SISTEMA DE PLUGINS EN DELPHI (Parte I)

Germán Estévez -Neftalí**-** Contact@

INTRODUCCIÓN

Más o menos todo el mundo que está metido un poco en informática y más concretamente en programación tiene una idea de qué es un plugin. ¿Cómo definirlo? Tal vez las primeras palabras que se me ocurren son "añadido" o "complemento". Si buscamos la traducción del inglés veremos que significa algo así como "enchufar" o "enchufable". Con ambas cosas ya podemos dar una aproximación en programación.

Un plugin (plug-in, addin, add-in, add-on, addon,...) podemos definirlo como:
"Un programa o librería que permite 'enchufarse' o 'añadirse' a otra aplicación y que consigue 'complementarla' o darle una 'funcionalidad añadida'.

Como ventaja podemos decir que un plugin nos proporciona una aplicación más modular; Además podemos hacer extensible una aplicación sin "tocar" el programa original. Si los plugins se cargan y descargan dinámicamente tenemos un "plus" en flexibilidad.

Para desarrollar un sistema de plugins debemos definir unas cuantas premisas:

- Tipos de plugins
- Tareas que vamos a asignar a los plugins
- Estructura que van a tener nuestros plugins.

QUESTIONES PRINCIPALES

Pensando en desarrollar un Sistema de Plugins en Delphi se me plantean unas cuantas dudas o cuestiones antes de comenzar con el proceso.

(1) ¿Qué ficheros voy a utilizar para desarrollar los plugins?
Se me plantean dos alternativas. Utilizar DLL's o utilizar BPL's.
(2) ¿Qué tareas van a desarrollar mis plugins?
Genéricas o específicas.
(3) ¿Cómo se van a cargar mis plugins?
Inicialmente se cargarán los disponibles o los que el usuario decida, se cargarán "bajo demanda" cuando se requiera su acción o puedo "mezclar" ambos tipos según el tipo de plugin que necesite.
(4) ¿Cómo se van a integrar y van a interactuar con la aplicación principal?
Podemos colocarlos/ubicarlos en un menú específico o podemos intentar "integrarlos" en el menú general de la aplicación (esta segunda opción seguramente nos complicará la programación).

TIPO DE FICHEROS PARA PROGRAMAR PLUGINS

Principalmente encuentro dos opciones, usar **DLL's** (Dynamic Link Libraries) o usar **BPL's** (Borland Package Library).

Por una parte las DLL son más universales, se pueden programar DLL en muchos lenguajes y esas DLL se podrían utilizar como plugins de la aplicación sin necesidad de estar hechos en Delphi. Por otro lado, las BPL son como "DLL ampliadas"; Poseen más funcionalidades que las DLL estándar y para nuestro ejemplo son más adecuadas, ya que nos darán más potencia combinadas con RTTI (Runtime Type Information).

TAREAS DE LOS PLUGINS

Se me ocurren dos tipos de tareas que puedan desarrollar los plugins creados para un sistema. Por un lado, características heterogéneas que podamos añadir a una aplicación, plugins independientes para tareas independientes. Interesante para poder "ampliar" nuestras aplicaciones y añadir nuevas funcionalidades (1). Por otro lado pienso, en plugins con variantes de una misma operación, de forman uno o varios grupo homogéneos que dan (a nivel de grupo) un funcionalidad (2).

(1) Pienso en esta caso en una aplicación a la que podemos añadir un plugin que "permite hacer backups sobre CD/DVD de los datos", un plugin que "añade la funcionalidad de compresión de ficheros" o un plugin que "añade la conexión con un webservice para realizar un trabajo extra". Y así hasta donde llegue nuestra imaginación.

(2) Pienso aquí en un grupo de plugins que permitan exportar datos a un fichero, por ejemplo, cada plugin para un formato diferente (XML, PDF, XLS, HTML, CSV,...) o un grupo de plugins para aplicar efectos a una imagen cada plugin para un efecto diferente.

PROCEDIMIENTO DE CARGA DE LOS PLUGINS

Básicamente se me ocurren dos procedimientos de carga para trabajar con plugins, es escoger uno, otro o ambos dependerá de las tareas que deban realizar estos "añadidos", del consumo de memoria necesario, de periodicidad con que se utilicen,...

Una primera opción puede ser la de cargar todos los plugins al iniciar. No es muy complicado revisar todos los plugins disponibles y realizar una carga secuencial de ellos. Quedan en memoria dispuestos para cuando se necesiten. Ocupan espacio en memoria, pero una vez cargados nos podemos olvidar de ellos.

La segunda opción, un poco más sofisticada, puede ser la de cargar los plugins bajo petición o demanda. Es decir, cargarlos cuando se necesitan y descargarlos cuando ya no hace falta. El consumo de memoria, en este caso, es menor, aunque el trabajo de carga y descarga se hace cada vez. Permite también mayor flexibilidad, ya que podrían añadirse nuevas características "en caliente", es decir, sin necesidad de reiniciar la aplicación.

La decisión de escoger la una o la otra dependerá del tipo de plugins que necesitemos, y se deberá tener en cuenta:

- La frecuencia de utilización; Si se utilizan muy a menudo será mejor el primer método.
- El tamaño de los plugins; Para plugins muy grandes o que consuman mucha memoria tal vez sea mejor el segundo método.
- Tiempo de respuesta necesario. Si esto es muy importante habrá que tener en cuenta que el segundo método es más lento que el primero. En el momento de utilizarlo con el segundo tardaremos más, pero al cargar la aplicación inicialmente penalizará más el primero.
- Sencillez y simplicidad a la hora de programar.
- ...

INTEGRAR PLUGINS CON EL PROGRAMA PRINCIPAL

Diferentes soluciones se me plantean a este problema. Las más sencillas de implementar, son también las más "toscas" de cara al usuario.

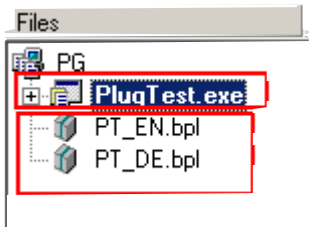
Dejando de lado plugins que no tengan acceso desde un menú o desde una barra de herramientas, y centrándonos en el resto, podemos llegar a conseguir diferentes niveles de "perfeccionamiento".

Desde el más básico, que puede ser el de mostrar todos los plugins disponibles en una ventana o menú creado expresamente para ello, hasta conseguir integrar las acciones de los plugins que se cargan, en los menús que presenta la aplicación; De tal forma que el usuarios no sepa distinguir entre las acciones que son propias de la aplicación principal de las que son "añadidas" por plugins.

CREANDO UN PROTOTIPO...

Una vez que tenemos las ideas claras (o eso creemos :-)) vamos a probar a generar nuestro primer "prototipo"; Nos servirá para asentar nuestras ideas e introducir la tecnología con la que vamos a trabajar. Se trata de una variante del prgrama "**Hola Mundo!!**". Nuestra variante añade plugIns para diferentes idiomas.

La estructura del proyecto se puede ver en el gráfico de la derecha. Incluye el programa principal (**PlugTest**) y dos packages que representan los dos PlugIns que vamos a programar inicialmente (**PT_EN** y **PT_DE**).



En nuestro caso los dos PlugIns lo que hacen es devolver la traducción del texto "Hola Mundo!!" en diferentes idiomas. Os animo a programa plugIns para nuevos idiomas a forma de ejercicio para seguir este manual.

PREMISAS

Las premisas de nuestro sistema de plugins versión 1.0 serán las siguientes:

- Nuestros plugins serán BPL's creadas mediante Borland Delphi.
- La carga de los plugins se realizará en el momento de ejecutarlos, y se descargarán una vez acabada la tarea concreta.
- Todos los plugins deben estar situados en un directorio llamado **AddIn**, que estará dentro del directorio de la aplicación principal.
- El acceso a los plugins se realizará mediante un menú en la aplicación principal.

Para este primer proyecto vamos a trabajar con plugins de traducción. En este caso, trabajaremos con un grupo homogéneo; Cada uno de ellos introducirá un nuevo idioma de traducción de nuestro texto estrella: "Hola mundo" (no podía ser otro).

ESTRUCTURA DEL PLUGIN

Sencilla; Necesitamos un "añadido"(plugin) que dado un texto ("Hola mundo!!"), nos devuelva la traducción de este a otro idioma. Además dotaremos a los plugins de métodos de información, como la versión, el autor y el idioma de traducción.

La nomenclatura para nuestros plugins será la PT_ES, PT_EN, PT_FR,... y así sucesivamente. Un prefijo común y dos letras para designar el idioma.

Debemos crear un Package en Delphi que contenga un formulario. La clase que define el formulario del PlugIn es la siguiente:

```
  //: Clase que define el formulario de nuestro plugIn.
  TTradPlug = class(TForm)
  published
    //: Procedimiento que devuelve el autor
    function Autor():string;
    //: función que deuelve la versión
    function Version():String;
    //: función de traduccion
    procedure Traduce();
    //: función que deviuelve el idioma.
    function Idioma():string;
  end;
```

El método **Traduce** es el que devuelve el texto "Hola Mundo!!" traducido a un idioma diferente, mientras que el resto de métodos son complementario e informativos del plugIn.

La implementación de los métodos es trivial en este ejemplo ya que sólo devuelven cadenas de texto, salvo el de traducción que muestra el texto en pantalla.

```
//: Procedimiento que devuelve el autor.
function TTradPlug.Autor(): string;
begin
  Result := 'Neftalí -Germán Estévez-';
end;

//: Texto hola mundo traducido...
procedure TTradPlug.Traduce();
begin
  MessageDlg('Hello World!!', mtInformation, [mbOK], 0);
end;

//: función que devuelve la versión
function TTradPlug.Version(): String;
begin
  Result := 'v1.0';
end;

//: función que devuelve el idioma.
function TTradPlug.Idioma: string;
begin
  result := '&English';
end;
```

Por último, sólo quedan dos procedimientos (que no por últimos son menos importantes) que utilizaremos para "Registrar" la Clase del formulario/PlugIn. Con este registro nos aseguraremos que después podamos acceder a la clase utilizando **RTTI**. De otra forma no sería posible.

Se trata de **Initialization** y **Finalization**; Se ejecutan de forma automática cuando se carga y se descarga el package. Aprovechando este funcionamiento los usaremos para hacer el **RegisterClass** y el **UnregisterClass**.

A continuación se puede ver el código de estos dos procedimientos que se implementan al final del formulario.

Con esto queda definida la estructura de un PlugIn.

```
// I N I T I A L I Z A T I O N
//-------------------------------------
initialization
  // Registrar la clase del form
  RegisterClass(TTradPlug);

// F I N A L I Z A T I O N
//-------------------------------------
finalization
  // Eliminar el registro de la clase
  UnregisterClass(TTradPlug);
```

PROGRAMA PRINCIPAL

Nuestro programa principal tendrá dos partes diferenciadas.

- Inicialmente buscará los PlugIns disponibles (directorio de **Addin**) para mostrarlos en un menú y hacerlos accesibles al usuario.
- Una vez que el usuario pulsa sobre un elemento del menú correspondiente a un plugIn, éste se carga, ejecuta su operación y se descarga, de forma que no se mantiene ocpando memoria.

La parte más interesante es la carga del propio PlugIn y el acceso por RTTI a la clase registrada para crear el formulario.

Una vez que el usuario selecciona un PlugIn para ejecutarlo, se ejecuta el procedimiento que hay a continuación:

```
// Se lanza cuando seleccionados un plugin del menu.
// When the user select a plugin on menu, this packages is charged
procedure TFormMain._ClickElement(Sender: TObject);
var
  Str:String;
  hndl:HMODULE;
  AClass:TPersistentClass;
  AForm:TForm;
  Routine: TMethod;
begin

  // Cual es el elemento pulsado?
  // Menú element that the user has clicked
  Str := TMenuItem(Sender).Name;
  // Buscar el fichero    // find the file BPL
  Str := Str + PLUG_EXTENSION;
  Str := _pathP + Str;
  // Existe?
  if FileExists(Str) then begin
    // Cargar  // Charge the plugin file
    hndl := LoadPackage(Str);

    // Acceder a la clase del menu  // Access to the form Class by RTTI
    AClass := GetClass(PLUG_MAIN_FORM);

    // Bloque de proteccion para la carga
    try
      // No ha podido acceder a ella
```

```
      // the class is correct (not nil)
      if (AClass <> nil) then begin
        AForm := TComponentClass(AClass).Create(Application) as TForm;

        // proteccion para liberar
        try
          // Acceso a la rutina    // Transalate routine
          Routine.Data := Pointer(AForm);
          // direccion del procedimiento
          // Returns the address of a published method.
          Routine.Code := (AForm).MethodAddress('Traduce');

          // Encontrada la rutina de acceso  // routine Ok
          if (Routine.Code = nil) then begin
            MessageDlg('Error, no se ha encontrado el procedimiento
             (Execute) para el correcto funcionamieno del Plug-in.',
             mtError, [mbOK], 0);
            Exit;
          end;

          // ejecutar    // execue the routine
          TExecuteRoutine(Routine)();

        finally
          Aform.Free;
        end;

        // Liberar / Descargar    // Unload the package from memory
        UnloadPackage(hndl);
      end
      else begin
        MessageDlg('La clase para acceder al plug-in parece que no
                    está correctamente registrada.', mtError, [mbOK], 0);
      end;
    except
      on E:Exception do begin
        MessageDlg('Error al cargar el plug-in.', mtError, [mbOK], 0);
      end;
    end;
  end;
end;
```

Se han marcado en negrita las operaciones más importantes que se realizan y que, por orden, son las siguientes:

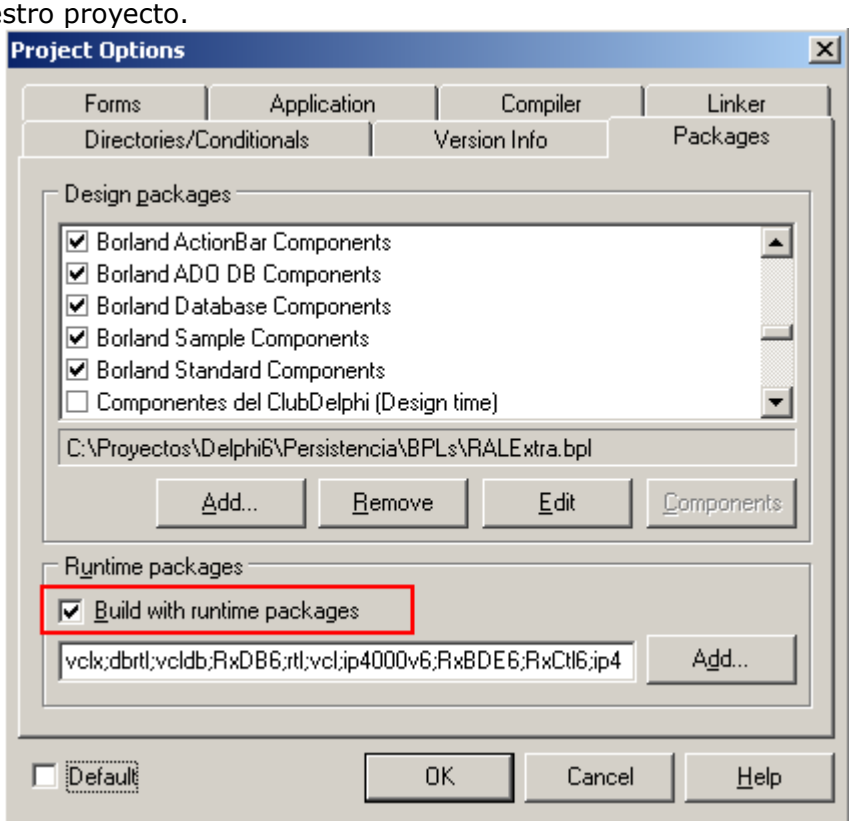
- **Carga del package en memoria:** Una vez que el fichero se ha localizado en disco, con el procedimiento **LoadPackage** de Delphi, cargamos esa BPL en memoria.
- **Acceder a la clase registrada:** Una vez cargado el package en memoria, queremos crear el formulario que hay definido en el package; Para ellos necesitamos tener acceso a la Clase del formulario. Si la clase se registró correctamente (**RegisterClass**) cuando el package se cargó, **GetClass** debería devolver la referencia a esa clase a partir de su nombre.
  Con eso ya se puede crear el formulario.
- **Acceder al método a ejecutar:** En nuestro caso para ejecutar la tarea que el plugIn realiza, necesitamos acceder al método **Traduce**. Para ello también usamos RTTI mediante el tipo **TMethod**, que nos dará acceso a ese método a parti de su nombre (siempre que esté definido como published).
- **Ejecutar el procedimiento:** Se ejecuta el procedimiento anterior (si hace falta se puede tipificar su cabecera).

- **Descargar el PlugIn:** Por último, para ello y una vez que ya no se necesita, podemos utilizar **UnloadPackage** de Delphi.

RESUMEN Y RECORDATORIO

Con esto (y si no nos hemos dejado nada), ya tenemos un sencillo programa que carga "añadidos" o PlugIns para realizar determinadas operaciones. En próximas entregas espero que podamos complicar los pasos y mostrar otras características de forma que podamos ampliar el sistema.

Hay un punto crucial en todo esto y que no debemos perder de vista; el hecho de trabajar con PlugIns (carga dinámica de packages) obliga a que nuestro programa esté "compilado con Runtime Packages"; esta opción es **obligatoria** y deberemos modifiarlas en las opciones de nuestro proyecto.



Por lo demás, espero que este artículo sea claro y explicativo.