Under Construction: Property Editors

by Bob Swart

Delphi offers a Tools API, which allows us to extend the functionality of the Delphi IDE itself. There are four different Tools API interfaces: for Experts (see Issue 3), Version Control Systems, Component Editors and Property Editors. They offer us the functionality we need to add new IDE features or enhance the existing ones!

Property Editors

Property editors are, like Experts and Version Control Systems, in this sense extensions of the Delphi IDE. That may sound very difficult or complex, but is in fact very easy. You can even write a property editor without knowing it – for enumerated types, for example. Remember the `color` property of a `TForm`? When you want to enter a value, you get a drop-down list showing all possible choices. That's an enumerated type property editor, a very easy one, and we can make one with only a few lines of code.

As an example, I've picked the Starfleet rank overview (as a confirmed *Trekkie* what do you expect!). Or, in Object Pascal, the enumerated type which is shown in Listing 1.

In this case, like all other enumerated types, we must make sure that the user cannot make a mistake, by typing "commonder" instead of "commander" for example. We need to offer a list of limited choices: a drop-down list. Well, that's exactly what the user gets when s/he drops the `TOfficer` component shown in Listing 2 onto a form (see Figure 1).

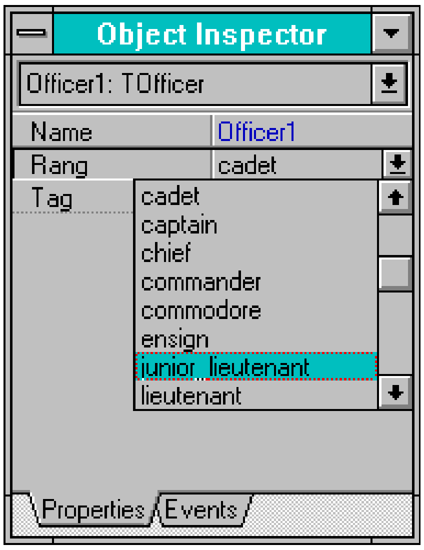
So, we haven't done anything special but our first personal property editor is up and running! And who knows, it might even be your second or third, now that you think about it...

There's actually much more to property editors than meets the eye and we've only scratched the surface. Let's look deeper and see if we can do even more. To do that, we need to check out the one Tools API source file which defines the behaviour of the property editors for which you do need to write code: DSGNINTF.PAS (in directory \DELPHI\SOURCE\VCL). This file contains not only the definition for the base class `TPropertyEditor`, but also numerous derived property editors for the properties of the VCL components of Delphi itself. Most of these property editors are available for our use as well, like the `TEnumProperty` we used for the `TRang` enumerated type in the first example.

Existing Property Editors

Before we actually take a look at a property editor from the inside, let's first examine what kinds of property editors already exist in Delphi. To do that, start a new project, add `uses DsgnIntf;` to the `implementation` section, compile, open the browser and search for `TPropertyEditor` – see Figure 2.

➤ *Figure 1*



➤ *Listing 1*

```
Type
  TRang = (cadet, midshipman, chief, ensign,
           junior_lieutenant, lieutenant,
           lieutenant_commander, commander,
           captain, commodore, admiral);
```

➤ *Listing 2*

```
unit Officer;
interface
uses Classes;
Type
  TRang = (cadet, midshipman, chief, ensign, junior_lieutenant, lieutenant,
    lieutenant_commander, commander, captain, commodore, admiral);
  TOfficer = class(TComponent)
  private
    FRang: TRang;
  published
    property Rang: TRang read FRang write FRang;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TOfficer]);
end;
end.
```

If I count correctly, there are at least 21 property editors registered by the DSGNINTF unit. Note, however, that there are actually even more property editors available, like the `TPictureEditor` in \DELPHI\LIB\PICEDIT.DCU (I'll return to this later on...).

TPropertyEditor

What does a property editor look like? Well, like an expert, it is derived from a base class, from which we need to override some methods in order to make things work our way. The `TPropertyEditor` base class is defined as shown in Listing 3 (I've left out the `private` parts, as we can't touch them anyway).

A `TPropertyEditor` edits a property of a component, or a list of components, selected into the Object Inspector. The property editor is created based on the type of the property being edited, as determined by the types registered by `RegisterPropertyEditor`. A `TPropertyEditor` is used by the Object Inspector whenever a property is modified.

For now, we will just focus on a subset of the methods which can be overridden to change the behaviour of the property editor (we'll get back to the others in a future instalment of this column).

`GetAttributes` is the most important method, as it determines the kind of property editor and its behaviour. There are three kinds of property editors (other than the default editbox-type): a dropdown value list (we've seen that one before), a sub-property list and a dialog. `GetAttributes` returns a set of type `TPropertyAttributes`:

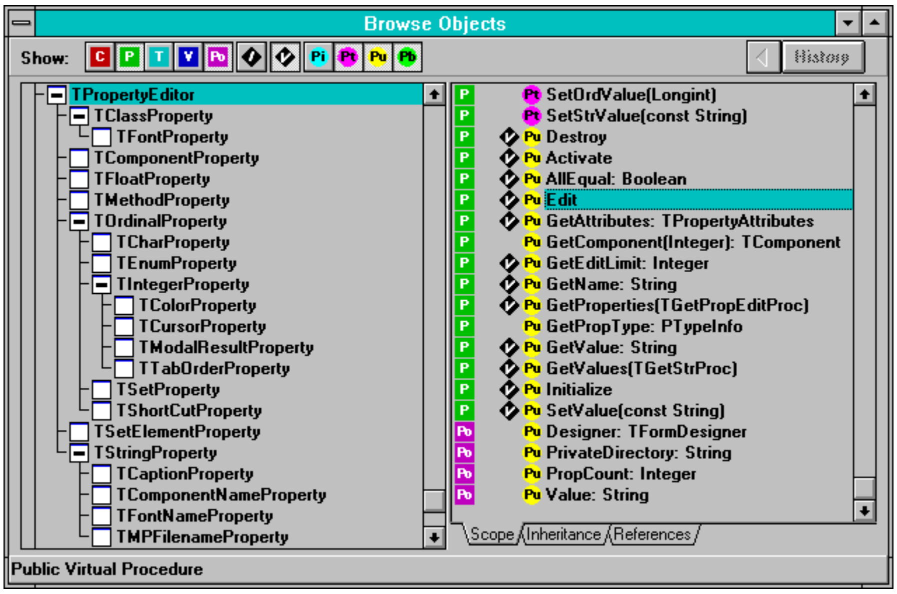
➣ `paValueList`: The property editor can return an enumerated list of values for the property. If `GetValues` calls `Proc` with values then this attribute should be set. This will cause the dropdown button to appear to the right of the property in the Object Inspector.

➣ `paSubProperties`: The property editor has sub-properties which will be displayed indented and below the current property in standard outline format. If `GetProperties` will generate property objects then this attribute should be set.

➣ `paDialog`: Indicates that the Edit method will bring up a dialog. This will cause the '...' button to be displayed to the right of the property in the Object Inspector.

➣ `paSortList`: The Object Inspector will sort the list returned by `GetValues` (by name).

➣ `paAutoUpdate`: Causes the `SetValue` method to be called on each change made to the editor instead of after the change has been approved (eg the `Caption` property).



```
TPropertyEditor
  TClassProperty
    TFontProperty
  TComponentProperty
  TFloatProperty
  TMethodProperty
  TOrdinalProperty
    TCharProperty
    TEnumProperty
    TIntegerProperty
      TColorProperty
      TCursorProperty
      TModalResultProperty
      TTabOrderProperty
    TSetProperty
    TShortCutProperty
  TSetElementProperty
  TStringProperty
    TCaptionProperty
    TComponentNameProperty
    TFontNameProperty
    TMPFilenameProperty
```

```
P  Pt SetOrdValue(Longint)
P  Pt SetStrValue(const String)
P  Pu Destroy
P  Pu Activate
P  Pu AllEqual: Boolean
P  Pu Edit
P  Pu GetAttributes: TPropertyAttributes
P  Pu GetComponent(Integer): TComponent
P  Pu GetEditLimit: Integer
P  Pu GetName: String
P  Pu GetProperties(TGetPropEditProc)
P  Pu GetPropType: PTypeInfo
P  Pu GetValue: String
P  Pu GetValues(TGetStrProc)
P  Pu Initialize
P  Pu SetValue(const String)
Pb Pu Designer: TFormDesigner
Pb Pu PrivateDirectory: String
Pb Pu PropCount: Integer
Pb Pu Value: String
```

Public Virtual Procedure

➤ *Figure 2*

```
Type
  TPropertyEditor = class
  protected
    function GetPropInfo: PPropInfo;
    function GetFloatValue: Extended;
    function GetFloatValueAt(Index: Integer): Extended;
    function GetMethodValue: TMethod;
    function GetMethodValueAt(Index: Integer): TMethod;
    function GetOrdValue: Longint;
    function GetOrdValueAt(Index: Integer): Longint;
    function GetStrValue: string;
    function GetStrValueAt(Index: Integer): string;
    procedure Modified;
    procedure SetFloatValue(Value: Extended);
    procedure SetMethodValue(const Value: TMethod);
    procedure SetOrdValue(Value: Longint);
    procedure SetStrValue(const Value: string);
  public
    destructor Destroy; override;
    procedure Activate; virtual;
    function AllEqual: Boolean; virtual;
    procedure Edit; virtual;
    function GetAttributes: TPropertyAttributes; virtual;
    function GetComponent(Index: Integer): TComponent;
    function GetEditLimit: Integer; virtual;
    function GetName: string; virtual;
    procedure GetProperties(Proc: TGetPropEditProc); virtual;
    function GetPropType: PTypeInfo;
    function GetValue: string; virtual;
    procedure GetValues(Proc: TGetStrProc); virtual;
    procedure Initialize; virtual;
    procedure SetValue(const Value: string); virtual;
    property Designer: TFormDesigner read FDesigner;
    property PrivateDirectory: string read GetPrivateDirectory;
    property PropCount: Integer read FPropCount;
    property Value: string read GetValue write SetValue;
  end;
```

➤ *Listing 3*

➢ paMultiSelect: Allows the property to be displayed when more than one component is selected. Some properties are not appropriate for multi-selection (eg the Name property).

➢ paReadOnly: The value is not allowed to change.

GetValue returns the string value of the property. By default this returns (unknown). This should be overridden to return the appropriate value. GetValues is called when paValueList is returned in GetAttributes. It should call the argument Proc for every value which is acceptable for this property. TEnumProperty will pass every element in the enumeration, as we can see in Figure 1.

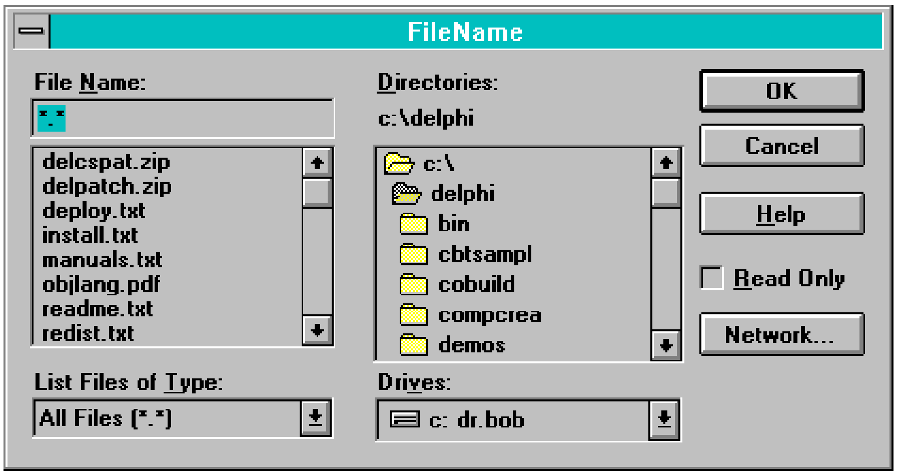
SetValue(Value) is called to set the value of the property. The property editor should be able to translate the string and call one of the SetXxxValue methods. If the string is not in the correct format or not an allowed value, the property editor should generate an exception describing the problem. SetValue can ignore all changes and allow all editing of the property to be accomplished through the Edit method (eg the Picture property).

Edit is called when the '...' button is pressed or the property is double-clicked. This can, for example, bring up a dialog to allow the editing the property in some more meaningful fashion than by text (eg the Font property).

TFileNameProperty

With these few basic methods we now have enough power at our disposal to write our first non-trivial property editor: an open filename dialog property editor for filename properties.

We must remember that writing components is essentially a non-visual task and writing property editors is no different. We have to write a new unit by hand in the editor (see Listing 4). We need to specify that we want a dialog type of property editor, so we return [paDialog] in the GetAttributes function. Then, we can do as we like in the Edit procedure, which in this case involves a TOpenDialog to let us find any existing file.


FileName

File Name: *.*

delcspat.zip
delpatch.zip
deploy.txt
install.txt
manuals.txt
objlang.pdf
readme.txt
redist.txt

Directories: c:\delphi

c:\
delphi
bin
cbtsampl
cobuild
compcrea
demos

List Files of Type: All Files (*.*)

Drives: c: dr.bob

☐ Read Only

➤ *Figure 3*

```
unit FileName;
interface
uses DsgnIntf;
Type
  TFileNameProperty = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
implementation
uses
  Dialogs, Forms;
function TFileNameProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TFileNameProperty.Edit;
begin
  with TOpenDialog.Create(Application) do
  try
    Title := GetName; { name of property as OpenDialog caption }
    Filename := GetValue;
    Filter := 'All Files (*.*)|*.*';
    HelpContext := 0;
    Options := Options + [ofShowHelp, ofPathMustExist, ofFileMustExist];
    if Execute then SetValue(Filename);
  finally
    Free
  end
end {Edit};
end.
```

➤ *Listing 4*

Note that we call the GetName function of the property editor to get the name of the actual property for which we want to fire up the TOpenDialog. For a property called FileName, this would result in the example shown in Figure 3.

In just a few lines of code we've written a TFileName property editor which will give great support at design time for all our components which use a property of type TFileName. This illustrates that property editors have an enormous potential for designers of Delphi components.

TFileModeProperty

We've seen how we can create and execute a simple TOpenDialog as a property editor. But instead of just a TOpenDialog component, we can of course show a complete newly designed form in our property editor's Edit method.

Let's design a FileMode property editor in which we can specify both the file access (read-only, write-only, read-write) and file sharing (deny-none, deny-read, deny-write, deny-all) values for a file. The code for a simple form to enable us to pick these options using two TRadioGroup controls is shown in Listing 5.

I've used two TRadioGroups to hold the items we can choose from in a list, just like a listbox or combobox. I think the TRadioGroup is probably one of the most underestimated components on the palette: one RadioGroup is capable of showing several radio buttons which are all easily accessible.

The form is shown in action in Figure 4, specifying a read-only deny-none filemode.

To turn this form into a property editor, we only need to activate it at the right time (ie in the Edit method of the TFileModeProperty editor), and set and get the actual value of the filemode property we need. See Listing 6.

TFileInfo

To illustrate the use of this TFileModeProperty editor, I've designed a new component called TFileInfo, derived from a standard non-visual TComponent, with two new properties: FileName and FileMode. The first one is connected to the TFileName property editor we created earlier, while the second one uses the TFileMode property editor to interactively set the desired FileMode. See Listing 7.

Note that this component registers its own property editors in the Register procedure where the component itself is registered. Figure 5 shows the TFileInfo component in action with the two new property editors.

TBUUCode

While TFileInfo is but an example component, we can easily extend the TBUUCode components for file UUEncoding and UUDecoding from the last few issues, using these property editors for the InputFile property.

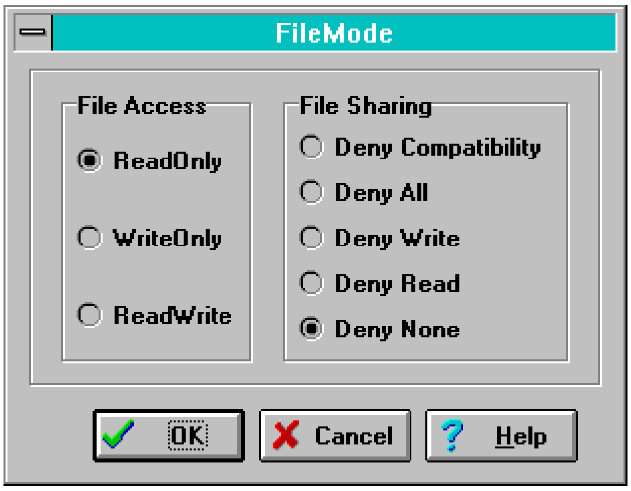
In fact, I've already done so and the new source code for the TBUUEnCode and TBUUDeCode components is (again) included on the subscribers' disk with this issue.

TPictureEditor

We've already seen how to make picture editors that behave like dialogs. And this reminds me of the most irritating property editor in Delphi: the picture editor for glyphs, icon, bitmaps etc.

It's not the fact that it doesn't work, it's just the fact that it isn't very user friendly. If I click on the Load button I get a TOpenDialog that gives me the option to select a .BMP, .ICO or whatever file I wish. However, I don't get to see what's actually inside this file until I close the TOpenDialog. Then I'm back in the Picture Editor and if I decide it's not the file I really want I have to click on the Load button again

➤ Figure 4


FileMode

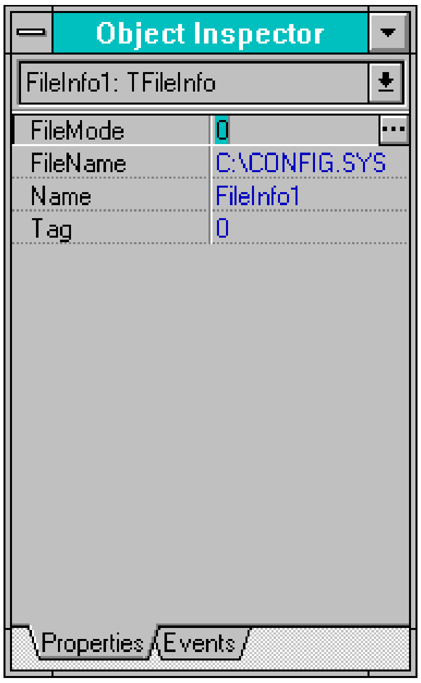
File Access: ReadOnly [✓] · WriteOnly · ReadWrite

File Sharing: Deny Compatibility · Deny All · Deny Write · Deny Read · Deny None [✓]

```
unit FileMode;
interface
uses
  Forms, Buttons, StdCtrls, ExtCtrls;
Type
  TFileModeDlg = class(TForm)
    OKBtn: TBitBtn;
    CancelBtn: TBitBtn;
    HelpBtn: TBitBtn;
    Bevel1: TBevel;
    FileAccess: TRadioGroup;
    FileSharing: TRadioGroup;
    procedure OKBtnClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  public
    FileShareMode: Word;
  end;
implementation
{$R *.DFM}
procedure TFileModeDlg.FormActivate(Sender: TObject);
begin
  FileAccess.ItemIndex := (FileShareMode MOD $10);
  FileSharing.ItemIndex := (FileShareMode SHR 4)
end;
procedure TFileModeDlg.OKBtnClick(Sender: TObject);
begin
  FileShareMode := FileAccess.ItemIndex + (FileSharing.ItemIndex SHL 4)
end;
end.
```

➤ Listing 5

➤ Figure 5


Object Inspector

FileInfo1: TFileInfo

FileMode	0
FileName	C:\CONFIG.SYS
Name	FileInfo1
Tag	0

```
unit FileProp;
interface
uses DsgnIntf;
Type
  TFileModeProperty = class(TIntegerProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
implementation
uses
  SysUtils, Controls, FileMode;
function TFileModeProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TFileModeProperty.Edit;
begin
  with TFileModeDlg.Create(nil) do
  try
    FileShareMode := GetOrdValue;
    if ShowModal = mrOk then
      SetOrdValue(FileShareMode)
  finally
    Free
  end
end {Edit};
end.
```

➤ *Listing 6*

```
unit FileInfo;
interface
uses
  Classes, SysUtils;
Type
  TFileInfo = class(TComponent)
  private
    FFileName: TFileName;
    FFileMode: Word;
  published
    property FileName: TFileName read FFileName write FFileName;
    property FileMode: Word read FFileMode write FFileMode;
  end;
procedure Register;
implementation
uses
  DsgnIntf, FileProp, FileName;
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TFileInfo]);
  RegisterPropertyEditor(TypeInfo(Word), TFileInfo, 'FileMode',
     TFileModeProperty);
  RegisterPropertyEditor(TypeInfo(TFilename), nil, '', TFilenameProperty)
end;
end.
```

➤ *Listing 7*

and start all over again. It's especially annoying if you have to browse through a directory with many TBitBtn bitmap files.

I would like a *preview* option, so I can see what the image in a file looks like *while* I'm browsing through a directory! It sounds exactly like a new property editor to me (Note: since Borland didn't provide us with the source of PICEDIT.DCU, we don't have the PICEDIT.DFM form either and have to write our own picture editor instead of enhancing the already existing one).

TImageForm

First of all, we have to design the actual dialog or form which will be used by our new property editor. Figure 6 shows my form: the area where the image of Dr.Bob is shown, in the lower-right corner, is used to display the image of any file which is currently selected in the file listbox. Depending on your needs, you can even stretch this image (not recommended for little TBitBtn bitmap images, but useful if you have large bitmaps and only want a preview).

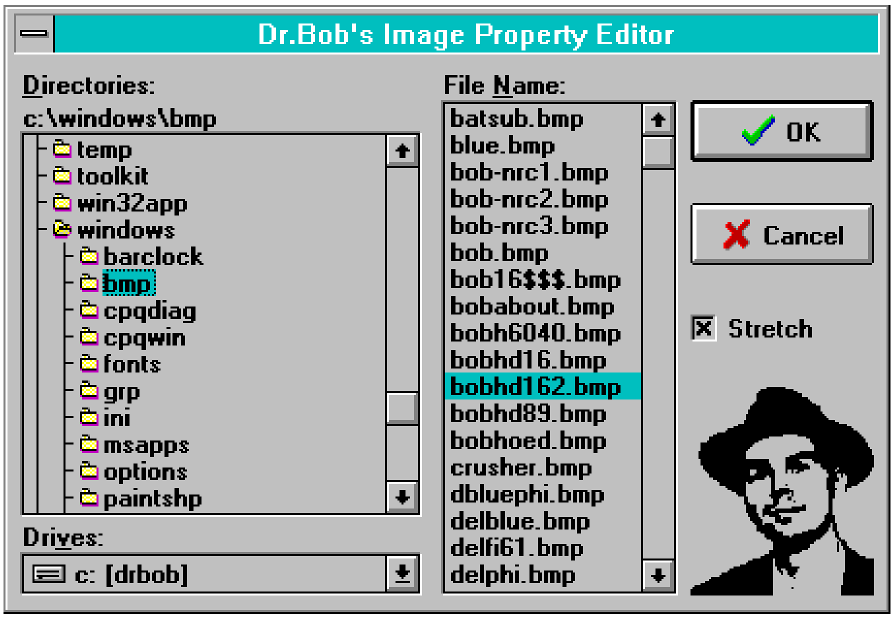
Note that I've used the component TDirectoryOutliner (from my *Performance Optimisation* articles in Issues 4 and 5) in this form. The complete source code is on the disk with this issue, but I've also uploaded the .DCU version of this property editor to the DELPHI forum on CompuServe, so if you don't have a subscription yet (shame on you!) then you can at least get it to work.

Now that we have a form to ask for which image to use, let's see how we can get this to work as a property editor. We need to take a look at GRAPHIC.PAS to see what kinds of pictures, glyphs and images exist in the first place. It seems we are limited to two descendants of TPersistent: TPicture and TGraphic, with descendent TBitmap of TGraphic. For this column, let's just focus on .BMP files, and hence on the TPicture and TBitmap classes only. This means we want to offer the new Image Property Editor for properties of type TPicture and TBitmap. See Listing 8.

Note that since we don't explicitly want the TPictureEditor to belong to one specific component, we have to register it ourselves here, and install it just like any other custom component or expert from the Delphi IDE's Options | Install Components dialog. After rebuilding your COMPLIB.DCL (remember to make that backup first!), you will get the new picture editor for each TPicture (in a TImage) or TBitmap (for example in a TSpeedButton or TBitBtn).

One last important thing: Delphi already had a property editor installed for TPictures and TBitmaps, namely the picture editor Borland provided. Won't we get into trouble if we want to use our own?

No, we won't, because it seems that the last property editor which has been registered for a particular component or property type will override the previous one. So, if you ever install another property


Dr.Bob's Image Property Editor

Directories:
c:\windows\bmp
temp
toolkit
win32app
windows
barclock
bmp
cpqdiag
cpqwin
fonts
grp
ini
msapps
options
paintshp

Drives:
c: [drbob]

File Name:
batsub.bmp
blue.bmp
bob-nrc1.bmp
bob-nrc2.bmp
bob-nrc3.bmp
bob.bmp
bob16$$$.bmp
bobabout.bmp
bobh6040.bmp
bobhd16.bmp
bobhd162.bmp
bobhd89.bmp
bobhoed.bmp
crusher.bmp
dbluephi.bmp
delblue.bmp
delfi61.bmp
delphi.bmp

➤ *Figure 6*

```
unit PictEdit;
interface
uses DsgnIntf;
Type
  TPictureEditor = class(TClassProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
  procedure Register;
implementation
uses
  SysUtils, Controls, Graphics, TypInfo, ImageFrm;
function TPictureEditor.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TPictureEditor.Edit;
begin
  with TImageForm.Create(nil) do
  try
    ImageDrBob.Picture := TPicture(GetOrdValue);
    if ShowModal = mrOk then begin
      if (GetPropType^.Name = 'TPicture') then
        SetOrdValue(LongInt(ImageDrBob.Picture))
      else { Bitmap }
        SetOrdValue(LongInt(ImageDrBob.Picture.Bitmap))
    end
  finally
    Free
  end
end {Edit};
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TPicture), nil, '', TPictureEditor);
  RegisterPropertyEditor(TypeInfo(TBitmap), nil, '', TPictureEditor)
  end;
end.
```

➤ *Listing 8*

editor for `TBitmap`s, for example, you will override the one we've just built.

More Property Editors

So far, we've only seen a few of the possible kinds of property editors we can write. We've focussed especially on `paDialog` property editors – in my view the easiest way to customise the entry of property values at design time. However, there are many more kinds of property editors and ways to write them, so we'll just have to come back later to explore these in detail in future columns. For now, I hope I've given you enough to chew on for a month!

Next Time

Next time we'll focus on a type of component we haven't touched at all so far: data-aware components. We'll explore how they work, learn what makes them tick and even develop one of our own: a data-aware multi-media player. After that, we'll slowly return to the subject of Tools APIs and Delphi IDE Experts when we start exploring the so-called Component Editors in the April issue.

Stay tuned and make sure you've always got a backup of your COMPLIB.DCL in a *save* place!

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Delphi and sometimes a bit of Pascal or C++. In his spare time, he likes to watch video tapes of Star Trek Voyager with his almost two year old son Erik Mark Pascal.