SISTEMA DE PLUGINS EN DELPHI (Parte I")

Germán Estévez -Neftalí- Contact@

INTRODUCCIÓN

Pues ha llovido mucho desde la primera parte de este "articulillo"; Por lo que he visto en los fuentes del ejemplo, lo empecé hace aproximadamente hace 2 años, así que eso es lo que lleva de retardo… ;-)

En ese primer artículo se daba una visión general de lo que podía ser un sistema de PlugIns. Unas ideas generales y algo de código para empezar. Quedaba en el tintero profundizar un poco más en el tema y ver un ejemplo un poco más "práctico" que pudiera servir en una aplicación real. Eso es lo que he intentado tratar en esta segunda parte, centrandome en un sistema de "**PlugIns Homogéneos**", con carga bajo petición (por parte del usuario).

TIPOS DE PLUGINS

Como ya vimos en la primera entrega, podemos dividir los plugIns en dos tipos según las tareas que desempeñan en una aplicación. Así podemos hablar de plugIns o grupos de ellos y catalogarlos como homogéneos si la tarea que realizan es similar o catalogarlos como heterogéneos (no-agrupados) si las tareas que desarrollan son independientes y no "agrupables" según su funcionalidad.

- Grupos homogéneos de PlugIns
- PlugIns heterogéneos

Esta división no sólo es conceptual en función de las características y desempeños de cada uno, sino que afecta directamente a la estructura con que se diseñarán estos elementos. Así, los PlugIns que pertenezcan a un grupo homogéneo tendrán estructura similar y un punto de entrada común (formulario base o procedimiento de ejecución). Mientras que los heterogéneos posiblemente no tengan una estructura común y la forma se ejecutarlos sea más "tosca" y menos "integrada" que los anteriores.

PLUGINS HOMOGENEOS

En este artículo vamos a tratar más profundamente esta variante de plugIns. Como ya hemos comentado se trata de plugIns con una estructura similar, aunque con variaciones en su funcionalidad. Tal vez con un ejemplo se vea más claro.

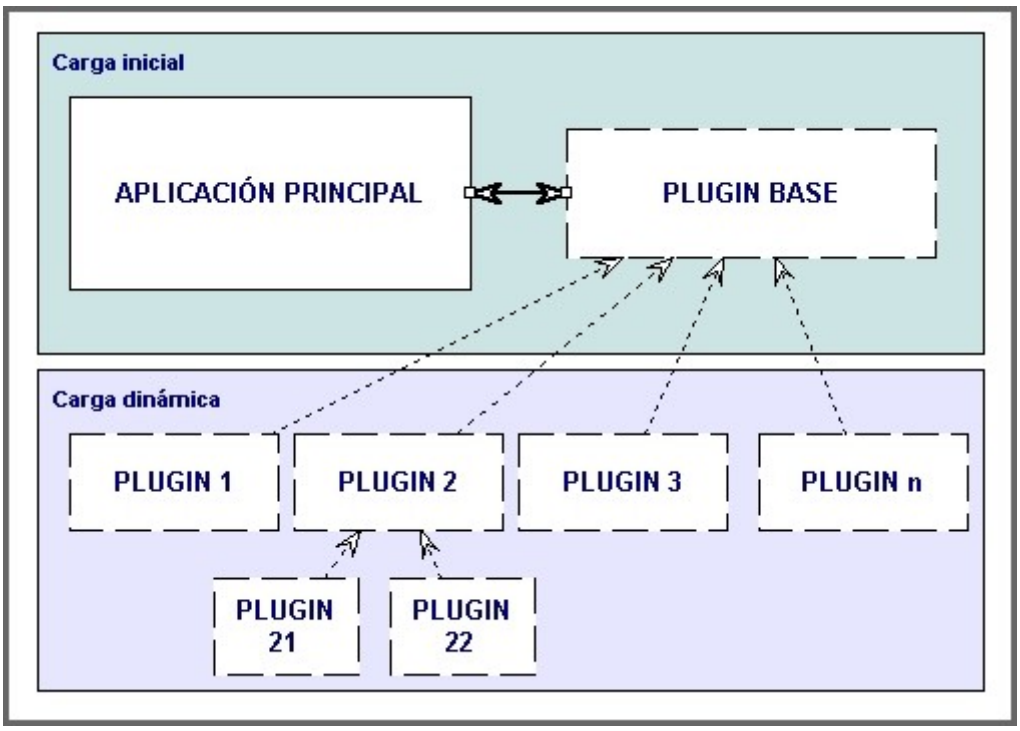
Tomemos como grupo homogéneo de PlugIns; Los efectos aplicables a una imagen dentro de un programa de diseño. A partir de una imagen podemos desarrollar plugIns que efectúen un determinado "cambio" de forma que la imagen resultante sea diferente a la original. La estructura y los parámetros de todos ellos parece claro que serán similares o idénticos. Todos ellos toman una imagen inicial y a partir de unos parámetros la modifican, para devolver una imagen de salida.

ESTRUCTURA FÍSICA

Para trabajar con esta estructura de plugins, utilizaremos un sistema de carga dinámica. Los plugins de programarán utilizando packages (BPL) con una estructura común y dependiendo de un package principal que contiene la Clase Base. Todos los plugins derivarán (heredarán) de una clase base que estará programada en el package principal.

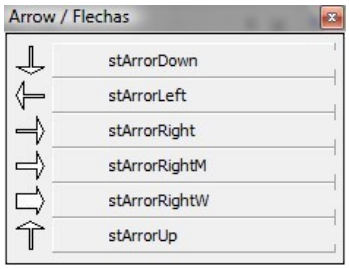
Al cargar la aplicación se carga (puesto que está "linkado" de forma estática -utilizando la clausula USES-) también el package correspondiente a la clase Base. Esto da acceso a todos los métodos que estén definidos en la clase base (y en los derivados) desde el programa principal.

El resto de packages se cargan de forma dinámica y todos deben derivar (sus clases) de la Clase Base programada en el package Base.

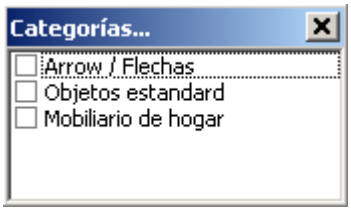


PROTOTIPO

El prototipo que vamos a realizar para ilustrar el artículo simula un programa para realizar gráficos y diagramas simples. El programa utilizará un sistema de plugIns para añadir bibliotecas de objetos que puedan añadirse a los gráficos. Cada pluging (BPL) añade una **nueva categoría** de elementos y cada categoría implementa **uno o varios objetos**.



Todos los objetos que implementa una categoría derivan de una Clase Base (**TShapeExBase**) y esta clase base se implementa en un package que está **linkado estáticamente a la aplicación principal** (se carga siempre al arrancar la aplicación) y es obligatorio que exista, de otra forma la aplicación fallaría al ejecutarse.



En la primera imagen que se ve a la derecha, vemos la ventana correspondiente al Plugin de "Arrows"; Aquí implementa la clase **TShapeExArrow** (que deriva de **TShapeExBase**) y en

esta clase se han programado los objetos que se ven en la imagen.

En nuestro ejemplo para este artículo **se cargan los plugIns bajo petición**. Es decir, en una primera pasada la aplicación revisa la existencia de PlugIns y detecta todos los ficheros presentes. Muestra una ventana con los plugns disponibles y la descripción de cada uno de ellos y a medida que el usuario los selecciona se cargan de forma dinámica. Imagen de la derecha.

El código de la carga es el siguiente:

```
   ...
1    // Comprobación
2    if not FileExists(AName) then begin
3      _mens(Format('No se ha podido cargar el package
4 &lt;%s&gt;;' +
5           'No existe en disco.',[AName]));
6      Exit;
7    end;
8
9    // Cargar
10   hndl := LoadPackage(AName);
11   desc := GetPackageDescription(PChar(AName));
12   Result := hndl;
13
14   // Acceder a la clase del menu
15   pName := ChangeFileExt(ExtractFileName(AName),
16 '');
17   b := ExClassList.Find(pName, i);
18   // Encontrada?
19   if (b) then begin
20     AClass := TPersistentClass(ExClassList.Objects[i]);
     end;
```

CLASE BASE (TShapeExBase)

La clase base para nuestro sistema de plugins se llama TShapeExBase. Esta clase sirve como punto de partida para todas las demás. Además de contener los métodos comunes a todos los plugins nos permitirá acceder desde la aplicación principal a todas las funciones de los plugins. Para ello los métodos importantes estarán definidos en esta clase y luego sobreescritos (override) en las clases derivadas.

```
1  {: Clase base lapa las clases implementadas en los plugins.}
2   TShapeExBase = class(TShape)
3   private
4     FShapeEx: string;
5
6     // Marca el tipo de Shape
7     procedure SetShapeEx(const Value: string); virtual;
8
9   protected
10    W, H, S: Integer;
11    X, Y:Integer;
12    XW, YH:Integer;
13    W2, H2, W3, H3, H4, W4, H8, W8, W16, H16:Integer;
14
15    // Método de pintado
16    procedure Paint; override;
17    procedure CalculateData();
```

```
18
19    // PROCEDIMIENTOS DE INFORMACION
20    //.....................................
21    // Autor del package
22    function Autor():string; virtual; abstract;
23    // Versión del Package
24    function Version():string; virtual; abstract;
25    // Fecha de creación
26    function FechaCreacion():TDate; virtual; abstract;
27
28  public
29    // constructor de la clase
30    constructor Create(AOwner: TComponent); override;
31    // destructor de la clase
32    destructor Destroy; override;
33
34  published
35    // Tipo de Shape
36    property ShapeEx: string read FShapeEx write SetShapeEx;
37  end;
```

En nuestro caso es una clase sencilla. La función implementa el dibujo de componentes derivados de un TShape en pantalla.

La propiedad **ShapeEx** es la más importante, e indica el tipo (identificador) de la figura. Equivalente a lo que en los **TShape** son los valores **stRectangle**, stEllipse, stSquare,…

En nuestra clase no puede ser un elemento tipificado como lo es en TShape, puesto que los nuevos plugins irán añadiendo elementos a esta propiedad que a priori no conocemos.
Se añaden también procedimientos de información acerca del plugin como pueden ser el Autor, la fecha de creación o la versión.

El método **Paint**, que para la clase base está vacío, en las clases derivadas será donde se implementen las instrucciones de pintado para cada uno de los elementos.

Finalmente la clase Base implementa el procedimiento **CalculateData** y al utiliza algunas variables en la parte protected, que precalculan datos y los ponen a disposición de las clases derivadas (protected), para facilitar la implementación del método Paint y dar acceso a medidas ya precalculadas.

En la clase Base además se definen dos Listas (TStringList) que nos servirán de apoyo a la hora de acceder a los diferentes objetos de los plugIns; Tanto para las clases, como para los Shapes definidos en cada clase.

```
1  //: Lista de clases registradas en los packages dinámicos
2  ExClassList:TStringList;
3  //: Lista de objetos registrados en una clase (tipos de Shapes)
4  ExShapeList:TStringList;
```

En la primera añadiremos **la referencia a la Clase** y el String correspondiente al nombre del package y en la segunda, para cada Shape implementado en la Clase, su valor de la **propiedad ShapeEx** (comentada anteriormente) y el apuntador a su clase.

De esta forma, por ejemplo, el PlugIn que implementa la clase **TshapeExArrow** que corresponde a la imagen que se ve más arriba, añadirá en las lista los siguientes valores:

```
1  // Registrar los tipos
2    ExShapeList.AddObject('stArrorRight', Pointer(TShapeExArrow));
3    ExShapeList.AddObject('stArrorRightW', Pointer(TShapeExArrow));
```

```
4   ...
5   // registrar la clase
6   ExClassList.AddObject('PlugArrows', Pointer(TShapeExArrow));
```

En las líneas anteriores podemos ver que el plugIn PlugArrow (1) tiene implementada la clase TShapeExArrow (1), y que dentro de esta clase hay 6 objetos diferentes de tipo ShapeEx; Cuyos identificadores son: stArrorRight, stArrorRight, stArrorRightM, stArrorLeft, stArrorUp y stArrorDown.

CLASES DERIVADAS

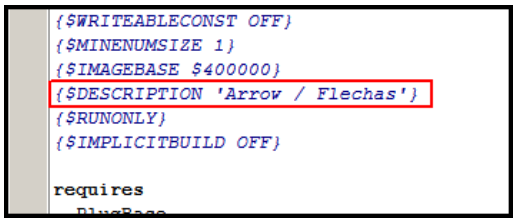
Tal y como está diseñada la estructura, las clases derivadas de la clase Base (TShapeExBase) deben redefinir el método **Paint** para definir cómo se define cada uno de los objetos de esa clase.

SISTEMA DE CARGA/DESCARGA

El sistema de carga es simple y lo único que hace de especial en este caso es comprobar primero si el package ya ha sido cargado, y si no es así llama a la función **CargarPackage** del formulario principal, utilizando el nombre del fichero.

Podemos ver por pasos y comentar qué hace esta función:

```
1   // Cargar
2   hndl := LoadPackage(AName);
3   desc := GetPackageDescription(PChar(AName));
4   Result := hndl;
```



En primer lugar (una vez hemos comprobado que el fichero existe) cargamos el package a partir de su nombre. Una vez cargado obtenemos la Descripción.

Para ello se llama a la función **GetPackageDescription** que se encuentra en la Unidad SysUtils.pas y que develve el valor almacenado en el DPK junto a la directiva **{$DESCRIPTION}** o **{$D}** que permite almacenar hasta 255 caracteres.

Todos los packages cuentan con una sección de INITIALIZATION donde añaden a las lista de clases (**ExClassList**) y a la lista de Shapes (**ExShapeList**) los elementos que ese package implementa. Estas dos clases son importantes puesto que nos facilitan mucho el trabajo a la hora de realizar todo tipo de operaciones con los elementos de cada packages. Además se registra la clase utilizando el método RegisterClass de Delphi. Por ejemplo, el package de "Arrows" contiene esta sección de **INITIALIZATION**:

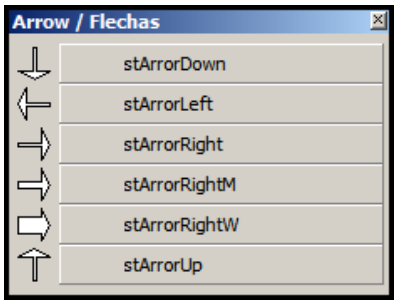
```
1   //=====================================================
2   ============
3   //
4   // I N I T I A L I Z A T I O N
5   //
6   //=====================================================
7   ============
8   initialization
9     // Registrar la clase del form
```

```
    RegisterClass(TShapeExArrow);
    // Registrar los tipos
    ExShapeList.AddObject('stArrorRight', Pointer(TShapeExArrow));
    ExShapeList.AddObject('stArrorRightW', Pointer(TShapeExArrow));
    ExShapeList.AddObject('stArrorRightM', Pointer(TShapeExArrow));
    ExShapeList.AddObject('stArrorLeft', Pointer(TShapeExArrow));
    ExShapeList.AddObject('stArrorUp', Pointer(TShapeExArrow));
    ExShapeList.AddObject('stArrorDown', Pointer(TShapeExArrow));
    // registrar la clase
    ExClassList.AddObject('PlugArrows', Pointer(TShapeExArrow));
  //=====================================================
    ============
```

Lo siguiente que vamos necesitamos, una vez que tenemos cargado el package, es crear la clase que se implementa en el package; Una vez hecho esto ya tendremos total acceso a los métodos que necesitemos y realmente ya habremos conseguido nuestro objetivo.

Para crear la clase utilizamos la lista de clases (**ExClassList**) que hemos comentado en el párrafo anterior y que hemos rellenado en la sección de inicialización; Otra opción también viable es utilizar **GetClass** de Delphi mediante RTTI junto con el nombre de la clase registrada (TShapeExArrow). También funcionaría, aunque en este caso, por comodidad, hemos utilizado estas listas auxiliares.

```
1 // Crear la clase
2   b := ExClassList.Find(pName, i);
3   // encontrada?
4   if (b) then begin
5     AClass := TPersistentClass(ExClassList.Objects[i]);
6
7     // OTRA OPCIÓN:
8     BClass := GetClass('TShapeExArrow');
9   end;
```



Para finalizar y después de haber realizado unas comprobaciones, llamamos al método **CargarCategoria**, que crea de forma dinámica la ventana asociada a esa categoría (con la descripción) y también crea el elemento individual asociada a cada Shape implementado en esa clase.

En este punto ya hemos hecho uso de todo lo implementado en ese package, puesto que ya hemos creado un objeto de todos los implementados.

```
    // Cargar los objetos de ese plugIn
    CargarCategoria(AClass, desc);
```

En este ejemplo, no descargamos los packages, puesto que los necesitamos para seguir trabajando con los objetos que tenemos en pantalla, lo que hacemos realmente es ocultar la ventana. Si la operación que desempeña el package no necesita que posteriormente esté cargado, bastaría con descargarlo utilizando **UnloadPackage**.

Hasta aquí las descripción de todo el proceso. Junto con el artículo os adjunto el ejemplo completo y bastante comentado. Es sencillo, pero muestra a la perfección el manejo práctico de este tipo de ficheros.

Espero que haya quedado claro y si hay comentarios o sugerencias, ya sabéis.
¡¡Disparad!! ;-D

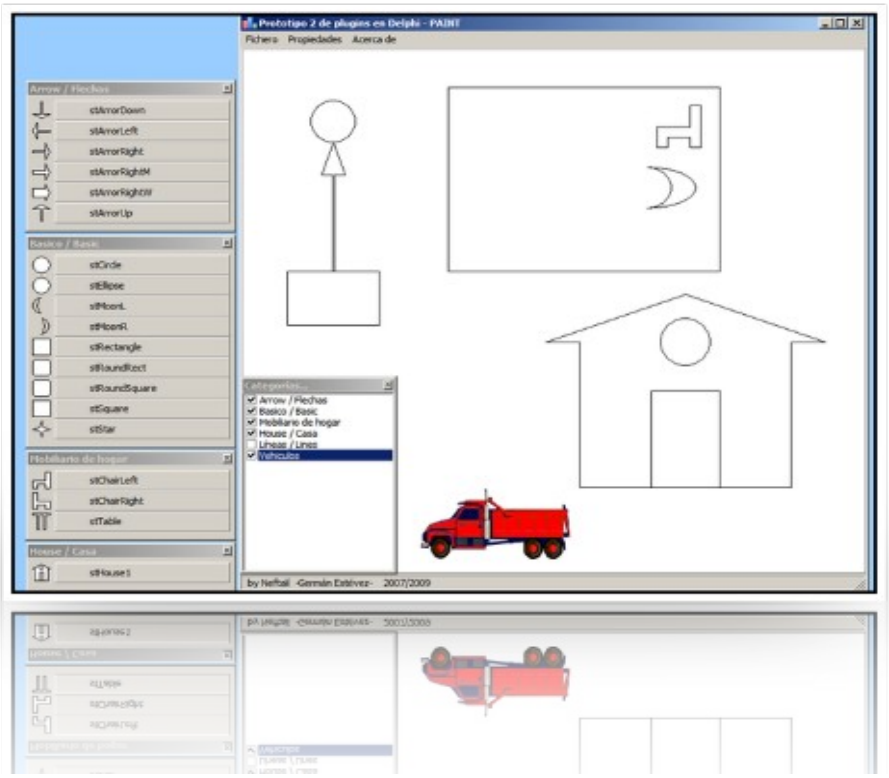


Imagen del programa de ejemplo.

El código del ejemplo se puede descargar desde mi web junto con los binarios (EXE + BPL's) en paquetes separados.