

Embedded Forms – Divide and Conquer

Anytime I discuss Embedded Forms, a lot of developers (notably non-Delphi developers) assume I'm referring to the old MDI interface. What I'm actually referring to is embedding actual child forms into parent forms (as an alternative to using Frames).

Assuming we have a ChildForm.Execute method as follows:

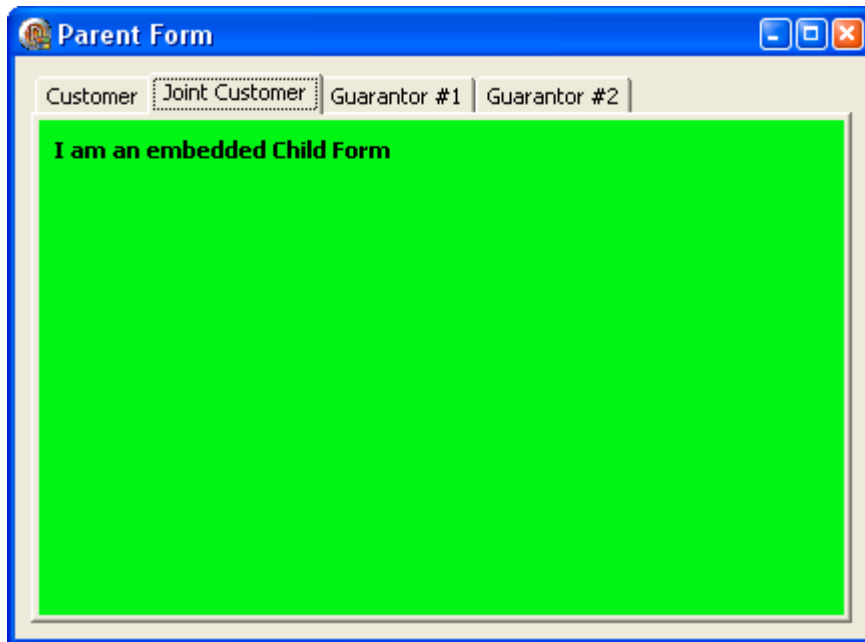
```
procedure TChildForm.Execute(Parent: TWinControl);  
begin  
    Self.Parent:=Parent;  
    BorderStyle:=bsNone; // Can be set at design time  
    Align:=alClient;     // Can be set at design time  
    Show;  
end;
```

From a Parent Form, it is simply a matter of Creating and Executing the ChildForm to embed it.

```
Procedure TParentForm.Create(Sender : TObject);  
Begin  
    TChildForm.Create(Self).Execute(Panel1);  
End;
```



Although this is a very simple example, the technique becomes much more powerful when you need to embed multiple forms into a parent form (in a PageControl for example), or have a form that can contain variable content based on some data.



In this scenario, a ChildForm is embedded into each TabSheet of the PageControl. In this example:

```
TChildFormCustomer.Create(Self).Execute(TabSheet1);  
TChildFormJointCustomer.Create(Self).Execute(TabSheet2);  
TChildFormGuarantor1.Create(Self).Execute(TabSheet3);  
TChildFormGuarantor2.Create(Self).Execute(TabSheet4);
```

The first advantage of this is **encapsulation**. Simply put, a lot of the complexity of each child form is encapsulated in that child form. The parent form primarily needs to create and initialise each child form.

The second advantage is **manageability**. The complexity is split more or less 4 ways with some overhead in the parent. Say each child takes 500 lines of code to say validate and display input, if this was one form, it would be over 2000 lines of code.

The third advantage is **modularity**. Each form is encapsulating its own code, so firstly code from TabSheet1 doesn't get mixed with TabSheet2. As a result it's much easier to divide each section between different programmers. Additionally debugging should be easier and it is possible to completely "turn off" one of the embedded tabs if the need arises.

Things start to get a little more interesting when you start looking at **inheritance** and **polymorphism**.

We may have different Customer Types, perhaps built in a class hierarchy something like this (not all relevant methods are included):

```
TChildForm
    Procedure TChildForm.Execute(Parent : TWinControl);

TChildFormCustomer = Class(TChildForm)
    Class Function Create(aOwner : Tcomponent;
                          CustData : TCustData) :
                          TChildFormCustomer;overload;
    Function Load(CustRef : String) : Boolean;virtual;
    Function Save : Boolean;virtual;

TChildFormPrivateCustomer = Class(TChildFormCustomer)
TChildFormBusinessCustomer = Class(TChildFormCustomer)
```

```
Procedure TParentForm.Create(Sender : TObject);
Var
    Customer : TChildFormCustomer;
Begin
    //1. Call our class builder, it will return either a
    //    TChildFormPrivateCustomer or TChildFormBusinessCustomer
    //    and in the parent form we don't care which!
    Customer:=TChildFormCustomer.Create(aOwner, CustData);
    //2. Load our data, it could either be an inherited or
    //    polymorphic method, again we don't really care.
    Customer.Load(CustData.CustRef);
    //3. Our standard Execute as per TChildForm;
    Customer.Execute(TabSheet1);
End;
```

... and the class builder could call the Load and Execute for us if we did something like this:

```
Class Function Create(aOwner : Tcomponent; CustData : TcustData; Parent : TWinControl)
: TChildFormCustomer;overload;
Begin
    Result:=Nil;
    If CustData.CustType = ctPrivate then
        Result:=TchildFormPrivateCustomer.Create(aOwner)
    Else
    If CustData.CustType = ctBusiness then
        Result:=TchildFormBusinessCustomer.Create(aOwner);
    If Assigned(Result) then
    Begin
        Result.Load(CustData);
        Result.Execute(Parent);
    End;
End;
```

Our calling code is then even more simplified to:

```
Customer:=TChildFormCustomer.Create(aOwner, CustData, TabSheet1);
```

Final Notes:**Why use this instead of Delphi Frames?**

Frames have the advantage of working at design time, but for 2 reasons I prefer to use the embedded form method.

Firstly you must select the Frame at Design Time to display. I have no doubts there are ways around this, but I find it too restrictive, notable when I don't know what I want to display until runtime.

Secondly, at design time Frames can let you get too sloppy with your interface to the Frame. For instance, create a Frame with one button and create an implementation. Embed the Frame in a parent, and double click the button (from the Parent) – Delphi will generate code like this:

```
procedure TForm1.Frame21Button1Click(Sender: TObject);  
begin  
    Frame21.Button1Click(Sender);  
end;
```

Now although you can manually do this with an embedded form, I believe that you are less likely too, and hence less so concentrate on encapsulation and interfacing when using frames.

Can you do the same in DotNET / WinForms?

Yes, refer to the Form.TopLevel Property

And...

This is the 3rd of 4 related Delphi 5 minute articles. In the first 2 articles we looked at some nifty class building techniques, using Array of Class and Class Functions. In this article we looked at embedded forms. The 4th will hopefully tie all the concepts together and deals with form messaging.