Interposer Classes

Custom components made easy

by Stephen Posey

The designers of Delphi's VCL made some fairly clever and sophisticated choices with regard to its component hierarchy. One evidence of this is that most of the components in the Component Palette descend from 'Custom' ancestors (eg `TButton` descends from `TCustomButton`), which are full implementations of the component but expose no properties. This architecture allows a component designer to customize the behavior of the components they create by publishing only those properties that are appropriate for their particular purposes.

Even so, there are some cases where it's not clear what the VCL designers were thinking when they chose which properties to make public or published in some components. One common complaint is that the `TPanel` class, though descended from `TWinControl`, does not expose the `Canvas` property.

The usual solution is to create a new component that reveals the property, which can then be added to the Component Palette. This is, of course, a functional and long-term solution, especially when there may be other features to be added to the component.

Sometimes, however, for one-shot use of a property, creating a whole new component is overkill. It is also possible to trick Delphi into allowing you to access that property *without* having to create and install a whole new component. A common technique for doing this is typecasting, which often involves creating an empty 'dummy' class in the form unit:

```
implementation
type
  TDummyPanel = class(TPanel)
  end;
```

Which is then explicitly typecast when needed:

```
TDummyPanel(Panel1).Canvas
```

This is also a useful technique for an entirely different purpose. The Delphi visibility directives operate such that classes declared in the same unit have access to one another's protected methods and fields (supposedly equivalent to 'friend' classes in C++). This declaration of `TDummyPanel` effectively re-declares `TPanel` as local to the current unit, thus other classes in the unit have access to `TDummyPanel`'s (and thence `TPanel`'s) protected members.

The technique I'm writing about in this article, which I call creating an 'interposer class', is closely related to the above. Listing 1 shows the basic approach. What this does is to redefine the *meaning* of `TPanel` at compile and runtime, by using a bit of identifier scoping sleight-of-hand. While this technique provides the same access to protected members as the typecasting technique, it also has some other desirable characteristics. Overall, I find this a much more elegant approach.

As far as the IDE is concerned, we're still working with the registered `TPanel` component, and can manipulate any `TPanels` (and their standard properties) that we've placed on the form in the normal visual fashion.

At the same time, we also now have access to the inherited (but formerly hidden) `Canvas` property that is exposed by declaring it in the `public` section of the interposer component. Access is gained simply by referring to the components by name (or otherwise), just as we've always done.

This technique will also allow *adding* entirely new methods, data fields and even properties to the interposer class, that are accessible at compile-time and runtime. A trivial example is provided by the `HiThere` method in Listing 1 (several more elaborate examples of 'interposed' components are included on the disk). In fact, it's entirely *syntactically* legal to add items to the `published` section of an interposer class. Such additions are available only at compile-time and runtime. However, they do not show up at design-time (ie in the Object Inspector). Such published members are essentially treated as if they were declared in the `public` section.

This brings up what is maybe the main weakness of the technique: while it provides tremendous power to redefine the meaning of a component (in this case `TPanel`), it nonetheless offers no means to create new design-time properties.

➤ *Listing 1*

```
unit ipunit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TPanel = class( ExtCtrls.TPanel ) // Interposer TPanel class
  public
    property Canvas ;
    procedure HiThere ;
  end ;
  TPanelDemoForm = class(TForm)
    Panel1: TPanel;  // Interposed TPanel!
    Image1: TImage;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
  public
  end;
var PanelDemoForm: TPanelDemoForm;
implementation
{$R *.DFM}
procedure TPanel.HiThere ;
begin
  ShowMessage( 'Hi there!' ) ;
end;
procedure TPanelDemoForm.Button1Click(Sender: TObject);
begin
  Panel1.Canvas.Draw( 0, 0, Image1.Picture.Icon ) ;
end;
procedure TPanelDemoForm.Button2Click(Sender: TObject);
begin
  Panel1.HiThere ;
end;
end.
```

This limitation may not be of consequence, however: in this case the `Canvas` property wouldn't be published anyway.

If you really need changes to the design-time interface, then you'll just have to convert the class into a component and install it on the Component Palette. By that same token, this is also a great technique for testing out the features and behavior of new components to make sure they're working properly, before adding them to the Component Palette.

It's also possible to place interposer classes into their own unit(s), the only stipulation is that the unit containing the interposer class(es) must appear in the `uses` clause *after* the unit with the original component declarations (eg if our interposer version of `TPanel` appeared in a unit called `EnhCtrls`, then `EnhCtrls` must appear in the `uses` clause after `ExtCtrls`). Also, the aforementioned feature allowing access to protected members will not work with this approach (though they *will* be accessible in the interposer unit). An example project on the disk illustrates this.

This technique is actually not limited to components from the Component Palette, it can also be used to subclass `TForm`! This opens a whole world of possibilities for creating highly customized, but easily inheritable, form behavior (even in Delphi 1!). Listing 2 shows a simple example (a much more elaborate one is on the disk).

Theoretically the same should also be possible with `TApplication`. The snag I ran across here is related to the automatically created `Application` object that all Delphi projects get. This object is created in a routine buried in the `initialization` code for the `Controls` unit and there appear to be several simultaneous configurations going on in order for the proper startup and cleanup of `TApplication` to occur. So far I haven't figured out a way to interpose this process without serious VCL surgery, which is precisely what this technique is trying to avoid.

By the way, if you ever need to access the original version of the interposed component, you can get to it by fully qualifying the declaring unit (`ExtCtrls.TPanel` or `Forms.TForm` in this case).

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls, Buttons;
type
  TForm = class( Forms.TForm ) // Interposer TForm class
  public
    procedure About ;
  end ;
type
  TDemoForm = class(TForm)  // Interposed Tform!
    BitBtn1: TBitBtn;
    procedure BitBtn1Click(Sender: TObject);
  private
  public
  end;
var DemoForm: TDemoForm;
implementation
{$R *.DFM}
procedure TForm.About ;
begin
 MessageBeep( MB_ICONEXCLAMATION ) ;
 MessageBox( Handle, 'TForm Interposer Class Demonstration',
   'About...', MB_OK or MB_ICONEXCLAMATION or MB_TASKMODAL );
end;
procedure TDemoForm.BitBtn1Click(Sender: TObject);
begin
  About;
end ;
end.
```

➤ *Listing 2*

The first sample project on the disk shows how to add various kinds of often requested features to various standard VCL controls and how to use an interposer unit. There are other useful techniques too, over and above the class interposition. You'll note that I placed some of the new properties into the `published` sections of the classes, to show that they could easily be converted into real visual components with a bit more work. One caveat about that though: Delphi is *very* sensitive to installations of components with the same name (Delphi 1 would crash badly, I'm not sure what Delphi 3 does), so if you convert these to real components, *change their names!*

The second sample project shows how to interpose a form, adding some fairly elaborate custom form behavior. This probably should also be placed into an interposer unit so that the custom behavior could easily be applied to more than one form, but I left that as an exercise for the reader.

Stephen L Posey works in Pittsburgh, USA; you can email him at slposey@concentric.net