Datos, datos ... Megabytes de datos

Por Antonio Quirós

Hace ya varios meses que *Visual Objects* está en el mercado y la tinta derramada sobre el producto va siendo cada vez más extensa. En este mismo medio se ha hablado ya de varias temáticas considerando los distintos ámbitos del producto. VO abarca muchos aspectos del desarrollo de aplicaciones. Podemos emplearlo como casi un compilador de C a la vez que como lenguaje cliente de un potente sistema gestor de bases de datos. Esto da mucha versatilidad al producto, pero hace que a veces permanezca oculta lo que constituye su faceta principal, que es la de ser un lenguaje con el que desarrollar aplicaciones de gestión, bien sea a través de conectar vía ODBC con una base de datos externa, bien sea a través de sus propios ficheros los DBF.

En este artículo trataré precisamente acerca de esa faceta. Desde luego es imposible agotar en el corto ámbito de unas cuantas páginas, la multiplicidad de aspectos que habría que tratar. Por tanto lo que deseo es arrojar algunas pinceladas sobre temas puntuales, esta vez en lo que se refiere a los archivos DBF, ya que hace unos meses dediqué otro artículo a la temática cliente/servidor y pienso aburrir al lector en breve con más información sobre dicha materia.

Así, pues, a lo largo de este artículo, trataré varios temas relativos a los datos, pero sin profundizar demasiado en cada uno de ellos por separado. Creo que todos son temas interesantes para el desarrollo de aplicaciones, temas prácticos, de los que el programador usa a diario en su código fuente. Abordaremos asuntos como RDDs, filtros, registros reciclables, browses, etc. Vamos a ello.

Las ventanas de datos y los servidores de datos

La arquitectura que *VO* nos propone para la gestión de los datos pasa por dos entelequias fundamentales: *Data Window* y *Data Server*. La primera es un tipo de ventana especialmente diseñada para el trabajo con datos, el segundo es un tipo especial de objeto que se encarga de facilitarnos todas las tareas relativas al acceso a los datos ubicados en archivos en disco.

En VO podemos trabajar sin ventanas de datos y/o sin servidores de datos. Por ejemplo, si no deseamos hacer un abordaje OOP podemos acceder a los datos al viejo estilo Clipper, con alias, etc. Igualmente, podemos usar las ventanas de diálogo no especializadas en datos que VO también nos proporciona. Pero las limitaciones son muchas. Así, pues, es prácticamente imposible construir un auténtico sistema MDI sin servidores de datos. Por otro lado, las ventanas de diálogo no permiten el uso de browses. Son dos ejemplos puntuales. Seguro que muchos han intentado andar este mismo camino viendo las formas en que se pueden tratar los datos con VO. Háganme caso, no hay solución: servidores de datos y ventanas de datos son las únicas alternativas válidas. Cualquier otro camino es un callejón sin salida.

Las ventanas de datos son objetos pertenecientes a la clase *DataWindow*. Podemos usar el editor de ventanas para crearlas o podemos hacerlo a través de código fuente nuestro directamente:

```
oDW := DataWindow{Self}
```

Siendo *Self* la ventana propietaria de nuestra Ventana de Datos. Podemos dimensionarla:

```
oDW:origin   := Point{10,10}
oDW:size     := Dimension{300,200}
```

Podemos asignarle características:

```
oDW:caption  := “Mantenimiento de clientes”
```

Y podemos hacer que se muestre:

```
oDW:Show()
```

Los servidores de datos son objetos de la clase *DbServer*. Igual que con las ventanas de datos podemos usar el editor de servidores de datos para crearlos, o bien hacerlo a través de nuestro código fuente:

```
oDS := DbServer{“clientes.dbf”}
```

Una vez instanciado así el servidor, ya podemos manipular a través del mismo todos los datos de *CLIENTES.DBF*. Por ejemplo, podemos movernos:

```
oDS:GoTop()
```

Buscar:

```
oDS:Seek(cValor)
```

Etc.

Lo bueno de ventanas de datos y servidores de datos es que ambos son componentes de una misma arquitectura, es decir, que entre ambos se entienden muy bien. Existe el método mágico *Use()* de la clase *DataWindow* que se encarga de enlazar ambas piezas y hacer que funcionen al unísono. Así, pues, si ejecutamos:

```
oDW:Use(oDS)
```

ya tenemos nuestros dos componentes interoperando. Pero ¿en qué consiste esta operación conjunta? Podemos entenderla a través de dos de sus consecuencias:

1. La ventana es registrada como cliente del servidor de datos, de forma que cualquier actividad registrada por el servidor es inmediatamente comunicada a la ventana.

2. De manera automática se crean en la ventana de datos controles para los campos del servidor, de forma que los datos del servidor son visibles a través de dichos controles y cualquier modificación del valor presente en el control se propaga a la base de datos.

La forma en que los datos se visualizan a través de la ventana es parametrizable, podemos elegir entre la visualización *tipo browse* y la visualización *tipo formulario*:

```
oDW:ViewForm()
```

o bien

```
oDW:ViewTable()
```

Mucha gente piensa que ha de usarse el editor de ventanas de datos para cualquier operación a realizar con las mismas. Esto no es cierto. Todo el código fuente que hasta ahora se ha explicado prescinde en su totalidad del editor de ventanas.

Drivers de base de datos reemplazables (RDD)

Existen dos formas de acceder a datos desde VO, la primera a través de *ODBC* y la segunda a través del tradicional sistema de drivers reemplazables que ya empleaba Clipper desde sus últimas versiones. Aquí no nos ocuparemos de ODBC, ya como se dijo más arriba esto constituirá materia para otro artículo. Nos centraremos, pues, en el asunto de los RDDs.

Visual Objects soporta trabajo a través de tres RDDs básicos: *DBFNTX* (Clipper), *DBFCDX* (FoxPro) y *DBFMDX* (dBase IV). Todos los rudimentos de lenguaje y editores en VO están preparados para trabajar por defecto con *DBFNTX*, pero si deseamos trabajar con cualquiera de los otros RDDs sólo tendremos que hacer algunos pequeños cambios en el uso de los métodos de la clase *DbServer*. Así, pues, si deseamos abrir un archivo en formato *DBFCDX*, tendríamos que escribir:

```
oDS := DbServer{“clientes”,,,”DBFCDX”)
```

La diferencia que existe entre los archivos índice de tipo *CDX* o *MDX* y los *NTX* es que los primeros son archivos que pueden contener varias claves mientras que los últimos sólo pueden contener una. Cuando ejecutamos una instrucción como la anterior, si el nombre del archivo

índice coincide con el del DBF, dicho archivo se abre de forma automática, dejando los registros en su orden natural. Luego se puede conmutar entre cualquiera de las distintas claves existentes en el *CDX* a través del método *SetOrder* de *DbServer*.

El driver para *CDX* que se acompaña con VO es de la empresa *Comix* (el mismo que se acompaña con *Clipper 5.3*) y sustituye a los antiguos drivers para *CDX* de *Succesware* que usaban las antiguas versiones de Clipper. Junto con dicho driver se acompañan las rutinas *ClipMore* del mismo fabricante que optimizan los filtros y otros tipos de acceso masivos de forma similar a como lo hace la denominada tecnología *Rushmore* de FoxPro. Todo esto, junto con razones más importantes que veremos más abajo cuando hablemos del asunto *filtros*, hace que el driver *CDX* sea el más aconsejable de usar entre todos los posibles.

Junto a este artículo se acompaña un pequeño ejemplo que pretende ser una utilidad genérica para la manipulación de bases de datos de tipo *DBFCDX*. Se encuentra en el directorio ejemplos en un archivo *AEF* denominado *DBFCDX.AEF*, dicho archivo se encuentra dentro de otro compactado denominado *DATOS.EXE*. Dicha utilidad permite abrir una archivo *DBF* y su índice *CDX* asociado si tiene el mismo nombre, en caso de tener nombres distintos, permite abrir de forma manual el *CDX* elegido. Un *listbox* nos permite elegir el orden a poner activo entre todos aquellos presentes en los CDX que se han cargado. Sobre la base de datos abierta podemos hacer las operaciones normales de altas, bajas, modificaciones y consultas a través de un proceso de browse sobre la misma. Igualmente podemos realizar las operaciones masivas típicas: *pack*, *reindex*, *recall all* y *zap*.

La elección de la *DBF* a abrir se hace a través de un método de la ventana de entorno que se invoca desde el menú principal. Dicho código puede verse en el fuente 1.

```
// --- Fuente 1 ---------------------------------------------------------
METHOD DBUOpenDBF(lShared) CLASS WEntorno

      LOCAL oOD AS OPENDIALOG
      LOCAL oDW AS WDATA
      LOCAL oDS AS DBSERVER
      LOCAL nOrder AS BYTE

      oOD := OpenDialog{Self, "*.dbf"}
      oOD:Show()

      IF !Empty(oOD:FileName)

            oDS    :=    DbServer{oOD:FileName,lShared,,"DBFCDX"}
            oDW    :=    WData{Self}
            Self:nWData++

            oDW:caption := oOD:FileName + ;
                        If(oDS:Shared, " | SHARED |", " | EXCLUSIVE |")

            IF Self:nWData = 1
                  Self:Menu:AppendItem(MenuDW{Self},"&Indices")
                  Self:Menu:AppendItem(MenuEdit{Self},"&Edición")
                  Self:Menu:AppendItem(MenuOpe{Self},                ;
                                       "&Operaciones masivas"        ;
                                      )
                  Self:menu := Self:menu
            END IF

            oDW:Use(oDS)
            oDW:Show()
            oDW:ViewTable()

            FOR nOrder:=1 TO oDS:OrderInfo(DBOI_ORDERCOUNT)
                  AADD(oDW:aOrderList,oDS:OrderInfo(DBOI_NAME,, nOrder) )
```

```
            NEXT

      END IF
// ---------------------------------------------------------------------------
```

Como puede verse en dicho fuente, se usa la caja estándar para abrir archivos, a través de la clase de VO *OpenDialog* que encapsula la llamada a la correspondiente función al API de Windows. A través del archivo seleccionado se instancia un objeto de la clase *DbServer* a través de la instrucción:

```
oDS :=  DbServer{ oOD:FileName, ;
                  lShared,,           ;
                  "DBFCDX"            ;
                }
```

Igualmente se instancia un objeto de la clase *WData* que no es más que una subclase de *DataWindow*, creada a través de la simple instrucción:

```
CLASS WData INHERIT DataWindow
```

Una vez que ambos objetos se vinculan a través del método *Use()* de la ventana de datos, todo funciona de modo automático, la ventana de datos muestra los datos referidos al archivo gestionado por el objeto de servidor de datos. Nótese solamente cómo, después de ejecutar el método *Show()* que pone a andar la ventana, se ejecuta también el método *ViewTable();* esto se hace así para que la visualización obtenida sea tipo browse. El hecho de que se ejecute después de *Show()* obedece a que si lo hacemos así el foco se queda en el browse con lo que el usuario puede comenzar a moverse a través de él con su teclado sin ninguna dificultad, si se ejecuta antes el browse no toma el foco hasta que no hacemos click con el ratón sobre él.

Otro tema importante es que al abrir la ventana se añaden una serie de opciones de menú nuevas. Esto es porque el sistema es multiventana, es decir, que permite abrir varias bases de datos y tener un conjunto de operaciones para cada una de las ventanas de forma independiente. El fuente 2 muestra el trozo de programa que realiza esta labor.

```
// --- Fuente 2 ----------------------------------------------------------

IF Self:nWData = 1
      Self:Menu:AppendItem(MenuDW{Self},"&Indices")
      Self:Menu:AppendItem(MenuEdit{Self},"&Edición")
      Self:Menu:AppendItem(MenuOpe{Self}, "&Operaciones masivas")
      Self:menu := Self:menu
END IF
// ---------------------------------------------------------------------------
```

Como puede verse empleamos el método *AppendItem()* de la clase *menu* para añadir nuevos submenús al principal. Estos submenús han sido diseñados con el generador y se añaden en tiempo de ejecución al principal. Como puede verse, este proceso sólo se produce cuando

nWData vale 1, con esta variable controlamos el número de ventanas abiertas que tenemos en el sistema, sólo cuando abrimos la primera se añaden los nuevos submenús y nunca en el resto de los casos.

Cuando se termina el proceso de la ventana lanzada, los nuevos submenús se desactivan quedando sólo el principal de nuevo. De esta operación se ocupa el método *Close()* de la ventana llamada. Esto puede verse en el fuente 3.

```
// --- Fuente 3 ----------------------------------------------------------
METHOD Close(oE) CLASS WDATA

      Super:Close(oE)

      Self:Owner:nWData--

      IF Self:Owner:nWData =    0
            Self:Owner:Menu := MenuDBU{Self:Owner}
      END IF
// ------------------------------------------------------------------------
```

Como puede verse, este método se ocupa no sólo de dejar el menú principal, sino también de controlar la variable *nWData* para decrementar en uno el contador de ventanas abiertas.

Indices y órdenes

Como ya hemos comentado, el driver *CDX* es de clave múltiple, esto quiere decir que en un solo archivo *CDX* podemos tener la totalidad de claves asociadas a un archivo *DBF*. Cuando abrimos el archivo *DBF* se abre de forma automática el índice *CDX* siempre que el nombre de ambos sea el mismo. Pero si se desea abrir un *CDX* de nombre distinto, bien como único *CDX,* bien para añadir sus claves a la del *CDX* del mismo nombre del *DBF,* usaremos el método *SetIndex()* de *DbServer*.

Nuestra *DBU* usa la siguiente técnica: si el *CDX* se llama igual que el *DBF* lo abre a través de la técnica automática, pero después tiene también una opción que permite abrir *CDX* de distinto nombre. Los distintos órdenes presentes en cada *CDX* se añaden a un array de órdenes que manipula la aplicación y del cual podemos seleccionar el índice activo a través de un *ListBox* que nos los muestra todos. El fuente 4 muestra el proceso de selección de un nuevo *CDX*.

```
// --- Fuente 4 ----------------------------------------------------------
METHOD DBUOpenCDX CLASS WData

LOCAL oOD AS OPENDIALOG
LOCAL nOrder AS BYTE

oOD := OpenDialog{Self, "*.cdx"}
oOD:Show()

IF !Empty(oOD:FileName)
    Self:Server:SetIndex(oOD:FileName)
    FOR nOrder:=1 TO Self:Server:OrderInfo(DBOI_ORDERCOUNT, oOD:FileName)
        AADD(Self:aOrderList, Self:Server:OrderInfo(DBOI_NAME,, nOrder) )
    NEXT
END IF
/ ------------------------------------------------------------------------
```

Nótese en dicho fuente el uso del método *OrderInfo()* de *DbServer*, que concentra todo un conjunto de información que nos es de gran utilidad en nuestros procesos. El array que añadimos es *aOrderList*. Véase como en el fuente 1 ya lo usamos para incorporar los órdenes del *CDX* abierto de forma automática. Ahora lo completamos con los del CDX abierto de forma manual. Dicho array es un dato que hacemos pertenecer a la ventana de datos que está mostrando el archivo DBF seleccionado. Más tarde un *ListBox* lo mostrará y nos permitirá seleccionar el orden activo. El *PushButton* que se encarga de ejecutar esta acción puede verse en el fuente 5. Como verá el lector se encuentra en una *DialogWindow* de nombre *DlgSelectOrder* que lo único que hace es mostrar el *ListBox oDCOrderList*.

```
// --- Fuente 5 ---------------------------------------------------------
METHOD PushButton1() CLASS DlgSelectOrder

    IF !Empty(Self:oDCOrderList:CurrentItem)
        Self:Owner:Server:SetOrder(Self:oDCOrderList:CurrentItem)
    END IF
    Self:EndDialog()
// ---------------------------------------------------------------------
```

Operaciones más y menos masivas

Una *DBU* como la que estamos realizando tiene como finalidad la de facilitarnos la realización de operaciones de todo tipo sobre la base de datos que seleccionemos. En nuestro caso las operaciones permitidas son: *Append, Delete, Pack, Reindex, Recall All* y *Zap*. El código fuente de todas estas operaciones puede verse en el fuente 6.

```
// --- Fuente 6 ---------------------------------------------------------
METHOD DBUPack CLASS Wdata

    IF Self:Server:Shared
        MessageBox(0,"Se necesita exclusividad", "Error", MB_ICONSTOP)
    ELSE
        Self:Owner:Pointer := Pointer{POINTERHOURGLASS}
        Self:Server:Pack()
        Self:Owner:Pointer := Pointer{POINTERARROW}
    ENDIF


METHOD DBURecall CLASS Wdata

    IF Self:Server:Shared
        MessageBox(0,"Se necesita exclusividad", "Error", MB_ICONSTOP)
    ELSE
        Self:Owner:Pointer := Pointer{POINTERHOURGLASS}
        Self:Server:Recall(,,DBSCOPEALL)
        Self:Owner:Pointer := Pointer{POINTERARROW}
    END IF


METHOD DBUReindex CLASS Wdata

    IF Self:Server:Shared
        MessageBox(0,"Se necesita exclusividad", "Error", MB_ICONSTOP)
    ELSE
```

```
        Self:Owner:Pointer := Pointer{POINTERHOURGLASS}
        Self:Server:Reindex()
        Self:Owner:Pointer := Pointer{POINTERARROW}
    END IF

METHOD DBUZap CLASS Wdata

    IF Self:Server:Shared
        MessageBox(0,"Se necesita exclusividad", "Error", MB_ICONSTOP)
    ELSE
        Self:Owner:Pointer := Pointer{POINTERHOURGLASS}
        Self:Server:Zap()
        Self:Owner:Pointer := Pointer{POINTERARROW}
    END IF

// ------------------------------------------------------------------------
```

Las opciones *Append* y *Delete* son simples *EventName* que añadimos en el diseño del correspondiente menú.

Filtros

Uno de los mejores servicios que nos presta el driver *CDX* es el de posibilitarnos filtros de alta optimización, no como los que podemos hacer con *NTX* que son superlentos. Para ello se emplea el método *OrderScope()* de *DbServer*. Fijamos el principio del ámbito del filtro, el final y ya está, todo parece como si nuestra base de datos sólo poseyera los registro presentes en el filtro. Para nuestra *DBU* ponemos otra *DialogWindow* que nos permite seleccionar el filtro a disponer. Una vez elegido el ámbito en dos controles de edición de dicha ventana, un *PushButton* de la misma se encarga de ponerlo activo. Por supuesto, es condición imprescindible que exista un índice activo que soporte el filtro. El código fuente del *PushButton* que dispone los filtros puede verse en el fuente 7.

```
// --- Fuente 7 ----------------------------------------------------------
METHOD PushButton1() CLASS WinScope

    Self:Owner:Server:OrderScope(TOPSCOPE,Self:oDCTopScope:Value)
    Self:Owner:Server:OrderScope(BOTTOMSCOPE,Self:oDCBottomScope:Value)
    Self:Owner:Server:GoTop()
    Self:EndDialog()
// ------------------------------------------------------------------------
```

Como puede verse se emplean las constantes *TOPSCOPE* y *BOTTOMSCOPE* para especificar si deseamos poner el principio o el fin del filtro y, por supuesto, el valor de ambos, que en este caso es el valor del control de edición de la *DialogWindow* de nombre *WinScope,* donde se requiere que el usuario teclee dicho valor. El *GoTop()* es muy importante para activar el filtro, eviatando algunos efectos indeseables que se producen cuando no lo usamos.

Terminando por el principio

Para terminar ya vamos a mostrar el código fuente de inicio de nuestra aplicación. Me refiero al método *Start()* de la misma y a la cabecera de creación de clase de sus dos ventanas más importantes: *WEntorno*, la *Shell* y *WData*, la ventana de datos donde mostramos cada uno de

los *DBF* abiertos. Como el lector habrá visto, casi todos los métodos pertenecen a una u otra de estas ventanas. El fuente 8 nos lo muestra.

```
// --- Fuente 8 ---------------------------------------------------------
CLASS WDATA INHERIT DATAWINDOW
    EXPORT aOrderList := {} AS ARRAY    // Array de órdenes

CLASS WEntorno INHERIT SHELLWINDOW
    EXPORT nWData := 0 AS BYTE          // Número de ventanas de datos

METHOD Start() CLASS App

    LOCAL oWin AS WENTORNO

    oWin := WEntorno{self}
    oWin:Caption := "DBUCDX - Utilidad para Bases de Datos"
    oWin:Menu := MenuDBU{Self}          // Menú principal
    oWin:Icon := IcoDBU{}               // Icono
    oWin:Show(SHOWZOOMED)

    Self:Exec()
// -----------------------------------------------------------------------
```

Un aspecto a tener en cuenta es que en casi todo el diseño se han usado los generadores sólo para lo imprescindible. Por ejemplo, no es necesario crear una ventana de entorno a través del generador, un trozo de código tan sencillo con el del fuente 8 la crea.

Y bien, con esto hemos hecho sólo un pequeño recorrido por algunos de los aspectos relativos a los datos con Visual Objects. Evidentemente, VO es un producto destinado a la gestión de datos y por tanto éste será, sin duda, el camino más transitado por los programadores que trabajen con el producto. Espero que estas breves líneas sirvan de ayuda en ese recorrido.