedure CmExit(var Message: TCmExit); message CM_Exit;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
    property DataField: String read GetDataField
      write SetDataField;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DrBob42', [TDMCalendar]);
end;
constructor TDMCalendar.Create(AOwner: TComponent);
begin
  inherited;
  FFieldDataLink := TFieldDataLink.Create;
  FFieldDataLink.OnDataChange := DataChange;
  FFieldDataLink.OnUpdateData := UpdateData
end;
destructor TDMCalendar.Destroy;
begin
  FFieldDataLink.Free;
  FFieldDataLink := nil;
  inherited
end;
```

```
function TDMCalendar.GetDataField: String;
begin
  Result := FFieldDataLink.FieldName
end;
function TDMCalendar.GetDataSource: TDataSource;
begin
  Result := FFieldDataLink.DataSource
end;
procedure TDMCalendar.SetDataField(const Value: String);
begin
  FFieldDataLink.FieldName := Value
end;
procedure TDMCalendar.SetDataSource(const Value:
  TDataSource);
begin
  FFieldDataLink.DataSource := Value
end;
procedure TDMCalendar.DataChange(Sender: TObject);
begin
  if Assigned(FFieldDataLink.Field) then
    if (FFieldDataLink.Field IS TDateField) or
      (FFieldDataLink.Field IS TDateTimeField) then
      CalendarDate := FFieldDataLink.Field.AsDateTime
end;
procedure TDMCalendar.Change;
begin
  FFieldDataLink.Modified;
  inherited
end;
procedure TDMCalendar.UpdateData(Sender: TObject);
begin
  if Assigned(FFieldDataLink.Field) then
    if (FFieldDataLink.Field IS TDateField) or
      (FFieldDataLink.Field IS TDateTimeField) then
      FFieldDataLink.Field.AsDateTime := CalendarDate
end;
procedure TDMCalendar.CmExit(var Message: TCmExit);
begin
  try
    FFieldDataLink.UpdateRecord
  except
    SetFocus;
    raise // re-raise exception
  end;
  inherited
end;
end.
```

➤ *Listing 2: TDMCalendar non-interface implementation.*

unit's `interface` section before you compile your new component.

The `FFieldDataLink` private field is truly used behind the scenes, and must be created when your component is created. The best place to do so is the constructor (and yes, you thus also need to free it inside the destructor). Once the `FFieldDataLink` is available, it's easy to connect the `DataSource` and `DataField` properties to it, since `FDataFieldLink` has a `DataSource` property and a `FieldName` property (which goes to `DataField`). This should result in a first implementation of `TDMCalendar` as can be seen in Listing 1.

Note that we could have used any component here: the example merely shows how to implement data-awareness.

Data-Aware Calendar

To continue with the calendar example, we should realise that apart from the connection to the `DataSource` and `DataField`, we have not written any code to actually connect to the specific field in the dataset. How can we make sure the calendar shows the date as specified by the date field in the dataset? And how do we make sure that the date field in the dataset is updated correctly when we change the day on the calendar? The first question will be answered by using the `OnDataChange` event of the `FFieldDataLink` (which will be fired when the data in the dataset changes, for example when the table is opened, closed or the user navigates through the records in the dataset); the second question is answered by responding to a change in the calendar, and modifying the value in the dataset, as we'll see in a moment.

Let's start with the `FFieldDataLink.OnDataChange` event. We need to write and connect our own event handler, which needs to obtain a date value from the connected field. Fortunately, the internal `FFieldDataLink` has a property named `Field` that, if assigned, points to the actual `TDateField` we need to work with. The code to obtain the date and assign it to the `Calendar` itself consists of only three lines of code:

```
if Assigned(
  FFieldDataLink.Field) then
  if (FFieldDataLink.Field IS
    TDateField) then
    CalendarDate :=
   FFieldDataLink.Field.AsDateTime;
```

Connecting the method `DataChange` that contains these lines to the `OnDataChange` event handler of the `FFieldDataLink` sub-component can be seen in Listing 2.

Two-Way Connection

Apart from the visual calendar showing a new date once the data in the table has changed, we should also make sure the field is changed when the user clicks on the calendar to change the date.

This can be done by using the `OnChange` event of the original `TCalendar` component. Or, better, by overriding the protected `Change` method, which is responsible for calling `OnChange` in the first place.

The `Change` method should make a call to the `Modified` method of the `FFieldDataLink` sub-component. This, in its turn, will trigger the `OnUpdateData` event handler to be fired. And that, in its turn, is the place where we can assign the `CalendarDate` value to the `Field`.

Why all this trouble to call the `OnUpdateData` event handler to do the work? Why not update the `Field` inside the `Change` method? There is a good reason: inside the `OnUpdateData` event handler, the sibling `OnDataChange` event handler is not fired (the one that would indicate a change of the data in the table, which indeed just happened, even if we ourselves were responsible for changing the data).

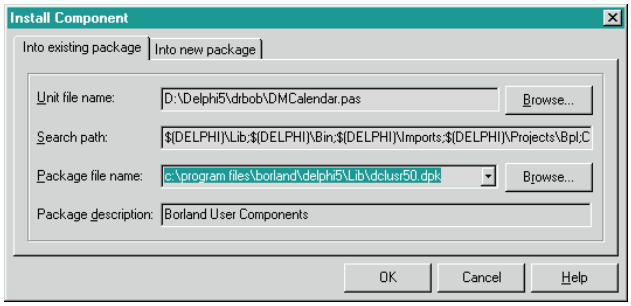
The implementation for the new `OnUpdateData` event handler looks a lot like the code inside the `OnDataChange` event, but assigns the date from `CalendarDate` to the `Field` (instead of the other way around as we did earlier):

```
if Assigned(
  FFieldDataLink.Field) then
  if (FFieldDataLink.Field IS
    TDateField) then
    FFieldDataLink.Field.AsDateTime
      := CalendarDate
```

The complete implementation can be seen in Listing 2 again.

It's now time to install this component in a package, for example the Delphi User Components in dclusr50.dpk (Figure 1). This will result in the `TCalendar` component itself being unavailable (the `Calendar` unit is now contained in both the Sample package dclsmp50.bpl and the Borland User Components dclusr50.bpl). To fix this, you could add the `TCalendar` component to the Delphi User Components as well. Otherwise, you will have a new data-aware calendar, but no original calendar anymore. But that particular issue is not the topic of this column.

➤ *Figure 1: Install TDMCalendar in Delphi User Components.*



Data-Aware Ancestors

The main problem with data-aware controls within Delphi is that they do not share a common ancestor. In fact, it is really hard to determine whether or not a certain control (ie component instance) is a data-aware control. Looking for the presence of the `DataSource` and `DataField` properties is about the only way you can know for certain.

Wouldn't it be a good idea to put the whole data-awareness inside an interface, let's call it `IDataAware`, and then make sure each data-aware component has implemented this interface (in a way that it ends up being data-aware in Delphi indeed)? Now that we've mentioned it, yes, let's go ahead and define an `IDataAware` interface, but start with a short introduction to interfaces first.

Introducing Interfaces

Interfaces are the cornerstone of COM (and, for some, also CORBA) development with Delphi. However, in this article we will not touch COM or CORBA at all. Instead, we'll focus on interfaces as an Object Oriented design principle, and an extension of the ObjectPascal class architecture. An interface is a design specification, literally an interface, but without an implementation. Only when we 'add' an interface to a class definition do we need to provide an implementation. In those cases, the class is said to literally 'implement' an interface.

Unlike class names, which start with a `T` (for type), interface names start with an `I` (for interface). The keyword used to define them is `interface` (versus `class` for classes), and the base class is `IUnknown` (versus `TObject` for classes). Apart from those things, the biggest difference (I can't state this enough) is that an interface contains no implementation: it's a definition of something that will be implemented later.

IHello

As a small example, let's consider the following interface definition:

```
type
  IHello = interface
    procedure HelloWorld;
  end;
```

The `IHello` interface can be used to specify that a class should implement the `HelloWorld` method. So we can use a `TEdit` or `TCalendar` component, and specify that it should implement the `IHello` interface as follows:

```
type
  TEditHello =
    class(TEdit, IHello)
    procedure HelloWorld;
  end;
```

What's missing from this snippet is the implementation of `TEdit.Hello.HelloWorld`. This wasn't needed when we defined the `IHello` interface, since an interface doesn't require an implementation, but now we need to implement `HelloWorld` for `TEditHello`.

Interface Instances

The next step is to create an instance of interfaces and of classes that implement interfaces. You can either create an instance of `TEditHello` (which includes the interface `IHello`) or only create an interface `IHello`:

```
var
  EditHello: TEditHello;
begin
  EditHello :=
    TEditHello.Create;
  EditHello.HelloWorld;
  EditHello.Free
end;
```

Note that you must free the component instance, of course, to avoid a memory leak.

Interface Only

Using interfaces only, we can write the following code, in which we still call the create constructor of the `TEditHello` class, but we only extract an interface (`IHello`) and as a consequence, we can only use the interface methods:

```
var
  EditHello: IHello;
begin
  EditHello :=
    TEditHello.Create;
  EditHello.HelloWorld;
  // next line does not compile
  EditHello.Free
end;
```

As you can see, an interface only sees the interface method (even when it's extracted from a full class: you'll get a compiler error when you try to access `EditHello.Text`, for example). Apart from that, the interface doesn't need to be freed explicitly (the call to `free` results in another compiler error), because reference counting makes sure interfaces are cleaned up whenever they get out of scope.

GetInterface

Sometimes we need to know if an object (a class instance) implements a certain interface. We can do this using the `GetInterface` function, which is defined at the `TObject` level, and returns `true` if the specified interface is implemented by the class instance at hand. The second argument (`Obj`) will obtain the reference to the interface, if it is implemented by the class:

```
if EditHello.QueryInterface(
  IHello, Obj) = S_OK then ...
```

The only thing we need to add to our interfaces for `GetInterface` (and the underlying `QueryInterface`) to work, is a Globally Unique Identifier (a GUID) so `GetInterface` can actually look them up. To define a GUID, just go to the first line of the interface definition and press `Ctrl+Shift+G` to insert a GUID, then you get a unique one:

```
type
  IHello = interface
   ['{B20CF5C0-4042-11D4-B84D-
     444553540000}']
```

As soon as an interface has a GUID, it can be used by the `GetInterface` method.

IDataAware Interface

In the first part of this article, I've shown you how to implement data-aware components or, rather, what properties (and implementation) to add to a class to make it data-aware. And once you have reached the stage where a certain functionality can be described as a signature or pattern, you're ready to abstract from it and define an interface to contain this signature. In our case, the `IDataAware` interface will enable 'normal' controls to implement the 'data-aware' interface. This is not the way it's currently done in Delphi, although I believe it could have been the way to add data-awareness to the VCL in the first place.

As I've mentioned earlier in this article, the main problem with data-aware controls within Delphi is that they do not share a common ancestor. In fact, it is really hard to determine whether or not a certain control (ie component instance) is a data-aware control. Looking for the presence of the `DataSource` and `DataField` properties is about the only way you can know for certain. If, on the other hand, data-awareness was defined by implementing the `IDataAware` interface, then as a consequence of using this `IDataAware` interface, we'd make the process of classifying a component as data-aware very simple: just use `GetInterface` to see if the component indeed implements the `IDataAware` interface and you're done. No question about it. Very elegant (at least I think so).

IDataAware

So, in this case, we need to store the fact that we need the `DataSource` and `DataField` properties inside the interface `IDataAware`, turning the declaration of a data-aware component into the code shown in Listing 3.

And since we already know that we need four additional methods (get and set `DataSource` plus get and set `DataField`) we can add these to the `IDataAware` interface, to enforce the fact that we need to add and implement them to our newly data-aware class. So, using reverse engineering, we can conclude that our `IDataAware` interface must be defined at least as in Listing 4.

Now that this works, we can ask every component instance whether or not it supports the `IDataAware` interface. And, if so, then it's a data-aware component. That's a good reason for using interfaces, right?

Using IDataAware

It's funny that the actual implementation of the `TDMCalendar` does not change much (apart from the `IDataAware` section within the class definition), so all it takes is a little compiler option that I

➤ *Above: Listing 3*

```
type
  TDMCalendar = class(TCalendar, IDataAware)
  published
    property DataSource: TDataSource;
    property DataField: String;
  end;
```

➤ *Below: Listing 4*

```
type
  IDataAware = interface
   ['{FFC47B41-0D51-11D5-8131-00104BF89DAD}']
    function GetDataSource: TDataSource;
    procedure SetDataSource(Value: TDataSource);
    function GetDataField: string;
    procedure SetDataField(const Value: string);
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    property DataField: String read GetDataField write SetDataField;
  end;
```

named INTERFACE to conditionally compile the unit in Listing 5 with or without IDataAware interface support. We can now install the IDataAware version of TDMCalendar in the Delphi User Components package, and use it just like any other data-aware component (but one that connects to a date field).

Testing For IDataAware

For a given component, we can now call the GetInterface function (defined at the object level), which returns true if the interface is implemented, and then passes the interface itself in the second argument. The code in Listing 6 checks the component DMCalendar1 to see if it implements the IDataAware interface and, if so, puts the interface in the DW argument. If it succeeds, we use the DataSource property from the interface to get to the DataSet and the ClassName of the DataSet, see Listing 6.

Since our DMCalendar1 component indeed implements the IDataAware interface, the code in Listing 6 shows TTable (in case you connected it to a TTable component, which I just did). Of course, you can use this technique to walk through a long list of components and for each one determine if it indeed implements the IDataAware interface, and if so use the interface (and especially the DataSource and DataField properties, since these are the only ones that matter).

The example application (see Figure 2) on disk contains an application that connects the TDMCalendar component to one of the date fields of the orders table, and also uses a button with the OnClick event implemented just like the source snippet in Listing 6 (showing the type of the table that connects to the orders.db). The TDBEdit shows the date field in text format, while the TDMCalendar component shows the date on the calendar itself.

Next Time

This month we have spent a lot of time exploring data-aware components. Next month I aim to continue with a somewhat related topic: dbExpress, the new cross-platform data access layer which is available in Kylix for Linux and will be made available in the next version of Delphi for Windows as well.

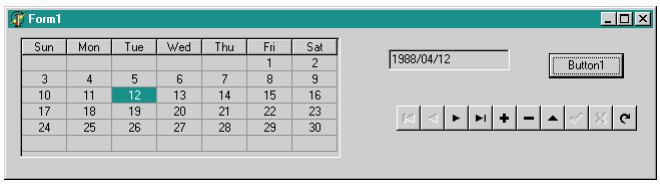
We will see how dbExpress works, what the relationship with

➤ *Listing 5: TDMCalendar interface implementation.*

```
{$DEFINE INTERFACE}
unit DMCalendar;
interface
uses
  Windows, SysUtils, Classes, Controls, Forms,
    Calendar, DB, DBCtrls;
{$IFDEF INTERFACE}
type
  IDataAware = interface
   ['{FFC47B41-0D51-11D5-8131-00104BF89DAD}']
    function GetDataSource: TDataSource;
    procedure SetDataSource(const Value: TDataSource);
    function GetDataField: string;
    procedure SetDataField(const Value: string);
    property DataSource: TDataSource read GetDataSource
       write SetDataSource;
    property DataField: String read GetDataField
      write SetDataField;
  end;
{$ENDIF}
type
  TDMCalendar = class(TCalendar {$IFDEF INTERFACE},
    IDataAware{$ENDIF})
  private
    FFieldDataLink: TFieldDataLink;
    function GetDataField: String;
    function GetDataSource: TDataSource;
    procedure SetDataField(const Value: String);
    procedure SetDataSource(const Value: TDataSource);
  protected
    // date changed in table
    procedure DataChange(Sender: TObject);
    // date changed by user in calendar
    procedure Change; override;
    // change data in table
    procedure UpdateData(Sender: TObject);
    procedure CmExit(var Message: TCmExit); message CM_Exit;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
    property DataField: String read GetDataField
      write SetDataField;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DrBob42', [TDMCalendar]);
end;
constructor TDMCalendar.Create(AOwner: TComponent);
begin
  inherited;
  FFieldDataLink :=
  TFieldDataLink.Create;
  FFieldDataLink.OnDataChange := DataChange;
  FFieldDataLink.OnUpdateData := UpdateData
end;
destructor TDMCalendar.Destroy;
begin
  FFieldDataLink.Free;
  FFieldDataLink := nil;
  inherited
end;
function TDMCalendar.GetDataField: String;
begin
  Result := FFieldDataLink.FieldName
end;
function TDMCalendar.GetDataSource: TDataSource;
begin
  Result := FFieldDataLink.DataSource
end;
procedure TDMCalendar.SetDataField(const Value: String);
begin
  FFieldDataLink.FieldName := Value
end;
procedure TDMCalendar.SetDataSource(const Value:
  TDataSource);
begin
  FFieldDataLink.DataSource := Value
end;
procedure TDMCalendar.DataChange(Sender: TObject);
begin
  if Assigned(FFieldDataLink.Field) then
    if (FFieldDataLink.Field IS TDateField) or
      (FFieldDataLink.Field IS TDateTimeField) then
      CalendarDate := FFieldDataLink.Field.AsDateTime
end;
procedure TDMCalendar.Change;
begin
  FFieldDataLink.Modified;
  inherited
end;
procedure TDMCalendar.UpdateData(Sender: TObject);
begin
  if Assigned(FFieldDataLink.Field) then
    if (FFieldDataLink.Field IS TDateField) or
      (FFieldDataLink.Field IS TDateTimeField) then
      FFieldDataLink.Field.AsDateTime := CalendarDate
end;
procedure TDMCalendar.CmExit(var Message: TCmExit);
begin
  try
    FFieldDataLink.UpdateRecord
  except
    SetFocus;
    raise // re-raise exception
  end;
  inherited
end;
end.
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DW: IDataAware;
begin
  if DMCalendar1.GetInterface(IDataAware, DW) then
    ShowMessage(DW.DataSource.DataSet.ClassName)
  else
    ShowMessage('no data-aware component');
end;
```

➤ *Listing 6*



➤ *Figure 2*

MIDAS is and, last but not least, see how we can migrate our existing database code to dbExpress. As a final bonus, I will show what is needed to make the `TDMCalendar` component we developed this month work in Kylix.

All this and more in the next issue, so stay tuned...

Bob Swart (aka Dr.Bob, www.drbob42.com) is an IT Consultant for the Kylix/Delphi OplossingsCentrum (KDOC.nl) and a freelance technical author.