Editores de Propiedades

Es este artículo vamos a ver como crear editores de propiedades. Para ello nos basaremos en dos editores que se utilizan en algunos de los componentes que podéis descargaros de esta página : **TFieldsProperty** y **TFormatMaskProperty**, estos se utilizan para editar propiedades de componentes como *TExcelExport*, *TFieldsType*, *TLog*, etc.

Antes de nada y como ya se desprende del párrafo anterior, un editor de propiedades va a servir para editar propiedades de un componente en el inspector de objetos de delphi. El inspector de objetos sólo trabaja con cadenas de caracteres, aunque las propiedades sean integer o de cualquier otro tipo, así que nuestro editor deberá hacer las correspondientes transformaciones de datos. A parte de la edición de propiedades, el editor que creemos nos podrá servir, por ejemplo, para acotar posibles valores, elección de un valor entre varios, mostrar cajas de diálogo para hacer más fácil de entrada de datos, etc.

Delphi define una serie de editores por defecto, los cuales servirán como base de los que podamos crear, es decir, nuestros editores tendrán como ancestro a alguno de los ya existentes y por lo tanto heredarán todas las propiedades y métodos de estos.

Los editores definidos por delphi (en su versión 6) en DesignEditors.pas son :

Editor	*Descripción*
TOrdinalProperty	Clase base para todos los editores de propiedades de ordinales (Enteros, enumerados,etc).
TIntegerProperty	Editor para enteros de cualquier tamaño.
TCharProperty	Para tipos Char y cualquier subrango de Char.
TEnumProperty	Tipos enumerados definidos por el usuario.
TFloatProperty	Para propiedades de coma flotante.
TStringProperty	Cadenas de caracteres.
TSetElementProperty	Para cada elemento de un conjunto (set). Cada elemento se muestra como un Boolean.
TSetProperty	Editor por defecto para conjunto de elementos (set).
TClassProperty	Editor para propiedades que son a su vez objetos.
TMethodProperty	Editor para propiedades que son punteros a métodos, es decir, eventos.
TComponentProperty	Editor para propiedades que se refieren a un componente. No es lo mismo que TClassProperty. En este caso el inspector de objetos permite al usuario especificar el componente a que se refiere (es el caso de la propiedad ActiveControl).
TColorProperty	Editor para Tcolor.
TFontNameProperty	Editor para nombre de fuentes. Muestra una lista desplegable con las fuentes.
TFontProperty	Editor de TFont.
TInt64Property	Editor para Int64 y derivados.
TNestedProperty	El editor utiliza el editor de propiedades del padre.
TInterfaceProperty	Editor para referencias de interfaces.
TComponentNameProperty	Editor para la propiedad Name
TDateProperty	Editor de la fecha de un TDateTime
TTimeProperty	Editor de la hora de un TDateTime.
TDateTimeProperty	Editor de TDateTime.

Editor	*Descripción*
TVariantProperty	Editor de tipos Variants

Entonces dependiendo de la funcionalidad que queramos que tenga haremos que nuestro editor descienda de alguno de estos. En nuestro caso vamos a heredar de TStringProperty para el editor de TFormatMask y de TClassProperty para el editor de TFields. En el primer caso porque la propiedad será una cadena de caracteres y en el segundo una clase.

Editor TFormatMaskProperty

Antes de nada decir que TFormatMask es una clase definida como *Type String* que contendrá una cadena de formato de fecha/hora que se utilizará en el componente TLog (puedes bajarlo de www.i-griegavcl.com). Para hacer más fácil la introducción de una máscara correcta y dar la posibilidad de tener unos valores predefinidos, vamos a hacer un editor de propiedades que mostrará una caja de diálogo para llevar a cabo todo esto.

Como hemos dicho para hacer el editor para la propiedad TFormatMask heredamos de TStringProperty. La definición de esta clase es :

```
TStringProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetEditLimit: Integer; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
```

Los procedimientos GetValue y SetValue que sirven para obtener y de devolver el valor de la propiedad del inspector de objetos. Están definidas de la siguiente forma :

```
function TStringProperty.GetValue: string;
begin
  Result := GetStrValue;
end;

procedure TStringProperty.SetValue(const Value: string);
begin
  SetStrValue(Value);
end;
```

GetStrValue y SetStrValue son métodos definidos en TPropertyEditor que lo único que hacen es recuperar o asignar el valor de la propiedad como una cadena de caracteres.

Como primer paso vamos a ver la definición del editor y después pasaremos a explicar paso a paso cada parte :

```
TFormatMaskProperty = class(TStringProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
  end;
```

El procedimiento Edit servirá para editar la propiedad y la funcion GetAttributes le informará al inspector de objetos sobre atributos especiales a tener en cuenta para editar la propiedad, estos atributos pueden ser los siguientes :

Atributo	*Descripción*
paValueList	Devuelve una lista de valores enumerados. Para rellenar la lista se utiliza GetValues(). (p.e. BorderStyle de Tform)
paSubProperties	La propiedad tiene subpropiedades que el inspector de objetos debe mostrar identadas, por ejemplo Tfont. El atributo paValueList es obligatorio en este caso.
paDialog	El inspector de objetos muestra un botón ellipsis(...) para invocar al metodo Edit del editor de propiedades que se encargará de mostrar una caja de diálogo.
paMultiSelect	La propiedad debe ser mostrada cuando se seleccionan varios componentes en el 'Form Designer', esto permite cambiar el valor de una propiedad de varios componentes a la vez.
paAutoUpdate	El método SetValue() se llama cada vez que se produce un cambio en la propiedad, en caso contrario sólo se llama cuando se pulsa enter o nos movemos a otra propiedad del inspector de objetos.
paFullWidthName	El valor no necesita ser mostrado. El inspector de objetos utiliza todo el ancho para mostrar el nombre de la propiedad.
paSortList	La lista mostrada por GetValues() aparecerá ordenada.
paReadOnly	El valor de la propiedad no puede ser modificado.
paRevertable	La propiedad puede revertir a su valor original. Hay propiedades de las que no se puede recuperar el valor original como es el caso de TFont.

Nosotros redefiniremos el procedimiento Edit y la Función GetAttributes para hacer que el editor muestre una caja de diálogo que nos permitir introducir la máscara. La implementación será :

```
function TFormatMaskProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog];
end;
```

Le indicamos al inspector de objetos que queremos que llame al método Edit de nuestro editor de propiedades [paDialog].

El procedimiento Edit será :

```
procedure TFormatMaskProperty.Edit;
var
  DatemaskEditor: TfrmDateMask;
begin
  DatemaskEditor := TfrmDatemask.Create(Application);
  try
     DatemaskEditor.txtTest.text:=GetValue;
     if DatemaskEditor.ShowModal=mrok then
     begin
        SetValue(DatemaskEditor.txtTest.text);
        designer.Modified;
     end;
  finally
    DatemaskEditor.Free;
```

```
    end;
  end;
```

Lo que hace este procedimiento es lo siguiente :

1.- Define una variable de tipo TFrmDateMask, esta clase es un form que nos servirá como caja de diálogo, más adelante veremos su definición.
2.- Crea una instancia del objeto.
3.- Recupera el valor de la propiedad (GetValue) y lo asigna a un TEdit del form de la instancia anterior.
4.- Cuando la caja (form) de diálogo se cierre pulsando el botón Ok. coge el valor del Edit anterior y lo asigna a la propiedad (SetValue).
5.- Avisa al Designer de que ha habido una modificación
6.- Libera la instancia.

En vez de utilizar los métodos GetValue y SetValue, en nuestro caso, hubiéramos podido utilizar GetStrValue y SetStrValue definidos en TPropertyEditor, aunque lo correcto es hacerlo con GetValue y SetValue.

Bueno, pues ya sólo nos falta registrar el editor para que delphi lo utilice :

```
RegisterPropertyEditor(TypeInfo(TFormatMask), nil, '',TFormatMaskProperty);
```

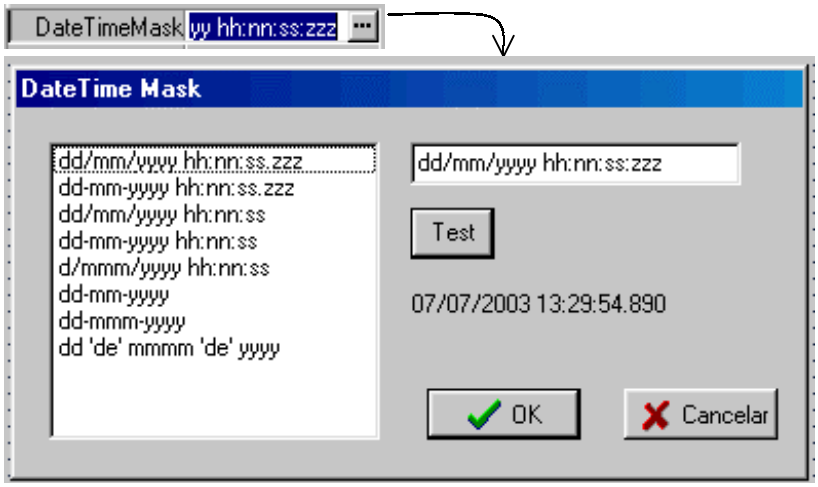
La definición de este procedimiento es :

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo; ComponentClass:
TClass; const PropertyName: string; EditorClass: TPropertyEditorClass);
```

dónde :

PropertyType informa sobre la propiedad a la que se le aplicará el editor. Es un puntero a la RunTime Type Information (RTTI) de la propiedad a editar.
ComponentClass indica el componente que contiene la propiedad. En nuestro caso lo ponemos a nil para poder utilizar el editor para este tipo de propiedad en cualquier componente.
PropertyName indica el nombre de la propiedad. Nosotros lo ponemos como una cadena vacía para que en distintos componentes pueda tener valores distinto y no tenga que llamarse siempre igual.
EditorClass es el editor de propiedades que hemos creado.

El resultado final será el siguiente :



La definición de la caja de diálogo TfrmDateMask es la siguiente y para crearla utilice 'New Application' del delphi. :

DateMask.dfm

```
object frmDateMask: TfrmDateMask
  Left = 289
  Top = 231
  BorderIcons = []
  BorderStyle = bsDialog
  Caption = 'DateTime Mask'
  ClientHeight = 178
  ClientWidth = 383
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  Position = poScreenCenter
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 13
  object lblTest: TLabel
    Left = 192
    Top = 88
    Width = 31
    Height = 13
    Caption = 'lblTest'
  end
  object FormatList: TListBox
    Left = 16
    Top = 16
    Width = 161
    Height = 145
    ItemHeight = 13
    Items.Strings = (
      'dd/mm/yyyy hh:nn:ss.zzz'
      'dd-mm-yyyy hh:nn:ss.zzz'
      'dd/mm/yyyy hh:nn:ss'
      'dd-mm-yyyy hh:nn:ss'
      'd/mmm/yyyy hh:nn:ss'
      'dd-mm-yyyy'
      'dd-mmm-yyyy'
      'dd '#39'de'#39' mmmm '#39'de'#39' yyyy')
    TabOrder = 0
    OnClick = FormatListClick
  end
  object txtTest: TEdit
    Left = 192
    Top = 16
    Width = 161
    Height = 21
    TabOrder = 1
    Text = 'dd/mm/yyyy hh:nn:ss.zzz'
  end
  object Button1: TButton
    Left = 192
    Top = 48
```

```
      Width = 41
      Height = 25
      Caption = 'Test'
      Default = True
      TabOrder = 2
      OnClick = Button1Click
    end
    object btnAceptar: TBitBtn
      Left = 200
      Top = 136
      Width = 75
      Height = 25
      TabOrder = 3
      Kind = bkOK
    end
    object btnCancelar: TBitBtn
      Left = 296
      Top = 136
      Width = 75
      Height = 25
      Caption = 'Cancelar'
      TabOrder = 4
      Kind = bkCancel
    end
  end
```

DateMask.pas

```
unit DateMask;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons;

type
  TfrmDateMask = class(TForm)
    FormatList: TListBox;
    lblTest: TLabel;
    txtTest: TEdit;
    Button1: TButton;
    btnAceptar: TBitBtn;
    btnCancelar: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormatListClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmDateMask: TfrmDateMask;

implementation

{$R *.dfm}
```

```
procedure TfrmDateMask.FormCreate(Sender: TObject);
begin
   lblTest.Caption:=formatdatetime(txtTest.Text,now);
end;

procedure TfrmDateMask.Button1Click(Sender: TObject);
begin
   if txtTest.Text<>'' then
      lblTest.Caption:=formatdatetime(txtTest.Text,now);
end;

procedure TfrmDateMask.FormatListClick(Sender: TObject);
begin
   txtTest.Text:=FormatList.Items[FormatList.itemindex];
end;

end.
```

Editor TFieldsPropery.

Este editor lo utilizaremos para editar la propiedad Fields (TFieldsList) del componente TFieldsType. Como esta propiedad es un StringList, utilizaremos como ancestro de nuestro editor la clase TClassProperty puesto que tendremos que editar una propiedad que es a su vez una clase (TFieldsList).

Al igual que en el editor anterior, decir que la clase TFieldsType permite seleccionar un subconjunto de campos del conjunto total de una tabla. Su definición es :

```
TFieldsType=Class(Tcomponent)
   private
      FTable:tdataset;
      FFields:TFieldsList;
   published
      property Table:tdataset read FTable Write FTable;
      property Fields:tFieldslist read FFields write FFields;
   public
      constructor Create(AOwner: TComponent);override;
      destructor destroy;override;
      procedure SetAllFields;
      function Locate(cad:string;var Index:integer):Boolean;
      procedure Edit;
end;
```

A su vez TFieldsList tiene la siguiente definición :

```
TFieldsList=class(TStringlist)
   public
      function Locate(cad:string;var Index:integer):Boolean;
end;
```

Y pasando al editor de propiedades que vamos a crear, lo primero mostrar como lo vamos a definir y después pasaremos a explicar porque hacemos las cosas como las hacemos :

```
TFieldsProperty = class(TClassProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
end;
```

Como podemos observar la definición es la misma que la del editor anterior, la única diferencia es de que clase heredamos, bueno esa y la implementación que lógicamente será distinta.

```
function TFieldsProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog];
end;
```

Este función es en todo igual a la del editor anterior, así que vale la explicación también para éste.

```
procedure TFieldsProperty.Edit;
var
  FieldsEditor: TfrmFields ;
  i,index:integer;

begin
  FieldsEditor := TfrmFields.Create(Application);
  try
     if TFieldsType(Getcomponent(0)).fields.Count>0 then
     begin
        fieldseditor.L2.Clear;
        fieldseditor.L1.Clear;
        TFieldsType(Getcomponent(0)).table.FieldDefs.update;
        for i:=0 to TFieldsType(Getcomponent(0)).table.FieldDefs.count -1 do
        begin
           if TFieldsType(Getcomponent(0)).fields.Locate(
                                TFieldsType(Getcomponent(0)).table.
                                        FieldDefs[i].Name,index) then
              fieldseditor.L2.Items.Add(
                           TFieldsType(Getcomponent(0)).table.FieldDefs[i].Name)
           else
              fieldseditor.L1.Items.Add(
                           TFieldsType(Getcomponent(0)).table.FieldDefs[i].Name);
        end;
     end
     else
     begin
        TFieldsType(Getcomponent(0)).setallfields;
        fieldseditor.L2.Items.Assign(TFieldsType(Getcomponent(0)).fields);
     end;
     if FieldsEditor.ShowModal=mrok then
     begin
        TFieldsType(Getcomponent(0)).fields.clear;
        TFieldsType(Getcomponent(0)).fields.Assign(fieldseditor.L2.Items);
        designer.Modified;
     end;
  finally
    FieldsEditor.Free;
  end;
end;
```

Los pasos seguidos son los mismos que en el editor de TFormatMaskProperty : Crear una instancia de una caja de diálogo, asignar valores según los valores contenidos en la propiedad, si Ok. asignar los valores adecuados de la caja de diálogo a la propiedad en el inspector de objetos y liberar la instancia. Lo nuevo es el método *GetComponent(0)* definido en TPropertyEditor. Este método nos sirve para acceder al objeto cuya propiedad estamos editando, de esta manera podremos tener acceso a otras propiedades del objeto, en nuestro caso necesitamos saber de que tabla extraer los nombres de los campos. El parámetro que se le pasa indica el índice del objeto puesto que el editor puede manejar varios objetos a la vez.

Puesto que este método nos devuelte un puntero de TPersistent, debemos hacer uso del polimorfismo de los objetos haciendo un 'cast' : *TFieldsType(Getcomponent(0))*, de esta manera ya tenemos acceso a la propiedad 'Table' del objeto : *TFieldsType(Getcomponent(0)).table*.

El resto es programación pura y dura y fácil de entender.

Bueno, pues sólo nos queda registrar el editor y como en el caso anterior tendremos que llamar al procedimiento RegistrerPropertyEditor :

```
RegisterPropertyEditor(TypeInfo(TFieldslist), TFieldsType, '',TFieldsProperty);
```

"Vaya", hemos encontrado otra diferencia con el editor TFormatMaskProperty, a saber, esta vez sí que pasamos el parámetro **ComponentClass**: (TFieldsType), y esto ¿por qué?, pues la razón es sencilla, la propiedad que estamos editando sólo tiene sentido si está en la clase TFieldsType, así que nos curamos en salud si sólo dejamos utilizar el editor para esta propiedad en esta clase.

La definición de TfrmFields utilizado en el procedimiento Edit del editor es la siguiente :

Fields.dfm

```
object frmFields: TfrmFields
  Left = 365
  Top = 229
  Width = 304
  Height = 275
  Caption = 'Campos a exportar'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Left = 3
    Top = 2
    Width = 74
    Height = 13
    Caption = 'Disponibles :'
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
  end
  object Label2: TLabel
    Left = 171
    Top = 2
    Width = 58
    Height = 13
    Caption = 'Actuales :'
    Font.Charset = DEFAULT_CHARSET
```

```
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
  end
  object L1: TListBox
    Left = 0
    Top = 17
    Width = 121
    Height = 185
    ItemHeight = 13
    TabOrder = 0
  end
  object L2: TListBox
    Left = 168
    Top = 17
    Width = 121
    Height = 185
    ItemHeight = 13
    TabOrder = 1
  end
  object B1: TButton
    Left = 128
    Top = 49
    Width = 33
    Height = 25
    Caption = '>>'
    TabOrder = 2
    OnClick = B1Click
  end
  object B2: TButton
    Left = 128
    Top = 81
    Width = 33
    Height = 25
    Caption = '>'
    TabOrder = 3
    OnClick = B2Click
  end
  object B3: TButton
    Left = 128
    Top = 113
    Width = 33
    Height = 25
    Caption = '<'
    TabOrder = 4
    OnClick = B3Click
  end
  object B4: TButton
    Left = 128
    Top = 145
    Width = 33
    Height = 25
    Caption = '<<'
    TabOrder = 5
    OnClick = B4Click
  end
  object btnAceptar: TBitBtn
    Left = 48
    Top = 215
```

```
    Width = 75
    Height = 25
    Caption = '&Aceptar'
    TabOrder = 6
    Kind = bkOK
  end
  object btnCancelar: TBitBtn
    Left = 168
    Top = 215
    Width = 75
    Height = 25
    Caption = '&Cancelar'
    TabOrder = 7
    Kind = bkCancel
  end
end
```

Fields.pas

```
unit fields;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Buttons, StdCtrls;

type
  TfrmFields = class(TForm)
    L1: TListBox;
    L2: TListBox;
    B1: TButton;
    B2: TButton;
    B3: TButton;
    B4: TButton;
    btnAceptar: TBitBtn;
    btnCancelar: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    procedure B1Click(Sender: TObject);
    procedure B4Click(Sender: TObject);
    procedure B2Click(Sender: TObject);
    procedure B3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;


implementation

{$R *.DFM}


procedure TfrmFields.B1Click(Sender: TObject);
var
   i:integer;
begin
   FOR I:=0 TO L1.Items.Count-1 DO
```

```
      l2.Items.Add(l1.Items[i]);
   l1.Items.Clear;
end;

procedure TfrmFields.B4Click(Sender: TObject);
var i:integer;
begin
   FOR I:=0 TO L2.Items.Count-1 DO
      l1.Items.Add(l2.Items[i]);
   l2.Items.clear;
end;

procedure TfrmFields.B2Click(Sender: TObject);
begin
   if l1.ItemIndex>-1 then
   begin
      l2.items.Add(l1.items[l1.itemindex]);
      l1.Items.Delete(l1.itemindex);
   end;
end;

procedure TfrmFields.B3Click(Sender: TObject);
begin
   if l2.ItemIndex>-1 then
   begin
      l1.items.Add(l2.items[l2.itemindex]);
      l2.Items.Delete(l2.itemindex);
   end;
end;

end.
```

Hasta aquí llegó este artículo. Posiblemente en otros abordaremos otros editores, hasta que esto ocurra espero que éste os sirva en vuestras tareas de programación de componentes.