

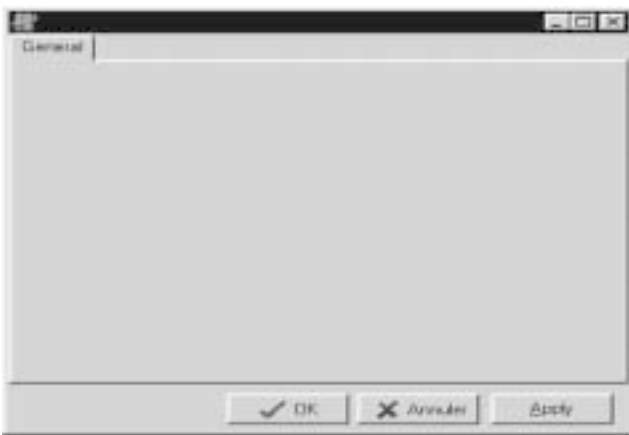
Showing Off

by Joanna Carter

There I was, sitting in Provence on a day when the temperature had reached 22°C, when I suddenly remembered our beloved editor had asked for another article; so here it is....

So far in this series on the Object Store, we have looked at those classes which are used in the day to day writing of an application. But, so far, we have not looked at the business of presenting objects and collections on a form so that they can be interacted with.

In order to make life simple, I have designed a base form that can be inherited from that will cope with editing any single object; I have called it TPDProperties to reflect the fact that is primarily a property editing form.



As you can see, it is a simple form with a Page Control and three buttons, but I have also written all the code that is necessary to allow correct editing of any object that is derived from TPDObject.

Here is the declaration for TPDProperties :

```
type
  TPDPropertiesClass = class of TPDProperties;

  TPDProperties = class(TForm)
    pnlButtons: TPanel;
    btnApply: TButton;
    btnOk: TBitBtn;
    btnCancel: TBitBtn;
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    procedure btnOKClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
    procedure btnApplyClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var
      Action: TCloseAction);
```

There is nothing more than the component declarations in this section.

```
private
  fObject: TPDObject;
```

This is a pointer to the object that the form is going to allow us to edit.

```
fDirty: Boolean;
```

I have added this field to allow us to keep track of whether the information in the edits that will be placed on a derived form has changed. Although it would be more usual to use the osModified flag in the ObjectState of the object to indicate that any changes have been made, this can provide an alternative mechanism.

```
procedure SetDirty(const Value: Boolean);
property Dirty: Boolean
  read fDirty
  write SetDirty;
protected
  function GetObject: TPDObject;
```

GetObject is a protected function that allows us to get at the underlying object from derived classes. As I will explain later, this can be 'overridden' to return an object of a more specific type than TPDObject.

```
procedure ReadProperties; virtual;
  abstract;
procedure WriteProperties; virtual;
```

These two methods are the only places on the form where you will need to write any significant amount of code. Their purpose is simple; ReadProperties allows us to assign the properties of the object into the edits on the form and WriteProperties allows us to assign the contents of edits back into the object.

```
public
  constructor Create(AObject: TPDObject);
    reintroduce; overload; virtual;
  constructor Create(AClass: TPDObjectClass);
    reintroduce; overload; virtual;
  destructor Destroy; override;
```

Create is overloaded to allow us to create a form in either of two ways. We can either pass an existing object into the form, such as for editing, or we can ask the form to create a new object of the desired class for the purpose of inserting that object into the store.

```
published
  procedure edtChange(Sender: TObject);
end;
```

Finally, I have provided an event handler that should be connected to the event of each control that is triggered when that edit changes.

```
constructor TPDProperties.Create(AObject:
  TPDObject);
begin
  inherited Create(nil);
  if fObject = nil then
  begin
    fObject := AObject.PDClassType.Create;
    fObject.Assign(AObject);
  end;
  try
    Caption := Format('%s : %s Properties',
      [fObject.DisplayName,
        fObject.Descriptor]);
    ReadProperties;
```

```

    Dirty := False;
except
    on E: EPDException do
        Application.MessageBox(PChar(E.Message),
            PChar(E.Caption),
            MB_OK or
            MB_ICONWARNING);
    end;
end;

constructor TPDProperties.Create(AClass:
TPDObjectClass);
begin
    fObject := AClass.Create;
    Create(fObject);
end;

```

The first constructor is the one that we can use for editing existing objects and the parameter is the object that requires editing. You will notice that, after calling the inherited constructor, the first thing I do is to create an object of the same class type as the object that is passed in, and then I assign the parameter object to the private field. This is to ensure that, if the object that was passed in gets freed outside of this form, we still have a valid reference that can be used within the form.

I have used two of the properties of TPDObject to construct a presentable caption for the form; the DisplayName will give us a neater version of the kind of object that we are dealing with, whilst Descriptor will give us a summary of the object itself.

ReadProperties is the method where we will read the properties of the object into the edits. This may cause an exception and so I have wrapped this section in a try-except block to allow us to give a visual representation of the exception that is appropriate to the environment that we are working in. Were we to be working in a HTML environment then, of course, you would not handle exceptions by using MessageBox. Finally, we initialise the 'dirty' flag.

```

procedure TPDProperties.SetDirty(const Value:
                                Boolean);
begin
    fDirty := Value;
    btnApply.Enabled := fDirty;
end;

```

The SetDirty method simply sets the private field to the value required and then uses that value to enable or disable the Apply button. As you will see later, this flag will also affect the behaviour of the OK button.

```

function TPDProperties.GetObject: TPDObject;
begin
    Result := fObject;
end;

```

This is the protected method that allows us to see the object behind the form from derived classes.

```

procedure TPDProperties.WriteProperties;
begin
    GetObject.Store;
end;

```

You should have noticed that ReadProperties was not only a virtual but also an abstract method; this is because there is absolutely nothing that we can know about reading properties in this class. However, WriteProperties will definitely want to tell Object to store itself: this method will be called as inherited from derived classes.

```

procedure TPDProperties.edtChange(Sender:
                                TObject);
begin
    Dirty := True;
end;

```

As you can see, this is the event handler that will respond to any edits that we have placed on the form, providing we have assigned the appropriate event to it. All that it does is to set the Dirty flag to true as soon as a change is made to any of the edits.

Now we come to the button logic. This should not need overriding in the majority of cases and is intended to work regardless of the class of the object being edited.

```

procedure TPDProperties.btnOKClick(Sender:
                                TObject);
begin
    if fDirty then
        btnApplyClick(nil);
    if not fDirty then
        Close;
end;

```

If the OK button is clicked then we need to be able to detect whether any changes have been made in the form. If the Dirty flag is set, then the logic in the Apply button will be called first and then the form will be closed. If we wanted to show this kind of form modally, then you would set ModalResult := mrOK instead of just closing the form.

```

procedure TPDProperties.btnCancelClick(Sender:
                                TObject);
begin
    Close;
end;

```

When you think about it, cancelling edits to an object in memory is as simple as just not bothering to save the changes and freeing the object, so all we need to do here is either to close the form or set ModalResult := mrCancel.

```

procedure TPDProperties.btnApplyClick(Sender:
                                TObject);
begin
    try
        WriteProperties;
        Dirty := False;
    except
        on E: EPDException do
            Application.MessageBox(PChar(E.Message),
                PChar(E.Caption),
                MB_OK or
                MB_ICONWARNING);
    end;
end;

```

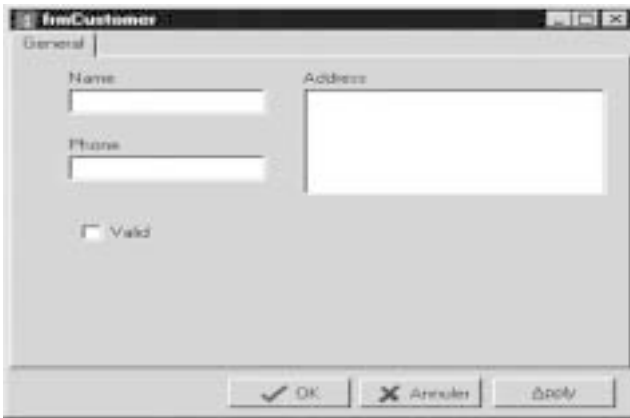
We have already seen that the OK button calls the Apply button code just before closing the form, and that is about the only difference between the two buttons. If you look at the way that most property pages work in Windows, you will see this behaviour. The Apply button updates the object being viewed without closing the form, whereas the OK button closes the form as well. It is this kind of consistent behaviour that makes an application feel more friendly to the user as it feels like they are only using another part of Windows rather than a different application.

If any of the validation checks that we make when we assign the edits to the properties should fail, then this should raise an exception which can be trapped here and shown to the user. The presence of this exception will do three things:

it will cause the Store of the object to fail, it will stop the Apply button becoming disabled because the Dirty flag is still set and it will stop the form from closing, thereby allowing corrections to be made to values in edits to comply with any validation.

```
procedure TPDProperties.FormClose(Sender:
    TObject; var Action: TCloseAction);
begin
    Action := caFree;
end;
```

Finally, if we are showing this form modally, then we would not write anything in the FormClose event as it would be possible to Free the form from within the calling code. If the form has been used non-modally, then we should use the FormClose event to ensure that the memory for the form is returned to the heap.



Here is a form that has been derived from TPDProperties for use with a Customer object. It is by no means intended to be a complex form, but merely to demonstrate the code that is required to adapt the base form for a specific class.

In addition to placing and naming the components on the form I have also assigned the OnChange event of all the edits to the edtChange method that I wrote in the base class.

Now if we look at the changes that have been made in this class we will find there are only three methods, one of which is only one line long. We may not be using data-aware controls, but I do not believe in making work for myself.

```
TfrmCustomer = class(TPDProperties)
    edtName: TEdit;
    edtPhone: TEdit;
    mmoAddress: TMemo;
    lblName: TLabel;
    lblPhone: TLabel;
    lblAddress: TLabel;
    xbValid: TCheckBox;
private
    { Déclarations privées }
protected
    function GetObject: TCustomer;
    procedure ReadProperties; override;
    procedure WriteProperties; override;
public
    { Déclarations publiques }
end;
```

As you can see there is very little to this class, only the additional components are added to the derived class, the rest are still in the base form.

```
function TfrmCustomer.GetObject: TCustomer;
begin
    Result := TCustomer(inherited GetObject);
end;
```

As I mentioned before the purpose of the GetObject method is to provide easy access to the object that is held in the private variable. Because the object is stored in a TPDObj pointer, we can use a method such as this to cast it back to the correct type for ease of use within the form.

```
procedure TfrmCustomer.ReadProperties;
begin
    with GetObject do
    begin
        edtName.Text := Name;
        edtPhone.Text := Phone;
        mmoAddress.Text := Address;
        xbValid.Checked := Valid;
    end;
end;

procedure TfrmCustomer.WriteProperties;
begin
    with GetObject do
    begin
        Name := edtName.Text;
        Phone := edtPhone.Text;
        Address := mmoAddress.Text;
        Valid := xbValid.Checked;
    end;
    inherited WriteProperties;
end;
```

You certainly won't need a degree in rocket science to write the above code, there just isn't the need for anything more complicated. If any validation is required, it is done in the Set methods of the properties. Now you can see just how little work would be required to create an HTML page instead of a form!

Conclusion

We looked at creating a base form that is capable of knowing about a persistent object. Then we designed a simple derived form to demonstrate how to handle ordinary string properties.

I have not covered things like Combo Boxes as I want to do that as part of the next article, which will cover the creation of a generic search/browse form which can also be used for providing lookup support.

Oh, by the way, I am not just sunning myself in the South of France, I am working down here for six weeks. Honest! *[Now it never occurred to us that she could be there for fun, did it, boys and girls? Ed.]*



Joanna Carter is a writer, developer and trainer specialising in requirements-driven training and consultancy. Joanna is involved with mentoring several companies in the setting up of mapping object-oriented systems into relational databases. You can e-mail her at JoannaC@btinternet.com and her web site is at joannac.btinternet.co.uk.