Focus Made Easy

by Hart Kerbel

A few years ago I found myself working on applications in which the users would often need to look to and from the monitor, often as they entered information. To help them quickly find where they are on the screen I decided to highlight the currently focused control, changing the color to yellow where appropriate.

This quickly became tedious and error prone to implement control-by-control: I quickly concluded this was an opportunity to leverage Delphi's strength and create a reusable control. The final result is CtlFocus, a non-visual control that makes it a snap for developers to add this functionality. This article describes the crafting of CtlFocus.

Analysis

What is required is a mechanism that highlights the focused control in such a way as to make it easy for the user to quickly locate it on the screen. When the control loses focus it must revert back to its non-focused state. This also holds true when the parent form loses focus.

The solution must be easy, reliable and reusable. Developers must also be able to override and extend the base functionality with a minimum of work.

Design

To make it as easy as possible for developers we'll package our project as a non-visual VCL control. The default behavior will be to alter the color when the control gains focus and revert back to the initial color when focus is lost. If the current control does not have a published `Color` property then the appearance is not altered.

To allow developers to extend the functionality of CtlFocus its key behaviors are encapsulated in virtual methods. Each method undertakes one well-defined task, making it easy to understand how to use our objects and to create descendant objects.

For example, the method `RestoreCtrlState` is responsible for restoring a control to its non-focused appearance: it's called whenever a change in focus is detected. It is conceivable that in the future developers will need to modify this behavior. Consider how awkward this would be if `RestoreCtrlState` performed other duties such as, for example, setting the appearance of the newly focused control (as performed by `SetColorProperty`). If it becomes difficult to override functionality developers will have to start duplicating and hacking code. This is both time consuming and error prone, it may very well be easier for them to abandon the control outright and find another solution.

As important as inheritance is, it is often overkill if all we need to do is slightly modify the behavior of our control. Let's say we have a `TEdit` control that represents sensitive information. Instead of using the default color of yellow we decide to use red. Creating a descendant control is not reasonable for this one special case, because it would have to check specifically for the name of the control we want to color red. Hardly a reusable approach.

So, how do we accommodate special cases? We provide events at just the right time and with just the right information. CtlFocus has two events, `OnAfterFocus` and `OnBeforeFocus`, both defined as:

```
TNotifyColorChangeEvent =
  procedure (Sender: TObject;
  const AComponent: TComponent;
  var ANewColor: TColor;
  var AChangeColor: Boolean)
  of object;
```

Table 1 describes the parameters. `OnBeforeFocus` is triggered just before a control receives focus. `OnAfterFocus` is triggered just as a control loses focus.

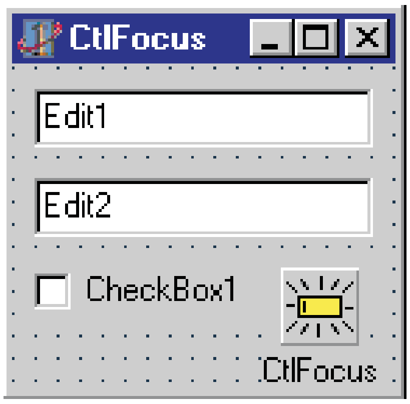
Figure 1 is a simple form, shown in design mode, with a CtlFocus and three visible controls: two edit boxes and a checkbox. Listing 1 is an example of how the `OnBeforeFocus` and `OnAfterFocus` events may be coded.

Figure 2 shows how the form appears when `Edit1` is focused. Since the default behavior is not modified the color is set to yellow. In Figure 3 `Edit2` is focused. Note how its color is red. That is because in the `OnBeforeFocus` event we explicitly check to see if `Edit2` is about to gain focus. If it is then we override the color:

Parameter Name	Type	Description
`Sender`	`TObject`	The object that generated the event. Not normally used.
`AComponent`	`TComponent`	A reference to the component that is about to have its appearance modified. The developer may use this to change attributes of the control other than the color.
`ANewColor`	`TColor`	The color to use for highlighting. On entry into the event handler this will be set to the value of the FocusedColor property. The color may overridden on a case-by-case basis by referring to the AComponent parameter.
`AChangeColor`	`Boolean`	Default value is true. Setting it to false instructs CtlFocus not to change the color.

➤ *Left: Table 1*

➤ *Figure 1*



```
unit fListing1;
interface
uses
  Windows, Messages, Graphics, Classes, Controls, Forms,
  CtlFocus, StdCtrls;
type
  TfrmListing1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
  CtlFocus: TCtlFocus;
    CheckBox1: TCheckBox;
    procedure CtlFocusBeforeFocus(Sender: TObject;
      const AComponent: TComponent; var ANewColor: TColor;
      var AChangeColor: Boolean);
  procedure CtlFocusAfterFocus(Sender: TObject;
    const AComponent: TComponent; var ANewColor: TColor;
    var AChangeColor: Boolean);
  private
  public
    end;
var frmListing1: TfrmListing1;
implementation
{$R *.DFM}
procedure TfrmListing1.CtlFocusBeforeFocus(Sender: TObject;
  const AComponent: TComponent; var ANewColor: TColor;
  var AChangeColor: Boolean);
begin
  if AComponent = Edit2 then
    ANewColor := clRed
  else if AComponent is TCheckBox then begin
    AChangeColor := False;
    TCheckBox(AComponent).Font.Style :=
      TCheckBox(AComponent).Font.Style + [fsbold] +
      [fsUnderline];
  end;
end;
procedure TfrmListing1.CtlFocusAfterFocus(Sender: TObject;
  const AComponent: TComponent; var ANewColor: TColor;
  var AChangeColor: Boolean);
begin
  if AComponent is TCheckBox then
    TCheckBox(AComponent).Font.Style :=
      TCheckBox(AComponent).Font.Style - [fsbold] -
      [fsUnderline];
end;
end.
```

➤ *Listing 1*

```
...
else if AComponent is TCheckBox then begin
  AChangeColor := False;
  TCheckBox(AComponent).Font.Style :=
    TCheckBox(AComponent).Font.Style + [fsbold] + [fsUnderline];
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
if AComponent is TCheckBox then begin
  AChangeColor := False;
  TCheckBox(AComponent).Font.Style :=
    TCheckBox(AComponent).Font.Style - [fsbold] - [fsUnderline];
end;
```

```
 if AComponent = Edit2 then
   ANewColor := clRed
```

The coding behind Figure 4 is slightly more complicated: see Listing 2. For all instances of `TCheckBox` and its descendants we're not changing the color but setting the text to bold underlined. Notice that we defeat CtlFocus' default behavior by setting `AChangeColor` to `False`. We undo the modifications to the font in the `OnAfterFocus` event, see Listing 3.

Listing 1 displays the complete source for the form. We have designed a component that will be flexible and easy to extend. Now we need to implement it.

Implementation

Because CtlFocus is non-visual it descends from `TComponent`. This gives us just what we want. `TComponents` show up in the component pallet, may be manipulated at design-time and are automatically saved with the parent form.

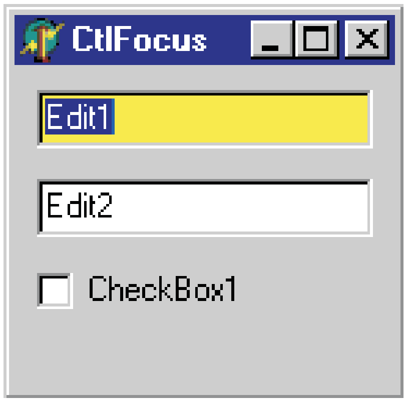
The tricky part in implementing CtlFocus is determining when a control is about to lose or gain focus. We need to find some events or messages that we can tap into.

As is often the case there are various choices. One of them is to hook into the `Screen` object's `OnActiveControlChange` and `OnActiveFormChange` events. Remember that each instance of our control will need to hook into these events, in essence creating a chain of event handlers. We must be sure to call the event handler that was assigned before we hooked in. This means every event handler in the chain will be called even when the host form is not active. The bottom line is that we must add a check to make sure that the control in question and the active event handler belong to the same form. Hooking into the `OnActiveControlChange` event looks like:

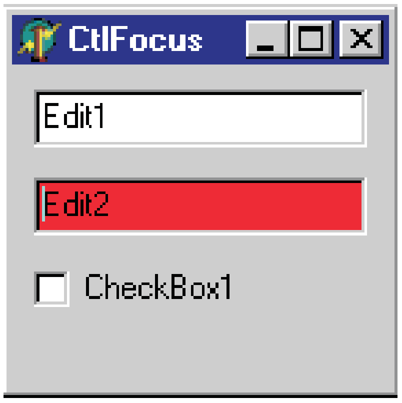
```
FSuperOnControlChangeEvent :=
  Screen.OnActiveControlChange;
Screen.OnActiveControlChange :=
  ActiveControlChange;
```

`FSuperOnControlChangeEvent` holds a reference to the event handler before we hooked into it. `ActiveControlChange` is our new event handler as shown in Listing 4.

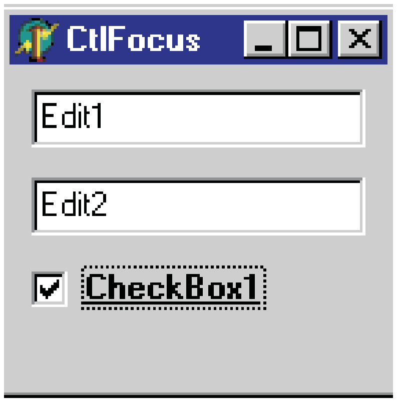
➤ *Figure 2*



➤ *Figure 3*



➤ *Figure 4*



Notice how we make sure that the event is fired due to a control on the form to which the event handler belongs. Also note how we call the original event handler.

Finally we must preserve the chain of handlers by re-hooking the original when our control is destroyed:

```
Screen.OnActiveControlChange :=
  FSuperOnControlChangeEvent;
```

The above method works, but since many event handlers may be invoked each time focus is shifted and we are hooking into public event handlers I am uneasy. Let's check out an alternative approach. The following is a more conventional method, but perhaps the techniques are not as well known.

The VCL sends internal messages, known as component messages. They are defined in Controls.pas and are identified with the `CM_` prefix. They are generated as the result of windows messages (`WM_`) and user activities. Most of the time we need not be concerned with these messages, because Delphi handles the details for us. `CM_` messages are barely discussed in the Delphi documentation, but I learned about them from the excellent book *Secrets of Delphi 2* and from the VCL source code.

It just so happens that there is a `CM_FOCUSCHANGED` message, which is fired whenever focus is changed from control to control.

`TCustomForm`, the base class of `TForm`, knows when focus has changed. It sends a `CM_FOCUSCHANGED` message to itself that in turn directs it to send a `CM_FOCUSCHANGED` to all its child controls. The message parameter is of type `TCMFocusChanged`. It has a `Sender` attribute of type `TWinControl` that references the currently focused control.

The pattern of a container control sending a message to itself and then to all of its children is common in the VCL. It is an elegant way for the container to inform all of its children that something has changed. Take a look at the *Default Button* sidebar for an example.

It is easy to whip up a quick proof-of-concept: add the declaration shown in Listing 5 to your form class. The name of the method is not important. I have applied the same convention used in the VCL source and by most developers. The code snippet in Listing 6 updates the form's caption with the control name.

Be sure to include the `inherited` statement. This ensures that the `CM_FOCUSCHANGED` message reaches all of its handlers.

There is a problem with the `CM_FOCUSCHANGED` message that we must overcome. The form only sends the `CM_FOCUSCHANGED` event to `TWinControl` descendants, thus shutting out our `TComponent` derived control. We could cheat and descend from `TWinControl`, but that's an ugly hack.

As you might expect, there is a clean solution. Windows developers who learned to program directly to the Win16 and Win32 APIs know the answer: subclass the host form. See the *Windows Subclassing 101* sidebar for a brief overview of the technique.

The first thing we do when subclassing a window is to save the current `WndProc`'s address. The second step is to substitute the new `WndProc`, in our case it is named `CtlFocusWndProc`. This takes place in the `Create` method, as shown in Listing 8. You should note that the form is only subclassed at runtime. Originally while I was developing CtlFocus control I forgot to do this, talk about weird behavior in the Delphi IDE! Let's take a look at a slightly simplified version of `CtlFocusWndProc`, see Listing 7. The complete version is in Listing 8.

Remember that every message that is sent to the host form will be diverted through `CtlFocusWndProc` so we must be as efficient as possible. As shown in Listing 7, we test

```
procedure TfrmScreen1.ActiveControlChange(Sender: TObject);
begin
  if Screen.ActiveCustomForm = self then begin
    // change focused appearance here
  end;
  if Assigned(FSuperOnControlChangeEvent) then
    FSuperOnControlChangeEvent(Sender);
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
TFocusForm = class(TForm)
private
  procedure CMFocusChanged(var Message: TCMFocusChanged);
    message CM_FOCUSCHANGED;
```

```
procedure TFocusForm.CMFocusChanged(var Message : TCMFocusChanged);
begin
  Caption := Message.Sender.Name;  // Sender is a reference to current control
  Inherited;
end;
```

➤ *Listing 6*

Default Button

Ever wonder how TButtons coordinate which one is the default? They do not communicate directly with each other, but rather leverage the `CM_FOCUSCHANGED` message. This sidebar deals with runtime behavior. The Delphi form designer acts differently. The default button is the one which responds to the `Enter` key. It is visually denoted with a black border. If a button has focus it is automatically the default. If a non-TButton control has focus then the first button with its Default property set `True` becomes the default. The implementation of `TButton` has a message handler for `CM_FOCUSCHANGED`. It checks first to see if the `Sender` attribute is a `TButton` or TButton descendant. If it is then it checks to see if `Sender` is a reference to itself. If so it is the default button otherwise it is not. If `Sender` is not a `TButton` then its appearance is determined by the `Default` property. All of this logic is accomplished in only seven lines of code. Take a look at the StdCtrls.pas file in the \Source\VCL directory to see the implementation. You may wonder why the buttons do not coordinate this amongst themselves. It is poor OO design to have objects in a container send messages directly to each other. Let the container coordinate this activity.

to see if the message is CM_FOCUSCHANGED. If it is we call FocusChanged (described a little later).

Finally, regardless of the message type, we call the original WndProc routine that we previously stored a reference to in FHostFormWndProc. This makes sense because we tap into the CM_FOCUSCHANGED event to highlight our control, but we'd better let the VCL complete the rest of the work.

CtlFocusWndProc is not quite complete. We need to consider the situations where a form loses and gains focus. CM_FOCUSCHANGED is only sent when focus is changed within a form. CM_ACTIVATE and CM_DEACTIVATE, on the other hand, are sent when the form gains/loses focus. These are the internal messages that fire the OnActivate and OnDeactivate events.

The FocusChanged method is the central starting point for modifying the controls appearance. If Enabled is True it simply calls RestoreCtrlState then AlterCtrlState. Note that I didn't make this method virtual. It does not perform any key functionality. On the other hand, AlterCtrlState and RestoreCtrlState are declared virtual.

AlterCtrlState is responsible for changing the appearance of a newly focused control. Before it does anything it triggers the OnBeforeFocus event. As discussed above, this gives the opportunity to customize CtlFocus's behavior:

```
clFocusedColor :=
  FFocusedColor;
bChangeColor := True;
DoBeforeFocus(bChangeColor,
  clFocusedColor);
```

DoBeforeFocus wraps the call to OnBeforeFocus. It is best to isolate event trigger functions to one location. bChangeColor and clFocusedColor may ultimately be modified in the developer's event handler as show in Listing 1. The next step is to change the color if appropriate:

```
if bChangeColor then
  SetColorProperty(
    FHostForm.ActiveControl,
    FLastColor,
    clFocusedColor);
```

The CM_FOCUSCHANGED message is sent *after* the focus has been changed, so we need to keep track of which control just received focus:

```
FLastFocusedCtrl :=
  FHostForm.ActiveControl;
```

RestoreCtrlState is responsible for setting the control's appearance back to its default. It is similar to AlterCtrlState, but instead of acting upon the active control it refers to the control reference saved in FLastFocusedCtrl.

The last method worth noting is SetColorProperty, which uses RTTI to change the color property of the component. RTTI is a large and complicated subject: I will briefly describe how CtlFocus uses it. Remember RTTI only pertains to published class members.

The RTTI definitions and implementation can be found in the TypInfo.pas unit. Refer to this unit and the references at the end of the article for more information. Unfortunately, the Delphi documentation is lacking on this subject.

The first step is to make sure that the component has a published Color property. The GetPropInfo procedure does this for us:

```
PropInfo := GetPropInfo(
  AComponent.ClassInfo,
  'Color', [tkInteger]);
```

The first parameter is a pointer to the RTTI data. The second parameter is the name of the property. Case is not important but of course spelling is. Special considerations are required here if this control is released in different languages. The final parameter, which is optional, lists the type(s) of the parameters that we are searching for. In our case we are looking for TColor, actually just a subrange of a long integer. If the result is nil then the property does not exist, or is not an integer type, and SetColorProperty does nothing.

The second step is to remember the current color so when the control loses focus in the future we can restore it. For this we call GetOrdProp:

```
procedure TCtlFocus.CtlFocusWndProc(
  var Message: TMessage);
begin
  if Message.Msg =
    CM_FOCUSCHANGED then
    FocusChanged;
  FHostFormWndProc(Message);
end;
```

➤ *Listing 7*

```
ACurrentColor := TColor(
  GetOrdProp(AComponent,
  PropInfo));
```

Notice that the second parameter is the result of the call to GetPropInfo (GetOrdProp returns a LongInt, typecast to a TColor).

The third and final step is to set the color of the control by calling SetOrdProp:

```
SetOrdProp(AComponent,
  PropInfo, LongInt(
  AFocusedColor));
```

The last parameter contains the new value for the color, typecast back to a long integer.

Final Note

I must admit there is a change to the design that one of these days I will redo. Perhaps some of you proficient object oriented designers picked up on it: the tight integration of changing the color can be improved upon. In my defense, when I was first designing CtlFocus I wanted to get it done swiftly. The ability to change the color plus the flexibility of overriding on a case-by-case basis was satisfactory. What I describe below is more elaborate and would have taken considerably longer to implement.

Suppose, for example, that the users decided that rather than change the color they would rather have a border displayed around the controls? We can do it with CtlFocus as it is designed today but it requires overriding the default behavior in event handlers or by creating a descendant control that overrides AlterCtrlState and RestoreCtrlState.

What I intend to do is factor out the implementation parts that are specific to changing the color and replace it with an abstract class, named TCtlFocusDisplay, that

```
unit CtlFocus;
interface
uses
  Messages, Classes, Graphics, Controls, Forms;
const
  DEFAULT_FOCUSED_COLOR = clYellow;
type
  TNotifyColorChangeEvent = procedure (Sender: TObject;
    const AComponent : TComponent; var ANewColor : TColor;
    var AChangeColor : Boolean ) of object;
type
  TCtlFocus = class(TComponent)
  private
    FHostForm : TCustomForm;  // form containing control.
    // Last control to be focused
    FLastFocusedCtrl : TWinControl;
    FHostFormWndProc : TWndMethod; // Host form's WndProc.
    FEnabled: Boolean;
    FFocusedColor: TColor;
    FLastColor : TColor;  // Remember the original color.
    FOnAfterFocus: TNotifyColorChangeEvent;
    FOnBeforeFocus: TNotifyColorChangeEvent;
  protected
    procedure FocusChanged;
    procedure AlterCtrlState; virtual;
    procedure RestoreCtrlState; virtual;
    procedure SetEnabled(const Value: Boolean); virtual;
    procedure SetFocusedColor(const Value: TColor); virtual;
    procedure CtlFocusWndProc(var Message: TMessage);
      virtual;
    procedure DoBeforeFocus(var AChangeColor: Boolean;
      var AFocusedColor: TColor); virtual;
    procedure DoAfterFocus(var AChangeColor: Boolean;
      var AFocusedColor: TColor;
      const AComponent ); virtual;
    function SetColorProperty(AComponent: TComponent; var
      ACurrentColor: TColor; const AFocusedColor: TColor) :
      Boolean; virtual;
    function RunTime: Boolean; // Returns true if runtime
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property Enabled : Boolean
      read FEnabled write SetEnabled default True;
    property FocusedColor : TColor read  FFocusedColor
      write SetFocusedColor default DEFAULT_FOCUSED_COLOR;
    property OnBeforeFocus : TNotifyColorChangeEvent
      read FOnBeforeFocus write FOnBeforeFocus;
    property OnAfterFocus : TNotifyColorChangeEvent
      read FOnAfterFocus write FOnAfterFocus;
  end; { TCtlFocus }
  procedure Register;
implementation
uses TypInfo;
destructor TCtlFocus.Destroy;
begin
  { Restore original WndProc. Technically only required if
    CtlFocus is dynamically created and destroyed, but a
    good practice to always follow. }
  if RunTime then
    FHostForm.WindowProc := FHostFormWndProc;
  inherited Destroy;
end; { Destroy }
constructor TCtlFocus.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FHostForm := TCustomForm(AOwner);
  FLastFocusedCtrl := nil;
  FFocusedColor := DEFAULT_FOCUSED_COLOR;
  Enabled := True;
  {- Subclass the host form if it is run time. -}
  if RunTime then begin
    FHostFormWndProc := FHostForm.WindowProc;
    FHostForm.WindowProc := CtlFocusWndProc;
  end;
end; { Create }
procedure TCtlFocus.SetEnabled(const Value: Boolean);
begin
  FEnabled := Value;
  if FEnabled then
    FocusChanged
  else
    RestoreCtrlState;
end; { SetEnabled }
procedure TCtlFocus.SetFocusedColor(const Value: TColor);
begin
  FFocusedColor := Value;
  if Enabled then
    FocusChanged;
end; { SetFocusedColor }
procedure TCtlFocus.CtlFocusWndProc(var Message: TMessage);
begin
  case Message.Msg of
    CM_FOCUSCHANGED:  // Focus has shifted within form.
      FocusChanged;
    CM_DEACTIVATE:  // Host form is about to loose focus.
      RestoreCtrlState;
    CM_ACTIVATE:  // Host form is about to (re)gain focus.
      FocusChanged;
  end; {case}
  { Pass all messages on to original WndProc. }
  FHostFormWndProc(Message);
end; { CtlFocusWndProc }
procedure TCtlFocus.FocusChanged;
begin
  if not Enabled then Exit;
  RestoreCtrlState;
  AlterCtrlState;
end; { FocusChanged }
{ Trigger OnBeforeFocus event then change color property }
procedure TCtlFocus.AlterCtrlState;
var
  bChangeColor : Boolean;
  clFocusedColor : TColor;
begin
  clFocusedColor := FFocusedColor; // Set default color.
  // Default action is to change the color.
  bChangeColor := True;
  // Opportunity to override default settings.
  DoBeforeFocus(bChangeColor, clFocusedColor);
  if bChangeColor then
    SetColorProperty(FHostForm.ActiveControl, FLastColor,
      clFocusedColor);
  FLastFocusedCtrl := FHostForm.ActiveControl;
end; { AlterCtrlState }
{ Trigger the OnAfterFocus event then restore the color. }
procedure TCtlFocus.RestoreCtrlState;
var
  bChangeColor : Boolean;
  sink : TColor;
begin
  if FLastFocusedCtrl <> nil then begin
    // The default action is to change the color.
    bChangeColor := True;
    DoAfterFocus(bChangeColor,FLastColor,FLastFocusedCtrl);
    if bChangeColor then
      SetColorProperty(FLastFocusedCtrl, sink, FLastColor);
  end;
end; { RestoreCtrlState }
{Trigger OnAfterFocus event just before control loses focus}
procedure TCtlFocus.DoAfterFocus(var AChangeColor: Boolean;
  var AFocusedColor: TColor; const AComponent: TComponent);
begin
  if Assigned(FOnAfterFocus) then
    FOnAfterFocus(Self, AComponent, AFocusedColor,
    AChangeColor);
end; { DoAfterFocus }
{Trigger OnBeforeFocus event just before control
 receives focus }
procedure TCtlFocus.DoBeforeFocus(var AChangeColor: Boolean;
  var AFocusedColor : TColor);
begin
  if not Assigned(FHostForm.ActiveControl) then
    Exit;  // No active control.
  if Assigned(FOnBeforeFocus) then
    FOnBeforeFocus(Self, FHostForm.ActiveControl,
      AFocusedColor, AChangeColor);
end; { DoBeforeFocus }
{- Set the color property of AComponent using RTTI. -}
function TCtlFocus.SetColorProperty(AComponent: TComponent;
  var ACurrentColor: TColor; const AFocusedColor: TColor):
  Boolean;
var
  PropInfo : PPropInfo;
begin
  if not Assigned(AComponent) then begin
    Result := False;
    Exit;
  end;
  PropInfo := GetPropInfo(AComponent.ClassInfo, 'Color',
    [tkInteger]);
  if PropInfo = nil then begin
    Result := False;
    Exit;
  end;
  Result := True;
  ACurrentColor := TColor(GetOrdProp(AComponent, PropInfo));
  SetOrdProp(AComponent, PropInfo, LongInt(AFocusedColor));
end; { SetColorProperty }
function TCtlFocus.RunTime: Boolean;
begin
  RunTime := not (csDesigning in ComponentState);
end; { RunTime }
procedure Register;
begin
  RegisterComponents('DDJ', [TCtlFocus]);
end; { Register }
end.
```

➤ *Listing 8*

encapsulates the generic functionality of modifying and restoring the focused control.

The beauty of this is that any number of schemes may be implemented without making any changes to the core control. One such implementation is, for example, to change the color, another is to draw a border. Organizations could perhaps create their very own `TCtlFocusDisplay` descendants for complete flexibility.

The truly savvy object orient designers will recognize this as the Strategy Pattern.

Windows Subclassing 101

Subclassing is a technique that allows the default behavior of a window to be modified. Conceptually it is vaguely similar to creating a descendant class from a base class via inheritance. The implementation is based on procedural programming techniques. Every window has a Window Procedure, know as the `WndProc`, associated with it. All messages sent to a window go through this procedure. By substituting our own `WndProc` for the current one we can override the window's behavior. The `SetWindowLong` API procedure is the traditional API call used to accomplish this.

In the old days of coding in C the `WndProc` routine tended to be one large switch statement. Each case was focused on a specific message. Happily Delphi nicely encapsulates this in the VCL. We seldom find ourselves explicitly subclassing or coding `WndProc` routines anymore. However, the VCL is architected so that if we need to get our hands dirty it's a snap. `TControl` defines the public property `WindowProc` of type `TWndMethod`. Assigning this property a reference to another `WndProc` (the actual name is not relevant) is all there is to it.

One final note: be sure to call the original `WndProc` routine with messages that are not handled. Forgetting to do so will quickly cause weird and unpredictable results.

Summary

With some planning and a little knowledge of Delphi and the Windows environment, implementing components such as CtlFocus is not really that difficult. The CtlFocus unit is well under 300 lines of code.

My experiences have taught me that considering the analysis and design before implementation is a big time saver. Even though this is a small project, taking the time to reflect how to create an extensible and flexible component *before* implementing it helped me to achieve my goal faster.

I hope you enjoy using CtlFocus as much as I did creating it.

References

The following are great books. All of them still have great information, even for Delphi 5:

Secrets of Delphi 2, Ray Lischner, Waite Group Press, 1996.

Delphi Component Design, Danny Thorpe, Addison Wesley Developers Press, 1997.

Developing Custom Delphi 3 Components, Ray Konopka, Coriolis Group Books, 1997.

Hart Kerbel is a contract software architect/developer based in Toronto, Canada. He can be reached by email at hart.kerbel@sympatico.ca