Creación de componentes VCL (III)

Componentes visuales

Ahora que ya sabemos programar nuestros componentes no visuales, vamos a aprender todo lo necesario para hacer un componente visual, de esos que "pinchamos" en nuestros formularios para que luego aparezcan al ejecutar el programa.

Índice

Índice 1

Introducción 2

Tipos de componentes visuales 2

Interactuando con los componentes 3

Buscando a nuestro padre 4

Definiendo el interfaz del componente 5

Interceptando los eventos 7

Carga de datos... 9

Manejando los bits para buscar las unidades 9

¿Y dónde están mis iconos? 11

Heredando de TCustomComponent: otro mundo 11

Los ejemplos 12

Introducción

Durante los dos artículos anteriores, hemos explicado detalladamente cómo podemos crear nuestros propios componentes no-visuales, y realizar con ellos ciertas tareas más o menos complejas de forma transparente. También hemos explicado cómo definir nuevos eventos, que el programador podrá codificar para responder ante ciertas situaciones que puedan resultar importantes. Ahora vamos a aplicar todo lo que ya hemos aprendido para crear otro tipo de componentes: los visuales.

Tipos de componentes visuales

En el primer artículo de esta serie, dijimos que podemos clasificar los componentes en dos grupos: visuales y no-visuales. Los segundos ya los hemos tratado, por ser los más sencillos. De los primeros (los componentes visuales), tenemos todavía mucho por aprender.

Para empezar, los componentes visuales podemos dividirlos a su vez en dos tipos:

1. *Componentes interactivos*: permiten que el usuario final los manipule, ya sea introduciendo datos, seleccionando elementos, etc. De forma que estos componentes pueden recibir el foco (con `SetFocus`) así como los eventos propios del teclado y del ratón. Normalmente, el propio sistema operativo es el encargado de dibujar el aspecto del componente, haciendo el componente las llamadas correspondientes para que este aspecto cambie.

2. *Componentes gráficos*: el propio componente es el encargado de dibujar en la pantalla lo que crea oportuno, bien a través de las funciones básicas del API de Windows (con el objeto `TCanvas`) o bien a través de otras librerías gráficas, como OpenGL, DirectX, etc. Estos componentes, no suelen recibir eventos del usuario final, aunque si eventos del propio programador, ya que su cometido no suele ir más allá de mostrar ciertos gráficos o imágenes en la pantalla.

En este artículo vamos a explicar en profundidad las técnicas más comunes para crear componentes interactivos, y dejaremos para el próximo el tema de los componentes gráficos.

Interactuando con los componentes

Si tuviéramos que crear un componente interactivo desde el principio, sería demasiado complejo, ya que tendríamos que luchar contra el propio API del sistema operativo, gestionando sus mensajes, las llamadas las funciones a bajo nivel, etc. Sin embargo, podemos aprovechar la mayoría del trabajo hecho por Borland en la VCL, y crear componentes interactivos a partir de otros ya existentes, aplicando la técnica de la herencia.

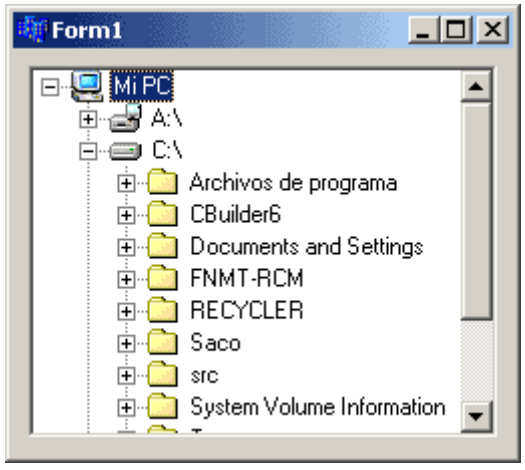
Dado que un componente es un objeto como otro cualquiera, podremos aplicar en él todas las técnicas de la orientación a objetos: encapsulación, herencia y polimorfismo.

La encapsulación ya la hemos utilizado sin apenas darnos cuenta, simplemente definiendo ciertos atributos como privados, para que no puedan ser accedidos desde fuera de la clase, y otros públicos o publicados (`published`) para ser usados por el programador que utilice el componente.

La herencia nos permite reutilizar código, haciendo que una clase dada (que llamaremos clase hija) adquiera todos los atributos y métodos públicos y protegidos de otra, llamada clase padre. De este modo, podemos aprovechar mucho código escrito, simplemente heredando de una clase ya escrita. Por si fuera poco, también es posible que una clase hijo tenga a su vez más descendencia, adquiriendo estos las características del padre y del "padre del padre", es decir: del abuelo.

El polimorfismo lo dejaremos para otra ocasión, ya que se trata de una técnica un poco más compleja, que requiere entender y manejar correctamente la herencia.

La técnica de la herencia, aplicada a los componentes, nos permite personalizar cualquier componente para un uso más específico, ya sea porque no se ajusta bien a lo que necesitamos, o porque queremos ampliar las posibilidades del componente. La propia VCL utiliza la herencia continuamente, bien para reutilizar código de clases padre, o bien para ofrecernos clases padre de las que nosotros podemos heredar. Muchas "suites de componentes" hacen también esto, proporcionando un componente heredado a partir de cada uno de los básicos, y añadiendo en este nuevas características.



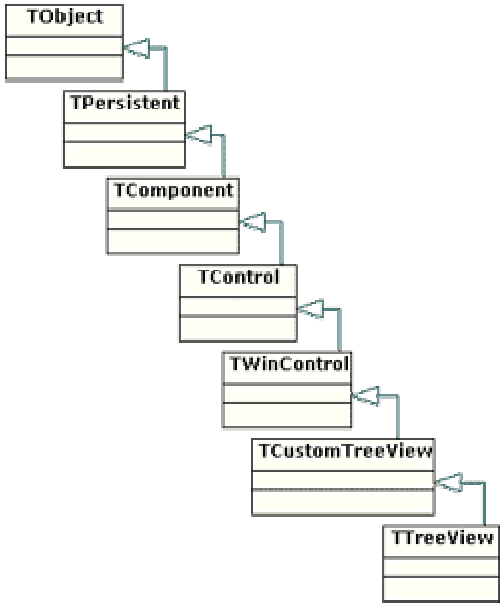
Para explicar cómo crear un componente interactivo, vamos a hacerlo a través de un ejemplo práctico: supongamos que necesitamos mostrar un árbol que muestre los directorios a partir de una carpeta dada, incluso supongamos que necesitamos mostrar el típico árbol de "Mi PC", mostrando todas las unidades y las carpetas de cada una.
Para ello, podríamos escribir un componente desde cero, gestionando todo lo referente a la estructura en árbol, el dibujado en pantalla, la gestión de mensajes, etc., o podríamos utilizar la herencia para reutilizar el código ya escrito,

concretamente basándonos en el componente `TTreeView`, que es el que nos permite mostrar en un formulario estructuras de tipo árbol. Sin embargo, el `TTreeView` que viene por defecto en Delphi, es muy genérico, y sirve para mostrar cualquier tipo de árbol. Nuestro caso es más específico: necesitamos mostrar un árbol, sí, pero más concretamente un árbol de directorios. Llamaremos a nuestra nueva creación `TArbolDirectorios`.

Buscando a nuestro padre

Lo primero que tenemos que decidir es de qué componente debemos heredar para tener la mayor parte del trabajo hecho. A priori, puede parecer sencillo ¿no? "Pues para tener un árbol tenemos que heredar de la clase `TTreeView`" diría cualquiera. Pues sí, pero no. La VCL esconde algunos secretos que tenemos que conocer si vamos a crear nuestros propios componentes heredados.

Antes hemos dicho que la VCL utiliza continuamente la herencia para sus propios componentes, es decir: que el componente `TTreeView` también tiene padre, abuelo, bisabuelo y una larga lista de ancestros, que han definido sus características, modo de actuación, apariencia, etc. Para que os hagáis una idea de la complejidad, podemos ver todos los antecesores de la clase `TTreeView` en la imagen de la derecha.



De todo este árbol genealógico, sólo vamos a fijarnos a partir `TComponent`:

- `TComponent`: este nos resulta familiar, ya que es la clase a partir de la que hemos creamos nuestro conversor de monedas de los anteriores números. Los componentes no-visuales deben heredar directamente de `TComponent`, ya que este proporciona las características básicas.

- `TControl`: se trata de la clase padre para todos los componentes visuales, ya sean gráficos o no.

- `TWinControl`: de esta clase descenderán todos los componentes dibujados directamente por Windows. La principal característica de estos objetos (llamados ventanas o Windows) es que están identificados por un número único llamado descriptor o manejador de ventana (en inglés handle).

- `TCustomTreeView`: se trata de la clase que permite mostrar un árbol dentro de un formulario de Delphi. La peculiaridad es que esta clase no tienen ningún método ni propiedad en la sección `published`, por lo que no puede ser manejada desde el entorno de Delphi.

- `TTreeView`: esta ya es la clase o componente final, que aparece registrado en la paleta de componentes, y que puede ser configurado en tiempo de diseño. En realidad, todo el trabajo de esta clase se limita a publicar las propiedades y eventos que han sido programados en la clase `TCustomTreeView`.

Como veis, lo que parecía obvio no lo es tanto. Casi todos los componentes de la VCL que aparecen en la paleta son clases casi vacías, que sólo publican ciertas propiedades y eventos de la clase padre (que suele anteponer el prefijo Custom a su nombre). Es en esa clase "Custom" donde se codifica todo lo necesario para que el componente funcione. La gente de Borland diseñó de esta forma los componentes para permitir crear componentes especializados, como nuestro `TArbolDirectorios`, pero que no muestren todas las propiedades y eventos, sino sólo los que sean propios del componente que estamos creando. En nuestro caso, no publicaremos todas las propiedades, ya que consideramos que algunas no son apropiadas para un árbol de directorios.

Definiendo el interfaz del componente

Ahora que ya sabemos quien va a ser el padre de nuestro componente, tenemos que decidir qué propiedades y eventos vamos a proporcionar a nuestros usuarios del componente.

A priori parecen interesantes o necesarias las siguientes:

- CarpetaRaiz: se trata de una cadena que contendrá la carpeta a partir de la que se creará el árbol de directorios. Puede ser la carpeta raíz de una unidad de disco (p.e. "C:\") para mostrar los directorios de todo la unidad, o bien una carpeta cualquiera, para mostrar las subcarpetas de esta. Un caso especial se dará cuando esta propiedad contenga el valor "Mi PC", con en el que mostraremos cada una de las unidades disponibles en nuestro sistema, pudiendo expandir estas unidades para mostrar sus carpetas.
- *CarpetaActual*: es un valor de tipo cadena que contiene la ruta de la carpeta seleccionada en el árbol. Si, por ejemplo, tenemos seleccionada la carpeta de primer nivel "Datos", y el nodo raíz es "C:\Temporal", esta propiedad contendrá el valor "C:\Temporal\Datos". También se puede modificar su valor, seleccionándose en ese caso el nodo correspondiente en el árbol (si el valor establecido es correcto).

- *MostrarPadre*: es un valor booleano (`true` o `false`) que nos permite configurar la aparición o no del nodo padre. Es decir: si en la propiedad "CarpetaRaiz" hemos introducido el valor "C:\Delphi", en el árbol aparecerá un nodo raíz llamado "Delphi", y colgando de él, todas las demás subcarpetas. Si establecemos esta propiedad a falso, el nodo raíz desaparecerá, mostrándose todas las subcarpetas en un primer nivel, sin colgar de ningún nodo padre.

Bueno, creo que con esto puede ser suficiente para tener un componente bastante versátil.
Un primer esqueleto del componente podría ser el que vemos en el siguiente listado:

```
TCustomArbolDirectorios = class(TCustomTreeView)
private
   FMostrarPadre: boolean;
   FCarpetaRaiz: string;

  procedure SetMostrarPadre(value: boolean);
  procedure SetCarpetaRaiz(value: string);

  function  GetCarpetaActual: string;
protected
  property MostrarPadre: boolean
           read FMostrarPadre
           write SetMostrarPadre;
  property CarpetaRaiz: string
           read FCarpetaRaiz
           write SetCarpetaRaiz;
  property CarpetaActual: string
           read GetCarpetaActual;
public
  constructor Create(AOwner: TComponent); override;
end;
```

Si lo estudiamos detenidamente, veremos que hemos heredado nuestra clase de `TCustomTreeView` (por las razones que ya hemos explicado), definiendo las propiedades que acabamos de mencionar, así como los correspondientes métodos de lectura (Get) y escritura (Set). Es estos métodos de escritura, haremos las validaciones correspondientes, como por ejemplo, verificar que la carpeta exista.

Y por último, ¿alguien se ha fijado en el nombre que le he dado a la clase del nuevo componente? ¿`TCustomArbolDirectorios`? ¿No dijimos que iba a llamarse `TArbolDirectorios`? Pues sí, pero he aplicado el mismo patrón de diseño que utiliza Borland para la VCL. En nuestra clase "Custom", definiremos toda la lógica de nuestro componente, sin publicar ninguna propiedad ni evento, y más tarde definiremos la clase definitiva (`TArbolDirectorios`) que simplemente publicará las propiedades que consideremos oportunas, tanto de `TCustomTreeView` como de `TCustomArbolDirectorios`. Podéis ver el esqueleto de `TArbolDirectorios` en el siguiente listado:

```
TArbolDirectorios = class(TCustomArbolDirectorios)
published
  // propiedades heredadas de TCustomTreeView
  property AutoExpand;
  property BorderStyle;
  property HotTrack;
  property Images;
  property ReadOnly;
  property RightClickSelect;

  // eventos heredados de TCustomTreeView
  property OnChange;
  property OnChanging;
  property OnCollapsed;
  property OnCollapsing;
  property OnDeletion;
  property OnEditing;
  property OnEdited;
  property OnExpanding;
  property OnExpanded;
  property OnGetImageIndex;
  property OnGetSelectedIndex;

  // propiedades heredadas de TCustomArbolDirectorios
  property MostrarPadre;
  property CarpetaRaiz;
  property CarpetaActual;
end;
```

Interceptando los eventos

Si queremos que el árbol se comporte de forma ágil, tenemos que ir creando lo distintos nodos bajo demanda, es decir: sólo crear las subcarpetas de aquellos nodos que se vayan expandiendo, y no todas a la vez. En caso contrario, si por ejemplo, intentamos que el componente muestre el árbol de "C:\", tendría que recorrer por completo la estructura de directorios de la unidad, para ir creando todos y cada uno de las carpetas encontradas. Sin embargo, utilizando este método, sólo se recorrerán las carpetas que cuelguen directamente de "C:\", que no serán demasiadas, y todas las que estén por dejado solo se crearán cuando se expanda la carpeta correspondiente. Para conseguir esto, tenemos que interceptar el momento en que se intenta expandir un nodo. Si no fuéramos programadores de componentes, no bastaría con codificar el evento `OnExpanding`, que es precisamente el que se dispara justo antes de expandir un nodo. Bastaría con asignar el evento, e insertar en ese momento los nodos correspondiente.

Sin embargo, los eventos de las clases padre no están ahí para ser usados desde los componentes hijos, sino para que el programador que utilice el componente final pueda asignarlos. La razón de esto es sencilla: dijimos que cada evento era un puntero a una función, por lo que ese puntero solo puede contener una dirección. No debemos utilizar ese puntero en los componentes derivados, ya que, si lo hacemos, estaremos ocupando ese puntero, cuando debería estar disponible para que lo utilice el programador final.

Tenemos un problema ¿no? Si os sirve de consuelo, Microsoft ha tenido este inconveniente en cuenta, a la hora de diseñar el modelo de eventos de los lenguaje de la plataforma .NET. Para solucionarlo, han definido lo que llaman "delegates", que no son más que listas de punteros para almacenar un conjunto de eventos (y no un solo puntero para almacenar un solo evento, como en la VCL). De este modo, podemos asignar distintos eventos un solo "delegate", ejecutándose estos en el orden establecido. Pero, como dijo el autor de Momo: esto es otra historia, y debe ser contada en otra ocasión.

> Un programador de componentes, no puede utilizar los eventos heredados del componente padre, ya que estos eventos solo están disponibles para quien utilice el componente final.

Nosotros, resignados programadores de componentes para Delphi, tenemos que buscar la manera de recibir el evento, sin utilizar ese puntero interno. Por suerte, Borland se dio cuenta de esto a la hora de diseñar la VCL, y nos ofrecen puntos de entrada para estos casos. Veamos: cada vez que se va a lanzar el evento de un componente, se hace a través de un método virtual (o dinámico, que básicamente es lo mismo). Este método, al ser virtual, podemos sobrescribirlo desde los descendientes, y si lo hacemos, se ejecutará nuestro código en vez de código del padre. Así, desde el método que hemos sobrescrito, podemos hacer cualquier proceso, y antes o después llamar al padre (con la instrucción `inherited`) para que ocurra lo que tenía que ocurrir, es decir: que se lance el correspondiente evento de usuario.

De este modo, el problema se reduce a averiguar qué método del padre tenemos que sobrescribir para que todo funcione. Para ello, tendremos que rebuscar en la ayuda de Delphi (en las páginas que documentan el componente `TCustomTreeView`) o incluso en el código fuente de la VCL (que está disponible en el directorio "Source" de nuestra instalación de Delphi).

Para nuestro caso, necesitamos recibir el evento `OnExpanding`, que en `TCustomTreeView` se lanza a través del método dinámico `CanExpand`. Así que nuestro objetivo es sobrescribir este método protegido, y lanzar en él algún método que nos inserte, bajo el nodo que estamos expandiendo, las subcarpetas encontradas. Esto lo haremos a través de un método que llamaremos, por ser originales, `CargarSubcarpetas`.

Carga de datos...

Otro de las tareas que nos queda por ver es cómo cargar los datos iniciales del árbol. Para ello, nos bastaría con un método que vacíe el árbol e introduzca los nodos iniciales, que logicamente, dependerán del valor de las propiedades "CarpetaRaiz" y "MostrarPadre". Además, cada vez que cambien estas dos propiedades, tendremos que llamar de nuevo al método en cuestión, para mostrar el nuevo árbol inicial. Después, cuando el usuario haga clic sobre algún nodo para expandirlo, se desencadenará todo lo que hemos explicado hace un minuto.

Existe un caso especial: cuando la propiedad "CarpetaRaiz" contenga el valor "Mi PC", en vez de mostrar un conjunto de subcarpetas, debemos mostrar el conjunto de unidades lógicas disponibles en nuestro equipo, y a partir de cada una de estas unidades, se desplegarán las subcarpetas correspondientes.

Manejando los bits para buscar las unidades

Como acabamos de decir, necesitamos algún método para averiguar las unidades lógicas disponibles en el sistema. Para ello, debemos echar mano del API Win32, que nos ofrece una función muy interesante: `GetLogicalDrives`. Sin embargo, para poder utilizar esta función tenemos que saber cómo trabajar a nivel de bits, ya que nos retorna un valor de 4 bytes, en el que cada bit individual representará la existencia o ausencia de una unidad. La posición del bit indicará la unidad (el primer bit por la derecha para la unidad A:, el siguiente para la unidad B:, y así sucesivamente), mientras que el valor (0 ó 1) indicará si la unidad está disponible o no.

Para ir recorriendo estos 32 bits, debemos hacer un bucle que recorra las letras desde la "A" a la "Z" (las posibles unidades), e ir verificando en cada vuelta el valor del bit: en la primera vuelta, el primer bit por la derecha, en la segunda vuelta, el segundo bit, en la tercena, el siguiente, etc. Para ello, haremos uso de una técnica para el acceso a bits individuales a través de lo que llamamos una "máscara". Esta técnica se utiliza para averiguar si un bit de una posición concreta tiene el valor 1 ó 0.

Vamos a explicarlo con un ejemplo: supongamos que tenemos el siguiente valor de 8 bits (1 byte): 101100101. Necesitamos averiguar el valor del primer y cuarto bit (la posición de los bits siempre se empieza a contar por la derecha). Para ello, haremos una operación "and" de ese bit con el valor 1. Si el resultado de esa operación es 1, significará que el bit que estamos buscando está activo (tiene el valor 1), y si el resultado es 0, significará que el bit está también a 0.

Por cierto, ya sabéis que el resultado a nivel de bit de una operación "and" es 1 (`true`) cuando los dos operandos son 1. Así que, para averiguar si el primer bit está activo, debemos hacer la operación del listado de la derecha. Como veis, se va operando cada uno de los bits, de forma vertical, dando como resultado otro número en notación binaria. El resultado final (1 si lo convertimos a decimal) nos indica que el primer bit está activo.

Si ahora queremos averiguar el valor del cuarto bit, podéis ver la operación tal y como aparece en el listado de la derecha. En este caso, como veis, el resultado final, en decimal, es 0, lo que nos indica que el cuarto bit no está activo.

Al segundo operando, que contiene un bit en la posición que más nos interesa, se le suele llamar "máscara de bits".

Bien, ya conocemos la base teórica que necesitamos para utilizar la función `GetLogicalDrives`. La implementación final pasa por obtener los 32 bits retornados por la función, e ir recorriendo, a través de un bucle, cada uno de ellos para averiguar su valor. Para ello, necesitaremos una máscara que con la que operar en cada vuelta del bucle. La máscara debe tener a 0 todos sus bits, excepto el bit de la posición de la vuelta que estemos dando, es decir:

1ª vuelta: todos a 0 excepto el primer bit: `00000000000000000000000000000001`
2ª vuelta: todos a 0 excepto el segundo bit: `00000000000000000000000000000010`
3ª vuelta: todos a 0 excepto el tercer bit: `00000000000000000000000000000100`
4ª vuelta: todos a 0 excepto el cuarto bit: `00000000000000000000000000001000`
enésima vuelta: todos a 0 excepto el bit *n*.

Para conseguir esto, debemos hacer uso del operador "shl", que desplaza todos y cada uno de los bits de un número, una posición a la izquierda, rellenando los espacios libres que queden a la derecha con ceros. Así, si desplazamos el valor 11011 a la izquierda, el resultado será 10110, perdiendo el valor del último bit (el cuarto), y añadiendo un cero al primer bit.

Combinando estas dos técnicas, podemos averiguar las unidades existentes, tal y como podéis ver en el siguiente listado:

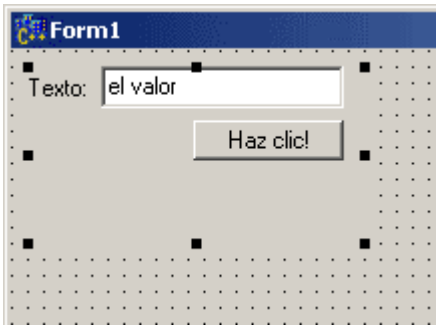
```
procedure BuscaUnidades;
var
   unidades: DWORD;
   mascara: DWORD;
   u: char;
   msg: string;
begin
   unidades := GetLogicalDrives;
   mascara := 1; // comenzamos con el primer bit activo

   msg := 'Unidades válidas: ';

   for u:='A' to 'Z' do
   begin
      // comprobar si está activo el bit
      if (unidades and mascara) <> 0 then
         msg := msg + u + ', ';

      mascara := mascara shl 1; // desplazar hacia la izq todos los bits
   end;

   // quitar la coma final
   msg := Copy(msg, 1, Length(msg) - 2);

   // mostrar el resultado
   ShowMessage(msg);
end;
```

¿Y dónde están mis iconos?

Como sabréis, el `TTreeView` proporciona una propiedad para enlazar un componente `TImageList` con el propio árbol. De esta forma, cuando se necesite mostrar un icono, se utilizarán las imágenes del ImageList, a través de su identificador. Los distintos iconos que mostraremos dentro del árbol, se buscarán a través de los identificadores que podéis ver en la Tabla 1, así que de este modo, podemos cambiar las imágenes del árbol, simplemente cambiando el ImageList asociado.

Heredando de TCustomComponent: otro mundo

Como hemos visto durante todo este ejemplo, heredando de un componente ya existente conseguimos personalizar su comportamiento final. Sin embargo, tenemos otra posibilidad muy interesante: crear un componente en los que se combinen otros componentes visuales (y no visuales), actuando como si fueran uno solo. Esto funcionaría siguiendo la misma filosofía que los objeto `TFrame` de Delphi, es decir: con un simple clic, incluimos en el formulario un componente todo lo complejo que queramos.



Para conseguir esto, tenemos que utilizar la misma técnica que hemos explicado con el ejemplo anterior, aunque heredaremos de otro componente base. En nuestro caso, para crear un componente combinado podríamos heredar de `TCustomComponent`, que es precisamente la clase que nos proporciona la VCL para crear componentes combinados. En el constructor de "nuestro nuevo hijito", procederemos a crear el resto de componentes que aparecerán en su interior. El principal inconveniente (que soluciona el componente `TFrame`) es que tenemos que actuar como si no tuviéramos un entorno de desarrollo, es decir: creando cualquier componente, o asignando los eventos y propiedades, en tiempo de ejecución, todo a través de código y sin ayudarnos del editor de propiedades y formularios del entorno de Delphi.

Para ello tenemos que tener que imitar lo que Delphi hace por nosotros: tener un atributo para cada uno de los componentes internos, que crearemos en el constructor y asignaremos su `Parent` a nuestro `CustomComponent`.

Si necesitamos hacer uso de los eventos de los componentes internos, debemos hacer lo mismo que hace el entorno de desarrollo: definir un método y asignarlo a la propiedad del componente que queramos.
Bueno, más vale una imagen que mil palabras, así que en el siguiente listado tenéis el código que debería tener un componente de este tipo, concretamente un componente que muestra una etiqueta (`TLabel`), un botón (`TButton`), y un

campo de edición (`TEdit`), y en la imagen de la izquierda el resultado de este componente funcionando.

Los ejemplos

Todo el código que hemos explicado durante este artículo podréis descargarlo en forma de componente:

ArbolDirectorios
http://users.servicios.retecal.es/sapivi/src/ArbolDirectorios.zip
Se trata del componente que ya hemos explicado. Permite mostrar estructuras de directorios indicando la carpeta padre, y permitiéndonos navegar por sus subcarpetas. Además, permite crear el típico árbol de "Mi PC", pudiendo explorar las distintas unidades de nuestro equipo. Se incluye también una pequeña aplicación de ejemplo que utiliza el árbol.

ComponenteCombinado
http://users.servicios.retecal.es/sapivi/src/ComponenteCombinado.pas
El código fuente de un componente combinado tal y como hemos explicado durante el artículo. El componente en cuestión (llamado `TMiComponenteCombinado`) muestra una etiqueta de texto (`TLabel`), un cuadro de edición (`TEdit`) y un botón (`TButton`), mostrando el texto del cuadro de edición cada vez que se hace clic sobre el botón.

¡Espero que os sea de utilidad!

Este artículo apareció publicado por primera vez en el número 4 de la revista Todo Programación, editada por Studio Press, S.L. y se reproduce aquí con la debida autorización.