Under Construction: Component Editors

by Bob Swart

Delphi offers a Tools API, which allows programmers to extend the functionality of the Delphi IDE itself. There are four different Tools API interfaces: for Experts (see Issues 3 and 7), Version Control Systems (more on those in a later column), Property Editors (see Issue 6) and finally Component Editors. They offer us the functionality we need to add new or enhance existing IDE features.

Component editors exist to allow the developer to edit or amend certain aspects of the behaviour of components used in his or her application. Like property editors, they are basically derived from a single base class, where some abstract methods need to be overridden and re- defined in order to give the component editor the desired behaviour. Unlike property editors, however, component editors are component-specific, and not property-specific. They are bound to a particular component type, and are generally executed by a click with the right mouse button on the component when it's been dropped on a form. The mouse click generally activates a pop-up menu: the context menu. This activation method is different to property editors, but otherwise the process of writing a component editor is essentially the same.

A default component editor is created for each component that is selected in the form designer based on the component's type (see also `GetComponentEditor` and `RegisterComponentEditor` in the file DSGNINTF.PAS). When the component is double-clicked the `Edit` method is called. When the context menu for the component is invoked the `GetVerbCount` and `GetVerb` methods are called to build the context menu. If one of the verbs is selected the `ExecuteVerb` method is called. `Paste` is called whenever the component is pasted to the clipboard. You only need to create your own component editor if you wish to add verbs to the context menu, change the default double-click behaviour, or add an additional clipboard format.

The class definition for the base class `TComponentEditor` can be found in DSGNINTF.PAS and is shown in Listing 1 (for both Delphi 1.0 and Delphi 2.0).

There are six virtual methods which can be overridden by the component editor developer, which leads to a somewhat easier editor building process compared to property editors.

➤ Create

`Create(AComponent, ADesigner)` is called to create the component editor. `AComponent` is the component to be edited by the editor. `ADesigner` is an interface to the designer to find controls and create methods (this is not used often).

➤ Edit

Called when the user double-clicks the component. The component editor can bring up a dialog in response to this method, for example, or some kind of design expert. If `GetVerbCount` is greater than zero, `Edit` will execute the first verb in the list (`ExecuteVerb(0)`).

➤ ExecuteVerb(Index)

The verb at location `Index` was selected by the user from the context menu. The meaning of this is determined by component editor.

➤ GetVerb

The component editor should return a string that will be displayed in the context menu. It is the responsibility of the component editor to place the `&` (accelerator key) character and the `'...'` characters as required in each case.

➤ GetVerbCount

The number of valid indices to `GetVerb` and `ExecuteVerb`, assumed to be zero based (that is, `0..GetVerbCount-1`).

➤ Copy

Called when the component is being copied to the clipboard. The component's filed image is already on the clipboard. This gives the component editor a chance to paste a different type of format which is ignored by the designer but might be recognised by another application.

➤ *Listing 1*

```
Type
  TComponentEditor = class
  private
    FComponent: TComponent;
    FDesigner: TFormDesigner;
  public
    constructor Create(
      AComponent: TComponent; ADesigner: TFormDesigner); virtual;
    procedure Edit; virtual;
    procedure ExecuteVerb(Index: Integer); virtual;
    function GetVerb(Index: Integer): string; virtual;
    function GetVerbCount: Integer; virtual;
    procedure Copy; virtual;
    property Component: TComponent read FComponent;
    property Designer: TFormDesigner read FDesigner;
  end;
```

```
TObject
  TPersistent
    TComponent
      TControl
        TGraphicControl
          TWinControl
            TCustomControl
```

➤ *Figure 1*

Default Component Editor

Apart from the general type `TComponentEditor`, there is also a default component editor which is used by most components unless another component editor is installed to override it. The `TDefaultEditor` implements `Edit` to search the properties of the component and to generate the (or navigate to an already existing) `OnCreate`, `OnChange` or `OnClick` event, whichever it finds first, or just the first alphabetic event handler which is available – see Listing 2.

Whenever the component editor modifies the component it must call the `Designer.Modified` method to inform the designer that the component on the form has been modified. If we only use the component editor to display some information (like a general About Box, for example) there is no need to inform the designer.

Custom Component Editors

When building custom components, there are generally a few possible base classes that can be considered. Every class from the VCL is derived from the `TObject` root. The class `TObject` contains the `Create` and `Destroy` methods that are needed to create and destroy instances of classes. The class `TPersistent`, derived from `TObject`, contains methods for reading and writing properties to and from a form file. `TComponent` is the class to derive all components from, as it contains the methods and properties that allow Delphi to use `TComponent` classes as design elements, view their properties with the Object Inspector and place these components in the Component Palette. See Figure 1 for the class hierarchy.

If we want to create a new non-visual component from scratch, then `TComponent` is the class we need to derive from. Visual component classes are derived from the `TControl` class, which already contains the basic functionality for visual design components, such as position, visibility, font and caption. Derived from `TControl` are `TGraphicControl` and `TWinControl`. The key difference between a `TGraphicControl` and a `TWinControl` is that a `TWinControl` contains an actual Windows handle, while a `TGraphicControl` does not. Therefore, derived from a `TWinControl` we will find classes such as the standard Windows controls, whilst controls like `TBevel`, `TImage`, `TSpeedButton` and `TShape` are derived from `TGraphicControl`. Finally, the class `TCustomControl` is much like both `TWinControl` and `TGraphicControl` together.

Why do we need to rehash this information, which we've already covered in previous columns? Well, basically, because we need to understand when the default component editor is useful in order to be able to determine when to write a custom component editor of our own. It would seem that since the default component editor is capable of generating event handler skeleton code for the `OnCreate`, `OnChange` or `OnClick` event, we should at first look for components that don't have or need these events. Like, for example, a component that does not have a Windows handle (ie does not need the input focus at any given time), and certainly non-visual components derived from `TComponent`.

Another class of components where a custom component editor would be handy are the dialog components, such as `TOpenDialog`, `TSaveDialog`, `ColorDialog`, `PrintDialog` and `PrinterSetupDialog`. The `FontDialog`, `FindDialog` and `ReplaceDialog` do already have some events, so these don't apply (the default component editor will put you in the code editor to write one of their event handlers). The five interesting dialogs have no events and hence no default behaviour for the default component editor.

What could a component editor mean to these dialogs? Well, a preview of what they'd look like at run time would be nice. Have you never had the need to know if you've set all the required properties for `TOpenDialog`? The only way you can check is by running you application. Wouldn't it be much simpler to just double-click on the dialog to get it to preview itself in its current state? Yes, I think so too and that's what we'll be doing in the rest of this article.

For this, we only need to override the edit method of the `TComponentEditor` class, see what kind of class our component actually is (using run-time type information, or RTTI) and then `Execute` it, as shown in Listing 3.

Note that the `inherited Edit` is needed to make sure the default component editor behaviour is still triggered when we are not one of the five classes derived from `TCommonDialog`. Since we're installing this component editor for all derived classes of `TCommonDialog` we need to ensure that all other classes (except the five we want) still get their default behaviour.

➤ *Listing 2*

```
Type
  TDefaultEditor = class(TComponentEditor)
  private
    FFirst: TPropertyEditor;
    FBest: TPropertyEditor;
    FContinue: Boolean;
    procedure CheckEdit(
      PropertyEditor: TPropertyEditor);
  protected
    procedure EditProperty(PropertyEditor: TPropertyEditor;
      var Continue, FreeEditor: Boolean); virtual; public
    procedure Edit; override;
  end;
```

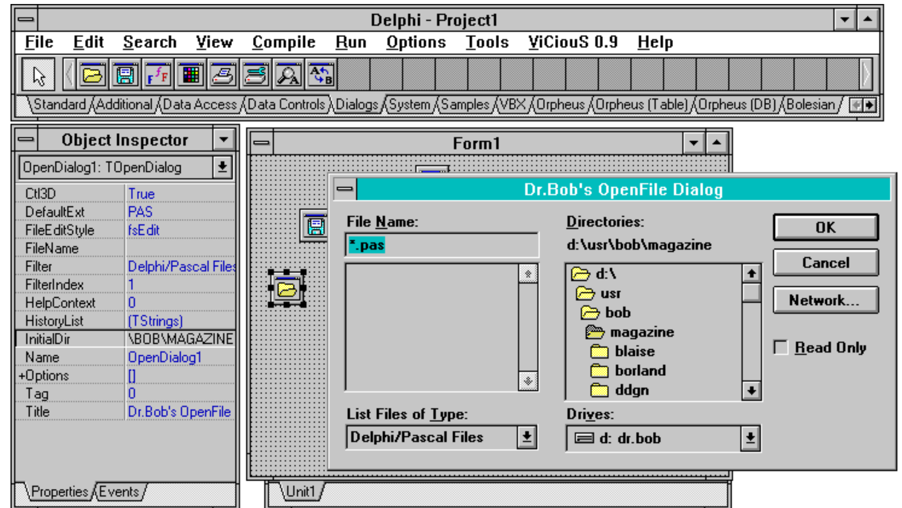
A component editor must be registered, just like components or property editors. However, it's much simpler compared to registering property editors, since we only need two parameters to function `RegisterComponentEditor`. The first one is the name (type) of the component for which this component editor is intended (`TDialog` in our case) and the second parameter is the type of the component editor itself (`TDialogEditor` in our case).

Installing a component editor is again much like installing a component or property editor: just add it to COMPLIB.DCL. Adding this component editor to the COMPLIB.DCL where the corresponding component `TXXXDialog` is already installed leads to the `Execute` method being executed at design-time.

A component editor can be created for a single class, or for a set of classes (all derived classes are included). In our case, the component editor is valid and meant for a specific set of derived classes from `TCommonDialog`, see Listing 4.

After we've installed this `TCommonDialogComponentEditor` in COMPLIB.DCL we can drop a `TOpenDialog` on a form and double-click on it. An instant preview will appear with all the properties exactly as they've been defined in the Object Inspector at run-time (Figure 2). So, has Delphi suddenly become an interpreter, or what?

Unfortunately, if we try to test our component editor on the `TFontDialog`, for example, the default behaviour (editing the `OnApply` event) does not happen. We do call the inherited `Edit` method, but it's the `TComponentEditor.Edit` method that we're calling, not the `Edit` method from the default component editor (`TDefaultEditor`). There are two ways to fix this. We could either make sure we install the `TDialogEditor` only for the five specific classes derived from `TCommonDialog` that we need, or we could derive our `TDialogEditor` from the `TDefaultEditor` class instead of the `TComponentEditor` class. Both solutions will work. We will implement both of them later in this article.



➤ *Figure 2*

```
procedure TCommonDialogComponentEditor.Edit;
begin
  if (Component IS TOpenDialog) then { also TSaveDialog }
    (Component AS TOpenDialog).Execute
  else if (Component IS TPrintDialog) then
    (Component AS TPrintDialog).Execute
  else if (Component IS TPrinterSetupDialog) then
    (Component AS TPrinterSetupDialog).Execute
  else if (Component IS TColorDialog) then
    (Component AS TColorDialog).Execute
  else
    { default behaviour }
    inherited Edit
end {Edit};
```

➤ *Listing 3*

```
unit CompEdit;
{ TCommonDialogComponentEditor version 0.1 }
interface
uses DsgnIntf;
Type
  TCommonDialogComponentEditor = class(TComponentEditor)
  public
    procedure Edit; override;
  end;
  procedure Register;
implementation
uses Dialogs;
procedure TCommonDialogComponentEditor.Edit;
begin
  if (Component IS TOpenDialog) then { also TSaveDialog }
    (Component AS TOpenDialog).Execute
  else if (Component IS TPrintDialog) then
    (Component AS TPrintDialog).Execute
  else if (Component IS TPrinterSetupDialog) then
    (Component AS TPrinterSetupDialog).Execute
  else if (Component IS TColorDialog) then
    (Component AS TColorDialog).Execute
  else
    inherited Edit { default behaviour }
end {Edit};
procedure Register;
begin
  { register TCommonDialogComponentEditor for
  TCommonDialog and all its derived classes }
  RegisterComponentEditor(TCommonDialog, TDialogEditor)
end;
end.
```

➤ *Listing 4*

Menu Component Editors

Component editors can do much more than we've looked at so far. In fact, we can create our own pop-up menus with several different options and actions to choose from. Let's try to do the same "Execute/Preview" task for the common dialogs, but this time add an information message as a second choice.

To avoid breaking the default behaviour of the classes derived from TCommonDialog we will only register our TCommonDialogComponentEditor version 0.2 for the five classes that have no event properties.

We need to override two functions and one procedure this time: function GetVerbCount, function GetVerb and procedure ExecuteVerb (see Listing 5).

Since we only want two menu options, one for the Dialog.Execute action and one for the "About" message, the function GetVerbCount should return a value of 2 (see Listing 6).

Now, remember that Delphi usually counts from 0 up, including in this case. So, the next function, GetVerb, takes an index parameter which can receive the values 0 or 1 for the two possible values of the menu entries. For 0 we should return Execute, while for 1 (or any other value) we should return About, as in Listing 7.

In order to show which common dialog we want to execute, we could change GetVerb for index 0 to the following:

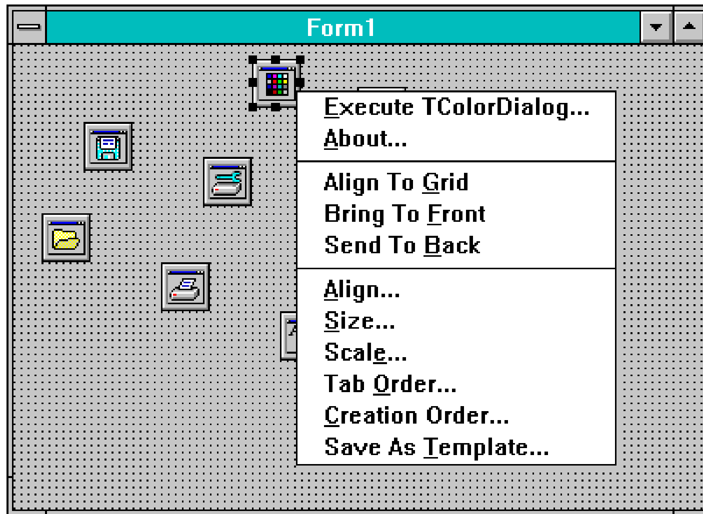
```
GetVerb := '&Execute ' +
  Component.ClassName + '...'
```

Finally, we can execute the verbs. For index 0 we do exactly the same thing we previously did in the TDialogEditor, namely, call the Execute method of the correct class, and for index 1 or higher we just show an 'About' MessageDlg. See Listing 8.

Notice that we can use the Component property to get to the actual component for which we are an editor. Also notice that we need to use the run-time type information of our associated component to get to its actual contents.

If we try this code, we get a nice pop-up component editor menu, as shown in Figure 3. If we execute the TColorDialog and select another colour, white for example, and close the dialog, we will notice that the Color property of the TColorDialog in the Object Inspector still has the old value. Why didn't it get updated to the new one we selected? Is this process one-way-only, or what? Actually, it seems that we needed to call the Designer.Modified method to

➤ Figure 3



➤ Listing 5

```
TCommonDialogComponentEditor = class(TComponentEditor)
public
  function GetVerbCount: Integer; override;
  function GetVerb(Index: Integer): string; override;
  procedure ExecuteVerb(Index: Integer); override;
end;
```

➤ Listing 6

```
function TCommonDialogComponentEditor.GetVerbCount: Integer;
begin
  GetVerbCount := 2
end {GetVerbCount};
```

➤ Listing 7

```
function TCommonDialogComponentEditor.GetVerb(Index: Integer): string;
begin
  if Index >= 1 then GetVerb := '&About...'
  else GetVerb := '&Execute...'
end {GetVerb};
```

➤ Listing 8

```
procedure TCommonDialogComponentEditor.ExecuteVerb(Index: Integer);
begin
  if index >= 1 then
    MessageDlg('TCommonDialogComponentEditor (c) 1996 by Dr.Bob',
      mtInformation, [mbOk], 0)
  else if (Component IS TOpenDialog) then { also TSaveDialog }
    (Component AS TOpenDialog).Execute
  else if (Component IS TPrintDialog) then
    (Component AS TPrintDialog).Execute
  else if (Component IS TPrinterSetupDialog) then
    (Component AS TPrinterSetupDialog).Execute
  else if (Component IS TColorDialog) then
    (Component AS TColorDialog).Execute;
end {ExecuteVerb};
```

inform the designer (and hence the Object Inspector) that the component has changed, ie that one or more of its properties has just got a new value. If we include this line (as last line of the `ExecuteVerb` method), then everything works fine as expected.

The complete source code for the new menu component editor `TCommonDialogComponentEditor` can be seen in Listing 9.

```
unit CompMenu;
{ TCommonDialogComponentEditor version 0.5 }
interface
uses DsgnIntf;
Type
  TCommonDialogComponentEditor = class(TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(index: Integer): String; override;
    procedure Executeverb(index: Integer); override;
  end;
  procedure Register;
implementation
uses Dialogs;
function TCommonDialogComponentEditor.GetVerbCount: Integer;
begin
  GetVerbCount := 2
end {GetVerbCount};
function TCommonDialogComponentEditor.GetVerb(index : Integer): String;
begin
  if Index >= 1 then
    GetVerb := '&About...'
  else
    GetVerb := '&Execute...'
end {GetVerb};
procedure TCommonDialogComponentEditor.ExecuteVerb(index: Integer);
begin
  if index >= 1 then
    MessageDlg('TCommonDialogComponentEditor (c) 1996 by Dr.Bob',
      mtInformation, [mbOk], 0)
  else if (Component IS TOpenDialog) then { also TSaveDialog }
    (Component AS TOpenDialog).Execute
  else if (Component IS TPrintDialog) then
    (Component AS TPrintDialog).Execute
  else if (Component IS TPrinterSetupDialog) then
    (Component AS TPrinterSetupDialog).Execute
  else if (Component IS TColorDialog) then
    (Component AS TColorDialog).Execute;
  Designer.Modified { inform the Object Inspector of the change }
end {Edit};
procedure Register;
begin
  RegisterComponentEditor(TCommonDialog, TCommonDialogComponentEditor)
end;
end.
```

➤ *Listing 9*

Menu Default Component Editors

Now that we've seen how we can write a new menu component editor for the `TCommonDialog` components that do not have a default behaviour, why not extend it to cover the ones that already do have a default behaviour, but without losing this new behaviour, ie offer a pop-up menu with as a first (default) choice the `OnEvent` handling code editor, as a second choice the dialog preview and as a third choice the About dialog.

For this we need another class `TCommonDialogDefaultEditor` which is derived from `TDefaultEditor`. Again, we need to override two functions and one procedure: function `GetVerbCount`, function `GetVerb` and procedure `ExecuteVerb`, see Listing 10.

```
TCommonDialogDefaultEditor = class(TDefaultEditor) { not TComponentEditor }
public
  function GetVerbCount: Integer; override;
  function GetVerb(Index: Integer): string; override;
  procedure ExecuteVerb(Index: Integer); override;
end;
```

➤ *Listing 10*

We now want three instead of two menu options: one for the default action (`TDefaultEditor.Edit`), one for the preview (calling `Dialog.Execute`) and a third for the About box, hence `GetVerbCount` should return a value of 3. The new `GetVerb` is shown in Listing 11.

```
function TCommonDialogComponentEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: GetVerb := '&OnEvent handler code';
    1: GetVerb := '&Execute ' + Component.ClassName + '...';
  else
    GetVerb := '&About...'
  end {case}
end {GetVerb};
```

➤ *Listing 11*

Finally, we execute the verbs, as is shown in Listing 12.

I think you'll get the idea by now, but just to be sure, the complete source code is on the disk of course. If we install and active this new `TCommonDialogDefaultEditor`, on a `TFindDialog` for example, then we get the context menu shown in Figure 4.

If we select the first option (or double-click on the component, which will by default execute the first menu verb), then we automatically jump to the code editor in the `OnEvent` handler code (the `OnFind` in this case). The second menu option will give the preview and the third will show the about message. Just as we've expected and quite handy, if I may say so myself *[Yes, I'm feeling in a generous mood, so you may! Editor]*.

Next Time

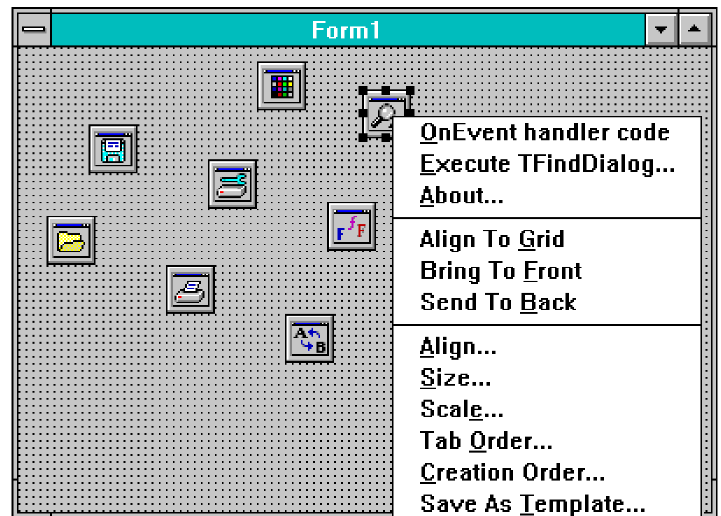
We've seen how to write our own component editors to extend the Delphi IDE itself. In fact, it almost looks like we've made Delphi a two-way interpreter!

Next Time, we'll explore the secrets of Delphi's own *Pandora's Box*: COMPLIB.DCL. In particular, we'll look at how we can register components automatically, how we can make sure we'll be able to re-build COMPLIB.DCL (even if we've corrupted it by accident), how to optimise COMPLIB.DCL and how we can share it amongst multiple developers.

Part one of *Delphi Development for Workgroups* starts next month; part two will be about Version Control Systems and the long awaited ViCiouS. You'd better be sure to be there...

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 two year old son Erik Mark Pascal.

➤ Figure 4




```
procedure TCommonDialogComponentEditor.ExecuteVerb(Index: Integer);
begin
  if index >= 2 then
    MessageDlg('TCommonDialogDefaultEditor (c) 1996 by Dr.Bob',
      mtInformation, [mbOk], 0)
  else if index = 1 then begin
    if (Component IS TFindDialog) then { also TReplaceDialog }
      (Component AS TFindDialog).Execute
    else if (Component IS TFontDialog) then
      (Component AS TFontDialog).Execute;
    Designer.Modified
  end else
    inherited Edit { TDefaultEditor.Edit for index = 0 }
end {ExecuteVerb};
```

➤ Listing 12