La potencia de los ClientDataSet

El mundo de la programación de bases de datos cambia tan rápidamente que casi no nos da tiempo a asimilar los nuevos protocolos. Comenzamos con DBase, Clipper, FoxPro, .., hasta hoy en día que tenemos Microsoft SQL Server, Interbase, Firebird, etc.

Luego tenemos el maremagnum de lenguajes de programación donde cada cual se come el acceso a datos a su manera: Java con sus infinitos frameswork tales como hibernate, struts, etc., Ruby con el archiconocido Ruby On Rails que utiliza el paradigma MVC (Modelo, Vista, Controlador), Microsoft a su rollo con ADO y su plataforma Microsoft.NET. Todo eso sin contar con los potentes lenguajes script que no tienen la atención que merecen como son PHP, Ruby, Python, TCL/TK, Groovy, etc. Hasta la mismísima CodeGear nos ha sorprendido con su IDE 3rdRails para programación en Ruby On Rails.

Pero si hay algo que hace que Delphi destaque sobre el resto de entornos de programación es su acceso a múltiples motores de bases de datos utilizando una misma lógica de negocio que abstrae al programador de las rutinas a bajo nivel. En la conocida tecnología de acceso a datos llamada MIDAS.

Las primeras versiones de Delphi contaban con el veterano controlador de bases de datos BDE (Borland Database Engine) que permitía acceder a las clásicas bases de datos DBASE, PARADOX, etc. En la versión 5 de Delphi se incluyeron los componentes IBExpres (IBX) que permitían tener un acceso directo a bases de datos Interbase y Firebird. Fue a partir de Delphi 6 cuando Borland apostó por la tecnología DBExpress, un sistema rápido mediante drivers que utilizando un mismo protocolo permitía conectividad con múltiples motores de bases de datos tales como Interbase, Firebird, Oracle, MySQL, Informix, Microsoft SQL Server, etc. Incluso en su última versión (DBX4) permite acceder a su nuevo motor de bases de datos multiplataforma llamado BlackFish programado integramente en .NET y Java (anteriormente se llamaba JDataStore y estaba programado en Java).

Luego tenemos también otros componentes muy buenos de acceso a datos tales como Zeos, Interbase Objects (IBO), FIBPlus, etc. Y por supuesto el conocido protocolo de acceso a datos de Microsoft llamado ADO, siendo su última versión ADO.NET la que tiene bastantes posibilidades de convertirse en un estandar para todos los entornos Windows.

Pero si hay un componente de acceso a bases de datos en Delphi que destaque sobre todos los demás ese es el **ClientDataSet**. Combina la facilidad de acceso a datos a través de la clase **TDataSet** y la potencia de controlar automáticamente las transacciones al motor de bases de datos, las SQL de consulta, actualización y eliminación así como la conexión y desconexión de las tablas con el servidor haciendo que el programador no tenga que preocuparse de las particularidades del motor de bases de datos.

El componente de la clase **TClientDataSet** no conecta directamente sobre una base de datos en concreto, si no que utiliza el componente **DataSetProvider** que actua de intermediario haciendo de puente entre los componentes de bases de datos (IBX,IBO,etc) y nuestra tabla **ClientDataSet**. El componente **ClientDataSet** es algo así como una tabla de memoria (como la que tienen los componentes RX) que se trae y lleva datos a las tablas de la base de datos encargándose automáticamente de las transacciones.

LA ESTRUCTURA CORRECTA DE UN PROGRAMA

Para crear una buena aplicación con acceso a bases de datos hay que dividir nuestro programa en tres partes principales:

Capa de acceso a datos: se encarga de conectar con un motor de bases de datos en concreto ya sea con componentes IBX, ADO, BDE, etc.

Lógica de negocio: aquí se definen como son nuestras tablas (CLIENTES, ARTICULOS,etc), los campos que contienen así como el comportamiento al dar de alta registros, modificarlos,

realización de cálculos internos, etc. Todo esto lo haremos con componentes de la clase **TClientDataSet** y **TDataSetProvider**.

Interfaz de usuario: se compone de los formularios, informes y menús de opciones que va a visualizar el usuario y que estarán internamente conectados con los ClientDataSet.

La interfaz de usuario sólo podrá acceder a la lógica de negocio y esta última sólo a la capa de acceso a datos. Así, si en un futuro queremos cambiar la interfaz de usuario (por ejemplo para Windows Vista) o el motor de base de datos no afectaría al resto de las capas del programa.

Para ello vamos a utilizar los componentes contenedores de la clase **TDataModule** para alojar la lógica de negocio y el acceso a datos. En nuestro ejemplo crearemos una base de datos de clientes definiendo las tres capas.

CREANDO LA BASE DE DATOS

En este ejemplo voy a crear una base de datos de clientes utilizando el motor de bases de datos Firebird 2.0 y con los componentes IBX. Las tablas las voy a crear con el programa **IBExpert** (http://www.ibexpert.com) cuya versión personal es gratis y muy potente. La base de datos se va a llamar **BASEDATOS.FDB**, pero si haceis las pruebas con Interbase entonces sería **BASEDATOS.GDB**. No voy a explicar aquí como funciona el programa **IBExpert** o **IBConsole** ya que hay en la red información abundate sobre ambos programas.

Creamos la tabla de clientes:

```
CREATE TABLE CLIENTES (
    ID          INTEGER NOT NULL,
    NOMBRE      VARCHAR(100),
    NIF         VARCHAR(15),
    DIRECCION   VARCHAR(100),
    POBLACION   VARCHAR(50),
    CP          VARCHAR(5),
    PROVINCIA   VARCHAR(50),
    IMPORTEPTE  DOUBLE PRECISION,
    PRIMARY KEY (ID)
)
```

Como quiero que el ID sea autonumérico voy a crear un generador:

```
CREATE GENERATOR IDCLIENTE
```

Y un disparador para que cuando demos de alta el registro rellene automáticamente el ID y autoincremente el contador del generador:

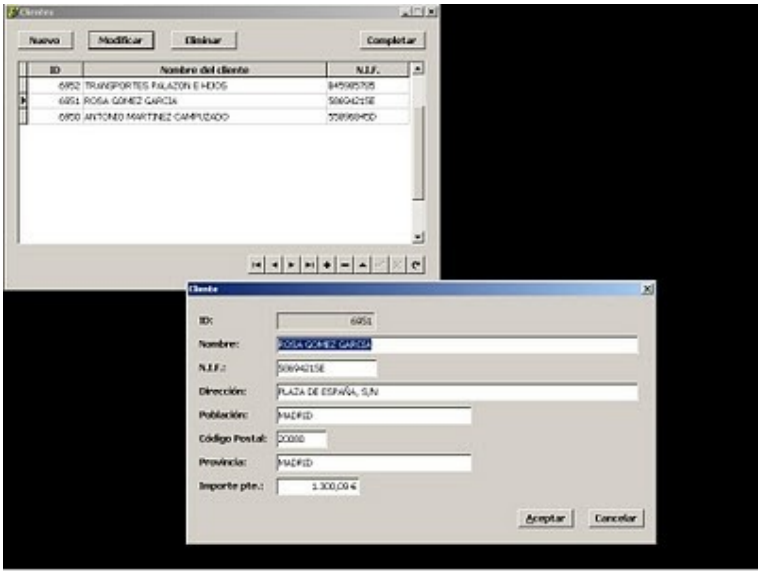
```
CREATE TRIGGER CONTADOR_CLIENTES FOR CLIENTES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  NEW.ID = GEN_ID( IDCLIENTE, 1 );
END
```

NOTA IMPORTANTE: Hay algunas versiones de Delphi 7 que tienen un error en los componentes IBExpress (IBX) que hacen que los componentes ClientDataSet no funcionen correctamente. Recomiendo actualizar los componentes IBX a la versión 7.04 que podeis encontrarla en:

http://codecentral.borland.com/Item.aspx?id=18893

Ahora comenzaremos a realizar el programa utilizando esta base de datos.

Después de haber creado la base de datos en Firebird 2.0 ya podemos comenzar a crear un nuevo proyecto que maneje dicha información. El objetivo del proyecto es hacer el siguiente mantenimiento de clientes:



CREANDO LA CAPA DE ACCESO A DATOS

La capa de acceso a datos encargada de conectar con Firebird va a incorporar los siguientes componentes:

- Un contenedor **DataModule** llamado **AccesoDatos**.

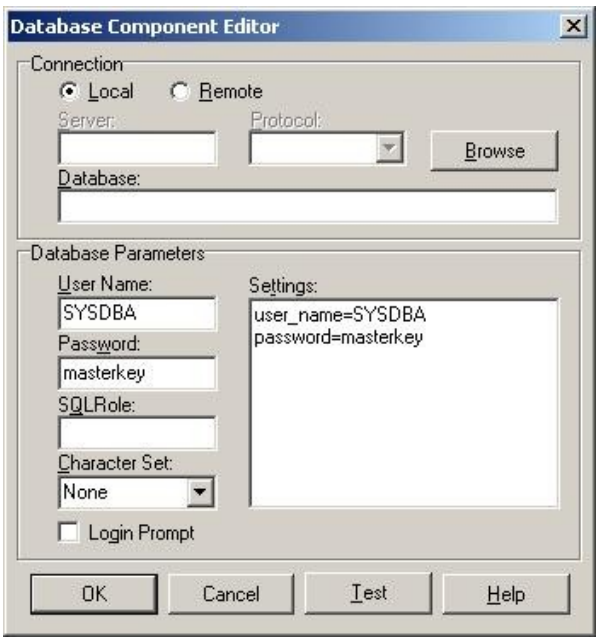
- Un compomente **IBDatabase** (pestaña **Interbase**) llamado **BaseDatos**.

- Un componente **IBTransaction** (pestaña **Interbase**) llamado **Transaccion**.

- Dos componentes **IBQuery** (pestaña **Interbase**) llamados **LstClientes** y **Clientes**.

Se sopone que la base de datos **BASEDATOS.FDB** esta al lado de nuestro ejecutable. Vamos a comenzar a configurar cada uno de estos componentes.

Hacemos doble clic sobre el componente **IBDatabase** y en el campo **User Name** le ponemos **SYSDBA.** En en password le ponemos **masterkey**:



Pulsamos **OK**, y después en la propiedad **DefaultTransaction** del componente **IBDatabase** seleccionamos **Transaccion**, desactivamos la propiedad **LoginPrompt** y nos aseguramos que en **SQLDialect** ponga un **3**.

Para el componente **IBTransaction** llamado **Transaccion** seleccionaremos en su propiedad **DefaultDatabase** la base de datos **BaseDatos**.

Como nuestra intención es tener una rejilla que muestre el listado general de clientes y un formulario para dar de alta clientes, lo que vamos a hacer es crear una tabla **IBQuery** para la rejilla llamada **LstClientes** y otra tabla para el formulario del cliente llamada **Clientes**. Sería absurdo utilizar un mismo mantenimiento para ambos casos ya que si tenemos miles de clientes en la rejilla, el tener cargados en memoria todos los campos del cliente podría relentizar el programa.

En los componentes **LstClientes** y **Clientes** vamos a configurar también:

Database: BaseDatos
Transaction: Transaccion
UniDirectional: True

El motivo de activar el campo **UniDirectional** es para que el cursor SQL trabaje más rápido en el servidor ya que los componentes **ClientDataSet** gestionan en memoria los registros leidos anteriormente.

Como en la rejilla sólo quiero mostrar los campos ID, NOMBRE y NIF entonces en el componente **LstClientes** en su propiedad **SQL** vamos a definir:

```
SELECT ID,NOMBRE,NIF FROM CLIENTES
ORDER BY ID DESC
```

y para el componente **Clientes**:

```
SELECT * FROM CLIENTES
WHERE ID=:ID
```

En la condición WHERE de la SQL hemos añadido el parámetro **:ID** para que se pueda más adelante acceder directamente a un registro en concreto a partir de su campo **ID**.

Una vez definida nuestra capa de acceso a datos ahora nos encargaremos de definir nuestra lógica de negocio.

Después de crear la base de datos y la capa de acceso a datos dentro del **DataModule**

vamos a crear la lógica de negocio.

Antes de seguir tenemos que hacer doble clic en los componente IBQuery de la capa de acceso a datos y pulsar la combinación de teclas **CTRL + A** para introducir todos los campos en el módulo de datos. Y en cada una de ellas hay que seleccionar el campo ID y quitar la propiedad **Required** ya que el propio motor de bases de datos va a meter el campo ID con un disparador.

Pero tenemos un problema, y es que no hemos seleccionado donde esta la base de datos. Para ello hacemos doble clic en el objeto **BaseDatos** situado módulo de datos **AccesoDatos**. Seleccionamos una base de datos Remota (Remote) con IP 127.0.0.1. Y la base de datos donde la tengamos, por ejemplo:

D:\Desarrollo\Delphi7\ClientDataSet\BaseDatos.fdb

De este modo, al hacer CTRL + A sobre las tablas conectará automáticamente sobre la base de datos para traerse los campos. No se os olvide luego desconectarla.

CREANDO LA LOGICA DE NEGOCIO

La capa de lógica de negocio también la vamos a implementar dentro de un objeto **DataModule** y va a constar de los siguientes componentes:

- Un **DataModule** llamado **LogicaNegocio**.

- Dos componentes **DataSetProvider** llamados **DSPLstClientes** y **DSPClientes**.

- Dos componentes **ClientDataSet** llamados **TLstClientes** y **TClientes**.

Ahora vamos a configurar cada uno de estos componentes:

- Enlazamos el DataModule **LogicaNegocio** con el DataModule **AccesoDatos** en la sección **uses**.

- Al componente **DSPLstClientes** le asignamos en su propiedad **DataSet** el componente **AccesoDatos.LstClientes**.

- Al componente **DSPClientes** le asignamos en su propiedad **DataSet** el componente **AccesoDatos.Clientes**.

- El componente **TLstClientes** lo vamos a vincular con **DSPLstClientes** mediante su propiedad **ProviderName**.

- El componente **TClientes** lo vamos a vincular con **DSPClientes** mediante su propiedad **ProviderName**.

- Debemos hacer doble clic en ambos ClientDataSets y pulsar la combinación de teclas CTRL + A para meter todos los campos.

CONTROLANDO EL NUMERO DE REGISTROS CARGADOS EN MEMORIA

Los componentes **ClientDataSet** tienen una propiedad llamada **PacketRecord** la cual determina cuandos registros se van a almacenar en memoria. Por defecto tiene configurado **-1** lo que significa que se van a cargar todos los registros en la tabla. Como eso no me interesa en el listado general del formulario principal lo que vamos a hacer es poner esta propiedad a **100**.

Por eso he ordenado la lista por el campo **ID** descendentemente para que se vean sólo los últimos 100 registros insertados. Una de las cosas que más me gustan de los componentes **ClientDataSet** es que se trae los 100 últimos registros y desconecta la tabla y la transacción quitándole trabajo al motor de bases de datos. Si el usuario que maneja el programa llega hasta el registro número 100 el propio componente conecta automáticamente con el servidor, se trae otros 100 registros y vuelve desconectar.

Lo único en lo que hay que tener cuidado es no acumular demasiados registros en memoria ya que puede relentizar el programa e incluso el sistema operativo si el PC no tiene mucha potencia.

El componente **ClientDataSet** llamado **TClientes** lo dejamos como está en **-1** ya que sólo lo vamos a utilizar para dar de alta un registro o modificarlo.

ENVIANDO LAS TRANSACCIONES AL SERVIDOR

Cuando se utilizan los clásicos métodos Insert, Append, Post y Delete con los objetos de la clase TClientDataSet el resultado de las operaciones con registros no tiene lugar en la base de datos hasta que envíamos la transacción al servidor con el método **ApplyUpdates**.

Por ello vamos a utilizar el evento **OnAfterPost** para enviar la transacción al servidor en el caso que haya sucedido alguna modificación en el registro:

```
procedure TLogicaNegocio.TClientesAfterPost( DataSet: TDataSet );
begin
  if TClientes.ChangeCount > 0 then
  begin
    TClientes.ApplyUpdates( 0 );
    TClientes.Refresh;
    if TLstClientes.Active then
      TLstClientes.Refresh;
  end;
end;
```

Después de enviar la transacción hemos refrescado la tabla **TClientes** y también la tabla **TLstClientes** para que se actualicen los cambios en la rejilla. Por último, cuando en el listado de clientes se elimine un registro también hay que enviar la trasacción al servidor en su evento **OnAfterDelete**:

```
procedure TLogicaNegocio.TLstClientesAfterDelete( DataSet: TDataSet );
begin
  TLstClientes.ApplyUpdates( 0 );
end;
```

Con esto ya tenemos controlada la inserción, modificación y eliminación de registros hacia el motor de bases de datos.

ESTABLECIENDO REGLAS DE NEGOCIO EN NUESTRAS TABLAS

Las reglas de negocio definidas en el módulo de datos le quitan mucho trabajo al formulario que esté vinculado con la tabla. En un primer ejemplo vamos a hacer que cuando se demos de alta un cliente su importe pendiente sea cero. Esto se hace en el evento **OnNewRecord** del componente **TClientes**:

```
procedure TLogicaNegocio.TClientesNewRecord( DataSet: TDataSet );
begin
  TClientesIMPORTEPTE.AsFloat := 0;
```

```
end;
```

Otra de las malas costumbres que solemos cometer en los programas es controlar los datos que introduce o no el usuario en el registro, cuya lógica la hacemos en el formulario. Esto tiene el inconveniente en que si en otro formulario hay que acceder a la misma tabla hay que volver a controlar las acciones del usuario.

Para evitar esto, tenemos que definir también en la capa de lógica de negocio las reglas sobre las tablas y los campos, lo que se llama comunmente **validación de campos**. El componente **ClientDataSet** dispone de la propiedad **Constraints** donde pueden definirse tantas reglas como queramos. Para verlo con un ejemplo, vamos a hacer que el usuario no pueda guardar el cliente si no ha introducido su nombre.

Para definir una regla en un **ClientDataSet** hay que hacer lo siguiente (con **TClientes**):

- Pulsamos el botón **[...]** en la propiedad **Constraints**.

- Pulsamos el botón **Add New**.

- En la propiedad **CustomConstraint** definimos la condición de error mediante SQL:

```
NOMBRE IS NOT NULL
```

- En el campo **ErrorMessage** del mismo **Constraint** ponemos el mensaje de error:

```
No ha introducido el nombre del cliente
```

Con esta regla definida, si en cualquier parte de nuestro programa hacemos un **Post** de la tabla clientes y esta vacío el campo **NOMBRE** el programa lanzará un mensaje de error sin tener que programar nada. Antes teníamos que hacer esto en el botón **Aceptar** del formulario para validar los campos. Con los **Constraints** las validadiones las hacemos en sin tener que programar.

Ahora vamos a definir otra regla la cual establece que un cliene no puede tener un importe pendiente superior a **2000 €**. Creamos un nuevo **Constraint** con la propiedad **CustomConstrait** definida con:

```
IMPORTEPTE <= 2000
```

y con la propiedad **ErrorMessage** que tenga:

```
El importe pendiente no puede ser superior a 2000 €
```

Se pueden definir tantas reglas como deseemos. Ahora vamos a hacer los formularios de mantenimiento de clientes.

Ya tenemos todo lo necesario para realizar el programa:

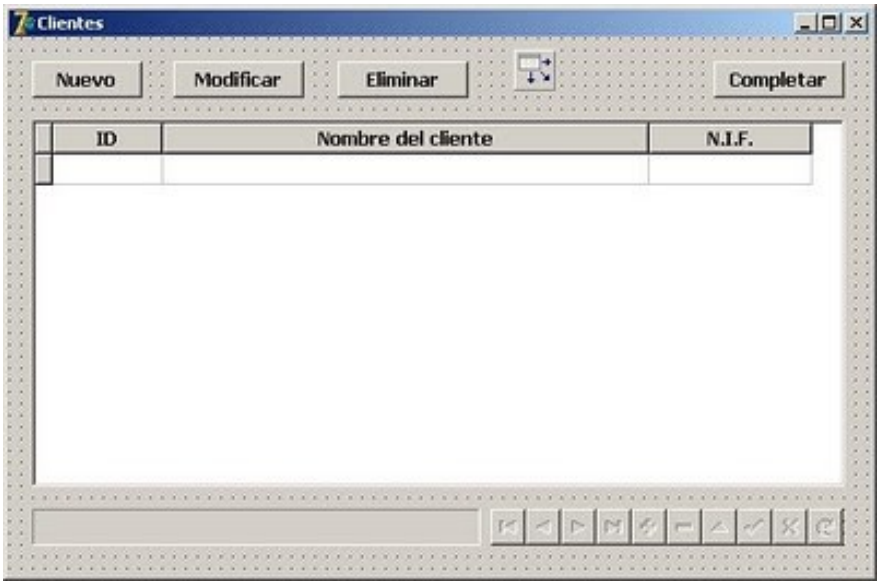
- La base de datos firebird: **BASEDATOS.FDB**

- La capa de acceso a datos en el módulo de datos: **AccesoDatos**.

- La lógica de negocio en el módulo de datos: **LogicaNegocio**.

CREANDO EL FICHERO GENERAL DE CLIENTES

El primer formulario que vamos a crear se va a llamar **FClientes** y va a contener el listado general de clientes:



Va a contener los siguientes componentes:

- 4 botones de la clase **TButton** para dar de alta clientes, modificarlos y eliminarlos. Habrá otro llamado **BCompletar** que utilizaremos más adelante para dar de alta clientes de forma masiva.

- Una rejilla de datos de la clase **TDBGrid** que va a contener 3 columnas para el listado del cliente: **ID**, **NOMBRE** y **NIF**. Su nombre va a ser **ListadoClientes**.

- Un componente **DataSource** (pestaña **Data Access**) llamado **DSLstClientes** que va a encargarse de suministrar datos a la rejilla.

- Un componente **DBNavitator** (pestaña **Data Controls**) para hacer pruebas con los registros. Lo llamaremos **Navegador**.

- Una barra de progreso a la izquierda del **DBNavigator** de la clase **TProgressBar** que mostrará el progreso de los clientes dados de alta automáticamente a través del botón **BCompletar**.

Ahora hay que vincular los componentes como corresponde:

- Lo primero es añadir en la sección **uses** el módulo de datos **LogicaNegocio** para poder vincular el listado de clientes a la rejilla.

- Vinculamos el componente **DSLstClientes** con el ClientDataSet llamado **TLstClientes** a través de su propiedad **DataSet**.

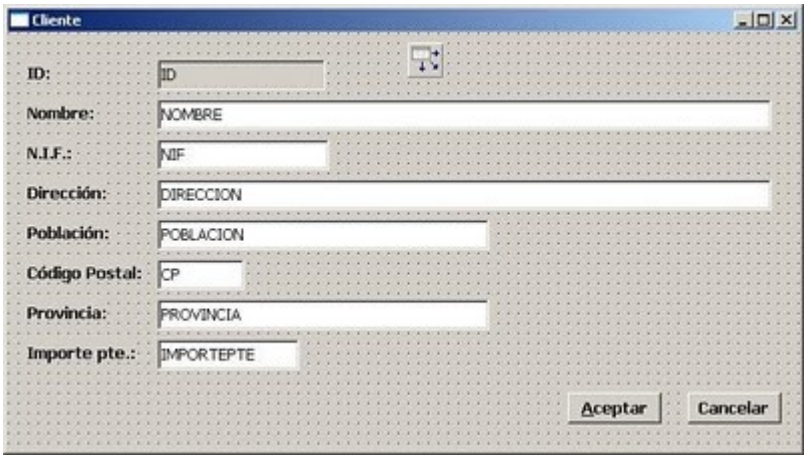
- En la propiedad **DataSource** de la rejilla de datos **ListadoClientes** asignamos el componente **DSLstClientes**.

- Vinculamos el componente **Navegador** al componente **DSLstClientes** mediante su propiedad **DataSource**.

Más adelante escribiremos código para cada uno de los botones.

CREANDO EL FORMULARIO DEL CLIENTE

Vamos a crear el siguiente formulario llamado **FCliente**:



Va a constar de tantos componentes de la clase **TLabel** y **TDBEdit** como campos tenga la tabla clientes. Todos los campos son modificables a excepción del **ID** que lo he puesto con su propiedad Enabled a False. También lo he oscurecido podiendo de **Color** el valor **clBtnFace**.

Para vincular los campos a la tabla real de clientes introducimos un componente **DataSource** llamado **DSClientes**, pero antes hay que añadir en la sección **uses** el módulo de datos **LogicaNegocio**.

Ahora se vincula el componente **DSClientes** con el ClientDataSet llamado **TClientes** situado en el módulo de datos **LogicaNegocio** a través de su propiedad **Dataset**. Con esto ya podemos vincular cada campo **TDBEdit** con el DataSource **DSClientes** y con su campo correspondiente especificado en la propiedad **DataField**.

Cuando se pulse el botón **Aceptar** ejecutamos el código:

```
procedure TFCliente.BAceptarClick( Sender: TObject );
begin
  LogicaNegocio.TClientes.Post;
  ModalResult := mrOk;
end;
```

Al botón **BAceptar** le ponemos en su propiedad **ModalResult** el valor **mrNone**. Esto tiene su explicación. Cuando pulsemos **Aceptar**, si el usuario no ha rellenado correctamente algún dato saltará un error definido en los **Constraint** y no hará nada, evitamos que se cierre el formulario. Así obligamos al usuario a introducir los datos correctamente. Sólo cuando el **Post** se realice correctamente le diremos al formulario que el resultado es **mrOk** para que pueda cerrarse.

En el botón **BCancelar** con sólo asignar en su propiedad **ModalResult** el valor **mrCancel** no será necesario hacer nada más.

Sólo una cosa más: si utilizais el teclado numérico en los campos de tipo **Float** vereis que la tecla decimal no funciona (aquí en España). Nos obliga a utilizar la coma que está encima de la barra de espacio. Para transformar el punto en coma yo lo que suelo hacer es lo siguiente (para el campo I**MPORTEPTE**):

```
procedure TFCliente.IMPORTEPTEKeyPress( Sender: TObject; var Key: Char );
begin
  if key = '.' then
    key := ',';
end;
```

Es decir, en el evento **OnKeyPress** cuando el usuario pulse el punto lo transformo en una coma. Si tuvieramos más campos float sólo habría que reasignar el mismo eventos al resto de campos.

Con esto ya tenemos terminado el formulario del cliente.

TERMINANDO EL FICHERO GENERAL DE CLIENTES

Una vez que tenemos la ficha del cliente vamos a terminar el mantenimiento de clientes asignado el código a cada botón. Comencemos con el botón **Nuevo**:

```
procedure TFClientes.BNuevoClick( Sender: TObject );
begin
  LogicaNegocio.TClientes.Open;
  LogicaNegocio.TClientes.Insert;
  Application.CreateForm( TFCliente, FCliente );
  FCliente.ShowModal;
  LogicaNegocio.TClientes.Close;
end;
```

Como puede apreciarse abrimos la tabla **TClientes** sólo cuando hay que dar de alta un cliente o modificarlo. Después la cerramos ya que para ver el listado de clientes tenemos nuestra tabla **TLstClientes**.

Para el botón **Modificar** es muy parecido:

```
procedure TFClientes.BModificarClick( Sender: TObject );
begin
  LogicaNegocio.TClientes.Params.ParamByName( 'ID' ).AsString :=
LogicaNegocio.TLstClientesID.AsString;
  LogicaNegocio.TClientes.Open;
  LogicaNegocio.TClientes.Edit;
  Application.CreateForm( TFCliente, FCliente );
  FCliente.ShowModal;
  LogicaNegocio.TClientes.Close;
  ListadoClientes.SetFocus;
end;
```

Aquí hay que detenerse para hablar de algo importante. Cuando abrimos una tabla por primera vez el cursor SQL dentro del motor de bases de datos se va al primer registro. ¿Cómo hacemos para ir al registro seleccionado por **TLstClientes**? Pues le tenemos que pasar el **ID** del cliente que queremos editar.

Esto se hace utilizando parámetros, los cuales hay que definirlos dentro del componente **ClientDataSet** llamado **TClientes** que está en el módulo de datos **LogicaNegocio**. Generalmente cuando se pulsa **CTRL + A** para traernos los campos de la tabla al **ClientDataSet** se dan de alta los parámetros. Como el componente **TClientes** está vinculado al IBQuery **Clientes** que tiene la SQL:

```
SELECT * FROM CLIENTES
WHERE ID=:ID
```

entonces al pulsar **CTRL + A** en el **ClientDataSet** nos da de alta automáticamente el parámetro **ID**. Si no fuera así, tendríamos que ir al componente **TClientes**, pulsar el botón **[...]** en su propiedad **Params** y dar de alta un parámetro con las propiedades:

DataType: ftInteger
Name: ID
ParamType: ptInput

Los parámetros dan mucha velocidad a un programa porque permiten modificar las opciones de la SQL sin tener que cerrar y abrir de nuevo la consulta, permitiendo saltar de un registro a otro dentro de una misma tabla a una velocidad impresionante.

Y por último introducimos el código correspondiente al botón **Eliminar**:

```
procedure TFClientes.BEliminarClick( Sender: TObject );
begin
```

```
  with LogicaNegocio.TLstClientes do
    if RecordCount > 0 then
      if Application.MessageBox( '¿Desea eliminar este cliente?', 'Atención',
        MB_ICONQUESTION or MB_YESNO ) = ID_YES then
        Delete;
  ListadoClientes.SetFocus;
end;
```

Antes de eliminar el registro nos aseguramos que de tenga algún dato y preguntamos al usuario si esta seguro de eliminarlo.

Con esto finalizamos nuestro mantenimiento de clientes a falta de utilizar el botón **Completar** donde crearemos un bucle que dará de alta tantos clientes como deseemos para probar la velocidad de la base de datos. Esto lo haremos ahora.

Antes de proceder a crear el código para el botón **Completar** vamos a hacer que la conexión del programa con la base de datos sea algo más flexible.

Hasta ahora nuestro objeto **IBDatabase** está conectado directamente a una ruta fija donde tenemos el archivo **BASEDATOS.FDB**. Pero si cambiamos de ruta el programa dejará de funcionar provocando un fallo de conexión.

Para evitar esto vamos a hacer que el programa conecte con la base de datos al mostrar el formulario principal **FClientes**. Para ello en el evento **OnShow** de dicho formulario ponemos lo siguiente:

```
procedure TFClientes.FormShow( Sender: TObject );
begin
  AccesoDatos.BaseDatos.DatabaseName := '127.0.0.1:' +
ExtractFilePath( Application.ExeName ) + 'BASEDATOS.FDB';
  try
    AccesoDatos.BaseDatos.Open;
  except
    raise;
  end;
  LogicaNegocio.TLstClientes.Active := True;
end;
```

Esto hace que conecte con la base de datos que está al lado de nuestro ejecutable. Así hacemos que nuestro programa sea portable (a falta de instalarle el motor de bases de datos correspondiente). Si la conexión con la base de datos es correcta entonces abrimos la tabla del listado de clientes (**TLstClientes**).

Cuando se trata de una base de datos **Interbase** la conexión puede ser local o remota. Si es local no es necesario poner la IP:

```
AccesoDatos.BaseDatos.DatabaseName := 'C:\MiPrograma\BaseDatos.gdb';
```

Aunque viene a ser lo mismo que hacer esto:

```
AccesoDatos.BaseDatos.DatabaseName := '127.0.0.1:C:\MiPrograma\BaseDatos.gdb';
```

Se trata de una conexión remota aunque estemos accediendo a nuestro mismo equipo. Si se tratara de otro equipo de la red habría que hacer lo mismo:

```
AccesoDatos.BaseDatos.DatabaseName :=
'192.168.0.1:C:\MiPrograma\BaseDatos.gdb';
```

Para bases de datos **Firebird** no existe la conexión local, siempre es remota. Así que si vamos a conectar con nuestro equipo en local habría que hacerlo así:

```
AccesoDatos.BaseDatos.DatabaseName := '127.0.0.1:C:\MiPrograma\BaseDatos.fdb';
```

Yo recomiendo utilizar siempre la conexión remota utilizando para ello un archivo INI al lado

de nuestro programa que contenga la IP del servidor. Por ejemplo:

```
[CONEXION]
IP=127.0.0.1
```

Así podemos hacer que nuestro programa se conecte en local o remoto sin tener que volver a compilarlo. Pero que no se os olvide dejar desconectado el componente **IBDatabase** en el módulo de acceso a datos **AccesoDatos** porque si no lo primero que ha a hacer es conectarse al arrancar el programa provocando un error.

DANDO DE ALTA CLIENTES DE FORMA MASIVA

Para probar el rendimiento de un programa no hay nada mejor que darle caña metiendo miles y miles de registros a la base de datos. Para ello voy a crear un procedimiento dentro del botón **Completar** que le preguntará al usuario cuandos clientes desea crear. Al pulsar **Aceptar** dará de alta todos esos registros mostrando el progreso en la barra de progreso que pusimos anteriormente:

```
procedure TFClientes.BCompletarClick( Sender: TObject );
var
  sNumero: string;
  i, iInventado: Integer;
begin
  sNumero := InputBox( 'Nº de clientes', 'Rellenando clientes', '' );
  Randomize; // Inicializamos el generador de números aleatorios
  if sNumero <> '' then
  begin
    with LogicaNegocio do
    begin
      TClientes.Open;
      Progreso.Max := StrToInt( sNumero );
      Progreso.Visible := True;
      TLstClientes.DisableControls;
      for i := 1 to StrToInt( sNumero ) do
      begin
        iInventado := Random( 99999999 ) + 1000000; // Nos invertamos un
número identificador de cliente
        TClientes.Insert;
        TClientesNOMBRE.AsString := 'CLIENTE Nº ' + IntToStr( iInventado );
        TClientesNIF.AsString := IntToStr( iInventado );
        TClientesDIRECCION.AsString := 'CALLE Nº ' + IntToStr( iInventado );
        TClientesPOBLACION.AsString := 'POBLACION Nº ' +
IntToStr( iInventado );
        TClientesPROVINCIA.AsString := 'PROVINCIA Nº ' +
IntToStr( iInventado );
        iInventado := Random( 79999 ) + 10000; // Nos inventamos el código
postal
        TClientesCP.AsString := IntToStr( iInventado );
        TClientesIMPORTEPTE.AsFloat := 0;
        TClientes.Post;
        Progreso.Position := i;
        Application.ProcessMessages;
      end;
      TClientes.Close;
      Progreso.Visible := False;
      TLstClientes.EnableControls;
    end;
  end;
end;
```

Lo que hemos hecho es inventar el nombre de usuario, dirección, NIF, etc. También he

desconectado y he vuelto a conectar la tabla **TLstClientes** de la rejilla utilizando los métodos **DisableControls** y **EnableControls** para ganar más velocidad. La barra de progreso llamada **Progreso** mostrará la evolución del alta de registros.

Realmente es una burrada dar de alta masivamente registros utilizando objetos **ClientDataSet** ya que están pensados para utilizarlos para altas, modificaciones y eliminación de registros de uno a uno y sin mucha velocidad. Si quereis hacer una inserción masiva de registros hay que realizar consultas SQL con INSERT utilizando el objetos de la clase **TIBSQL** que es mucho más rápido que los **ClientDataSet**. Ya explicaré en otro momento la ventaja de utilizar dichos componentes.

Con esto finalizamos la introducción a los componentes **ClientDataSet**.