Visual Form Inheritance And Frames: Friend Or Foe?

by Guy Smith-Ferrier

Delphi 5 saw the introduction of frames, which have a lot in common with visual form inheritance (VFI), introduced in Delphi 2. My goal in this article is to give you an analysis of the strengths and weaknesses of VFI and frames, plus an explanation of why you would want to use them and what you would use them for.

VFI: A Quick Recap

VFI allows developers to create a form and then create new forms based on it. As such it is a clever blend of classic OOP inheritance, using Delphi's class inheritance, and visual inheritance, using Delphi's form designer. In short, you create a new form, save it, right click the form and select `Add to Repository`. Fill in the dialog box and your new form is made available to all projects.

A project can then create a new form which inherits from the first form in the repository by selecting `File|New` and then the page which has the form on it, clicking on the `Inherit` radio button, and then the `Ok` button. Voila! A new form, which inherits from the one in the repository.

Delphi's implementation is quite clever and fun to illustrate. Add components to the original form or modify, move or resize components in the original form and the changes are reflected in all of the original form's sub-forms. The sub-forms themselves can make their own modifications to existing components and add their own. You can also inherit from the sub-forms. The only restriction is that you can't delete components added to your parent forms (but you can set their `Visible` property to `False`). In addition, any component can easily be reset to its parent's behaviour after it has been changed by right clicking and selecting `Revert to Inherited`. Unfortunately, there is no selection process so all properties are restored to their inherited values. This is only a small inconvenience.

➤ *Listing 1*

```
procedure TBasicForm.FormKeyPress(
  Sender: TObject; var Key: Char);
if Key = #13 then begin
  Key:=#0;
  SendMessage(Handle,
    WM_NEXTDLGCTL, 0, 0);
end;
```

VFI: Why Use It?

Great, so now we know all about VFI. I have to admit that my view is that, despite a feature being impressive, I have to see good benefits to make me use it. That's what I hope to accomplish here.

There are at least four reasons why you might want to use VFI. The first is that you might want to create a form which is as simple as Delphi's own `TForm` but has one or two additional features or modifications and you want all forms in your application to inherit from this form. VFI will let you do this. Say, for example, you grudgingly accept that users want to use the `Enter` key to move from control to control throughout your application. This is a commonly requested feature and the code to achieve this is very straightforward: set the form's `KeyPreview` to `True` and add an `OnKeyPress` event (see Listing 1).

The problem for normal application development is that you would have to add this code to each and every form in the application. This would effectively mean block copying the code from the first form to each and every new form. As a general rule of programming if ever you encounter a problem where the solution is to block copy code then you should look for a better solution. VFI is a better solution. Create a new form which has the amendments to allow the `Enter` key to move to the next control and add the form to the repository. From here on you have two choices. The first choice is that you can remember to always create new forms from the form in the repository. Another rule I have is that whenever a programmer has to remember to do something then the programmer will, in the fullness of time, eventually forget what it is that they have to remember (I include myself in this group). So any solution where you have to remember to do something isn't a good solution. The second choice is to get Delphi to always use your form for all new forms. You can do this from `Tools | Repository`: select the page containing the form, select the form itself and check on the `New Form` checkbox.

Another reason for having a standard form used throughout the application is that this form is a good place to drop a component renaming component (which renames components). It is a very useful component which exists solely to aid application development and plays no part at runtime. It uses `TComponent.Notification` to detect when a developer drops a new component onto a form and then it renames the new component according to your own naming rules. Ray Lischner has a very customizable component renamer in his (still) excellent *Secrets Of Delphi 2* book, but there are freeware components available and you can easily write your own. The point is that the 'base' form is an excellent location for such a component.

The only problem with this solution is that it requires foresight. You must define your own basic form and add it to the repository right at the start of the project. A trick I often use is to define a new

form which is identical to TForm, add that to the repository, and always use that form thereafter. This allows me the incredibly powerful effect of being able to retrospectively change every form in my application once I have worked out what I want them to look like.

But, like I said, this requires foresight. An alternative, if you have bought into this idea after a project is already under way, is to change your PAS and DFM files. In the PAS file you would modify the form's class declaration line from

```
TForm1 = class(TForm)
```

to

```
TForm1 = class(TBasicForm)
```

In the DFM file you would modify the equivalent line from

```
object Form1: TForm1
```

to

```
object Form1: TBasicForm
```

There will also be more changes to make if you have added events to the parent form, but at least it is possible to change a form's ancestry after it has been created.

Getting back to the reasons why you might want to use VFI, the second reason is to create forms which have commonly used functionality. The most common example of this is the basic data maintenance form. Whether you use the BDE, ADO, IBExpress or another solution, the probability is that most of the maintenance forms in your application have a common look and feel and common functionality. VFI is perfect for ensuring this common look and feel and functionality. Define the basic maintenance form and add it to the repository. What I like about this approach is that it respects the most fundamental truth of application development: everything changes. When a change is required it is introduced into the parent and it cascades through all of the children. Clearly this is just common or garden

```
{$IFDEF CUSTOMERSPECIFIC}
  Form := TCustomerSpecificMaintForm.Create(Application);
{$ELSE}
  Form := TMaintForm.Create(Application);
{$ENDIF}
```

➤ *Above: Listing 2*

inheritance doing its usual stuff and this is a very old benefit of OOP. But that doesn't change how powerful and useful it is.

The third reason affects your design philosophy for customization. Vertical market applications often need to be customized for a specific customer. The best solution is to attempt to integrate the specific customer's needs into the basic package. But this is not always the optimum solution, as excessive use of this approach can lead to application bloat. The traditional solution to this problem is to copy the entire application to a separate directory and make the necessary modifications to the new copy. This is a truly awful solution and is one of the reasons why we have inheritance. As a result it isn't too surprising to learn that this is the third benefit of VFI. To solve this problem you would leave the original application completely intact and create a new customer-specific form which inherits from the original form. The customer-specific modifications would now be made to the child form. The original code would include conditional compilation as shown in Listing 2.

Clearly there is a law of diminishing returns involved in this process as the number of customer-specific versions increases, but if this number is low then VFI is an excellent solution.

The fourth reason is to create new data modules which inherit from other data modules. This would allow you to reuse data modules and still be able to make amendments which are only relevant to a specific context. This idea was more fully covered in Issue 53.

VFI: Problems

As you know from previous articles, it isn't my way to show only the good points in a feature and pretend that the bad points don't

➤ *Below: Listing 3*

```
TBasicForm = class(TForm)
private
  FEnterAsTab: boolean;
published
  property EnterAsTab: boolean
   read FEnterAsTab
   write FEnterAsTab;
end;
```

exist. So that brings me to the bad points. If you've ever written a component you'll know that one of the most common things you want to do is add your own properties to your component. It's the same for VFI: you want to add your own properties to a form which will appear in the Object Inspector. Unfortunately, this is where the system falls down.

Let's say my new form's class definition looks like the one shown in Listing 3. The idea is that the form has a property which allows the programmer to decide whether Enter should be treated as a Tab. When you add the form to the repository and create a new form which inherits from this form, you would expect to see the EnterAsTab property to appear in the Object Inspector. But it does not. The problem is that Delphi cheats. When you create any form in the IDE it is always a TForm despite what the Object Inspector or the class declaration in the PAS file might say. As a result your own properties are ignored.

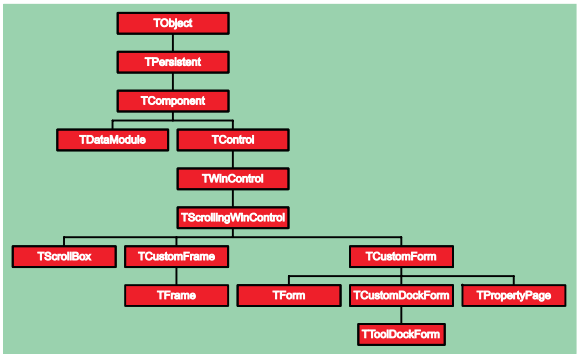
Fortunately, we can work around this by creating a new component which exists solely to hold the form's properties and to transfer them from the component to the form when the component is created at runtime. It isn't elegant but it works. Listing 4 shows a component called TBasicFormProperties which achieves this.

The class uses the TComponent.Loaded method to copy across its EnterAsTab property to the form's EnterAsTab property. Although this shows the basic approach of such a component it could be improved a little. The problem with this component is that it is tightly coupled

with the form to which it refers. A better solution would be to use RTTI to walk through the component's published properties looking for exact matches with published properties of the component's owner (ie the TBasicForm) and then assigning the values appropriately.

Another problem with VFI is that it relies on all of the components which are on the parent form including csInheritable in their ComponentStyle property. If a form contains a component which is not inheritable then Delphi will refuse to create the child form. A simple solution to this problem is to create new components which inherit from the offending components and change their constructor so that the csInheritable enumerated type is included in ComponentStyle. Listing 5 shows the constructor for a TInheritableNotebook which inherits from TNotebook and adds back the csInheritable enumerated type.

Of course, this is a bit shortsighted. There is usually a good reason why the component does not include csInheritable and this solution simply overrides the setting and ignores whatever problem results from inheriting from a form which includes such a component. In the case of TNotebook, csInheritable is excluded because of the way TNotebook is streamed. If you place a component on the TNotebook, save the application, close it and reload it, you will see that Delphi cannot reload the inherited form. However, the relative impact on any application of this limitation is likely to be very small as, in the whole of Delphi 5 Enterprise's VCL, there are only two components which do not include csInheritable (TTabbedNotebook and TNotebook) and both of these are on the Win3.1 page.



➤ *Figure 1*

Frames: A Quick Recap

At first sight frames have a lot in common with component templates which were introduced in Delphi 3. Don't misunderstand me here, I'm a big Delphi fan and I think it is a great product which is very well designed. However, it is unrealistic to expect that, just because you know and love a product, the product does not have any features which are an unmitigated disaster. Component templates are such a feature in my view. They get my award for the worst feature in any release of Delphi. The reason? They are unmaintainable and effectively block copy a collection of components. As such, they create the typical maintenance nightmare you get from any solution which copies instead of inherits. Frames (and form components) are the correct answer to the problem.

So what is a frame? A frame is a form-like component which can be dropped onto a form at design-time. You can create a new frame from File | New Frame. Your new frame looks and behaves just like a form but for clarity in the form designer the grid is not visible. You can drop components onto the frame and design using all of the techniques you use to design forms. However, although frames have a lot in common with forms, they are not actually TForms. Figure 1 shows the frame and form class hierarchy.

To use a frame you can select the frame icon (the first icon on the Standard page of the palette) and drop it on a form. The resulting behaviour is different to all other components in Delphi, in that it offers a dialog box showing all of the available frames in the application and allows the developer to select one to drop onto the form.

Frames have a lot in common with VFI because when the original frame is changed all frames which inherit from it reflect the change. In addition, all the features for customisation of the new frame (eg moving components to different locations and adding new components) and reverting to the inherited frame are the same as for VFI. Furthermore, frames can be added to and retrieved from the repository in same way as forms. One feature which frames have over VFI is that frames can be added directly to the palette (right click on a frame and select Add To Palette).

Frames: Why Use Them?

There are (at least) two reasons why you might want to use frames. The first is to replace component templates. Component templates allow you to create a pseudo-component which is a composite component, ie a component which is a collection of other components. A classic example is the dual

```
TBasicFormProperties = class(TComponent)
private
  FEnterAsTab: boolean;
public
  procedure Loaded; override;
published
  property EnterAsTab: boolean read FEnterAsTab write FEnterAsTab;
end;
procedure TBasicFormProperties.Loaded;
begin
  inherited;
  if Owner is TBasicForm then
    TBasicForm(Owner).EnterAsTab:=FEnterAsTab;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
constructor TInheritableNotebook.Create(AOwner: TComponent);
begin
  inherited;
  Include(FComponentStyle, csInheritable);
end;
```

listboxes and associated buttons used in `Project | Options | Forms` (see Figure 2).

The combination of the two listboxes, the four buttons and the drag and drop events which go with the listboxes is an ideal example of a composite component which gets reused again and again in applications.

The second reason to use frames is to drop 'forms' onto other forms. The purpose of this is to create a plug-in architecture. A good working example of this idea (sadly not written in Delphi and therefore not using frames) is Microsoft Management Console (used with Microsoft Transaction Server, Internet Information Server and others). If you haven't seen MMC (see Figure 3) you'll be seeing more of it in Windows 2000.

The idea is that you write a 'host' program. This host program has an interface to read plug-in modules. In MMC this is a set of 'snap-in' interfaces. In a Delphi program this could be a set of classes or interfaces where each plug-in is probably a dynamically loaded package (using `LoadPackage` and `UnloadPackage`). The host program's API would allow plug-ins to populate a treeview on the left-hand side with nodes. Each node would have a 'form' or frame which would be automatically displayed on the right-hand side as its associated node was clicked on. There is much more to the idea of a plug-in architecture than this simple sketch and a whole article could be written on this subject, but I hope this is enough to explain the idea.

Where frames help in this architecture is to allow the 'forms' (frames) to be easily developed independently of the host program. Of course, anyone who has tried to write such an architecture prior to Delphi 5 will know that you don't need frames to achieve this result. `TForms` can be embedded on other forms at runtime using the code shown in Listing 6.

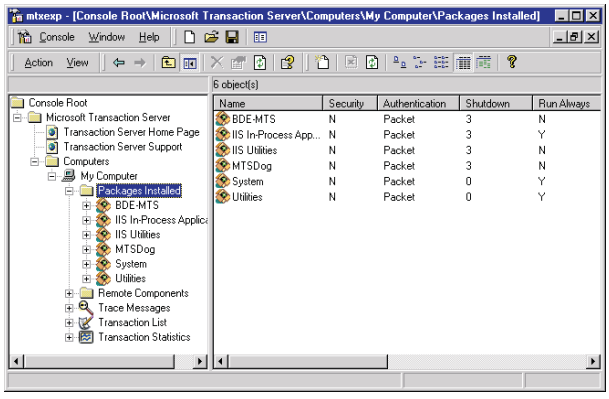
However, frames are more lightweight than forms and more suited to this task.

Frames: Problems

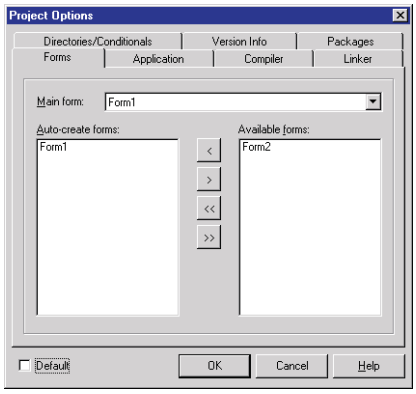
Just as frames share many benefits with VFI they also share many of their problems. Frames do not allow components which do not include `csInheritable` in their `ComponentStyle` to be added to the original frame. The workaround for frames is the same as the workaround for VFI, but also has the same caveat as for VFI.

Another annoying problem is that the process of storing a form in the repository and checking on the `New Form` checkbox to ensure that all new forms created inherit from the one in the repository (as described earlier) also applies to frames. When you set this up and then create a new frame, you don't get a new frame at all. Instead, you get a new form which inherits from the one in the repository just as if you had asked for a new form instead of a new frame. This behaviour is still true in Delphi 5 Update Pack 1 (aka Delphi 5.01) so I am left wondering whether this is, in fact, the intended behaviour.

Another apparent problem is the problem of inheritance. One of the benefits of VFI was that forms inherited from their parents and this included code inheritance. Frames also support inheritance in the same way with the exception that one frame cannot inherit from another frame. At first sight this appears to be a hindrance to the plug-in architecture described above. The problem is that all plug in frames will want to have a large amount of code inherited from some base class to get the plug-in architecture to work. Without code inheritance this appears to be a problem.



➤ *Figure 3*



➤ *Figure 2*

As you can guess from the way the last paragraph was phrased, there is a solution to this problem. The solution is a sign of our times. Many years ago programmers would make their programs extensible using DLL plug-ins and a function-based architecture. Of course, the programming world moved on from function-based programming to object oriented techniques, so it became more common to provide the same architectures using classes. However, our world has moved on even further and the solution today is to use interfaces. Interfaces are the solution to this problem.

The idea is that you would create an interface called, say, `IPlugInFrame`, and you would modify the class declaration for each frame to include this interface, for example:

```
TCustomerFrame =
  class(TFrame, IPlugInFrame)
```

(Delphi has no objection to you manually modifying either frame or form class declarations to include your own interfaces). Of course, at this point, you could argue that all we've achieved is the ability to share a common interface amongst frames. What we have not achieved is the ability to share common *code* amongst frames. This is one reason for the introduction of the `implements` directive in Delphi 4. This directive

allows an interface to be implemented by a property instead of directly by the class. As such it represents a simple indirection. Listing 7 fleshes out the frame class.

The `IPlugInFrame` interface and the `TPlugInFrame` class are left blank to concentrate on just the solution. The frame's constructor creates a `TPlugInFrame` object from which the `IPlugInFrame` interface is automatically extracted. Thus the code in the `TPlugInFrame` class is reused for every frame (which is implemented in this way) giving the same benefits as code inheritance.

```
// create the new form (the HostForm is the owner)
Form:=TPlugInForm1.Create(HostForm);
Form.BorderStyle:=bsNone;
// add the new form to the scroll box on the right hand side
Form.Parent:=HostForm.RHSScrollBox;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
IPlugInFrame = interface
end;
TPlugInFrame = class(TInterfacedObject, IPlugInFrame)
end;
TCustomerFrame = class(TFrame, IPlugInFrame)
private
  FPlugInFrame: IPlugInFrame;
public
  constructor Create(AOwner: TComponent); override;
  property PlugInFrame: IPlugInFrame
  read FPlugInFrame write FPlugInFrame
  implements IPlugInFrame;
end;
constructor TCustomerFrame.Create(AOwner: TComponent);
begin
  inherited;
  PlugInFrame:=TPlugInFrame.Create;
end;
```

Form Components

One related subject which I haven't mentioned here is creating form components. A form component is a component which uses a container to hold many other components. The container could be a form (as the name implies) but it is much easier to implement if the container is another standard component such as a scroll box, panel or group box. This has many of the benefits of VFI and frames but also some other drawbacks.

Friend Or Foe?

So what we have learnt is that VFI and frames have a lot in common but also some differences. Table 1 shows a summary of the differences between VFI, frames and form components.

Using this table you can decide which approach is best for you. Clearly, form components cannot be designed visually and this makes development slow and painstaking because you have to manually write the code to create the components and initialise their properties. If you decide to take this approach you would be well advised to write a wizard which would allow you to design the component visually on a form and then generate the equivalent code to create such a component. A problem with VFI is that the resulting forms cannot be dropped onto other forms at design-time (you can drop them onto other forms in code at runtime though). One of the biggest differences between form components and VFI/frames is that form components are fixed by the component designer whereas VFI/frames can still be modified by the end programmer. Having the layout and content fixed is better if you want the layout and content fixed but worse if you don't want it fixed. That may sound like a very obvious statement but it is intended to show that there is no right or wrong here but simply what fits your requirements.

➤ *Table 1*

	Form Components	Visual Form Inheritance	Frames
Can design visually ?	No	Yes	Yes
Available from...	Palette	Repository	Palette or Repository
Can be placed on a form at design-time ?	Yes	No	Yes
The end programmer can modify the form/frame ?	No	Yes	Yes
Can new forms/frames inherit from a parent other than TForm/TFrame ?	Yes	Yes	No
Can add interfaces ?	Yes	Yes	Yes

Conclusion

Visual Form Inheritance was a great feature when it was added in Delphi 2 and it remains a very viable and useful feature today. The introduction of frames in Delphi 5 should not be seen as a replacement for VFI, as VFI is suitable for situations which frames are not and vice versa. Frames should certainly be seen as a replacement for component templates.

Both VFI and frames have their pros and cons and form components can be appropriate in situations which neither VFI and frames are suited to.

Guy Smith-Ferrier is Technical Director of Enterprise Logistics Ltd (www.EnterpriseL.com), a training and development company specialising in Delphi which is now running ADO courses. He can be contacted at gsmithferrier@EnterpriseL.com