Salvador Jover – mailto:s.jover@wanadoo.es

Objetos Auxiliares VI

Entramos de lleno en la segunda parte del conversor: un pequeño ejemplo que nos permite conocer el uso de propiedades como Names, Values y Objects, en la clase TStrings.

Iniciábamos en el capítulo anterior una primera entrega del ejemplo que hoy vamos a acabar de implementar. Es posible que, después de leído dicho capítulo, todavía te queden algunas dudas sobre la finalidad del mismo, puesto que el editor apenas nos da pistas sobre el objetivo final. Las primeras lineas de "Objetos Auxiliares V" las dedicábamos a contar una pequeña historia que intentaba justificarlo, para que pudiera tener un poco de sentido. Era la historia de nuestro administrativo, al que bautizamos con el nombre de Manolo, y al que colocamos en la tesitura de obtener, rápidamente, los desgloses de los importes en una serie de registros. Tenía que calcular el cambio en moneda, necesario para efectuar los pagos, pudiendo elegir las monedas sobre las que quería obtenerlo.

Para añadir, borrar o editar dichos registros nos valíamos de un pequeño editor, cuyo desarrollo entregábamos en el módulo "Datos.pas". Teníamos parte del camino hecho, y en su recorrido, conseguíamos dar utilidad a las propiedades *Names* y *Values*, que era uno de los motivos de hacerlo.

Dejábamos claro, que el objetivo no era hacer una aplicación en toda la extensión de la palabra, y sin embargo, a medida que se va a desarrollar nuestro objeto conversor, a medida que incrementa su funcionalidad, nos va a facilitar el que al menos lo parezca :-). De hecho, se partía inicialmente de la premisa de que no existía registro alguno al iniciarse la ventana del editor. Ahora nos vemos obligados a considerar la posibilidad de que esto no sea así. El motivo es fácil de entender: vamos a aprovechar las facilidades que nos brinda la clase TStrings para guardar o cargar la información en un fichero. Esto nos va a permitir guardar los datos antes de acabar nuestro programa, o bien recuperarlos en cualquier momento.

Finalmente, y como ya comentábamos, TConversorCambio también nos va a permitir jugar con la propiedad Objects y asociar un registro (en este caso es una estructura de tipo registro -*record*-, pero la metodología seguirá siendo similar si consideramos objetos salvando los temas de reserva de memoria). Cada una de estas estructuras, nos serán necesarias para poder hacer los cálculos de los desgloses. Será el método *AgregarLista( )*, pero eso ya lo veremos un poco más adelante.

```
Type

  EErrorMascaras = Class(Exception);
  EInvalidIndex = Class(Exception);
  EInvalidSeleccion = Class(Exception);

  TValor = Record
     Concepto: String[255];
     Entrada: Double;
  end;

  PEuro = ^TEuro;
```

```
  TMoneda = Integer;

  TMascaras = (UN_CENT, DOS_CENT, CINCO_CENT, DIEZ_CENT, VEINTE_CENT,
               CINCUENTA_CENT, UN_EUROS, DOS_EUROS, CINCO_EUROS, DIEZ_EUROS,
               VEINTE_EUROS, CINCUENTA_EUROS, CIEN_EUROS,DOSCIENTOS_EUROS,
               QUINIENTOS_EUROS);

  TEuro = Record
    VEuros: Double;
    VResto: Double;  //Lo que reste del cambio
    VDesglose: ARRAY [Low(TMascaras)..High(TMascaras)] of TMoneda;
  end;

  TSeleccion = Set of TMascaras;

  TOnAfterChangeMascaraEvent = procedure (Sender: TObject) of Object;
  TOnAfterChangeConversorEvent = procedure (Sender: TObject) of Object;

  TConversorCambio = class(TComponent)

private
            fEuro: TEuro;
   fSeleccion: TSeleccion;
   fLista: TStrings;
   FOnAfterChangeMascaraEvent: TOnAfterChangeMascaraEvent;
   FOnAfterChangeConversorEvent: TOnAfterChangeConversorEvent;
   FCount: Integer;
   function GetName(Index: Integer): String;
   function GetValue(Index: Integer): String;
   procedure SetOnAfterChangeMascaraEvent(const Value: TOnAfterChangeMascaraEvent);
   procedure SetOnAfterChangeConversorEvent(const Value:
          TOnAfterChangeConversorEvent);
   function GetDesgloseCambio(Index: Integer; fMascara: TMascaras): Integer;
   function GetDesgloseTotal(fMascara: TMascaras): Integer;

protected
   procedure Calcular; virtual;
   property Lista: TStrings read fLista;

public
   constructor Create(AOwner: TComponent); Override;
   destructor Destroy; Override;

   procedure Agregar_Mascara(Value:TMascaras);
   procedure Eliminar_Mascara(Value:TMascaras);
   procedure Inicializar_Mascaras;
   function Buscar_Mascara(Value:TMascaras): Boolean;
   function Nominal_Mascara(Value:TMascaras): String; virtual;
   function Contar_Mascaras: Integer;
   function Total_Mascaras: Integer;

   procedure BorrarLista;
   function AgregarLista(const Values: Array of TValor): Boolean;

   procedure SaveToFile(const FileName: String);
   procedure LoadFromFile(const FileName: String);

   function GetTotal: String;
   function GetResto: String;

   function ShowEditor(Position: Integer): Integer; virtual;

   property Values[Index:Integer]: String read GetValue;
   property Names[Index: Integer]: String read GetName;
```

```
    property DesgloseCambio[Index: Integer; fMascara: TMascaras]: Integer read
             GetDesgloseCambio;
    property DesgloseTotal[fMascara: TMascaras]: Integer read GetDesgloseTotal;

    property Count: Integer read FCount; //número de elementos

    property OnAfterChangeMascaraEvent: TOnAfterChangeMascaraEvent read
             FOnAfterChangeMascaraEvent write SetOnAfterChangeMascaraEvent;
    property OnAfterChangeConversorEvent: TOnAfterChangeConversorEvent read
             FOnAfterChangeConversorEvent write SetOnAfterChangeConversorEvent;
  end;
```

Listado 1. Interfaz de TConversorCambio

Metidos en harina...

Lo primero que vamos a hacer es comentar algunos aspectos relevantes del **listado 1,** donde se nos presenta el interfaz que declara la clase TConversorCambio, y que nos pueden ayudar a encontrar sentido al diseño del objeto conversor.

Previo a esto, nos va a ser de gran ayuda, analizar como van a interactuar los elementos principales de nuestra aplicación, sin entrar en detalles muy concretos que nos desvíen de la idea general: Por un lado disponemos de una instancia de TForm que va a representar el interfaz gráfico del usuario. Esta ventana, que a partir de este momento denominaremos Ventana Principal, será la responsable de comunicarse con el objeto conversor, y lo hará invocando correctamente los métodos públicos del mismo. Esta comunicación se establece en los dos sentidos Interfaz-Componente y Componente-Interfaz. Esta claro que nuestro interfaz gráfico puede invocar un método para entregar nuevos valores a nuestro conversor, - lo hemos llamado con muy poco sentido de originalidad AgregarLista( ) - , pero toda vez que hemos actualizado el objeto conversor, se hará necesaria la entrega del resultado de los cálculos y lo que es mas importante, la notificación de que nuestro interfaz debe actualizarse.

En este punto, podemos encontrar sentido a la declaración de los dos eventos en la clase TConversorCambio, responsables de que nuestro interfaz sea coherente con el contenido del objeto conversor. El primero notificará los cambios en la activación/desactivación de una moneda. El segundo hará lo propio cada vez que sea necesario actualizar el interfaz, y será este último, el responsable de invocar los métodos oportunos para leer los nuevos valores resultado de la modificación.

¿Y nuestro editor...? Aquí aparece un punto importante en la toma de decisiones del diseño del componente. Tenemos que decidir quien ha de invocar el editor; si lo ha de hacer la ventana principal o lo ha de hacer el propio objeto conversor. Cualquiera de los dos va a poder hacerlo. Sin lugar a duda, elegimos que lo haga nuestro conversor y para ello, hacemos público el método *ShowEditor( )*, que permitirá a nuestro interfaz crear la ventana de edición de registros. Los motivos son bastante evidentes, aunque en un principio puedan pasar desapercibidos: claridad y sencillez en el código, menor número de referencias entre los distintos módulos. Se gana en claridad. El coste de trasladar la información del interfaz al módulo editor es infinitamente más gravoso que el de hacerlo entre dos objetos descendientes de la clase TStrings, cuando entregamos el contenido en la lista de cadenas del objeto conversor a la lista de cadenas en el Editor. Pero lo vamos a ver con mayor detalle en la siguiente sección.

De la declaración de tipos del listado1, podemos destacar como nos hemos valido de una enumeración (TMascaras), y del conjunto (TSeleccion), como “flags” que nos permitan conocer si una selección de moneda está activa o no. Como siempre, existen distintas formas de hacer lo mismo y tendría similar

función el uso de un array de booleanos. Este tema lo vamos a ver con detenimiento en el apartado de Selección de Máscaras, donde comentaremos los métodos que hace público el conversor, no solo para seleccionar/deseleccionar una moneda, sino también aquellos que se hacen necesarios para saber el total de monedas elegidas, obtener información particular sobre si una de ellas está elegida o no o sobre el mismo rótulo que deba mostrar la unidad monetaria.

Hablábamos en los párrafos anteriores de que se producía una comunicación en el sentido Interfaz-Conversor y que ésta tenía lugar a través del método AgregarLista( ). La elección de hacerlo así y no de un modo natural, dando capacidad de escritura a las propiedades *Names* y *Values* en la clase TConversorCambio, es una decisión de diseño. La implementación del método *AgregarLista( )* hacía más evidente el objetivo final del ejercicio, y que no era otro que ser un ejemplo del uso de la propiedad Objects en la clase TStrings. Parecido razonamiento podemos hacer a que el ascendiente elegido, en este caso TObject, no haya podido ser TComponent, y haber accedido a las capacidades que nos brinda el Inspector de Objetos y trabajar en tiempo de diseño. Para nuestros objetivos no se hacía necesarias tales consideraciones, por lo que queda para el lector, si es que encuentra interés en hacerlo así, modificar la ascendencia del conversor y el registro del componente en la paleta de componentes del IDE de Delphi. Propiedades que se hacen públicas, tales como los eventos, se considerarían a partir de entonces como publicadas. O la adición mismo de una propiedad en la parte publicada del objeto, que represente al fichero, y que en su escritura o lectura, se efectúe mediante la invocación de los métodos SaveToFile( ) o LoadFromFile( ), del objeto conversor.

Otro comentario interesante que podemos hacer, y esta apreciación se la debo a Julio García, es la siguiente:

Vamos a comparar estas dos estructuras

```
TEuro = record
  Veuros: Double;
  VResto: Double;
  Vdesglose: ARRAY[Low(TMascaras)..High(TMascaras) of TMoneda;
end;

TEuro = record
  Veuros: Double;
  VResto: Double;
  T1Centimo, T2Centimos, T5Centimos, T10Centimos,
  T20Centimos, T50Centimos, T1Euro, T2Euros,
  T5Euros, T10Euros, T20Euros, T50Euros, T100Euros,
  T200Euros, T500Euros: TMoneda;
end;
```

Ambas estructuras nos dicen lo mismo. Quizás la segunda resulte en este caso algo escandalosa dado que contiene muchos campos, mientras que la primera es sencilla y clara. Además, y esto es lo más importante, el array hace sencilla la implementación de un bucle que recorra el índice. El ahorro de código es notable.

Por cierto, no he dicho que en el campo *VEuros* será almacenado un decimal que representa el valor del registro a desglosar. *VResto* representa lo que no se puede repartir con el tipo de cambio elegido. Y el array lo desglosado. Esto que ahora comentamos afecta a las estructuras asociadas a la lista que mantiene el objeto conversor. Si hablamos de la estructura de totales el array representará el acumulado total de desgloses y los dos primeros camplo los totales correspondientes a ese concepto.

No obstante, dado que se entrega con el artículo, las fuentes sobre las que estamos trabajando, no estará de más que se ejecute la aplicación y se vea el resultado de la manipulación del interfaz gráfico. Probadlo y volvamos para analizar con más detalle cada uno de los métodos de nuestro conversor.

Creación y Destrucción del Conversor

El constructor de nuestro componente es bien sencillo y en él nos limitamos a inicializar aquellas variables que intervienen. Inicializamos el contador de lineas fCount, inicializamos la selección de monedas fSeleccion, y tras crear la lista de cadenas fLista, hacemos lo propio con la estructura de totales fEuro, cuyos valores son puestos a cero.

constructor TConversorCambio.Create(AOwner: TComponent);

```
var
fMascara: TMascaras;
begin
   inherited Create(AOwner);
   fcount:= 0;
   fSeleccion:= []; // todavía no hay selección de máscara [conjunto = vacio]
   //La clase TStrings nos permitirá mantener la lista de conceptos y una
   //estructura TEuro asociada a cada una de las cadenas.
   fLista:= TStringList.Create;

   with fEuro do // inicializamos la estructura de totales
      begin
      VEuros:= 0.0;
      VResto:= 0.0;
      fMascara:= MENOR_MASCARA;
      while fMascara <= MAYOR_MASCARA do
        begin
        VDesglose[fMascara]:= 0;
        fMascara:= Succ(fMascara);
        end;
      end;
end;
```

En el destructor nos limitamos a destruir la lista de cadenas fLista, y tras ésta, la invocación al destructor heredado para que destruya la parte de objeto que no hemos creado nosotros.

```
destructor TConversorCambio.Destroy;
begin
   BorrarLista; //liberamos todas las estructuras
   fLista.Free;//destruimmos la lista
   inherited Destroy;
end;
```

Selección de Máscaras (Agregar, Eliminar, Buscar, Contar, etc...)

El método Agregar_Mascara, recibe como parámetro una referencia por valor cuyo tipo es TMascaras, y que representa la unidad monetaria que vamos a seleccionar.

```
procedure TConversorCambio.Agregar_Mascara(Value:TMascaras);
begin
   if not (Value in fSeleccion) then
       begin
       fSeleccion:= fSeleccion + [Value];
       if Assigned(OnAfterChangeMascaraEvent) then OnAfterChangeMascaraEvent(Self);
       end;
end;
```

Es un método tan sencillo como breve. Si no está la unidad monetaria que recibe como parámetro dentro de las elegidas, las que permanecen activas, procederemos a activarla mediante el uso habitual de los operadores aplicables a las variables de tipo conjunto (Set), que son el operador '+' y el operador '-'. Aquí

podemos matizar la posibilidad de hacerlo también mediante el uso de Include( ), Exclude( ), con la misma funcionalidad. Hecho esto, y añadido el nuevo valor al conjunto fSeleccion, que es la variable global que almacena las máscaras elegidas, (las unidades monetarias selecionadas), será invocado el Evento que nos indica que el conjunto de selección de monedas ha cambiado, dando a nuestro interfaz la oportunidad de hacer cuantas acciones crea convenientes en la implementación del evento.

Eliminar la selección de una unidad monetaria es similar al observado en la linea anterior.

```
procedure TConversorCambio.Eliminar_Mascara(Value:TMascaras);
begin
   if (Value in fSeleccion) then
      begin
      fSeleccion:= fSeleccion - [Value];
      if Assigned(OnAfterChangeMascaraEvent) then OnAfterChangeMascaraEvent(Self);
      end;
end;
```

En este caso, con toda lógica, solo tendrá lugar la eliminación si dicha selección está activa en el conjunto fSeleccion, dando paso si es así a su eliminación y a la invocación del evento, en las mismas condiciones que las señaladas más arriba.

Saber si una máscara, una moneda ha sido elegida es tan sencillo que nos basta una sola linea para obtener un valor booleano de verdadero o falso. El operador **in** nos devolverá verdadero si Value, la máscara que pasamos como parámetro, pertenece al conjunto fSeleccion.

```
function TConversorCambio.Buscar_Mascara(Value:TMascaras): Boolean;
begin
   Result:= (Value in fSeleccion);//¿esta activa dicha máscara
end;
```

Y aunque el método Contar_Mascaras, añada algo más de código, no será menos evidente. Declaramos una variable del tipo TMascaras y nos posicionamos en el menor de los posibles valores que puede adoptar. Y recorremos todos los valores de la enumeración mediante un bucle While...do, siendo la evaluación al valor True del método BuscarMascara( ) el que nos permita al final del mismo, obtener el total de máscaras activas.

```
function TConversorCambio.Contar_Mascaras: Integer;
   var
   xMascara: TMascaras;
begin
   xMascara:= MENOR_MASCARA; //nos situamos en la primera de las máscaras
   Result:= 0;
   while xMascara <= MAYOR_MASCARA do //y las recorremos todas
      begin //si está activa
      if Buscar_Mascara(xMascara) then Result:= Result + 1; //la contamos
      xMascara:= Succ(xMascara); //siguiente máscara
      end;
end;
```

Por otro lado, y ya para acabar con esta sección, se nos hace necesaria una función que nos permita nombrar cada una de las unidades elegidas. Hacerlo de esta forma y no ligarnos a Constantes, nos permitirá en un futuro, sobreescribir este pequeño método en un descendiente, para así modificar la rotulación que se puede hacer necesaria en la cabecera de listados o tablas.

```
function TConversorCambio.Nominal_Mascara(Value:TMascaras): String;
begin
   Result:= '';
   case Value of
      UN_CENT:             Result:='1 Ctms.';
       ...
       ...
       ...
```

```
        QUINIENTOS_EUROS:    Result:= '500 Euros.';
    end;
end;
```

Y ya para finalizar el apartado, una pequeña reflexión que me parece muy oportuna y en la que intentaré razonar un pequeño error, con la intención de que pueda ser advertido. Parece razonable apoyarse en el conocimiento que se tiene del flujo de la aplicación, para construir a la medida que se nos haga necesaria los métodos del componente sobre el que estamos trabajando. Sin embargo, esta postura tan práctica en ocasiones nos puede hacer implementar algunos métodos que aun siendo libres de generar errores en ese contexto, implican estados incoherentes en el mismo objeto. Vamos a poner un ejemplo: Inicializar_Mascaras.

```
procedure TConversorCambio.Inicializar_Mascaras;
begin
     fSeleccion:= [];
end;
```

Conscientes de que no podemos acceder desde el exterior, desde la Ventana Principal, a la variable fSeleccion que nos indica el conjunto de monedas activas, puesto que fSeleccion es una variable privada del objeto, podemos tener la tentación de implementar un método tal cual el anterior, que nos permita eliminar de golpe el total de selecciones hechas, facilitando los estados de inicialización. Podéis observar cómo es invocada indirectamente esta rutina a través del método Nuevo1Click( ) con la rutina EstadoInicial.

¿Cual es entonces el error? El error está en lo que no hace el método y que es, inicializar las estruturas que almacenan los totales *fEuro* y hacer lo propio con los objetos asociados a cada una de las cadenas de *fLista*. Así pues, al ser un método público del Conversor, podría ser llamado en cualquier parte del programa, provocando que se deseleccionaran el total de monedas activas sin que afectara al estado del interfaz, que seguiría reflejando los valores de las estructuras utilizadas para el cálculo. Es peligroso hacer público un método que sea “correcto” cuando su invocación se haga en determinadas condiciones y no en otras, quedando condicionados a tener la suficiente memoria para recordar en que situaciones debemos emplearlos y en cuales no.

Borrar una lista del Conversor.

Entramos de lleno en el uso de la propiedad Objects en la clase TStrings. Cuando deseemos borrar una lista de registros del Conversor, y esto deberá hacerse antes de agregar los nuevos valores, deberemos recorrer cada una de las cadenas de la variable Lista, del tipo TStringList, y liberar la memoria reservada a la estructura,

```
  TEuro = record
    VEuros: Double;
    VResto: Double;
    VDesglose: ARRAY [Low(TMascaras)..High(TMascaras)] of TMoneda;
  end;
```

Observemos el procedimiento:

```
procedure TConversorCambio.BorrarLista;
   var
   xIndice: Integer;
   Nodo: PEuro;
begin
   for xIndice:= 0 to Lista.Count-1 do
      begin
      Nodo:= PEuro(Lista.Objects[xIndice]);
      Dispose(Nodo);
      Lista.Objects[xIndice]:= Nil;
      end;
```

```
      Lista.Clear;
      fCount:= 0;
   end;
```

Recorremos todos los elementos de la lista. Y para cada uno de ellos, es liberada la memoria que reservamos en el momento de la asociación de la estructura al elemento de la lista, desligándonos finalmente del objeto mediante la asignación a **Nil.**

Hecho esto, nos basta invocar el método *Clear* y poner a cero el contador de elementos del objeto conversor.

Agregar una lista y calcular los resultados...

Comentábamos anteriormente que nuestro diseño, estaba condicionado por la decisión de implementar un metodo que permitiera agregar una lista de valores en nuestro objeto TConversorCambio, en lugar de permitir la modificación individual de cada uno de los elementos de la lista y objetos asociados. Habíamos declarado como de solo lectura la propiedad *Lista.*

Vamos a analizar paso a paso el método *AgregarLista( )*, que recibirá como parámetro un array abierto del tipo TValor, permitiéndonos recibir una cantidad indeterminada de valores que en el momento de su implementación no nos es conocido.

Como valor de retorno devolvemos un booleano que nos indica si la operación se ha llevado con éxito o no.

Pero recordemos primero que encierra el tipo TValor:

```
  TValor = Record
     Concepto: String[255];
     Entrada: Double;
  end;
```

TValor es un estructura de tipo Registro cuyos campos son un String y un Decimal. Es decir, que va a recibir en el campo Concepto la cadena de caracteres que nos indicaba el motivo asociado a cada valor. Y en el campo Entrada, el valor decimal a analizar. Pongamos un ejemplo:

```
Concepto-----------> “Fra. 22 de Electricidad Manresa, S.L.”
Entrada-------------> 425,38
```

Así pues, en dicho array abierto, se nos va a entregar desde el interfaz del usuario, desde la ventana principal de la aplicación, la lista de elementos que ha de contener el Conversor, y será en este procedimiento, y concretamente en el método *Calcular*, en donde haremos todos los desgloses correspondientes a cada uno de los valores.

```
begin
   Result:= False;
   BorrarLista; //inicializamos la lista

   try
    for xIndice:=Low(Values) to High(Values) do
      begin
      s:= TValor(Values[xIndice]).Concepto;
      f:= TValor(Values[xIndice]).Entrada;

      New(DataEuro);
      with DataEuro^do
         begin
         VEuros:= f;
         VResto:= 0.0;
         fMascara:= MENOR_MASCARA;
```

```
            while fMascara <= MAYOR_MASCARA do
               begin
               VDesglose[fMascara]:= 0;
               fMascara:= Succ(fMascara);
               end;
            end;

         Lista.AddObject(s, Tobject(Dataeuro));
         Inc(fCount);
         end;
      Calcular; //ya podemos efectuar las operaciones necesarias
      Result:= True; //devolvemos la estructura de totales
      Except
      on E:Exception do
        if Assigned(OnAfterChangeConversorEvent) then OnAfterChangeConversorEvent(Self);
      end;
   end;
```

Tras poner el resultado de la función a *false*, iniciamos la misma con la invocación del método *BorrarLista*, que como se ha comentado en el apartado anterior nos va a permitir inicializar la lista de cadenas en la que almacena el conversor cada uno de los registros, liberando la memoria de las estructuras que ha utilizado. Hecho esto, estamos preparados para recibir los valores de los campos *Concepto* y *Entrada* del array abierto, con el recorrido de un bucle *For..do* que recorra todo el array. En dicho bucle, y para cada uno de registros añadidos a la lista de cadenas, reservamos memoria para una nueva estructura de desgloses, inicializada a 0, y que será asociada mediante la propiedad Objects. De esa forma, un ítem de la lista, no solo tiene el concepto recibido sino una estructura donde ha almacenado el valor y donde, hechos los calculos necesarios, almacenará los desgloses correspondientes a la selección de monedas activas.

Y lógicamente, para cada entrada añadida al conversor, es incrementado el contador de registros.

Hecho esto, invocamos el método *Calcular*, encapsulando en él toda la operatoria necesaria para efectuar los desgloses. Con éxito o sin el, nos tenemos que asegurar que el interfaz gráfico sea notificado de los nuevos valores en curso.

Podemos analizar igualmente que es lo que ocurre cuando internamente se produce la llamada al procedimiento *Calcular*:

El primer paso lo empleamos para asegurarnos que la lista contiene elementos, pues en caso contrario, no tiene sentido realizar ningun calculo, por lo que, tras invocar el evento, salimos del procedimiento.

```
   //nos aseguramos que la lista contiene algún valor
   if Lista.Count <= 0 then
     begin  //tenemos que dejar que el usuario actualice el interfaz
     if Assigned(OnAfterChangeConversorEvent) then OnAfterChangeConversorEvent(Self);
     exit;
     end;
```

A continuación, previo a ningun calculo, vamos a proceder a inicializar la estructura de totales. Seguimos la misma mecánica que vimos en el método *AgregarLista* para cada una de los registros asociados a los elementos de las lista de cadenas.

```
   //inicializamos la estructura de totales
   with fEuro do
      begin
      VEuros:= 0.0;
      VResto:= 0.0;
      fMascara:= MENOR_MASCARA;
      while fMascara <= MAYOR_MASCARA do
         begin
         fEuro.VDesglose[fMascara]:= 0;
```

```
            fMascara:= Succ(fMascara);
            end;
         end;
```

A partir de aquí, el algoritmo para el cálculo de los desgloses. Básicamente el algoritmo hace lo siguiente:

Para cada elemento de la lista, recorre todas las máscaras activas empezando por la mayor hasta la menor de ellas. Si está activa la máscara, deduce del valor total la parte unitaría a esa moneda, acumulando los totales en la estructura fEuro. El resultado al finalizar el bucle que recorre toda los elementos de la lista es que las estructuras asociadas que almacenan el desglose de cada valor pueden ser consultadas mediante métodos y propiedades que hace públicas el conversor, tales como la propiedad *DesgloseCambio[]* o el método *GetResto*.

Aquí empieza el bucle que recorrerá todos los registros del conversor:

```
      //para cada una de las cadenas de la lista (y sus objetos asociados)
      for xIndice:= 0 to Lista.Count-1 do
         begin
         //obtenemos el valor sobre el que haremos el desglose
         Valor_e:= Trunc(PEuro(Lista.Objects[xIndice])^.VEuros);
            Valor_c:=  StrToInt(FloatToStr(Round((PEuro(Lista.Objects[xIndice])^.VEuros -
Valor_e)*100)));
         fEuro.VEuros:= fEuro.VEuros + PEuro(Lista.Objects[xIndice])^.VEuros;
        //y acumulamos el total de euros
```

Toda vez que estamos situados sobre un registro determinado y ya tenemos el acumulado en el campo *fEuro.VEuros*, y las variables *Valor_e* y *Valor_c* sobre las que se operará temporalmente con el nuevo importe, se inicia el bucle que analizará para esos valores como se ha de repartir el cambio.

```
        zIndice:= Integer(TMascaras(MAYOR_MASCARA));
        While zIndice >= 0 DO
             begin
             fMascara:= TMascaras(zIndice);
             //empezamos a filtrar las condiciones
             if Buscar_Mascara(fMascara) then
                  begin
                  if fMascara >= UN_EUROS then
                       begin
                       While Valor_e >= VUnidad[fMascara] do
                            begin
                            Inc(PEuro(Lista.Objects[xIndice])^.VDesglose[fMascara]);
                            Valor_e:= Valor_e - VUnidad[fMascara];
                            end;
                       end
                  else
                       begin
                       if Valor_e > 0 then
                            begin
                            Valor_c:= Valor_c + Valor_e * 100;
                            Valor_e:= 0;
                            end;
                       While Valor_c >= VUnidad[fMascara] do
                            begin
                            Inc(PEuro(Lista.Objects[xIndice])^.VDesglose[fMascara]);
                            Valor_c:= Valor_c - VUnidad[fMascara];
                            end;
                       end;
                  //acumulamos el total
                  fEuro.VDesglose[fMascara]:= fEuro.VDesglose[fMascara] +
                                 PEuro(Lista.Objects[xIndice])^.VDesglose[fMascara];
                  end;
             Dec(zIndice);
```

```
            end;//endbegin while


          if Valor_e > 0 then
             fEuro.VResto:= fEuro.VResto + StrToFloat(IntToStr(Valor_e));
          if Valor_c > 0 then fEuro.VResto:= fEuro.VResto + (Valor_c / 100);
          //hemos acumulado el valor de lo que no ha sido desglosado en la estructura
          //de totales
          end;
```

Concluido con éxito el algoritmo, procedemos a invocar el evento que permite actualizar el interfaz gráfico.

```
    if Assigned(OnAfterChangeConversorEvent) then OnAfterChangeConversorEvent(Self);
end;
```

Abrir y Guardar un fichero en nuestro Conversor.

Si queremos que los resultados que almacena nuestro Conversor no desaparezcan al cerrar nuestra aplicación, nos veremos obligados a crear los métodos adecuados para salvar el contenido a un fichero. Nos basta para ello, dotar a nuestro objeto de un métodos *LoadFromFile* y *SaveToFile*, nombrados así por seguir la linea que se inició desde la VCL.

Eso sí, queremos aprovechar todo lo que nos pueda ofrecer la clase TStrings, y en este caso concreto, prácticamente nos va a solucionar aquellos detalles propios del almacenamiento o la apertura del fichero.

Veamos como podemos guardar el contenido del conversor en un fichero:

```
procedure TConversorCambio.SaveToFile(const FileName: String);
var
StringTemp: String;
ListaTemp: TStrings;
S: Array [0..MAX_MASCARAS-1] of Char;
xIndice: Integer;
begin
```

Creamos una lista temporal. El motivo de crearla es porque dicho fichero almacenará en su primera linea la situación [activa/no activa] de las máscaras mediante ceros (0) y unos (1). Hecho esto, resta asignar a dicha lista temporal el contenido de la lista de nuestro conversor. Y ya en ese punto, generar la igualdad añadiendo a la cadena de texto el carácter '=' seguido del valor decimal asociado a la cadena. No nos hace falta nada más para reconstruir posteriormente la situación actual.

```
    ListaTemp:= TStringList.Create; //nos valemos de una lista temporal
    try
       //vamos a exigirle al conversor que al menos exista una seleccion de 2 máscaras
       if Contar_Mascaras < 2 then
           Raise EInvalidSeleccion.Create('Seleccion Máscaras debe ser mayor de dos');
       //tomaremos todo el contenido del miembro lista del conversor
       //incluidos los objetos. Estos nos serviran para recuperar los
       //valores asociados a cada lista
       ListaTemp.Assign(Lista);
       for xIndice:= 0 to MAX_MASCARAS-1 do //recorremos todas las máscaras
         if Buscar_Mascara(TMascaras(xIndice)) then S[xIndice]:= '1'
         else S[xIndice]:= '0';
       StringTemp:= 'MASCARA='+String(s); //ya tenemos la primera línea
       ListaTemp.Insert(0,StringTemp); //la instamos en primer lugar
       //Y recorremos el resto de lista para construir el resto de igualdades
       for xIndice:= 1 to ListaTemp.Count - 1 do
```

```
         if ListaTemp.Strings[xIndice] <> '' then
            ListaTemp.Strings[xIndice]:= ListaTemp.Strings[xIndice] +  '=' +
               FormatFloat('#0.00', PEuro(ListaTemp.Objects[xIndice])^.VEuros);
      ListaTemp.SaveToFile(Filename);//guardamos el fichero
   finally
      ListaTemp.Free;//y liberamos la lista temporal
   end;
end;
```

Reconstruir el estado del objeto conversor a partir de la carga del archivo, teniendo en cuenta que somos capaces, (la clase TStrings incorpora dicha capacidad), de leer cadenas donde existe la igualdad '=', es algo que resultará trivial. Podemos leer la primera linea y recuperar el estado de selección de cada una de las monedas. Y podemos, tal y como hicimos en el método *AgregarLista( )*, generar los objetos asociados a cada una de las cadenas en la lista temporal, y añadirlos finalmente mediante el método AddObject( ) que implementa la clase TStrings. Tal que así:

```
Lista.AddObject(ListaTemp.Names[xIndice], Tobject(Dataeuro));
```

```
procedure TConversorCambio.LoadFromFile(const FileName: String);
var
  ListaTemp: TStrings;
  S: String [MAX_MASCARAS];
  xIndice: Integer;
  DataEuro: PEuro;
  fMascara: TMascaras;
begin
   ListaTemp:= TStringList.Create; //necesitamos una lista temporal
                                   //para leer las igualdades
   try
```

Nos valemos de un lista temporal en la que cargamos el contenido del archivo que vamos a abrir. ¿Hay algo más sencillo?

```
ListaTemp.LoadFromFile(Filename);//cargamos el fichero en la lista temporal
//leemos el contenido del valor de la primera linea
//que no es otro que la selección de máscaras almacenada en el fichero
// Ejemplo: MASCARA=10010001110101
// Un valor 1 nos indica que está activada y 0 que esta desactivada
```

Vamos a recuperar la máscara de ceros y unos que nos indica cuales monedas están activas. Así que nos vamos a la primera de las lineas de la lista. ¡¡¡Aquí también nos podemos valer de las propiedades Names y Values para hacerlo!!!

```
S:=ListaTemp.Values[ListaTemp.Names[0]];
fSeleccion:= []; //inicializamos la seleccion de máscaras
//recorremos el string comprobando si esta activada o no
```

Un bucle para incorporar al conjunto fSeleccion las máscaras correspondientes...

```
for xIndice:= 0 to MAX_MASCARAS - 1 do
   if S[xIndice+1] = '1' then
      fSeleccion:= fSeleccion + [TMascaras(xIndice)] //activamos la máscara
   else  //si no está activada la desactivamos
      if S[xIndice+1] = '0' then fSeleccion:= fSeleccion - [TMascaras(xIndice)]
      else Raise EErrorMascaras.Create('Error: formato no valido de archivo');
BorrarLista;//ya podemos borrar el contenido de la lista de conversor
//disponemos a inicializar la lista temporal con los valores recogidos
//del fichero, creando la estructura de objetos
```

Y otro bucle para inicializar las nuevas estructuras y recoger los valores decimales sobre los que operaremos.

```
      for xIndice:= 1 to ListaTemp.Count - 1 do
         begin
         New(DataEuro);  //reservamos memoria
         with DataEuro^do
               begin                    {y es guardado finalmente en el registro}
               VEuros:= StrToFloat(ListaTemp.Values[ListaTemp.Names[xIndice]]);
               VResto:= 0.0;
               fMascara:= MENOR_MASCARA;
               while fMascara <= MAYOR_MASCARA do
                     begin
                     VDesglose[fMascara]:= 0;
                     fMascara:= Succ(fMascara);
                     end;
               end;
```

Vale. Ahora ya entregamos el resultado a la lista del objeto conversor

```
         Lista.AddObject(ListaTemp.Names[xIndice], Tobject(Dataeuro));
```

E incrementamos el contador.

```
         Inc(fCount); {incrementamos el contador de lineas}
         end;
   finally
```

En cualquier caso, debemos asegurarnos de que se finalmente se libera la lista temporal. De la misma forma procedemos a invocar el método para que se recalculen los desgloses. Ahora ya no hay necesidad el método *AgregarLista( )* porque el objeto conversor, en este punto, ya esta actualizado. Es una de las ventajas que nos da encapsular la operatoria en un procedimiento.

```
      ListaTemp.Free;
      Calcular; //nos permitira crear la estructura de objetos y de totales
   end;
end;
```

Editando los datos

Para acabar, interesa comentar un pequeño detalle referente al Editor, expuesto en el capítulo anterior. Hemos sobreescrito el constructor de la ventana frmDatos, donde se implementa el editor al que hacemos referencia, para que pueda construirse de acuerdo a dos condiciones. Por un lado es necesario condicionar *(count > 0)*, tal y como observaréis en el código fuente, para discriminar la inicialización de variables en el evento de creación del formulario a dos situaciones distintas, que nuestro editor no contenga inicialmente ningun registro o que por el contrario sí que los deba contener, en cuyo caso, también necesitará en cual de ellos de posicionarse. Este último dato se lo entregamos mediante el parámetro *Posicion*.

```
constructor TfrmDatos.Create(AOwner: TComponent; carga: boolean; Posicion: Integer);
begin
inherited Create(AOwner);
fHazCarga:= carga;
Itemindex:= Posicion;
fConversor:= (AOwner as TConversorCambio);
end;
```

No son necesarios demasiados comentarios a las siguientes lineas en las que se crea de forma dinámica la ventana modal que representa el editor y que, antes de su destrucción, nos permite saber la posición del registro actual sobre el que nos posicionaremos al recobrar el foco en la ventana principal de la aplicación.

```
function TConversorCambio.ShowEditor(Position: Integer): Integer;
var
```

```
Dat: TfrmDatos;
begin
     Dat:= TfrmDatos.Create(Self, (count > 0), Position);
     try
          if Dat.ShowModal = mrOk then BorrarLista;
          Result:= Dat.stg_Tabla.Row;
     finally
     Dat.Free;
     end;
end;
```

La despedida...

Han quedado algunas cosas en el tintero pero creo que es el punto ideal para dejar ya este pequeño ejemplo que hemos compartido. Y pienso que los objetivos que marcamos al iniciarlo se han cubierto con creces. :-)

Ya... se que lo podemos mejorar. Ahora mismo, mientras escribo estas palabras de despedida me gustaría iniciarlo de nuevo y eliminar todo aquello que no me gusta, lo que me parece que puede ser enfocado de otra forma más sencilla. De hecho, tengo que agradecer a Julio García y a Mario Rodríguez el que me dieran algunos consejos para hacer más coherente el diseño. Seguro que mi torpeza no ha hecho honor a sus comentarios.

Así que nos despedimos hasta el siguiente número de Síntesis en el que veremos el último de los capítulos de la serie. Por fin... ea. ;-)