Creación de componentes VCL (I)

Componentes no-visuales

En los últimos años, el desarrollo basado en componentes se ha convertido en una de las técnicas de programación con más éxito. En este artículo vamos a aprender a crear componentes similares a los de la VCL, para que otros puedan utilizar nuestro código en Delphi.

Índice

Introducción

Una de las piedras angulares del desarrollo en entornos RAD, como Delphi, es la programación basada en componentes. En estos entornos, la labor de un programador se parece más a la de un "ensamblador" de piezas de software que la de un "constructor" de software. Con esto conseguimos mayor rapidez de desarrollo, y sobre todo, mayor simplicidad, ya que sólo tenemos que saber cómo "montar" esas piezas para que nuestro programa funcione. Delphi fue uno de los primeros entornos en aplicar con éxito esta filosofía, y hoy en día son muchos los que apuestan por esta idea, como por ejemplo Microsoft con su lenguaje de programación C#.

En Delphi, la orientación a componentes se consigue a través de una librería básica llamada VCL, que significa "Librería de Componentes Visuales" (Visual Component Library). Esta librería, además de proporcionar los componentes más básicos (como botones, etiquetas de texto, cuadros combinados, etc.), nos ofrece los mecanismos para crear nuestros propios componentes.

Pero... ¿qué es un componente?

Para explicar esto debemos conocer, al menos básicamente, la programación orientada a objetos, ya que la programación basada en componentes se apoya sobre ella. Vamos a suponer que todo el mundo sabe lo que son las clases, con sus atributos y sus métodos.

Un componente es una clase de uso específico, lista para usar, que puede ser configurada o utilizada de forma visual, desde el entorno de desarrollo.

La principal diferencia, respecto a una clase normal, es que la mayor parte del trabajo lo podemos hacer de forma visual, con el ratón y ajustando las opciones que se nos ofrece en nuestro entorno.

> ***Programación Orientada a Objetos***
> La programación orientada a objetos, comúnmente POO o en inglés OOP, es un modelo de programación que estructura los programas dando más énfasis a los datos que a los procedimientos. En la programación procedural (la de Pascal, C, Basic, etc.) se utilizan funciones que hacen algo con los datos. Sin embargo en la POO (con C++, Java, Object Pascal, etc.) se utilizan objetos (datos) que hacen cosas. Si queréis aprender más sobre POO, podéis consultar la sección básica, donde explicaremos paso a paso la programación orientada a objetos.

En la programación orientada a objetos, debemos codificar una serie de operaciones, más o menos laboriosas, para preparar los objetos para su uso. Programar estas operaciones requiere su tiempo, su complejidad y pueden ser origen de errores. Si embargo, en la programación basada en componentes, todas estas operaciones las realizamos de forma visual, para así poder dedicar la atención a nuestro problema.

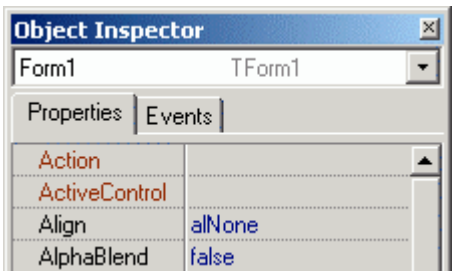
Tipos de componentes

Aunque hay muchos tipos, podemos diferenciar claramente dos grupos: “Visuales” y “No visuales”

Los componentes visuales son aquellos que, al utilizarlos, muestran algún elemento (o dibujo) en la pantalla y es el usuario de nuestros programas el que interactúa con él. El componente es el principal responsable de dibujar en la pantalla lo que sea oportuno, dependiendo de su estado, del valor de sus atributos, etc. Hay muchos componentes de este tipo, como pueden ser los botones (`TButton`), etiquetas de texto (`TLabel`), formas (`TShape`), etc.

Los componentes no visuales son aquellos que no aparecen en la ventana, y se insertan en un formulario para que el programador los utilice. Son más fáciles de programar que los componentes visuales, ya que no tienen ningún tipo de interfaz gráfico. Ejemplos de componentes no visuales podrían ser un temporizador (`TTimer`), un tabla (`TTable`) o una conexión a base de datos (`TConnection`, `TSQLConnection`, etc.).

Propiedades y eventos



Con un primer vistazo al "Inspector de objetos" de Delphi, nos quedan claras dos cosas: un componente tiene propiedades y eventos (según las dos pestañas que vemos en la imagen de la derecha). Estos dos conceptos son nuevos y debemos dominarlos.

Las **propiedades** son datos públicos del componente, muy parecidas a los atributos de una clase, aunque se accede a ellas a través de dos métodos: un método para leer su valor, y otro para modificarlo. Existen propiedades de sólo lectura, en las que podemos consultar pero no modificar su valor, y propiedades de sólo escritura. Por ejemplo, las propiedades “Alto” (Width) y “Ancho” (Height) de un botón permiten que un programador pueda cambiar las dimensiones del componente. Cuando el programador cambia alguna de ellas, el componente debe redibujarse en la pantalla, para mostrar los nuevos cambios. Si miramos la pestaña “Properties” del “Inspector de Objetos”, veremos que cada una de las líneas mostradas es una propiedad del componente.

Los **eventos** son funciones del componente, que se ejecutarán automáticamente cuando ocurra “algo importante”. Un programador puede poner el código que quiera en el evento, para así poder hacer una acción cuando ese “algo importante” ocurra. En realidad, los eventos son punteros a funciones, aunque ya veremos esto en el próximo artículo. Del mismo modo, la pestaña “Events” del “Inspector de Objetos” nos muestra cada uno de los eventos disponibles.

Dado que un componente es una clase como cualquier otra, tenemos además métodos y atributos, aunque no los podamos ver con el “Inspector de Objetos”.

Los **métodos** son funciones, que permiten realizar acciones. Normalmente, se utilizan métodos para dos tareas distintas: realizar algo importante (como

repintar en pantalla, cambiar el foco o algo así), o para establecer el valor de los atributos internos, haciendo algún tipo de comprobación previa. Como las propiedades pueden ser leídas o escritas a través de métodos, a veces es equivalente la llamada a un método y el cambio de una propiedad. Por ejemplo, para mostrar un botón tenemos dos posibilidades:

```
MiBoton.Visible := true;

o bien

MiBoton.Show;
```

Y por último, los **atributos**. Tienen la misma misión que en programación orientada a objetos, es decir: almacenar datos internos al objeto (o clase). En el maravilloso mundo de los componentes, los atributos siempre son internos y de uso privado, y debemos utilizar las propiedades para que un programador pueda leer o establecer un dato.

Sabiendo esto, podemos decir que la principal misión del programador de componentes es definir un grupo de propiedades, métodos y eventos para que otros programadores puedan utilizar el componente de forma sencilla y rápida.

¿Y cómo se hace todo esto?

Bueno, ya sabemos lo más importante. Ahora vamos a ponernos manos a la obra y vamos a desarrollar nuestro primer componente. Para empezar por lo más fácil, vamos a realizar un componente no-visual, que nos permita convertir el valor de distintas monedas. Nos puede resultar útil si vamos a desarrollar una aplicación de cambio de divisas, financiera, etc. Llamaremos a nuestra pequeña creación `TConversorMonedas`.

Para poder hacer una conversión de divisas necesitamos tres datos:

- El valor origen, que será la cantidad que deseamos convertir. Por ejemplo: 1.
- La divisa en que está dado ese valor. Por ejemplo: euros.
- La divisa que queremos obtener. Y siguiendo con el ejemplo: pesetas.

En resumen, en el ejemplo queremos convertir 1 euro en pesetas, por lo que el componente debe informarnos de que el resultado es 166,386.

Como resultado de la operación obtendremos un valor ya convertido a la divisa indicada. Pero vamos a empezar por el principio.

El esqueleto de un componente

Lo más normal es que cada componente se codifique en su propia unidad, aunque esto en absoluto es imprescindible, y muchas veces es necesario codificar varios componentes en una misma unidad (la VCL de Borland lo hace así). En nuestro caso vamos a crear la unidad `ConversorMonedas.pas` para definir el componente.

Ya hemos dicho que un componente es una clase como cualquier otra, aunque no hemos dicho que debemos heredarla a partir de la clase `TComponent`, o cualquiera de sus descendientes. Esta clase nos proporciona todos los métodos y propiedades para que el componente pueda ser manejado y registrado dentro del entorno de desarrollo.

Así, con lo que ya sabemos podemos escribir nuestra primera aproximación del componente `TConversorMonedas`, en la que definiremos una clase heredada de `TComponent`:

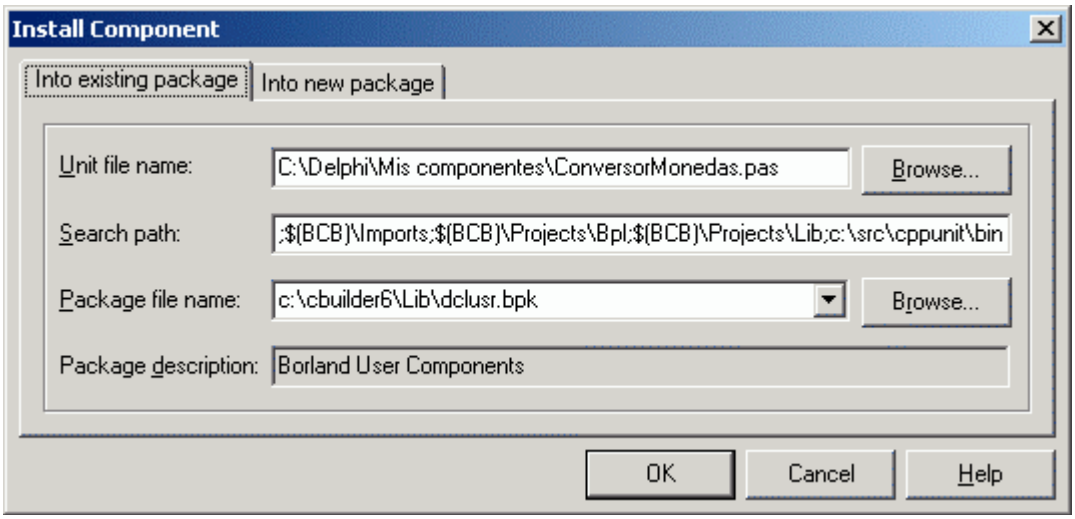
```
type
        TConversorMonedas = class(TComponent)

        end;
```

Esto es un componente, aunque por ahora no hace nada interesante. Para probarlo debemos registrarlo en el entorno de desarrollo y así podremos verlo dentro de la paleta de componentes. Esto se hace a través del procedimiento `RegisterComponents` definido dentro de la unidad `Classes`. Es fácil de utilizar: debemos crear un procedimiento llamado `Register` en la misma unidad que el componente. En este procedimiento, tenemos que hacer una llamada al procedimiento `RegisterComponent`, indicando, en este orden, la solapa en la que mostrar el componente, y la clase que lo representa. La unidad completa, con el procedimiento `Register`, quedaría como vemos en el siguiente listado:

```
unit ConversorModenas;

interface

uses classes;

type
        TConversorMonedas = class(TComponent)

        end;

        procedure Register;

implementation

procedure Register;
begin
        RegisterComponents('Mis componentes', [TConversorMonedas]);
end;

end.
```

Bien, ya podemos registrar nuestro componente, a través de la opción de menú Component - Install Component. En la ventana que se mostará, seleccionamos la unidad `ConversorMonedas.pas` en el campo “Unit file name” y hacemos clic sobre el botón OK.

Acto seguido se procederá a compilar el componente y si todo va bien, aparecerá un nuevo icono en la paleta de componentes, dentro de una nueva solapa llamada "Mis componentes".



Estos pasos que hemos dado son comunes a todos los componentes que hagamos en un futuro, así que debemos entenderlos bien.

Resumiendo, lo que hemos hecho ha sido definir la clase del componente y registrarlo en el entorno de desarrollo.
Por cierto, esto mismo se puede hacer de forma automática a través del menú Component - New component, que nos creará esqueleto básico de un componente. De todas formas, es bueno saber hacer las cosas a mano, y luego aprovechar las herramientas que nos ofrece el entorno.

Definiendo propiedades

Ahora es cuando vamos a empezar a personalizar nuestro componente. Lo primero que tenemos que hacer es definir las propiedades que el usuario podrá manipular. Como antes hemos visto, nos valdrá con definir un valor a convertir, un valor convertido y los tipos de monedas de ambos valores.

Estos tipos de moneda tendrán que ser un valor concreto dentro de un conjunto de valores soportados, así que nos aprovechamos del tipo de dato enumerado de Pascal. Declararemos el tipo de dato `TTipoMoneda`, que define todas las monedas soportadas.

```
TTipoMonedas = (tmEuro, tmDolar, tmYen, tmPeseta);
```

Una vez que tenemos esto, podemos definir los atributos que almacenarán estos valores dentro del componente. A continuación mostramos sólo la definición del componente:

```
TConversorMonedas = class(TComponent)
private
        FValorConvertir: float;
        FValorConvertido: float;

        FMonedaConvertir: TTipoMoneda;
        FMonedaConvertido: TTipoMoneda;
end;
```

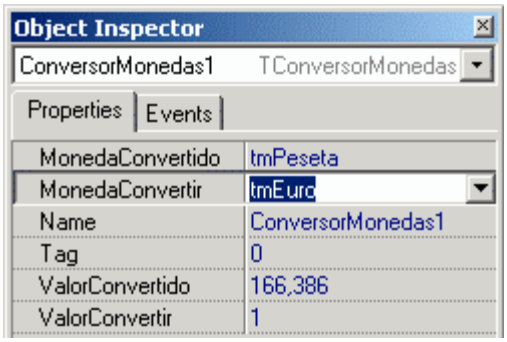
El componente ya es capaz de almacenar unos valores, aunque como lo hemos definido dentro de la sección *private*, nadie podrá modificar ni consultar estos valores. Para ello debemos definir unas propiedades, dentro de la sección *published*, para poder ser modificadas desde el entorno de Delphi.

La sección *published* contiene todas aquellas propiedades y eventos que serán mostrados en el Inspector de Objetos de Delphi. Además, se puede acceder a ellos tal y como si fuesen públicos:

```
TConversorMonedas = class(TComponent)
private
        FValorConvertir:  float;
        FValorConvertido: float;

        FMonedaConvertir:  TTipoMoneda;
        FMonedaConvertido: TTipoMoneda;

published
        property ValorConvertir: float
                read  FValorConvertir
                write FValorConvertir;

        property ValorConvertido: float
                read FValorConvertido;

        property MonedaConvertir: TTipoMoneda
                read  FMonedaConvertir
                write FMonedaConvertir;

        property MonedaConvertido: TTipoMoneda
                read FMonedaConvertido
                write FMonedaConvertido;
end;
```

Vamos a fijarnos en las líneas dentro de la sección *published*. Se trata de definiciones de propiedades, utilizando la palabra reservada *property*. Con esto definimos una serie de datos que serán leídos y escritos de forma indirecta. En nuestro caso, decimos que cada vez que se lea el valor de la propiedad `ValorConvertir`, se accederá al atributo privado `FValorConvertir`, y cada vez que se escriba, se almacenará en dicho atributo. Lo mismo ocurre con las propiedades `MonedaConvertir` y `MonedaConvertido`. Sin embargo, la propiedad `ValorConvertido` tiene algo especial: si prestamos un poco de atención a su declaración, veremos que tienen una cláusula `read` pero no tiene la correspondiente cláusula `write`. Esto es debido a que sobre esta propiedad se puede leer pero no se puede escribir, es decir: que es de solo lectura.

Si compilamos y volvemos a instalar el componente, podremos ver cómo las nuevas propiedades aparecerán en el Inspector de objetos. Además, en las propiedades de tipo `TTipoMoneda`, solo se podrá seleccionar entre aquellos valores que permite el tipo enumerado.



Por cierto, si alguno de vosotros no puede ver la propiedad `ValorConvertido`, que no cunda el pánico. El Inspector de objetos de Delphi, por defecto no muestra las propiedades de sólo lectura. Si utilizáis la versión 6 (o posterior), basta con acceder a la opción Tools - Environment Options - Object Inspector, y marcar la opción "Show read only properties". En las versiones anteriores no podremos ver este tipo de propiedades.

Y por último, lo que tenemos que hacer es asegurarnos de que los atributos tienen los valores adecuados. Esta operación vamos a hacerla a través de un método público, llamado Convertir, desde el que se harán todos los cálculos necesarios para hacer la conversión de `ValorConvertir` a `ValorConvertido`.
La definición del método es la siguiente:

```
public
        procedure Convertir;
```

Nos debemos asegurar de que, cada vez que se cambie el valor origen (`ValorConvertir`), se cambie el valor destino (`ValorConvertido`), así como al cambiar los tipos de divisa.

Esto lo podemos hacer asignando métodos a las operaciones de escritura de las propiedades. De este modo, conseguimos ejecutar un código cada vez que alguien intente almacenar un valor en una propiedad. Para ello, debemos cambiar la definición de las propiedades y añadir los tres métodos que vemos a continuación:

```
protected
        procedure SetValorConvertir(value: float);
        procedure SetMonedaConvertir(value: TTipoMoneda);
        procedure SetMonedaConvertido(value: TTipoMoneda);

published
        property ValorConvertir: float
                        read  FValorConvertir
                        write SetValorConvertir;

        property ValorConvertido: float
                        read FValorConvertido;

        property MonedaConvertir: TTipoMoneda
                        read  FMonedaConvertir
                        write SetMonedaConvertir;

        property MonedaConvertido: TTipoMoneda
                        read  FMonedaConvertido
                        write SetMonedaConvertido;
```

Y en la codificación de los métodos "Set" lo que debemos hacer es volver a hacer la conversión cada vez que se modifique algo, como vemos a continuación:

```
procedure TConversorMonedas.SetValorConvertir(value: float);
begin
        if value <> FValorConvertir then
        begin
                FValorConvertir = value;
                Convertir;
        end;
end;

procedure TConversorMonedas.SetMonedaConvertir(value: TTipoMoneda);
begin
        if value <> FMonedaConvertir then
        begin
                FMonedaConvertir = value;
                Convertir;
        end;
end;

procedure TConversorMonedas.SetMonedaConvertido(value: TTipoMoneda);
begin
        if value <> FMonedaConvertido then
        begin
                FMonedaConvertido = value;
                Convertir;
        end;
end;
```

Ahora lo único que nos queda es compilar e instalar de nuevo el componente, y jugar con él. Podremos ver cómo, en el Inspector de objetos, se cambiará el contenido de la propiedad `ValorConvertido` cada vez que cambiemos alguna de las demás propiedades. En ejecución podremos utilizar el componente igual que cualquier otro, leyendo o escribiendo sobre sus propiedades.

Los ejemplos

Ya para terminar tan sólo queda deciros que podéis descargar el código fuente del `TconversorMonedas`

http://users.servicios.retecal.es/sapivi/src/ConversorMonedas1.pas

Podéis instalarlo en vuestro Delphi o simplemente estudiar y jugar con el código.

Autor: JM

Este artículo apareció publicado por primera vez en el número 1 de la revista Todo Programación, editada por Studio Press, S.L. y se reproduce aquí con la debida autorización.