

# What's Your Orientation? A View of Objects in Delphi

by Rob Bracken

Several of the speakers at September's POSK meeting stressed the usefulness of objects when you write large systems. Delphi's IDE is a superb tool, but (like Visual Basic) it doesn't actually encourage you to write object-oriented programs. I thought, therefore, that it would be a good idea to look at objects and to compare them with a non-object-oriented approach. I've tried to keep it simple, so I haven't explained how the code works, but I hope it'll give you an idea of how useful objects can be.

## Delphi and objects

The IDE makes extensive use of objects. Every component on the palette is an object, and when you drop one on a form, you are creating an instance of that object (yes, even at design time!). You then write code to respond to events and manipulate the objects you've created.

"But," I hear you cry, "isn't this object-oriented programming?" The answer to this is – strictly speaking – no. It's been called "object-based", but it's not object-oriented. You're using object types that were created by someone else, not objects you have written yourself.

## An example

Let's look at a simple example. Suppose you have several controls on a form which you want to enable and disable as a group. Create a form and drop a few controls on it. Drop a check box on it and put some code in the OnClick event handler that sets the Enabled property of each control to the same value as the Checked property of the check box. For example:

```
Button1.Enabled := CheckBox1.Checked;
```

Save and run the program. Now, when you check and uncheck the check box, the controls are enabled and disabled. This does what it's supposed to do, but now drop another control on the form. Unless you add a line of code to enable and disable it, it will be unaffected by the state of the check box.

Stop the program and cut the controls (except the controlling check box) to the clipboard. Drop a panel on the form and paste the controls into it. Comment out the code in the check box's OnClick event and replace it with

```
Panel1.Enabled := CheckBox1.Checked;
```

Save and run the program again. The controls are still enabled and disabled by clicking the check box, but this time you only had to write one line of code. Additionally, if you add controls to the panel, they are automatically enabled and disabled when you click the check box, without you having to remember to write any more code. This is because the TPanel object enables and disables all its controls, without you having to do anything, however many controls it has.

This shows a basic difference between object-oriented and non object-oriented programming. The non object-oriented approach is to write code that manipulates something, which may be a record, an object, an array or any other item. The object-oriented approach is to write an object that knows how to do the manipulation and then write code that tells the object to do it. I find it easier to think about where the code lives within the program. For a non object-oriented approach, the code to do something lives in an area such as a form, probably with a lot of other, unconnected code. For an object-oriented approach, the code to do something lives inside an object.

## The family silver

One problem with the TPanel is that it doesn't "grey out" its controls when it disables them. This is a problem with the way that TPanel disables the controls. We can, however, create a new type of panel that does "grey out" its controls. To do this, we can use another feature of objects – inheritance.

Suppose for a moment that Delphi doesn't support object inheritance. To create a new type of TPanel we would have to copy all the code from the existing TPanel and modify it. Suppose now that we find a bug in the original TPanel. We have two places where we have to fix it, making maintenance more difficult. Fortunately, Delphi does support inheritance, so we can create a new panel type by descending from the existing TPanel. (Actually, Delphi's designers have provided a TCustomPanel type which should be used as the parent for new types of panel, but I'm going to use TPanel to make life easier.)

Create a new package (File/New and select Package). Save it somewhere. Click Add on the Package Editor and click the New Component tab. Select TPanel as the Ancestor and set the Class Name to TGreyOutPanel. Select the Additional palette page, select a path for the unit and click OK. Declare the following procedure inside the Private declarations section of the TGreyOutPanel in the Interface section:

```
procedure CMEnabledChanged(  
    var Msg: TMessage);  
message cm_EnabledChanged;
```

Put the following code into the unit's Implementation section:

```
procedure  
    TGreyOutPanel.CMEnabledChanged(  
        var Msg: TMessage);  
var  
    I: Integer;  
begin  
    inherited;  
    Repaint;  
    for I := 0 to ControlCount - 1 do  
        Controls[ I ].Enabled := Enabled;  
end;
```

(This is how the Raize Components panel does it.) Click the Install button on the Package Editor to install the new component. You'll see a new component on the Additional palette page in the IDE. If you hover the cursor over it, you'll see GreyOutPanel in the hint window.

Go back to your form and cut all the controls out of the TPanel to the clipboard. Delete the TPanel and replace it with a new, TGreyOutPanel from the Additional palette page. Give it the name Panel1. Paste the controls back into the GreyOutPanel and run the program. Now, when you use the check box to disable the panel, the controls are "greyed out". We accomplished this by writing 10 lines of code. What we've done is use the effort that went in to the development of a TPanel and built on it to produce something that fits what we need. We also now have a reusable object that can save us time when we create more forms.

## An easy life

Suppose now that you're halfway through your project and the person who's going to use it wants a visual indication of which group of controls has the focus. Oh, and by the way, they want all the forms to behave the same way. You decide that you'll make the panel's background yellow when one of its controls has the focus. You'd be tempted to go in to all the forms and add OnEnter/OnExit event handlers to all the controls on all the panels. This would be the non object-oriented way to do it. It's quicker and more reliable, however, to change the behaviour of the panel itself. Add the following procedure to your new TGreyOutPanel type:

```
// New procedure declaration
procedure CMFocusChanged(
    var Message: TCMFocusChanged);
    message CM_FOCUSCHANGED;

{Helper function for CMFocusChanged.
Check to see if the control or any of
its children are focused}

function GroupFocused(
    Sender: TWinControl): boolean;
var
    I: integer;
begin
    {Check to see if the control is
    focused}
    Result := Sender.Focused;

    {Check to see if one of its children
    (or sub-children) is focused}
    I := 0;
    while (not Result) and
        (I < Sender.ControlCount) do
    begin
        if Sender.Controls[I] is
            TWinControl then
            Result := GroupFocused(
                TWinControl(Sender.Controls[I]));
        Inc(I);
    end;
end;

{Take action when the focus changes}
procedure TGreyOutPanel.CMFocusChanged(
    var Message:
        TCMFocusChanged);
```

```
begin
    inherited;
    {Set the background colour, depending
    on whether the panel or one of its
    children is focused. Setting the
    colour repaints the panel.}
    if GroupFocused(Self) then
        Self.Color := clYellow
    else
        Self.Color := clBtnFace;
end;
```

Recompile the package and install it. Recompile and run your program – the panel will change to yellow when one of its children has the focus. This is a major improvement over the non object-oriented approach. We added 22 lines of code to the new panel object and affected a (potentially) large number of components. Moreover, if we add or remove controls, we get the same behaviour with no further code changes, making maintenance much easier.

## Talking back

You often need to know when something happens to an object. You may, for instance, have an object in a separate thread, monitoring a serial port. In this case, you'd probably want to know when some data arrives so you can deal with it. You could put your program in a loop and poll the input data queue, but you wouldn't be able to do much else. It would also be difficult to keep track of more than one serial port. A better way is for the monitor object to tell you when it receives something. You could then break off, process it and go back to what you were doing before.

Delphi's way of handling this is with events. Objects provide places to plug in event-handling procedures. The objects then call the plugged-in procedures whenever a particular event happens. Let's add an event to the TGreyOutPanel. Supposing you want to display the name of the current group of controls in a status bar at the bottom of the form. When the user moves from one group of controls to another, you want the name to change in the status bar.

You can do this by adding an OnClick event handler to all the controls in all the groups. Each OnClick handler would display the group's name in the status bar. This is a daunting task, however, so let's see if we can change the TGreyOutPanel instead.

We're already taking action when one of the controls on the panel gets the focus. This would be a good place to add an event. I actually added two events – OnGroupFocused and OnGroupUnFocused. One is fired when a control on the panel gets the focus, the other when the focus moves off the panel. Add the following code to the Private declarations of TGreyOutPanel:

```
FOnGroupFocused: TNotifyEvent;
FOnGroupUnFocused: TNotifyEvent;
```

Add the following to the Published declarations:

```
property OnGroupFocused: TNotifyEvent
    read FOnGroupFocused
    write FOnGroupFocused;
property OnGroupUnFocused: TNotifyEvent
    read FOnGroupUnFocused
    write FOnGroupUnFocused;
```

Finally, change the CMFocusChanged procedure to be like this:

```
{Take action when the focus changes}
procedure TGreyOutPanel.CMFocusChanged(
    var Message: TCMFocusChanged);
begin
    inherited;
    {Set the background colour, depending
    on whether the panel or one of its
    children is focused. Setting the
    colour repaints the panel.}
    if GroupFocused(Self) then begin
        Self.Color := clYellow;
        if Assigned(FOnGroupFocused) then
            FOnGroupFocused(Self);
        end
    else begin
        Self.Color := clBtnFace;
        if Assigned(FOnGroupUnFocused) then
            FOnGroupUnFocused(Self);
        end;
    end;
end;
```

Compile and Install the package. Go back to your form and display the Object Inspector for a GreyOutPanel. Look at the Events page – there are now two extra events. You can add event handlers in exactly the same way as you do for e.g. an OnClick event. Add a TStatusBar to your form and add one panel. In the OnGroupFocused event handler, set its text to show that the panel has the focus. In the OnGroupUnFocused handler, set the text to show that it has lost the focus. Here are sample handlers:

```
procedure
    TForm1.GreyOutPanel1GroupFocused(
        Sender: TObject);
begin
    StatusBar1.Panels[0].Text :=
        'GreyOutPanel has focus';
end;

procedure
    TForm1.GreyOutPanel1GroupUnFocused(
        Sender: TObject);
begin
    StatusBar1.Panels[0].Text :=
        'GreyOutPanel has lost focus';
end;
```

Compile and run your form. When you click on a control, the status bar will show when the group inside the panel gets and loses the focus. Again, this is about where you place the code that gives you feedback. You can let the object tell you when something happens, or you can expend a lot of effort keeping track of the object and working out for yourself when an event occurs. If you decide to keep track of the object yourself and then use it somewhere else, you will have to duplicate the tracking code in the new place.

## Objections?

You don't unlock the full power of Delphi until you start using objects. I hope this article has fired your interest given you a glimpse of how they can help you develop your next project. Object strongly!

## More Information

- “An Introduction to Object-Oriented Programming” by Timothy Budd. (Addison Wesley Longman Inc. ISBN 0-201-82419-1)
- “Design Patterns – Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (often referred to as the “Gang of Four”). (Addison Wesley Longman Inc. ISBN 0-201-63361-2)
- The Delphi VCL (use the source!). It is a well-designed example of an object-oriented system and makes extensive use of design patterns (a form is an example of a Mediator pattern).



*Rob Bracken is Managing Director of Bracken Software Limited. He uses Delphi to build software systems and is always looking for new challenges. You can contact Rob on [robb@brackensoftware.co.uk](mailto:robb@brackensoftware.co.uk).*