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

Objetos Auxiliares V

¿Quién dijo que este era el último capítulo de la serie...?. ¿Seguro que fui yo...? :-)

En el capítulo anterior, nos despedíamos con la promesa de que éste, que ahora inicias la lectura, fuera un artículo realmente práctico. Después de los cuatro primeros, en los que hemos intentado escudriñar, con la mayor profundidad posible, las fuentes de la VCL, y en concreto las clases que Borland mismo engloba como clases auxiliares, era hora ya de intentar resumir en un ejemplo muy básico, algunos de los aspectos importantes sobre los que ha girado la serie. Y por cierto, hoy tampoco vamos a ver las dos clases que nos faltan: TStack y TQueue. Ni siquiera, pienso, que puedan ser abordables en el próximo :-) De haberme conformado con la escritura de una veintena de lineas de código, suficiente para saber como es aplicado un método determinado, quedaría zanjada nuestra pequeña aventura, puesto que las clases que nos faltan por comentar, y que responden a los TDAs (Tipo de Datos Abstractos) Pila y Cola, son realmente sencillas, toda vez que nos hemos introducido anteriormente en la clase TList.

He intentado llegar un poco más lejos con el ejemplo que vamos a comentar. La idea que mueve, o digámoslo de otra manera, la moraleja que se esconde tras estas pequeñas lineas que compartimos, es que podemos apoyarnos a menudo en estas clases más de lo que lo hacemos. Pueden formar parte de aquellos objetos que construyamos, extendiendo la funcionalidad de los mismos. Quiero decir con todo esto, que al finalizar esta serie, el ejemplo ya no es importante si no sales realmente convencido de que puedes sacar partido a todos estos objetos que hemos ido viendo en cada uno de los artículos. El ejemplo en si no es nada. Obviaremos detalles que en un desarrollo real deberíamos tener en cuenta, tales como el uso de librerías dinámicas, (sin ir más lejos, las imágenes que contienen algunos de los botones forman parte de nuestro ejecutable, o las propias cadena literales que rotulan cada uno de los botones), tratamiento más serio de la captura o lanzamiento de excepciones y muchos más aspectos que nos alejarían de lo que realmente se pretende con la implementación.

Dicho todo esto, creo que vale la pena que nos metamos en harina, como se suele decir en mi pueblo... o más aun: que nos dejemos de gaitas y vayamos al grano...

La historia...

Es una verdadera tortura plantear un artículo como éste, sin un buen ejemplo. Finalicé el anterior y, tras unos días de descanso, inicié la planificación del que ahora nos encontramos. No se me ocurría nada realmente interesante, que me pudiera motivar. Bueno sí, pequeños ejemplos como los que compartimos en el segundo de los capítulos, a raíz de los métodos Sort y Assign en la clase TList. Y fueron pasando los días. Y por fin, en esta última semana, con la fecha de entrega “pisándome los talones”, surgió un pequeño rayo de luz (pequeñito).

En España, en estos momentos, andamos de cabeza con el dichoso Euro. Digo esto porque sé que la revista es leída por un amplio sector de personas fuera de España, y más concretamente de Hispano-América y quizás se sientan algo ajenos al problema. El caso es, que también existen empresas pequeñas... No todas las empresas son grandes y robustas y también las hay menudas, que mantienen contra viento y marea sistemas tradicionales de pago, como lo es y ha sido el de efectivo.

En esas premisas, desafiando a aquellas que han mecanizado sus sistemas de pago a través de medios informáticos, perdura un tipo de empresario que paga semanalmente y al contado. Es una raza a extinguir. Sus oficinas suelen tener un ritmo endiablado en los días de pago: el teléfono no para de sonar, hay que subir al banco a recoger el efectivo (cambio) necesario para efectuar los pagos de la semana, los salarios de los trabajadores, los proveedores... todos quieren cobrar.

El administrador espera de pies, mirando de reojo a un administrativo que suda la gota gorda haciendo cálculos y cálculos... para acabar diciendo: -Pues jefe... hace falta... (y replica el jefe)

-¡¡Mira Manolo, que la última vez te quedaste corto... !!

(dejémoslos discutiendo, ahora que ya sabéis un poco de la historia)

...Y los hechos.

Vamos a intentar ayudar un poco a nuestro acalorado administrativo y crearemos una pequeña utilidad que le haga mas llevadera la vida, facilitándole el proceso de cálculo del cambio necesario para los pagos. Vamos a crear un pequeño Conversor de Cambio.

El proceso de diseño de nuestra "Utilidad" es quizá lo más importante de todo el proyecto y a lo que realmente debemos dedicar la mayor parte del tiempo invertido. Quiero decir que debemos sentarnos y emborronar cuantas hojas hagan falta hasta saber que es lo que queremos exactamente.

Analicemos:

La unidad monetaria actual que ha adoptado la Comunidad Europea es el Euro, como bien es sabido. Lo que quizás no sea tan conocido desde otras comunidades, es que con su implantación , los europeos dispondrán de un total de 8 monedas y 7 billetes distintos para hacer efectivos sus pagos. A saber: tenemos billetes de 500, 200, 100, 50, 20, 10, y 5 Euros, y monedas de 2, 1 Euro y 50, 20, 10, 5, 2, y 1 Céntimo. (Manolo, nuestro acalorado administrativo continua con sus cálculos... y su jefe de pie, mirándole con cara de muy pocos amigos.)

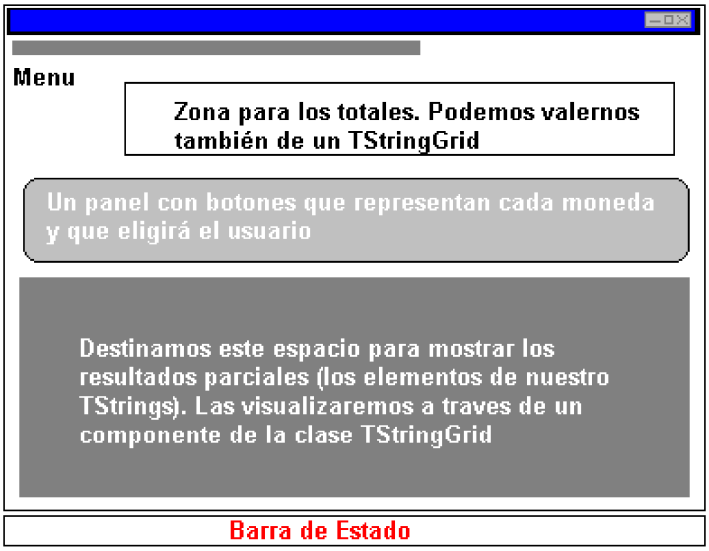

Figura 1: Borrador que representa el interfaz de la utilidad de conversión

Sabido este detalle vital (15 opciones que van a poder ser elegidas, simultáneas), nos interesa tener una lista en el que se detallen por un lado el concepto al que corresponde y por otro lado el valor sobre el que hay que calcular el desglose. Un ejemplo: "Pago al Electricista.....100´72 Euros". Aquí podemos dar entrada a nuestra clase TStrings, que será la que almacene el CONCEPTO en cada una de estas cadenas.

Elegido un tipo de desglose, común a todos los pagos, nos limitaremos a recorrer la lista y a calcularlo individualmente, obteniendo al final de la misma, mediante un sistema de acumulados, los totales necesarios correspondientes a cada una de las monedas. Y aquí podemos introducir otra de las propiedades vistas y que dejamos al estudio de este ejemplo: la matriz Objects, que nos va a permitir asociar a cada cadena una estructura de tipo Record (TEuro), que contiene:

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

En realidad, y siguiendo las recomendaciones habituales de los libros sobre POO, Mario Rodríguez (otro componente del grupo albor que de sobra conocéis) me comentaba con mucha razón, que podríamos abordar el problema mediante la creación de clases nuevas que respondan a cada una de las abstracciones. La creación de una clase tiene mayores ventajas sobre el uso de registros. Eso está bastante claro. Un objeto, es capaz de relacionarse con su entorno, con otros objetos a través de sus métodos. Un registro no. Una estructura del tipo Record, puede almacenar la información que necesitamos pero nada hay que nos garantice que aquello que contiene sea coherente. Un ejemplo: Imaginemos un registro que nos permitía almacenar una fecha. ¿Existe algo que nos garantice que no pueda tomar valores tales como el 31 de Febrero?. Sin embargo, para facilitar el desarrollo de este pequeño ejemplo que vamos a ver, elegiremos el registro, reservando dinámicamente la memoria necesaria, y liberándolo finalmente, cuando ya no sea necesario.

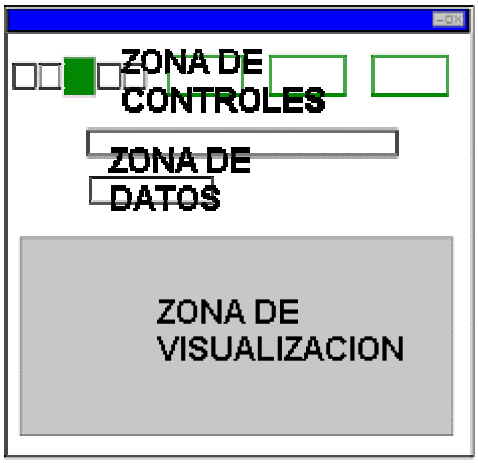

Figura 2: Borrador que representa el interfaz de nuestro pequeño editor

Así pues, ya nos podemos hacer una idea del interfaz gráfico, y lo perfilaremos a grandes rasgos, como se puede ver en la **figura 1**. Esa será la ventana principal de nuestra ” utilidad ”, que no tiene porque coincidir, por obligación, con la ventana principal de nuestra aplicación. Podemos imaginar por ejemplo, que sea una mas de las opciones dentro de otra aplicación de gestión.

Por otro lado, vamos a necesitar un método de introducción de datos: podemos servirnos del componente TStringGrid del que hablábamos en la figura 1, que hubiera sido quizás lo más práctico, o lo más intuitivo. No lo vamos a hacer así. Crearemos un pequeño editor. Esto nos va a permitir ver con más detenimiento las posibilidades de dos de las propiedades comentadas en el artículo anterior: *Names* y *Values*. Es decir, implementaremos los controles necesarios para navegar entre los registros y editar, insertar o modificar respectivamente, tal y como lo hiciéramos desde el TDBNavigator (eso si, más casero).

Esbozamos una pequeña ilustración tal cual la **figura 2**.

Vamos a comenzar nuestro trabajo con este pequeño editor.

Un pequeño editor para nuestro Conversor.

Nuestro editor, al que hemos llamado (TfrmDatos), es una ventana que será invocada al pulsar con un doble click sobre una de las filas (registros) del componente TStringGrid de la figura 1. El que está en la

parte inferior. Nos olvidaremos de momento del hecho, de que, a su creación, debería obtener todas las cadenas del Conversor y de que, previo a su destrucción, debería también actualizar el Conversor con la nueva lista de cadenas para que éste realice los cálculos oportunos.

De hecho, ahora partiremos de la premisa de que nuestro editor contiene al momento de su invocación Cero (0) elementos y de que, a su destrucción, no hace nada. Hemos reducido, por decirlo de alguna forma, la dimensión de nuestro problema original, para hacerlo un poco más asequible. Se limitará a insertar, editar o borrar elementos de la lista de cadenas, y podrá navegar al primero, anterior, posterior y último de los registros.

Necesitamos un objeto que se responsabilice de almacenar cada una de las cadenas del editor. Para ello, nos apoyamos en un objeto descendiente de la clase TStrings como lo es *Listatemp* (TStringList). En este caso, podemos hacer uso de la funcionalidad que nos brindan las propiedades Names y Values, y construir cadenas del tipo 'CONCEPTO=VALOR', que podrán ser leídas por nuestro objeto, permitiéndonos obtener respectivamente, el concepto o el valor, según sean necesario lo uno o lo otro.

```
function TfrmDatos.Situarse(Index1,Index2: Integer):String;
begin
   if index2 > 0 then
      begin
      case index1 of
      0:   Result:= listatemp.Names[index2-1];
      1:   Result:= listatemp.Values[listatemp.Names[index2-1]];
      end;
      end
   else
      Result:= '';
end;
```

Queda claro, ¿no?. Imaginemos una cadena de caracteres tal que: 'Fra. de Moisés=43,56'. Invocada la función Situarse( ), y pasado como parámetro de la misma Index1 = 0, el selector CASE producirá que el valor de retorno sea una cadena que contenga: 'Fra. de Moisés'. Si invocamos la función, siendo 1 el valor del parámetro Index1, ha de devolver la parte derecha de la igualdad. Index2 será nuestro selector de registro.

Es decir, si nos vamos a valer de la clase TStrings para almacenar cadenas con el formato CONCEPTO=VALOR, necesitaremos mantener un indice que nos recuerde en que posición del array nos encontramos. Ese índice lo vamos a llamar *itemindex*, un poco por analogía con el indice existente en la propiedad *items* de componentes como TListBox.

¿Qué más nos puede hacer falta?... Vamos a ver.... Una variable que almacene el estado actual: si estamos navegando, insertando o editando un registro e incluso si esta inactiva porque no contiene todavía elemento alguno.

```
TEstado = (esInactivo, esNavegar, esEditar, esInsertar, esBorrar);
```

Y poco más. Algunos procedimientos que nos permitan mantener coherentes la transición de estados, impidiendo por ejemplo que podamos borrar un registro que no haya sido previamente validado. Todo esto lo iremos explicando a lo largo del artículo.

Lo tenéis con mayor detalle en el **listado 1**

```
type
   //Definimos los posibles estados que permitimos
   TEstado = (esInactivo, esNavegar, esEditar, esInsertar, esBorrar);

  TfrmDatos = class(TForm)
     ...
     ...
  private
     Listatemp: TStringList;
     Itemindex: Integer;      // nos inventamos un indice para movernos
     Estado: TEstado;   // necesitamos una variable que almacene el estado actual
     function Situarse(Index1,Index2: Integer):String;
     procedure ActualizarEstado(index: Integer);
     procedure EstadoInicial;
     procedure HazEdicion;
     procedure HazInsercion;
     procedure HazNavegacion;
     procedure HazCancelacion;
     procedure CopiaTabla(index: integer);//copia la tabla a partir de un indice
  public

end;
```

Listado 1. Métodos que declara la clase TfrmDatos (nuestro editor)

He intentado que la interfaz sea lo más clara posible, que estén lo más diferenciados posibles cada uno de los estados, y más concretamente la situación activo/no activo de los controles que permiten tanto la manipulación de datos como los que facilitan la navegación, rótulos informativos de situación, etc...

Vamos a construir un interfaz gráfico que contenga todos los elementos descritos hasta el momento. En la **figura 3** puedes ver el interfaz, ya en el proceso final de depuración del editor, mientras comprobaba que respondía básicamente a lo solicitado por nosotros.

Creación y destrucción de la ventana frmDatos.

Nuestra premisa era que partíamos, inicialmente, de una lista vacía. Por ello, en nuestro procedimiento de creación de la ventana, nos limitamos a crear un descendiente de la clase TStrings.

Toda vez que ha sido creado, y ajustada la cabecera de la tabla inferior (nos referiremos así cuando hagamos mención del componente TStringGrid de la **figura 3**), inicializamos el estado de aquellos controles que mantienen valor constante. No es el caso de los controles de Navegación que varía su activación dependiendo no solo del estado sino también de la posición del registro. Para solventar esta situación, hemos implementado el procedimiento ActualizarEstado( ) en el que se tiene en cuenta la posición del registro.

```
procedure TfrmDatos.Timer1Timer(Sender: TObject);
begin
   if not edi_Texto.Enabled then edi_Texto.color:= clGray
   else edi_Texto.color:= clWindow;
   if not edi_Valor.Enabled then edi_Valor.color:= clGray
   else edi_Valor.color:= clWindow;
end;
```

Por último, el procedimiento de reloj, aquí tiene un uso casi anecdótico. Se limitará a modificar el color de las dos casillas de edición, advirtiéndonos de aquellos momentos en que su valor *Enabled* es false, es decir, están desactivadas (cuando no hallan registros por ejemplo).

```
procedure TfrmDatos.FormCreate(Sender: TObject);
begin
   //Creamos la lista de cadenas sobre la que vamos a trabajar
   listatemp:= TStringList.Create;
   stg_Tabla.Cells[0,0]:= '                                          CONCEPTO';
   stg_Tabla.Cells[1,0]:= '    VALOR';
   EstadoInicial;
   Timer1Timer(Timer1);
end;
```

A la destrucción del formulario, haremos lo propio con la lista de cadenas.

```
procedure TfrmDatos.FormDestroy(Sender: TObject);
begin
   listatemp.Free;
end;
```

Podemos analizar cual es el estado inicial de partida: *Itemindex*, dado que no existe cadena alguna en nuestro editor, toma como valor 'cero' Y todos los controles de nuestra ventana se hallan desactivados o inicializados a *false*. Todos menos uno. El botón que implementa la capacidad de añadir un registro a nuestro editor (nuevo) se halla activo. Este será nuestro estado de partida.

```
procedure TfrmDatos.EstadoInicial;
begin
   //Definimos el estado valido inicial
   estado:= esInactivo;
   itemindex:= 0; //todavía no existe ningún registro en nuestro editor
   //inicializamos las etiquetas informativas del estado actual
   lab_Contador.Caption:= '0/0';
   lab_Estado.caption:= 'SIN REGISTROS';
   //inicializamos los cuadro de edición de entrada de registros
   edi_Texto.text:= '';
   edi_Texto.Enabled:= false;
   edi_Valor.text:= '';
   edi_Valor.Enabled:= false;
   //inicializamos los controles de navegación
   spb_Primero.enabled:= false;
   spb_Anterior.enabled:= false;
   spb_Posterior.enabled:= false;
   spb_Ultimo.enabled:= false;
   //inicializamos los controles de manipulación de datos
   bib_Nuevo.enabled:= true;
   bib_Validar.enabled:= false;
   bib_Borrar.enabled:= false;
end;
```

Primero, anterior, posterior y último.

Antes de iniciar la explicación te aclaro un pequeño matiz: *itemindex* sincroniza la posición del registro en nuestro TStringList (lo visualizas a través de las casillas de edición), con la posición del foco en la tabla inferior, que nos da una vista global de los registros existentes. La navegación se produce con sincronía ya que tanto para movernos sobre la tabla, como para hacerlo sobre el array, nos apoyamos constantemente en el valor de la variable *itemindex* (indica la posición que ocupa el registro activo). Esto es quizás, uno de los aspectos más importantes que encierra la aplicación. Itemindex es la referencia que toman tanto la tabla inferior como la lista de cadenas para sincronizar su posición.

El primer y ultimo de los registros apenas tienen dificultad:

```
procedure TfrmDatos.spb_PrimeroClick(Sender: TObject);
begin
```

```
   itemindex:= 1;
   edi_texto.text:= Situarse(0, itemindex);
   edi_valor.text:= Situarse(1, itemindex);
   stg_tabla.row:= 1;
   ActualizarEstado(itemindex)
end;

procedure TfrmDatos.spb_UltimoClick(Sender: TObject);
begin
   itemindex := listatemp.count;
   edi_texto.text:= Situarse(0, itemindex);
   edi_valor.text:= Situarse(1, itemindex);
   stg_tabla.row:= stg_tabla.RowCount - 1;
   ActualizarEstado(itemindex);
end;
```

Su estructura es semejante y se limitan a situarse, respectivamente en el primer y último registro, sincroniza tabla y array, y actualiza estado de controles de navegación por posición de registro.

Por otro lado, si analizamos anterior y posterior:

```
procedure TfrmDatos.spb_AnteriorClick(Sender: TObject);
begin
   if itemindex > 1 then
      begin
      edi_texto.text:= Situarse(0, itemindex - 1);
      edi_valor.text:= Situarse(1, itemindex - 1);
      itemindex := itemindex - 1;
      stg_tabla.row:= itemindex;
      ActualizarEstado(itemindex);
      end;
end;

procedure TfrmDatos.spb_PosteriorClick(Sender: TObject);
begin
   if itemindex < listatemp.count then
      begin
      edi_texto.text:= Situarse(0, itemindex + 1);
      edi_valor.text:= Situarse(1, itemindex + 1);
      itemindex := itemindex + 1;
      ActualizarEstado(itemindex);
      stg_tabla.row:= itemindex;
      end;
end;
```

Concluiremos que aun siendo similar a los dos primeros, se necesita comprobar que no se halla ni en la primera ni en la última posición, para no generar una excepción al salirse del rango de la matriz de cadenas. Anterior decrece en una unidad *itemindex*, mientras que posterior la incrementa.

Ahora bien, hemos comentado que había una sincronía entre la tabla y nuestro componente TStringList. Hasta este momento, la pulsación de los botones: primero, anterior, posterior y último, la establecía. Al contrario, desde la pulsación de un simple Click de selección en la tabla, existe también sincronía con la posición actual de itemindex sobre la lista de cadenas. En la implementación del método stg_TablaClick (es decir al seleccionar una de las filas):

itemindex:= stg_tabla.Row;

es la rutina de asignación que la garantiza.

```
procedure TfrmDatos.stg_TablaClick(Sender: TObject);
begin
   //solo en el caso de que haya algún elemento, este enfocado y en estado
```

```
   //de navegación
   // ES MUY IMPORTANTE ESTA ULTIMA RESTRICCION
   if (itemindex > 0) and (stg_tabla.focused) and (estado = esNavegar) then
      begin
      itemindex:= stg_tabla.Row; //actualizamos el índice
      edi_texto.text:= Situarse(0, itemindex); //obtenemos los valores
      edi_valor.text:= Situarse(1, itemindex);
      HazNavegacion;   //Hala... otra vez a navegar
      ActualizarEstado(itemindex);
      end;
end;
```

Es importante que tan solo se ejecute el cuerpo del procedimiento en el estado de Navegación, pues de encontrarnos en un estado diferente o ejecutarse en un estado de inactividad (itemindex = 0), generaría bien un cambio de estado (rompiendo el diagrama de estados), bien una excepción al salir del rango de la matriz.

Añadir, editar o borrar un registro.

Deberíamos tener en mente a estas alturas que la adición, edición, cancelación o borrado de un registro, llevaría aparejada una transición de estado,(es necesario haber diseñado un esquema de transición de estados, aunque fuera garabateado, como lo hacemos rápidamente en la **figura 4**). Este diagrama nos permite valorar en que medida se cumplen los objetivos que queremos.

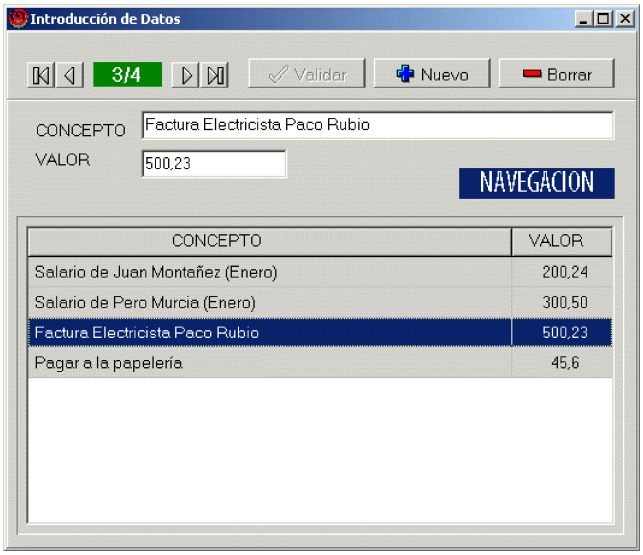

Figura 3: Interfaz gráfico de nuestro editor al finalizar el desarrollo

Podemos describir el planteamiento propuesto en el diagrama:

Partimos de un estado inicial ***esInactivo***. Cualquier intento de adquirir un nuevo estado debe pasar obligatoriamente por la adición de un nuevo registro, que nos mueve hacia el estado ***esInsertar***. Iniciado este estado, bien puede suceder la validación del mismo, que nos llevaría al estado ***esNavegar***, o la cancelación de la inserción, que nos regresaría al estado de partida. A partir de ese momento, el estado ***esNavegar*** se convierte en el centro de referencia, manteniendo una independencia entre los tres estados principales sobre los que se sostiene el editor: ***esInsertar***, ***esEditar*** o ***esBorrar***.

Así pues, para pasar a estado de inserción, nos valemos de la pulsación del botón cuyo *caption* es 'Nuevo' y que se acompaña del signo de la suma. Además, permitimos que la pulsación de dicha tecla acarree la invocación del método de adición.

```
procedure TfrmDatos.bib_NuevoClick(Sender: TObject);
begin
   HazInsercion; // Premaramos el estado de los controles que intervienen
   itemindex:= listatemp.Add('='); //Creamos una nuevo elemento en nuestra lista
   //Si hay uno o mas elementos antes de la pulsación
   //Aumentamos en una fila nuestra tabla StringGrid
   {Notese que el método Add ya nos situa en el último elemento de la lista
   pero como no existe dicho método desde la tabla, debemos fingirlo}
```

```
   if itemindex > 0 then
      begin
      stg_tabla.RowCount:= stg_Tabla.rowcount + 1;
      stg_tabla.row:=stg_tabla.RowCount - 1;
      end;
   stg_tabla.Enabled:= false;//desactivamos la tabla
   Inc(itemindex); // nos situamos correctamente ya que el método add nos
                   // devuelve la posición dentro de la matriz que es siempre
                   // menor en un elemento, tal como ocurre cuando hacemos un
                   //bucle for para recorrerla: (count-1)
   edi_texto.Text:= '';
   edi_valor.Text:= '0,0';
   edi_texto.enabled:= true;
   edi_valor.enabled:= true;
   edi_texto.SetFocus; //enfocamos la casilla de edición CONCEPTO
end;
```

El procedimiento *HazInsercion* nos prepara la transición al estado de inserción (esInsertar). Su invocación afecta a aquellos controles que tienen un comportamiento constante ante dicho estado. Añadimos un elemento a la lista de cadenas (una cadena vacía con formato '=') y depende de la posición en que nos encontremos, añadimos o no una fila a nuestra tabla. Fijaos que de estar en la posición de la primera cadena del array, la tabla ya presenta de partida esa fila, por lo que no es necesario añadir una fila más. Finalmente, y después de vaciar de contenido las casillas de edición, entregamos el foco al cuadro de edición del concepto, permaneciendo en el estado de inserción hasta que no se valida mediante la pulsación <enter> o la de cancelación <esc>.

```
 procedure TfrmDatos.HazInsercion;
begin
   estado:= esInsertar;
   lab_Estado.caption:= 'INSERCION';
   //actualizamos el estado de los controles de navegación
   spb_Primero.enabled:= false;
   spb_Anterior.enabled:= false;
   spb_Posterior.enabled:= false;
   spb_Ultimo.enabled:= false;
   //actualizamos el estado de los controles de manipulación de datos
   bib_Nuevo.enabled:= false;
   bib_Validar.enabled:= true;
   bib_Borrar.enabled:= false;
end;
```

No dejamos más opción que validar, y al hacerlo, regresaremos al estado de Navegación, eje central sobre el que nos movemos.

```
procedure TfrmDatos.bib_ValidarClick(Sender: TObject);
begin
   // comprobamos rapidamente que nos encontramos frente a un número decimal
   try
      strToFloat(edi_valor.text);
   except
   on E:Exception do
      begin
      ShowMessage('Introduce un valor decimal correcto: '+ edi_valor.text);
      edi_valor.setfocus;
      Exit;
      end;
   end;

   // si es correcto el valor decimal y cumple que el concepto es no
   // nulo o bien que no contiene carácter en blanco
   if (edi_texto.text <> '') and (edi_texto.text <> ' ') then
      begin  // actualizamos el valor en nuestro StringList en forma de igualdad
listatemp.Strings[itemindex-1]:= edi_texto.text + '=' + edi_valor.text;
      //Y paralelamente en nuestro StringGrid
```

```
      stg_tabla.cells[0, itemindex]:= edi_texto.text;
      stg_tabla.cells[1, itemindex]:= edi_valor.text;
      //Nos situamos sobre el StringGrid
      stg_tabla.row:= itemindex;
      HazNavegacion;     //Navegamos
      ActualizarEstado(itemindex); // Actualizamos estado de los botones de navegación
      end
   else
      begin
      ShowMessage('Introduce un texto en el cuadro de edición de CONCEPTO');
      //Un error se ha producido y volvemos a las casillas de edición
      edi_texto.setfocus;
      end;
end;
```

El procedimiento de validación, principalmente se limitará a comprobar que los valores introducidos son correctos: no se admiten cadenas vacías, (ni siquiera deberíamos permitir la existencia de cadenas repetidas dentro de la lista de Strings. Prescindimos de este requisito para no oscurecer la comprensión del código innecesariamente. Es un buen momento para repasar los comentarios que hacíamos sobre Sort, que nos podrían ofrecer nuevas capacidades a nuestro editor). En el apartado del campo Valor, deberemos asegurarnos de que solo se almacenan números decimales.

Pero, entonces ¿cuando entramos en estado de edición?:

```
procedure TfrmDatos.edi_TextoEnter(Sender: TObject);
begin
   if (estado = esNavegar) then
      begin
      estado:= esEditar;
      HazEdicion;
      end;
end;
```

Al pulsar <enter> sobre los cuadros de edición Concepto y Valor, entraremos en estado de Edición. Hemos necesitado condicionarlo a encontrarnos, al momento de la pulsación, en estado de navegación (esNavegar), puesto que en estado de Inserción, también es invocado el procedimiento al escribir sobre el nuevo registro. De esa forma, mantenemos la independencia entre ambos estados.

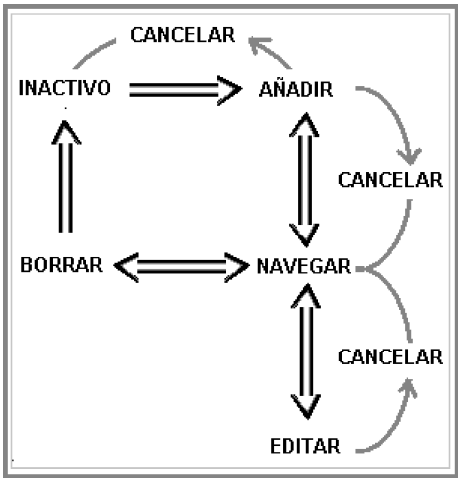

Figura 4: Diagrama de estados

Y nos queda el procedimiento de Borrado de registro.

También surge la necesidad de definir un estado de borrado. Si observamos la transición de estados que hemos establecido en la figura 4, el proceso de cancelación afecta al estado de Inserción y al de Edición pero no al de Borrado. Al establecer el estado, permitimos condicionar la pulsación de <ESC> a no encontrarnos en estado de Borrado, puesto que ya tenemos un mensaje condicional que nos permite abortar el proceso después de la pulsación del botón cuyo *caption* es Borrar.

```
procedure TfrmDatos.bib_BorrarClick(Sender: TObject);
var
saltar: boolean;
begin
saltar:= false;
estado:= esBorrar;
lab_estado.caption:= 'BORRAR'; // mostramos el rótulo antes de la ventana modal
if MessageDlg('¿Quieres borrar la linea '+
         Situarse(0,itemindex)+ '________'+ Situarse(1,itemindex)+
```

```
'?',mtWarning,[mbOk,mbCancel],0)= mrOk then
   begin  //si decidimos borrar la linea
   lab_estado.caption:= 'BORRAR';    //mantenemos el rótulo
   listatemp.Delete(itemindex-1); //eliminamos el elemento de la lista pero
                                  //falta eliminarlo de la tabla
   //Nos situamos correctamente en el elemento anterior al borrado
   if (itemindex > 1) and (listatemp.Count >= 1) and (listatemp.count < itemindex) then
     begin
     if (itemindex = (stg_tabla.RowCount-1)) then saltar:= true; //estamos en la última
                                                 //fila (debemos omitir copiar tabla())
     Dec(itemindex);
     end;
   //Salvo que estemos en el primer elemento que en vez de decrecer, lo dejamos así
   if (itemindex = 1) and (listatemp.Count = 0) then itemindex:= 0;

   HazNavegacion;  //Hala... a navegar

   //Vamos a operar con el resto de elementos que faltan (sobretodo la tabla)
   if itemindex > 0 then  //si despues de borrar todavía queda algún elemento
      begin
      estado:= esNavegar; //fijamos estado
      edi_texto.text:= Situarse(0, itemindex); // actualizamos las casilla
      edi_valor.text:= Situarse(1, itemindex); // de edición
      if saltar then    stg_tabla.RowCount:= stg_tabla.RowCount - 1   //no hace falta
                                           //borrar tabla solo eliminar ultima fila
       else CopiaTabla(itemindex);  //operamos sobre la tabla
      ActualizarEstado(itemindex);
      end
   else  //en caso contrario
      begin //pasamos a un estado de inactividad
      EstadoInicial;  //ya incluye estado en inactivo
      //inicializamos campos
      stg_tabla.row:= 1;
      stg_tabla.rowcount:= 2;
      stg_tabla.Cells[0,1]:= '';
      stg_tabla.Cells[1,1]:= '';
      end;
   end;
end;
```

Y quizás, lo que pueda llamarnos más la atención es la existencia dentro del proceso de borrado del método CopiaTabla( ). Intentaré razonarlo:

Nuestra tabla es un componente de la clase TStringGrid. Este componente, oculta una gran parte de los métodos disponibles en la clase TStrings, obligándonos a redefinirlos si queremos acceder a su uso. No podemos en principio, hacer uso del método Assign( ) que nos permitiría recibir el contenido de la lista tras la eliminación de un elemento. La opción que se me ocurre más sencilla es recorrer la tabla secuencialmente, desde el registro en que se produce el borrado hasta el último de los registros, recuperando en la fila anterior el valor de la posterior y acabando dicho recorrido con la eliminación de la última de las filas en la tabla. Dada las características de nuestra aplicación (pequeña cantidad de pagos -poco número de registros-) planificar un algoritmo de esas características apenas es problemático. No estamos hablando de una base de datos, tal y como las podemos concebir y eso nos puede dar licencia, pienso, para admitirlo.

```
procedure TfrmDatos.CopiaTabla(index: integer);
var
xIndice: Integer;
begin
   //recorremos toda la tabla (OJO, HE DICHO TODA)-------!
   for xIndice:= index to  stg_tabla.RowCount - 2 do    {!}
      begin                                             {!}
      stg_tabla.Cells[0,xIndice]:= stg_tabla.Cells[0,xIndice+1];
      stg_tabla.Cells[1,xIndice]:= stg_tabla.Cells[1,xIndice+1];
```

```
      end;
   //Toda pero a partir de determinada posición.
   //Cada fila copia el valor de la fila posterior
   //y al final eliminamos la que sobra
   stg_tabla.RowCount:= stg_tabla.RowCount - 1;
end;
```

Cancelar, editar o insertar...

Cuando declaramos el tipo TEstado en la interfaz del módulo del editor omitíamos un estado de Cancelación. Hemos prescindido de él porque, digámoslo de una manera sencilla, es un estado transitorio que no requiere la intervención del usuario más que para su activación. A diferencia del estado de inserción, por poner un ejemplo, Cancelar se ejecuta sin interrupciones que le puedan apartar de su objetivo final:regresar al estado anterior. No es el caso de Insertar. Entramos en este estado tras la ejecución del procedimiento btn_NuevoClick( ), ya sea mediante el botón o de la pulsación de <+> y durante su corto espacio de vida, comparte procedimientos comunes a otros estados, como el de edición: se introducen valores en las casillas de edición. Ese es el momento en que se hace necesario la definición de un estado puesto que nos va a permitir diferenciar el objetivo dentro de procedimientos comunes a la transición entre los estado definidos.

Nos valemos de los eventos para invocar un estado determinado. Esta corresponde a la implementación del evento OnKeyDown en ambos cuadros de edición:

```
procedure TfrmDatos.edi_TextoKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
   // al pulsar sobre ESC entramos en estado de Cancelación
   if (key = VK_ESCAPE) and (estado <> esBorrar) then HazCancelacion;
   //al pulsar '+' simulamos la pulsación de NUEVO
   if (key = VK_ADD) and (estado = esNavegar) then bib_NuevoClick(sender);
   //al pulsar '-' simulamos la pulsación de BORRAR
   if (key = VK_SUBTRACT) and (estado = esNavegar) then bib_BorrarClick(sender);
end;
```

Tomando como referencia HazCancelación y siguiendo el flujo del programa una vez que ha sido invocada la rutina, observamos como el estado nos permite condicionar nuestras acciones. En el caso de tener que deshacer un estado de edición será tan fácil como volver a situarnos en el registro actual, retomando de nuevo todos los valores antes de iniciar el estado de edición.

```
procedure TfrmDatos.HazCancelacion;
begin
   {SOLO PARA EL CASO DE QUE NOS ENCONTREMOS EDITANDO UN REGISTRO}
   if (estado = esEditar) then
      begin //reponemos los valores originales
      edi_texto.text:= Situarse(0, itemindex);
      edi_valor.text:= Situarse(1, itemindex);
      HazNavegacion; //Volvemos al estado original
      ActualizarEstado(itemindex);
      end;
```

Reponer el estado anterior a una inserción es algo más complicado. Tenemos que tener en cuenta que ya se ha añadido una cadena a nuestro lista TStringList. Esa será nuestra primera prioridad: borrar la cadena añadida y obtener el valor de *itemindex* previo a la inserción. Es de gran importancia este paso, ya que, al cancelar la acción de insertar un elemento, podríamos regresar a un estado inicial donde *itemindex* toma como valor 0 (no hay elementos en el editor), originando una excepción al invocar la lectura del registro actual a través del método Situarse( ) (que toma como lectura posiciones en la matriz en *itemindex-1*). Vistas

las dos posibilidades: existían elementos en el editor antes de la inserción, o no existían elementos, condicionamos la ejecución de unas acciones u otras al valor de *itemindex*

```
      {SOLO PARA EL CASO DE QUE NOS ENCONTREMOS INSERTANDO UN REGISTRO}
   if (estado = esInsertar) then
      begin //reponemos los valores originales
      listatemp.Delete(itemindex-1);
      // si se produce la cancelación se procede a decrecer el indice
      if (itemindex > 1) and (listatemp.Count >= 1) and (listatemp.count < itemindex)
then Dec(itemindex);
      // ojo.. en este caso estamos al principio y no podemos decrecer pero
      // si debemos fijar a 0
      if (itemindex = 1) and (listatemp.Count = 0) then itemindex:= 0;

      if itemindex = 0 then    //si no hay filas
         begin //reponemos el estado inicial o inactivo
         edi_texto.Text:= '';
         edi_valor.text:= '';
         EstadoInicial;
         stg_tabla.rowcount:= 2;  // modificamos la tabla
         stg_tabla.row:= 1;
         end
      else
         begin //reponemos el estado de navegación
         edi_texto.text:= Situarse(0, itemindex);
         edi_valor.Text:= Situarse(1, itemindex);
         stg_tabla.cells[0, itemindex]:= edi_texto.text; //modificamos la tabla
         stg_tabla.cells[1, itemindex]:= edi_valor.text;
         stg_tabla.rowcount:= stg_tabla.RowCount - 1;
         HazNavegacion;
         end;
      ActualizarEstado(itemindex)
      end;
end;
```

En cualquier caso, obtenido el valor de nuestra variable índice *itemindex*, podemos operar sobre la tabla inferior, garantizando una sincronía entre tabla y lista de Strings.

Frente a este procedimiento anterior, HazEdicion y HazCancelacion, mantienen una estructura paralela, que nos facilita enormemente el mantenimiento y la depuración del código implementado.

```
procedure TfrmDatos.HazEdicion;
begin
   estado:= esEditar;
   lab_Estado.caption:= 'EDICION';
   //actualizamos el estado de los controles de navegación
   spb_Primero.enabled:= false;
   spb_Anterior.enabled:= false;
   spb_Posterior.enabled:= false;
   spb_Ultimo.enabled:= false;
   //actualizamos el estado de los controles de manipulación de datos
   bib_Nuevo.enabled:= false;
   bib_Validar.enabled:= true;
   bib_Borrar.enabled:= false;
   stg_tabla.Enabled:= false;
end;
```

```
procedure TfrmDatos.HazNavegacion;
begin
   estado:= esNavegar;
   lab_Estado.caption:= 'NAVEGACION';
   spb_Primero.enabled:= true;
   spb_Anterior.enabled:= true;
   spb_Posterior.enabled:= true;
   spb_Ultimo.enabled:= true;
   bib_Nuevo.enabled:= true;
   bib_Validar.enabled:= false;
   bib_Borrar.enabled:= true;
   stg_tabla.Enabled:= true;
end;
```

Ambos procedimientos, como puedes observar, se limitan a mantener coherente nuestro interfaz gráfico con el estado de nuestro editor de registros. Afectan a aquellos controles que tienen siempre el mismo comportamiento frente a un estado definido, pero ese no es el caso que comentaremos a continuación.

```
procedure TfrmDatos.ActualizarEstado(index: Integer);
begin
  if index < 1 then
     begin   //no hay filas
     spb_primero.enabled:= false;
     spb_anterior.enabled:= false;
     spb_posterior.enabled:= false;
     spb_ultimo.enabled:= false;
     bib_validar.enabled:= false;
     bib_nuevo.enabled:= true;
     bib_borrar.enabled:= false;
     lab_Contador.Caption:= '0/0';
     edi_texto.Enabled:= false;
     edi_valor.Enabled:= false;
     end;
  if (index = 1) and (listatemp.Count = 1) then
     begin
     spb_primero.enabled:= false;
     spb_anterior.enabled:= false;
     spb_posterior.enabled:= false;
     spb_ultimo.enabled:= false;
     lab_Contador.Caption:= IntToStr(1)+'/'+IntToStr(1);
     edi_texto.Enabled:= true;
     edi_valor.Enabled:= true;
    end;
  if (index = 1) and (listatemp.Count > 1) then
     begin
     spb_primero.enabled:= false;
     spb_anterior.enabled:= false;
     spb_posterior.enabled:= true;
     spb_ultimo.enabled:= true;
     lab_Contador.Caption:= IntToStr(itemindex)+'/'+IntToStr(listatemp.Count);
     edi_texto.Enabled:= true;
     edi_valor.Enabled:= true;
    end;
  if (index > 1) and (index < listatemp.Count) then
     begin
     spb_primero.enabled:= true;
     spb_anterior.enabled:= true;
     spb_posterior.enabled:= true;
     spb_ultimo.enabled:= true;
     lab_Contador.Caption:= IntToStr(itemindex)+'/'+IntToStr(listatemp.Count);
     edi_texto.Enabled:= true;
     edi_valor.Enabled:= true;
    end;
  if (not (index = 0)) and (not (index = 1)) and (index = listatemp.Count) then
     begin
     spb_primero.enabled:= true;
     spb_anterior.enabled:= true;
     spb_posterior.enabled:= false;
     spb_ultimo.enabled:= false;
     lab_Contador.Caption:= IntToStr(itemindex)+'/'+IntToStr(listatemp.Count);
     edi_texto.Enabled:= true;
     edi_valor.Enabled:= true;
     end;
end;
```

Listado 2: Implementación del procedimiento ActualizarEstado( ).

Cuando dependemos de la posición del registro...

Nos queda un pequeño escollo por salvar y que afecta a todos aquellos objetos que van a modificar el valor de alguna de sus propiedades, según la posición del registro. Todos están relacionados, de alguna forma, con la necesidad informar al usuario de la posición del registro actual, representado por la variable *itemindex*. Esto resulta definitivo, vital en nuestro intento de aumentar el grado de satisfacción del usuario. Debe saber en todo momento en donde está, y que está haciendo.

Echa un mirada al **listado 2** -ActualizarEstado( )- y comentamos.

Si observas muchos de los métodos, prácticamente todos siguen un mismo esquema lógico: se inicia un estado, hay modificaciones en función del estado, y se acaba llamando al método de ActualizarEstado, que rectifica en algún caso la activación de un control determinado, según sea el valor de itemindex. Este es el parámetro que recibe el procedimiento.

Vamos a analizar las distintas alternativas que se me han ocurrido pueden suceder:

- `(index < 1)`
- No hay registros en el editor. Todos los controles deben estar desactivados. Las etiquetas mostrarán el estado de inicialización.

- `((index = 1) and (listatemp.Count = 1))`
  Estamos en el primero de los registros pero tampoco podemos navegar, puesto que solo hay un registro. Vuelven a estar todos los controles de navegación desactivados pero esta vez dejamos activos los cuadros de edición para que pueda ser modificado el registro.

- `((index = 1) and (listatemp.Count > 1))`
  Estamos situados en el primer registro de una lista con múltiples registros. Via libre para todo menos para ir hacia atrás: anterior y primero son desactivados.

- `((index > 1) and (index < listatemp.Count))`
  Situación ideal para la navegación. Via libre. Todo esta activado.

- `((not (index = 0)) and (not (index = 1)) and (index = listatemp.Count)))`
  Estamos situados en el ultimo de los registros. Date cuenta, de que, de no poner las dos primeras condiciones cumpliría también el caso primero o el caso segundo. Necesitamos debilitar la condición para que este suceso tan solo se produzca cuando nos encontramos en el último de los registros. En ese caso, habilitaremos los controles de navegación anterior y primero, y deshabilitaremos los de último y posterior.

Hay muchas formas de poner lo mismo que yo he puesto. Hay programadores que prefieren ligar el estado de un botón al estado de uno o mas botones, de forma que se excluyan según su activación o no activación. Al final, me parece que lo realmente importante es que estén recogidos todos los estados posibles y que exista una estructura lo suficientemente clara que permita una rápida depuración y localización de errores. Sin una mínima estructura lógica, a poco que pueda crecer nuestra aplicación, adolecerá de ser poco fácil de mantener, convirtiéndose con el tiempo en una soga alrededor de nuestro frágil cuello, apretándonos un poco más cada día.

Primera conclusión y despedida.

El resto de procedimientos incluidos en el código fuente, tienen menor importancia y en general, participan de crear un flujo dinámico en la aplicación. Se intenta sobre todo que el usuario se sienta cómodo

con ella y que sea realmente útil para él.Como resumen resaltaría, cómo nos hemos valido de la propiedad Names y Values, que nos han evitado tener que implementar métodos para “filtrar” las cadenas de la lista.

Nos queda por ver, y será en el próximo número, (donde continuaremos nuestro pequeño ejemplo), el uso de la propiedad Objects y la asociación de estructuras a cada una de las cadenas que componen nuestro componente TStringList. Junto al artículo se acompaña el código fuente del editor. Espero que hayáis encontrado ameno este rato que hemos compartido.

Y Manolo ... ¿Qué habrá sido de él? Andará discutiendo todavía con su jefe.Con un poco de suerte, en el próximo número, tendrá un motivo menos de que preocuparse.

Nos vemos pronto. Un saludo a todos.