Un enfoque modular para nuestra aplicación.

Autor: Salvador Jover

Introducción: Descubriendo nuevos planteamientos

No se si estaréis de acuerdo pero los comienzos de muchos programadores en un entorno como el nuestro, pasan por etapas que suelen ser comunes a una mayoría, sobretodo si la persona que toma contacto con el entorno es un programador novel y con poca o ninguna experiencia. Y me podría poner como ejemplo a mi mismo, recordando cual era mi visión de la generación de código hace unos años, cuando iniciaba mis pasos en Delphi, en contraposición a la actual, con algo más de experiencia, donde la observación de las técnicas empleadas por otros programadores son capaces de sugerirte nuevas posibilidades en el planteamiento de nuestros desarrollos. Y el motivo, en mi opinión, no es otro que la metodología usada en el aprendizaje de nuestra herramienta, que dista bastante de ser la más correcta a largo plazo.

El problema quizás viene en querer obtener resultados inmediatos. Diríamos usando un dicho popular, eso de empezar nuestra casa por el tejado. Nuestro programador, al que llamaremos Listillo (por ponerle un nombre), en una primera toma de contacto descubre el lienzo o formulario y sin darse cuenta, va a entrar en la etapa que podríamos llamar de "Relación de Objetos", en la que se le educa en una filosofía un tanto "dañina", en la que prima únicamente responder a los eventos producidos por el sistema y establecer relaciones entre los distintos componentes del interfaz. A Listillo le basta entender que puede relacionar dos componentes de una forma sencilla, para que instantáneamente, comience el diseño de su aplicación a través del interfaz que requerirá el usuario. Es una relación causa-efecto que el mismo entorno favorecerá de forma involuntaria. Un botón aquí, otro allá. Una casilla de edición aquí y respondiendo al evento OnKeyDown, una implementación que modifique una etiqueta situada unos centímetros mas arriba. En el cuento de los tres cerditos este sería el primero de los hermanos, construyendo su casita con paja. Sobre este punto, recuerdo perfectamente los comentarios que me hacía un profesor de la Universidad y buen amigo (Francisco Mora) acerca de estas primeras etapas de aprendizaje, durante un curso especifico de Delphi. Me explicaba Francisco, como estas primeras etapas se cubren con gran rapidez y como, al ascender a esa loma imaginaria donde creemos encontrar la cima, solo descubrimos que existe una verdadera montaña tras ella.

Nuestro programador puede entonces iniciar una segunda etapa en la que se descubre la creación de componentes y conceptos como herencia, polimorfismo, y otros muchos mas, propios de la Orientación a Objetos. Bien. Podemos llamar a esa segunda etapa con ese nombre, que parece adaptarse bien a sus características: "Orientación a Objetos".

Relacionar objetos no suele ser suficiente sobretodo si tenemos en cuenta que muchos de los componentes no se adaptan siempre a los requisitos deseados. Normalmente adolecerán de alguna función necesaria para nuestro desarrollo. ¿Solución…? Seguramente, la opción más fácil para Listillo es hacer una búsqueda en foros y paginas especializadas en Internet. Son las típicas preguntas que vemos día a día: "¿alguien conoce un componente que sea como un panel pero que además luzca un hermoso gradiente y la etiqueta se pueda situar en dirección horizontal. Es urgente por favor." La respuesta suele desembocar en páginas con vocación de almacén donde uno se sonríe pensando en no volver a trabajar más en una sola línea de código. *–Si aquí tengo de todo, para que pasar trabajo… Solo tengo que echar mano de...*

Y esta visión un tanto infantil suele durar hasta que se descubre que a pesar de existir estos grandes hipermercados de componentes, no siempre se acaba encontrando lo que uno busca. Y si existe, posiblemente vale un dinero que no se tiene. Por lo que a menudo, el programador acaba viéndose en la necesidad de tener conocimientos acerca de la Orientación a Objetos y en poco tiempo entrará de lleno en el mundo de la de la construcción de componentes. La idea que mueve esta segunda etapa puede ser: "si no tienes lo que buscas quizás lo puedas construir tu mismo…". Podríamos relacionar esta etapa con el segundo cerdito del cuento, el que construía su casa con madera.

Y el lobo sopló, y resopló…

De no haber existido un lobo feroz, los dos cerditos bien hubieran podido vivir muchos años, más felices que el tío de la tiza (es una expresión de mi pueblo), pero he aquí que existe ese lobo en forma de nuevos requerimientos para nuestras aplicaciones: las cuales vemos crecer día a día, y acaban demandando nuevas modificaciones a un coste razonable. Aquello que antes nos parecía seguro y estable ahora ya empieza a ser incomodo y la cantidad de horas que dedicamos a reinventar la rueda no llegan a compensar el flujo monetario de retorno a ese esfuerzo (valorando horas/moneda). Las referencias entre los distintos módulos, donde todo el mundo es amigo y conoce a todo el mundo, posiblemente sea una de las razones mas visibles del problema.

Así que podemos finalmente, tras una larga peregrinación, conocer una tercera etapa en donde intentamos encontrar documentación sobre buenas técnicas de construcción de software. Y sobre este punto ya es un tanto más difícil encontrar documentación aplicada a Delphi. Tendríamos que revisar las publicaciones sobre modelos y patrones de diseño más clásicos y no se aplican a nuestro entorno. Podemos leer "Patrones de diseño" de Erich Gamma, que ha llegado a ser una verdadera referencia para muchos programadores. También probé, mientras preparaba los artículos de ModelMaker, la lectura de "UML y Patrones" de Craig Larman, (también de la editorial Prentice Hall) pero una siempre acaba por preguntarse como aplicar lo que lee

a sus desarrollos sin perder un tiempo que no se tiene.

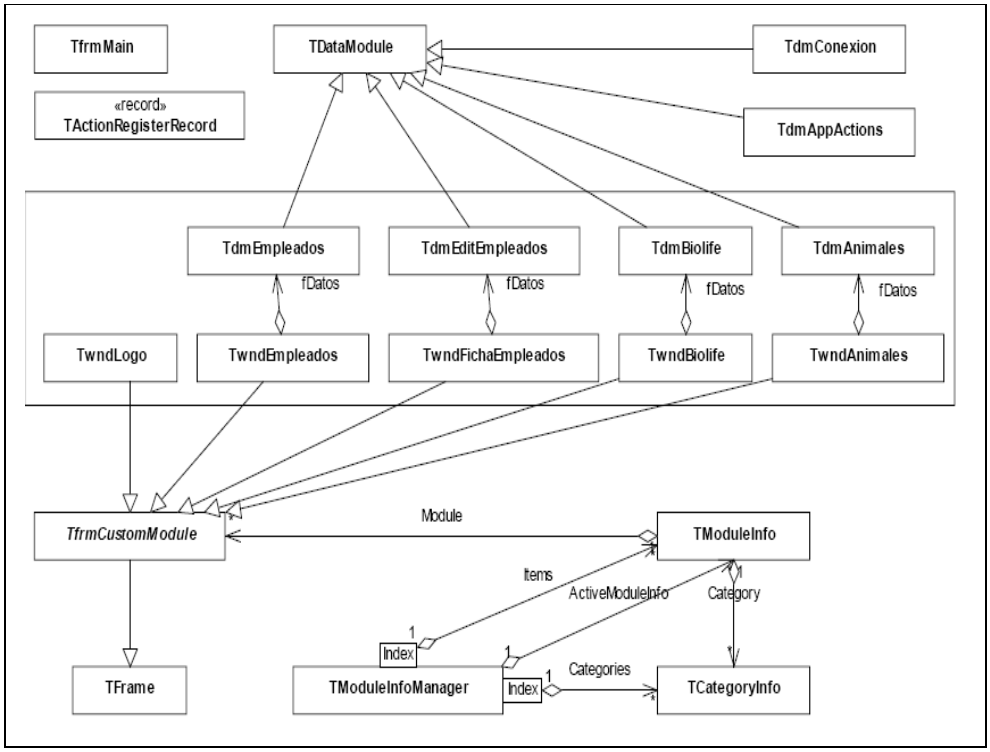
Reconozcamos que no es demasiado habitual este enfoque en las publicaciones que podemos encontrar sobre Delphi, las cuales basan el esfuerzo mayor en "mostrar" y no tanto en "rentabilizar". Decía Ian Marteens al principio del capítulo 23 de "La cara oculta de Delphi 6: (Herencia visual y prototipos):
"La clave para desarrollar aplicaciones, y no perder dinero en el proceso, es terminarlas lo antes posible. Y cuando se programa en Delphi, una de las principales ayudas la ofrece un recurso conocido como herencia visual, que permite definir prototipos de formularios con el diseño visual y el código fuente común de varias ventanas".

Y prosigue durante 10 páginas más, un pequeño proyecto en el que hace uso de esta técnica. Yo os aconsejaría su lectura, sobretodo para aquellos que andan buscando sistemas modulares basados en la herencia visual.

Pero este artículo, el que ahora nos ocupa, nace en el contexto de la publicación de las tres entradas de mi blog de fechas 4, 5 y 8 de Mayo, donde me quejaba de este último tema (al hablar de la necesidad de crear un buen Framework) y de otros más relacionados. Así que es un buen momento para empezar a verlo sin más demora.

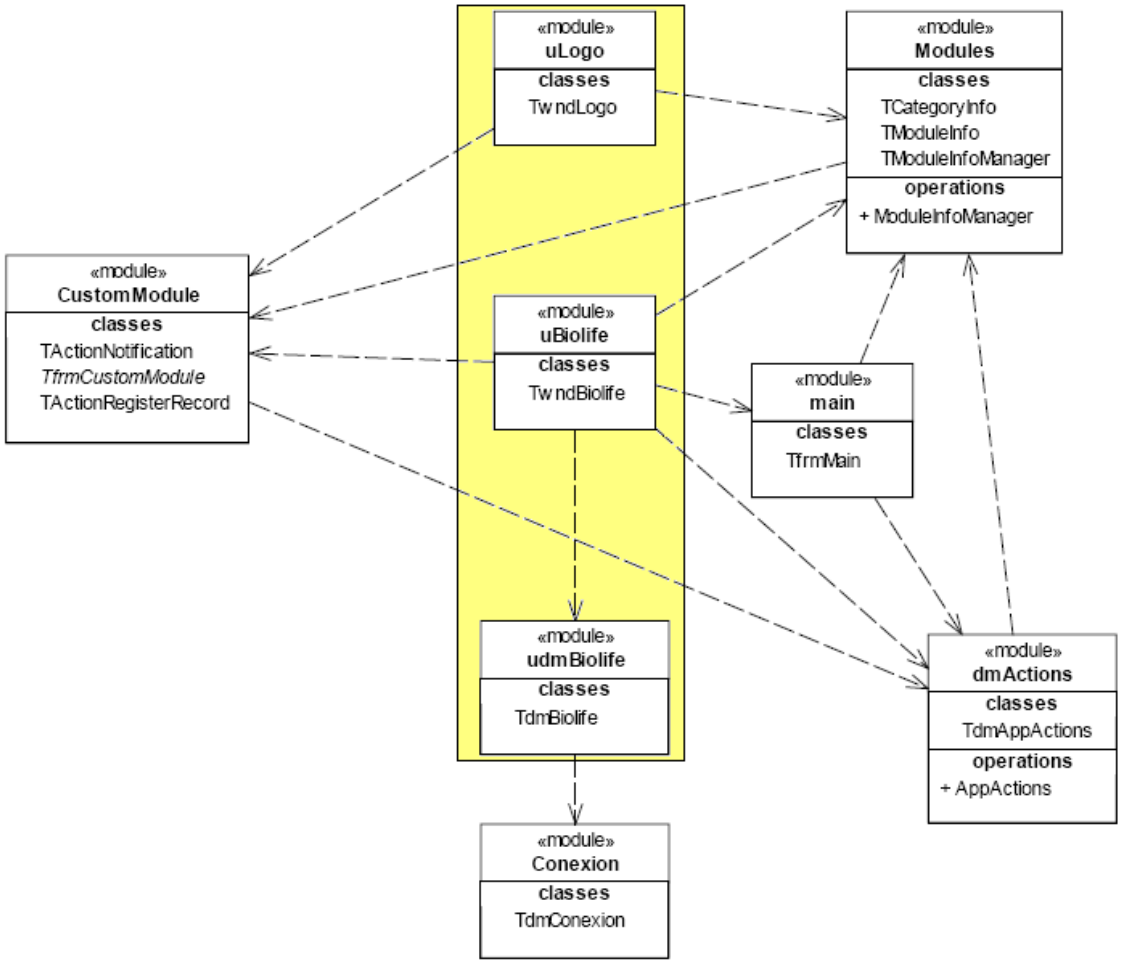
El framework de Developer Express.

Posiblemente, existan muchas formas de hacer lo mismo que la planteada en las páginas web de esta empresa, conocida por unos componentes de gran calidad y no menos complejidad.



En una de las secciones de su Web, intentaban educar a sus desarrolladores dentro de lo que entienden como buenas prácticas de desarrollo y se mostraba un buen ejemplo de desarrollo modular. Actualmente ya no existe dicho enlace, al menos yo no he podido encontrarlo y quizás por esa razón, y movido por la publicación de todos los comentarios escritos en las entradas del blog, pensé que valía la pena recoger esa idea y retomarla, respetando el espíritu de quien la había creado. Podéis ver un poco mas arriba una imagen que muestra la relación de las clases en un diagrama, tal y como se plantea en el ejemplo que vamos a comentar.

En la siguiente imagen podéis apreciar como se relacionan los módulos principales (se han dejado los mínimos con el fin de que se aprecie mejor). Una vez, visto, ya podremos empezar a contar qué se persigue y como se llega al resultado.



El objetivo que se persigue es trabajar con nuestro desarrollo permitiendo que podamos extender de forma sencilla nuevos módulos que amplíen su funcionalidad. Básicamente es eso. Aquí la inversión de tiempo la haremos en la primera fase de diseño del proyecto y se rentabilizara a medida que va siendo extendida.

Supongamos que queremos que añadir a nuestro desarrollo un modulo de personal, representado en una rejilla y una ficha de edición para el registro de datos. La idea es que sea el propio modulo que va a ser añadido, el que registre qué acciones debe habilitar el interfaz principal cada vez que se active y asimismo, personalice los distintos menús, barras de botones y paneles de opciones. Justo al contrario de cómo razonaba nuestro programador, Listillo, en las primeras etapas vividas. Si pudiéramos ojear su código en aquellos momentos, probablemente observaríamos que la creación de los distintos formularios sería responsabilidad de la ventana principal o conocida por ella y esto, aunque no es malo en si mismo, sí impide la modularidad puesto que trabajamos con la idea concreta de la clase de módulo y no con su abstracción.

Finalmente y volviendo al ejemplo actual, podemos comentar que esta información debe ser recogida y gestionada por alguien (ya veremos quien) que finalmente informará a nuestro interfaz principal de las acciones a tomar. Si os fijáis, este esquema de trabajo, evita que la ventana principal "conozca" cada uno de los módulos añadidos, eliminando referencias que nos condicionarían en un desarrollo modular.

Una imagen de nuestro framework:

El gestor de acciones

Empecemos a razonar las distintas entidades que participaran y como primer objetivo, tenemos aquella que se ha de preocupar de gestionar y recoger qué acciones habilitaran cada uno de los módulos. En términos de clases necesitamos una que asuma las demandas propias del objetivo encomendado, como pueden ser informar del número

de acciones almacenadas y permitir una referencia segura a las mismas, de forma que puedan ser ejecutadas.

Estudiemos algunos detalles de su interfaz público:

```
TdmAppActions = class(TDataModule)
   private
     procedure ActionManagerExecute(Action: TBasicAction; var Handled:
Boolean);
   …
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property ActionCount: Integer read GetActionCount;
    property Actions[Index: Integer]: TBasicAction read GetAction;
    property Key: string read GetKey;
    property Pc: string read FPc;
    property Usuario: string read FUsuario;
    property ManejadorDeAcciones: TActionManager read FManejadorDeAcciones
write SetManejadorDeAcciones;
  end;
```

Lo primero que observamos en el modulo *dmActions* es que la primera invocación de la función global *AppActions* será responsable de crear la instancia del objeto contenedor de acciones o bien devolver una referencia al mismo si ya está creado.

```
function AppActions: TdmAppActions;
begin
  if(dmAppActions = nil) then
    dmAppActions := TdmAppActions.Create(Application);
  Result := dmAppActions;
end;
```

Respecto al ejemplo de Developer Express, se puede apreciar que he movido el componente *TActionManager*, que originalmente se sitúa dentro del datamodule hacia la ventana principal, por motivos del cambio de componentes en la interfaz principal. Esto origina que necesite una propiedad adicional para realizar la asignación de la referencia *FManejadorDeAcciones* e intente garantizar que dicho puntero siempre contenga una referencia valida al acceder a la matriz de acciones. Una opción podría haber sido redefinir el constructor pero partimos de que el modulo de datos se crea antes que la ventana principal por lo que difícilmente podríamos conocer en ese momento el parámetro de la clase *TActionManager*.

En el procedimiento de escritura de propiedad, es decir, cuando se produce la asignación valida de la referencia, asignamos el evento *OnExecute* del componente *ActionManager* que nos permitirá enlazar con el modulo responsable de resolver el estado de cada acción y ejecutarla.

```
procedure TdmAppActions.SetManejadorDeAcciones(const Value: TActionManager);
begin
```

```
  FManejadorDeAcciones := Value;
  if FManejadorDeAcciones <> nil then begin
     FManejadorDeAcciones.OnExecute:= ActionManagerExecute;
     FakeVCLActions;
  end
  else FManejadorDeAcciones.OnExecute:= Nil;
end;

procedure TdmAppActions.ActionManagerExecute(Action: TBasicAction; var
Handled: Boolean);
begin
  // Llamada al metodo de ejecución de modulo activo (el visualizado)
  if (ModuleInfoManager.ActiveModuleInfo <> nil) then
    Handled:= ModuleInfoManager.ActiveModuleInfo.Module.ExecuteAction(Action);
end;
```

Precisamente, cuando resolvemos:

ModuleInfoManager.ActiveModuleInfo.Module.ExecuteAction(Action);

lo que estamos pidiendo al Manejador del Módulos (*ModuleInfoManager*), la clase experta en la gestión de los distintos módulos, es que se ejecute la acción correcta en el Modulo activo, el que visualiza en ese momento la interfaz del usuario.

Falta comentar, antes de poner nuestros ojos en las siguientes clases, que para no complicar excesivamente este artículo, he omitido conscientemente una serie de métodos que debería implementar la clase *TdmAppActions***.** Estos métodos nos deberían permitir recargar dinámicamente el contenedor de acciones en el momento en el que un nuevo módulo se une. De esa forma, estaríamos mas cerca de alcanzar ese objetivo de trabajar modularmente en tiempo de ejecución, mediante la carga de librerías que encapsulen cada nuevo modulo añadido. Ahora mismo, tal y como está planteada la aplicación, nos estamos conformando con trabajar modularmente en tiempo de diseño.

¿Quién es quien…? ¡Vaya lio…! ☺

Lo mejor es que abráis en un editor de texto el modulo *Modules.pas* y nos centremos en las tres clases que aparecen:

TCategoryInfo

TModuleInfo

TModuleInfoManager

¿Qué hace cada una de ellas? Intentemos explicarlo:

Para aislar nuestra ventana principal de las clases concretas de cada módulo añadido, se propicia que el gestor de módulos (*TModuleInfoMaganer*) siempre trabaje con una abstracción de una clase concreta, común a todos los módulos añadidos. Para nosotros, esa abstracción será *TfrmCustomModule*, el ascendiente obligado en la creación de nuevas clases que pueda gestionar *TModuleInfoManager*.

Así pues, nuestra instancia global de la clase *TModuleInfoManager,* es el

verdadero corazón del framework y será responsable de:

-Ordenar la visualización de un modulo a demanda del usuario.

-Mantener y dar acceso a una lista de categorías que puedan agrupar los distintos módulos. Por ejemplo, podemos tener una categoría genérica COMPRAS que aglutina distintos módulos vinculados directamente con ella, como por ejemplo el alta de proveedores o la generación de un albaran de entrada de existencias. Este punto muestra la relación o vínculo existente entre las clases *TCategoryInfo* y *TModuleInfoManager*. La clase *TCategoryInfo* es una clase auxiliar que se responsabilizará de guardar la información de cada una de las categorías. La clase *TModuleInfoManager* se apoyará en esa información en determinados momentos (por ejemplo al agrupar modulos por categorías)

-Mantener y responsabilizarse del registro de cada modulo. Esta parte es clave ya que se va a establecer una relación real entre el nombre genérico del módulo con su clase, lo que permitirá que el interfaz sea capaz de invocar cada modulo simplemente obteniendo el nombre del mismo y sin conocer la clase concreta que va finalmente a llamar.

-Mantener y dar acceso a una lista de módulos con la información asociada a cada uno de ellos. Esta es la vinculación que existe entre la clase *TModuleInfoManager* y la clase *TModuleInfo*, que será responsable de guardar la información que recibe del registro de cada uno de los módulos.

-Asegurar una referencia válida al modulo activo, que es el que visualiza el usuario. Cualquier otra clase, se dirigirá a ésta para obtener el frame activo a través de la propiedad *ActiveModuleInfo*

Podemos ver su interfaz:

```
 TModuleInfo = class(TObject)
  private
    FCategory: TCategoryInfo;
    FHasParametros: Boolean;
    FImageIndex: Integer;
    FModule: TfrmCustomModule;
    FModuleClass: TfrmCustomModuleClass;
    FName: string;
    procedure DoModuleDestroy(Sender: TObject);
    function GetActive: Boolean;
  protected
    procedure AsignaParametros(AParametros: Array of Variant);
    procedure CreaModulo(AParametros: Array of Variant);
    procedure DestroyModule;
  public
    constructor Create(const AName: string; AModuleClass:
TfrmCustomModuleClass;
            ACategory: TCategoryInfo; AImageIndex: Integer = -1);
    destructor Destroy; override;
    procedure Abrir(AParent: TWinControl);
    procedure AbrirConParametros(AParent: TWinControl; AParametros: Array of
            Variant);
```

```
    function HasParametros: Boolean;
    procedure Hide;
    property Active: Boolean read GetActive;
    property Category: TCategoryInfo read FCategory;
    property ImageIndex: Integer read FImageIndex;
    property Module: TfrmCustomModule read FModule;
    property Name: string read FName;
  end;
```

Pero lo mas interesante para poder comprender como se relacionan las distintas clases es reproducir la cadena de llamadas desde el modulo concreto. Todo empieza cuando el modulo invoca las funciones de registro en su inicialización.

```
initialization
  ModuleInfoManager.AddCategory('Home', 0);
  ModuleInfoManager.RegisterModule('Help', TwndLogo,
ModuleInfoManager.GetCategoryByName('Home'), 0);
```

Es decir, que al inicializarse añadimos la categoría si es necesario y registramos en el Gestor de módulos el nombre de la clase, la clase y la categoría asociada a la misma. El último parámetro será la imagen mostrada por el interfaz (0).

La llamada al procedimiento *RegisterModule* la tenéis a continuación:

```
procedure TModuleInfoManager.RegisterModule(const AName: string; AModuleClass:
        TfrmCustomModuleClass; ACategory: TCategoryInfo = nil; AImageIndex:
        Integer = -1);
var
  AModuleInfo: TModuleInfo;
begin
  //Aqui se podria revisar la seguridad antes de registrar el modulo
  AModuleInfo := GetModuleInfoByName(AName);
  //Es lanzada una excepcion si el modulo ya existe con el mismo nombre
  if(AModuleInfo <> nil) then
    raise Exception.CreateFmt('El módulo con nombre "%s" ya existe', [AName]);
  // Creamos una categoria si no existe todavía
  if CategoryCount = 0 then
    AddCategory('Default', -1);   //por defecto le llamamos "Default"
{  if ACategory = nil then
    ACategory := Categories[0]; }  //dejamos que ACategory pueda ser nil
  // Creamos la instancia de información y la añadimos a la lista
  AModuleInfo := TModuleInfo.Create(AName, AModuleClass, ACategory,
AImageIndex);
  FModuleList.Add(AModuleInfo);
end;
```

Lo mas destacado es el objetivo final: crear una instancia de la clase *TModuleInfo* y guardarla en una lista, de forma que permita al gestor de módulos obtener al información que necesita para poder trabajar con la clase genérica y resolver correctamente en la clase concreta, apoyándose en el polimorfismo y la redefinición de métodos de las clases concretas.

Más que comentar línea a línea de código, quizás lo más efectivo es ejecutar desde Delphi la aplicación que se entrega de ejemplo, paso por paso, para ver sobre el terreno

el orden de llamada de las distintas funciones y procedimientos. No obstante, sí que debería comentar, que al estudiar los nuevos requerimientos que me pedía este ejemplo, frente al original de Developer Express, me vi en la necesidad de añadir un procedimiento *AbrirModuloConParametros*, que nos permitiera adaptarnos a la necesidad de tener parámetros adicionales al activar el módulo, como pudiera ser en el caso de que el modulo representara una ficha de edición y recibiera como parámetro la clave primaria del registro, para así abrir de forma parametrizada el dataset. O como también sucede, un parámetro que represente la operación que deseamos efectuar *[ver la constante TOperacion que se declara en el modulo de acciones]*. Un ejemplo de esto precisamente lo tenéis en la Edición de Fichas de Empleados (*uEditEmpleado.pas*). Sobre este punto y hablando del parámetro matricial abierto de tipo *Array of Variants*, sería interesante que buscarais en mi blog un par de entradas que hablan acerca de este tema, con el título *¿Te topaste con un variant?*

Por no perder demasiado tiempo, en el ejemplo me he valido de una tabla en lugar de hacer uso de una consulta, lo cual no resulta demasiado afortunado pero sí rápido,☺. Quiero decir con esto, que en condiciones normales, en una aplicación cliente-servidor deberíamos de restringir los registros de la rejilla garantizando que retornan una cantidad apropiada. En muchos casos, podría ser interesante interponer, antes de su presentación visual, un sistema de filtro en forma de ventana, permitiendo al usuario seleccionar el rango de registros a mostrar. En ese punto, podría ayudarnos el evento *OnShowModule* que hemos implementado en la clase *TfrmCustomModule,* y que se dispara antes de visualizarse el frame.

Hay una línea de código que puede pasar desapercibida y que también me gustaría comentar. Mirad la implementación del procedimiento *Abrir* de la clase *TModuleInfo* y concretamente la línea en que se produce la asignación del nombre genérico del modulo (*FName*) al ascendente de la clase concreta, representado en la referencia *Module*.

```
procedure TModuleInfo.Abrir(AParent: TWinControl);
begin
  if Module = nil then CreaModulo([]);
  Module.Parent := AParent;
  Module.Align := alClient;
  Module.ModuloName:= FName; // <- Interesante
  Module.Show;
end;
```

Esta línea la añadí con el fin de que previa a la ejecución del método Show, que hará que nuestro modulo se convierta en el modulo activo, podamos a través del nombre obtener algunas funcionalidades adicionales, desde el mismo modulo descendiente y tras haber sobrescrito el mensaje de Windows que recibe la ventana al ser activada. Tomad por ejemplo una misma clase que se registre en distintos módulos y que previa a su visualización, según quien la invoque haga o muestre cosas distintas.

El cabeza de familia: TfrmCustomModule y sus hijos…

Una vez que hemos presentado, por un lado aquella clase (*TdmAppActions*) que gestiona las acciones disponibles para cada módulo. Y por otro lado, el grupo de clases que gestionan la presentación de cada uno de ellos y su acceso desde el interfaz principal (*TmoduleInfoManager*, *TModuleInfo* y *TcategoryInfo*), nos queda estudiar la clase *TfrmCustomModule*, ascendiente de cada uno de los módulos que queramos añadir.

Veamos su interfaz y el de uno de sus descendientes, y comentemos lo que nos parezca más interesante.

```
  TfrmCustomModule = class(TFrame)
    dsGeneral: TDataSource;
    procedure dsGeneralDataChange(Sender: TObject; Field: TField);
    procedure dsGeneralStateChange(Sender: TObject);
  private
    FAcciones: TAcciones;
    FHasParametros: Boolean;
    FOnDestroy: TNotifyEvent;
    FPc: string;
    FUsuario: string;
    FSupportedActionList: TList;
    FOnShowModule: TNotifyEvent;
    FModuloName: String;
    function GetKey: string;
    function GetNotificationByAction(Action: TBasicAction):
TActionNotification;
    procedure SetModuloName(const Value: String);
    procedure SetAcciones(const Value: TAcciones);
  protected
    procedure DoShowModule; virtual;
    //acciones básicas de gestion de tablas
    procedure DoActionAlta(Action: TBasicAction); virtual;
    procedure DoActionAnterior(Action: TBasicAction); virtual;
    procedure DoActionCancelar(Action: TBasicAction); virtual;
    procedure DoActionEliminar(Action: TBasicAction); virtual;
     procedure DoActionGuardar(Action: TBasicAction); virtual;
    procedure DoActionModificar(Action: TBasicAction); virtual;
    procedure DoActionPosterior(Action: TBasicAction); virtual;
    procedure DoActionPrimero(Action: TBasicAction); virtual;
    procedure DoActionUltimo(Action: TBasicAction); virtual;

    procedure RegisterAction(const Action: TBasicAction; ANotification:
            TActionNotification);
    procedure RegisterActions; virtual;

    procedure GoToFicha(const AOperacion: TOperacion); virtual; abstract;

    property Key: string read GetKey;
    property Pc: string read FPc;
    property Usuario: string read FUsuario;
  public
    procedure WMNCPaint(var Msg: TMessage); message WM_NCPAINT;
    constructor Create(AOwner: TComponent); override;
```

```
    constructor CreateWithParams(AOwner: TComponent; AParams: Array of
Variant);
            virtual;
    destructor Destroy; override;
    procedure AsignaParametros(AParams: Array of Variant); virtual;
    function ExecuteAction(Action: TBasicAction): Boolean; override;
    function HasParametros: Boolean;
    function IsActionSupported(Action: TBasicAction): Boolean;
    function HayCambios: Boolean; virtual;
    procedure UpdateActionsState; virtual;
    procedure UpdateActionsVisibility; virtual;
    property OnDestroy: TNotifyEvent read FOnDestroy write FOnDestroy;
    property OnShowModule: TNotifyEvent read FOnShowModule write
FOnShowModule;
    property ModuloName: String read FModuloName write SetModuloName;
    property Acciones: TAcciones read FAcciones write SetAcciones;

  end;


TwndAnimales = class(TfrmCustomModule)
…
  private
    { Private declarations }
    fDatos: TdmAnimales;
    procedure DoBeforeShow(Sender: TObject);
  protected
    procedure DoActionAlta(Action: TBasicAction); override;
    procedure DoActionModificar(Action: TBasicAction); override;
    procedure DoActionEliminar(Action: TBasicAction); override;
    procedure DoActionCancelar(Action: TBasicAction); override;
    procedure RegisterActions; override;
    procedure GoToFicha(const AOperacion: TOperacion); override;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure UpdateActionsState; override;
  end;
```

La clase *TfrmCustomModule* se diseña con el fin de que pueda servir como aglutinante de todos los módulos descendientes de el, y como podemos imaginar, contiene toda la funcionalidad mínima común a todos los módulos hijos. En el caso que nos ocupa, la clase *TwndAnimales*, que debería mostrar la tabla de animales de la base de datos *dbdemos.gdb*, tan solo tiene que registrar todas las acciones que desea activar en el interfaz.

```
procedure TwndAnimales.RegisterActions;
begin
  inherited RegisterActions;

  RegisterAction(AppActions.Actions[KALTA], DoActionAlta);
  RegisterAction(AppActions.Actions[KMODIFICAR], DoActionModificar);
  RegisterAction(AppActions.Actions[KELIMINAR], DoActionEliminar);
  RegisterAction(AppActions.Actions[KGUARDAR], DoActionGuardar);
  RegisterAction(AppActions.Actions[KCANCELAR], DoActionCancelar);
  RegisterAction(AppActions.Actions[KPRIMERO], DoActionPrimero);
  RegisterAction(AppActions.Actions[KANTERIOR], DoActionAnterior);
  RegisterAction(AppActions.Actions[KPOSTERIOR], DoActionPosterior);
  RegisterAction(AppActions.Actions[KULTIMO], DoActionUltimo);
end;
```

Algunas de estas acciones registradas, podría darse el caso de que fueran comunes a todos los módulos, que es el caso actual de los procedimientos de navegación, alta, baja y modificación de los registros, y de los que existe con un comportamiento por defecto en el *TfrmCustomModule*. Bastaría sobrescribirlos como hemos hecho en las acciones *DoActionAlta*, *DoActionEliminar* y *DoActionCancelar*

Veamos el ejemplo del procedimiento Eliminar, que es sobrescrito para dar la oportunidad al usuario de prevenir un borrado accidental:

```
procedure TwndAnimales.DoActionEliminar(Action: TBasicAction);
Var
   MsgText, MsgCaption : String;
   NL : String;
   MsgType, UserResp : integer;
begin
   NL := #13 + #10;   {New Lin}
   MsgCaption := '¿Deseas eliminar el registro activo (umodulo)?';
   MsgText := MsgText + 'Pulsa Ok para eliminar el registro activo.' + NL;
   MsgText := MsgText + 'Si deseas cancelar pulsa CANCEL.';
   MsgType := MB_OKCANCEL + MB_ICONWARNING + MB_DEFBUTTON2 + MB_APPLMODAL;


   UserResp := MessageBox( Handle, PChar(MsgText), PChar(MsgCaption),
MsgType);
   Case UserResp of
      IDOK :
          begin
             inherited;
          end;
      IDCANCEL :
          begin
          end;
   end;
end;
```

Nos falta conocer de que forma, el modulo descendiente condiciona la visibilidad o la disponibilidad de las distintas acciones. Veamos como lo hace:

```
procedure TwndAnimales.UpdateActionsState;
begin
  with AppActions, dsGeneral do begin
  TAction(Actions[KALTA]).Enabled      := (State in [dsBrowse]);
  TAction(Actions[KMODIFICAR]).Enabled := (State in [dsBrowse]);
  TAction(Actions[KELIMINAR]).Enabled  := (State in [dsBrowse]);
  TAction(Actions[KGUARDAR]).Enabled   := (State in [dsEdit, dsInsert]) and
                                                    DataSet.Modified;
  TAction(Actions[KCANCELAR]).Enabled:= (State in [dsEdit, dsInsert]);
  TAction(Actions[KPRIMERO]).Enabled   := (State in [dsBrowse]);
  TAction(Actions[KANTERIOR]).Enabled  := (State in [dsBrowse]);
  TAction(Actions[KPOSTERIOR]).Enabled := (State in [dsBrowse]);
  TAction(Actions[KULTIMO]).Enabled    := (State in [dsBrowse]);
  end;
end;
```

Al sobrescribir el metodo *UpdateActionsStat*e, que es virtual en la clase *TfrmCustomModule*, conseguimos condicionar las acciones de acuerdo al estado booleano deseado, tan solo accediendo al gestor de acciones y estableciendo las condiciones de visibilidad o disponibilidad. Este método, es invocado cada vez que uno

de los módulos se vuelve activo o cuando se produce algún evento en el que estemos interesados, como pueda ser una actualización del estado de la tabla o de posición de registro.

¿Falta mucho para llegar…?

Una de las preguntas típicas de mi hijo a mitad de un viaje largo, es cuánto falta para llegar... Es el indicador de que al viaje, por muy entretenido que pueda ser, es cansado y aquí, si se me permite la comparación, pasa algo parecido. Porque descubrimos a medida que vamos ampliando nuestro framework que siempre es posible mejorar los puntos implementados y considerar nuevos condicionantes que nos den la sensación de que el viaje no acaba nunca.

Si ahora estamos diseñando un framework para facilitar nuestro trabajo en tiempo de diseño, ¿Por qué no considerar modificarlo para que los distintos módulos sean cargados como librerías independientes, sea cual sea el formato que elijamos para dichos módulos? Además, ¿para qué anclarnos a la visualización en el interior de un contenedor? Podríamos ver interesante que nuestros módulos tuvieran la capacidad de generar el contenedor sobre el que se van a visualizar y establecerse como una aplicación sdi (y no como la clásica mdi) a voluntad del usuario.

¿Alguien hablo de la seguridad? Yo no la he visto por ningún lado. En el ejemplo me he limitado a guardar el usuario y equipo con el fin de visualizarlo únicamente. Pero también podríamos dotar a nuestro desarrollo de un sistema de comprobación de usuario y restricción de opciones a nivel de interfaz, condicionado por el nivel de seguridad de nuestro usuario.

Y por último y para no hacer de este artículo una historia interminable, podríamos considerar un sistema de actualización integrado en el framework, común a todas nuestras aplicaciones y desarrollos.

Es decir que a medida que veamos crecer nuestros requerimientos también crecerán las consideraciones a tener en cuenta en los módulos finales y recordarlas, pasados unos meses, puede ser una pesadilla. Por dicha razón, hablaba en las entradas del blog de que puede ser necesario ayudarnos de un Wizard de creación de módulos, que nos facilite la creación de los mismos de forma automática y sin tener que recordar múltiples detalles que a largo plazo olvidaríamos. En dicho Wizard se plasmarían todos los comentarios y toda la estructura mínima que debemos considerar.

Como decía Nicolás Aragón (Nico) muy acertadamente en una de las entradas del blog, esto lo podemos hacer de diversas formas y no existe tampoco una razón, mas que la de la practicidad, de que se integre en el ide. Lo suyo sería generar un experto, y que estuviera disponible desde los menús de Delphi pero también podría ser una plantilla que mediante una aplicación propia o externa generase las unidades necesarias.

Creo que nada mas… Cualquier comentario adicional, si queréis, lo podemos ver desde mi blog Recibid un saludo y mi deseo de que pueda haber contribuido en algo todos estos comentarios.

Salvador Jover