Texto diseñado para enseñar en profundidad a desarrollar aplicaciones basadas en lenguaje Visual Basic. El texto cubre tanto aspectos básicos como avanzados, es decir que no sólo se estudian los fundamentos del lenguaje sino que también se tratan en profundidad el acceso a datos, la programación orientada al objeto, los componentes ActiveX, el acceso al API, etc. Se trata de un manual de muy amplio contenido con alrededor de 1000 páginas de documentación, además de numerosos ejemplos listos para ser cargados desde Visual Basic.

Entre los requisitos previos, basta con conocimientos de fundamentos de programación y conocer al menos un lenguaje, sea este el que sea y conocer el S.O. Windows a nivel de usuario.

PROGRAMACIÓN CON VISUAL BASIC 6

GRUPO Eidos

Programación con Visual Basic
Versión 1.0.0

www.LaLibreriaDigital.com

Índice

1

Introducción

Introducción a la programación en Windows con Visual Basic

La continua evolución en las necesidades de los programadores, hace que las herramientas de desarrollo sean actualmente uno de los elementos con más rápida evolución en el mundo de la informática.

Programación Orientada a Objetos, Cliente / Servidor, multimedia y sobre todo Internet, como la gran revolución en el mundo de las comunicaciones por ordenador, son requisitos indispensables para cualquier lenguaje que aspire a ocupar un puesto destacado en el mercado de las herramientas de programación.

Uno de los sistemas operativos más populares actualmente, para el que se realizan un gran número de aplicaciones es Windows, y una de las principales herramientas para programar en este sistema es Visual Basic (o VB como también lo denominaremos). Este curso se basa precisamente en estos elementos, la programación de aplicaciones Windows utilizando Visual Basic 6 como entorno de desarrollo.

Visual Basic es una herramienta multipropósito, es decir, podemos crear programas utilizando este lenguaje para resolver diferentes tipos de problemas; sin embargo, debido al gran número de programadores dedicados a desarrollar programas de gestión, este curso va a estar orientado fundamentalmente al manejo de datos.

En los siguientes apartados, veremos brevemente, las nociones básicas de la programación bajo el entorno Windows.

El sistema controla a la aplicación

Aquellos programadores que pasen de trabajar en DOS a Windows necesitarán afrontar una nueva filosofía de trabajo. Cuando creamos un programa para DOS, dicho programa al ejecutarse, acapara todos los recursos del ordenador, controlando teclado, vídeo, estados de espera del programa, etc., todo está en manos del programa hasta que finalice su ejecución. Esto es lógico, ya que al ser un sistema monotarea, la única aplicación en ejecución puede acceder directamente a todos los recursos de la máquina. Existe una excepción en los llamados programas residentes, que una vez cargados son accesibles desde el programa principal mediante una combinación de teclado. Aunque esta técnica es muy útil bajo DOS, no se puede considerar multitarea, ya que no es un sistema preparado para ello.

En Windows se da el caso contrario, al ser un sistema multitarea y poder tener más de un programa en ejecución al mismo tiempo, es el sistema operativo el que tiene el control de todos los programas lanzados. Asignando a cada programa un espacio de direcciones propio, de modo que no entren en conflicto mutuamente. Para comunicarse con los programas, utiliza un sistema de mensajes; Windows recibe y envía peticiones a los programas abiertos, administra las prioridades para los programas, la memoria, y en definitiva se encarga del buen funcionamiento de todos los aspectos del sistema.

Interfaz consistente

Cada programa realizado en DOS puede tener, de hecho la mayoría lo tienen, un aspecto y modo de comunicación con el usuario diferente, lo que habitualmente llamamos interfaz. Esto supone un derroche de tiempo tanto para el usuario como para el programador. Los usuarios cada vez que han de utilizar un nuevo programa de este tipo, deben emplear tiempo en aprender a comunicarse con él. Y los programadores han de gastar parte del tiempo de desarrollo en crear la interfaz de la aplicación.

En Windows todos los programas tienen un único interfaz, lo que redunda en beneficios para usuarios y programadores. Los primeros cuando aprenden a utilizar una aplicación para Windows, ya tienen parte del camino recorrido para las siguientes, debido a que el uso de la ventana que contiene la aplicación es igual para todas. Sólo han de concentrarse en el aprendizaje del contenido de la aplicación y no el continente. Similar situación se da respecto a los programadores, no han de preocuparse del interfaz en absoluto, ya que viene proporcionado por Windows, sólo han de desarrollar las rutinas que resuelvan los problemas con que el programa pueda encontrarse.

Eventos y mensajes

Se dice que Windows es un sistema conducido por eventos, ya que es la forma que tiene de saber lo que está pasando en todo momento. Cualquier cosa que ocurre en Windows es un evento: mover el ratón, abrir un menú, pulsar un control, redimensionar una ventana, etc.

Windows actúa como central de información, tomando los eventos producidos en el sistema y enviando mensajes al programa o programas que necesitan ser notificados de lo que ocurre.

Puesto que Windows hace llamadas a los programas en forma de mensajes, estos han de disponer de un mecanismo para poder saber lo que el sistema les está diciendo. Este mecanismo es el llamado *procedimiento de ventana*, que contiene un *bucle de mensajes* o estructura de tipo *Case – End Case*, en la que se ejecuta el código programado para ese mensaje.

Si no estamos interesados en realizar ninguna operación especial para determinados mensajes, no escribiremos código asociado para ellos, con lo que sólo se ejecutarán las acciones por defecto programadas por el propio Windows.

Lo anteriormente explicado no quiere decir que siempre debamos escribir un procedimiento de ventana para el programa. En Visual Basic este procedimiento es creado y gestionado internamente por el producto para cada ventana, sin que tengamos que preocuparnos en condiciones normales de su mantenimiento; esta es entre otras, una de las muchas facilidades que nos proporciona la herramienta. Sin embargo, también será posible acceder a estos mensajes, tanto para recogerlos como para enviarlos, y hacer llamadas a funciones para las que VB no da soporte, a través del API de Windows.

Librerías de enlace dinámico (DLL)

Cuando en DOS escribimos un programa, muchas de las funciones que usamos en el programa están contenidas en una o varias librerías (ficheros .LIB). Al generar el ejecutable, enlazamos con esas librerías, y el código de tales funciones se añade a nuestro ejecutable. Si creamos más de una aplicación que haga llamadas a las mismas funciones, estas se añadirán a cada fichero .EXE, con lo que tendremos varios programas con código repetido para las rutinas comunes.

Para solventar este problema, en Windows disponemos de DLL's (Dynamic Link Library), que forman uno de los componentes básicos en la arquitectura del sistema operativo. Una DLL es una librería que contiene código compilado y pre-enlazado, accesible desde cualquier programa que se encuentre en ejecución. Podemos seguir utilizando librerías normales LIB, pero las ventajas derivadas del uso de DLL's son mucho mayores. A continuación exponemos algunos de los motivos.

Centralizar el código en un único lugar

Con esto conseguimos un doble objetivo: evitar el código redundante en las aplicaciones, al no incluir el código de las funciones comunes en cada programa; y más facilidad en el mantenimiento de versiones. Una vez modificada una rutina y generada de nuevo la DLL, los cambios son accesibles para todos los programas que la llamen. Con el método tradicional, además de generar de nuevo la librería, tendríamos que recompilar todos los programas que hacen uso de ella, enlazándolos con la nueva versión de la librería, de forma que pudieran incluir las rutinas modificadas o nuevas.

Ahorro de recursos del sistema

Cuando un programa hace una llamada a una rutina de una DLL, y la carga en memoria, las sucesivas llamadas a esa rutina de ese programa o cualquier otro, no la volverán a cargar. Sólo se carga una vez y permanece en memoria mientras se necesite.

Independencia del lenguaje

Podemos tener una DLL construida en Visual C++ y llamar a sus funciones desde nuestro programa en Visual Basic, sólo es necesario conocer el nombre de la función a usar y su protocolo de llamada, esto es, los parámetros a pasar, tipo y valor de retorno de la función.

Aplicaciones SDI y MDI

Una aplicación SDI o de Interfaz de Documento Simple, es aquella en que se utiliza una sola ventana para interactuar con el usuario, el ejemplo más claro es el Bloc de Notas de Windows.

Una aplicación MDI o de Interfaz de Documento Múltiple, usa una ventana principal, que actúa como contenedora de una o más ventanas abiertas dentro de la principal. Como ejemplo de este tipo de programas, tenemos a Word. Cada ventana contenida dentro de la principal corresponde con un documento o fichero, el cual podemos maximizar para que ocupe todo el área disponible de la ventana, iconizar, u organizar en mosaico o cascada.

OLE

OLE es una tecnología que nos permite compartir información entre las diferentes aplicaciones Windows. Desde los tiempos de las aplicaciones DOS, los usuarios han necesitado compartir información entre los distintos programas que usaban, pero las utilidades disponibles para este sistema eran insuficientes. La naturaleza monotarea del DOS complicaba además este problema, al tener que incorporar las aplicaciones sus propios conversores o disponer de programas que realizaran ese trabajo.

Con la llegada de Windows, comenzamos a beneficiarnos de las ventajas del portapapeles, como puente para compartir datos de forma sencilla entre dos aplicaciones. Después llegó DDE (Dynamic Data Exchange o Intercambio Dinámico de Datos) como solución más avanzada para compartir información entre programas, aunque sólo permitía utilizar texto y gráficos.

Vinculación e incrustación de objetos (OLE), nace en un principio como un avance en cuanto a comunicación de datos entre aplicaciones. Si estamos desarrollando un programa que necesita de las características que proporciona una hoja de cálculo, no necesitamos crearla nosotros, ya existen varias en el mercado sobradamente probadas y fiables, ¿para que volver a realizar el trabajo que ya han hecho otros?. Si esa hoja de cálculo, por ejemplo Excel, soporta OLE, incluimos un objeto OLE de Excel en nuestra aplicación, y toda la funcionalidad de Excel pasará a la nuestra.

El programa que contiene un objeto OLE se llama *Aplicación Contenedora*, y el programa que crea y maneja los objetos OLE se llama *Aplicación Servidora*.

Vinculación

Siguiendo con el ejemplo de la hoja de cálculo, un programa que tenga un objeto (fichero XLS) vinculado, al editar el objeto, lanza la aplicación asociada a ese tipo de objeto, en este caso Excel. A partir de ahora utilizamos Excel para modificar el fichero, una vez terminado, grabamos y salimos, con lo que volvemos a nuestro programa original.

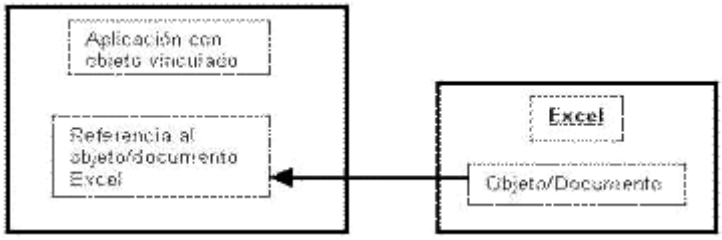


Figura 1. Esquema de vinculación OLE.

Este modo de trabajo es útil cuando un objeto necesita ser compartido por varias aplicaciones, ya que la aplicación guarda una referencia al objeto.

Incrustación

Una aplicación con un objeto incrustado, contiene el propio objeto y no una referencia a un fichero externo. Cuando editemos el objeto no se iniciará la aplicación que genera ese objeto (también llamada aplicación servidora OLE), sino que las barras de menú y herramientas de esa aplicación sustituirán a las de la nuestra durante la edición del objeto; esto es conocido como *Negociación de menú*.

Los cambios realizados en un objeto incrustado sólo serán visibles por la aplicación contenedora de ese objeto.

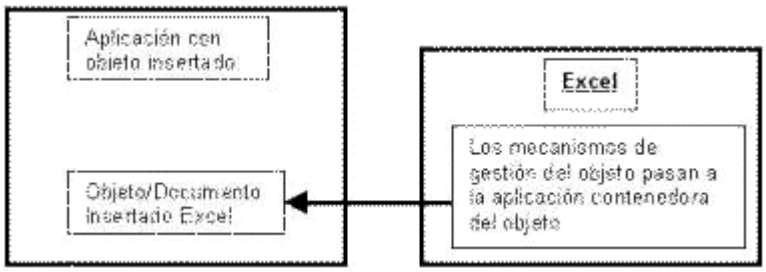


Figura 2. Esquema de incrustación OLE.

ActiveX

En los últimos tiempos, la denominación OLE ha dejado paso a ActiveX. El motivo de ello, es una campaña de marketing realizada por Microsoft y destinada a relacionar la tecnología OLE con todo lo que tenga que ver con Internet. La base de esta tecnología sigue siendo la misma, pero la tendencia progresiva es la de crear objetos y componentes cada vez más pequeños y reutilizables, que faciliten su transmisión a través de la red.

Controles ActiveX y modelo COM / DCOM

Hemos visto que OLE en su versión 1.0, nace como una tecnología de comunicación entre aplicaciones. En la versión 2.0, la idea de comunicar aplicaciones se extiende a la de tratar todos los objetos como componentes que pueden ser montados en las aplicaciones, igual que si fueran piezas de una máquina. Este es el origen del llamado modelo COM (Component Object Model), y las piezas son los controles OCX o ActiveX que tomamos de la caja de herramientas e insertamos en los formularios de las aplicaciones.

La evolución de los controles OCX ha permitido que estos pasen de estar disponibles desde la máquina local a cualquier máquina conectada a la red, con lo cual, la especificación COM ha pasado a ser DCOM (Distributed Component Object Model), o COM distribuido. De la misma forma, el termino *activo* ha sido incorporado a todos los elementos que tengan que ver con Internet, así hemos pasado de controles OCX a ActiveX, y de Automatización OLE a Automatización ActiveX

Automatización ActiveX

La automatización ActiveX consiste en la posibilidad de integrar en nuestra aplicación, capacidades existentes en otras aplicaciones, en forma de componentes ActiveX, de manera que no necesitemos volver a codificar el trabajo hecho anteriormente por otros.

La aplicación que incorpora los componentes se denomina cliente, y la que suministra los componentes, servidora.

De esta forma, podríamos por ejemplo, crear una aplicación que incorporara las capacidades de corrector ortográfico de Word en sus controles de texto.

Programación orientada a objeto en Windows

La aplicación de técnicas de programación orientada a objeto en Windows, ha supuesto una gran ayuda en el desarrollo de las últimas versiones del sistema. De igual forma, el empleo de la orientación a objetos en los programas destinados a funcionar bajo Windows, se traduce en una mayor sencillez a la hora de enfocar los problemas a resolver y en un código más fácil de mantener.

Windows como sistema OOP

Al programar en Windows, estamos programando en un sistema orientado a objetos. Todos los elementos son tratados como objetos: ventanas, botones, cuadros de texto, listas desplegables, etc. Por este motivo es muy importante adaptar nuestra forma de enfocar la programación a estas técnicas de desarrollo

2

Visual Basic

Desde sus comienzos

La aparición de Visual Basic, supuso una auténtica revolución en los lenguajes de programación. Hasta entonces, realizar un programa en Windows, suponía poco menos que ser un experto en C++, y conocer todo el entramado del famoso y extenso API, mensajes, funciones, etc.

Visual Basic ponía al alcance de la mayoría de los programadores, sin necesidad de ser unos expertos en programación Windows, la posibilidad de construir aplicaciones en este sistema. Los aspectos más complejos del programa eran resueltos de forma transparente por la herramienta, permitiendo al desarrollador centrarse en lo que la aplicación debía resolver.

También es cierto que las posibilidades iniciales eran limitadas, pero en las sucesivas versiones de esta herramienta se han ido incorporando o reforzando aspectos tan importantes como la implementación OOP, aunque todavía no disponga de herencia; manejo de bases de datos locales y remotas, permitiendo usar ficheros tipo Access como formato nativo de datos; dBASE, Paradox mediante controladores ISAM; SQL Server, Oracle a través de ODBC, etc. Todo esto y diversos aspectos más, han hecho que una herramienta pensada originalmente para crear sencillas aplicaciones Windows, llegue a ser uno de los productos más robustos para la creación de grandes aplicaciones corporativas.

La situación actual

En Visual Basic 6 se ha realizado un gran trabajo de mejora del producto, si bien desde las versiones 4 y 5 dicha labor de actualización se ha notado sobremanera, incorporando y adaptando las más

novedosas tecnologías de desarrollo, de forma que se puede considerar que estamos ante una de las mejores herramientas visuales existentes.

Algunas de las características más destacables, entre las muchas que incorpora son: la posibilidad de crear ejecutables en Código Nativo para incrementar la velocidad de ejecución; múltiples asistentes para ayudarnos en las más variadas situaciones; creación de controles ActiveX; editor de código fuente mejorado; creación de nuestros propios eventos; acceso a datos mediante OLE DB/ADO; nuevos controles de datos optimizados para ADO; generador de informes propio (ya no es necesario diseñarlos mediante Crystal Reports); jerarquía de objetos para la manipulación de ficheros, etc.

Tipos de proyectos

Al iniciar el entorno de Visual Basic, ya nos encontramos con la primera novedad. En esta versión podemos elegir al comenzar el tipo de proyecto sobre el que vamos a trabajar entre los disponibles en la ventana de diálogo inicial del producto (figura 3).

Figura 3. Ventana de diálogo inicial de Visual Basic 6.

- **EXE estándar.** Ejecutable habitual de VB.
- **EXE ActiveX.** Ejecutable servidor OLE.
- **DLL ActiveX.** DLL servidora OLE.
- **Control ActiveX.** Permite crear un control de usuario con la Edición de Creación de controles.
- **Asistente para aplicaciones de VB.** Nos guía mediante una serie de pasos, en la construcción de una aplicación, creando la funcionalidad básica, ahorrándonos el trabajo más frecuente, de forma que sólo tengamos que incluir código en los puntos que deseemos personalizar.
- Administrador de asistentes de VB. Permite configurar cada uno de los asistentes del entorno.

- **Controles de VB Edición Empresarial.** Al iniciar el proyecto, incluye en la caja de herramientas un conjunto de controles especiales para esta versión de Visual Basic.
- **Complemento.** Utilidad que añade capacidades al entorno de Visual Basic.
- **DLL de Documento ActiveX.** Librería para ejecutarse dentro de Internet Explorer como parte de otra aplicación.
- **EXE de Documento ActiveX.** Aplicación que se ejecuta dentro de Internet Explorer.

Apertura de un proyecto

La ventana de diálogo que se muestra al inicio de VB, también nos permite abrir un proyecto sobre el que hayamos trabajado anteriormente. Sólo hay que seleccionar si es un proyecto existente (figura 4) o uno reciente (figura 5).

Figura 4. Apertura de un proyecto existente.

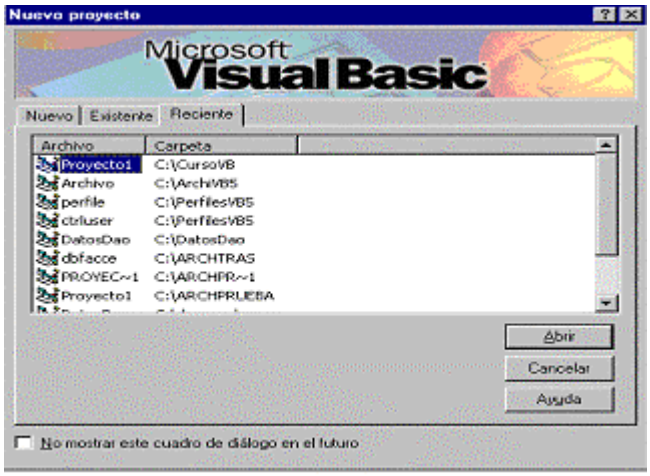


Figura 5. Apertura de un proyecto reciente.

Componentes básicos de un programa Visual Basic

En toda aplicación Visual Basic a la que nos enfrentemos, tendremos que manejar los siguientes elementos:

Proyecto

Aquí es donde se agrupan los diferentes módulos que contiene la aplicación: formularios, módulos de código, módulos de clase, etc., organizados en diferentes carpetas.

Figura 6. Ventana de Proyecto de una aplicación

Formularios

Un formulario es lo mismo que una ventana. Nosotros utilizaremos el término *formulario* ya que es el usado por VB. El formulario es el eje central en los programas, y se le pueden incluir los controles necesarios para que actúe de interfaz entre el usuario y la aplicación. Puesto que es un objeto, posee varias propiedades que pueden ser modificadas para adaptarlo a los requerimientos de la aplicación, así como métodos, en los cuales el programador puede insertar código para que se comporte según sea necesario.

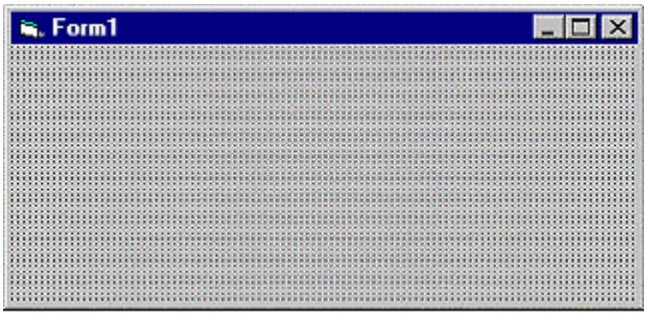


Figura 7. Formulario de programa

Controles

Son todos los elementos incluidos dentro de un formulario: botones de comando, listas desplegables, rejillas, cuadros de texto, botones de opción, de chequeo, menús, etc., en los cuales el usuario inserta información, selecciona valores o pulsa con el ratón para desencadenar ciertas acciones. Son objetos al

igual que los formularios, por lo que también disponen de propiedades para cambiar su aspecto o posición, y métodos para responder a los eventos que reciban.

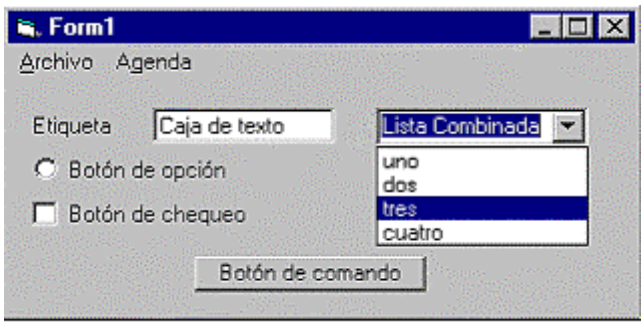


Figura 8. Formulario conteniendo varios controles.

Propiedades

Como dijimos anteriormente, tanto formularios como controles disponen de propiedades que el programador puede alterar. La manera de hacerlo es mediante la ventana de propiedades de VB, que mostrará las propiedades del objeto a modificar.

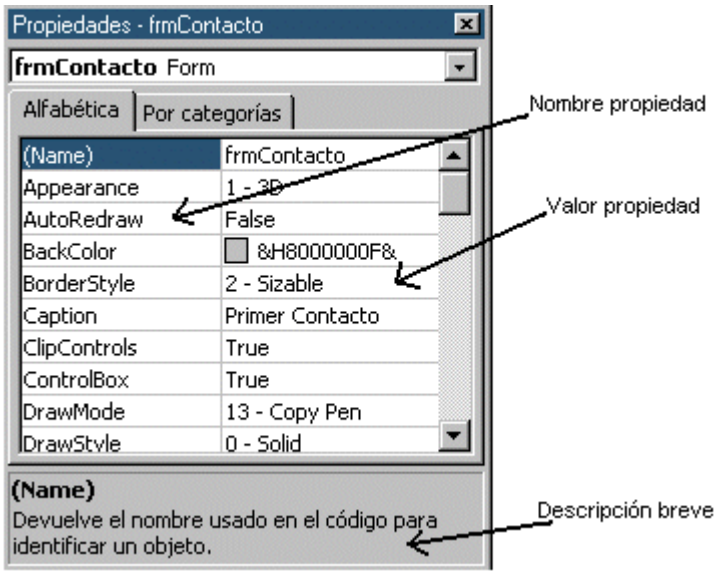


Figura 9.Ventana de propiedades de un objeto formulario.

Dependiendo del objeto con el que estemos trabajando, esta ventana mostrará propiedades comunes para todos los objetos como *Name*, y otras propiedades que serán particulares para un tipo determinado de objetos, por ejemplo *ControlBox* para objetos formulario.

Si pulsamos el botón derecho del ratón en esta ventana, se abrirá un menú con las siguientes opciones:

- **Descripción.** Muestra una breve descripción de la propiedad actualmente marcada.
- **Acoplable.** Una ventana es acoplable cuando está *pegada* a un lateral de la pantalla o a otra ventana, y no lo es cuando se puede mover libremente por la pantalla.

- **Ocultar.** Hace desaparecer la ventana.

Las dos últimas opciones aparecen en la mayoría de ventanas del entorno.

Eventos y métodos

Podemos incluir código asociado a los objetos para que se comporten de la forma que queramos al recibir una acción del usuario. Para ello disponemos de la *ventana de código (figura 10)* del formulario, en la que situaremos las instrucciones que debe seguir un control al recibir un determinado evento.

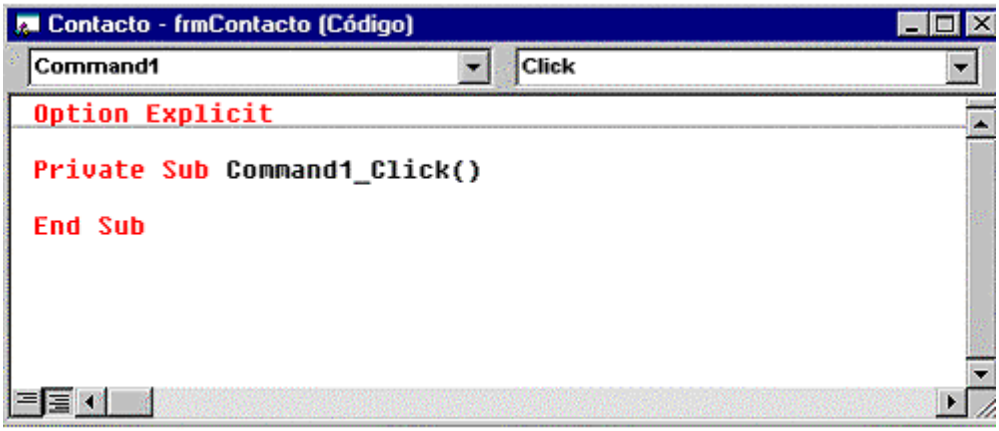


Figura 10. Ventana de código con el método vacío de pulsación de un botón.

Funciones y procedimientos

Son rutinas que a diferencia de los métodos, no están ligadas a ningún objeto. Pueden recibir parámetros, y en el caso de las funciones, devolver un valor. Se crean en la ventana de código. Las veremos con detalle cuando estudiemos los componentes del lenguaje.

Hasta aquí hemos visto muy brevemente los componentes básicos de una aplicación de tipo ejecutable estándar de VB. Veamos seguidamente, como unir todas las piezas para dar forma a un programa.

La primera aplicación

Seguidamente vamos a ver los pasos necesarios para crear una aplicación que nos sirva de aproximación al entorno de desarrollo, el cual trataremos en otro tema con mayor profundidad. De esta forma nos iremos acostumbrando a manejar controles, formularios, ventanas de código, etc.

En primer lugar hemos de fijar el objetivo del programa, es decir, analizar lo que deseamos que el programa haga. En concreto, vamos a diseñar una ventana para que un supuesto comercio de venta de libros y música pueda realizar una encuesta sobre sus clientes.

Diseñar el formulario

Al iniciar VB seleccionamos un nuevo proyecto de tipo ejecutable estándar, lo que creará un nuevo proyecto que contendrá un formulario vacío, listo para editar.

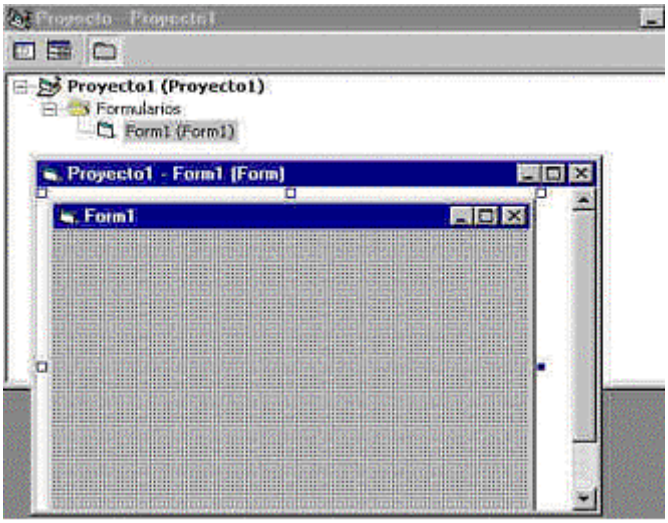


Figura 11. Proyecto incluyendo formulario.

También podemos hacer esta operación mediante la opción del menú de VB *Archivo+Nuevo proyecto*.

Desde el primer momento ya estamos trabajando con objetos. El formulario que tenemos en pantalla es la representación visual de un objeto de la clase *Form*. Al ser un objeto, tiene un conjunto de propiedades que podemos cambiar mediante una ventana existente a tal efecto.

Podemos visualizar la ventana de propiedades (figura 13) de varias formas , como vemos en la figura 12.

- Hacer clic con el botón derecho del ratón dentro del formulario, para visualizar el menú contextual del mismo. Dentro de este menú tenemos una opción para visualizar las propiedades.
- Pulsar el botón de propiedades en la barra de herramientas de VB.
- Seleccionar el menú *Ver+Ventana Propiedades* de VB.
- Pulsar F4.

Procederemos a continuación a modificar algunas de las propiedades por defecto del formulario, comentando al mismo tiempo en que consiste la propiedad modificada. Las propiedades no comentadas tanto para formularios como controles, pueden ser consultadas por el lector empleando la ayuda de Visual Basic. Aquí sólo nos detendremos en las propiedades más significativas o que requieran mención por guardar relación con algún aspecto del ejemplo en que estemos trabajando.

Figura 12. Diferentes maneras de visualizar la ventana de propiedades.

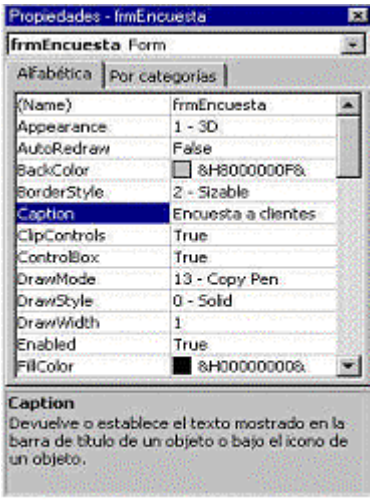


Figura 13. Ventana de propiedades.

- Name. Cadena de caracteres con el nombre identificativo del objeto utilizado en el código fuente de la aplicación. Asignaremos al formulario el nombre frmEncuesta en esta propiedad.

- Caption. Texto que se visualiza en la barra de título del formulario. Asignaremos la cadena Encuesta a clientes a esta propiedad

- BorderStyle. Establece el tipo de borde para la ventana, en función del cual, tendrá un comportamiento determinado. La tabla 1 muestra los valores numéricos o constantes disponibles para esta propiedad.

Constante	**Valor**	**Descripción**
VbSNone	0	Sin borde
VbFixedSingle	1	Simple fijo. Puede incluir un cuadro del menú de control, una barra de título, un botón Maximizar y un botón Minimizar. Sólo puede cambiar de tamaño utilizando los botones Maximizar y minimizar

VbSizable	2	Tamaño ajustable. Puede cambiar de tamaño arrastrando con el botón en cualquiera de sus bordes. Este es el valor por defecto.
VbFixedDouble	3	Diálogo fijo. Puede incluir un cuadro del menú de control y una barra de título, pero no los botones Maximizar ni Minimizar. No puede cambiar de tamaño.
VbFixedToolWindow	4	Ventana fija. No puede cambiar de tamaño, muestra el botón Cerrar, y el texto de la barra de título aparece con un tamaño de fuente reducido. El formulario no aparece en la barra de tareas de Windows 95.
VbSizableToolWindow	5	Ventana de tamaño ajustable. Muestra el botón Cerrar y el texto de la barra de título aparece con un tamaño de fuente reducido. El formulario no aparece en la barra de tareas de Windows 95.

Tabla 1

En el formulario frmEncuesta, esta propiedad tiene el valor 2, vbSizable.

- ControlBox. Propiedad lógica (admite los valores True o False), que sirve para indicar si el formulario dispondrá de un menú de control. El valor para el formulario frmEncuesta es True.

- StartUpPosition. Establece la posición de comienzo para un formulario la primera vez que se visualiza. La tabla 2 muestra las constantes disponibles para esta propiedad.

Constante	**Valor**	**Descripción**
vbStartUpManual	0	El formulario no se visualiza en ningún lugar determinado
vbStartUpOwner	1	Centrado con respecto a una ventana padre o MDI a la que pertenece el formulario
vbStartUpScreen	2	Centrado en la pantalla
vbStartUpWindowsDefault	3	Esquina superior izquierda

Tabla 2. Valores para la propiedad StartUpPosition

El formulario frmEncuesta tendrá el valor 2, vbStartUpScreen en esta propiedad.

- Visible. Propiedad lógica. Si es True, el formulario se muestra al usuario, si es False, el formulario permanece oculto. Esta propiedad tendrá el valor True en frmEncuesta

- WindowState. Devuelve o asigna el estado visual del formulario. Los diferentes valores se muestran en la tabla 3.

Constante	Valor	Descripción
VbNormal	0	(Predeterminado) Normal
vbMinimized	1	Minimizada (minimizada a un icono)
vbMaximized	2	Maximizada (ampliado al tamaño máximo)

Tabla 3. Valores para la propiedad WindowState

La ventana frmEncuesta tendrá el valor 0, vbNormal en esta propiedad. Otras propiedades referentes a la posición y tamaño del formulario, son las siguientes:

- Left. Contiene la posición del borde interno izquierdo del formulario.
- Top. Contiene la posición del borde interno superior del formulario.
- Height. Altura del formulario.
- Width. Anchura del formulario.

La asignación de valores a las propiedades puede hacerse en tiempo de diseño o de ejecución, existiendo propiedades que sólo pueden manejarse en uno de estos dos ciclos de vida de la aplicación.

- Tiempo de diseño. Es el tiempo durante el que estamos creando los formularios, controles y código de la aplicación.
- Tiempo de ejecución. Tiempo durante el cual se ejecuta la aplicación.

Algunas propiedades para formularios, controles y objetos en general pueden ser comunes, es decir, pueden estar presentes en varios objetos al mismo tiempo. Este es el caso de la propiedad Name, las propiedades que indican el tamaño del objeto, etc.

Grabar el formulario

Durante el tiempo de diseño del formulario, es conveniente grabar nuestro trabajo en un fichero cada cierto tiempo o cuando efectuemos modificaciones sobre el mismo. Para ello seleccionaremos la opción del menú de VB *Archivo+Guardar <NombreForm>*, donde NombreForm será el formulario con el que estamos trabajando actualmente, o la combinación de teclado Ctrl+S. Esto abrirá una caja de diálogo estándar de Windows para introducir el nombre de fichero con el que se guardará el formulario. Los formularios están contenidos en ficheros con la extensión FRM, y es recomendable dar a estos ficheros, un nombre igual o similar al utilizado para la propiedad Name del formulario, de manera que podamos reconocerlos fácilmente y mantener cierta coherencia entre las denominaciones de los ficheros y los objetos que contienen.

Grabar el proyecto

Al igual que grabamos los formularios, es conveniente grabar el proyecto con el que estamos trabajando. Aunque podemos dejarle el nombre que le asigna VB por defecto, es mejor asignarle un nombre identificativo del trabajo que va a resolver la aplicación. Para ello, nos situaremos en la ventana Explorador de proyectos mediante la opción *Ver+Explorador de proyectos* del menú de VB, la barra de herramientas, o *Ctrl+R* desde el teclado.

Figura 14. Botón del Explorador de proyectos de la barra de herramientas.

Una vez en el explorador de proyectos, seleccionaremos el nombre del proyecto. A continuación abriremos la ventana de propiedades del proyecto, de igual forma que lo hemos hecho para el formulario y le daremos el nombre que estimemos oportuno, que en este caso será Encues1, la figura 15 muestra la ventana de propiedades con el nombre del proyecto.

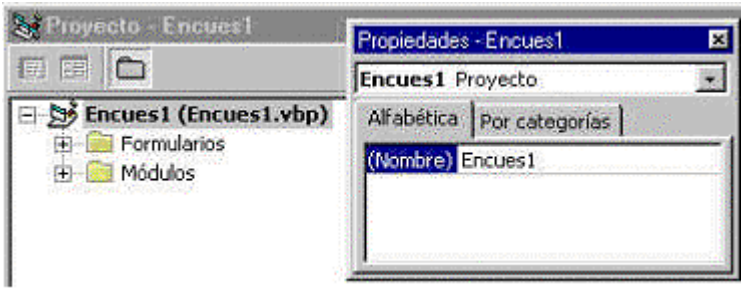


Figura 15. Ventana de propiedades del proyecto.

Asignado el nombre al proyecto, seleccionaremos la opción Archivo+Guardar proyecto del menú de VB, que nos mostrará un cuadro de diálogo mediante el cual guardaremos el proyecto. Los proyectos en VB, se guardan en ficheros con la extensión VBP, que contienen la información sobre todos los elementos que forman parte del proyecto. En este caso el fichero se llamará Encues1.VBP y en la figura 16 podemos ver como el Explorador de proyectos muestra entre paréntesis el nombre de fichero al que está asociado cada elemento.

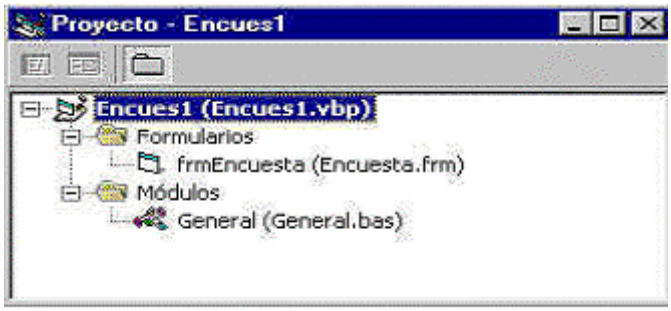


Figura 16. Proyecto con los elementos grabados en ficheros.

El procedimiento inicial Sub Main()

Un procedimiento es un fragmento de código o rutina, que se ejecuta durante el transcurso de un programa. En VB, el código se organiza en procedimientos, uno de cuyos tipos es el denominado Sub. Todos los procedimientos deben llevar un nombre de forma que puedan ser identificados y llamados desde diferentes puntos de la aplicación. Un procedimiento con el nombre Main, tendrá un tratamiento diferente del resto, ya que servirá como punto de entrada al programa, siendo el primero que se ejecute.

La ventaja del uso de Main, reside en que podemos realizar labores de inicialización en el programa, como puedan ser, recuperar del Registro la ruta de los datos de la aplicación, solicitar la clave del usuario, mostrar una pantalla de presentación, cargar el formulario principal de la aplicación, etc., en definitiva, preparar el entorno de trabajo de la aplicación antes de mostrarlo al usuario.

Para escribir el procedimiento Main o cualquier otro que no pertenezca directamente a ningún formulario, deberemos emplear un módulo de código estándar. La opción Proyecto+Agregar módulo del menú de Visual Basic, incluirá un nuevo módulo de código al proyecto. Es posible incluir un módulo ya existente, pero en este caso vamos a incluir uno nuevo. La figura 17 muestra el proyecto con el nuevo módulo y la ventana de edición de código para el mismo.

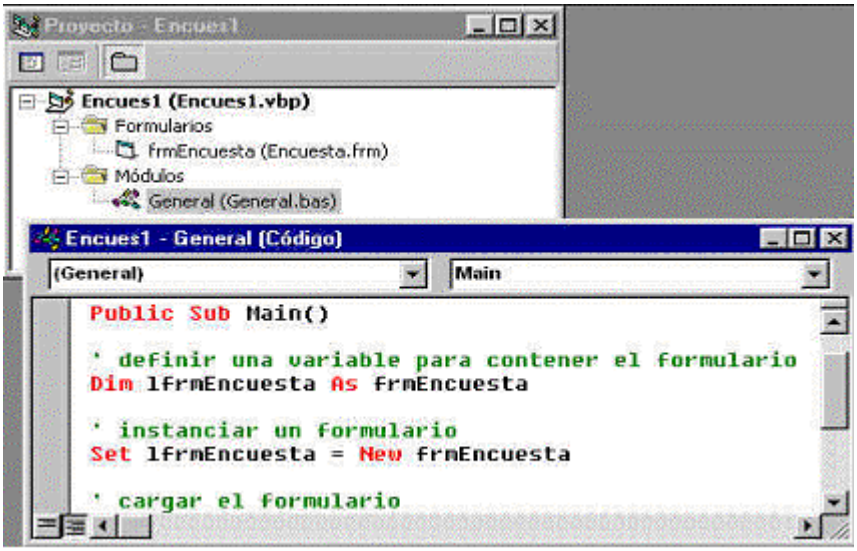


Figura 17. Nuevo módulo de código incluido en el proyecto

Al igual que con los anteriores elementos que hemos creado en este proyecto, desde el explorador de proyectos seleccionaremos este módulo y abriremos su ventana de propiedades para asignarle el nombre General. A continuación, en la ventana de edición de código del módulo, escribiremos el procedimiento Main, que vemos en el código fuente 1.

```
Public Sub Main()
' definir una variable para contener el formulario
Dim lfrmEncuesta As frmEncuesta
' instanciar un formulario
Set lfrmEncuesta = New frmEncuesta
' cargar el formulario
Load lfrmEncuesta
```

```
' mostrarlo
lfrmEncuesta.Show
End Sub
```

Código fuente 1

Como puede comprobar el lector, lo que hacemos aquí es declarar una variable del mismo tipo que el formulario del proyecto. Creamos a continuación un objeto formulario y lo asignamos a la variable, cargamos el objeto y lo visualizamos, de esta forma se inicia el programa.

Para guardar el módulo de código, seguiremos los mismos pasos ya explicados para los otros componentes del proyecto, teniendo en cuenta que este tipo de módulo se guarda en ficheros con extensión BAS, por ello el fichero que contendrá este módulo será General.BAS.

Establecer el objeto inicial de la aplicación

El haber escrito un procedimiento Main en nuestro programa no significa que la ejecución vaya a comenzar obligatoriamente por el mismo. Para que esto sea así, hemos de realizar previamente un pequeño ajuste en el proyecto. Nos situaremos pues en el explorador de proyectos y ejecutaremos la opción Proyecto+Propiedades de <NombreProy>, donde NombreProy será el nombre de proyecto actualmente en desarrollo, lo que mostrará una ventana de propiedades del proyecto ampliada, como podemos ver en la figura 18.

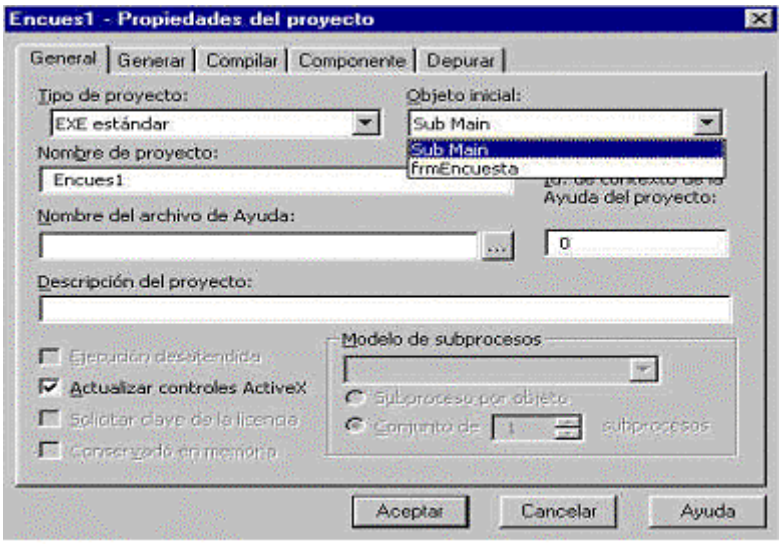


Figura 18

En esta ventana comprobamos que la propiedad Objeto inicial es una lista que contiene los nombres de los formularios incluidos en el proyecto y el procedimiento Sub Main. Esto quiere decir, que si en lugar de elegir Main como objeto inicial del proyecto, seleccionamos uno de los formularios, la aplicación comenzará por dicho formulario sin ejecutar Main ni todo el código de inicialización que pudiéramos haber incluido en dicho procedimiento.

En este caso, seleccionaremos para esta propiedad el valor Sub Main y aceptaremos la ventana de propiedades. A partir de ahora, cada vez que ejecutemos el programa, se buscará la existencia de un

procedimiento Main en cualquiera de los módulos del proyecto, para comenzar la ejecución por el mismo. Si no se hubiera creado dicho procedimiento, se producirá un error.

Volvemos a insistir en que el uso de Main es la práctica más recomendable, proporcionando una serie de ventajas comentadas en el apartado anterior. Por dicho motivo deberíamos considerar el iniciar siempre nuestras aplicaciones utilizando este procedimiento.

Ejecutar el proyecto

Durante la creación del proyecto, podemos ejecutar la aplicación desde el propio entorno de VB para observar su funcionamiento, mediante la opción Ejecutar+Iniciar del menú de VB, pulsando F5 o desde la barra de herramientas con el botón que aparece en la figura 19. De esta forma evitamos tener que generar el ejecutable de la aplicación cada vez que efectuemos una modificación.

Figura 19. Botón de inicio de la aplicación, en la barra de herramientas.

Al iniciar la aplicación, obtendremos una ventana similar a la mostrada en la figura 20.

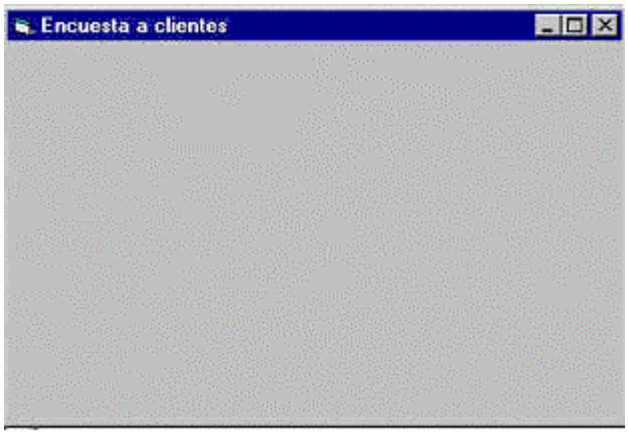


Figura 20

La primera impresión que nos podemos llevar, al observar el resultado del trabajo realizado hasta el momento, es que sólo hemos conseguido crear una simple ventana. Bien, esto es comprensible, pero miremos la situación desde otro punto de vista. Mediante un editor visual, hemos diseñado un formulario y establecido varias propiedades para el mismo, después hemos escrito una rutina de apenas diez líneas de código, con la que ponemos en funcionamiento el programa, consiguiendo una ventana dimensionable, que contiene un menú de control, título y todos los elementos estándar de una ventana Windows.

Todos aquellos lectores que hayan trabajado con alguna de las primeras versiones de herramientas de desarrollo para Windows, recordarán la complejidad y aridez de realizar lo que nosotros acabamos de hacer utilizando casi exclusivamente el ratón. Con otros entornos de programación, debíamos de escribir casi todo el código fuente de creación y bucle de eventos de la ventana, ficheros de recursos, compilar el trabajo realizado cada vez que se efectuaba una modificación en el código antes de probar

la aplicación y un largo etcétera de detalles, que ponían a prueba los nervios del más tranquilo sólo para crear una sencilla ventana.

Por este motivo, la aparente facilidad en crear y ejecutar un formulario es mérito de Visual Basic, que automatiza un gran número de tareas para hacer al programador su trabajo más fácil.

Controles estándar

Controles son todos aquellos elementos que se encuentran en el interior del formulario y que proporcionan al mismo un medio para comunicarse con el usuario.

En este apartado, veremos algunos de los controles más habituales que podemos encontrar en la mayoría de las aplicaciones Windows, siguiendo los pasos necesarios para insertarlos en el formulario del ejemplo, frmEncuesta. Comentaremos al mismo tiempo las características de cada uno de los controles empleados.

En primer lugar hemos de abrir el Cuadro de herramientas, una ventana que, a modo de paleta, contiene los controles disponibles para ser insertados en el formulario. La figura 21 muestra el aspecto de esta ventana, con los controles más habituales.

Figura 21. Cuadro de herramientas.

Insertar un control en el formulario

Podemos insertar un control del Cuadro de herramientas en el formulario de dos formas:

- Hacer doble clic en el control, con lo cual se insertará directamente en el centro del formulario con un tamaño predeterminado.

- Hacer clic en el control, el puntero del ratón cambiará de forma. Pasar al formulario, y en el lugar donde queramos situar el control, pulsar y arrastrar el ratón, dibujando el control. Al soltar, aparecerá el control en el formulario.

Cada vez que insertemos un control, tomará una serie de propiedades por defecto, que podremos cambiar para adaptar a los requerimientos de la aplicación.

TextBox

Consiste en un cuadro en el que puede escribirse información. Es lo más parecido a los campos de texto en las aplicaciones que funcionan bajo DOS, pero con mayores prestaciones.

Entre las propiedades de un TextBox, podemos destacar las siguientes:

- Name. Esta propiedad la tienen todos los controles, se trata del nombre usado para identificar al control dentro del código de la aplicación.
- Text. Cadena de caracteres que se visualiza en el control.
- MaxLength. Número máximo de caracteres que pueden introducirse en el control. Si se asigna 0, no hay límite de caracteres.
- BorderStyle. Indica el tipo de borde para el control. Los valores permitidos son:
  - 0. None. El control se muestra sin borde.
  - 1. Fixed Single. El control aparece con borde.
- PasswordChar. Asignando un caracter en esta propiedad, dicho caracter será visualizado en el TextBox en lugar del caracter real que teclee el usuario. Esta propiedad está especialmente indicada para tomar valores de tipo clave de usuario.
- ToolTipText. Introduciendo una cadena en esta propiedad, dicha cadena será visualizada en una viñeta flotante cuando el usuario sitúe el ratón sobre el control. De esta manera, obtenemos una breve ayuda sobre la tarea del control.
- MultiLine. Permite que el control disponga de una o varias líneas para editar el texto. Los valores admitidos son:
  - True. Se puede editar más de una línea de texto.
  - False. Sólo es posible escribir una línea en el control.
- ScrollBars. Valor que indica si un TextBox tiene barras de desplazamiento y de que tipo. Obviamente, esta propiedad tiene sentido utilizarla cuando configuramos el TextBox para editar varias líneas. La tabla 4 muestra los valores disponibles para esta propiedad.

Constante	**Valor**	**Descripción**
vbSBNone	0	(Predeterminado) Ninguna
vbHorizontal	1	Horizontal
vbVertical	2	Vertical
vbBoth	3	Ambas

Tabla 4. Valores para la propiedad ScrollBars

En frmEncuesta incluiremos los siguientes TextBox:

- txtNombre. Su cometido será guardar el nombre del cliente.
- txtApellidos. Contendrá los apellidos del cliente.
- txtObservaciones. Este control se configurará para editar varias líneas, de modo que el usuario pueda introducir un mayor número de anotaciones.

En todos estos TextBox, se ha eliminado el valor que por defecto tiene su propiedad Text, para que inicialmente no presenten ningún valor en el formulario. También se ha introducido una breve descripción de la tarea del control en la propiedad ToolTipText, de manera que el usuario sepa que valor introducir en cada uno. La figura 22 muestra la ventana con los TextBox insertados.

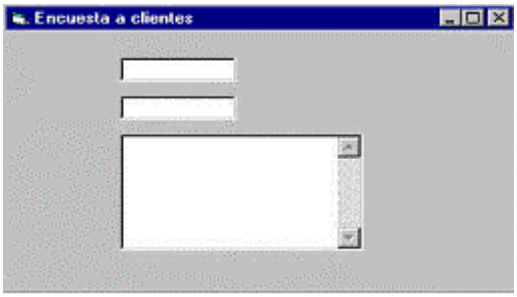


Figura 22. Formulario frmEncuesta con los TextBox creados

Label

Visualiza un texto informativo para el usuario. Se trata de un control estático, ya que el usuario no puede interaccionar con él, sin embargo, mediante código es posible cambiar el texto mostrado. Uno de los usos más corrientes para los controles Label, es indicar al usuario el tipo de valor que debe contener un TextBox u otro control situado junto a él, como vamos a hacer en el formulario frmEncuesta.

Como en este caso no vamos a manejar estos controles desde el código del programa, dejaremos todas sus propiedades con el valor que por defecto le asigna VB, excepto la propiedad Caption, que es la encargada de mostrar el texto de este control al usuario. La figura 23 muestra el aspecto del formulario después de incluir los controles Label para explicar la función de los TextBox.

Figura 23. Controles Label insertados en el formulario.

CheckBox

La finalidad de este control es proporcionar al usuario una elección del estilo Verdadero/Falso. Para ello, dispone de un pequeño cuadro que podemos marcar (signo de verificación) para aceptar la opción que propone el control, o desmarcar para no aceptarla.

Existe un tercer valor para el control, denominado Grayed o Atenuado, que consiste en que el control se encuentra marcado y bajo un tono gris, esto no significa sin embargo, que el control se encuentre deshabilitado, sigue estando operativo.

A propósito de habilitar o deshabilitar controles. La propiedad Enabled en un control, permite deshabilitar el mismo al asignarle el valor False, y habilitarlo al asignarle True. Un control o formulario deshabilitado, no responde a las acciones que el usuario pueda realizar sobre el mismo

Alguna de las propiedades destacables de este control son las siguientes:

- Value. Indica el estado del CheckBox, que puede ser uno de los siguientes valores:
  - 0.Unchecked. No marcado, valor por defecto.
  - 1.Checked. Marcado.
  - 2. Grayed. Atenuado.
- Style. Define el aspecto y comportamiento del control. Las constantes empleadas con este control las vemos en la tabla 5.

Constante	**Valor**	**Descripción**
vbButtonStandard	0	El control se muestra en su forma habitual
vbButtonGraphical	1	El control se muestra con el aspecto de un CommandButton. Al marcarlo, quedará hundido hasta que se vuelva a desmarcar.

Tabla 5.Valores para la propiedad Style

La figura 24 muestra un CheckBox con estilo gráfico, cuyo estado es marcado.

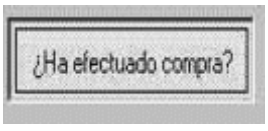


Figura 24. CheckBox gráfico marcado

Esta propiedad también está disponible para los controles CommandButton y OptionButton, que veremos más adelante. Su aspecto y comportamiento bajo estilo gráfico, estará en función de las características propias de cada control.

- Alignment. Esta propiedad permite establecer la posición del texto y del control. Los valores a emplear son los que aparecen en la tabla 6.

Constante	Valor	Descripción
vbLeftJustify	0	Valor por defecto. El control se encuentra en la parte izquierda y el texto asociado en la derecha.
vbRightJustify	1	El control se encuentra en la parte derecha y el texto asociado en la izquierda.

Tabla 6. Constantes para la propiedad Alignment del control CheckBox.

En el formulario del ejemplo situaremos un CheckBox al que llamaremos chkCompra, como vemos en la figura 25, que nos servirá para averiguar si el cliente al que se realiza la encuesta, ha comprado algún artículo (CheckBox marcado) o no ha realizado compra (CheckBox vacío).

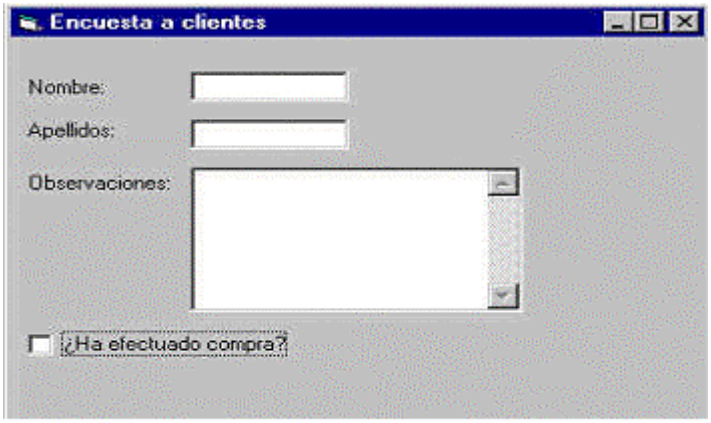


Figura 25. CheckBox chkCompra incluido en el formulario frmEncuesta.

Frame

Un control Frame permite agrupar controles de un mismo o distinto tipo, que compartan una característica común dentro del formulario. Uno de los usos más habituales es agrupar controles OptionButton que veremos a continuación.

Una vez insertado un Frame en el formulario, dibujaremos los controles necesarios dentro del mismo. El Frame reconocerá dichos controles y al ser desplazado por el programador dentro del formulario, los controles contenidos se moverán con el Frame. Si situamos un control dibujado fuera del Frame sobre este, al mover el Frame, el control no se moverá, debiendo hacerlo el programador aparte.

Otro aspecto importante de este control, es el hecho de que si asignamos False a su propiedad Enabled, no podremos acceder a ninguno de los controles contenidos en el Frame.

Para frmEncuesta insertaremos un Frame al que llamaremos fraMedio, y que se muestra en la figura 26.

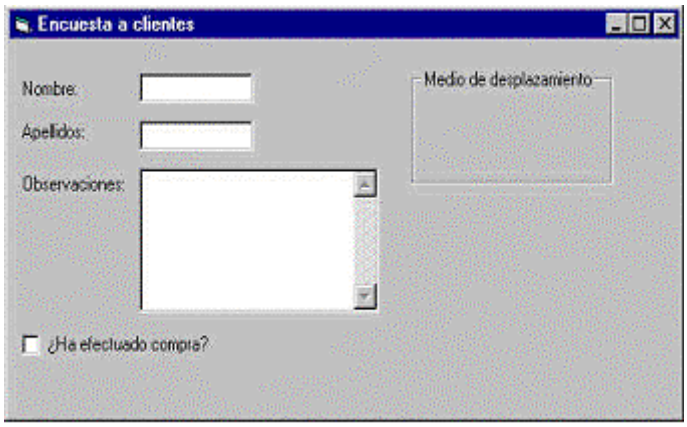


Figura 26. Control Frame fraMedio.

OptionButton

Este control, al igual que el CheckBox, muestra al usuario una selección que se puede activar o desactivar. Habitualmente se muestran en un grupo de varios OptionButton que presentan opciones entre las que se debe seleccionar una.

La diferencia fundamental entre los OptionButton y los CheckBox, es que en los primeros, la selección de un botón del grupo, desactiva el botón que hubiera hasta el momento; en los segundos es posible seleccionar más de un CheckBox al mismo tiempo.

Insertaremos en el Frame fraMedio dos OptionButton que llamaremos optPropio y optTrans, que utilizará el usuario para seleccionar el medio de transporte utilizado por el cliente del comercio. Compruebe el lector como la pulsación sobre uno de estos controles, desactiva el otro. La figura 27 muestra el resultado.

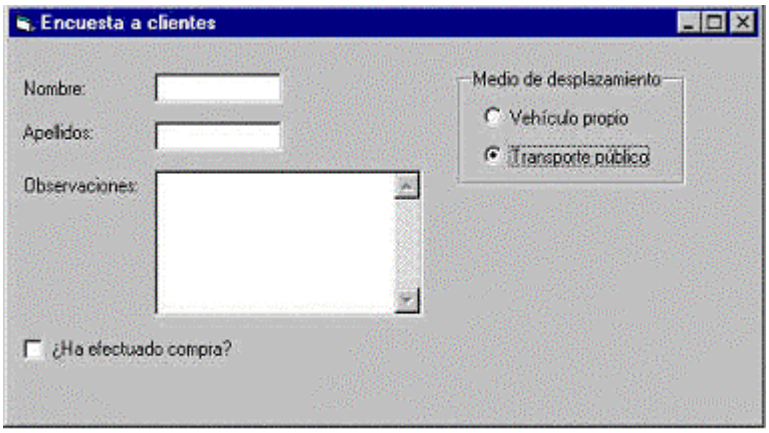


Figura 27. Formulario con los OptionButton.

ListBox

Visualiza una lista de elementos de tipo caracter, de los que podemos seleccionar uno o varios. Si el número de elementos es superior a los que puede mostrar el control, se añadirá automáticamente una barra de desplazamiento. Entre las propiedades de este control, podemos mencionar las siguientes:

- **List.** Array que contiene los elementos de la lista. Para agregar o eliminar elementos podemos usar esta misma propiedad en la ventana de propiedades del control o los métodos AddItem() y RemoveItem() en el código del programa.

- **IntegralHeight**. Esta propiedad permite obligar a que se muestren los elementos por completo en la lista, los valores a usar son los siguientes:

    - True. Ajusta el tamaño del control para que los elementos de la lista se muestren por completo.

    - False. No ajusta el tamaño del control, por lo que los elementos en la parte superior e inferior de la lista se verán de forma parcial.

- **Sorted**. Permite ordenar alfabéticamente los elementos de la lista si el valor de esta propiedad es True.

- **MultiSelect**. Permite realizar selecciones de varios elementos de la lista simultáneamente. Los valores para esta propiedad son:

    - 0. No se pueden efectuar selecciones múltiples. Valor por defecto.

    - 1. Selección múltiple simple. Al hacer clic sobre un elemento o pulsar la tecla Espacio, se selecciona dicho elemento.

    - 2. Selección múltiple extendida. Al pulsar la tecla Mayúsculas y hacer clic, o pulsar Mayúsculas y una de las teclas de dirección (Flecha Arriba, Abaja, Izquierda, Derecha), se seleccionan todos los elementos existentes entre la última selección y la actual.

Si por otra parte se pulsa Control y se hace clic sobre un elemento, este único elemento se activará o desactivará en función de su estado actual.

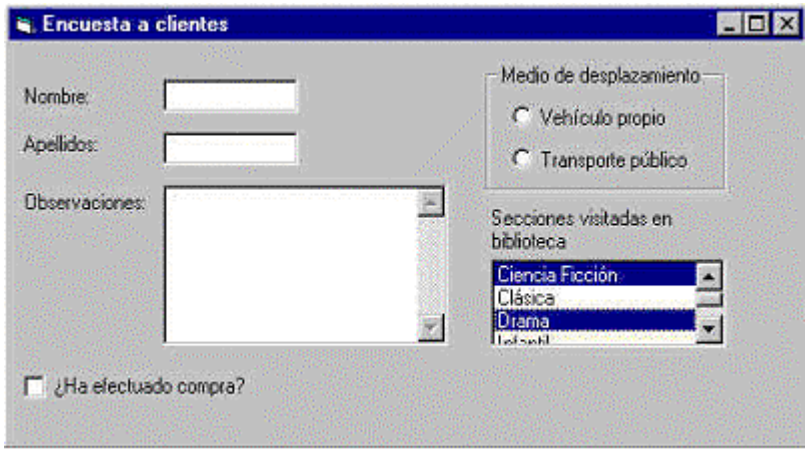


Figura 28. Formulario mostrando el ListBox lstLibros.

Para el formulario frmEncuesta dibujaremos un ListBox al que llamaremos lstLibros, mostrado en la figura 28, el cual contendrá las diferentes secciones sobre libros del establecimiento, de forma que el usuario pueda seleccionar cual ha visitado el cliente encuestado. Debido a que un cliente puede ver varias secciones, el tipo de selección en este ListBox será múltiple simple. Para facilitar la localización de las secciones, los elementos de la lista aparecerán ordenados.

ComboBox

Este control está basado en dos controles vistos con anterioridad: TextBox y ListBox, ya que contiene una zona en la que el usuario puede editar texto y otra con una lista de valores. Según el estilo del control, la lista puede estar visible en todo momento o permanecer oculta y ser desplegada por el usuario. Por dicho motivo la propiedad Style para este control puede tomar los valores mostrados en la tabla 7.

Constante	Valor	Descripción
vbComboDropDown	0	(Predeterminado) Cuadro combinado desplegable. Incluye una lista desplegable y un cuadro de texto. El usuario puede seleccionar datos en la lista o escribir en el cuadro de texto.
VbComboSimple	1	Cuadro combinado simple. Incluye un cuadro de texto y una lista, que no se despliega. El usuario puede seleccionar datos en la lista o escribir en el cuadro de texto. El tamaño de un cuadro combinado simple incluye las partes de edición y de lista. De forma predeterminada, el tamaño de un cuadro combinado simple no muestra ningún elemento de la lista. Incremente la propiedad Height para mostrar más elementos de la lista.
vbComboDrop-DownList	2	Lista desplegable. Este estilo sólo permite la selección desde la lista Desplegable

Tabla 7. Valores para la propiedad Style de un ComboBox.

Figura 29. Formulario con el control cboMusica activo.

En la ventana frmEncuesta insertaremos un ComboBox llamado cboMusica de estilo simple y con sus elementos ordenados. De esta forma, si el estilo musical no se halla entre los elementos de la lista se podrá escribir en la zona de texto del control. La figura 29 muestra el formulario con el ComboBox desplegado.

Si deseamos encontrar un elemento de la lista rápidamente, sólo tenemos que escribir su primera letra y pulsar Flecha Abajo o Arriba, el ComboBox buscará la cadena más parecida.

CommandButton

Representa un botón que al ser pulsado, desencadena una determinada acción. Es posible definir una tecla de atajo al igual que en los menús, en la propiedad Caption, mediante el uso del signo "&" junto a la tecla aceleradora, con lo cual quedará resaltada la letra junto a la que se sitúe.

En el formulario frmEncuesta se han creado dos CommandButton: cmdAceptar y cmdCancelar, el primero efectúa una serie de operaciones, en función de la manipulación que se haya llevado a cabo con el resto de controles del formulario, esto lo veremos en la parte dedicada al código de este formulario. El otro botón finaliza la ejecución del formulario descargándolo, también lo veremos en su código.

Figura 30. Formulario al completo, incluyendo los CommandButton.

Manipulación de los controles mediante código

En el estado actual, es posible ejecutar la aplicación, sin embargo, sólo conseguiríamos accionar los controles del formulario sin un resultado efectivo. El siguiente paso consiste en añadir código al formulario para obtener el resultado de las selecciones que haga el usuario del programa, concretamente cuando se pulse el botón cmdAceptar. Para ello hemos de codificar el evento Click() de dicho control, que es el que se invoca al pulsar un CommandButton.

Nos situaremos en la ventana de edición de código del formulario mediante la opción Ver+Código del menú de VB o haciendo doble clic en el botón a codificar, lo que nos llevará inmediatamente a su evento Click(), el cual estará vacío y en él escribiremos las líneas mostradas en el código fuente 2.

```
Private Sub cmdAceptar_Click()
' código que se ejecutará al pulsar el botón
Dim lnInd As Integer
' datos personales del cliente
MsgBox "Contenido del campo Nombre: " & Me.txtNombre.Text _
   , , "Información"
MsgBox "Contenido del campo Apellidos: " & Me.txtApellidos.Text _
   , , "Información"
' anotaciones varias
MsgBox "Contenido del campo Observaciones: " & vbCrLf & _
   Me.txtObservaciones.Text, , "Información"
' información sobre compras
If Me.chkCompra.Value Then
   MsgBox "El cliente ha comprado artículos", , "Información"
Else
   MsgBox "El cliente no ha comprado artículos", , "Información"
End If
' transporte utilizado por el cliente
If Me.optPropio.Value Then
   MsgBox "El cliente ha llegado al establecimiento" & vbCrLf & _
      "utilizando su propio vehículo", , "Información"
End If
If Me.optTransp.Value Then
   MsgBox "El cliente ha llegado al establecimiento" & vbCrLf & _
      "utilizando transporte público", , "Información"
End If
' visualizar las secciones de libros
' en las que ha estado el cliente
For lnInd = 0 To Me.lstLibros.ListCount - 1
   If Me.lstLibros.Selected(lnInd) Then
      MsgBox Me.lstLibros.List(lnInd), , _
         "Sección de biblioteca visitada"
   End If
Next
' mostrar el estilo musical seleccionado
MsgBox Me.cboMusica.Text, , _
   "Estilo musical seleccionado por el cliente"
End Sub
```

Código Fuente 2

La ventana de código tiene un aspecto similar al mostrado en la figura 31.

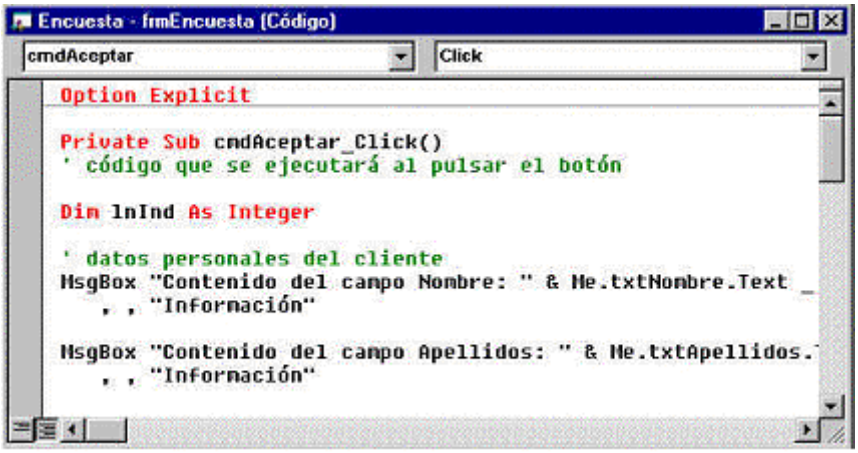


Figura 31. Ventana de edición de código.

En este procedimiento se emplea la función MsgBox(), que muestra el conocido cuadro de diálogo con un mensaje para el usuario.

Mediante la propiedad Value en los controles CheckBox y OptionButton, usada en la estructura de control If Then...End If, podemos saber si se ha pulsado alguno de estos controles.

En cuanto al ListBox, el uso combinado de las propiedades ListCount, Selected y List en una estructura de control For...Next nos sirve para averiguar los elementos de la lista seleccionados.

Si queremos escribir una línea de comentario para el código, hemos de poner en primer lugar el signo de comilla simple " ' " seguido por el texto de comentario.

Al estar dentro del módulo de código de un formulario, y siendo este un objeto, la palabra clave *Me* (Ej.: Me.txtObservaciones) nos sirve para referirnos al propio formulario desde dentro de su módulo de código. Por este motivo las siguientes líneas serían equivalentes, siendo recomendable utilizar la última, por estar más próxima a una sintaxis OOP.

Caption = txtNombre.Text ‘ sin hacer referencia al objeto formulario

frmEncuesta.Caption = frmEncuesta.txtNombre.Text ‘ utilizando el nombre del formulario

Me.Caption = Me.txtNombre.Text ‘ utilizando la referencia a sí mismo del formulario

En lo que respecta al botón cmdCancelar, este se ocupa de descargar el formulario, finalizando de esta forma el programa. El procedimiento Click() de este botón es el siguiente:

```
Private Sub cmdCancelar_Click()
Unload Me
End Sub
```

Código fuente 3

Creación del ejecutable del programa

Una vez que disponemos de una versión estable del programa, podemos crear el fichero EXE con la opción *Archivo+Generar <NombreApp>.exe...*, donde NombreApp es el nombre del proyecto que se utiliza para el fichero ejecutable. Se abrirá una caja de diálogo en la que podemos cambiar el nombre por defecto del ejecutable. Dicha caja dispone de un botón *Opciones...* que mostrará a su vez una ventana con diversa información para crear el ejecutable.

En la pestaña Generar de esta ventana, podemos introducir el número de versión de la aplicación, comentarios sobre el autor, compañía, producto ,etc.; un icono perteneciente a alguno de los formularios existentes, para que sirva como icono del programa, argumentos que serán enviados al programa cuando se ejecute, y declaraciones para las constantes utilizadas en la compilación condicional.

Para la pestaña Compilar, podemos seleccionar el tipo de compilación: P-Code o código nativo, y dentro de este último, varios tipos de optimizaciones.

En la pestaña Generar de esta ventana (figura 32), podemos introducir el número de versión de la aplicación, comentarios sobre el autor, compañía, producto ,etc.; un icono perteneciente a alguno de

los formularios existentes, para que sirva como icono del programa, argumentos que serán enviados al programa cuando se ejecute, y declaraciones para las constantes utilizadas en la compilación condicional.

Para la pestaña Compilar (figura 33), podemos seleccionar el tipo de compilación: P-Code o código nativo, y dentro de este último, varios tipos de optimizaciones.

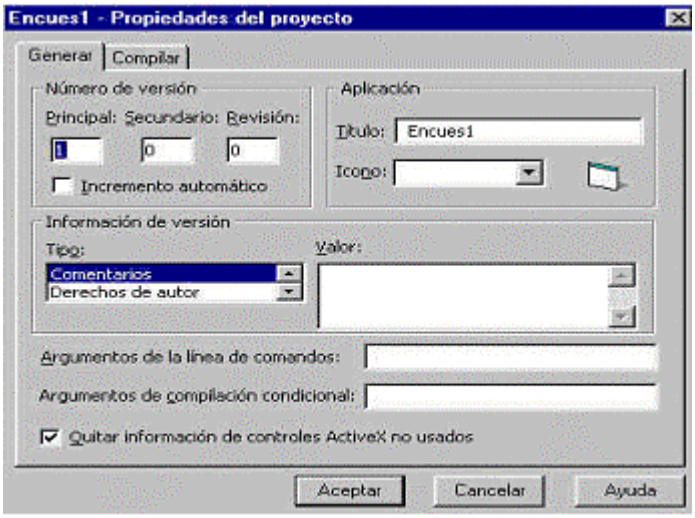


Figura 32. Propiedades del proyecto, pestaña Generar.

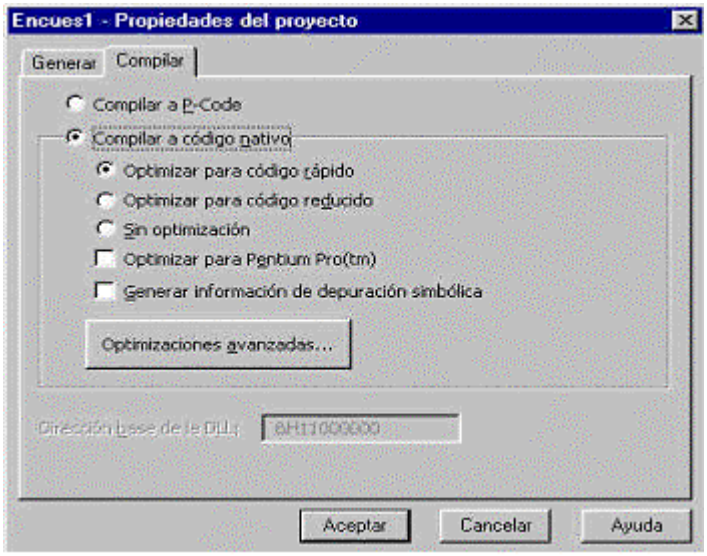


Figura 33. Propiedades del ejecutable, pestaña Compilar.

Distribución de la aplicación

Construido el ejecutable, ya sólo queda la creación de un programa que se ocupe de la instalación de nuestra aplicación en los equipos que necesiten usarla. Para ello usaremos la utilidad Asistente para empaquetado y distribución, que podemos encontrar en el menú de Herramientas de Microsoft Visual

Studio 6.0, correspondiente al grupo de programas de Microsoft Visual Studio 6.0, y al que podemos acceder mediante la barra de tareas de Windows, tal como vemos en la figura 34.

Figura 34. Grupo de programas de Visual Basic.

Iniciado el asistente, se mostrará una pantalla introductoria al mismo, en donde podemos elegir el modo de distribución o empaquetado del proyecto, como se puede ver en la figura 35.

En este caso, después de seleccionar el fichero que contiene el proyecto, pulsaremos sobre el botón Empaquetar para crear una instalación normal.

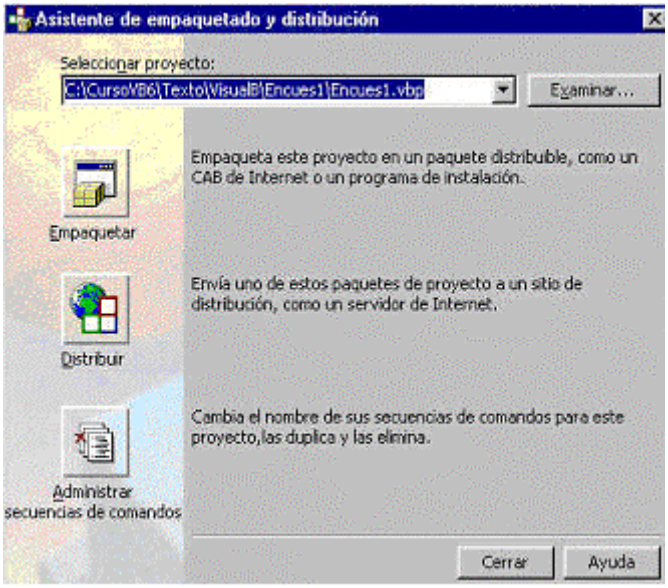


Figura 35. Pantalla principal del asistente.

El modo de empleo de este asistente y en general, de todos los incorporados en VB se basa en completar, si es necesario, las pantallas en cada paso y pulsar el botón Siguiente para pasar al próximo. Si hemos olvidado algún aspecto o tenemos que modificar alguno de los pasos realizados con anterioridad, pulsaremos el botón Anterior hasta llegar al punto requerido.

En cualquier momento podemos suspender el asistente pulsando el botón Cancelar, y obtener ayuda sensible al contexto pulsando Ayuda.

Una vez elegido el tipo de operación a realizar, seleccionaremos el tipo de empaquetado (figura 36) para la aplicación: instalación estándar. También podemos crear un fichero de dependencias con la información que necesita el instalador sobre los ficheros que componen la aplicación.

Figura 36. Selección del tipo de instalación.

El siguiente paso consistirá en indicar la unidad y carpeta de nuestro equipo en la que se realizará el empaquetado de la aplicación. Si instalamos desde disquetes, lo más habitual será seleccionar la unidad A y su directorio raíz como carpeta (figura 37).

Figura 37. Selección de la carpeta para el empaquetado.

Continuaremos con la selección de ficheros que componen el programa. El asistente muestra una lista con los que por defecto se van a incluir en la instalación (figura 38), pudiendo por nuestra parte, añadir cuantos más necesitemos, como por ejemplo, una base de datos, pulsando el botón Agregar de esta ventana.

Debemos especificar el modo en que se van a generar los ficheros .CAB (figura 39), que contendrán la instalación: tamaño y cantidad. En el caso de instalar desde disquetes, deberemos seleccionar la opción para crear múltiples ficheros de este tipo.

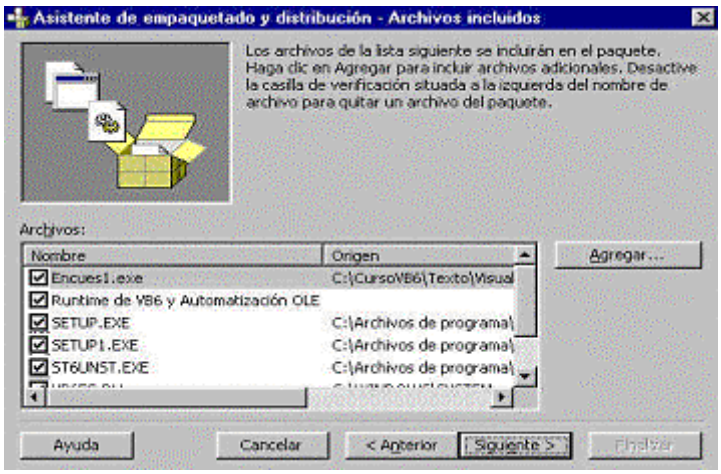


Figura 38. Relación de ficheros que formarán parte de la instalación.

Figura 39. Opciones para los ficheros .CAB.

Figura 40. Introducir título para la instalación.

Seguidamente indicaremos el título que aparecerá en la pantalla durante el proceso de instalación. Por defecto será el mismo del proyecto, como aparece en la figura 40.

Es posible indicar la ubicación en el menú de inicio de Windows de la aplicación (figura 41), pudiendo crear grupos, elementos de menú y sus propiedades. Dicho trabajo lo realizaremos en el siguiente paso.

Figura 41. Configuración del menú Inicio para la aplicación.

El emplazamiento de ciertos ficheros de la aplicación, se basa en los valores de las macros utilizadas por este asistente, que le indican la ubicación en el equipo para dicho fichero. En este paso, es posible, cambiar esa ubicación por defecto, situando el fichero en otro lugar, como podemos ver en la figura 42.

Figura 42. Emplazamiento de los ficheros en la instalación.

En el siguiente paso, se muestran los ficheros susceptibles de ser compartidos por otras aplicaciones del equipo. Para poder compartirlos, lo único que debemos hacer es seleccionar la casilla de verificación adjunta a cada uno de ellos.

Figura 43. Establecer ficheros compartidos.

Por último aparecerá la pantalla final del asistente, en la que podremos dar un nombre a la instalación (figura 44). Pulsando el botón Finalizar, se generará el programa de instalación en el soporte seleccionado por el usuario.

Figura 44. Final de la instalación.

Este es, en líneas generales, el proceso llevado a cabo en la creación de un programa de instalación para una aplicación. Dependiendo de la tarea a desempeñar por la aplicación, pueden aparecer algunos pasos adicionales en este asistente.

Ampliar la primera aplicación

Hasta el momento, la aplicación que estamos desarrollando resulta útil para una toma sencilla de datos. Pero ¿qué sucedería si necesitásemos más de un formulario para realizar operaciones distintas

en el mismo programa?. El proyecto de ejemplo Encues2, partiendo del contenido de Encues1, incorporará los nuevos elementos que veremos a partir de ahora.

Debido a que en este tema estamos realizando el proceso de desarrollo de una aplicación partiendo de cero, el lector observará alguna diferencia entre el proyecto que cargue en el entorno y el estado actual que mostramos en el tema. Esto es debido a que el proyecto de ejemplo contiene la aplicación terminada, y en este punto del tema, nos faltan algunos controles y código que añadir.

Supongamos que además del formulario para realizar la encuesta a los clientes, el establecimiento quiere controlar en otro formulario, los artículos comprados por el cliente. Crearemos para esta labor un nuevo formulario al que llamaremos frmCompras, que podemos ver en la figura 45.

Figura 45. Formulario frmCompras.

Como puede observar el lector, se trata de escribir en el ComboBox de este formulario, los artículos comprados por el cliente e incorporarlos a la lista pulsando el botón Agregar artículo. Al pulsar el botón Aceptar, se mostrarán uno a uno los artículos comprados. Ya que las operaciones son similares a las vistas en el anterior formulario, no se incluyen en este texto. El lector interesado, puede consultarlas desde la ventana de código fuente del formulario.

Llegados a este punto, nos encontramos con el problema de manejar los dos formularios de la aplicación. Para poder trabajar con ambos, deberíamos abrirlos al mismo tiempo al iniciarse la aplicación, o uno de los formularios tendría que llamar al otro. En cualquier caso, la aplicación daría una evidente sensación de desorganización, con dos formularios aislados funcionando sin ningún tipo de conexión.

El anterior modelo de aplicación es el conocido como Interfaz de Documento Sencillo (SDI), en el cual, la aplicación consta por lo general, de un formulario en el que se realiza todo el trabajo.

Sin embargo, cuando tenemos que manejar varios formularios, es más adecuado el empleo de una aplicación que siga el modelo de Interfaz de Documento Múltiple (MDI). En este tipo de aplicación, disponemos de un formulario principal o MDI, que actúa como contenedor de formularios. A los formularios contenidos en un formulario MDI, se les conoce como formularios secundarios MDI, formularios hijo o documentos; su particularidad reside en que su espacio de trabajo está limitado al tamaño del formulario MDI, no pudiendo desplazarse fuera de los límites de dicho formulario.

Incluir un formulario MDI

Para añadir un formulario MDI al proyecto, podemos seleccionar la opción Proyecto+Agregar formulario MDI del menú de VB, o bien, abrir el menú contextual del Explorador de proyectos y seleccionar la opción que realiza esta misma operación, con lo cual se añadirá un nuevo formulario de

este tipo al proyecto. La figura 46, muestra el contenido del proyecto Encuesta, al que se ha agregado un formulario MDI que llamaremos mdiInicio.

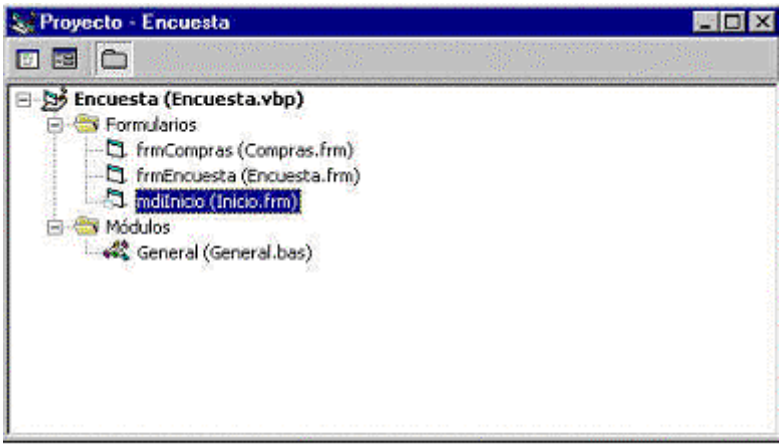


Figura 46. Proyecto Encuesta con un formulario MDI.

Distinguiremos rápidamente el formulario MDI, ya que tiene un fondo más oscuro que un formulario normal, como se muestra en la figura 47.

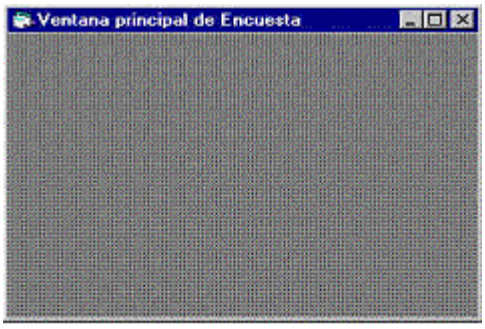


Figura 47. Formulario MDI del proyecto Encuesta.

En lo que respecta a las propiedades, destacaremos las siguientes:

- AutoShowChildren. Contiene un valor que indica si los formularios secundarios MDI se visualizan al ser cargados. Los valores de esta propiedad son los siguientes:
  - True. Valor por defecto. Los formularios secundarios se muestran cuando se cargan.
  - False. Los formularios secundarios no se muestran cuando se cargan, debemos emplear su método Show() para visualizarlos.
- WindowState. Contiene un valor que sirve para mostrar la ventana en un determinado estado visual. Esta propiedad está disponible tanto para formularios MDI como formularios normales. La tabla 8 muestra los valores que puede tomar esta propiedad.

Constante	Valor	Descripción
VbNormal	0	(Predeterminado) Normal
VbMinimized	1	Minimizada (minimizada a un icono)
VbMaximized	2	Maximizada (ampliado al tamaño máximo)

Tabla 8. Valores para la propiedad WindowState.

Para el formulario mdiInicio y para la mayoría de los formularios MDI en general, esta propiedad se establece a vbMaximized, ya que al tener que mostrar formularios secundarios en su interior, es más cómodo visualizarla de modo que ocupe todo el área disponible.

Ahora sólo queda modificar el procedimiento Main(), para que sea este formulario el primero en cargarse cuando se inicie la aplicación, como podemos ver en el código fuente 4.

```
Public Sub Main()
' instanciar un formulario
Set gmdiInicio = New mdiInicio
' cargar el formulario
Load gmdiInicio
' mostrarlo
gmdiInicio.Show
End Sub
```

Código fuente 4

La variable que contiene el formulario MDI se ha declarado como global en la zona de declaraciones del módulo, ya que los formularios secundarios necesitarán acceder a ella.

Referente al código de este formulario debemos destacar el procedimiento de evento QueryUnload(). Este evento se produce antes del cierre de un formulario, tanto MDI como normal, y el programador puede incluir por ejemplo, código para comprobar si se ha grabado toda la información pendiente, o mostrar un mensaje para cancelar el cierre del formulario. El código fuente 5 muestra las líneas que contiene este procedimiento en el formulario frmInicio.

```
Private Sub MDIForm_QueryUnload(Cancel As Integer, UnloadMode As Integer)
Dim lnRespuesta As Integer
lnRespuesta = MsgBox("¿Seguro que desea finalizar la aplicación?", _
   vbYesNo + vbDefaultButton2, "Atención")
' si se asigna a Cancel un valor
' distinto de cero, no se cierra
' el formulario
If lnRespuesta = vbNo Then
   Cancel = 1
End If
End Sub
```

Código Fuente 5

Este procedimiento recibe dos parámetros: Cancel, que contiene por defecto el valor 0. Si en el procedimiento se cambia dicho valor, el formulario no se cerrará. El otro parámetro es UnloadMode,

que informa de la causa que ha provocado la llamada a este evento. Los valores de este parámetro pueden manejarse de forma más cómoda a través de las constantes mostradas en la tabla 9.

Constante	Valor	Descripción
vbFormControlMenu	0	El usuario eligió el comando Cerrar del menú Control del formulario.
vbFormCode	1	Se invocó la instrucción Unload desde el código.
vbAppWindows	2	La sesión actual del entorno operativo Microsoft Windows está finalizando.
vbAppTaskManager	3	El Administrador de tareas de Microsoft Windows está cerrando la aplicación.
vbFormMDIForm	4	Un formulario MDI secundario se está cerrando porque el formulario MDI también se está cerrando.

Tabla 9. Valores que puede tomar el parámetro UnloadMode del procedimiento QueryUnload().

En lo que respecta a los formularios frmEncuesta y frmCompras, a partir de ahora será necesario que se comporten como formularios secundarios MDI. Para ello, debemos establecer su propiedad MDIChild a True. Esta propiedad indica, cuando su valor es True, que el formulario se comportará como secundario MDI o hijo de la ventana MDI en una aplicación, quedando su zona de actuación limitada al área de la ventana MDI. Si por el contrario, el valor de la propiedad es False, su comportamiento visual no dependerá de la MDI, pudiendo desplazarse fuera de los límites de dicha ventana.

Crear un menú para el formulario MDI

Creado el formulario MDI, hemos de establecer un medio mediante el cual, este formulario pueda cargar los formularios secundarios de la aplicación. Dicho medio es el típico menú de opciones existente en la mayoría de las ventanas.

Para crear un menú, utilizaremos el Editor de menús (figura 49), al cual accederemos mediante la opción del menú de VB Herramientas+Editor de menús, la combinación de teclado Ctrl+E o su botón en la barra de herramientas (figura 48).

Figura 48. Botón de acceso al Editor de menús en la barra de herramientas de VB.

Debido a que la confección de menús se trata con mayor profundidad en el tema Diseño de menús, no realizaremos una explicación detallada de este proceso, remitiendo al lector a dicho tema para los detalles involucrados en la creación de menús.

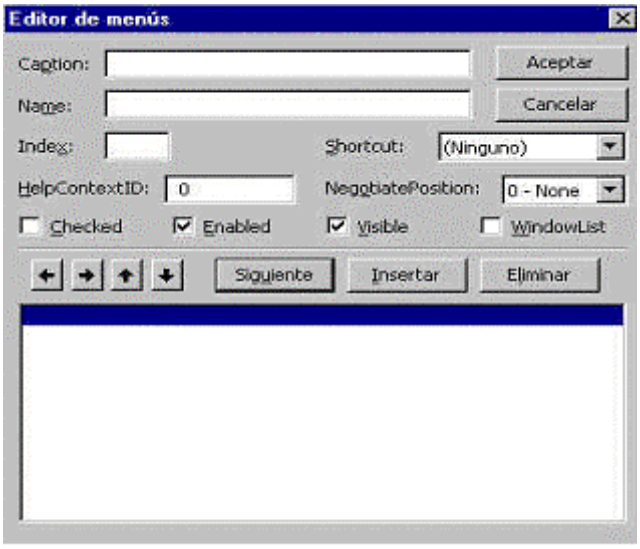


Figura 49. Editor de menús.

Una opción de menú es en realidad un control, al igual que un TextBox o un CommandButton, sólo que con una apariencia visual diferente. Esto podemos comprobarlo una vez creado el menú, si abrimos la ventana de propiedades del formulario, veremos que cada opción de menú aparece como un control en dicha ventana.

Para crear una opción de menú, escribiremos su nombre en la propiedad Caption del editor y el nombre que usaremos en el código para dicha opción en la propiedad Name. Las opciones serán situadas en la lista de la parte inferior del editor. Si queremos que una opción sea de un nivel inferior, pulsaremos el botón que tiene una flecha hacia la derecha, de forma que dicha opción quedará marcada con una serie de puntos que indican su nivel. Para hacer que una opción de nivel inferior suba de nivel, pulsaremos el botón que tiene una flecha hacia la izquierda. Las opciones que no tienen puntos en la lista son las que forman parte del nivel superior o barra de menús. La figura 50 muestra el editor de menús, una vez creadas las opciones para el formulario mdiInicio.

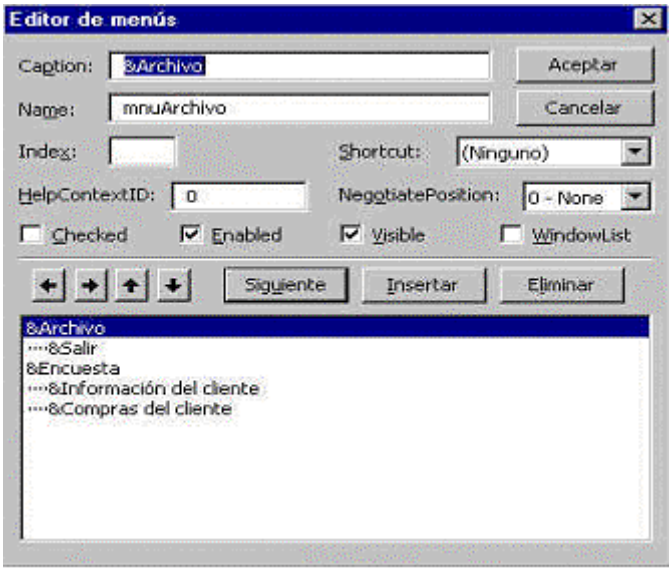


Figura 50. Opciones para el menú del formulario frmInicio.

Terminada la creación de opciones de menú, asociaremos a cada una el código que queramos ejecutar. Esta labor es similar a la que vimos para añadir código a un CommandButton, debemos situarnos en la ventana de edición de código del formulario mediante el menú de VB, opción Ver+Código, o bien, desde el formulario en modo de diseño, seleccionar la opción de menú a codificar. Ambos modos de operar, nos llevarán al evento Click() de dicha opción de menú, en el cual escribiremos el código necesario. El código fuente 6 que aparece a continuación, contiene el código de las opciones del menú de frmInicio correspondientes a la finalización del programa y la carga de los formularios secundarios.

```
Private Sub mnuEnInfo_Click()
' cargar el formulario para realizar
' encuestas a clientes
Dim lfrmEncuesta As frmEncuesta
Set lfrmEncuesta = New frmEncuesta
Load lfrmEncuesta
lfrmEncuesta.Show
' deshabilitar esta opción de menú
Me.mnuEnInfo.Enabled = False
End Sub
' ------------------------------------
Private Sub mnuEnCompras_Click()
' cargar el formulario para llevar
' el control de las compras realizadas
Dim lfrmCompras As frmCompras
Set lfrmCompras = New frmCompras
Load lfrmCompras
lfrmCompras.Show
' deshabilitar esta opción de menú
Me.mnuEnCompras.Enabled = False
End Sub
' ------------------------------------
Private Sub mnuArSalir_Click()
' finalizar la ejecución
Unload Me
End Sub
```

Código fuente 6

Observe el lector que al cargar los formularios secundarios, se deshabilita la opción de menú empleada para realizar dicha carga. El motivo de este proceder se debe a que dejando la opción disponible, podríamos cargar la misma ventana varias veces, y en este caso, sólo nos interesa tener una ventana de cada tipo abierta a la vez. Al cerrar la ventana, volveremos a habilitar la opción correspondiente en el formulario MDI, en el evento que descarga el formulario secundario. Veamos una muestra en el código fuente 7.

```
Private Sub Form_Unload(Cancel As Integer)
' habilita la opción del menú en el formulario
' MDI que carga este formulario
gmdiInicio.mnuEnInfo.Enabled = True
End Sub
```

Código fuente 7

Llegados a este punto, la conversión de la aplicación al modelo MDI ha sido completada satisfactoriamente. Al ejecutar el proyecto, dispondremos de una ventana inicial MDI, que servirá de

contenedora para los formularios dedicados a realizar la toma de datos para la encuesta, lo que dará al usuario una mayor sensación de control y seguridad.

Operaciones estándar mediante el control CommonDialog

El control CommonDialog proporciona al programador, un medio de comunicación estándar con el usuario, para las operaciones más habituales de configuración del color, tipo de letra, impresora, apertura/grabación de ficheros y acceso a la ayuda.

El uso de este control proporciona ventajas evidentes. Por una parte, el programador no necesita emplear tiempo en desarrollar, por ejemplo, un formulario y su código correspondiente, para realizar la apertura o grabación de ficheros, el control ya incorpora todo lo necesario, sólo es necesario asignar valores a sus propiedades y mostrarlo. Por otro lado, el usuario de la aplicación sólo necesitará aprender una vez a utilizarlo, ya que al ser un elemento común a todas las aplicaciones Windows, en nuestras siguientes aplicaciones o las de otro programador, el modo de manejo será idéntico.

Si el Cuadro de herramientas de VB no presenta este control, podemos incluirlo abriendo la ventana Componentes, a la cual accederemos mediante el menú de VB, opción Proyecto+Componentes, o la combinación de teclado Ctrl+T. Una vez en esta ventana, seleccionaremos el componente Microsoft Common Dialog Control 6.0, tal como se muestra en la figura 51.

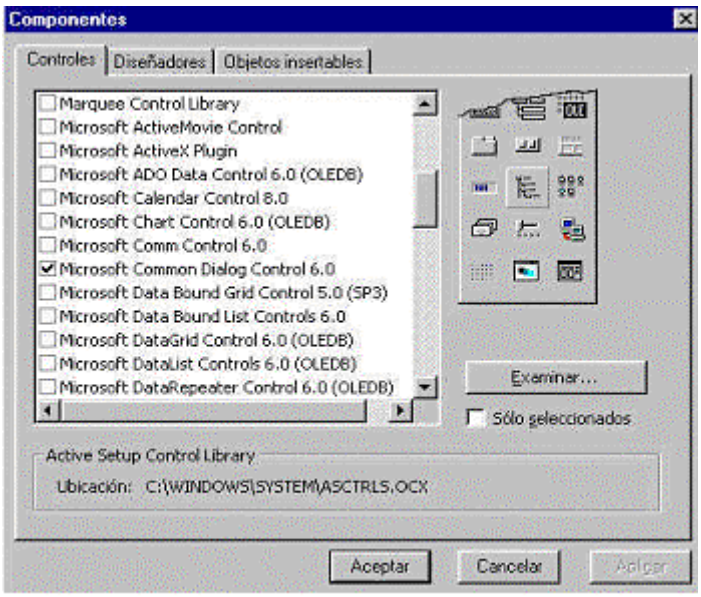


Figura 51. Selección del control CommonDialog en la ventana de Componentes de VB.

Con esta acción, ya podemos disponer del control CommonDialog en el Cuadro de herramientas.

Figura 52. Control CommonDialog del Cuadro de herramientas.

Un control CommonDialog puede ser usado tanto en un form MDI como en uno normal. La forma de insertarlo en el formulario es la misma que para cualquiera de los controles vistos anteriormente, con

la particularidad de que una vez en el formulario, este control se presenta en forma de icono, sin admitir ningún tipo de edición visual, por lo que todo el trabajo a realizar con él, debe hacerse desde la ventana de propiedades y el código del programa.

Una de las propiedades más importantes en un CommonDialog es Flags, a la que se asignará, en función del tipo de diálogo a mostrar una o más constantes para configurar dicho diálogo.

Vamos a incluir en el formulario mdiInicio, un control CommonDialog al que llamaremos dlgOperaciones. Ya que este control puede mostrar diferentes aspectos según el tipo de labor a desempeñar, necesitaremos crear algunas opciones adicionales al menú del formulario. Los siguientes puntos explican cada uno de estos modos de trabajo del control.

Figura 53. Formulario mdiInicio, incluyendo un control CommonDialog.

Color

El cuadro de dialogo de color, permite seleccionar entre un conjunto de colores disponibles o configurar un color personalizado por el usuario. Al aceptar el diálogo, la propiedad Color del control, contendrá el valor del color seleccionado por el usuario.

Las constantes para la propiedad Flags, referentes al color se muestran en la siguiente tabla 10.

Constante	**Valor**	**Descripción**
cdCClFullOpen	&H2	Se presenta todo el cuadro de diálogo, incluyendo la sección Definir colores personalizados.
cdlCCHelpButton	&H8	Hace que el cuadro de diálogo presente un botón Ayuda.
cdlCCPreventFullOpen	&H4	Desactiva el botón de comando Definir colores personalizados y evita que el usuario defina colores personalizados.
cdlCCRGBInit	&H1	Establece el valor de color inicial del cuadro de diálogo.

Tabla 10. Constantes para la propiedad Flags usados en el CommonDialog de selección de color.

La opción de menú Configurar+Color ventana principal, nos permite cambiar el color de fondo de la ventana MDI de la aplicación, mostrando el control dlgOperaciones con el diálogo de selección de color. El código fuente 8 muestra el código para esta opción. Observe el lector cómo es posible incluir más de una constante en la propiedad Flags empleando el operador Or.

```
Private Sub mnuCoColor_Click()
' configurar diálogo,
' se puede asignar más de una constante usando
' el operador Or
Me.dlgOperaciones.Flags = cdlCCPreventFullOpen Or cdlCCHelpButton
' mostrar diálogo para selección de color
Me.dlgOperaciones.ShowColor
' aplicar color seleccionado al color de
' fondo del formulario mdi
Me.BackColor = Me.dlgOperaciones.Color
End Sub
```

Código Fuente 8

Impresión

Para las labores de impresión existen dos cuadros de diálogo. Por defecto se muestra el diálogo Imprimir, pero existe la posibilidad de mostrar el cuadro Configurar impresora, para seleccionar y configurar una de las impresoras conectadas al sistema.

Las constantes para la propiedad Flags, referentes a la impresión se muestran en la tabla 11.

Constante	**Valor**	**Descripción**
CdlPDAllPages	&H0	Devuelve o establece el estado del botón de opción Todo.
CdlPDCollate	&H10	Devuelve o establece el estado de la casilla de verificación Intercalar.
cdlPDDisablePrintToFile	&H80000	Desactiva la casilla de verificación Imprimir en un archivo.
cdlPDHelpButton	&H800	Hace que el cuadro de diálogo presente el botón Ayuda.
cdlPDHidePrintToFile	&H100000	Oculta la casilla de verificación Imprimir en un archivo.
cdlPDNoPageNums	&H8	Desactiva el botón de opción Páginas y el control de edición asociado.
cdlPDNoSelection	&H4	Desactiva el botón de opción Selección.
cdlPDNoWarning	&H80	Evita la presentación de un mensaje de advertencia cuando no hay ninguna impresora predeterminada.
CdlPDPageNums	&H2	Devuelve o establece el estado del botón de opción Páginas.

CdlPDPrintSetup	&H40	Hace que el sistema presente el cuadro de diálogo Configurar impresora en vez del cuadro de diálogo Imprimir.
cdlPDPrintToFile	&H20	Devuelve o establece el estado de la casilla de Verificación Imprimir en un archivo.
CdlPDReturnDC	&H100	Devuelve un contexto de dispositivo para la impresora seleccionada en el cuadro de diálogo. El contexto de dispositivo se devuelve en la propiedad hDC del cuadro de diálogo.
cdlPDReturnDefault	&H400	Devuelve el nombre de la impresora predeterminada.
CdlPDReturnIC	&H200	Devuelve un contexto de información para la impresora seleccionada en el cuadro de diálogo. El contexto de Información proporciona una manera rápida de obtener información acerca del dispositivo sin crear un contexto de dispositivo. El contexto de información se devuelve en la propiedad hDC del cuadro de diálogo.
CdlPDSelection	&H1	Devuelve o establece el estado del botón de opción Selección. Si no se especifican cdlPDPageNums ni cdlPDSelection, el botón de opción Todo está activado.
cdlPDUseDevModeCopies	&H40000	Si un controlador de impresora no acepta varias copias, el establecimiento de este indicador desactiva el control Número de copias del cuadro de diálogo Imprimir. Si un controlador no acepta varias copias, el establecimiento de este indicador señala que el cuadro de diálogo almacena el número de copias solicitado en la propiedad Copies.

Tabla 11. Constantes para la propiedad Flags usados en el CommonDialog para los cuadros de diálogo Imprimir / Configuración de impresión.

Mediante la opción de menú Archivo+Imprimir, mostraremos el cuadro de diálogo del mismo nombre, utilizado por las aplicaciones para configurar la impresión de documentos. Lo vemos en las líneas de código que aparecen en el código fuente 9.

```
Private Sub mnuArImprimir_Click()
' deshabilitar el CheckBox de impresión
' a un fichero
Me.dlgOperaciones.Flags = cdlPDDisablePrintToFile
' mostrar diálogo para imprimir documentos
Me.dlgOperaciones.ShowPrinter
End Sub
```

Código fuente 9

En lo que respecta al diálogo de configuración de impresoras, la opción de menú del formulario mdiInicio Configurar+Impresora, visualiza dicho cuadro como vemos en el código fuente 10.

```
Private Sub mnuCoImpresora_Click()
' el cuadro de diálogo será el de selección
' y configuración de impresora
Me.dlgOperaciones.Flags = cdlPDPrintSetup
' mostrar diálogo para configuración de impresora
Me.dlgOperaciones.ShowPrinter
End Sub
```

Código fuente 10

Tipo de letra

Este cuadro nos permite seleccionar un tipo de fuente o letra para poder asignarlo a cualquier objeto de la aplicación que admita las propiedades de fuente. La tabla 12 muestra los valores para la propiedad Flags que pueden usarse para este diálogo.

Constante	**Valor**	**Descripción**
cdlCFANSIOnly	&H400	Especifica que el cuadro de diálogo sólo permite la selección de una de las fuentes que utilicen el juego de caracteres de Windows. Si este indicador está activado, el usuario no podrá seleccionar una fuente que sólo contenga símbolos.
cdlCFApply	&H200	Activa el botón Aplicar en el cuadro de diálogo.
cdlCFBoth	&H3	Hace que el cuadro de diálogo enumere las fuentes de impresora y de pantalla disponibles. La propiedad hDC identifica el contexto de dispositivo asociado con la impresora.
CdlCFEffects	&H100	Especifica que el cuadro de diálogo permite los efectos de tachado, subrayado y color.
cdlCFFixedPitchOnly	&H4000	Especifica que el cuadro de diálogo selecciona sólo fuentes de tamaño de punto fijo.
CdlCFForceFontExist	&H10000	Especifica que se presentará un mensaje de error si el usuario intenta seleccionar una fuente o un estilo que no exista.
CdlCFHelpButton	&H4	Hace que el cuadro de diálogo presente un botón Ayuda.
cdlCFLimitSize	&H2000	Especifica que el cuadro de diálogo selecciona sólo tamaños de fuente dentro del intervalo especificado por las propiedades Min y Max.
cdlCFNoFaceSel	&H80000	No hay ningún nombre de fuente seleccionado.

CdlCFNoSimulations	&H1000	Especifica que el cuadro de diálogo no permite simulaciones de fuente con Interfaz de dispositivo gráfico (GDI).
CdlCFNoSizeSel	&H200000	No hay ningún tamaño de fuente seleccionado.
CdlCFNoStyleSel	&H100000	No se seleccionó ningún estilo.
CdlCFNoVectorFonts	&H800	Especifica que el cuadro de diálogo no permite selecciones de fuentes vectoriales.
cdlCFPrinterFonts	&H2	Hace que el cuadro de diálogo enumere únicamente las fuentes aceptadas por la impresora, según lo especificado por la propiedad hDC.
cdlCFScalableOnly	&H20000	Especifica que el cuadro de diálogo sólo permite la selección de fuentes escalables.
cdlCFScreenFonts	&H1	Hace que el cuadro de diálogo enumere únicamente las fuentes de pantalla compatibles con el sistema.
cdlCFTTOnly	&H40000	Especifica que el cuadro de diálogo sólo permite la selección de fuentes TrueType.
cdlCFWYSIWYG	&H8000	Especifica que el cuadro de diálogo sólo permite la selección de fuentes que estén disponibles tanto en la impresora como en la pantalla. Si este indicador está activado, los indicadores cdlCFBoth y cdlCFScalableOnly también deben definirse.

Tabla 12. Constantes de la propiedad Flags usados en el CommonDialog Fuente.

Un detalle de suma importancia, es que debemos asignar a Flags antes de visualizar el diálogo, alguno de estos valores: cdlCFScreenFonts, cdlCFPrinterFonts o cdlCFBoth. De no hacerlo así, se producirá un error.

Para ilustrar la selección de un tipo de letra, la operación que vamos a efectuar, es cambiar el tipo de letra en los CommandButton que contiene el formulario frmCompras, como es natural, deberemos tener abierto dicho formulario o de lo contrario se producirá un error, aunque en el caso de que ocurra, dicho error será tratado en una etiqueta de manejo de errores. La opción de menú en frmInicio que ejecuta esta operación es Configurar+Tipo de letra en compras, y el código que contiene es el que vemos en el código fuente 11.

```
Private Sub mnuCoTipoLetra_Click()
Dim lfrmCompras As frmCompras
' establecer rutina de manejo de errores
On Error GoTo ChkError
' tomar el formulario activo dentro de la
' ventana mdi
Set lfrmCompras = Me.ActiveForm
' configurar diálogo
Me.dlgOperaciones.Flags = cdlCFScreenFonts
```

```
' mostrar cuadro diálogo selección de tipo de letra
Me.dlgOperaciones.ShowFont
' si no se selecciona una fuente, terminar el procedimiento
If Len(Trim(Me.dlgOperaciones.FontName)) = 0 Then
   Exit Sub
End If
' modificar el tipo de letra del formulario de compras
' por el seleccionado en el diálogo
lfrmCompras.cmdAceptar.FontName = Me.dlgOperaciones.FontName
lfrmCompras.cmdCancelar.FontName = Me.dlgOperaciones.FontName
lfrmCompras.cmdAgregar.FontName = Me.dlgOperaciones.FontName
' -----------------
' etiqueta para salir del procedimiento
Salir:
Exit Sub
' -----------------
' etiqueta para manejo de errores
ChkError:
MsgBox "La ventana de compras no está abierta", , "Error"
Resume Salir
End Sub
```

Código fuente 11

Apertura de ficheros

Este cuadro nos permite seleccionar uno o varios ficheros para abrir y manipular su contenido. El control no es el encargado de la apertura del fichero seleccionado, su labor es devolvernos el nombre y ruta de dicho fichero/s, siendo la aplicación la encargada de la apertura y manipulación

Debido a que tanto el diálogo de apertura como el de grabación son muy similares, las constantes empleadas en la propiedad Flags son las mismas para ambos tipos de cuadro.

Constante	**Valor**	**Descripción**
cdlOFNAllowMultiselect	&H200	Especifica que el cuadro de lista Nombre de archivo permita varias selecciones. El usuario puede seleccionar varios archivos en tiempo de ejecución presionando la tecla MAYÚS y utilizando las teclas FLECHA ARRIBA y FLECHA ABAJO para seleccionar los archivos deseados. Al terminar, la propiedad FileName devuelve una cadena que contiene los nombres de todos los archivos seleccionados. Los nombres de la cadena están delimitados por espacios en blanco.
cdlOFNCreatePrompt	&H2000	Especifica que el cuadro de diálogo pida al usuario la creación de un archivo que no existe actualmente. Este indicador establece automáticamente los Indicadores cdlOFNPathMustExist y CdlOFNFileMustExist.

cdlOFNExplorer	&H80000	Usa la plantilla del cuadro de diálogo Abrir archivo de tipo Explorador. Funciona en Windows 95 y en Windows NT 4.0.
cdlOFNExtensionDifferent	&H400	Indica que la extensión del nombre de archivo devuelto es distinta de la extensión especificada por la propiedad DefaultExt. Este indicador no está definido si la propiedad DefaultExt es Null, si las extensiones coinciden o si el archivo no tiene extensión. El valor de este indicador se puede comprobar después de cerrar el cuadro de diálogo.
cdlOFNFileMustExist	&H1000	Especifica que el usuario sólo puede introducir nombres de archivos existentes en el cuadro de texto Nombre de archivo. Si este indicador está activado y el usuario escribe un nombre de archivo no válido, se mostrará una advertencia. Este indicador establece automáticamente el indicador cdlOFNPathMustExist.
cdlOFNHelpButton	&H10	Hace que el cuadro de diálogo presente el botón Ayuda.
cdlOFNHideReadOnly	&H4	Oculta la casilla de verificación Sólo lectura.
cdlOFNLongNames	&H200000	Usa nombres de archivo largos.
cdlOFNNoChangeDir	&H8	Hace que el cuadro de diálogo restablezca como directorio actual el que lo era en el momento de abrirse el cuadro de diálogo.
cdlOFNNoDereferenceLinks	&H100000	No resuelve la referencia en vínculos del sistema (también conocidos como accesos directos). De forma predeterminada, la elección de un vínculo hace que el sistema resuelva la referencia que contiene.
cdlOFNNoLongNames	&H40000	Nombres de archivos cortos.
cdlOFNNoReadOnlyReturn	&H8000	Especifica que el archivo devuelto no tendrá Establecido el atributo de Sólo lectura y no estará en un directorio protegido contra escritura.
cdlOFNNoValidate	&H100	Especifica que el cuadro de diálogo común Permite caracteres no válidos en el nombre de archivo devuelto.
cdlOFNOverwritePrompt	&H2	Hace que el cuadro de diálogo Guardar como Genere un cuadro de mensajes si el archivo Seleccionado ya existe. El usuario tiene que Confirmar si desea sobrescribir el archivo.

cdlOFNPathMustExist	&H800	Especifica que el usuario sólo puede escribir Rutas de acceso válidas. Si este indicador está activado y el usuario escribe una ruta no válida, se mostrará un mensaje de advertencia.
cdlOFNReadOnly	&H1	Hace que la casilla de verificación Sólo lectura esté activada inicialmente cuando se crea el cuadro de diálogo. Este indicador también señala el estado de la casilla de verificación Sólo lectura cuando se cierra el cuadro de diálogo.
cdlOFNShareAware	&H4000	Especifica que se pasarán por alto los errores por infracción al compartir.

Tabla 13.Constantes para la propiedad Flags usados en el CommonDialog para los diálogos Abrir / Guardar como.

En lo que se refiere a las propiedades de este control para el manejo de ficheros, podemos destacar las siguientes:

- DefaultExt. Contiene una cadena con la extensión predeterminada del fichero a seleccionar.
- DialogTitle. Contiene una cadena con el título para el cuadro de diálogo. Esta propiedad sólo funciona con los cuadros de diálogo de manejo de ficheros.
- FileName. Cadena con el nombre y la ruta del fichero a manipular. Esta propiedad puede establecerse antes de mostrar el cuadro, para disponer de un fichero inicial a seleccionar, o emplearse una vez aceptado el diálogo, para recuperar el fichero seleccionado por el usuario.
- FileTitle. Cadena con el nombre, sin incluir la ruta, del fichero seleccionado.
- InitDir. Directorio inicial del cual se mostrará el contenido al visualizar el diálogo.
- Filter. Cadena con uno o más filtros para los ficheros. Un filtro especifica el tipo de ficheros que serán mostrados por el cuadro de diálogo. El formato de esta cadena es: descripción del tipo de fichero con la extensión entre paréntesis - separador - extensión usada por el cuadro de diálogo. Por ejemplo: Texto (*.txt)|*.txt. Esta cadena hace que el diálogo sólo muestre los fichero con la extensión .TXT.

  Si fuera necesario incluir más de un tipo de ficheros, se deberá utilizar el separador para delimitarlos: Texto (*.txt)|*.txt | Código (*.bas)|*.bas.
- FilterIndex. Valor numérico que indica cuál de los filtros especificados en la propiedad Filter, aparecerá como predeterminado al mostrar el cuadro. Si no se indica un valor, se utiliza 1.
- MaxFileSize. Número que informa sobre el tamaño máximo del nombre de archivo abierto.

Para las operaciones de apertura en el proyecto Encuesta, seleccionaremos la opción Archivo+Abrir, del menú del formulario frmInicio, que contiene las líneas de código que aparecen en el código fuente 12.

```
Private Sub mnuArAbrir_Click()
' configurar el control para mostrar el
```

```
' cuadro Abrir
Me.dlgOperaciones.DialogTitle = "Apertura de ficheros para el proyecto"
Me.dlgOperaciones.InitDir = "C:\"
Me.dlgOperaciones.Filter = "Código (*.bas)|*.bas|" & _
   "Formularios (*.frm;*.frx)|*.frm;*.frx"
Me.dlgOperaciones.FilterIndex = 2
' mostrar el diálogo
Me.dlgOperaciones.ShowOpen
' resultados del cuadro
MsgBox "La ruta del proyecto es: " & Me.dlgOperaciones.filename, , _
   "Apertura"
MsgBox "El nombre del fichero seleccionado es: " & _
   Me.dlgOperaciones.FileTitle, , "Apertura"
End Sub
```

Código fuente 12

Grabación de ficheros

Para guardar ficheros utilizando el control CommonDialog, las propiedades a utilizar son las mismas que para abrir, existiendo alguna pequeña variación en las constantes para la propiedad Flags. En el programa de ejemplo, opción de menú Archivo+Guardar como, del formulario frmInicio, podemos comprobar como se utiliza este control. A continuación tenemos el código de esta opción (código fuente13).

```
Private Sub mnuArGuardarcomo_Click()
' configurar el control para mostrar el
' cuadro Guardar como
Me.dlgOperaciones.DialogTitle = "Grabar ficheros de texto"
Me.dlgOperaciones.InitDir = "C:\"
Me.dlgOperaciones.Filter = "Texto (*.txt)|*.txt"
Me.dlgOperaciones.Flags = cdlOFNHideReadOnly Or _
   cdlOFNOverwritePrompt
' mostrar el diálogo
Me.dlgOperaciones.ShowSave
' resultados del cuadro
MsgBox "La ruta es: " & Me.dlgOperaciones.filename, , _
   "Grabación"
MsgBox "El nombre del fichero a grabar es: " & _
   Me.dlgOperaciones.FileTitle, , "Grabación"
End Sub
```

Código Fuente 13

Ayuda

En lo que respecta a la ayuda, el control CommonDialog no muestra ningún cuadro de diálogo específico, sino que visualiza directamente un fichero de ayuda conforme a los valores asignados a las propiedades que sobre la ayuda dispone este control, y que son las siguientes:

- HelpFile. Sirve para asignar una cadena con la ruta y nombre del fichero de ayuda a mostrar.
- HelpCommand. Constante que indica el tipo de ayuda a visualizar. La tabla 14, contiene los valores disponibles para esta propiedad.

Constante	**Valor**	**Descripción**
cdlHelpCommandHelp	&H102&	Ejecuta una macro de Ayuda.
cdlHelpContents	&H3&	Presenta el tema de contenido de Ayuda que se haya definido en la opción Contents de la sección [OPTION] del archivo .hpj.
cdlHelpContext	&H1&	Presenta Ayuda acerca de un contexto concreto. Cuando utilice este valor, también tiene que especificar un contexto mediante la propiedad HelpContext.
cdlHelpContextPopup	&H8&	Presenta, en una ventana emergente, un determinado tema de Ayuda identificado por un número de contexto definido en la sección [MAP] del archivo .hpj.
cdlHelpForceFile	&H9&	Asegura que WinHelp presentará el archivo de Ayuda correcto. Si actualmente se está presentando el archivo de Ayuda correcto, no se realizará ninguna acción. Si se está presentando un archivo de Ayuda incorrecto, WinHelp abrirá el archivo correcto.
cdlHelpHelpOnHelp	&H4&	Presenta Ayuda acerca del uso de la aplicación de Ayuda propiamente dicha.
cdlHelpIndex	&H3&	Presenta el índice del archivo de Ayuda especificado. La aplicación sólo debe utilizar este valor con archivos de Ayuda de índice único.
cdlHelpKey	&H101&	Presenta Ayuda acerca de una palabra clave determinada. Cuando utilice este valor, también tiene que especificar una palabra clave mediante la propiedad HelpKey.
cdlHelpPartialKey	&H105&	Presenta el tema encontrado en la lista de palabras clave que coincida con la palabra clave pasada en el parámetro dwData si hay coincidencia exacta. Si hay varias coincidencias, se presenta el cuadro de diálogo Buscar, con los temas encontrados en el cuadro de lista Ir a. Si no hay coincidencia, se presenta el cuadro de diálogo Buscar. Para presentar el cuadro de diálogo Buscar sin pasar una palabra clave, utilice un puntero de tipo Long a una cadena vacía.
cdlHelpQuit	&H2&	Notifica a la aplicación de Ayuda que el archivo de Ayuda especificado ya no está en uso.
cdlHelpSetContents	&H5&	Determina el tema de contenido que se presenta cuando un usuario presiona la tecla F1.
cdlHelpSetIndex	&H5&	Establece el contexto especificado por la propiedad HelpContext como índice actual del archivo de Ayuda especificado por la propiedad HelpFile. Dicho índice permanece como actual hasta que el usuario consulta otro archivo de Ayuda diferente. Utilice este valor sólo para los archivos de Ayuda que tengan varios índices.

Tabla 14. Constantes disponibles para la propiedad HelpCommand.

- HelpKey. Cadena con el tema a mostrar del fichero de ayuda.Para poder utilizar esta propiedad, se debe establecer en la propiedad HelpCommand la constante cdlHelpKey.

- HelpContext. Contiene el identificador de contexto del tema de ayuda. Para poder utilizar esta propiedad, se debe establecer en la propiedad HelpCommand la constante cdlHelpContext.

Para ilustrar el uso de este control con ficheros de ayuda, disponemos de la opción Ayuda+Contenido del formulario frmInicio, que contiene el código fuente 14.

```
Private Sub mnuAyContenido_Click()
' asignar el fichero de ayuda a utilizar
Me.dlgOperaciones.HelpFile = "c:\windows\help\mspaint.hlp"
' indicar que vamos a buscar dentro del fichero
' de ayuda un tema concreto
Me.dlgOperaciones.HelpCommand = cdlHelpKey
' establecer el tema a buscar
Me.dlgOperaciones.HelpKey = ""
' mostrar la ayuda
Me.dlgOperaciones.ShowHelp
End Sub
```

Código Fuente 14

Agregar funcionalidad al formulario principal

Llegados a este punto, la aplicación estaría en condiciones de ser utilizada, sin embargo, existen ciertos elementos, esencialmente visuales, que podemos incorporar a nuestro formulario MDI, con los cuales conseguiremos facilitar el trabajo del usuario, además de proporcionar un aspecto más profesional al programa. Estos elementos son: una barra de herramientas, que facilite en forma de botones, el acceso a los elementos del menú, y una barra de estado, que proporcione información diversa al usuario sobre la hora, fecha, estado del teclado, etc.; datos todos ellos, no esenciales, pero que proporcionan un valor añadido a la aplicación, y harán que esta sea mejor valorada con respecto a otras que no lo tengan.

Estos elementos, como la mayoría de los que hemos tratado hasta ahora, son controles que seleccionamos del cuadro de herramientas e insertamos en el formulario; configurando sus propiedades y escribiendo código para los mismos, conseguiremos el resultado necesario.

Para poder disponer de estos controles, debemos abrir la ventana Componentes y seleccionar el elemento Microsoft Windows Common Controls 6.0, de forma que serán agregados al Cuadro de herramientas, un grupo de controles entre los cuales se encuentran los que vamos a utilizar.

ImageList

Este control se utiliza para almacenar un conjunto de imágenes que serán empleadas por otro control para visualizarlas. No es un control que se use directamente en el formulario, sino que sirve como apoyo para otro control, en este caso para la barra de herramientas, ya que contendrá las imágenes de los botones que serán mostradas en dicha barra.

Figura 54. Control ImageList, en el Cuadro de herramientas.

Figura 55. Formulario frmInicio mostrando un control ImageList.

Una vez incluido un ImageList en el formulario frmInicio (figura 55), al que daremos el nombre imlImagenes, accederemos a su ventana Páginas de propiedades, mediante el menú contextual del control o pulsando en el botón que existe en la propiedad Personalizado de su ventana de propiedades, para configurar las propiedades que no están accesibles en la ventana de propiedades habitual.

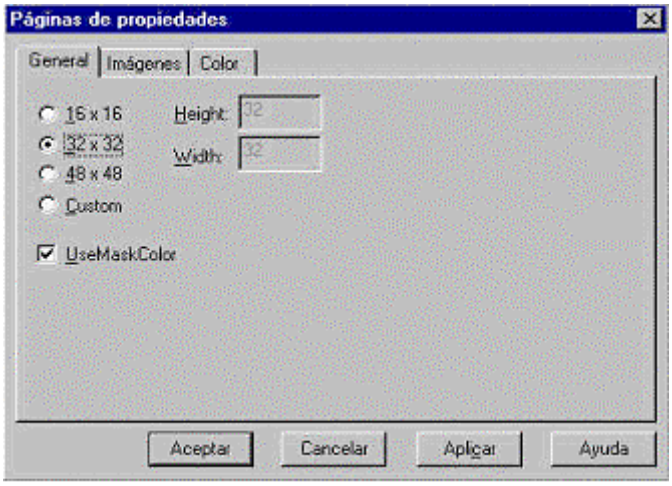


Figura 56. Pestaña General de la ventana Páginas de propiedades para un control ImageList.

En primer lugar, estableceremos en la página de propiedades General (figura 56), el tamaño de la imagen que será mostrada por el control que haga uso del ImageList.

Seguidamente pasaremos a la página Imágenes, en la que seleccionaremos los ficheros de tipo icono o bitmap, que van a formar parte de la lista. Usaremos el botón Insertar imagen para incluir nuevos ficheros en el control que serán visualizados en la lista Imágenes. El botón Quitar imagen, eliminará la imagen actualmente seleccionada en la lista. Después de incluir varios ficheros, la lista de imágenes quedará como muestra la figura 57.

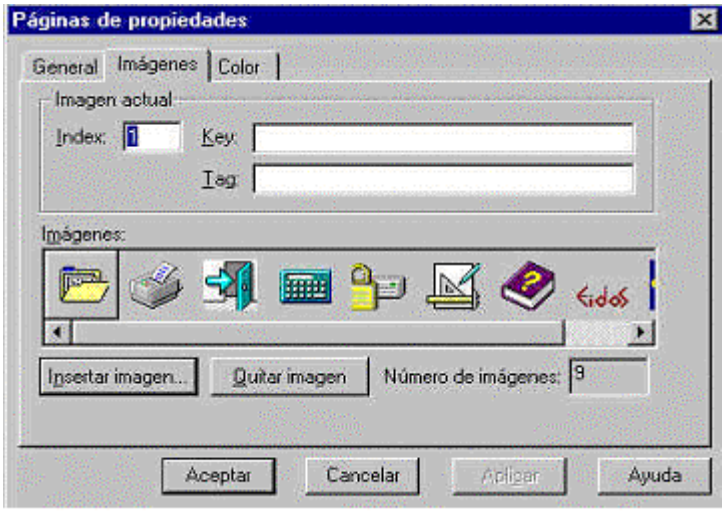


Figura 57. Página de propiedades Imágenes, de un control ImageList.

Toolbar

Este control contiene una colección de objetos Button, que por lo general, corresponden a las opciones de menú del formulario que incluye la Toolbar

Figura 58. Control Toolbar, en el Cuadro de herramientas.

Después de insertar un control de este tipo en el formulario frmInicio, al que llamaremos tbrBarra, daremos valores a sus propiedades. En la ventana de propiedades, entre otras, tenemos las siguientes:

- Align. Constante que indica la posición de la barra de herramientas dentro del formulario en la que está incluida. Los valores disponibles se muestran en la tabla 15.

Constante	Valor	Descripción
VbAlignNone	0	(Predeterminado en formularios no MDI) Ninguno: el tamaño y la posición pueden establecerse en tiempo de diseño o en el código. Este valor se pasa por alto si el objeto está en un formulario MDI.
VbAlignTop	1	(Predeterminado en formularios MDI) Arriba: el objeto está en la parte superior del formulario y su ancho es igual al valor de la propiedad ScaleWidth del formulario.
VbAlignBottom	2	Abajo: el objeto está en la parte inferior del formulario y su ancho es igual al valor de la propiedad ScaleWidth del formulario.

VbAlignLeft	3	Izquierda: el objeto está en la parte izquierda del formulario y su ancho es igual al valor de la propiedad ScaleWidth del formulario.
VbAlignRight	4	Derecha: el objeto está en la parte derecha del formulario y su ancho es igual al valor de la propiedad ScaleWidth del formulario.

Tabla 15. Constantes disponibles para la propiedad Align, de un control Toolbar.

El valor para esta propiedad en el control tbrBarra será vbAlignTop.

- ToolTipText. Cadena informativa que se muestra al usuario cuando pasa el puntero del ratón sobre el área del control. En nuestro control no contendrá ningún valor, de esta manera evitaremos que aparezca junto con los ToolTipText de los botones de la barra, lo que podría confundir al usuario.

- ShowTips. Valor lógico que si es True, muestra el contenido de la propiedad ToolTipText de los botones visualizados en la Toolbar.

En la ventana Páginas de propiedades de este control, disponemos de la página General, que vemos en la figura 59.

Figura 59. Página de propiedades General, para un control Toolbar.

En la propiedad ImageList, asignaremos el valor imlImagenes, que corresponde al control ImageList que contiene las imágenes que serán usadas por los botones de la Toolbar.

En la página de propiedades Botones, usaremos los botones Insertar botón y Quitar botón para agregar o eliminar botones al control. Disponemos adicionalmente de una serie de propiedades para configurar cada botón de la barra, entre las cuales podemos destacar las siguientes.

- Index. Número de botón de la barra cuyas propiedades aparecen actualmente en la página de propiedades.
- Caption. Cadena que será mostrada dentro del botón, junto a la imagen .
- Key. Cadena para identificar de forma unívoca el botón del resto de botones de la barra.
- Value. Estado del botón, según las siguientes constantes:
  - tbrUnpressed. El botón aparece sin pulsar, este es el valor por defecto.
  - tbrPressed. El botón se muestra pulsado.
- Style. Constante que indica la apariencia y comportamiento del botón. La tabla 16 muestra los valores disponibles.

Constante	Valor	Descripción
tbrDefault	0	(Predeterminado) Botón de apariencia normal.
tbrCheck	1	Botón de estilo CheckBox.
tbrButtonGroup	2	Grupo de botones entre los cuales siempre se encontrará uno pulsado. La pulsación de un nuevo botón, hará que dicho botón quede presionado, y el que hubiera pulsado vuelva a su estado normal.
tbrSeparator	3	Este tipo de botón no se muestra realmente como tal, sino que se trata de un espacio de 8 pixels que sirve para separar dos botones de la Toolbar.
tbrPlaceholder	4	Este tipo de botón, al igual que el separador, muestra un espacio vacío, pero a diferencia del separador, dispone de la propiedad Width, que puede ser establecida por el programador, para incluir en dicho espacio, un control de un tipo diferente al botón; un ComboBox por ejemplo.
tbrDropDown	5	MenuButton desplegable. Abre una lista de opciones.

Tabla 16. Constantes disponibles para los botones de un control Toolbar.

- ToolTipText. Cadena que informa al usuario de la función que desempeña el botón, mostrada al pasar el usuario el ratón sobre el botón.
- Image. Número que se corresponde a una de las imágenes del control ImageList conectado al control Toolbar, y que será mostrada en el botón.

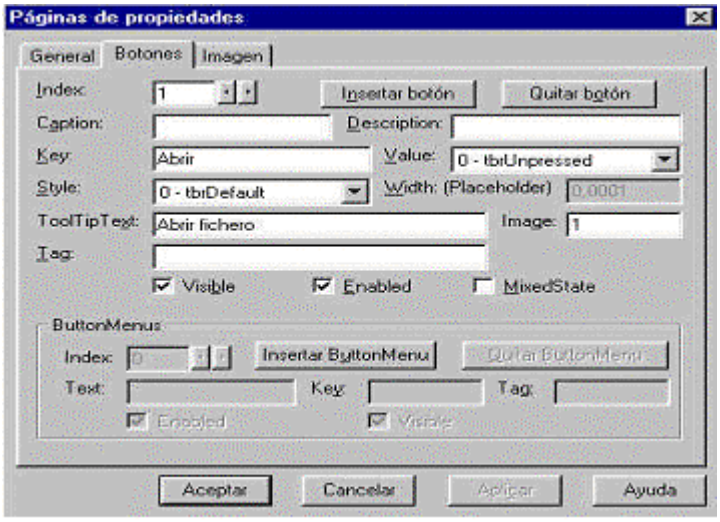


Figura 60. Página de propiedades Botones, para un control Toolbar.

Después de establecer todas las propiedades para la Toolbar, escribiremos el código para su procedimiento de evento ButtonClick(), que es ejecutado cada vez que se pulsa uno de los botones en este control. Aquí debemos codificar las acciones a realizar en función del botón pulsado, como se muestra en el código fuente 15.

```
Private Sub tbrBarra_ButtonClick(ByVal Button As MSComctlLib.Button)
' se ha pulsado un botón de la barra
' de herramientas, comprobar que botón
' es y ejecutar la opción de menú
' correspondiente
Select Case Button.Key
Case "Abrir"
   mnuArAbrir_Click
   Case "Salir"
   mnuArSalir_Click
   Case "Encuestas"
   mnuEnInfo_Click
Case "Configurar"
   mnuCoImpresora_Click
Case "Ayuda"
   mnuAyContenido_Click
 End Select
End Sub
```

Código fuente 15

En la anterior página de propiedades de este control (Botones), disponemos también de un apartado llamado ButtonMenus, localizado en un Frame del mismo nombre. Si tomamos cualquiera de los botones de la barra y establecemos su propiedad Style a tbrDropdown, podremos desde este apartado añadir opciones a dicho botón, de manera que se convertirá en una combinación de botón y lista desplegable; cuando se pulse el botón, se ejecutará el código ya visto en el anterior fuente, pero cuando seleccionemos un elemento de la lista, deberemos codificar el evento ButtonMenuClick() del control ToolBar, si deseamos responder con alguna acción al usuario.

En este caso, hemos tomado el botón Ayuda de la barra, configurándolo de esta manera, y le hemos incorporado tres opciones, como vemos en la figura 61.

Figura 61. Botón de la Toolbar con estilo DropDown.

Las líneas de código para estas opciones se muestran en el código fuente 16. Para evitar comprobar la opción pulsada, en la propiedad Key del ButtonMenu pasado como parámetro, se ha establecido en tiempo de diseño la opción de la ayuda a consultar.

```
' asignar el fichero de ayuda a utilizar
Me.dlgOperaciones.HelpFile = "c:\windows\help\mspaint.hlp"
' indicar que vamos a buscar dentro del fichero
' de ayuda un tema concreto
Me.dlgOperaciones.HelpCommand = cdlHelpKey
' establecer el tema a buscar
Me.dlgOperaciones.HelpKey = ButtonMenu.Key
' mostrar la ayuda
Me.dlgOperaciones.ShowHelp
```

Código fuente 16

StatusBar

Muestra una barra de estado, normalmente en la parte inferior del formulario, con información de utilidad diversa para el usuario, como pueda ser la fecha, hora, estado del teclado, etc

Figura 62. Control StatusBar, en el Cuadro de herramientas.

Este control está formado por un conjunto de objetos Panel, cada uno de los cuales se encarga de mostrar una clase diferente de información en la barra de estado: fecha, hora, etc. Insertaremos un control de este tipo en frmInicio al que llamaremos sbrEstado, que pasamos a configurar seguidamente.

En lo que a propiedades se refiere, la ventana Páginas de propiedades, en su página General, destaca la siguiente propiedad:

- Style. Constante que sirve para modificar el estilo de la barra, con dos valores:
    - sbrNormal. La barra muestra todos los paneles que se han definido.

  - sbrSimple. La barra muestra un único panel.

- SimpleText. Texto que se visualizará cuando el valor de la propiedad Style sea sbrSimple.

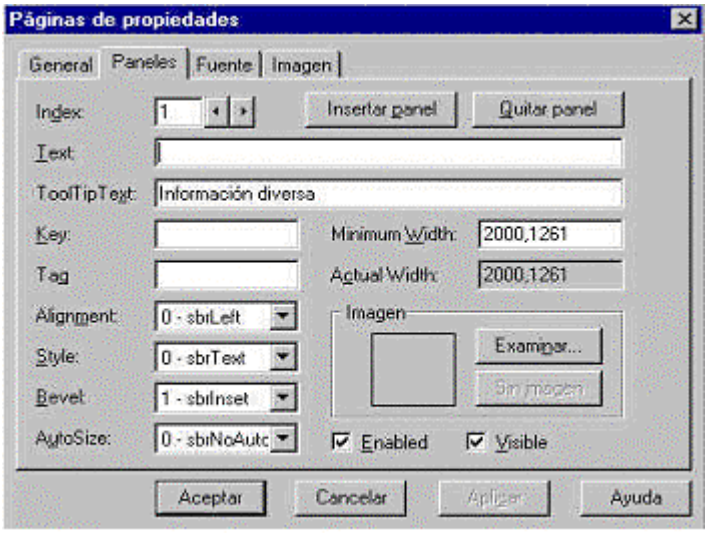


Figura 63. Página Paneles del control StatusBar.

En la página paneles (figura 63),agregaremos o eliminaremos paneles mediante los botones Insertar panel y Quitar panel respectivamente. En lo que se refiere a las propiedades de que dispone esta página, podemos destacar:

- Index. Número de panel al que corresponden las propiedades actualmente visualizadas por la página.

- Text. Si el estilo del panel es de texto, se mostrará la cadena contenida en esta propiedad.

- Key. Cadena para identificar de forma unívoca el panel del resto de paneles de la barra.

- Minimum Width. Si el panel es demasiado grande para la información que muestra, como puede ser el caso de los estados del teclado; con esta propiedad podemos ajustar el ancho del panel.

- Alignment. Constante que identifica el modo en que el texto del panel será alineado. La tabla 17 muestra los valores disponibles.

Constante	**Valor**	**Descripción**
SbrLeft	0	(Predeterminado). El texto aparece alineado a la izquierda y a la derecha del mapa de bits.
SbrCenter	1	El texto aparece centrado y a la derecha de mapa de bits.
SbrRight	2	El texto aparece alineado a la derecha y a la izquierda del mapa de bits.

Tabla 17. Constantes disponibles para la propiedad Alignment de un objeto Panel.

- Style. Tipo de panel identificado por una de las constantes mostradas en la tabla 18.

Constante	Valor	Descripción
sbrText	0	(Predeterminado). Texto y mapa de bits. Establece el texto con la propiedad Text.
sbrCaps	1	Tecla BLOQ MAYÚS. Muestra las letras MAYÚS en negrita cuando BLOQ MAYÚS está activada y atenuada cuando está desactivada.
sbrNum	2	Tecla BLOQ NUM. Muestra las letras NÚM en negrita cuando BLOQ NUM está activada y atenuadas cuando está desactivada.
SbrIns	3	Tecla INSERT. Muestra las letras INS en negrita cuando INSERT está activada y atenuadas cuando no lo está.
SbrScrl	4	Tecla BLOQ DESPL. Muestra las letras DESPL en negrita cuando BLOQ DESPL está activada y atenuadas cuando no lo está.
SbrTime	5	Hora. Muestra la hora actual con el formato del sistema.
SbrDate	6	Fecha. Muestra la fecha actual con el formato del sistema.
SbrKana	7	Kana. Muestra las letras KANA en negrita cuando BLOQ DESPL está activada y atenuada cuando está desactivada.

Tabla 18. Constantes disponibles para la propiedad Style, de un objeto Panel.

- Bevel. Configura el estilo de biselado para un objeto Panel. Los valores posibles se muestran en la siguiente tabla 19.

Constante	Valor	Descripción
sbrNoBevel	0	Ninguno. El objeto Panel no presenta bisel y el texto aparece directamente sobre la barra de estado.
sbrInset	1	(Predeterminado). Hacia adentro. El objeto Panel aparece hundido en la barra de estado.
sbrRaised	2	Hacia afuera. El objeto Panel aparece levantado sobre la barra de estado.

Tabla 19. Constantes disponibles para la propiedad Bevel, de un objeto Panel.

El lector habrá observado que también es posible asignar una imagen a un panel, sólo ha de seleccionar el fichero que contiene el gráfico, pulsando el botón Examinar del frame Imagen.

Vistas las propiedades de este control, crearemos nuestro propio conjunto de paneles para el control sbrEstado y ejecutaremos la aplicación, para comprobar el resultado de los nuevos controles que acabamos de incorporar. La figura 64 muestra el formulario en ejecución.

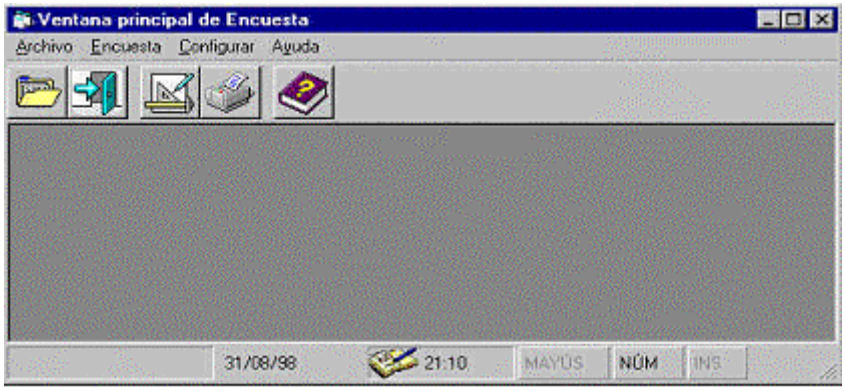


Figura 64. Formulario principal de la aplicación, mostrando los controles Toolbar y StatusBar.

Suponemos que el lector estará de acuerdo, en que el aspecto actual de este formulario MDI, resultará más atractivo al usuario, captará mejor su atención y le proporcionará más información con un simple vistazo.

Crear una ventana Splash o de presentación

Se denomina ventana Splash, a un formulario cuyo área de trabajo está ocupado habitualmente por una imagen con un logotipo, y que no dispone de ningún elemento para que el usuario pueda interaccionar con él: barra de título, menú de control, botones de maximizar, minimizar, cerrar, etc.

Este tipo de formulario, se visualiza durante unos segundos al inicio de la aplicación, proporcionando tiempo a la misma para que prepare su entorno de trabajo: apertura de ficheros, lectura de valores del Registro, etc. De esta forma, el usuario sabrá que el programa está trabajando internamente mientras se muestra este formulario, ya que de lo contrario, la aplicación podría dar la impresión de que se ha bloqueado.

Por las razones antes expuestas, si una vez terminado el desarrollo de una aplicación queremos darle ese toque de profesionalidad, al estilo de las más conocidas aplicaciones comerciales, es obligado incluir un formulario de este tipo.

Pasemos por tanto a nuestra aplicación Encues2. El medio más rápido para incluir un formulario de este tipo es mediante la opción Proyecto+Agregar formulario del menú de VB, que nos mostrará una ventana con varios tipos de formularios predefinidos, entre los cuales seleccionaremos Pantalla inicial, tal y como se muestra en la figura 65.

Pulsando el botón Abrir de este cuadro, se añadirá al proyecto un formulario inicial con un conjunto de elementos por defecto habituales en este tipo de ventana, como podemos ver en la figura 66.

Es posible, naturalmente, crear este tipo de formulario partiendo de cero, pero usando esta ayuda, el formulario es creado con todos los elementos indispensables para el programador, que bien puede utilizarlo tal cuál o realizarle las modificaciones que estime oportunas para adaptarlo a su aplicación.

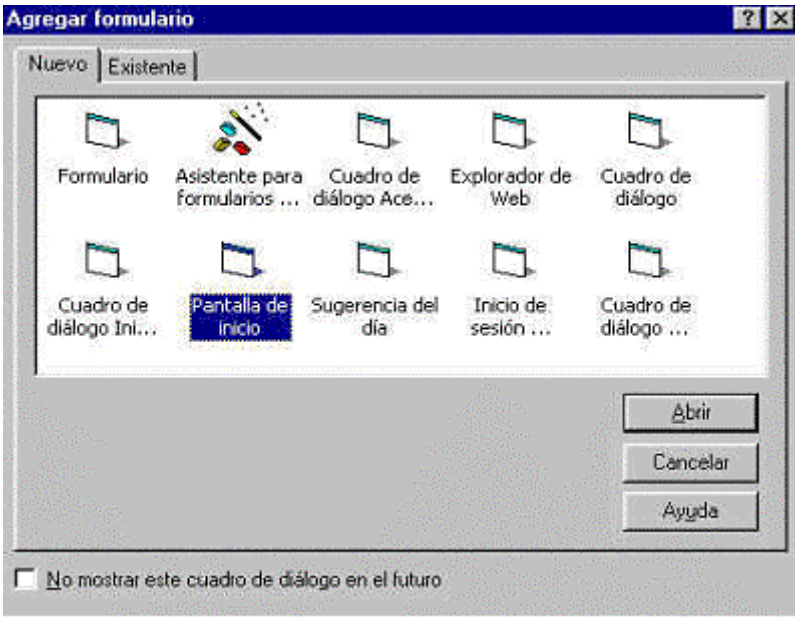


Figura 65. Cuadro de diálogo para agregar un formulario al proyecto.

Figura 66. Formulario Splash creado automáticamente por Visual Basic.

El formulario tiene el nombre frmSplash, y a continuación se repasan las propiedades que deben configurarse para conseguir el efecto de ventana inicial deseado.

- BorderStyle. Contiene el valor 3 que corresponde a un diálogo fijo, que no se puede cambiar de tamaño.
- ControlBox. El valor de esta propiedad es False, de modo que no se mostrará el menú de control del formulario ni el título.
- MDIChild. Esta propiedad contiene False ya que el formulario no actuará como secundario MDI, puesto que deberá ser mostrado antes que el formulario MDI de la aplicación.
- StartUpPosition. Su valor es 2, de manera que será presentado en la parte central de la pantalla.

- ShowInTaskbar. Al ser un formulario que será visualizado tan sólo unos instantes, el valor de esta propiedad será False, para que no se pueda acceder a la ventana a través de la barra de tareas de Windows.

Como información de cara al usuario, se incluyen varios controles Label, en los que se puede situar el nombre del programa, fabricante, versión, sistema operativo bajo el que funciona, etc. También disponemos de un control Image, llamado imgLogo, al que se le asigna un fichero gráfico en su propiedad Picture, y que tiene como finalidad mostrar un logotipo, normalmente el del fabricante. Todos estos elementos se encuentran englobados dentro de un control Frame.

Para regular el tiempo que este formulario permanece en pantalla, utilizaremos un nuevo control llamado Timer, que explicamos a continuación.

Timer

Este control no es otra cosa que un temporizador, que permite ejecutar el código existente en su procedimiento Timer() transcurrido un determinado intervalo de tiempo. Al igual que otros controles ya explicados, una vez insertado en el formulario, se muestra como un icono, ya que no dispone de interfaz visual, es decir, al ejecutar el formulario no será visualizado.

Figura 67. Control Timer, en el Cuadro de herramientas.

Las principales propiedades de este control son las siguientes.

- Enabled. Permite activar o desactivar el temporizador.
    - True. Pone en marcha el temporizador.
    - False. Detiene el temporizador.
- Interval. Valor numérico que determina en milisegundos, las llamadas que se realizarán al evento Timer() del control. El número asignado a esta propiedad debe estar en el rango de 1 a 65.535; si por ejemplo, asignamos 3.000, se ejecutará el evento Timer() cada 3 segundos. Asignando 0 a esta propiedad, el control permanecera inactivo.

Insertaremos pues, un control de este tipo en el formulario al que llamaremos tmrTemporizador, que será activado en cuanto el formulario sea cargado, ya que su propiedad Enable tiene el valor True; el código de su evento Timer() se ejecutará cada 2 segundos, puesto que la propiedad Interval contiene el valor 2000.

A continuación, realizaremos algunas adaptaciones en los controles Label y modificaremos el fichero gráfico de imgLogo, con lo que el aspecto de frmSplash será el mostrado en la figura 68.

Debido a que este formulario será a partir de ahora, el primero que deba visualizarse al ejecutar la aplicación, tendremos que hacer ciertos retoques en el código. El más inmediato será en el procedimiento Main(), cuyas líneas de código modificadas se muestran en el código fuente 17.

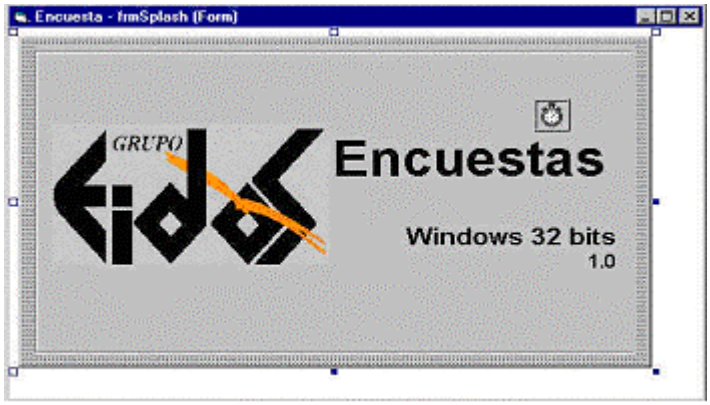


Figura 68. Formulario frmSplash adaptado a la aplicación Encues2.

```
Public Sub Main()
Dim lfrmSplash As frmSplash
' instanciar la ventana de presentación
Set lfrmSplash = New frmSplash
' cargar la ventana de presentación
Load lfrmSplash
' mostrarla
lfrmSplash.Show
End Sub
```

Código fuente 17

Y el lector se estará preguntando: "ya que el procedimiento Main() es el encargado ahora de mostrar el formulario splash, ¿donde se realiza la carga del formulario principal de la aplicación?". Muy sencillo, la ventana splash es la que se ocupa a partir de ahora del formulario MDI.

El proceso es el siguiente: cuando se carga el formulario frmSplash, se pone en marcha el control Timer que incorpora; cada dos segundos, se llama al evento Timer() de dicho control y se ejecuta el código fuente 18

En este procedimiento se utiliza la variable mnEventosTimer, definida a nivel de módulo, como un contador de las veces que se produce este evento. Así, la primera vez se carga el formulario MDI y se vuelve a mostrar el formulario splash mediante el método Show(), ya que al visualizar el MDI, oculta el formulario splash que se estaba mostrando hasta ese momento. Mediante esta técnica, conseguimos mostrar simultáneamente ambos formularios. Finalmente, en el segundo evento del control Timer, se cierra la ventana splash, de forma que finaliza la presentación del programa, quedando listo para ser utilizado por el usuario.

```
Private Sub tmrTemporizador_Timer()
' asignar valor al contador de Timer
mnEventosTimer = mnEventosTimer + 1
' la primera vez que se llama al
' temporizador...
If mnEventosTimer = 1 Then
   ' instanciar el formulario mdi de la aplicación
   Set gmdiInicio = New mdiInicio
   ' cargar el formulario mdi
   Load gmdiInicio
   ' mostrarlo
   gmdiInicio.Show
```

```
   ' volver a mostrar el formulario splash
   ' ya que al visualizar la ventana mdi
   ' queda oculto
   Me.Show
Else
   ' en la siguiente llamada al temporizador
   ' cerrar la ventana splash
   Unload Me
End If
End Sub
```

Código fuente 18

3

El lenguaje

Introducción

Basic como lenguaje, ha supuesto el punto de entrada de muchas personas en el mundo de la programación. Quien más y quien menos ha utilizado este lenguaje para hacer sus primeros *pinitos* como programador. Gracias a su facilidad de aprendizaje, desde los tiempos de MS-DOS, versiones de Basic como QBasic, GWBasic, etc., se han utilizado para sentar las bases de las técnicas de programación en personas que, bien seguían programando en este lenguaje, o dependiendo de las aplicaciones a crear, pasaban a otro lenguaje más adecuado, pero siempre con la ventaja de poseer la sólida base adquirida con Basic.

Con el advenimiento de Windows como entorno de trabajo mayoritariamente adoptado por los usuarios, y Visual Basic, como herramienta de programación que se adapta perfectamente a dicho entorno, se ha producido un, digamos, *retorno a casa*; muchos programadores que habían empezado con Basic y habían pasado a otros productos que daban la potencia que Basic no ofrecía, han vuelto a sus orígenes debido a la gran cantidad de mejoras incorporadas en las sucesivas versiones aparecidas.

Desde las antiguas versiones de Basic hasta el actual Visual Basic 6, el aspecto del lenguaje ha sido mejorado en algunos puntos y reforzado añadiendo nuevas características que estudiaremos en este capítulo.

Visual Basic para aplicaciones (VBA)

Usualmente en el entorno ofimático, algunas aplicaciones (procesador de textos, hoja de cálculo, etc.) se acompañan de un sencillo lenguaje de macros para facilitar la tarea del usuario. El problema hasta

hace poco, era que las rutinas creadas para unas aplicaciones, no servían en otras, ya que los lenguajes aportados por cada producto eran incompatibles.

Visual Basic para Aplicaciones (VBA a partir de ahora) viene a suplir este problema. Un programa creado con VBA está pensado para ejecutarse desde una aplicación llamada *aplicación servidora*, de forma que el programa VBA personalice y configure a la aplicación servidora hasta puntos en los que mediante las tradicionales macros sería muy difícil o imposible de llegar.

Mediante la tecnología ActiveX y el modelo DCOM, los objetos de las diferentes aplicaciones, pueden compartir información, incluso los mismos objetos pueden ser utilizados en varias aplicaciones; una vez creado un objeto, no es necesario volver a programarlo para otra aplicación, simplemente se usa el que ya hay creado.

VBA y Visual Basic 6

Visto lo anterior, podemos deducir que VBA es una variante de VB6 adaptada para trabajar en aplicaciones servidoras. La diferencia principal es que mientras que con VB podemos crear aplicaciones independientes, un programa VBA necesita la aplicación servidora para poder funcionar; cosa lógica si pensamos que se trata de un complemento de la aplicación servidora.

Aunque existen diferencias como la anterior, hay que tener en cuenta que existe un mayor número de parecidos que de diferencias. Tanto en el lenguaje como en el editor y el resto de los elementos, si el lector ya utiliza Visual Basic se adaptará bien a VBA.

Emplazamiento del código de la aplicación

El código fuente de los programas se incluye en los llamados módulos de código, que contienen el código de un formulario, clase, control, procedimientos generales de la aplicación, etc.

Es una buena costumbre a la hora de organizar el código de un proyecto, situar todos los procedimientos que no estén directamente relacionados con ningún formulario, control, clase, etc., en módulos de código estándar (ficheros .BAS); de manera que tengamos separados los procedimientos y métodos de los formularios en su módulo; las propiedades y métodos de una clase en el suyo, y las funciones y procedimientos generales para toda la aplicación en otro.

Comentarios

Durante la escritura del programa, se hace necesario incluir los llamados comentarios o notas aclaratorias sobre lo que hacen diferentes partes del programa.

En VB se utiliza el carácter apóstrofe (‘) para indicar que lo que sigue a continuación en esa línea es un comentario; puede ocupar una línea completa o bien se puede combinar una sentencia junto a un comentario como podemos ver en el código fuente 19.

Al ser sólo elementos informativos de cara al programador, en la fase de compilación y generación de la aplicación serán ignorados por VB.

```
Public Sub Main()
Dim lnNum As Integer
```

```
' este comentario ocupa toda la línea
lnNum = 10 ' en esta línea se combina código y comentario
End Sub
```

Código fuente 19

MsgBox()

Durante la ejecución de los ejemplos y en las aplicaciones que desarrollemos, se hará necesario informar al usuario de ciertos valores, para lo cual utilizaremos esta función, que visualiza una caja de diálogo con un mensaje informativo, a la que el usuario deberá responder pulsando en uno de los botones que aparezcan en dicha caja; podemos adicionalmente, recoger el valor de retorno de esta función que indica mediante un Integer el botón pulsado.

Sintaxis:

```
MsgBox(cMensaje, nBotones, cTítulo, cFichAyuda, nContexto)→
nBotón
```

- **cMensaje.** Cadena de caracteres con el texto informativo para el usuario, si queremos organizar el texto en varias líneas, podemos incluir un retorno de carro y avance de línea con la constante de VB vbCrLf.

- **nBotones.** Número o suma de números que identifican a los botones e icono que aparecen en la caja. En lugar de los números, podemos utilizar las constantes de la tabla 20 que darán mayor claridad al código.

Constante	**Tipo de botón, icono y mensaje**
vbOKOnly	Botón Aceptar.
vbOKCancel	Botones Aceptar y Cancelar.
vbAbortRetryIgnore	Botones Anular, Reintentar e Ignorar.
vbYesNoCancel	Botones Sí, No y Cancelar.
vbYesNo	Botones Sí y No.
vbRetryCancel	Botones Reintentar y Cancelar.
vbCritical	Icono de mensaje crítico.
vbQuestion	Icono de pregunta.
vbExclamation	Icono de advertencia.
vbInformation	Icono de información.
vbDefaultButton1	Primer botón es el predeterminado.

vbDefaultButton2	Segundo botón es el predeterminado.
vbDefaultButton3	Tercer botón es el predeterminado.
vbDefaultButton4	Cuarto botón es el predeterminado.
vbApplicationModal	Mensaje de aplicación modal; el usuario debe responder al mensaje antes de poder continuar con la aplicación.
vbSystemModal	Mensaje modal de sistema; el usuario debe responder al mensaje antes de poder continuar con cualquier aplicación.

Tabla 20. Constantes disponibles para los botones de la función MsgBox().

- **cTítulo.** Cadena de caracteres con el título del mensaje, si se omite se utiliza el nombre de la aplicación.
- **cFichAyuda.** Cadena de caracteres con el nombre del fichero de ayuda para el mensaje.
- **nContexto.** Número que contiene el contexto de ayuda en el fichero especificado en cFichAyuda.
- **nBotón.** Número que contiene el valor de retorno del botón pulsado en la caja de mensaje. Los valores están disponibles en forma de constantes que podemos ver en la tabla 21.

Constante	**Botón pulsado**
vbOK	Aceptar
vbCancel	Cancelar
vbAbort	Anular
vbRetry	Reintentar
vbIgnore	Ignorar
vbYes	Sí
vbNo	No

Tabla 21

Si no necesitamos recuperar el valor que devuelve la función, utilizaremos MsgBox() como indica el código fuente 20.

```
Public Sub Main()
MsgBox "Introduzca un valor", vbOKOnly, "Atención"
End Sub
```

Código fuente 20

El resultado lo muestra la figura 69.

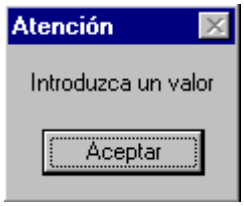


Figura 69. Resultado de la función MsgBox().

Si necesitamos saber cual botón ha sido pulsado, lo haremos como indica el código fuente 21.

```
Public Sub Main()
‘ declaramos una variable para
‘ guardar el valor del botón pulsado
Dim lnRespuesta As Integer
‘ asignamos a la variable el resultado de MsgBox()
lnRespuesta = MsgBox("Introduzca un valor", vbYesNoCancel + vbCritical, "Atención")
‘ en función del valor de respuesta de MsgBox()
‘ actuamos de una forma u otra
If lnRespuesta = vbCancel Then
    Exit Sub
End If
End Sub
```

Código fuente 21

La forma de utilizar una función en el código y las estructuras de control como If...End If serán tratadas posteriormente en otros apartados de este capítulo.

InputBox()

Esta función muestra una caja de diálogo en la que existe un TextBox para que el usuario introduzca un valor, devolviendo una cadena con el resultado la acción tomada por el usuario: si pulsa el botón *Aceptar* o *Enter*, el valor de retorno será el contenido del TextBox de la caja de diálogo. Si pulsa el botón *Cancelar* o *Escape*, se devolverá una cadena vacía.

Sintaxis:

```
InputBox(cTexto, cTitulo, cTextoDefecto, nXPos, nYPos,
cFicheroAyuda, nContextoAyuda) ---> cadena
```

- **cTexto.** Cadena de caracteres con un texto informativo para el usuario. Sigue las mismas normas que para la función MsgBox().
- **cTitulo.** Cadena de caracteres con el título para la ventana.
- **cTextoDefecto.** Cadena con el texto que aparecerá en el TextBox. Si no se proporciona, el control aparecerá vacío.
- **nXPos, nYPos.** Coordenadas de situación de la ventana.

- **cFicheroAyuda.** Nombre del fichero de ayuda asociado a esta ventana.
- **nContextoAyuda.** Número con el identificador del tema de ayuda.

```
Public Sub Main()
Dim lcValor As String
lcValor = InputBox("Introduzca nombre completo", "Toma de datos")
End Sub
```

Código fuente 22.

El resultado del código fuente 22, es la figura 70.

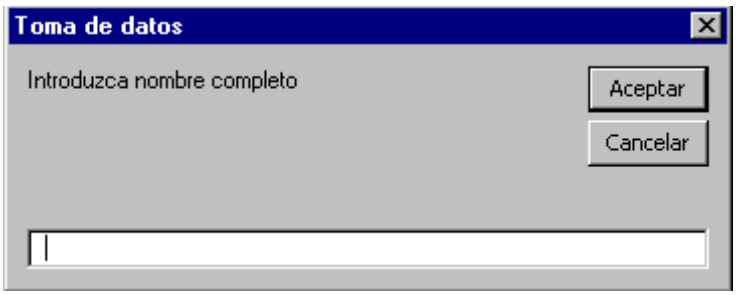


Figura 70. Función InputBox() en ejecución.

Variables

Una variable es un elemento del código que almacena un valor, este valor puede cambiar durante el ciclo de vida de la variable. Las variables tienen las siguientes características:

Declaración

Antes de usar una variable, es recomendable declararla utilizando alguna de estas sentencias: *Dim, Public, Private o Static.* Si no la declaramos, VB la crea internamente, pero la no declaración de variables en el código es causa segura de problemas. Observemos el código fuente 23. Aquí hemos declarado todas las variables menos *lcApellido1*, que hemos incluido en el código y asignado un valor directamente, lo que no provoca un error en ejecución.

El problema sobreviene al asignar a *lcNomCompleto* el valor de las variables *lcNombre* y *lcApellido1*, hemos olvidado poner el "1" en *lcApellido1*. En este punto VB no puede determinar si está trabajando con una nueva variable o con una existente, erróneamente tecleada. Por eso decide crear una nueva variable *lcApellido* sin valor, lo que supone que a la variable *lcNomCompleto* se le asigne el contenido de *lcNombre* y de la nueva variable *lcApellido* que está vacía.

```
Dim lcNombre As String
Dim lnNumero As Integer
Dim lcNomCompleto As String
lcNombre = "Juan"
lnNumero = 152
lcApellido1 = "Torres"
lcNomCompleto = lcNombre & lcApellido
```

```
‘ el contenido de lcNomCompleto es "Juan"
```

Código fuente 23

Para evitar este tipo de problemas, disponemos de la sentencia *Option Explicit*, que situada en la zona de declaraciones del módulo de código, obliga a declarar todas las variables utilizadas, en caso contrario, al intentar ejecutar el programa, mostrará un error informándonos del problema.

Option Explicit sólo comprobará las variables en el módulo en el que esté situada, si queremos que VB sitúe esta sentencia automáticamente en todos los módulos, hemos de seguir los siguientes pasos:

- Abrir el menú *Herramientas.*
- Seleccionar la opción *Opciones*, aparecerá su caja de diálogo.
- Hacer clic en la pestaña *Editor.*
- Marcar la opción *Requerir declaración de variables.*

En todos los ejemplos utilizados en este curso se utilizará la sentencia *Option Explicit*, reiterando al lector la conveniencia de su uso como medio de obtener un código más libre de errores.

Tipo

Al declarar una variable, podemos asignarle el tipo de dato que va a contener. La tabla 22 muestra una relación de los tipos de datos disponibles en Visual Basic.

Tipo de dato	**Rango de valores disponibles**
Byte	0 a 255
Boolean	True o False
Integer (entero)	-32.768 a 32.767
Long (entero largo)	-2.147.483.648 a 2.147.483.647
Single (coma flotante de precisión simple)	Valores positivos: 1,401298E-45 a 3,402823E38 Valores negativos: -3,402823E38 a -1,401298E-45
Double (coma flotante de precisión doble)	Valores positivos: 4,94065645841247E-324 a 1,79769313486232E308 Valores negativos: -1,79769313486232E308 a -4,94065645841247E-324
Currency (entero a escala – valores monetarios)	-922.337.203.685.477,5808 a 922.337.203.685.477,5807
Decimal	+/-79.228.162.514.264.337.593.543.950.335 (a)

	+/-7,9228162514264337593543950335 (b) +/-0,0000000000000000000000000001 (c)
Date (fecha)	1 de Enero de 100 a 31 de Diciembre de 9999
Object	Cualquier tipo de objeto
String (cadena de longitud variable)	De 0 a 2.000 millones
String (cadena de longitud fija)	De 1 a 65.400 aprox.
Variant (usando números)	Cualquier número hasta intervalo de un Double
Variant (usando caracteres)	Igual que para un String de longitud variable

(a) Sin punto decimal.
(b) Con 28 posiciones a la derecha del signo decimal.
(c) Número más pequeño distinto de cero.

Tabla 22. Tipos de datos en Visual Basic.

Para especificar el tipo de la variable utilizaremos la cláusula *As* en la declaración, seguida del tipo de dato, como se indica en el código fuente 24.

```
Dim lcNombre As String ‘ variable de tipo carácter
```

Código fuente 24

Si no indicamos el tipo de la variable en la declaración, se tomará por defecto el tipo Variant, como se muestra en el código fuente 25. Variant es un tipo de dato genérico, es decir, que puede contener cualquier tipo de los existentes en VB con excepción de cadenas de longitud fija y tipos definidos por el usuario.

Si observamos el código fuente 26, vemos que en una misma variable Variant hemos introducido diferentes tipos de datos, y VB ha efectuado la conversión de forma automática al asignarle un tipo distinto de dato, con lo que podemos hacernos una idea de la flexibilidad al usar Variant; este hecho tiene sus inconvenientes, por supuesto, ya que hemos de tener en cuenta a efectos de rendimiento, que el uso de variables Variant en el programa supone un trabajo extra para VB, que ha de estar comprobando que tipo de dato contiene la variable cuando va a realizar alguna operación con ella.

```
‘ las dos líneas siguientes son equivalentes
Dim lvntCaja
Dim lvntCaja As Variant
```

Código fuente 25

```
Public Sub Main()
Dim lvntCualquiera As Variant
lvntCualquiera = #5/10/98#
If IsDate(lvntCualquiera) Then
    MsgBox "La variable lvntCualquiera contiene una fecha"
End If
lvntCualquiera = 28530
If IsNumeric(lvntCualquiera) Then
    MsgBox "La variable lvntCualquiera contiene un número"
End If
lvntCualquiera = "mesa"
If VarType(lvntCualquiera) = vbString Then
    MsgBox "La variable lvntCualquiera contiene una cadena de caracteres"
End If
End Sub
```

Código fuente 26

Por este motivo, es aconsejable que al declarar una variable en nuestro código, si sabemos de antemano que esa variable va a contener el mismo tipo de dato durante la ejecución del programa, la declaremos con el tipo de dato que le corresponda. Esto evitará a VB tener que comprobar el tipo de dato, y producirá un ejecutable más rápido.

Ámbito

Entendemos como ámbito, el nivel de visibilidad de una variable dentro del código de una aplicación, es decir, los lugares del código de una aplicación donde vamos a poder utilizar el valor de una variable en función de cómo se haya declarado.

El ámbito más restringido para una variable es la declaración dentro de un procedimiento. En el código fuente 27, la variable *lnValor* se declara con el llamado *ámbito local* o a *nivel de procedimiento*.

```
Private Sub Convertir()
Dim lnValor As Integer
............
............
End Sub
```

Código Fuente 27

Las variables con ámbito local sólo son visibles en el procedimiento que las ha declarado. Si intentamos utilizarlas desde otro procedimiento, se producirá un error.

Dentro de un procedimiento, podemos declarar las variables utilizando dos sentencias diferentes:

- **Dim.** Las variables declaradas con Dim se crean al entrar en el procedimiento que las contiene, y se destruyen al salir de él, liberando los recursos que ocupaban, de forma que no conservan su valor entre llamadas al mismo procedimiento. El código fuente 28 es un ejemplo de declaración de este tipo.

```
Private Sub Calculo()
' declarar la variable
Dim lcNombre As String
' asignarle un valor
lcNombre = "Juan"
' al salir del procedimiento se destruye
' la variable
End Sub
```

Código fuente 28

- **Static.** Las variables declaradas con Static, una vez creadas, mantienen su valor entre llamadas al procedimiento, y no se destruyen hasta que no finaliza la aplicación. A pesar de que su ciclo de vida es el mismo del programa, sólo son visibles dentro del procedimiento en el que fueron declaradas, no se puede acceder a ellas desde otro procedimiento. En el código fuente 29, podemos ver un ejemplo de su comportamiento.

```
Public Sub Main()
' en la primera llamada se crea la variable
Calculo
' en la siguiente llamada, la variable ya está
' creada y se acumula el nuevo valor al que
' ya tenía
Calculo
End Sub
‘-----------------------------------------------
Private Sub Calculo()
' declarar la variable
Static scNumero As Integer
' sumar un valor a la variable
scNumero = scNumero + 100
' visualizar valor
MsgBox "La variable scNumero contiene " & scNumero
' al salir del procedimiento, la variable
' no se destruye
End Sub
```

Código fuente 29

En el código fuente 29, creamos un procedimiento *Calculo()*, que contiene una variable Static *scNumero*. La primera vez que llamamos a *Calculo()*, se crea la variable y toma un valor. En la segunda llamada a *Calculo()*, *scNumero* contiene el valor 100 que se le había asignado en la anterior llamada y que conserva, por lo que al sumar una nueva cantidad a la variable, quedará con el valor 200.

Cuando necesitemos compartir una variable entre varias rutinas del mismo módulo, tendremos que declararla en la zona de declaraciones del módulo con la sentencia *Dim o Private*. El ámbito para estas variables se denomina a *nivel de módulo.*

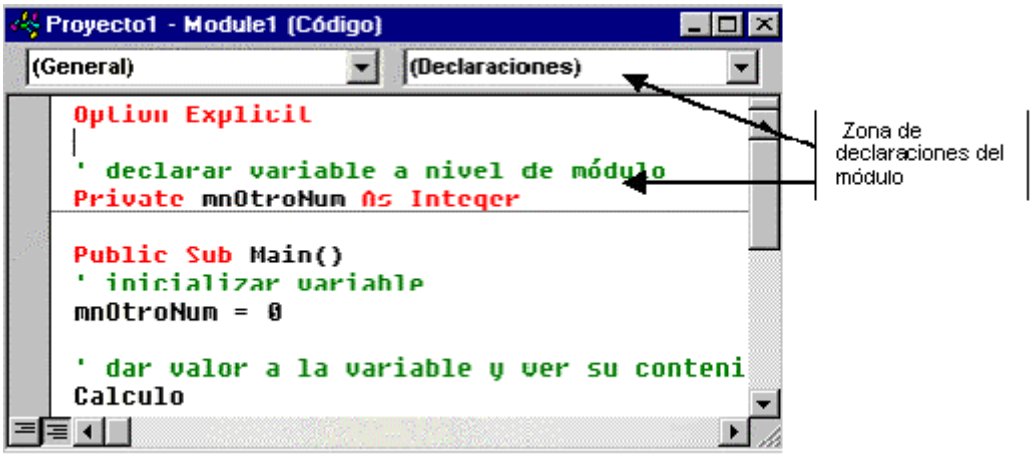


Figura 71. Zona de declaraciones del módulo de código.

En el código fuente 30, vemos como la variable *mnOtroNum* es accesible desde varios procedimientos del módulo.

```
Option Explicit
' declarar variable a nivel de módulo
Private mnOtroNum As Integer
‘ ----------------------------------------

Public Sub Main()
' inicializar variable
mnOtroNum = 0
' dar valor a la variable y ver su contenido
Calculo
MsgBox "La variable mnOtroNum contiene " & mnOtroNum
Calculo
MsgBox "La variable mnOtroNum contiene " & mnOtroNum
End Sub
‘ ----------------------------------------

Public Sub Calculo()
' dar valor a la variable
mnOtroNum = mnOtroNum + 200
End Sub
```

Código fuente 30

Finalmente, si queremos que la variable sea visible desde cualquier punto de la aplicación, es necesario declararla como Public a nivel de módulo; este es el llamado *ámbito global*. Con este tipo de ámbito no tenemos restricciones de visibilidad, las variables son accesibles desde cualquier punto de la aplicación, se pueden usar desde todos los módulos del proyecto.

Denominación

Las variables se identifican en el código mediante un nombre. Este nombre puede estar formado por cualquier carácter excepto los que tienen un significado especial para VB, como pueden ser paréntesis, símbolos de operadores, etc. Tampoco está permitido utilizar nombres de instrucciones, palabras

reservadas ni funciones propias de VB. Lo más conveniente para el grupo de trabajo o programador independiente, es seguir una serie de reglas o normas de codificación.

```
‘ variable correctamente declarada
Dim lnElementos As Integer
‘ variables incorrectamente declaradas
Dim 12Hola
Dim Valor(?Arriba
```

Código fuente 31

Como *normativa de notación*, podemos entender un conjunto de reglas que utilizaremos en el desarrollo de una aplicación para denominar a los diferentes elementos que la componen: variables, constantes, controles, objetos, etc. Si bien esto no es inicialmente necesario, ni la herramienta de programación obliga a ello, en la práctica se ha demostrado que una serie de normas a la hora de escribir el código redundan en una mayor velocidad de desarrollo y facilidad de mantenimiento de la aplicación. Siendo útil no sólo en grupos de trabajo, sino también a programadores independientes.

En el Apéndice I vienen descritas una serie de normas de codificación, que no son en absoluto obligatorias a la hora de escribir el código de la aplicación, pero si pretenden concienciar a los lectores de la necesidad de seguir unas pautas comunes a la hora de escribir dicho código, de manera que al compartir código entre programadores, o cuando tengamos que revisar una aplicación desarrollada varios meses atrás, empleemos el menor tiempo posible en descifrar lo que tal o cual variable significa en el contexto de una rutina o módulo.

Una vez declarada la variable, en función del tipo de dato con que se ha definido, habrá que asignarle un valor.El código fuente 32 es una pequeña muestra de la manera de asignar valores a las variables de tipos más comunes.

```
‘ declarar las variables
Dim lnCodigo As Integer
Dim lcNombre As String
Dim lcIniciales As String * 4
Dim ldtFActual As Date
Dim loVentana as Object
lnCodigo = 234
lcNombre = "José Carlos"
‘ al asignar una cadena a una variable String de
‘ longitud fija, si la cadena a asignar excede de la
‘ longitud que tiene la variable, el resto se trunca
lcIniciales = "JCGF"
‘ al inicializar una fecha se hará usando el carácter #
‘ y en el formato mes/dia/año
ldtFActual = #10/24/98#
‘ para inicializar un objeto usaremos la palabra clave Set
Set loVentana = New frmMuestra
```

Código fuente 32

Constantes

Durante el desarrollo de una aplicación, necesitaremos usar ciertos valores fijos para todo el programa. Supongamos que vamos a editar los registros de una tabla de una base de datos y queremos saber

cuando agregamos un registro nuevo o modificamos uno existente. Una forma de resolverlo sería declarar una variable y asignarle un "1" para indicar que estamos agregando y "0" cuando estamos modificando. Si este tipo de situaciones se repite con frecuencia a lo largo del programa, nos encontraremos con un gran número de valores difíciles de recordar.

Para solucionar este problema disponemos de las constantes, que son un elemento del lenguaje que contendrá un valor que no podrá ser alterado. Se declaran usando la instrucción *Const*, y siguen las mismas reglas de ámbito que las variables.

De esta forma podríamos declarar el código fuente 33.

```
Private Const REGISTRO_AGREGANDO As Integer = 1
Private Const REGISTRO_MODIFICANDO As Integer = 0
‘ ---------------------------------------------
Private Sub GrabarRegistro()
Select Case mnEstadoRegistro
Case REGISTRO_AGREGANDO
' crear nuevo registro
.........
.........
.........
Case REGISTRO_MODIFICANDO
' editar registro actual
.........
.........
.........
End Select
End Sub
```

Código Fuente 33

Visual Basic dispone de un extenso conjunto de constantes predefinidas, que nos ayudarán a la hora de escribir el código, y que podemos consultar en el Examinador de objetos en el caso de que no las recordemos.

En el código fuente 34, cargamos un formulario y al mostrarlo podemos utilizar un número para indicar si es o no modal, o podemos usar las constantes que para el caso proporciona VB: *vbModal* y *vbModeless.*

```
Public Sub Main()
Load Form1
‘ las dos líneas siguientes serían equivalentes
Form1.Show 1
Form1.Show vbModal ‘ esta es más fácil de recordar
End Sub
```

Código fuente 34

El código fuente 34 sólo maneja dos valores, por lo cual podemos argumentar que no es necesario el uso de las constantes. El código fuente 35, sin embargo, utiliza la función MsgBox(), en la que podemos utilizar un mayor conjunto de números para indicar el tipo de icono a presentar al usuario, botones y cual será el botón por defecto. De la misma manera, también tenemos varios valores para la respuesta dada por el usuario. Veamos el mismo ejemplo utilizando directamente los valores numéricos y después con las constantes predefinidas.

```
‘ usando valores directos
Public Sub Main()
Dim lnRespuesta As Integer
lnRespuesta = MsgBox("¿Valores correctos?", 16 + 3 + 256, "¡Atención!")
If lnRespuesta = 2 Then
    Exit Sub
End If
End Sub
‘ ------------------------------------------------------------
‘ usando constantes predefinidas
Public Sub Main()
Dim lnRespuesta As Integer
lnRespuesta = MsgBox("¿Valores correctos?",
vbCritical + vbYesNoCancel + vbDefaultButton2, "¡Atención!")
If lnRespuesta = vbCancel Then
    Exit Sub
End If
End Sub
```

Código fuente 35

Definitivamente, el último de los dos fuentes es mucho más comprensible gracias al uso de constantes.

El tipo Enumerado

Una variable declarada como de tipo enumerado, consta de un conjunto de valores de tipo Long, a los que se accede mediante un nombre miembro.

La forma de declarar una variable enumerada es utilizando la instrucción *Enum*, que tiene la siguiente sintaxis:

```
Enum cVariable
      cMiembro [= lValor]
      cMiembro [= lValor]
          .........
          .........
          .........
End Enum
```

Una declaración Enum consta de las siguientes partes:

- **cVariable.** El nombre de variable que va a contener el tipo enumerado. Se han de seguir las mismas normas de declaración que para el resto de variables.

- **cMiembro.** Nombre de cada uno de los miembros integrantes de la enumeración.

- **lValor.** Valor opcional de tipo Long para un miembro de la enumeración. Si este valor no se proporciona, el primer miembro tendrá el valor cero, y los siguientes tendrán el valor del miembro anterior más uno.

Los tipos enumerados deben crearse en la zona de declaraciones del módulo. Son públicos por defecto, pudiendo acceder a ellos desde cualquier punto de la aplicación.

Aunque un tipo enumerado toma valores por defecto, podemos asignar valores desde uno a todos los miembros del tipo, pudiendo comenzar las asignaciones por cualquiera de los miembros.

El código fuente 36, declara un tipo enumerado con los valores por defecto:

```
Enum Estaciones
    Primavera ' 0
    Verano ' 1
    Otoño ' 2
    Invierno ' 3
End Enum
```

Código fuente 36

En el código fuente 37, asignamos valores comenzando por un miembro que no sea el primero:

```
Enum Estaciones
    Primavera ' 0
    Verano ' 1
    Otoño = 16 ' 16
    Invierno ' 17
End Enum
```

Código fuente 37

En el código fuente 38, realizamos dos asignaciones con valores negativos y positivos:

```
Enum Estaciones
    Primavera = -6 ' -6
    Verano ' -5
    Otoño = 208 ' 208
    Invierno ' 209
End Enum
```

Código fuente 38

La forma de referirse a los valores de un tipo enumerado en el código es mediante: *Variable.Miembro*, como nos muestra código fuente 39.

```
Private Sub Main(
' mostrar los valores del tipo enumerado Estaciones
MsgBox "Estación Primavera, valor: " & Estaciones.Primavera
MsgBox "Estación Verano, valor: " & Estaciones.Verano
MsgBox "Estación Otoño, valor: " & Estaciones.Otoño
MsgBox "Estación Invierno, valor: " & Estaciones.Invierno
End Sub
```

Código fuente 39

Conversión de tipos

Al estudiar los diferentes tipos de datos en VB, hemos visto como Variant puede almacenar la mayoría de los tipos existentes; por esta causa habrá situaciones en las que debamos convertir su tipo a otro distinto para poder realizar una operación. Este caso puede darse tanto en variables Variant como de otro tipo, que requieran adaptarse, si el contenido de la variable lo permite, a un tipo distinto.

Al hablar de cambio de tipo, no nos referimos a que una variable declarada por ejemplo como Boolean pueda transformarse a String, ya que no está permitido; lo que queremos decir es que tomamos el contenido de la variable y lo pasamos a otra variable previa conversión de tipo al que nos interesa, para lo cual el valor que se convierte ha de ser aceptado por la variable receptora sin provocar un error de tipos.

Denominamos *error de tipos* al error producido al intentar asignar a una variable un tipo de dato distinto del que contiene su definición. Visual Basic intentará en el momento de la asignación hacer una conversión de tipos para adaptar el valor a la variable receptora, provocando un error en el caso de que esto no sea posible. También podemos forzar la conversión del tipo mediante una serie de funciones destinadas a tal efecto relacionadas en la tabla 23.

Función	**Convierte a**
CBool(expresión)	Boolean
CByte(expresión)	Byte
CCur(expresión)	Currency
CDate(expresión)	Date
CDbl(expresión)	Double
CDec(expresión)	Decimal
CInt(expresión)	Integer
CLng(expresión)	Long
CSng(expresión)	Single
CVar(expresión)	Variant
CStr(expresión)	String

Tabla 23. Funciones de conversión.

Igualmente se producirá un error si al utilizar estas funciones, no puede realizarse la conversión a un tipo de datos determinado.

En el código fuente 40 se intentan realizar varias conversiones de tipos, produciéndose errores en algunas por problemas con los tipos de datos. Las líneas que generan error están indicadas con comentarios.

```
Public Sub Main()
Dim lbSiNo As Boolean
Dim lnNumInt As Integer
Dim llNumLong As Long
Dim lcCadena As String
Dim lcCadenaBis As String
Dim ldtFecha As Date
' convertir a Boolean una cadena
lcCadena = "321"
lbSiNo = CBool(lcCadena) ' True
lcCadena = "prueba"
lbSiNo = CBool(lcCadena) ' error de tipos
' convertir a numérico Integer una cadena
lcCadena = "4677"
lnNumInt = lcCadena ' 4677
' aquí se produce un error de desbordamiento,
' ya que al convertir la cadena a número, este
' queda como Long y no es posible asignarlo
' a un Integer
lcCadena = "124677"
lnNumInt = lcCadena ' error
lnNumInt = CInt(lcCadena) ' error
' convertir a fecha una cadena
lcCadena = "30/7/1998"
ldtFecha = lcCadena ' conversión directa
ldtFecha = CDate(lcCadena) ' conversión mediante función
lcCadena = "palabra"
ldtFecha = CDate(lcCadena) ' error de tipos
' convertir un número a cadena
lnNumInt = 26544
lcCadenaBis = lnNumInt
' convertir una fecha a cadena
lcCadenaBis = #12/30/97#
End Sub
```

Código fuente 40

Para prevenir posibles errores debidos a la conversión de tipos, disponemos de una serie de funciones que nos informan del tipo de datos que contiene la expresión a convertir, algunas de las cuales podemos ver a continuación:

IsNumeric(xExpresion)

Devuelve un valor lógico que indica si la expresión pasada como parámetro se puede convertir o no a número.

IsDate(xExpresion)

Devuelve un valor lógico que indica si la expresión pasada como parámetro se puede convertir o no a fecha.

TypeName(variable)

Devuelve una cadena que informa del valor de la variable pasada como parámetro. Dicha cadena puede contener los siguientes valores: Byte, Integer, Long, Single, Double, Currency, Decimal, Date, String, Boolean, Error, Empty, Null, Object.

VarType(variable)

Devuelve un número indicando el tipo de dato contenido en el parámetro. Estos números están disponibles a través de un conjunto de constantes predefinidas para facilitar su identificación: vbInteger, vbString, vbLong, etc.

En el código fuente 41, vamos a ver al código fuente 40 optimizado para comprobar el tipo de dato antes de efectuar la conversión; sólo se muestras las líneas optimizadas que ahora no provocan error.

```
Public Sub Main()
Dim lbSiNo As Boolean
..........
..........
..........
lcCadena = "prueba"

If IsNumeric(lcCadena) Then
    lbSiNo = CBool(lcCadena)
End If
..........
..........
..........
lcCadena = "124677"
vntNum = Val(lcCadena)
MsgBox "El tipo de la variable vntNum es: " & TypeName(vntNum)
' convertir a fecha una cadena
..........
..........
..........
lcCadena = "palabra"
If IsDate(lcCadena) Then
    ldtFecha = CDate(lcCadena)
End If
..........
..........
..........
End Sub
```

Código fuente 41

Continuación de línea de código

Hay momentos en que al escribir una línea compleja de código, esta no cabe en la parte visible de la pantalla; en estos casos la recomendación es no escribir superando el margen derecho de la pantalla, sino poner un carácter de guión bajo "_", que VB interpreta como continuación de línea y seguir en la línea siguiente. El compilador lo interpretará como si fuera una única línea. El código fuente 42 nos lo muestra.

```
Public Sub Main()
Dim lcNombre As String
Dim lcPriApellido As String
Dim lcSegApellido As String
Dim lcNomCompleto As String
lcNombre = "Pedro"
lcPriApellido = "Mesa"
lcSegApellido = "Gutierrez"
‘ si la línea es demasiado larga
‘ terminamos en la siguiente
```

```
lcNomCompleto = lcNombre & lcPriApellido & _
lcSegApellido
End Sub
```

Código fuente 42

Instrucciones en una línea

Utilizando el carácter de dos puntos ":", podemos incluir más de una sentencia en la misma línea. Sólo hemos de separar cada sentencia con dicho carácter.

```
Public Sub Main()
Dim lnValor As Integer
Dim lnNumero As Integer
Dim lnTotal As Integer
lnValor = 231: lnNumero = 45: lnTotal = lnValor + lnNumero
End Sub
```

Código fuente 43

En el código fuente 43, asignamos valores a variables y hacemos una suma de las mismas, todo en la misma línea de código.

Aunque es interesante conocer la existencia de esta característica, por motivos obvios de legibilidad del código, no es recomendable su uso.

Expresiones

Una expresión es un conjunto de elementos del lenguaje: variables, constantes, operadores, palabras clave, etc., que se unen para obtener un resultado de tipo carácter, lógico o numérico. La expresión del código fuente 44 realiza varias operaciones matemáticas para producir un resultado.

```
lnTotalOp = 5 + 15 / 10 * 21
```

Código fuente 44

Operadores

Cuando necesitemos combinar diversos elementos del código para obtener un resultado, por ejemplo: sumar dos variables o unir dos cadenas de caracteres para formar otra nueva, haremos uso de los operadores. Visual Basic dispone de un nutrido grupo de operadores agrupados en diferentes categorías, en función del tipo de expresión a construir.

Aritméticos

Realizan el conjunto habitual de operaciones matemáticas. Como norma general, si uno de los operandos es Null, el resultado de la operación será Null. También en la mayor parte de los casos, si una expresión es Empty, se evaluará como 0.

- ^ **Potencia.** Para realizar una potenciación, hemos de situar a la izquierda el operando que actúe como base y a la derecha el exponente. En el código fuente 45 vemos un ejemplo.

```
ldResultado = 15 ^ 3 ‘ 3375
```

Código fuente 45

El resultado puede ser negativo si el exponente es un entero. Podemos realizar varios cálculos de potencia en una sola instrucción, siendo evaluados de izquierda a derecha.

El tipo de dato devuelto por la operación es Double.

- *** Multiplicación.** Vemos un ejemplo en el código fuente 46.

```
ldResultado = 44 * 5 ‘ 220
```

Código fuente 46

El tipo de dato devuelto por una multiplicación se adapta al que sea más aproximado al resultado, con las siguientes variaciones:

  - Si la multiplicación es entre un Single y un Long, el resultado es un Double.
  - Cuando el resultado es un Variant con valor Long, Single o Date, que se sale de rango, se convierte a Variant con valor Double.
  - Cuando el resultado es un Variant con valor Byte, que se sale de rango, se convierte a Variant con valor Integer.
  - Cuando el resultado es un Variant con valor Integer, que se sale de rango, se convierte a Variant con valor Long.

- / División. Divide dos números y devuelve un resultado con precisión decimal. Vemos un ejemplo en el código fuente 47.

```
ldResultado = 428 / 17 ‘ 25,1764705882353
```

Código fuente 47

El tipo de dato resultante es Double excepto en los siguientes casos:

- Si ambos operandos son Byte, Integer o Single, resulta un Single, provocándose un error si excede su rango válido.
- Si ambos operandos son Variant con valor Byte, Integer o Single, resulta un Single. En el caso de que exceda su rango válido será un Variant con valor Double.
- Cuando sea un Decimal y otro tipo, el resultante será Decimal.

- \ **División.** Divide dos números y devuelve un resultado con valor entero. Vemos un ejemplo en el código fuente 48.

```
ldResultado = 428 \ 17 ‘ 25
```

Código fuente 48

Antes de hacer la división, se redondean los números para convertirlos a Byte, Integer o Long, estos mismos tipos serán los resultantes de la operación, o un Variant con el valor de esos tipos. La parte fraccionaria de los números se trunca.

- **Mod.** Devuelve el resto de la división de dos números. Vemos un ejemplo en el código fuente 49.

```
ldResultado = 428 Mod 17 ‘ 3
```

Código Fuente 49

El resultado es un tipo Byte, Integer o Long; o un Variant con el valor de esos tipos.

- **+ Suma.** Aparte de efectuar una suma de números, permite incrementar una fecha en una cantidad determinada de días o concatenar cadenas de caracteres. Para esto último se recomienda usar el operador &, de manera que en un rápido vistazo al código sepamos si estamos sumando o concatenando. Vemos unos ejemplos en el código fuente 50.

```
‘ con números:
lnResultado = 2533 + 431 ‘ 2964

‘--------------------------------

‘ con fechas:
‘ la asignación de fecha a una variable es
‘ en el formato #mes/día/año#
ldtFecha = #3/15/97# ‘ 15/03/97
ldtFecha = ldtFecha + 5 ‘ 20/03/97
```

Código fuente 50

Cuando una de las expresiones a sumar no es Variant, se producen los siguientes resultados:

- La suma se produce cuando ambas partes son números, o una es Variant.
- La concatenación se produce cuando ambas partes son String, o una es Variant.
- Si un operando es Variant Empty, devuelve el resto de la expresión sin modificar.
- Si un operando es numérico y el otro String se produce un error.
- Cuando las expresiones a sumar son Variant, se aplica lo siguiente:
- Se produce una suma cuando ambos operandos contienen números, o uno es numérico y el otro cadena de caracteres.
- Se concatenan las expresiones si las dos son cadenas de caracteres.
- Por norma general, en la suma de números, el tipo del resultado es el del operando con mayor precisión, siendo este orden de precisión: Byte, Integer, Long, Single, Double, Currency y Decimal. En este punto se dan una serie de excepciones:
- El resultado es Double al sumar un Single y un Long.
- Se produce una conversión a Variant con valor Double si el resultado es Long, Single o Date con valor Variant que excede su rango.
- Se produce una conversión a Variant con valor Integer si el resultado es Byte con valor Variant que excede su rango.
- Se produce una conversión a Variant con valor Long si el resultado es Integer con valor Variant que excede su rango.
- Al sumar a un tipo Date siempre se obtiene un Date.

- **- Resta.** Realiza una resta entre dos números o la negación de un número. Vemos unos ejemplos en el código fuente 51.

```
‘ resta normal
lnResultado = 100 – 25 ‘ 75
‘ negación
lnNum1 = 84
lnNum2 = -lnNum1 ‘ –84
```

Código fuente 51

Los tipos de datos resultantes siguen las mismas reglas que en la suma, dándose el caso adicional de que al restar dos tipos Date, el resultado es Double.

Comparación

Estos operadores devuelven True (Verdadero) o False (Falso) como resultado de comparar dos expresiones. Si alguna de estas expresiones es Null, se devuelve siempre Null.

Al realizar comparaciones entre cadenas de caracteres hay que tener en cuenta la siguiente instrucción

Option Compare TipoComparacion

Se usa a nivel de módulo para especificar el tipo de comparación entre cadenas en el módulo donde se define. TipoComparacion puede ser *Binary*, *Text o Database*, siendo Binary el modo usado por defecto.

Cuando se usa Binary, se realiza una comparación basada en los valores binarios de los caracteres; con lo que una expresión como "a" = "A" resultaría falso.

Cuando se usa Text, se realiza una comparación basada en los propios caracteres sin distinguir mayúsculas de minúsculas y en la configuración del país establecida en el sistema. Aquí una expresión como "a" = "A" resultaría verdadero.

La opción Database sólo se puede usar con Access, con la que se hacen comparaciones basadas en el orden de los datos.

La tabla 24 muestra los operadores disponibles y su resultado al ser aplicados en expresiones.

Operador	**Resultado True**	**Resultado False**
< (Menor que)	Expresión1 < Expresión2	Expresión1 >= Expresión2
<= (Menor o igual que)	Expresión1 <= Expresión2	Expresión1 > Expresión2
> (Mayor que)	Expresión1 > Expresión2	Expresión1 <= Expresión2
>= (Mayor o igual que)	Expresión1 >= Expresión2	Expresión1 < Expresión2
= (Igual a)	Expresión1 = Expresión2	Expresión1 <> Expresión2
<> (Distinto de)	Expresión1 <> Expresión2	Expresión1 = Expresión2

Tabla 24. Operadores de comparación.

Vemos unos ejemplos en el código fuente 52.

```
lbResultado = "hola" < "otra cadena" ' Verdadero
lbResultado = 550 <= 300 ' Falso
lbResultado = 5 > 2 ' Verdadero
lbResultado = 87 >= 22 ' Verdadero
lbResultado = "larga" = "corta" ' Falso
lbResultado = 342 <> 65 ' Verdadero
‘-----------------------------------
‘ a continuación definimos una variable variant y otra numérica
‘ y hacemos dos pruebas de comparación
Public Sub Main()
Dim lbResultado As Boolean
Dim lvntNumero As Variant
Dim lnNum As Integer
‘ si el Variant se puede convertir a numérico...
lvntNumero = "520"
```

```
lnNum = 2380
lbResultado = lnNum > lvntNumero ‘ Verdadero
‘ si el Variant no se puede convertir a numérico...
lvntNumero = "hola"
lnNum = 2380
lbResultado = lnNum > lvntNumero ‘ aquí se produce un error
End Sub
```

Código fuente 52

El modo de comparación entre valores que no son de tipo Variant son los siguientes:

- Se realiza una comparación numérica si las dos expresiones son de tipo numérico; Byte, Integer, Double, etc., o una es un Variant con valor numérico
- Se realiza una comparación de cadenas si las dos expresiones son String, o una es un Variant.
- Se produce un error de tipos cuando una expresión es numérica y la otra es Variant de cadena de caracteres que no puede convertirse a número.
- Con una expresión numérica y otra Empty, se realiza una comparación numérica dando a Empty el valor 0.
- Con un String y un Empty, se realiza una comparación de cadenas dando a Empty el valor de cadena de longitud cero.
- Si ambas expresiones son Variant, dependerá del tipo interno de la expresión, quedando las comparaciones de la siguiente manera:
- Hay comparación numérica si ambas expresiones son numéricas.
- Hay comparación de cadenas si ambas expresiones son cadenas de caracteres.
- Cuando una expresión es numérica y la otra una cadena, la expresión numérica es menor que la de cadena.
- Si una expresión es Empty y la otra numérica, se realiza comparación numérica dando a Empty el valor 0.
- Si una expresión es Empty y la otra una cadena, se realiza comparación de cadenas dando a Empty el valor de cadena de longitud cero.
- Cuando las dos expresiones son Empty el resultado es igual.
- En comparaciones numéricas, entre un Single y un Double, se redondea a Single. Entre un Currency y un Single o Double, se convierte el último a Currency. Al comparar un Decimal con un Single o Double, el último se convierte a Decimal.

Hasta aquí hemos visto los operadores de comparación habituales, existen además dos operadores adicionales de comparación que poseen algunas variantes que veremos a continuación.

- **Is.** Compara dos variables que contienen un objeto, si las dos hacen referencia al mismo objeto se devuelve True, en caso contrario se devuelve False.

Sintaxis:

```
bResultado = oObjetoA Is oObjetoB
```

Vemos un ejemplo en el código fuente 53.

```
Public Sub Main()
‘ declaramos variables para contener objetos Form
Dim frmComenzar As frmSaludo
Dim frmIniciar As frmSaludo
Dim frmOtra As frmDistinto
Dim lbContenido As Boolean
‘ creamos un objeto frmSaludo y lo asignamos a una variable
Set frmComenzar = New frmSaludo
‘ asignamos el contenido de frmComenzar a otra variable de objeto
Set frmIniciar = frmComenzar
‘ creamos un objeto frmDistinto y lo asignamos a otra variable
Set frmOtra = New frmDistinto
‘ la siguiente línea resulta Verdadero ya que las dos
‘ variables contienen una referencia a un mismo objeto
lbContenido = frmComenzar Is frmIniciar
‘ la siguiente línea resulta Falso ya que las variables
‘ contienen referencias a objetos distintos
lbContenido = frmComenzar Is frmOtra
End Sub
```

Código fuente 53

- **Like.** Compara dos cadenas de caracteres en base a un patrón. Si la cadena en la que se efectúa la búsqueda coincide con el patrón se devuelve True, en caso contrario se devuelve False.

  Sintaxis:

  ```
  bResultado = cCadenaBuscar Like cPatron
  ```

  Los resultados de este operador dependen en gran medida de la instrucción Option Compare vista anteriormente. El patrón de búsqueda puede estar compuesto por caracteres normales y los caracteres especiales de la tabla 25.

  Si queremos realizar una búsqueda de los propios caracteres especiales, hay que incluirlos dentro de corchetes. Para buscar un carácter de corchete de cierre es necesario incluirlo sólo en el patrón. Podemos establecer un rango de búsqueda en la lista de caracteres, incluyendo un guión "-" entre los valores inferior y superior; este rango ha de incluirse en orden ascendente, de menor a mayor.

  Para buscar el propio carácter de exclamación "!", no hay que incluirlo entre corchetes. Para buscar el carácter de guión "-", se debe poner al principio o final de la lista de caracteres.

Carácter del patrón	Coincidencia en cadena a buscar
?	Cualquier carácter

*	Ninguno o varios caracteres
#	Un dígito (0-9)
[ListaCaracteres]	Cualquier carácter de la lista
[!ListaCaracteres]	Cualquier carácter que no esté en la lista

Tabla 25. Caracteres usados en el patrón de Like.

Vemos unos ejemplos en el código fuente 54.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = "B" Like "[A-M]" ‘ Verdadero
lbResultado = "D" Like "[!A-M]" ‘ Falso
lbResultado = "Z" Like "[!A-M]" ‘ Verdadero
lbResultado = "hola amigos" Like "h*s" ‘ Verdadero
lbResultado = "Es25s" Like "Es##s" ‘ Verdadero
lbResultado = "Elefante" Like "Elef?nte" ‘ Verdadero
lbResultado = "Ella tiene 2 hojas" Like "*# [f-i]*" ‘ Verdadero
lbResultado = "SUPERIOR" Like "S?E*" ‘ Falso
End Sub
```

Código fuente 54

Concatenación

Los operadores "&" y "+" se utilizan para unir varias cadenas de caracteres en una sola, siendo conveniente el uso de "&" para evitar ambigüedades a la hora de revisar el código.

Sintaxis:

```
cCadenaUnida = cCadenaA & cCadenaB
```

Si una de las dos cadenas no es un tipo String, se hace una conversión a Variant con valor

String dando también como resultado Variant con String. Si alguna de las expresiones es Null o Empty se considera cadena de longitud cero. En el código fuente 55 podemos ver algunas de las posibilidades de concatenación de valores

```
Public Sub Main()
Dim lcResultado As String
Dim lnNum As Integer
Dim lvntCambia As Variant
‘ vamos a concatenar cadenas sencillas
‘ atención al espacio en blanco entre cadenas,
‘ es importante tenerlo en cuenta
lcResultado = "hola" & "amigos" ‘ "holaamigos"
lcResultado = "hola " & "amigos" ‘ "hola amigos"
‘ vamos a concatenar una cadena y un número
lnNum = 457
lcResultado = "el número " & lnNum ‘ "el número 457"
```

```
‘ el contenido de lvntCambia es Empty
lcResultado = "el valor " & lvntCambia ‘ "el valor "
lvntCambia = Null ‘ asignamos Null a lvntCambia
lcResultado = "el valor " & lvntCambia ‘ "el valor "
End Sub
```

Código fuente 55

Lógicos

Efectúan operaciones lógicas entre dos expresiones devolviendo un valor Boolean, y a nivel de bit retornando un numérico.

And. Realiza una conjunción entre dos expresiones. Sintaxis:

```
XResultado = xExpresionY And xExpresionZ
```

La tabla 26 muestra los valores que pueden tomar las expresiones y el resultante de la operación.

ExpresiónY	ExpresiónZ	Resultado
True	True	True
True	False	False
True	Null	Null
False	True	False
False	False	False
False	Null	False
Null	True	Null
Null	False	False
Null	Null	Null

Tabla 26 Valores resultantes del operador And.

Vemos un ejemplo en el código fuente 56.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = 2 > 1 And "hola" = "hola" ' True
lbResultado = 2 = 1 And "hola" = "hola" ' False
lbResultado = 2 < 1 And "hola" > "hola" ' False
End Sub
```

Código fuente 56.

En comparaciones a nivel de bit, And devuelve los siguientes resultados:

Bit de Expresión Y	Bit de Expresión Z	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 27. Valores resultantes del operador And a nivel de bit.

- **Eqv.** Efectúa una equivalencia entre dos expresiones.

  Sintaxis:

  ```
  xResultado = xExpresionY Eqv xExpresionZ
  ```

- La tabla 28 muestra los valores que pueden tomar las expresiones y el resultante de la operación. En el código fuente 57 podemos observar un ejemplo.

ExpresiónY	ExpresiónZ	Resultado
True	True	True
True	False	False
False	True	False
False	False	True

Tabla 28. Valores resultantes del operador Eqv.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = 75 > 21 Eqv 63 > 59 ' True
lbResultado = 21 > 75 Eqv 63 > 59 ' False
End Sub
```

Código fuente 57

En comparaciones a nivel de bit, Eqv devuelve los resultados que aparecen en la tabla 29.

Bit de ExpresiónY	Bit de ExpresiónZ	Resultado
0	0	1

0	1	0
1	0	0
1	1	1

Tabla 29. Valores resultantes del operador Eqv a nivel de bit.

- **Imp.** Realiza una implicación entre dos expresiones.

  Sintaxis:

  ```
  xResultado = xExpresionY Imp xExpresionZ
  ```

  La tabla 30 muestra los valores que pueden tomar las expresiones y el resultante de la operación y en el código fuente 58 podemos ver un ejemplo.

ExpresiónY	**ExpresiónZ**	**Resultado**
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True
False	Null	True
Null	True	True
Null	False	Null
Null	Null	Null

Tabla 30. Valores resultantes del operador Imp.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = 50 > 30 Imp 64 > 56 ' True
lbResultado = 22 > 88 Imp 42 > 90 ' True
lbResultado = 31 > 20 Imp 26 > 85 ' False
End Sub
```

Código fuente 58

En comparaciones a nivel de bit, Imp devuelve los resultados que aparecen en la tabla 31:

Bit de ExpresiónY	**Bit de ExpresiónZ**	**Resultado**
0	0	1

0	1	1
1	0	0
1	1	1

Tabla 31. Valores resultantes del operador Imp a nivel de bit.

- **Not.** Realiza una negación sobre una expresión.

  Sintaxis:

  ```
  xResultado = Not xExpresion
  ```

  La tabla 32 muestra los valores de la expresión y el resultante de la operación:

Expresión	**Resultado**
False	True
Null	Null
True	False

Tabla 32. Valores resultantes del operador Not.

Podemos ver en el código fuente 59 unos ejemplos.

```
Public Sub Main()
Dim lbResultado As Boolean
Dim lbValorOperacion As Boolean
' evaluar una expresión con resultado verdadero
lbValorOperacion = 35 < 70 ' True
' negar el resultado de la anterior operación,
' con lo que verdadero pasará a falso
lbResultado = Not lbValorOperacion ' False
' evaluar una expresión con resultado falso
lbValorOperacion = 35 > 70 ' False
' negar el resultado de la anterior operación,
' con lo que falso pasará a verdadero
lbResultado = Not lbValorOperacion ' True
End Sub
```

Código fuente 59

En comparaciones a nivel de bit, Not devuelve los resultados que aparecen en la tabla 33.

Bit de Expresión	**Resultado**
0	1

1	0

Tabla 33. Valores resultantes del operador Not a nivel de bit.

- **Or.** Realiza una disyunción entre dos expresiones.

  Sintaxis:

  ```
  xResultado = xExpresionY Or xExpresionZ
  ```

  La tabla 34 muestra los valores que pueden tomar las expresiones y el resultante de la operación. En el código fuente 60 podemos ver un ejemplo.

ExpresiónY	**ExpresiónZ**	**Resultado**
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

Tabla 34. Valores resultantes del operador Or.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = 21 = 21 Or 65 < 12 ' True
lbResultado = 21 <> 21 Or 65 < 12 ' False
lbResultado = 21 > 21 Or 65 > 12 ' True
End Sub
```

Código fuente 60

En comparaciones a nivel de bit, Or devuelve los resultados que vemos en la tabla 35.

Bit de ExpresiónY	**Bit de ExpresiónZ**	**Resultado**
0	0	0

0	1	1
1	0	1
1	1	1

Tabla 35. Valores resultantes del operador Or a nivel de bit.

Xor. Realiza una exclusión de dos expresiones. Cuando una de las expresiones es Null, el resultado es Null. Para el resto de casos el resultado se ajusta a la tabla 36. En el código fuente 61, vemos unos ejemplos. Sintaxis:

```
xResultado = xExpresionY Xor xExpresionZ
```

ExpresiónY	**ExpresiónZ**	**Resultado**
True	True	False
True	False	True
False	True	True
False	False	False

Tabla 36. Valores resultantes del operador Xor.

```
Public Sub Main()
Dim lbResultado As Boolean
lbResultado = 48 > 20 Xor 50 > 17 ' False
lbResultado = 48 > 20 Xor 50 < 17 ' True
lbResultado = 48 < 20 Xor 50 < 17 ' False
End Sub
```

Código fuente 61

En comparaciones a nivel de bit, Xor devuelve los resultados que aparecen en la tabla 37.

Bit de ExpresiónY	**Bit de ExpresiónZ**	**Resultado**
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 37. Valores resultantes del operador Xor a nivel de bit.

Prioridad de operadores

En una línea de código en la que existan varias operaciones, estas se realizarán en un orden determinado conocido como *prioridad de operadores*. Para poder alterar este orden, deberemos utilizar paréntesis, agrupando dentro las operaciones que queremos evaluar en primer lugar. Dentro de los paréntesis también se mantendrá la prioridad entre los operadores.

Aparte de la prioridad que se mantiene dentro de un mismo grupo de operadores, existe otra prioridad entre diferentes categorías de operadores, de tal forma que en una expresión que tenga

operadores de distinta categoría, se evaluarán en primer lugar los operadores aritméticos, seguirán los de comparación y finalmente los lógicos.

Los operadores de comparación mantienen igual prioridad, resolviéndose en orden de aparición, de izquierda a derecha. Los operadores aritméticos y lógicos se ajustan a la prioridad indicada en la tabla 38.

Aritméticos	**Lógicos**
Potenciación (^)	Not
Negación (–)	And
Multiplicación y división (*, /)	Or
División entera (\)	Xor
Resto de división (Mod)	Eqv
Suma y resta (+, –)	Imp
Concatenación (&)	

Tabla 38. Prioridad de operadores.

```
Public Sub Main()
Dim lnTotalOp As Integer
Dim lbResultado As Boolean
' prioridad aritmética
lnTotalOp = 5 * 2 + 3 '13
lnTotalOp = 5 * (2 + 3) ' 25
' prioridad aritmética y comparación
' en esta línea el resultado es verdadero ya que
' se efectúa en primer lugar la operación aritmética
' y después la comparación
lbResultado = 5 * 7 > 10
' en esta línea el resultado es falso porque
' el paréntesis obliga a realizar la comparación antes
' de la operación aritmética
lbResultado = 5 * (7 > 10)
End Sub
```

Código fuente 62

El código fuente 62 muestra algunos ejemplos de prioridad normal y alterada mediante el uso de paréntesis.

Arrays

Un array, también denominado matriz o tabla, es un conjunto de datos del mismo tipo, agrupados bajo un único nombre de variable. A cada uno de los elementos del array se accede mediante índices, que determinan la posición dentro del array. Si un array tiene un solo conjunto o lista de datos se le denomina *array unidimensional*; en el caso de que los elementos del array sean a su vez otro array se trata de un *array multidimensional*.

Declaración y asignación de valores

La declaración de un array es muy similar a la de una variable, sólo tendremos que indicar el número de elementos que contendrá el array, teniendo en cuenta que por defecto el cero se cuenta como primer elemento. Veamos su sintaxis.

```
Dim aNombreArray(nDimension) As Tipo
```

En lo que respecta a cuestiones de ámbito, se siguen las mismas normas que para las variables.

En el código fuente 63 vamos a declarar un array unidimensional, de cuatro elementos.

```
Dim aNombres(3) As String
```

Código fuente 63

La estructura de este array aparece en la figura 72.

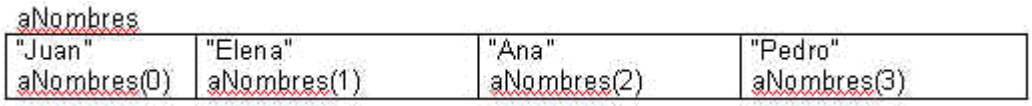


Figura 72. Estructura de un array simple.

Ahora en el código fuente 62 declararemos un array multidimensional, con tres elementos, los cuales a su vez, contendrán cada uno otro array de cuatro elementos.

```
Dim aNombresMult(2,3) As String
```

Código Fuente 64

La estructura de este array aparece en la figura 73. Al declarar un array, el valor inicial de sus elementos estará en función del tipo de dato con el que se haya declarado; por ejemplo, si es un array numérico, todos sus elementos serán cero, si es de tipo cadena serán cadenas de longitud cero, etc.

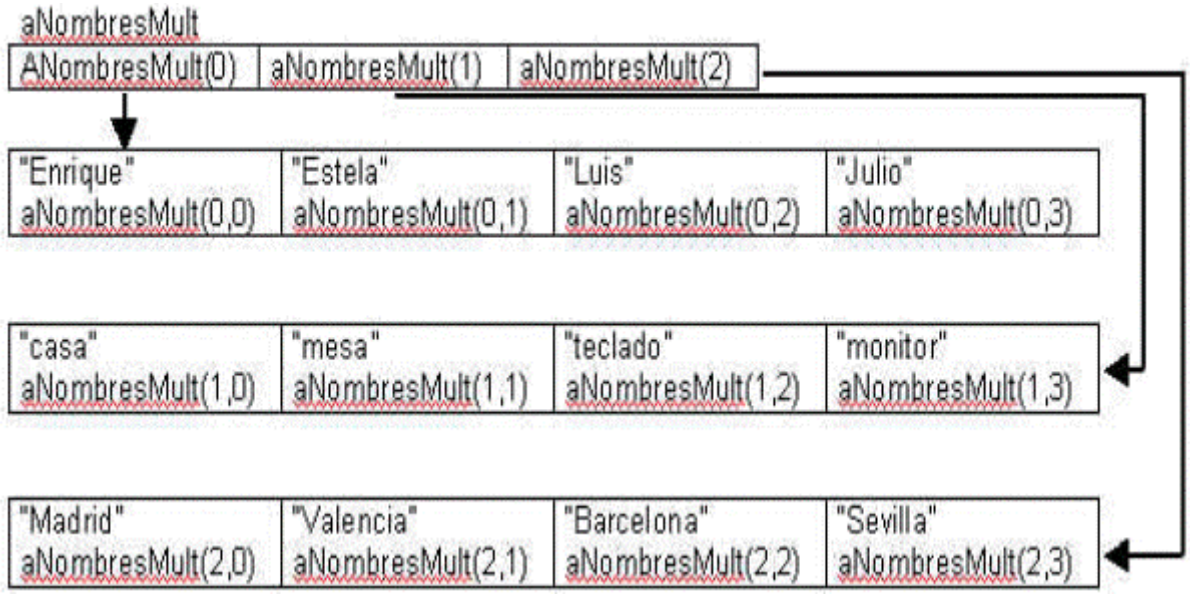


Figura 73. Estructura de un array multidimensional.

Para asignar valor a un elemento de un array usaremos el nombre del array, el índice en el que vamos a depositar un valor y el operador de asignación "=" como indica el código fuente 65.

```
aNombres(2) = "Juan"
```

Código fuente 65

En el código fuente 65 hemos introducido en el elemento dos del array *aNombres* una cadena de caracteres.

De forma similar, para recuperar el valor de un elemento de un array, utilizaremos el nombre del array y el número de elemento a recuperar, asignándolo por ejemplo a una variable.

```
lcUnNombre = aNombres(2)
```

Código fuente 66

El código fuente 66 asigna el contenido del elemento dos del array *aNombres* en la variable *lcUnNombre.*

Es posible igualmente, asignar el contenido de un array a otro, de la misma forma que se asignan valores a una variable. De esta manera, evitamos tener que recorrer elemento a elemento del array origen, para copiarlo en un array de destino.

En el código fuente 67, podemos ver un ejemplo de este tipo de operación.

```
Public Sub Main()
```

```
' declarar arrays
Dim lcNombres(1 To 3) As String
Dim lcValores() As String
' asignar valores a uno de los arrays
lcNombres(1) = "Ana"
lcNombres(2) = "Elisa"
lcNombres(3) = "Laura"
' copiar el array asignado al
' array vacío
lcValores = lcNombres
End Sub
```

Código fuente 67

Debemos tener en cuenta, que si el array destino al que vamos a asignar el contenido de otro array, es estático, la asignación no se producirá, generándose un error. Por este motivo, el array destino deberá ser siempre dinámico.

Devolución de arrays desde funciones

Una función puede devolver como valor de retorno un array, para ello, es necesario que la variable que recibe el resultado de la función, sea un array del mismo tipo de dato que el devuelto por la función. El código fuente 68, muestra un ejemplo de este tipo de casos.

```
Public Sub Main()
' declarar array
Dim lcDatos() As String
' llamar a la función,
' asignando el resultado al array
lcDatos = Recuperar
End Sub
' ------------------------------------
Public Function Recuperar() As String()
' declarar array
Dim lcValores(1 To 3) As String
' asignar valores a sus elementos
lcValores(1) = "Silla"
lcValores(2) = "Mesa"
lcValores(3) = "Puerta"
' devolver el array como
' resultado de la función
Recuperar = lcValores
End Function
```

Código fuente 68

Establecer los límites de los índices

Por defecto, el primer índice del array es cero, siendo posible cambiar esta configuración de dos maneras:

Utilizando la instrucción Option Base nIndiceInferior, en la que *nIndiceInferior* puede ser 0 o 1. Por defecto el índice inferior es 0, pero si nos sentimos más cómodos podemos utilizar el 1. Esta instrucción se usa en la zona de declaraciones del módulo y hemos de usarla en los módulos en los que deseemos variar el índice inferior por defecto para los arrays. El código fuente 69 muestra un ejemplo.

```
Option Base 1
' ---------------
Public Sub Main()
Dim aNombres(3) As String
aNombres(1) = "Julio" ‘ el primer índice del array es 1
aNombres(2) = "Enrique"
aNombres(3) = "Carmen"
End Sub
```

Código fuente 69

Un detalle importante a tener en cuenta, es que si declaramos un array con ámbito público en un módulo que tenga Option Base 1, y dicho array lo usamos en otro módulo en el que el límite inferior para los arrays sea 0, el array seguirá conservando 1 como límite inferior. Podemos comprobarlo en el código fuente 70.

```
‘ Module1
Option Explicit
‘ el límite inferior para los arrays
‘ declarados en este módulo será 1
Option Base 1
‘ declaramos el array en este módulo,
‘ pero será visible en todos los módulos
Public aPalabras(2) As String
’-------------------------
Public Sub Main()
VerArray
End Sub
’-------------------------
‘ Module2
Option Explicit
Public Sub VerArray()
‘ damos valor a los elementos del array
‘ empezando por el elemento 1,
‘ si intentáramos usar el 0 provocaríamos
‘ un error
aPalabras(1) = "mesa"
aPalabras(2) = "silla"
End Sub
```

Código fuente 70

Otro modo más flexible de indicar los límites para un array se basa en el uso de la cláusula **To** en el momento de declarar el array, como aparece en el código fuente 71.

```
Dim aNombreArray( nIndInferior To nIndSuperior ) As Tipo
```

Código fuente 71

De esta forma, podríamos ejecutar el código fuente 72.

```
Public Sub Main()
```

```
Dim aNombres(1 To 5) As String
aNombres(1) = "Elena"
aNombres(2) = "Juan"
aNombres(3) = "Ana"
aNombres(4) = "Enrique"
aNombres(5) = "Luis"
End Sub
```

Código fuente 72

Esta cláusula es muy versátil, ya que podemos utilizar un número determinado de elementos para un array, asignándoles el intervalo de índice que mejor convenga al contexto del procedimiento o módulo. Como ejemplo para este punto, en el código fuente 73 declaramos un array con cinco elementos, pero le asignamos el intervalo de índices del 18 al 22, teniendo siempre en cuenta que si intentamos utilizar un número de índice fuera de ese intervalo, se producirá un error.

```
Public Sub Main()
Dim aNombres(18 To 22) As String
aNombres(18) = "Elena"
aNombres(19) = "Juan"
aNombres(20) = "Ana"
aNombres(21) = "Enrique"
aNombres(22) = "Luis"
End Sub
```

Código fuente 73

Arrays dinámicos

Todos los arrays vistos hasta ahora son *estáticos*, es decir, una vez declarados, no se puede modificar su dimensión (el número de elementos que contienen), sin embargo, habrá ocasiones en que necesitemos incorporar nuevos elementos en tiempo de ejecución; aquí es donde entran a trabajar los *arrays dinámicos*, que podrán ser redimensionados para ampliar su número de elementos.

Los pasos para crear y manejar un array dinámico son los siguientes:

- Declarar el array sin dimensión.

```
Dim aNombreArray() As Tipo
```

Código fuente 74

- Cuando necesitemos añadir elementos al array, usaremos la sentencia ReDim.

```
ReDim aNombreArray(nNuevaDimen)
```

Código fuente 75

- Al utilizar ReDim, si el array contenía datos, estos se perderán. Para poder ampliar la dimensión de un array sin perder los datos que contuviera, hemos de usar la cláusula Preserve.

```
ReDim Preserve aNombreArray(nNuevaDimen)
```

Código fuente 76

Lógicamente, si la nueva dimensión es más pequeña que la existente, los datos de los índices superiores se perderán.

En código fuente 77, creamos un array dinámico y le añadimos nuevos elementos en tiempo de ejecución, visualizando su contenido mediante la llamada a otro procedimiento del módulo. En este ejemplo se utiliza el bucle *For...Next*, que veremos en el apartado dedicado a estructuras de control; igualmente usamos las funciones para manejo de arrays: LBound() y UBound(), que devuelven un número indicando el límite inferior y superior respectivamente del array pasado como parámetro. Estas funciones son útiles cuando no conocemos a priori los límites del array y necesitamos dimensionarlo durante la ejecución del programa.

```
Option Explicit
' declarar un array visible a nivel de módulo
Private aNombres() As String
' -------------------------------------
Public Sub Main()
Dim lnElementosCompletos As Integer
Dim lnInd As Integer
' dimensionamos el array con dos elementos: 0 y 1
ReDim aNombres(1)
' asignamos valor sólo al elemento 1
aNombres(1) = "casa"
VerArray ' vemos su contenido
' redimensionamos el array ampliando su
' número de elementos, pero perdemos
' el elemento 0
ReDim aNombres(1 To 3)
aNombres(2) = "Francia"
aNombres(3) = "Alemania"
' al visualizar el array, esperamos ver
' en el elemento 1 el valor "casa", pero
' esto no es así, ya que al redimensionar
' el array se pierden los valores
' anteriores
VerArray
' volvemos a redimensionar el array
' ampliando los elementos a 5,
' esta vez usamos Preserve para conservar
' los elementos que hubiera en el array
ReDim Preserve aNombres(1 To 5)
aNombres(1) = "Italia"
aNombres(4) = "España"
aNombres(5) = "Reino Unido"
VerArray
' guardamos el número de elementos de array
lnElementosCompletos = UBound(aNombres)
' redimensionamos el array con dos elementos más
ReDim Preserve aNombres(1 To UBound(aNombres) + 2)
' damos valor a los elementos nuevos,
' la variable lnElementosCompletos nos sirve
' para saber cuantos elementos del array
```

```
' tienen valor, y dar valor sólo a los
' elementos que no lo tienen
For lnInd = lnElementosCompletos + 1 To UBound(aNombres)
    aNombres(lnInd) = "Elemento nuevo " & CStr(lnInd)
Next
VerArray
' redimensionar el array disminuyendo
' el número de elementos
ReDim Preserve aNombres(1 To 3)
VerArray
End Sub
' ------------------------------------
Public Sub VerArray()
' visualizar el contenido del array
Dim lnInd As Integer
' usando este bucle, recorremos el array desde
' el primer elemento hasta el último,
' la variable lnInd guarda el índice
' del elemento a consultar
For lnInd = LBound(aNombres) To UBound(aNombres)
    MsgBox "Elemento " & lnInd & " del array" & vbCrLf & _
    "Contenido: " & aNombres(lnInd)
Next
End Sub
```

Código fuente 77

Para borrar el contenido de un array utilizaremos la instrucción *Erase* pasándole como parámetro el nombre del array a borrar, veámoslo en el código fuente 78.

```
Erase laNombres
```

Código fuente 78

Si el array es de tipo estático, inicializará todos sus valores a cero, cadena de longitud cero, Empty, Nothing, etc., en función del tipo de dato del array.

En caso de que el array sea de tipo dinámico, se eliminará completamente el espacio ocupado por el array en memoria, debiendo utilizar ReDim para definir de nuevo su dimensión antes de intentar introducir valores en él.

Manipulación de arrays multidimensionales

Veamos a continuación, un ejemplo con arrays multidimensionales. Dijimos que en este tipo de arrays, cada elemento a su vez es otro array, por este motivo la forma de acceder a los elementos varía ligeramente del acceso a arrays de una dimensión. En el código fuente 79 se declara un array con dos dimensiones, se da valor a sus elementos y se visualiza el contenido.

```
Public Sub Main()
Dim lnInd As Integer
Dim lnInd2 As Integer

' declaramos un array con dos dimensiones,
' cada dimensión tendrá tres elementos
```

```
Dim aNomMulti(1 To 2, 1 To 3) As String
' damos valor a los elementos:
' en la primera dimensión
' los elementos tendrán nombres
' de objetos comunes
aNomMulti(1, 1) = "mesa"
aNomMulti(1, 2) = "casa"
aNomMulti(1, 3) = "silla"

' en la segunda dimensión
' los elementos tendrán nombres
' de elementos urbanos

aNomMulti(2, 1) = "calle"
aNomMulti(2, 2) = "avenida"
aNomMulti(2, 3) = "plaza"

' utilizamos dos bucles For...Next para
' visualizar los elementos de cada dimensión
' con el primer bucle nos situamos
' en cada dimensión

For lnInd = 1 To 2
    MsgBox "Vamos a ver el contenido de la dimensión " & lnInd
    ' con en el segundo bucle
    ' vemos el contenido de cada
    ' elemento de la dimensión
    For lnInd2 = 1 To 3
        MsgBox "Dimensión " & lnInd & " elemento " & lnInd2 & vbCrLf & _
        "Contenido " & aNomMulti(lnInd, lnInd2)
    Next
Next
End Sub
```

Código fuente 79

Seleccionar información en un array

Si disponemos de un array en el que sólo queremos utilizar determinados valores, la función Filter() nos permite especificar un criterio de selección, por el cuál obtendremos un nuevo array con los elementos especificados en dicho criterio.

Sintaxis:

```
Filter(aBuscar,cCadenaBuscar[,bIncluir[,nComparar]])
```

- aBuscar. Array unidimensional en el que se va a filtrar la información. Si tiene más de una dimensión, o es nulo, se producirá un error.

- cCadenaBuscar. Cadena que se va a buscar.

- bIncluir. Valor lógico que indica que tipo de coincidencias se van a recuperar. Si el valor es True, se devolverán los elementos coincidentes con cCadenaBuscar; si es False, se obtendrán los elementos no coincidentes.

- nComparar. Constante que indica el tipo de comparación a efectuar (ver valores en función StrComp()).

En el caso de que no existan valores que cumplan los requisitos, esta función devolverá un array vacío. En el código fuente 80 podemos ver un ejemplo del uso de esta función. El resultado de la ejecución de Filter(), será un nuevo array con los siguientes valores: papel, pincel y tonel.

```
Public Sub Main()
' declarar arrays
Dim lcDatos(1 To 5) As String
Dim lcResultado() As String
' asignar valores al array principal
lcDatos(1) = "papel"
lcDatos(2) = "pincel"
lcDatos(3) = "lienzo"
lcDatos(4) = "tonel"
lcDatos(5) = "techo"
' filtrar los elementos del array
' obteniendo un nuevo array
lcResultado = Filter(lcDatos, "el")
End Sub
```

Código fuente 80

Tipos definidos por el usuario

Un tipo definido por el usuario es un nuevo tipo de dato que puede estar formado por uno o más elementos de los tipos ya existentes, incluyendo arrays e incluso otros tipos de usuario. Una vez definido un nuevo tipo, se pueden declarar variables de ese tipo en el código de la aplicación. En cuanto al ámbito de las variables de un tipo de usuario, si el tipo está definido como Private, se podrán declarar variables de ese tipo sólo en el módulo que contiene la definición del tipo; mientras que si el tipo está definido como Public, se podrán declarar variables de dicho tipo desde cualquier módulo de la aplicación, esto último es aplicable en módulos estándar, ya que en otro tipo de módulos: formulario, clase, etc., sólo pueden declararse tipos como Private.

Sintaxis:

```
Type NombreTipo
    NombreElemento As Tipo
    .....
    .....
End Type
```

Veamos, en el código fuente 81, un sencillo ejemplo de creación de tipo definido por el usuario, que iremos ampliando posteriormente. Creamos un nuevo tipo de usuario, y en el procedimiento Main() declaramos una variable de ese tipo, asignando valor a sus elementos.

```
Option Explicit
Private Type Empleado
    Nombre As String
    Telefono As String
    FNacim As Date
End Type

' ----------------

Public Sub Main()
```

```
' declarar variable del nuevo tipo
Dim xNuevoEmp As Empleado
' asignar valores a los elementos de
' la variable tipo
xNuevoEmp.Nombre = "Juan García"
xNuevoEmp.Telefono = "555.33.44"
xNuevoEmp.FNacim = #9/20/69#
End Sub
```

Código fuente 81

Ahora creamos un nuevo tipo que sirva para almacenar direcciones y en el tipo Empleado definimos un nuevo elemento que sea de ese tipo. En el código fuente 82, la variable declarada como Empleado guardará ahora la dirección.

```
Option Explicit
Private Type Direccion
Calle As String
NumPortal As Integer
Poblacion As String
End Type
' ----------------
Private Type Empleado
Nombre As String
' el siguiente elemento del tipo
' es a su vez otro tipo de usuario
Direcc As Direccion
Telefono As String
FNacim As Date
End Type
' ----------------
Public Sub Main()
' declarar variable del nuevo tipo
Dim xNuevoEmp As Empleado
' asignar valores a los elementos de
' la variable tipo
xNuevoEmp.Nombre = "Juan García"
xNuevoEmp.Telefono = "555.33.44"

xNuevoEmp.FNacim = #9/20/69#
' las siguientes líneas asignan valor
' a los elementos del tipo incluido
' a su vez dentro de otro elemento
xNuevoEmp.Direcc.Calle = "Olmos"
xNuevoEmp.Direcc.NumPortal = 21
xNuevoEmp.Direcc.Poblacion = "Madrid"
End Sub
```

Código fuente 82

Y para finalizar con este apartado, vamos a realizar un cambio en uno de los elementos del tipo Empleado. Supongamos que queremos almacenar varios teléfonos, para ello cambiamos el elemento Teléfono por un array de tres elementos en el que poder incluir más información. En el código fuente 83 vemos sólo las líneas de código afectadas por esta modificación.

```
Private Type Empleado
.......
```

```
.......
' cambiamos el siguiente elemento
' por un array
Telefono(1 To 3) As String
.......
.......
End Type
' ----------------
Public Sub Main()
.......
.......
' asignar valores a los elementos de
' la variable tipo
xNuevoEmp.Nombre = "Juan García"
' guardar los números de teléfono en
' el elemento que contiene un array
xNuevoEmp.Telefono(1) = "555.33.44"
xNuevoEmp.Telefono(2) = "475.11.22"
xNuevoEmp.Telefono(3) = "551.23.44"

xNuevoEmp.FNacim = #9/20/69#
.......
.......
End Sub
```

Código fuente 83

Procedimientos

El código en VB se agrupa en unidades conocidas como procedimientos, que contienen una serie de variables, expresiones, estructuras, etc., que realizan una tarea determinada. Pueden recibir parámetros y llamar a su vez a otros procedimientos, pero no pueden anidarse. En este apartado vamos a ver los procedimientos Sub y Function, dejando los Property para el capítulo dedicado a la programación orientada a objetos.

Para incluir el código de un procedimiento en un módulo, podemos escribir su declaración y cuerpo directamente en el módulo o seleccionar la opción *Herramientas+Agregar procedimiento* del menú de VB, que abrirá una ventana de diálogo para introducir el tipo de procedimiento, una vez aceptada la ventana, creará el procedimiento vacío y nos situará en él.

Sub

Sintaxis:

```
[Private|Public][Static] Sub NombreProc([Parámetros])
código
[Exit Sub]
código
End Sub
```

- Sub...End Sub. Definen el comienzo y final respectivamente del procedimiento.
- Private. Un procedimiento declarado con Private sólo es visible en el módulo donde se haya declarado.
- Public. Un procedimiento declarado con Public es visible en todos los módulos del proyecto.

- Static. Convierte a Static todas las variables locales del procedimiento.
- NombreProc. Nombre que se utiliza para conocer al procedimiento dentro del código.
- Parámetros. Variable o lista de ellas separadas por comas, con información que recibe el procedimiento para utilizarla dentro de su código. Cada variable pasada como parámetro tiene la siguiente sintaxis:

  ```
  [Optional] [ByVal | ByRef] [ParamArray] NombreVar [( )] [As
  Tipo] [= ValorPredet]
  ```

  - Optional. Indica que no es necesario que el parámetro contenga valor; una vez usado, el resto de parámetros también deberán ser opcionales.
  - ByVal. La variable se pasa por valor, esto quiere decir que se pasa una copia del valor de la variable, con lo cual si se cambia el valor del parámetro, no se cambia el valor de la variable original.
  - ByRef. Predeterminado. La variable se pasa por referencia, esto quiere decir que se pasa la dirección de la variable, con lo que puede ser cambiada desde el procedimiento.
  - ParamArray. Se utiliza como último parámetro de la lista, e indica que ese parámetro es un array de elementos Variant, de forma que podemos pasar un número variable de parámetros al procedimiento. Esta opción no se puede usar junto con las tres anteriores.
  - NombreVar. Nombre que daremos al parámetro que recibe el valor.
  - Tipo. Tipo de datos del parámetro. El parámetro recibido puede ser un array.
  - ValorPredet. Si el parámetro es Optional, podemos indicar un valor por defecto para el caso de que no se pase ningún valor al parámetro.

Si durante la ejecución de un procedimiento Sub, se encuentra la sentencia Exit Sub, el curso del programa será desviado fuera de ese procedimiento hasta el procedimiento que hizo la llamada.

Veamos el código fuente 84 con un ejemplo de uso de un procedimiento y de paso de parámetros.

```
Public Sub Main()
' declaramos variables y le asignamos
' valor para pasarlas al procedimiento
Dim lcCadena As String
Dim lnDato As Integer
lcCadena = "prueba de título"
lnDato = 2352
' llamamos al procedimiento
' y le pasamos parámetros
Calcular lcCadena, lnDato
' esta variable ha cambiado en el procedimiento
MsgBox "Contenido de variable lcCadena: " & lcCadena
' esta variable conserva su valor
MsgBox "Contenido de variable lnDato: " & lnDato
End Sub

' ------------------------------------------------------------

Public Sub Calcular(rcTitulo As String, ByVal vnNum As Integer, _
```

```
   Optional rcPalabra As String = "HOLA")
' cambiamos el contenido de los parámetros
' este parámetro cambiará aquí y quedará cambiado
' para el procedimiento que llamó a este
rcTitulo = "Título en Calcular"
' este parámetro cambiará aquí, pero seguirá
' conservando su valor original en el
' procedimiento que llamó a este
vnNum = 9999
MsgBox "Estamos en Calcular() / valor de variable rcTitulo: " & rcTitulo
MsgBox "Estamos en Calcular() / valor de variable vbNum: " & vnNum
' como no se ha pasado valor al parámetro rcPalabra,
' toma el valor por defecto
MsgBox "Estamos en Calcular() / valor de variable rcPalabra: " & rcPalabra
End Sub
```

Código fuente 84

Function

Sintaxis:

```
[Public|Private][Static] Function NombreFunc([Parámetros])[As Tipo]
código
[Exit Function]
código
NombreFunc = valor
End Function
```

Un procedimiento Function es básicamente igual que un Sub excepto en que con Function podemos devolver un valor a la línea de código que hizo la llamada a Function. Debido a que se puede retornar un valor, también es posible definir el tipo de dato del valor de retorno; esto se hace con la sentencia *As Tipo* situada después de los paréntesis que contienen los parámetros a pasar a la función.

Una vez dentro del cuerpo de la función, la manera de retornar un valor, es usar el mismo nombre de la función como si de una variable se tratara y asignarle el valor a devolver.

Si durante la ejecución de un procedimiento Function, se encuentra la sentencia Exit Function, el curso del programa será desviado fuera de ese procedimiento hasta el procedimiento que hizo la llamada.

El resto de elementos de la función no comentados aquí, pueden consultarse en el punto dedicado al procedimiento Sub, puesto que son equivalentes.

En elcódigo fuente 85 vemos un ejemplo de uso de procedimiento Function:

```
Public Sub Main()
' declaramos variables y le asignamos
' valor para pasarlas a la función
Dim llDato As Long
Dim llValorRet As Long
llDato = 5432
' llamamos a la función Calcular y
' recogemos el resultado en la
' variable llValorRet
llValorRet = Calcular(llDato)
MsgBox "El resultado de la función es: " & llValorRet
```

```
End Sub

' ------------------------------------------------
Public Function Calcular(rlValor As Long) As Long
' multiplica el número pasado como parámetro
' y devuelve su valor
' se utiliza el nombre de la función
' para depositar el valor a devolver
Calcular = rlValor * 5
End Function
```

Código fuente 85

Una de las cualidades de las funciones es que pueden formar parte de una expresión. Así en el código fuente 83, podemos evitar el uso de variables intermedias, utilizando directamente la función con MsgBox() para mostrar el resultado del procedimiento como se indica en el código fuente 86

```
MsgBox "El resultado de la función es: " & Calcular(llDato)
```

Código fuente 86

Con un procedimiento Sub esto no es posible, ya que no puede formar parte de una expresión. Utilizando Sub, deberíamos usar una variable pasándola al procedimiento por referencia, de forma que recupere el valor que queremos, al finalizar el Sub, usaremos esa variable para mostrar el resultado, de forma similar al código fuente 87.

```
' llamamos al procedimiento Calcu
' la variable llDato es pasada por
' referencia de forma predeterminada

Calcu llDato

' al salir del procedimiento, la variable
' llDato tiene el valor ya calculado

MsgBox "El resultado del procedimiento es: " & llDato
```

Código fuente 87

Sub Main(). Un procedimiento especial

Toda aplicación ha de tener un punto de entrada de la ejecución; en el caso de VB ese punto de entrada puede ser bien uno de los formularios del programa o un procedimiento que tenga el nombre Main(), que será el primero que se ejecute.

Para configurar el punto de inicio de la aplicación, debemos abrir la ventana de propiedades del proyecto en la opción de menú de VB *Proyecto+Propiedades de <proyecto>*, en la ficha *General* de esta ventana hay una lista desplegable llamada *Objeto inicial* en la que podremos seleccionar este aspecto.

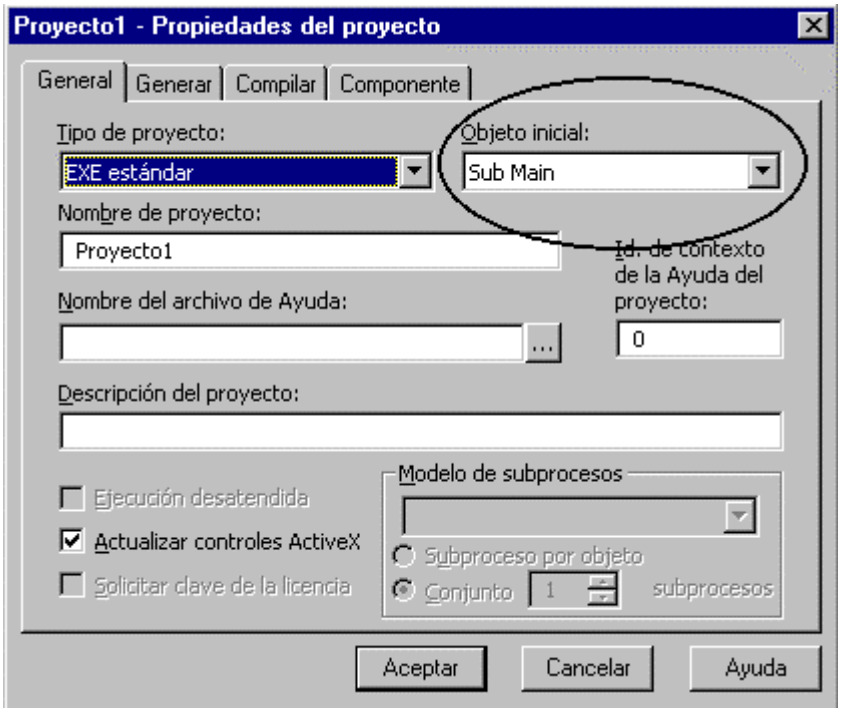


Figura 74. Cuadro de propiedades del proyecto. Configuración del punto de entrada a la aplicación.

Estructuras de control

Una estructura de control es un componente del lenguaje que se utiliza para dirigir el flujo de la aplicación en determinada dirección, en función del resultado obtenido al evaluar una expresión, de modo que unas líneas de código sean ejecutadas y otras no. También se utilizan para ejecutar un mismo bloque de código un número repetido de veces, en este caso se les denomina *bucles*.

Las estructuras de control pueden anidarse, es decir, que dentro del cuerpo de una estructura podemos situar otra estructura igual o diferente. En los sucesivos ejemplos veremos esta característica.

If...Then...Else...End If

Mediante esta estructura podemos seleccionar el código a ejecutar en función de que una condición a evaluar sea verdadera o falsa.

Sintaxis:

```
If condición Then
    [código-if]
[ElseIf condición Then]
    [código-elseif]
[Else]
    [código-else]
End If
```

En primer lugar se comprueba el valor de la condición If, si es True se ejecuta el bloque de código asociado; en el caso de que la condición sea False, se comprueba si existen líneas ElseIf, que pueden ser una o más, para ver si alguna de sus condiciones es True. Si ningún ElseIf es verdadero, se ejecuta

el código asociado a Else si existe, pasando por último a la primera línea después de End If, que es la que cierra esta estructura.

En el momento en que se ejecuta el código de una condición que sea verdadera, el control del programa pasa, después de ejecutar ese código a la primera línea después de End If, ignorando las siguientes condiciones que existan.

Veamos, en el código fuente 88, un ejemplo demostrativo del uso de esta estructura de control:

```
Public Sub Main()
Dim lnNum As Integer
Dim lcNombre As String
' asignamos valor a las variables
lnNum = 125
lcNombre = "Pedro"
' la condición de la siguiente línea
' devuelve False, por lo que pasamos
' a la siguiente
If lnNum = 11 Then
   MsgBox "Valor de la variable lnNum: " & lnNum
' esta línea da True, por lo que
' se ejecuta su código asociado
ElseIf lcNombre = "Pedro" Then
  ' después de ejecutar esta línea
  ' pasaremos a End If
   MsgBox "Valor de la variable lcNombre: " & lcNombre
' si no hubiera existido ningún valor
' verdadero, se habría ejecutado la
' siguiente línea
Else
   MsgBox "No hay coincidencias"
End If
End Sub
```

Código fuente 88

En el código fuente 89, vamos a variar un poco el código fuente 88 para incluir un If anidado.

```
Public Sub Main()
Dim lnNum As Integer
Dim lnOtroNum As Integer
Dim lcNombre As String
lnNum = 11
lnOtroNum = 2000
lcNombre = "Pedro"
' evaluamos la primera condición
' que devuelve True, por lo que
' ejecutamos su bloque de código
If lnNum = 11 Then
   ' en este bloque de código
   ' hay un If anidado, que al
   ' comprobarse y ser True se ejecuta
   If lnOtroNum = 2000 Then
      MsgBox "Estamos en un If...End If anidado"
   End If
ElseIf lcNombre = "Pedro" Then
   MsgBox "Valor de la variable lcNombre: " & lcNombre
Else
   MsgBox "No hay coincidencias"
```

```
End If
End Sub
```

Código fuente 89

También es posible utilizar If en una sola línea de código, vemos su sintaxis:

```
If condición Then [código-if] [Else] [código-else]
```

Aunque es interesante conocer esta forma y usarla en expresiones sencillas, por razones de legibilidad del código no es muy conveniente su uso, quedando mucho más clara la forma de estructura explicada al comienzo de este apartado. Veamos en el código fuente 90 un ejemplo.

```
Public Sub Main()
Dim lnNum As Integer
Dim lcNombre As String
lnNum = 11
lcNombre = "Pedro"
If lnNum = 121 Then lcNombre = "mesa" Else lcNombre = "silla"
End Sub
```

Código fuente 90

Select Case...End Select

Ejecuta un bloque de código entre varios disponibles, en función del resultado de una expresión.

Sintaxis:

```
Select Case expresión
[Case ListaExpresiones]
    [código-case]
[Case Else]
    [código-else]
End Select
```

Una vez evaluada la expresión, se buscará entre uno o más de los Case del cuerpo de la estructura, una coincidencia en su lista de expresiones con la expresión del Select. Las expresiones de la lista estarán separadas por comas, pudiendo situarse de la siguiente manera:

- expresión.
- expresión To expresión (el primer valor ha de ser el menor del intervalo).
- Is OperadorComparación expresión.

Si hay varias cláusulas Case que cumplen con la condición a buscar, se ejecutará el bloque de código del primer Case que cumpla la condición.

Si no hay ningún Case que cumpla la condición, se ejecutará el código de Case Else si existe.

Una vez que se ha ejecutado o no el código asociado a alguna de las cláusulas, el control del programa pasará a la primera línea después de End Select, que es la que cierra esta estructura.

```
Public Sub Main()
' inicializar variable y dar valor
Dim lnNum As Integer
lnNum = 1250
' evaluar el valor de la variable y
' buscar algún Case que coincida
Select Case lnNum
Case 2000, 1250, 800
   ' aquí se cumple la condición en el segundo
   ' elemento de la lista, por lo que se
   ' ejecuta la siguiente línea
   MsgBox "Estoy en Case de varios valores"
Case Is < "1000"
   ' este Case no se cumple, por aquí
   ' no pasaría de ninguna manera
   MsgBox "Estoy Case < '1000' "
Case 500 To 1300
   ' aquí también se cumpliría la condición
   ' por el intervalo de valores, pero no
   ' se ejecuta este código porque ya hay un
   ' Case anterior que se cumple, por lo cual
   ' no pasa por aquí
   MsgBox "Estoy en Case 500 To 1300"
Case Else
   MsgBox "No se cumple ninguna condición"
End Select
End Sub
```

Código fuente 91

En el código fuente 91, se comprueba el valor de una variable mediante un Select Case, ejecutando un determinado código al encontrar el valor de la variable. En el código fuente 92 veremos una adaptación del anterior, pero usando Select Case anidados.

```
Public Sub Main()
' inicializar variables y dar valor
Dim lnNum As Integer
Dim lcNombre As String
lnNum = 1250
lcNombre = "Pedro"
' evaluar el valor de lnNum y
' buscar algún Case que coincida
Select Case lnNum
Case Is > 2000, Is < 1250, Is = 800
   ' aquí no se cumple la condición
   ' en ningún elemento de la lista
   MsgBox "Estoy en Case de varios valores"
Case Is > "1000"
   ' aquí si se cumple la condición,
   ' la cadena se convierte automáticamente
   ' a número para la comparación
   ' comenzamos un Select Case anidado
   Select Case lcNombre
   Case "Ana", "Juan"
      ' aquí no hay coincidencias
      MsgBox "Estoy en Case Ana, Juan"

   Case Is = "SILLA"
      ' aquí tampoco hay coincidencias
      MsgBox "Estoy en Case SILLA"
```

```
   Case Else
      ' al no haber ninguna coincidencia
      ' se ejecuta el código de esta opción
      MsgBox "No se ha cumplido nada para lcNombre"

   End Select
Case 500 To 1300
   ' aquí también se cumpliría la condición
   ' por el intervalo de valores, pero no
   ' se ejecuta este código porque ya hay un
   ' Case anterior que se cumple, por lo cual
   ' no pasa por aquí
   MsgBox "Estoy en Case 500 To 1300"
' en este Select Case no existe un Case Else
' no es obligatorio incluirlo
End Select
End Sub
```

Código fuente 92

For...Next

Esta estructura es de tipo bucle, ejecuta un mismo grupo de líneas repetidas veces.

Sintaxis:

```
For varContador = nPrincipio To nFin [Step nIncremento]
    [código]
    [Exit For]
    [código]
Next [varContador]
```

La variable varContador actúa como contador de las ejecuciones del bucle; se inicializa con el valor de nPrincipio, comenzando a ejecutar el código contenido en el cuerpo de la estructura. Al llegar a Next, si se ha incluido la cláusula Step, se sumará o restará a varContador el valor de nIncremento y se volverá a ejecutar el bloque de código sucesivamente hasta que varContador tenga el mismo valor que nFin o superior, momento en el cual, el programa pasará a la siguiente línea después de Next.

Si durante la ejecución del código contenido en For...Next, se encuentra una sentencia Exit For, el control del programa pasará inmediatamente a la línea siguiente a Next. Es posible poner varias sentencias Exit For dentro de un bucle For...Next, para permitir la salida del bucle en diversas situaciones.

El valor de nIncremento influye en la manera de procesar el contador. Si nIncremento es un número positivo o cero, varContador ha de ser menor o igual a nFin; si es un número negativo, varContador ha de ser mayor o igual que nFin; si no se especifica nIncremento, por defecto el bucle For...Next se incrementará en 1

En el código fuente 93 podemos ver varias formas de uso de esta estructura de control:

```
Public Sub Main()
Dim lnContador As Integer
Dim lnCont2 As Integer
Dim lnAcumula As Integer
Dim lcCadena As String
```

```
' bucle simple, recorremos el contador
' y acumulamos valores en una variable
lnAcumula = 50
For lnContador = 1 To 10
   lnAcumula = lnAcumula + 3
Next lnContador
' bucle que contiene otro bucle anidado,
' el de primer nivel, suma una cantidad y la
' incluye en una variable,
' el bucle anidado convierte a cadena
' el número y lo acumula a una
' variable de cadena,
' no se ha incluido la cláusula Step
' en estos bucles ni en el anterior,
' por lo que el incremento es en 1
lnAcumula = 20
lcCadena = ""
For lnContador = 1 To 10
   lnAcumula = lnAcumula + 5
   For lnCont2 = 1 To 3
      lcCadena = lcCadena & Str(lnAcumula)
   Next
Next
' en el siguiente bucle se hace un incremento
' del contador en dos, por lo que sólo se
' ejecuta cinco veces
lnAcumula = 12
For lnContador = 1 To 10 Step 2
   lnAcumula = lnAcumula * 2
Next
' en este bucle el contador se decrementa
' en cuatro, por lo que el valor inicial
' del contador ha de ser mayor que el final
lnAcumula = 56
For lnContador = 28 To 6 Step -4
   lnAcumula = (lnAcumula * 4) / 2
Next
' el siguiente bucle no llega a ejecutarse
' debido a que el valor inicial del contador
' es superior al final, y no se ha
' especificado una cláusula Step con
' un incremento en negativo
lnAcumula = 1
For lnContador = 10 To 6
   lnAcumula = lnAcumula + 20
Next
End Sub
```

Código Fuente 93

Do...Loop

Bucle que repite la ejecución de un bloque de código mientras una condición sea True o hasta que la condición sea True.

Sintaxis. Disponemos de dos formas de escribir este bucle:

```
Do [While|Until condición]
   [código]
   [Exit Do]
   [código]
Loop
```

O bien esta otra:

```
Do
    [código]
    [Exit Do]
    [código]
Loop [While|Until condición]
```

Cuando el flujo de la aplicación llega a esta estructura, se evalúa la condición, entrando a ejecutar el bloque de código en función de la partícula usada con Do:

Si es While y la condición es True, se ejecutará el bloque de código hasta que sea False o encuentre una sentencia Exit Do; en el caso de que sea False, el control del programa pasará a la primera línea después de Loop, que es la palabra que cierra este tipo de estructura.

Si es Until y la condición es False, se ejecutará el bloque de código hasta que sea True o encuentre una sentencia Exit Do; cuando sea True, el control del programa pasará a la primera línea después de Loop.

Utilizando más de una estructura Do...Loop anidada, si forzamos una salida con Exit Do desde uno de los bucles interiores, el control de la aplicación pasará al bucle de nivel superior al que estamos situados en ese momento. Es posible poner varias sentencias Exit Do dentro de un bucle Do.Loop, para permitir la salida del bucle en diversas situaciones.

En el código fuente 94, vemos algunas formas de crear este tipo de bucles.

```
Public Sub Main()
Dim lnNum As Integer
Dim lnInterior As Integer
Dim lnCuenta As Integer
Dim lnAcumula As Integer
' el siguiente es un sencillo bucle
' que se mantiene en ejecución hasta que
' dentro de su código no se cambia el
' valor de una variable
lnNum = 1
lnAcumula = 1
Do While lnNum = 1
   lnAcumula = lnAcumula + 1

   If lnAcumula = 20 Then
      lnNum = 2
   End If
Loop
' en este tipo de construcción el código
' se ejecuta al menos una vez, ya que la
' expresión de evaluación de la condición
' se realiza al final del bucle
lnNum = 2
Do
   lnAcumula = lnAcumula + 1
   If lnAcumula = 20 Then
      lnNum = 2
   End If
Loop While lnNum = 1
' con este Do...Loop utilizamos
' la cláusula Until, y no saldremos
' del bucle hasta que la condición
```

```
' no sea verdadera
lnNum = 1
lnAcumula = 0
Do Until lnNum = 2
   lnAcumula = lnAcumula + 1
      If lnAcumula = 20 Then
      lnNum = 2
   End If
   Loop
' aquí utilizamos bucles anidados
' de los que no saldremos hasta que
' las variables alcancen unos valores
' determinados. En el bucle más interior
' salimos usando Exit Do, y en el exterior
' no salimos hasta que no se cumple
' la condición del bucle
lnNum = 1
lnAcumula = 0
lnInterior = 1
lnCuenta = 1
Do While lnNum = 1
   lnInterior = lnInterior + 1
   Do While lnInterior = 10
      lnCuenta = lnCuenta + 1
            If lnCuenta = 5 Then
         Exit Do
      End If
   Loop
   If lnCuenta = 5 Then
      lnNum = 2
   End If
Loop
End Sub
```

Código fuente 94

En estos últimos ejemplos tenemos una clara muestra de combinación de diferentes tipos de estructuras, ya que dentro de los bucles Do...Loop se han incluido estructuras If...End If.

While...Wend

Bucle que ejecuta un grupo de líneas de código, mientras una condición sea True, puede anidarse y es similar a Do...Loop, pero no proporciona la flexibilidad de este último. Sintaxis:

```
While condición
    [código]
Wend
```

Veamos un ejemplo, en el código fuente 95.

```
Public Sub Main()
Dim lnNum As Integer
lnNum = 1
While lnNum < 100
    lnNum = lnNum + 1
Wend
End Sub
```

Código Fuente 95

For Each...Next

Ejecuta un bloque de código para cada elemento de un array o colección de objetos.

Sintaxis:

```
For Each vntElemento In ArrayCol
    [código]
    [Exit For]
    [código]
Next [vntElemento]
```

Al llegar a este bucle, el programa comprueba si hay al menos un elemento en ArrayCol; de ser así, introduce su valor en vntElemento, que debe ser una variable de tipo Variant, y ejecuta el bloque de código de la estructura; terminados estos pasos, pasa el siguiente elemento de ArrayCol a vntElemento, y vuelve a ejecutar el código, y así sucesivamente hasta procesar todo ArrayCol o hasta que encuentre una sentencia Exit For, lo que pasará el control del programa a la línea posterior a Next.

Veamos el código fuente 96.

```
Public Sub Main()
' declarar variables y llenar
' el array de valores
Dim laCiudades(1 To 5) As String
Dim lvntCiudad As Variant
laCiudades(1) = "París"
laCiudades(2) = "Londres"
laCiudades(3) = "Berlín"
laCiudades(4) = "Roma"
laCiudades(5) = "Bruselas"
' procesar el array, si encontramos
' un determinado elemento, salir del bucle
For Each lvntCiudad In laCiudades
   MsgBox "Ciudad en proceso: " & lvntCiudad
   If lvntCiudad = "Roma" Then
      Exit For
   End If
Next
End Sub
```

Código Fuente 96

En el código fuente 97, nos basamos en el código fuente 96, pero con algunas variaciones para usar bucles anidados.

```
Public Sub Main()
' declarar variables y llenar
' los arrays de valores
Dim laCiudades(1 To 5) As String
Dim laMonum(1 To 7) As String
Dim lvntCiudad As Variant
Dim lvntLetra As Variant
Dim lcMonumento As String
laCiudades(1) = "París"
laCiudades(2) = "Londres"
laCiudades(3) = "Berlín"
```

```
laCiudades(4) = "Roma"
laCiudades(5) = "Bruselas"
laMonum(1) = "C"
laMonum(2) = "o"
laMonum(3) = "l"
laMonum(4) = "i"
laMonum(5) = "s"
laMonum(6) = "e"
laMonum(7) = "o"
' procesar el array, si encontramos
' un determinado elemento, entrar en
' un For Each...Next anidado
For Each lvntCiudad In laCiudades
   MsgBox "Ciudad en proceso: " & lvntCiudad
   If lvntCiudad = "Roma" Then
      ' tomar cada elemento del array laMonum
      ' y agregarlo a la variable lcMonumento
      ' una vez procesado todo el array
      ' mostrar el contenido de lcMonumento
      For Each lvntLetra In laMonum
         lcMonumento = lcMonumento & lvntLetra
      Next

      MsgBox "Contenido del array laMonum " & lcMonumento

   End If
Next
End Sub
```

Código fuente 97

With...End With

Realiza una serie de acciones sobre un objeto o tipo definido por el usuario.

Sintaxis:

```
With oObjeto
    [código]
End With
```

Indicando el nombre de un objeto al comienzo de esta estructura, es posible hacer llamadas a los métodos y modificar las propiedades de dicho objeto, sin necesidad de utilizar el nombre del objeto dentro del cuerpo de la estructura, sólo tenemos que usar la notación: *.propiedad* o *.método()*.Vemos un ejemplo en el código fuente 98.

```
Public Sub Main()
' suponiendo que en la aplicación tenemos
' un formulario llamado frmPrueba,
' creamos un nuevo formulario de esta clase
' y lo asignamos a la variable frmNueva
Dim frmNueva As frmPrueba
Set frmNueva = New frmPrueba
' ahora con una estructura With
' modificamos propiedades del
' formulario y llamamos a su
' método Show
With frmNueva
   .Caption = "Título cambiado"
```

```
   .Height = 4100
   .Width = 3200
   .Left = 1000
   .Show
End With
End Sub
```

Código fuente 98

Goto

Desvía el flujo del programa a una etiqueta o sección de código identificada por un nombre y dos puntos ":".

El uso de GoTo y etiquetas de código en los procedimientos, salvo contadas excepciones como puede ser el tratamiento de errores, es muy desaconsejable debido a la poca estructuración del código y complejidad de mantenimiento que producen; por este motivo y sólo en casos muy determinados, desaconsejamos al lector el uso de este tipo de codificación, ya que por lo general, siempre existirá una forma más estructurada de realizar las tareas.

En el código fuente 99 y 100 vamos a ver dos fuentes que realizan la misma tarea, en uno se utiliza GoTo junto a una etiqueta, en el otro se hace una llamada a un procedimiento para realizar esa tarea.

Código fuente 99, versión con GoTo.

```
Public Sub Main()
' declaramos variables y les
' asignamos valor
Dim lcNombre As String
Dim lnNum As Integer
lcNombre = "Aurora"
' enviamos el control del programa a
' la etiqueta VerMsg, las líneas de
' código entre Goto y la etiqueta
' no se ejecutarán
GoTo VerMsg
lnNum = 111
lcNombre = lcNombre & lnNum
VerMsg:
MsgBox "El valor de lcNombre es: " & lcNombre
End Sub
```

Código fuente 97

Código fuente 100, versión con llamada a procedimiento.

```
Public Sub Main()
' declaramos variables y les
' asignamos valor
Dim lcNombre As String
Dim lnNum As Integer
lcNombre = "Aurora"
' llamamos al procedimiento VerMsg
' y le pasamos como parámetro la variable
' lcNombre, al terminar de ejecutarse
```

```
' devolverá el control a esta línea
' con lo que el resto de líneas de
' este procedimiento si se ejecutarán
VerMsg lcNombre
lnNum = 111
lcNombre = lcNombre & lnNum
End Sub
' ----------------------------------
Public Sub VerMsg(rcNombre As String)
MsgBox "El valor del Nombre es: " & rcNombre
End Sub
```

Código fuente 100

Directivas

Las directivas del compilador son una serie de instrucciones especiales que permiten compilar condicionalmente porciones de código en función de la definición de ciertas constantes especiales para el compilador.

```
#Const NombreConstante = xExpresión.
```

Mediante esta instrucción se definen las constantes de compilación condicional, y se les asigna una expresión que puede ser un número, cadena, otra constante condicional, o expresión que no incluya la palabra clave Is

No es posible declarar estas constantes como públicas, siendo siempre privadas para el módulo en que se declaran.

#If...Then...#Else...#End If

Esta estructura de control sirve para compilar condicionalmente porciones de código dependiendo de la existencia de una constante de compilación condicional. El comportamiento es análogo a la estructura If...Then...End If vista en el apartado dedicado a estructuras de control.

En el código fuente 101 se ha definido una constante de compilación al comienzo del módulo, por lo que en el bloque #If...#End If se compilará sólo el código que esté en la expresión de comprobación de existencia de la constante.

```
' definimos constante de compilación condicional
#Const Vale = 1
Public Sub Main()
Dim lcNombre As String
lcNombre = "hola"
' si la constante está definida...
#If Vale Then
   ' este código si se compilará
   MsgBox "Valor de la variable " & lcNombre
#Else
   ' este código no se compilará
   MsgBox "No hay directiva definida"
#End If
End Sub
```

Código fuente 101

Existe además un pequeño truco en el que podemos emplear las directivas del compilador, para comentar un extenso bloque de código de forma que no se ejecute, debemos incluir al principio de cada línea el carácter de comentario, con lo cual el compilador ignorará ese grupo de líneas. Pues bien, si encerramos ese bloque de código en una directiva #If...#End If preguntando por una constante de compilación condicional inexistente, ese código no se ejecutará, como muestra el código fuente 102.

```
Public Sub Main()
Dim lcNombre As String
lcNombre = "hola"
' la constante Comentario no está definida,
' por lo que el código del siguiente #If...#End If
' no se compila ni ejecuta
#If Comentario Then
lcNombre = "cadena uno"
lcNombre = "cadena otra"
#End If
lcNombre = "cadena válida"
MsgBox "Contenido de la variable " & lcNombre
End Sub
```

Código fuente 102

Estamos de acuerdo en que la actual versión de Visual Basic ya proporciona capacidades de comentar bloques de código, pero esta ha sido una buena solución hasta la versión actual y siempre es bueno conocer su existencia.

Manejo de errores

Desafortunadamente, por muy bien que planifiquemos nuestro código, siempre aparecerán errores de muy diverso origen.

Tenemos por un lado los errores producidos durante la escritura del código, aunque por suerte VB está preparado para avisarnos cuando escribimos alguna sentencia de forma incorrecta. Si intentamos introducir algo como lo que aparece en el código fuente 103,

```
Dim lcCadena As String
lcCadena = "Astral"
If lcCadena = "hola"
```

Código fuente 103

la línea que contiene el If provocará inmediatamente un aviso de error de sintaxis por falta de Then, que nos evitará problemas en ejecución, ya que rápidamente rectificaremos la línea y asunto resuelto.

Por otra parte están los errores en tiempo de ejecución, que provocan la interrupción del programa una vez que se producen. En este tipo de errores, la sintaxis es correcta, pero no la forma de utilizarla, como ocurre en el código fuente 104.

Y también tenemos los errores lógicos que quizá son los que más quebraderos de cabeza puedan ocasionarnos. En este tipo de errores, el código está correctamente escrito, produciéndose el problema

por un fallo de planteamiento en el código, motivo por el cual por ejemplo, el control del programa no entra en un bucle porque una variable no ha tomado determinado valor; sale antes de tiempo de un procedimiento al evaluar una expresión que esperábamos que tuviera un resultado diferente, etc.

Por estos motivos, y para evitar en lo posible los errores lógicos, conviene hacer un detallado análisis de lo que deseamos hacer en cada procedimiento, ya que este tipo de errores pueden ser los que más tiempo nos ocupen en descubrirlos.

```
Public Sub Main()
Dim lcCadena As String
Dim lnNum As Integer
lcCadena = "A"
' al intentar ejecutar esta línea
' se produce un error ya que intentamos
' asignar una cadena a una variable
' de tipo Integer
lnNum = lcCadena
End Sub
```

Código fuente 104

Como nadie está libre de errores, VB nos proporciona el objeto genérico *Err* para el tratamiento de errores en la aplicación, y la sentencia *On Error* para activar rutinas de tratamiento de errores.

Err

El objeto Err se instancia automáticamente al iniciarse el programa y proporciona al usuario información de los errores producidos en el transcurso de la aplicación; tiene un ámbito global, es decir, podemos usarlo directamente en cualquier punto del programa.

Un error puede ser generado por Visual Basic, un objeto o el propio programador; cuando se produce el error, las propiedades del objeto se rellenan con información sobre el error ocurrido y se reinicializan a cero al salir del procedimiento, con el método Clear(), o con la instrucción Resume.

Propiedades

- Number. Número de error de los establecidos en VB o proporcionado por el programador.
- Source. Nombre del proyecto en ejecución.
- Description. Cadena informativa sobre la causa del error.
- HelpFile. Ruta del fichero de ayuda para el error.
- HelpContext. Contiene un identificador para la ayuda contextual sobre el error.
- LastDllError. Código de error del sistema en la última llamada a una DLL.

Métodos

- Clear. Inicializa a cero o cadena vacía las propiedades del objeto Err.

- Raise(nNumber, cSource, cDescription, cHelpFile, nHelpContext). Provoca un error durante la ejecución del programa, los parámetros a pasar coinciden con las propiedades del objeto explicadas anteriormente.

On Error

Esta sentencia activa o desactiva una rutina de manejo de errores, que es una parte de código incluida en una etiqueta. On Error tiene los siguientes modos de empleo:

- **On Error GoTo Etiqueta.** Veamos el código fuente 105

```
Public Sub Main()
Dim lcCadena As String
Dim lnNum As Integer
' definimos un controlador de errores
On Error GoTo CompErr
lcCadena = "A"
' la siguiente línea provoca un error
' por lo que el control pasa a la
' etiqueta que tiene el controlador
lnNum = lcCadena
lcCadena = "otro valor"
lnNum = 232
Exit Sub
CompErr:
' tomamos información del error generado
' mediante el objeto Err
MsgBox "Error: " & Err.Number & vbCrLf & _
   "Descripción: " & Err.Description
End Sub
```

Código fuente 105

Si en un procedimiento que incluye un controlador de errores, no ponemos la instrucción Exit Sub antes de la etiqueta que da comienzo al controlador, en el caso de que no se produzca un error, al llegar a la etiqueta, el código en ella incluido será ejecutado, y como es lógico nosotros sólo queremos que ese código se ejecute únicamente cuando haya un error. Por ese motivo las rutinas de control de error se sitúan en una etiqueta al final de un procedimiento y antes del comienzo de la etiqueta se pone una sentencia Exit Sub para que el control del programa salga del procedimiento en el caso de que no existan errores.

Si queremos que el control de la aplicación vuelva al lugar donde se originó el error, tendremos que utilizar la instrucción Resume en alguna de sus variantes:

- *Resume.* Terminado de controlar el error, el programa vuelve a la línea en donde se originó dicho error. Hemos de tener cuidado de solucionar el error en la etiqueta de control de errores, ya que de lo contrario, podríamos entrar en un bucle infinito. De esta forma, el controlador de errores del fuente 105 se podría modificar como se indica en el código fuente 106.

```
CompErr:

MsgBox "Error: " & Err.Number & vbCrLf & _
```

```
   "Descripción: " & Err.Description
lcCadena = "22"

Resume
```

Código fuente 106

Si en el código fuente 104 no hiciéramos una nueva asignación a la variable lcCadena con un contenido que pueda convertirse a número, al volver al código del procedimiento se detectaría de nuevo el error y nos volvería a mandar al controlador de errores, entrando en un bucle sin salida.

- *Resume Next.* Lo veremos más adelante.
- *Resume Etiqueta*. El control pasa al código indicado en una etiqueta de código una vez haya finalizado el controlador de errores.

- **On Error Resume Next.** Cuando se produzca un error, el control del programa pasará a la siguiente línea a la que ha provocado el error. Esta forma es la más recomendable para el control de errores, puesto que podemos hacer un seguimiento del error en los lugares del procedimiento más susceptibles de que se produzca; también es la más estructurada, ya que no desvía mediante saltos el flujo de la aplicación. Veamos el código fuente 107.

```
Public Sub Main()
Dim lcCadena As String
Dim lnNum As Integer
' definimos un controlador de errores
On Error Resume Next
lcCadena = "A"
' la siguiente línea provoca un error
lnNum = lcCadena
' comprobamos el objeto Err
' si el número de error es diferente
' de cero, es que hay un error
If Err.Number <> 0 Then
   MsgBox "Error: " & Err.Number & vbCrLf & _
      "Descripción: " & Err.Description
   ' inicializar el objeto error
   ' y solucionar el error
   Err.Clear
   lcCadena = "22"
End If
lnNum = lcCadena
End Sub
```

Código fuente 107

- **On Error GoTo 0.**

Desactiva el manejador de error que haya actualmente en el procedimiento, con lo cual, si se produce un error, el flujo del programa no pasará a la etiqueta con el controlador de errores.

Como se ha comentado al ver el objeto Err, es posible para el programador generar un error para una determinada situación en la aplicación, en el código fuente 108 vemos como provocarlo.

```
Public Sub Main()
' inicializar variables y dar valor
Dim lcCadena As String
lcCadena = "Alfredo"
If lcCadena = "Alfredo" Then
   ' generar un error con el método Raise del objeto Err
   Err.Raise vbObjectError + 1200, , _
      "No se permite la asignación de esa cadena"
End If
End Sub
```

Código Fuente 108

Funciones más habituales

Visual Basic pone a disposición del programador un conjunto de funciones para ayudarle en las tareas más comunes durante el desarrollo del código de la aplicación. A continuación, repasaremos algunas de estas funciones, agrupadas por categorías.

Numéricas

- Abs(nNum). Devuelve el valor absoluto del número pasado como parámetro.

```
lnNum = Abs(32.657) ‘ 33
```

Código fuente 109

- Int(nNum) – Fix(nNum). Devuelven la parte entera del número pasado como parámetro. La diferencia entre ambas funciones es que con números negativos Int() devuelve el entero negativo menor o igual que nNum, y Fix() devuelve el entero negativo mayor o igual que nNum.

```
lnNum = Int(-32.657) ‘ -33
lnNum = Fix(-32.657) ‘ –32
```

Código fuente 110

- Sgn(nNum). Devuelve un número indicando el signo del parámetro numérico. Si nNum es positivo devuelve 1, si es negativo devuelve –1, y si es cero devuelve 0.

```
lnNum = Sgn(21) ‘ 1
lnNum = Sgn(-21) ‘ -1
lnNum = Sgn(0) ‘ 0
```

Código fuente 111

- Round(xExpresion [, nDecimales]). Redondea la expresión numérica xExpresion, pasada como parámetro, con la precisión indicada en nDecimales.

```
ldNumero = Round("15,80") ' 16
ldNumero = Round(32.2135, 3) ' 32,214
```

Código fuente 112

- Randomize nNum. Prepara el generador de número aleatorios basándose en el número pasado como parámetro, si no se pasa el parámetro, se usa el valor del reloj del sistema como número base. Una vez invocado Randomize, se utilizará la función Rnd() para generar número aleatorios.

- Rnd(nNum). Devuelve un número aleatorio de tipo Single, el parámetro nNum influirá en el modo de generar el número aleatorio basándose en una secuencia de números. Si nNum es menor que cero, se devuelve el mismo número siempre. Si es mayor que cero o no se usa, se devuelve el siguiente número de la secuencia. Si es cero devuelve el último número generado.

- Sqr(nNum). Devuelve un número de tipo Double resultado de calcular la raíz cuadrada del número pasado como parámetro.

```
ldNum = Sqr(8) ' 2,82842712474619
```

Código fuente 113

- Exp(nNum). Devuelve un número de tipo Double resultado de elevar *e* a una potencia.

```
ldNum = Exp(12) ' 162754,791419004
```

Código fuente 114

- Log(nNum). Devuelve un número de tipo Double que especifica el logaritmo de un número.

```
ldNum = Log(16) ' 2,77258872223978
```

Código fuente 115

- Funciones trigonométricas.

  - Sin(nAng). Devuelve el valor del seno del ángulo pasado como parámetro.
  - Cos(nAng). Devuelve el valor del coseno del ángulo pasado como parámetro.
  - Tan(nAng). Devuelve el valor de la tangente del ángulo pasado como parámetro.
  - Atn(nNum). Devuelve el valor del arcotangente del número pasado como parámetro.

  Los valores devueltos por estas funciones son de tipo Double.

Cadenas de caracteres

- Len(cCadena). Devuelve un número Long con la longitud de caracteres de una cadena, incluidos los espacios en blanco. También sirve para averiguar los bytes necesarios para almacenar una variable.

```
llCad = Len(" hola ") ' 6
```

Código fuente 116

- Space(nEspacios). Devuelve una cadena de espacios en blanco según el número pasado como parámetro.

```
lcCadena = Space(5) & "hola" ' "      hola"
```

Código fuente 117

- String(nNum, cCaracter). Devuelve una cadena de caracteres repetidos iguales a cCaracter, un número nNum de veces.

```
lcCadena = String(10, "Z") ' "ZZZZZZZZZZ"
```

Código fuente 118

- StrComp(cCadena1, cCadena2 [,nTipoCompara]). Devuelve un número Variant Integer con el resultado de la comparación entre dos cadenas de caracteres.
  - cCadena1, cCadena2. Cadenas a comparar.
  - nTipoCompara. Constante que indica el tipo de comparación a efectuar; la tabla 39 muestra los valores disponibles.

Constante	**Valor**	**Descripción**
vbUseCompareOption	-1	Realiza la comparación según el valor de Option Compare
vbBinaryCompare	0	Realiza una comparación binaria
VbTextCompare	1	Realiza una comparación de textp
vbDatabaseCompare	2	Realiza la comparación según los valores de la base de datos. Sólo se puede usar con Access.

Tabla 39. Constantes de comparación

Si no se utiliza este parámetro, se tomará el valor establecido en Option Compare.

Los valores devueltos pueden ser:

- -1. cCadena1 es menor que cCadena2.
- 0. cCadena1 es igual que cCadena2.
- 1. cCadena1 es mayor que cCadena2.
- Null. Alguna de las dos cadenas es Null.

Veamos un ejemplo en el código fuente 119.

```
Public Sub Main()
Dim lcCorta As String
Dim lcLarga As String
Dim lnResulta As Integer
lcCorta = "es la pequeña"
lcLarga = "esta cadena es mayor"
lnResulta = StrComp(lcCorta, lcLarga)  ' -1
End Sub
```

Código fuente 119

- InStr([nInicio,] cCadenaBuscar, cCadenaBuscada[, nTipoCompara]). Devuelve un valor Variant Long que indica la primera ocurrencia de una cadena dentro de otra. Los parámetros de esta función son los siguientes:

  - nInicio. Indica el carácter de comienzo de la búsqueda, si no se especifica comienza por el primero.
  - cCadenaBuscar. Cadena en la que se realiza la búsqueda.
  - cCadenaBuscada. Cadena que se busca dentro de cCadenaBuscar.
  - nTipoCompara. Constante que indica el tipo de comparación a efectuar (ver valores en función StrComp()).

  Los valores devueltos son los siguientes:

  - Null. Si una o ambas cadenas son Null.
  - 0. Si cCadenaBuscar es longitud cero, o no se encuentra cCadenaBuscada, o nInicio es mayor que la longitud de cCadenaBuscar.
  - nInicio. Cuando el valor de cCadenaBuscada es una cadena de longitud cero.
  - Posición encontrada. Cuando cCadenaBuscada se encuentre dentro de cCadenaBuscar.

  Veamos un ejemplo en el código fuente 120.

```
lcCadenaBuscar = "El mueble está repleto"
llPosicion = InStr(2, lcCadenaBuscar, "mueble") ' 4
```

Código fuente 120

- InStrRev(cCadenaBuscar, cCadenaBuscada[, nInicio[, nTipoCompara]]). Al igual que en InStr(), esta función busca una cadena dentro de otra, pero comenzando la búsqueda desde el final de cCadenaBuscar. Nótese que la sintaxis varía ligeramente con respecto a InStr(), aunque la descripción de los parámetros es la misma.

- Left(cCadena, nLongitud). Extrae comenzando por la parte izquierda de cCadena, una subcadena de nLongitud de caracteres.

```
lcCadena = Left("Efectivamente", 4) ' "Efec"
```

Código fuente 121

- Right(cCadena, nLongitud). Extrae comenzando por la parte derecha de cCadena, una subcadena de nLongitud de caracteres.

```
lcCadena = Right("Efectivamente", 4) ' "ente"
```

Código fuente 122

- Mid(cCadena, nInicio [, nCaractExtraer]). Extrae de cCadena, una subcadena de nCaracExtraer caracteres, comenzando en la posición nInicio. Si cCadena es Null, esta función devuelve Null. Si nInicio es mayor que el número de caracteres contenidos en cCadena, se devuelve una cadena de longitud cero. Si no se especifica nCaractExtraer, se devuelven todos los caracteres desde nInicio hasta el final de cCadena.

```
lcCadenaExtraer = "El bosque encantado"
lcCadena = Mid(lcCadenaExtraer, 11, 7) ' "encanta"
```

Código fuente 123

- Mid(varCadena, nInicio[, nCaractReemplazar]) = cCadenaSust / (Instrucción). Mid utilizado como instrucción, reemplaza una cantidad nCaractReemplazar de caracteres dentro de una variable varCadena, utilizando una cadena cCadenaSust y comenzando en la posición nInicio de varCadena.

```
lcCadenaExtraer = "El bosque encantado"
Mid(lcCadenaExtraer, 4, 5) = "teclado" ' "El teclae encantado"
```

Código fuente 124

- Replace(cCadena, cEncontrar, cReemplazar [, nInicio [, nNumCambios [, nTipoCompara]]]). Busca una subcadena dentro de otra cadena, efectuando una sustitución por otra subcadena, un número determinado de veces, vemos un ejemplo en el código fuente 121.

  - cCadena. Cadena en la que se va a realizar la sustitución.

  - cEncontrar. Cadena a sustituir.

- cReemplazar. Cadena de reemplazo.
- nInicio. Posición de cCadena desde la que se va a empezar la búsqueda, por defecto es 1.
- nNumCambios. Número de sustituciones a realizar, el valor predeterminado es -1, que indica que se realizarán todos los reemplazos posibles.
- nTipoCompara. Constante que indica el tipo de comparación a efectuar (ver valores en función StrComp().

El resultado es una nueva cadena con las sustituciones realizadas. En el caso de no obtener el resultado esperado, los valores devueltos pueden ser los siguientes:

- Cadena vacía. Si cCadena está vacía o nInicio es mayor que su longitud.
- Un error. Si cCadena es Null.
- Copia de cCadena. Si cEncontrar está vacía.
- Copia de cCadena con las ocurrencias de cEncontrar eliminadas. Si cReemplazar está vacía.
- Copia de cCadena. Si nNumCambios es 0.

```
Public Sub Main()
Dim lcCadena As String
Dim lcResulta As String
lcCadena = "En el bosque encantado habita un hado"
lcResulta = Replace(lcCadena, "ado", "cambio")

' valor de lcResulta:
' "En el bosque encantcambio habita un hcambio"

End Sub
```

Código fuente 125

- StrReverse(cCadena). Invierte el orden de los caracteres de cCadena, devolviendo el resultado en otra cadena. Si cCadena es Null, se produce un error.

```
lcResulta = StrReverse("A ver quien entiende esto")
' Resultado de StrReverse():
' otse edneitne neiuq rev A
```

Código fuente 126

- Trim(cCadena), LTrim(cCadena), RTrim(cCadena). Estas funciones eliminan los espacios en blanco en la parte izquierda de la cadena pasada como parámetro (LTrim), en la parte derecha (RTrim), o en ambos lados (Trim).

```
lcCadenaEspacios = " probando "
lcCadenaNoEsp = LTrim(lcCadenaEspacios) ' "probando "
```

```
lcCadenaNoEsp = RTrim(lcCadenaEspacios) ' " probando"
lcCadenaNoEsp = Trim(lcCadenaEspacios) ' "probando"
```

Código fuente 127

- LCase(cCadena), UCase(cCadena). Estas funciones convierten la cadena pasada como parámetro a minúsculas (LCase) o a mayúsculas (UCase).

```
lcCadena = "La Torre"
lcCadMayMin = LCase(lcCadena) ' "la torre"
lcCadMayMin = UCase(lcCadena) ' "LA TORRE"
```

Código fuente 128

- LSet varCadena = cCadena, RSet varCadena = cCadena / Instrucciones. Estas instrucciones ajustan a la izquierda (LSet) o a la derecha (Rset) la cadena cCadena dentro del contenido de una variable varCadena.

```
lcCadAjustar = "contenido largo de cadena"
LSet lcCadAjustar = "ajuste" ' "ajuste "
lcCadAjustar = "contenido largo de cadena"
RSet lcCadAjustar = "ajuste" ' " ajuste"
```

Código fuente 129

- StrConv(cCadena, nConvert, LCID). Convierte la cadena cCadena en función del valor de nConvert, que puede ser un número o una constante de VB. Entre estas constantes, destacan:
  - vbUpperCase. Convierte la cadena a mayúsculas.
  - vbLowerCase. Convierte la cadena a minúsculas.
  - vbProperCase. Convierte la primera letra de cada palabra a mayúsculas.
  - vbUnicode. Convierte la cadena a Unicode.
  - vbFromUnicode. Convierte una cadena Unicode a la página de códigos del sistema.

  Es posible especificar el identificador de la versión local del sistema, utilizando el parámetro LCID; si no se utiliza, se empleará el identificador por defecto.

```
lcCadena = "la toRrE dE lonDres"
lcCadMayMin = StrConv(lcCadena, vbLowerCase) ' "la torre de londres"
lcCadMayMin = StrConv(lcCadena, vbUpperCase) ' "LA TORRE DE LONDRES"
lcCadMayMin = StrConv(lcCadena, vbProperCase) ' "La Torre De Londres"
```

Código fuente 130

- Format(xExpresión[, xFormato[, nPrimerDíaSemana[, nPrimerDíaAño]]]). Formatea la expresión xExpresión pasada como parámetro con un nombre o cadena de formato especificado en xFormato y la devuelve en una cadena.

  El parámetro nPrimerDíaSemana es una constante de la tabla 40 que indica el primer día de la semana.

Constante	**Valor**	**Valor día semana**
VbUseSystem	0	Utiliza el valor de API NLS.
VbSunday	1	Domingo (predeterminado)
VbMonday	2	Lunes
VbTuesday	3	Martes
vbWednesday	4	Miércoles
vbThursday	5	Jueves
VbFriday	6	Viernes
VbSaturday	7	Sábado

Tabla 40. Constantes para el parámetro nPrimerDíaSemana.

El parámetro nPrimerDíaAño es una constante de la tabla 41 que indica el primer día del año.

Constante	**Valor día año**
vbFirstFourDays	Empieza en la primera semana del año que tenga al menos cuatro días
vbFirstFullWeek	Empieza en la primera semana completa del año
vbFirstJan1	Valor por defecto. Empieza en la semana donde está el 1 de enero
VbUseSystem	Utiliza el valor de API NLS

Tabla 41. Constantes para el parámetro nPrimerDíaAño.

  - Nombres de formato para fecha:

    - General Date. Muestra una fecha en formato estándar dependiendo de la configuración del sistema, ejemplo: "10/05/98".

    - Long Date. Muestra la fecha en formato completo, ejemplo: "domingo 10 de mayo de 1998".

- Medium Date. Muestra el formato de fecha mediana en función del idioma seleccionado en el sistema. Las pruebas efectuadas en este modo para la fecha 10/05/98 devolvieron la siguiente cadena: "10-may-aa", por lo que suponemos que debe existir algún bug en el tratamiento de fechas con este tipo de formato.

- Short Date. Muestra la fecha en el formato corto del sistema, ejemplo: "10/05/98".

- Long Time. Muestra la hora en el formato largo, ejemplo: "17:14:52".

- Medium Time. Muestra la hora en el formato mediano del sistema, ejemplo: 17:15:40 pasa a "05:15".

- Short Time. Muestra la hora en el formato corto del sistema, ejemplo: 17:16:39 pasa a "17:16".

o Caracteres utilizados en el formateo de fechas:

- : . Separador de hora.

- / . Separador de fecha.

- c. Muestra la fecha en formato dd/mm/aa y la hora en formato hh:mm:ss.

- d. Muestra el número de día sin cero a la izquierda, ejemplo: 7/5/98 devolvería "7".

- dd. Muestra el número de día con cero a la izquierda, ejemplo: 7/5/98 devolvería "07".

- ddd. Muestra el nombre abreviado del día, ejemplo: 7/5/98 devolvería "jue".

- dddd. Muestra el nombre completo del día, ejemplo: 7/5/98 devolvería "jueves".

- dddddd. Muestra la fecha en formato completo, ejemplo: 7/5/98 devolvería "jueves 7 de mayo de 1998".

- w. Muestra un número correspondiente al día de la semana tomando el domingo como base (1), ejemplo: 7/5/98 devolvería "5".

- ww. Muestra un número que corresponde a la semana del año de la fecha, ejemplo: 7/5/98 devolvería "19".

- m. Muestra el número de mes sin cero a la izquierda, ejemplo: 7/5/98 devolvería "5".

- mm. Muestra el número de mes con cero a la izquierda, ejemplo: 7/5/98 devolvería "05".

- mmm. Muestra el nombre del mes en forma abreviada, ejemplo: 7/5/98 devolvería "may".

- mmmm. Muestra el nombre completo del mes, ejemplo: 7/5/98 devolvería "mayo".

- q. Muestra el número de trimestre del año a que corresponde la fecha, ejemplo: 7/5/98 devolvería "2".
- y. Muestra el número de día del año de la fecha, ejemplo: 7/5/98 devolvería "127".
- yy. Muestra un número de dos dígitos para el año, ejemplo: 7/5/98 devolvería "98".
- yyyy. Muestra un número de cuatro dígitos para el año, ejemplo: 7/5/98 devolvería "1998".
- h. Muestra un número correspondiente a la hora, sin cero a la izquierda, ejemplo: 08:05:37 devolvería "8".
- hh. Muestra un número correspondiente a la hora, con cero a la izquierda, ejemplo: 08:05:37 devolvería "08".
- n. Muestra un número correspondiente a los minutos, sin cero a la izquierda, ejemplo: 08:05:37 devolvería "5".
- nn. Muestra un número correspondiente a los minutos, con cero a la izquierda, ejemplo: 08:05:37 devolvería "05".
- s. Muestra un número correspondiente a los segundos, sin cero a la izquierda, ejemplo: 08:05:03 devolvería "3".
- ss. Muestra un número correspondiente a los segundos, con cero a la izquierda, ejemplo: 08:05:03 devolvería "03".
- ttttt. Muestra la hora con el formato completo, ejemplo: "08:05:03".
- AM/PM am/pm A/P a/p. Muestran la franja horaria de la hora a formatear de la misma forma que el nombre del formato, ejemplos: 18:10:42 con AM/PM devolvería PM, con a/p devolvería p, y así con cada nombre de estos formatos.

Algunos ejemplos de formateo de fechas aparecen en el código fuente 131.

```
ldtFecha = #5/7/98#
lcCadena = Format(ldtFecha, "Long Date") ' "jueves 7 de mayo de 1998"
lcCadena = Format(ldtFecha, "ddd dd/m/yyyy") ' "jue 07/5/1998"
```

Código fuente 131

- Nombres de formato para números:
  - General Number. Muestra el número sin separadores de millar, para este y los demás ejemplos numéricos utilizaremos el número 125401.25 definido como Double, en este caso concreto devuelve "125401,25".
  - Currency. Muestra el número en formato de moneda, el ejemplo devuelve "125.401 Pts".
  - Fixed. Muestra como mínimo un dígito a la izquierda y dos a la derecha del separador decimal, el ejemplo devuelve "125401,25".

- Standard. Muestra el separador de millar y la precisión decimal de Fixed, el ejemplo devuelve "125.401,25".
- Percent. Muestra el número multiplicado por 100, con el signo de porcentaje y dos dígitos decimales, el ejemplo devuelve "12540125,00%".
- Scientific. Muestra el número mediante notación científica, el ejemplo devuelve "1,25E+05".
- Yes/No. Muestra "No" si el número es 0, en otro caso devuelve "Si" como el ejemplo.
- True/False. Muestra "Falso" si el número es 0, en otro caso devuelve "Verdadero" como el ejemplo.
- On/Off. Muestra "Desactivado" si el número es 0, en otro caso devuelve "Activado" como el ejemplo.

o Caracteres utilizados en el formateo de números:

- 0 . Marcador de dígito. Si la expresión a formatear tiene un dígito en la posición donde aparece el cero se muestra el dígito, en otro caso se muestra cero, como vemos en el código fuente 132.

```
Format(" 123", "0###") ' "0123"
```

Código fuente 132

- # . Marcador de dígito. Si la expresión a formatear tiene un dígito en la posición donde aparece #, se muestra el dígito, en otro caso no se muestra nada, ejemplo: " 23" devolvería "23", como observamos en el código fuente 133.

```
Format(" 23", "####") ' "23"
```

Código fuente 133

- "." .Marcador de decimales; en función de la configuración del país se mostrará un punto o coma; si se especifica una precisión menor a mostrar, se efectuará un redondeo automático, ejemplo: "54.38" devolvería "54,4", como aparece en el código fuente 134.

```
Format(54.38, "##.#") ' "54,4"
```

Código fuente 134

- % . Marcador de porcentaje; multiplica el número por 100 y le añade este carácter, véase el código fuente 135.

```
Format(28, "######%") ' "2800%"
```

Código fuente 135

- "," . Marcador de millar; en función de la configuración del país se mostrará un punto o coma, vemos un ejemplo en el código fuente 136

```
Format(2800, "###,###") ' "2.800"
```

Código fuente 136

- E- E+ e- e+ . Notación científica; si el número a formatear tiene a la derecha de un signo de precisión científica, un marcador 0 o # se mostrará en formato científico como vemos en el código fuente 137.

```
Format(5, "###E+###") ' "500E-2"
```

Código fuente 137

- + $ ( ). Literales, estos caracteres son mostrados en cualquier lugar de la cadena de formato donde sean situados (código fuente 138).

```
Format(45000, "$######(##)") ' "$450(00)"
```

Código fuente 138

- \ . Muestra el carácter situado a continuación de la barra inversa, como se observa en el código fuente 139.

```
Format(45000, "##\@####\°") '  4@5000
```

Código fuente 139

o Caracteres utilizados en el formateo de cadenas:

- @ . Marcador de carácter. Si la cadena de formato tiene un carácter en el lugar donde aparece este marcador, se muestra el carácter, en caso contrario, se muestra un espacio en blanco. Los caracteres de la cadena a formatear se rellenan por defecto de derecha a izquierda, excepto cuando se usa el carácter ! en la cadena de formato.

```
Format("hola", "@@@@@@@@@@") ' " hola"
```

Código fuente 140

- & . Marcador de carácter. Si la cadena de formato tiene un carácter en el lugar donde aparece este marcador, se muestra el carácter, en caso contrario, no se muestra nada vemos un ejemplo en el código fuente 141.

```
Format("hola", "&&&&&&&&&&") ' "hola"
```

Código fuente 141

- < . Convierte los caracteres de la cadena a minúscula, como aparece en el código fuente 142.

```
Format("LETRA PEQUEÑA", "<") ' "letra pequeña"
```

Código fuente 142

- >. Convierte los caracteres de la cadena a mayúscula, lo vemos en el código fuente 143.

```
Format("letra grande", ">") ' "LETRA GRANDE"
```

Código fuente 143

- ! . Rellena los caracteres de la cadena de izquierda a derecha, código fuente 144.

```
Format("hola", "!@@@@@@@@@@") ' "hola "
```

Código fuente 144

- FormatNumber(xExpresion[, nDigitosDecimales [, nIndDigito [, nParentesisNeg [, nAgrupar]]]]). Formatea como un número un valor, vemos un ejemplo en el código fuente 145.
  - xExpresion. Valor a formatear.
  - nDigitosDecimales. Número que indica la cantidad de decimales a mostrar en el número formateado. Si este valor es -1, se utilizará la configuración regional del sistema.
  - nIndDigito. Constante de estado, que indica si en los valores fraccionales se mostrará un cero indicativo.
  - nParentesisNeg. Constante de estado, que indica si se mostrará el número formateado entre paréntesis, cuando este sea negativo.

- nAgrupar. Constante de estado, que indica si el número se agrupará utilizando un delimitador de grupo.

Los valores para las constantes de estado se muestran en la tabla 42.

Constante	Valor	Descripción
vbTrue	–1	True
vbFalse	0	False
vbUseDefault	–2	Valor por defecto

Tabla 42. Constantes para valores de estado.

```
lcResulta = FormatNumber(-15.3255, 5, , vbTrue) ' "(15,32550)"
```

Código fuente 145

- FormatPercent(xExpresion[, nDigitosDecimales [, nIndDigito [, nParentesisNeg [, nAgrupar]]]]). Formatea una expresión como un porcentaje, incorporando el caracter "%". Los parámetros empleados, son los mismos que los utilizados en la función FormatNumber().

```
lcResulta = FormatPercent(128) ' "12.800,00%"
```

Código fuente 146

- FormatCurrency(xExpresion[, nDigitosDecimales [, nIndDigito [, nParentesisNeg [, nAgrupar]]]]). Formatea una expresión como valor de divisa, empleando el símbolo de divisa establecido en la configuración regional del panel de control del sistema. Los parámetros empleados, son los mismos que los utilizados en la función FormatNumber().

```
lcResulta = FormatCurrency(128) ' "128 Pts"
```

Código fuente 147

- Asc(cCadena). Devuelve un número Integer que corresponde al código del primer carácter de cCadena.

```
Asc("Mesa") ' 77
```

Código fuente 148

- Chr(lCodigoCaracter). Devuelve un carácter correspondiente al código pasado como parámetro a esta función.

```
Chr(77) ' "M"
```

Código fuente 149

- Join(cCadenas() [, cDelimitador]). Devuelve una cadena formada por los elementos contenidos en un array, odemos ver un ejemplo en el código fuente 145.

  - cCadenas(). Array unidimensional que contiene las cadenas para formar el resultado de esta función.

  - cDelimitador. Cadena utilizada para separar las cadenas que componen el resultado. Por defecto se utiliza el espacio en blanco. Si se emplea una cadena de longitud cero, la cadena resultante contendrá todas las subcadenas unidas.

```
Public Sub Main()
Dim lcCadenas(1 To 5) As String
Dim lcResulta1 As String
Dim lcResulta2 As String
Dim lcResulta3 As String
' asignar valores al array de cadenas
lcCadenas(1) = "Aquí"
lcCadenas(2) = "usamos"
lcCadenas(3) = "una"
lcCadenas(4) = "nueva"
lcCadenas(5) = "función"
' utilizar Join() de diversas formas
lcResulta1 = Join(lcCadenas())  'Aquí usamos una nueva función
lcResulta2 = Join(lcCadenas(), "==")  'Aquí==usamos==una==nueva==función
lcResulta3 = Join(lcCadenas(), "")  'Aquíusamosunanuevafunción
End Sub
```

Código fuente 150

- Split(cCadena [, cDelimitador [, nNumCadenas [, nTipoCompara]]]). Esta función es la inversa de Join(). Toma una cadena pasada como parámetro, y en función de los valores de configuración de esta función, devuelve un array cuyos elementos son subcadenas que forman la cadena original, vemos un ejemplo en el código fuente 151.

  - cCadena. Cadena que contiene las subcadenas que van a ser introducidas en el array. También se puede incluir el caracter o caracteres que actuarán como delimitador. Si la longitud de esta cadena es cero, el array devuelto no contendrá elementos.

  - cDelimitador. Caracter empleado para identificar las cadenas a incluir en el array. Por defecto se utiliza el espacio en blanco. Si se emplea una cadena de longitud cero, el array resultante tendrá como único elemento, el valor de cCadena.

  - nNumCadenas. Número de subcadenas a devolver. El valor -1 significa que se devolverán todas las subcadenas posibles.

  - nTipoCompara. Constante que indica el tipo de comparación a efectuar (ver valores en función StrComp().

```
Public Sub Main()
Dim lcOrigen As String
Dim lcValores() As String
Dim lnInd As Integer
lcOrigen = "Traspasar estos valores a un array"
lcValores = Split(lcOrigen)
For lnInd = 0 To UBound(lcValores)
   MsgBox lcValores(lnInd), , "Elemento " & lnInd & " del array"
Next
End Sub
```

Código fuente 151

Fecha y hora

- Date. Devuelve la fecha actual del sistema, lo vemos en el código fuente 152.

```
ldtFecha = Date ' 20/03/98
```

Código fuente 152

- Date = varFecha / Instrucción. Establece la fecha actual del sistema mediante un valor de tipo fecha varFecha.

```
ldtFecha = #11/21/98# ' 21 de Noviembre de 1998
Date = ldtFecha
```

Código fuente 153

- Time. Devuelve la hora actual del sistema.

```
ldtHora = Time ' 11:49:55
```

Código fuente 154

- Time = varHora / Instrucción. Establece la hora actual del sistema mediante un valor asignado en varHora, este valor puede ser una cadena reconocible con formato de hora, por lo que Time hará la conversión automática a tipo fecha/hora, en caso contrario, se producirá un error.

```
ldtHora = #10:30:21#
Time = ldtHora
```

Código fuente 155

- Now. Devuelve la fecha y hora actual del sistema.

```
ldtFechaHora = Now ' 20/03/98 12:05:21
```

Código fuente 156

- FormatDateTime(xExpresion[, nFormato]). Formatea una expresión, retornando un valor de fecha/hora.
  - xExpresion. Expresión a formatear.
  - nFormato. Constante que especifica el formato que se aplicará a xExpresion, según la tabla 43.

Constante	**Valor**	**Descripción**
VbGeneralDate	0	Muestra una fecha y/o hora en formato estándar, dependiendo de la configuración del sistema, ejemplo: "10/05/98".
VbLongDate	1	Muestra la fecha en formato completo, ejemplo: "domingo 10 de mayo de 1998".
vbShortDate	2	Muestra la fecha en el formato corto del sistema, ejemplo: "10/05/98".
vbLongTime	3	Muestra la hora en el formato largo, ejemplo: "17:14:52".
vbShortTime	4	Muestra la hora en el formato corto del sistema, ejemplo: 17:16:39 pasa a "17:16".

Tabla 43. Constantes de formato para fecha/hora.

Si no se utiliza este parámetro, se aplicará por defecto vbGeneralDate.

El código fuente 157 nos muestra un ejemplo.

```
' el resultado de este ejemplo en ambos casos es:
' "viernes 20 de noviembre de 1998"
lcResulta = FormatDateTime("20-11-98", vbLongDate)
lcResulta = FormatDateTime(#11/20/1998#, vbLongDate)
```

Código fuente 157

- MonthName(nMes[, bAbreviado]). Devuelve el nombre del mes, en función del orden numérico que ocupa en el calendario. Podemos ver un ejemplo en el código fuente 158.
  - nMes. Número correspondiente al mes: 1-Enero, 2-Febrero, etc.
  - bAbreviado. Valor lógico, que mostrará el nombre del mes abreviado en el caso de que sea True.

```
lcResulta = MonthName(4) ' abril
```

Código fuente 158

- WeekdayName(nDiaSemana[, bAbreviado [, nPrimerDiaSemana]]). Devuelve el nombre del día de la semana, en función del orden numérico que ocupa.

  - nDiaSemana. Número del día de la semana. El nombre devuelto, depende del valor establecido en el parámetro nPrimerDiaSemana.

  - bAbreviado. Valor lógico, que mostrará el nombre del día abreviado en el caso de que sea True.

  - nPrimerDiaSemana. Constante que indica el primer día de la semana (consultar la tabla para estas constantes en la función Format()).

```
lcDia = WeekdayName(6) ' sábado
```

Código fuente 159

- DateAdd(cIntervalo, nNumIntervalo, dtFecha). Suma o resta un intervalo de tiempo nNumIntervalo representado en cIntervalo a la fecha dtFecha pasada como parámetro, devolviendo un valor de tipo fecha con el resultado.

  Si nNumIntervalo es positivo, se agrega tiempo a dtFecha, si es negativo se quita tiempo.

  El parámetro cIntervalo puede tener los valores de la tabla 44, en función del espacio de tiempo a manejar.

Valor	**Tipo de intervalo**
yyyy	Año
q	Trimestre
m	Mes
y	Día del año
d	Día
w	Día de la semana
ww	Semana
h	Hora

n	Minuto
s	Segundo

Tabla 44. Valores para el parámetro cIntervalo.

En el código fuente 160 inicializamos una variable con una fecha y le aplicamos diferentes intervalos de tiempo.

```
ldtFecha = #4/25/98#
' sumar a una fecha meses, trimestres y días
' del año. Restarle también meses
ldtNuevaFecha = DateAdd("m", 2, ldtFecha) ' 25/06/98
ldtNuevaFecha = DateAdd("m", -3, ldtFecha) ' 25/01/98
ldtNuevaFecha = DateAdd("q", 3, ldtFecha) ' 25/01/99
ldtNuevaFecha = DateAdd("y", 100, ldtFecha) ' 3/08/98
```

Código fuente 160

- DateDiff(cIntervalo, dtFecha1, dtFecha2 [, nPrimerDíaSemana [, nPrimerDíaAño]]). Calcula un intervalo de tiempo cIntervalo entre las fechas dtFecha1 y dtFecha2, devolviendo el resultado en un tipo Variant de valor Long.

  Los valores disponibles para cIntervalo pueden consultarse en la función DateAdd(), y los valores para nPrimerDíaSemana y nPrimerDíaAño en la función Format().

  En el código fuente 161 se definen dos fechas y se realizan varias operaciones para calcular diferentes tipos de diferencias entre ellas.

```
' inicializar las variables con fechas
ldtFecha1 = #8/17/98#
ldtFecha2 = #4/21/99#
' calcular la diferencia de años
lvntDif = DateDiff("yyyy", ldtFecha1, ldtFecha2) ' 1
' calcular la diferencia trimestres
lvntDif = DateDiff("q", ldtFecha1, ldtFecha2) ' 3
' calcular la diferencia de semanas
lvntDif = DateDiff("ww", ldtFecha1, ldtFecha2) ' 35
```

Código fuente 161

- DatePart(cIntervalo, dtFecha [, nPrimerDíaSemana [, nPrimerDíaAño]]). Extrae el tipo de intervalo cIntervalo de una fecha dtFecha en forma de valor Variant Integer. Los valores disponibles para nPrimerDíaSemana y nPrimerDíaAño pueden consultarse en la función Format(). En el código fuente 162 vemos que se extrae diferentes intervalos de una fecha.

```
' inicializar una variable de fecha
ldtFecha = #8/17/98#
' extraer el año
lvntParte = DatePart("yyyy", ldtFecha) ' 1998
```

```
' extraer el trimestre
lvntParte = DatePart("q", ldtFecha) ' 3
' extraer el día de la semana
lvntParte = DatePart("w", ldtFecha) ' 2
' extraer el día del año
lvntParte = DatePart("y", ldtFecha) ' 229
```

Código fuente 162

- Timer. Esta función devuelve un valor numérico Single con los segundos que han pasado desde la medianoche, se utiliza para medir el tiempo transcurrido desde el comienzo hasta el final de un proceso. Lo vemos en el código fuente 163.

```
' tomar los segundos al iniciar
' el proceso
lfTiempoInicio = Timer
' comenzar el proceso
' .........
' tomar los segundos al finalizar
' el proceso
lfTiempoFin = Timer
' hallar la diferencia
lfTiempoTrans = lfTiempoFin - lfTiempoInicio
```

Código fuente 163

- DateSerial(nAño, nMes, nDía). Compone una fecha en base a los números pasados como parámetro. Veamos el código fuente 164.

```
ldtFecha = DateSerial(1990, 5, 28) ' 28/05/90
```

Código fuente 164

- DateValue(cFecha). Compone una fecha en base a la cadena pasada como parámetro, esta cadena ha de tener un formato que pueda ser convertido a fecha, en caso contrario, se producirá un error. En el código fuente 165 vemos un ejemplo.

```
ldtFecha = DateValue("15 feb 1985") ' 15/02/85
```

Código fuente 165

- TimeSerial(nHora, nMinutos, nSegundos). Crea un valor de hora con los números pasados como parámetro (código fuente 166).

```
ldtHora = TimeSerial(18, 20, 35) ' 18:20:35
```

Código fuente 166

- TimeValue(cHora). Devuelve un valor de hora creado con la cadena cHora pasada como parámetro; esta cadena ha de tener un contenido convertible a hora, en caso contrario, ocurrirá un error. En el código fuente 167 aparece un ejemplo.

```
ldtHora = TimeValue("3:40:10 PM") ' 15:40:10
```

Código fuente 167

4

El entorno de desarrollo (IDE)

El nuevo aspecto de nuestra mesa de trabajo

El entorno de desarrollo de Visual Basic, ya desde la versión 5, dispone de un modo de trabajo basado en un interfaz MDI. Siguiendo la filosofía del Microsoft Visual Studio de dotar a todas las herramientas de la compañía de una interfaz de trabajo similar, el entorno de desarrollo ha sido adaptado al modo de trabajo de los demás productos de desarrollo de Microsoft, por lo que ahora se presenta con una ventana principal que integra todos los demás elementos en su interior: editores visuales, de código, barras de herramientas, etc. Aunque el gran avance reside en que este entorno es totalmente configurable a los gustos y necesidades del programador, pudiendo personalizar las barras de botones, añadiendo o quitando elementos, situarlas en cualquier parte del IDE y lo que es más interesante, la posibilidad de crear barras de botones *a la carta*, con las opciones que el programador decida. Este entorno lo podemos ver representado en la figura 75.

Para quien se sienta más cómodo con el entorno a la antigua usanza de las versiones 4 y anteriores, también existe la posibilidad de seguir trabajando con el IDE en modo interfaz SDI.

También podemos construir nuestros propios complementos e integrarlos en el entorno gracias a un asistente destinado a tal efecto.

En este análisis del entorno de desarrollo, comenzaremos viendo los menús y cada una de sus opciones. Posteriormente haremos un estudio en mayor profundidad de aquellos elementos que puedan resultar más complejos, o que el programador vaya a usar asiduamente y necesite conocer más a fondo.

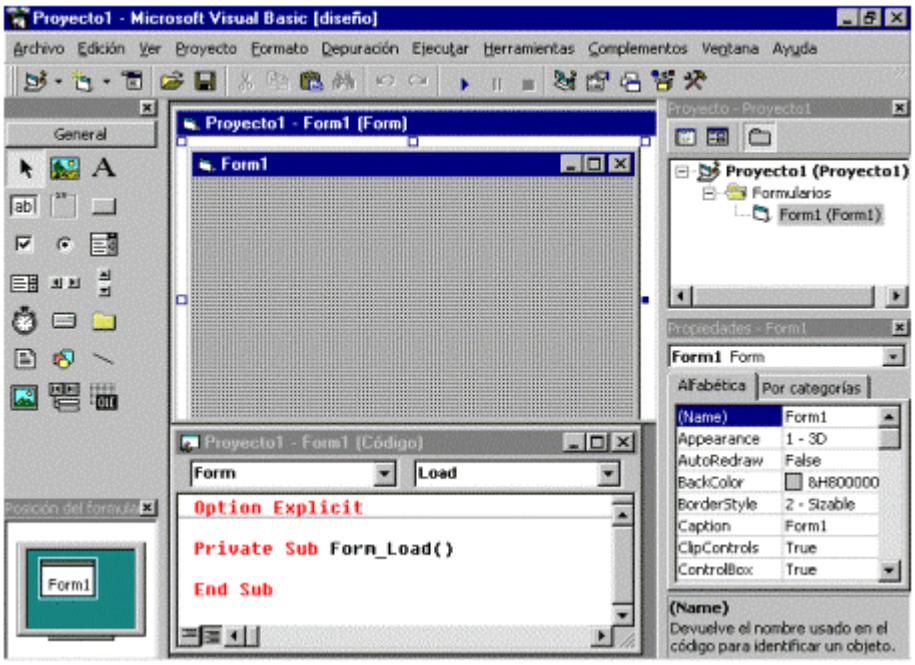


Figura 75. Entorno de desarrollo de Visual Basic 5.

Menú Archivo

- **Nuevo Proyecto.** Abre un nuevo proyecto previa grabación del actual, pidiendo al usuario que seleccione el tipo de proyecto a crear. La combinación de teclado para esta opción es *Ctrl+N*.

- **Abrir Proyecto.** Carga un proyecto existente en el entorno cerrando el proyecto actual. La combinación de teclado para esta opción es *Ctrl+O*.

  El botón en la barra de herramientas es:  . Al referirnos a barra de herramientas, estamos indicando que el botón puede estar en la barra de botones principal de VB o en alguna de las barras que permanecen ocultas y son visibles con la opción Barras de herramientas del menú Ver.

- **Agregar Proyecto.** Añade un proyecto nuevo o existente al proyecto actualmente abierto, creando un grupo de proyectos. Como se muestra en la figura 76

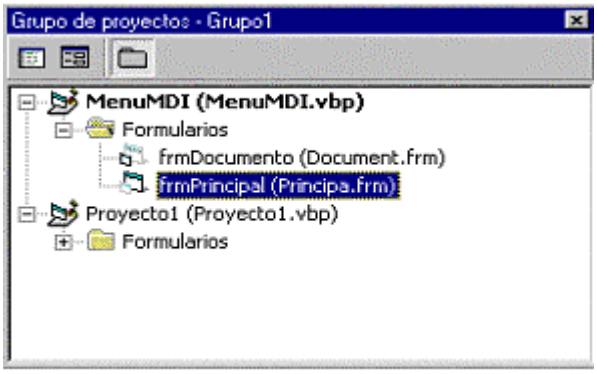


Figura 76. Explorador de proyectos mostrando grupo de proyectos.

El botón en la barra de herramientas es:  Este botón puede desplegarse, mostrando los diversos tipos de proyectos que podemos agregar.

- **Quitar Proyecto.** Elimina el proyecto seleccionado en el explorador.

- **Guardar proyecto, Guardar proyecto como, Guardar grupo de proyectos, Guardar grupo de proyectos como.** Guarda el proyecto o grupo actual, si es la primera vez que se guarda, aparecerá la caja de diálogo para dar un nombre a los componentes del proyecto.

  El botón en la barra de herramientas es: 

- **Guardar <NombreFichero>, Guardar <NombreFichero> como.** Guarda en un fichero el formulario o código que actualmente se está editando. La combinación de teclado para esta opción es *Ctrl+S.*

- **Imprimir.** Imprime el código fuente de un módulo o la imagen del formulario. El cuadro de diálogo que aparece al seleccionar esta opción se muestra en la figura 77. La combinación de teclado para esta opción es *Ctrl+P.*

  El botón en la barra de herramientas es: 

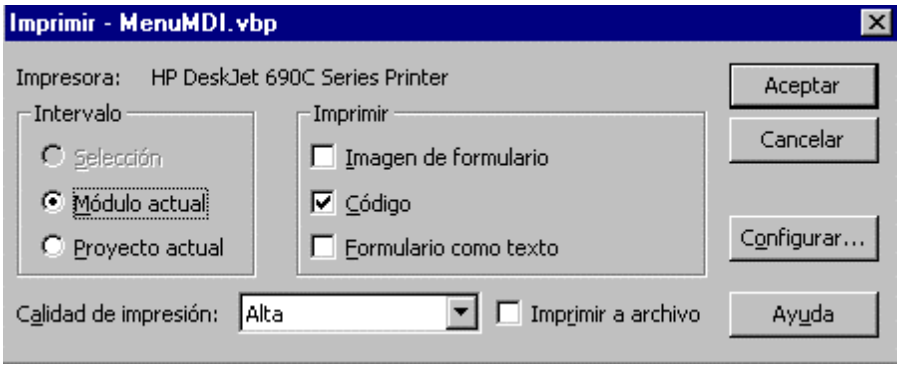


Figura 77. Cuadro de diálogo Imprimir de Visual Basic.

- **Configurar impresora.** Visualiza la caja de diálogo estándar de configuración de impresora.

- **Generar NombreProyecto.exe>.** Crea el ejecutable para el proyecto actual.

- **Generar grupo de proyectos.** Crea los diferentes ejecutables para cada uno de los proyectos integrantes de un grupo.

- **Últimos proyectos ejecutados.** Es un grupo de cuatro opciones que contienen el nombre de los últimos proyectos cargados en el entorno.

- **Salir.** Finaliza la ejecución de VB. La combinación de teclado para esta opción es *Alt+Q.*

Menú Edición

- **Deshacer.** Deshace la última operación efectuada en el código o en los controles de un formulario. La combinación de teclado para esta opción es *Ctrl+Z o Alt+Retroceso.*

  El botón en la barra de herramientas es: 

- **Rehacer.** Repone la última acción efectuada por *Deshacer*. Si por ejemplo, en el editor de código fuente eliminamos una línea, esta puede ser recuperada con el comando *Deshacer*, y eliminada otra vez con el comando *Rehacer*, que restaura la eliminación.

  El botón en la barra de herramientas es: 

- **Cortar.** Pasa al portapapeles el control o texto seleccionado, eliminándolo de su lugar de origen. Es necesario que exista algún elemento seleccionado para que este comando esté habilitado. La combinación de teclado para esta opción es *Ctrl+X o Mayús+Supr.*

  El botón en la barra de herramientas es: 

- **Copiar.** Pasa al portapapeles el control o texto seleccionado, sin eliminarlo de su lugar de origen. Es necesario que exista algún elemento seleccionado para que este comando esté habilitado. La combinación de teclado para esta opción es *Ctrl+C o Ctrl+Ins*

  El botón en la barra de herramientas es: 

- **Pegar.** Inserta el contenido del portapapeles en el punto donde estemos situados. La combinación de teclado para esta opción es *Ctrl+V o Mayús+Ins.*

  El botón en la barra de herramientas es: 

- **Pegar vínculo.** Posibilita pegar un vínculo a un origen DDE. El portapapeles debe contener un origen DDE válido y el control seleccionado debe ser un vínculo DDE válido. También puede usarse esta opción para transferir datos de otras aplicaciones usando el control contenedor ActiveX, por ejemplo un rango de celdas copiadas al portapapeles desde una hoja de cálculo.

- **Eliminar.** Suprime el control o texto seleccionado. La combinación de teclado para esta opción es *Supr*

El botón en la barra de herramientas es: 

- **Eliminar tabla de la base de datos.** Suprime la tabla seleccionada en la ventana de Diagrama de base de datos.

- **Seleccionar todo.** Selecciona todo el código fuente o controles del formulario. La combinación de teclado para esta opción es *Ctrl+A.*

- **Seleccionar todas las columnas.** Selecciona todas las columnas en la ventana del diseñador de consultas.

- **Tabla.** Esta opción despliega un submenú con el que podemos efectuar las siguientes operaciones sobre una tabla de una base de datos, seleccionada en una ventana de diagramas de base de datos.

  - Establecer clave principal. Indica la clave primaria de la tabla.
  - Insertar columna. Añade una columna nueva a la tabla.
  - Eliminar columna. Borra una columna existente de la tabla.

  Buscar. Abre una caja de diálogo para introducir un texto a buscar y seleccionar los parámetros de búsqueda, como podemos ver en la figura 78. La combinación de teclado para esta opción es *Ctrl+F*.

  El botón en la barra de herramientas es: 

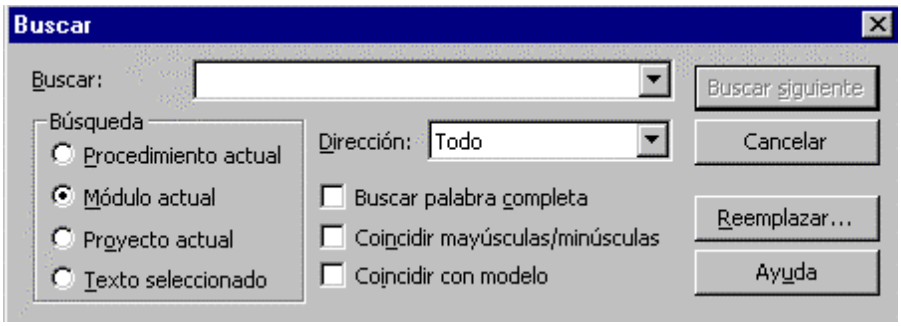


Figura 78. Ventana de búsqueda de código fuente.

- **Buscar siguiente.** Buscar la siguiente ocurrencia del texto especificado en la opción *Buscar*. La combinación de teclado para esta opción es *F3* para la siguiente ocurrencia y *Mayús+F3* para la ocurrencia anterior.

  El botón en la barra de herramientas es: 

- **Reemplazar.** Abre una caja similar a la de la opción *Buscar* pero con opciones adicionales para reemplazar el texto buscado por otro introducido por el usuario. La combinación de teclado para esta opción es *Ctrl+H.*

  El botón en la barra de herramientas es: 

- **Aplicar sangría.** Desplaza las líneas de código seleccionadas hasta el siguiente tabulador. La combinación de teclado para esta opción es *Tab.*

El botón en la barra de herramientas es: 

- Anular sangría. Desplaza las líneas de código seleccionadas hasta el anterior tabulador. La combinación de teclado para esta opción es *Mayús+Tab.*

  El botón de la barra de herramientas es: 

- **Insertar archivo.** Incluye el contenido de un archivo de texto en la ventana de código.

- **Mostrar propiedades y métodos.** Abre una lista desplegable en la ventana de código fuente con las propiedades y métodos disponibles para el objeto que se encuentre en ese momento en el código. Si se invoca este comando en una zona en blanco del editor, aparecerá un conjunto de métodos globales a la aplicación.

  Para encontrar un elemento de la lista podemos teclearlo, con lo que se efectuará una búsqueda dinámica del elemento de la lista lo más similar al que llevamos tecleado. También podemos usar las teclas de dirección o el ratón para movernos por la lista. Una vez localizado el elemento, podemos seleccionarlo haciendo doble clic, pulsando la tecla *Tab* o *Enter*. La combinación de teclado para esta opción es *Ctrl+J.*

  El botón en la barra de herramientas es: 

- **Mostrar constantes.** Abre una lista desplegable en la ventana de código fuente cuando vamos a dar un valor a una variable que puede ser una constante definida de VB, o en el parámetro de una función que admita constantes.

  Un claro ejemplo de esta opción lo tenemos al definir una variable de tipo *Boolean*, cuando vamos a asignarle un valor, si abrimos esta lista obtendremos como valores disponibles *False* y *True*. La combinación de teclado para esta opción es *Ctrl+Mayús+J.*

  El botón en la barra de herramientas es: 

  Información rápida. Muestra información sobre la variable, función, propiedad, etc., seleccionada en la ventana de Código, vemos un ejemplo en la figura 79. La combinación de teclado para esta opción es *Ctrl+I*

  El botón en la barra de herramientas es: 

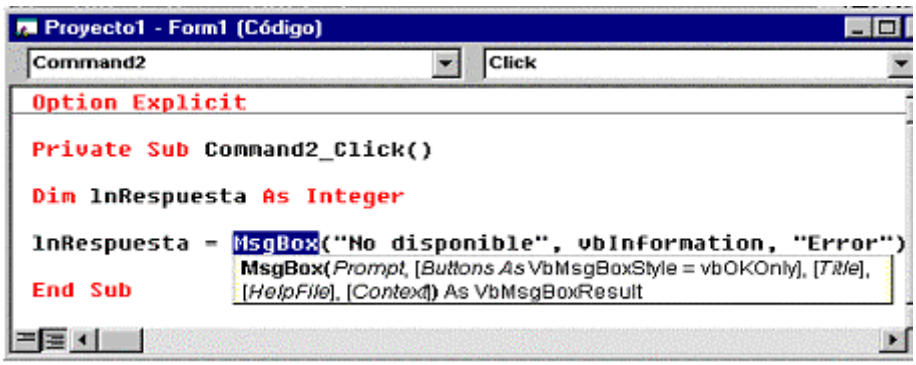


Figura 79. Información sobre sintaxis de la función MsgBox() en la ventana de Código.

- **Información de parámetros.** Muestra información sobre los parámetros de la función actual. La combinación de teclado para esta opción es *Ctrl+Mayús+I.*

  El botón en la barra de herramientas es: 

- **Palabra completa.** Completa el resto de la palabra que estamos escribiendo si tiene suficiente información para identificarla. La combinación de teclado para esta opción es *Ctrl+Alt+A*.

  El botón en la barra de herramientas es: 

- **Ir a la fila.** Nos permite desplazarnos a un registro de un tabla, consulta, etc., cuando por ejemplo, hemos ejecutado una tabla desde la ventana Vista Datos. Las opciones del submenú que despliega esta opción son:

  - Primera. Efectúa un desplazamiento a la primera fila de la tabla.
  - Última. Efectúa un desplazamiento a la última fila de la tabla.
  - Siguiente. Efectúa un desplazamiento a la siguiente fila de la tabla.
  - Anterior. Efectúa un desplazamiento a la anterior fila de la tabla.
  - Fila. Muestra un cuadro de diálogo para introducir la fila a la que realizar el desplazamiento.
  - Nuevo. Crea una nueva fila para agregar datos.

- **Marcadores.** Despliega un submenú con opciones para marcar líneas de código fuente, de forma que podamos acceder rápidamente a líneas que necesitemos consultar frecuentemente durante una sesión de trabajo con un proyecto. Estos marcadores no se conservan entre proyectos, por lo que al volver a abrirlo deberemos de insertar unos nuevos. Una vez insertado un marcador, aparecerá junto a la línea de código el siguiente símbolo:  .

  Las opciones del submenú son las siguientes:

  - **Alternar marcador.** Inserta/elimina un marcador

    El botón en la barra de herramientas es: 

  - **Marcador siguiente.** Posiciona el cursor en el siguiente marcador.

    El botón en la barra de herramientas es: 

  - **Marcador anterior.** Posiciona el cursor en el anterior marcador.

    El botón en la barra de herramientas es: 

  - **Borrar todos los marcadores.** Elimina todos los marcadores definidos.

    El botón en la barra de herramientas es: 

Menú Ver

- **Código.** Abre la ventana de Código para el formulario o control seleccionado. La combinación de teclado para esta opción es *F7*.

El botón en la barra de herramientas es: 

- **Objeto.** Visualiza el objeto seleccionado en el Explorador de Proyectos o Ventana de Código. La combinación de teclado para esta opción es *Mayús+F7*.

El botón en la barra de herramientas es: 

- **Definición.** Estando situados en el nombre de un procedimiento, al ejecutar este comando nos desplazamos al código de dicho procedimiento. La combinación de teclado para esta opción es *Mayús+F2*.

- **Última posición.** Después de haber ejecutado la opción anterior, si queremos volver al punto del código en donde estábamos situados antes de ir a la definición de un procedimiento, ejecutaremos esta opción que nos devolverá a la línea de código original. La combinación de teclado para esta opción es *Ctrl+Mayús+F2*.

- **Examinador de objetos.** Abre la ventana del Examinador de objetos, en donde podemos ver todos los componentes que forman parte del proyecto, librerías de objetos, constantes, clases, métodos, funciones, etc. La combinación de teclado para esta opción es *F2*.

  El botón en la barra de herramientas es: 

- **Ventana Inmediato.** Durante la ejecución de un programa desde el entorno, es posible interrumpir momentáneamente dicha ejecución y abrir la ventana Inmediato, en la cual podemos introducir expresiones para evaluar su resultado, comprobar y asignar el valor de variables y propiedades, etc. La combinación de teclado para esta opción es *Ctrl+G*.

  El botón en la barra de herramientas es: 

- **Ventana Locales.** En el modo de depuración del programa, esta opción muestra la ventana Locales en donde se pueden ver las variables de la rutina y módulo actual junto a sus valores, que se van actualizando dinámicamente en el transcurso de la aplicación.

  El botón en la barra de herramientas es: 

- **Ventana Inspección.** Visualiza la ventana de Inspección, en la cual se pueden definir expresiones para que el programa entre en modo de depuración cuando estas expresiones cambien de valor y ver su resultado. La combinación de teclado para esta opción es *Ctrl+Alt+G*.

  El botón en la barra de herramientas es: 

- **Pila de llamadas.** Muestra la lista de llamadas entre procedimientos que se ha realizado hasta el momento en la aplicación. Cuando desde un procedimiento o función se llama a otro, el último llamado se sitúa al comienzo de una lista o pila de llamadas. Igualmente al ir terminando su ejecución, cada procedimiento es eliminado de la pila. La combinación de teclado para esta opción es *Ctrl+L*.

  El botón en la barra de herramientas es: 

- **Explorador de proyectos.** Abre el Explorador de proyectos, que muestra los elementos que componen nuestro actual proyecto o grupo de proyectos. La combinación de teclado para esta opción es *Ctrl+R.*

  El botón en la barra de herramientas es: 

- **Ventana Propiedades.** Visualiza las propiedades para el formulario, control o módulo actual . La combinación de teclado para esta opción es *F4.*

  El botón en la barra de herramientas es: 

- **Ventana Posición del formulario.** Visualiza una vista previa de la disposición de los formularios en la pantalla antes de ejecutar la aplicación (figura 80).

Figura 80. Ventana de Posición del formulario.

El botón en la barra de herramientas es: 

- **Páginas de propiedades.** Muestra una ventana con las propiedades adicionales de un control de usuario. La combinación de teclado para esta opción es *Mayús+F4.*

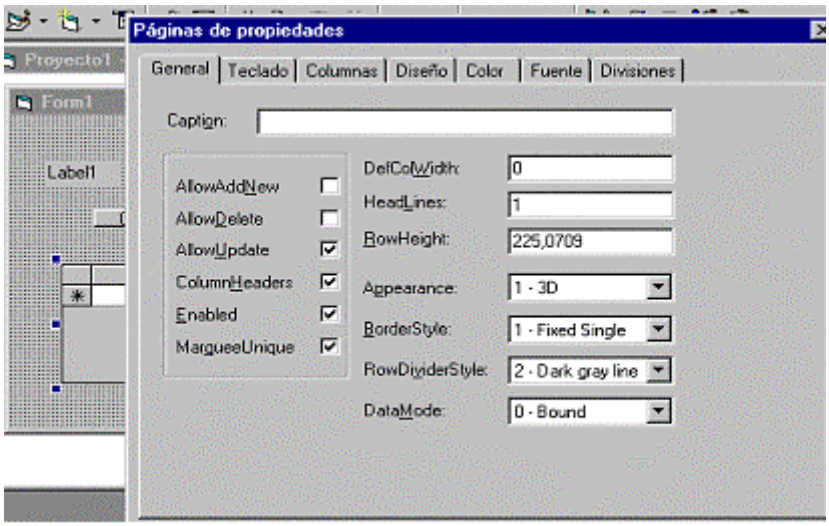


Figura 81. Páginas de propiedades para el control DBGrid.

- **Tabla.** Abre un submenú que nos permite ver diferentes aspectos de una tabla que esté seleccionada en la Ventana Diagrama de base de datos. Las opciones de este submenú son:

  - Propiedades de columna. Visualiza las propiedades de cada columna de la tabla seleccionada.

  - Nombres de columna. Visualiza los nombres de cada columna de la tabla.

  - Claves. Esta opción sólo muestra los nombres de las columnas que están incluidos en alguna de las claves de la tabla.

  - Sólo nombre. Sólo muestra el nombre de la tabla seleccionada.

  - Personalizado. Visualiza la tabla según la información establecida en la vista personalizada de la ventana Diagrama.

- **Zoom.** Posibilita la visualización del diagrama de la base de datos acercando o alejando sus elementos en función de un porcentaje. La subopción Selección muestra en primer plano la tabla seleccionada en el diagrama.

  La figura 82 muestra un diagrama de la base de datos Northwind, incluida con los ejemplos de SQL Server, en la que podemos ver varias tablas con las opciones anteriormente comentadas: nombres de columna, claves, etc., y su zoom establecido a 75%. En concreto, la tabla Suppliers muestra los nombre de las columnas; Products las columnas que forman la clave; Order Details las propiedades de las columnas; y Categories sólo su nombre.

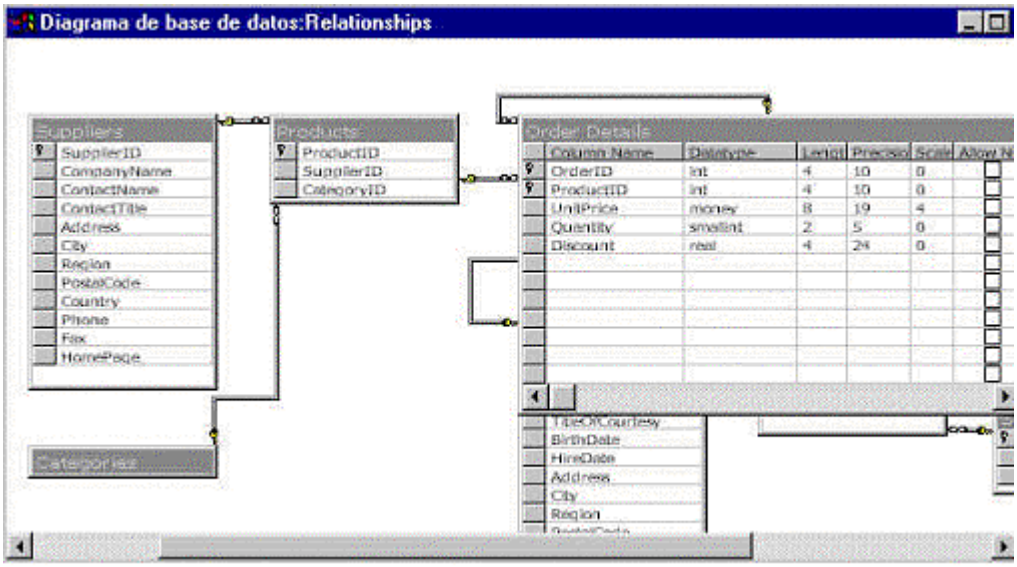


Figura 82. Vista del diagrama de una base de datos.

- Mostrar paneles. Al ejecutar una tabla o vista en la ventana Vista Datos u otro diseñador de datos, por defecto se muestran las filas que componen dicha tabla. Con las opciones de este submenú, es posible mostrar la siguiente información adicional:

  - Diagrama. Este panel muestra una representación gráfica de las tablas que componen la consulta ejecutada. No confundir con la ventana Diagrama vista anteriormente.

- Cuadrícula. Permite especificar al programador la información de la consulta: columnas a mostrar, rango de filas, etc.
- SQL. Visualiza la consulta actual en sintaxis SQL.
- Resultados. Panel más habitual en el que podemos ver las filas y columnas que componen la consulta.

La figura 83 muestra la ejecución de una tabla de una base de datos desde la ventana Vista Datos con todos los paneles de la ventana resultante abiertos.

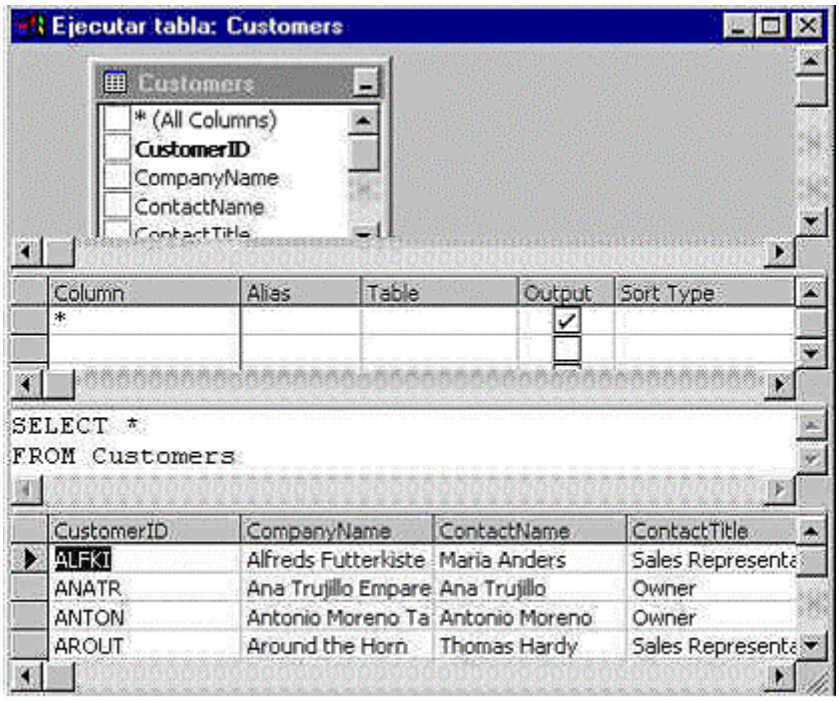


Figura 83. Ejecución de una tabla de Vista Datos, mostrando sus paneles de información.

- **Cuadro de herramientas.** Visualiza los controles habituales de VB y permite incorporar nuevos controles personalizados, lo vemos en la figura 84.

Figura 84. Caja de herramientas de VB.

El botón en la barra de herramientas es: 

- **Ventana de la vista Datos.** Muestra la ventana Vista Datos (figura 85), en la que podemos establecer vínculos con las diferentes bases de datos instaladas en el sistema y crear nuevos objetos DataEnvironment.

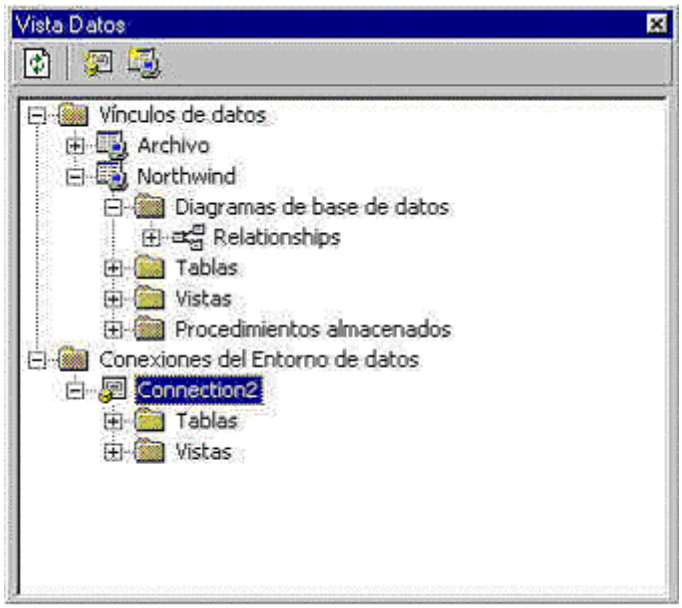


Figura 85. Ventana Vista Datos.

- **Paleta de colores.** Abre la ventana de selección de color para un formulario o control, como se muestra en la figura 86.

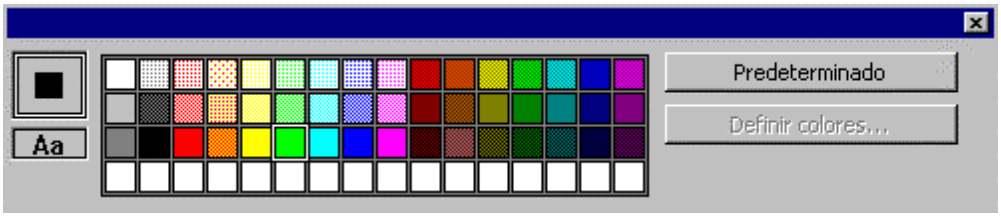


Figura 86. Paleta de colores de VB.

- **Barras de herramientas.** Muestra un submenú con las diferentes barras de herramientas disponibles en VB (figura 87). Cada opción contiene una marca que indica la barra que está visible. Para ocultar una determinada barra, lo único que debemos hacer es volver a pulsar sobre la opción que esté marcada.

  Este menú también está disponible al hacer clic sobre cualquier barra de herramientas con el botón secundario del ratón

  La opción *Personalizar* permite además al usuario crear sus propias barras de herramientas, véase figura 88, con las opciones que le sean de más utilidad.

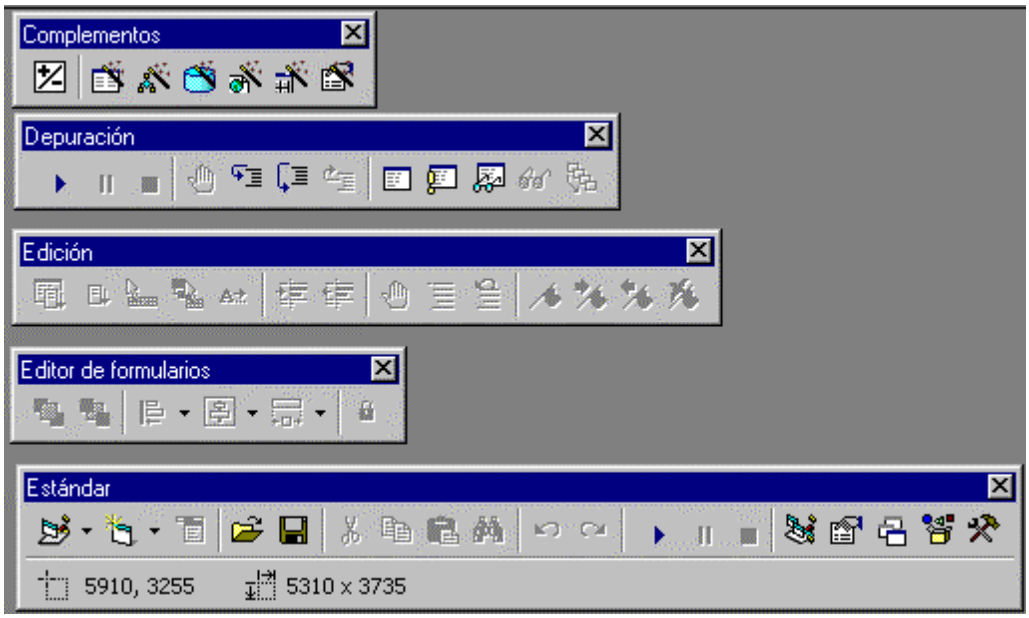


Figura 87. Barras de herramientas de VB.

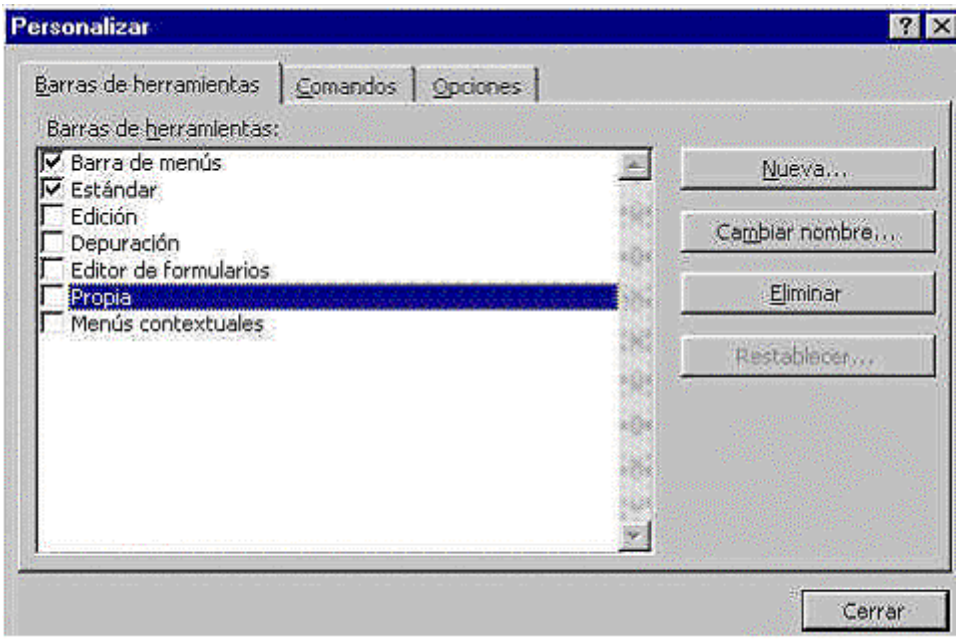


Figura 88. Ventana para personalizar las barras de herramientas de VB.

En la ventana de personalización de barras, la pestaña Barras de herramientas nos permite seleccionar una existente o crearla nueva para agregarle las opciones que necesitemos.

Desde la pestaña Comandos, seleccionaremos las opciones y las arrastraremos hasta la barra en que vayamos a añadir. Una vez soltada la opción en la barra, es posible seleccionarla y abrir un menú contextual para modificar su icono asociado, mostrar además del icono una descripción de la opción, etc. La figura 89, muestra esta ventana y una nueva barra de herramientas con varias opciones personalizadas.

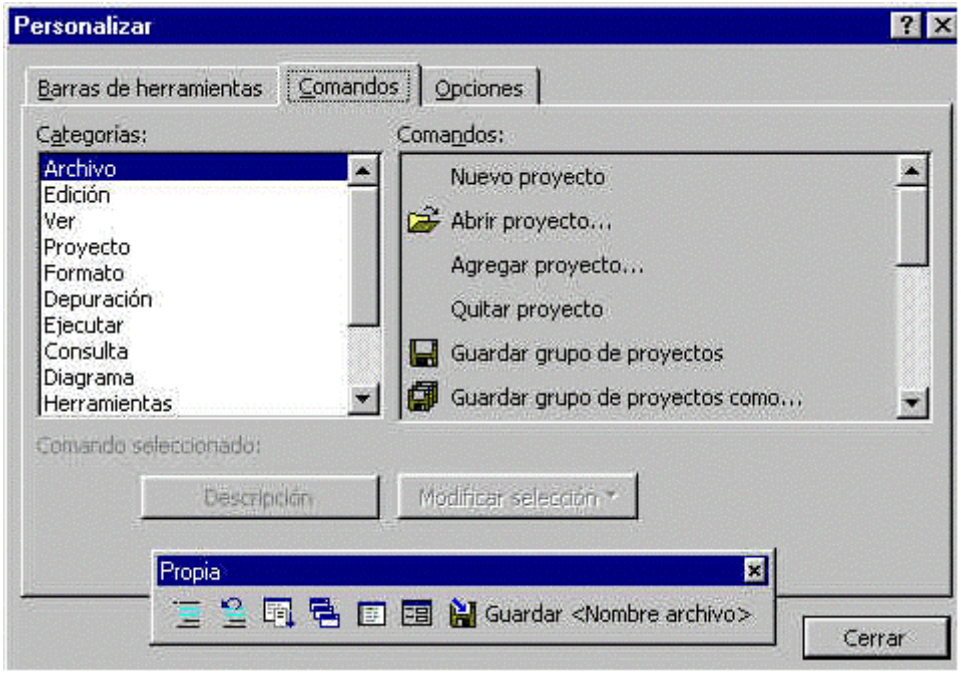


Figura 89. Nueva barra de herramientas personalizada, y comandos disponibles.

Finalmente, la pestaña Opciones, nos permite establecer entre otras, la visualización de los iconos en un tamaño superior, descripciones abreviadas en los botones, etc.

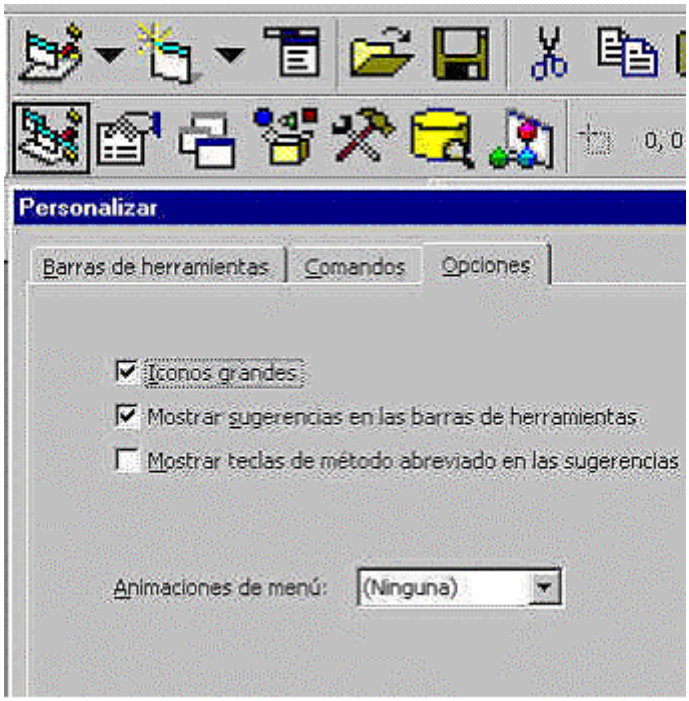


Figura 90. Pestaña Opciones de personalizar barras de herramientas, con la visualización de iconos grandes activada.

Menú Proyecto

- **Agregar formulario.** Abre una caja de diálogo para insertar un nuevo formulario al proyecto de los diferentes tipos disponibles, o uno ya existente.

  El botón en la barra de herramientas es:  . Este es un botón desplegable, que permite agregar diferentes elementos al proyecto.

- **Agregar formulario MDI.** Similar a la opción anterior con la salvedad de que en esta sólo nos permite insertar un formulario MDI en la aplicación. Si la aplicación ya tuviera un formulario MDI, esta opción estaría deshabilitada.

  El botón en la barra de herramientas es: 

- **Agregar módulo.** Abre la caja de diálogo para añadir un módulo de código nuevo o existente al proyecto.

  El botón en la barra de herramientas es: 

- **Agregar módulo de clase.** Abre la caja de diálogo para añadir un módulo de clase. Este tipo de módulos contienen el código para la definición de una clase (propiedades y métodos).

  El botón en la barra de herramientas es: 

- **Agregar control de usuario.** Abre la caja de diálogo para crear un nuevo control o añadir un control existente.

  El botón en la barra de herramientas es: 

- **Agregar página de propiedades.** Permite incorporar páginas de propiedades para un control.

  El botón en la barra de herramientas es: 

- **Agregar documento de usuario.** Si el proyecto es EXE o DLL de ActiveX, con esta opción podemos insertar un documento de usuario nuevo o existente en el proyecto.

  El botón en la barra de herramientas es: 

- **Agregar DHTML Page, Agregar Data Report, Agregar WebClass, Agregar Microsoft UserConnection, Más diseñadores ActiveX.** Este conjunto de opciones nos permiten incorporar al proyecto cualquiera de los diseñadores ActiveX mencionados.

- **Agregar archivo.** Agrega un archivo al proyecto actual, formulario, módulo, clase, control, etc. No incorpora el archivo físico al proyecto, sino que guarda una referencia al mismo; por este motivo, si modificamos el archivo, ese cambio afectará a todos los proyectos que tengan una referencia a él. La combinación de teclado para esta opción es *Ctrl+D.*

- **Quitar <NombreArchivo>.** Elimina del proyecto el elemento actualmente seleccionado, no lo borra físicamente del disco, sino que quita la referencia que se mantiene hacia él dentro del proyecto.

- **Referencias.** Muestra la ventana de Referencias (figura 91), de forma que podamos agregar o quitar referencias a librerías de objetos.

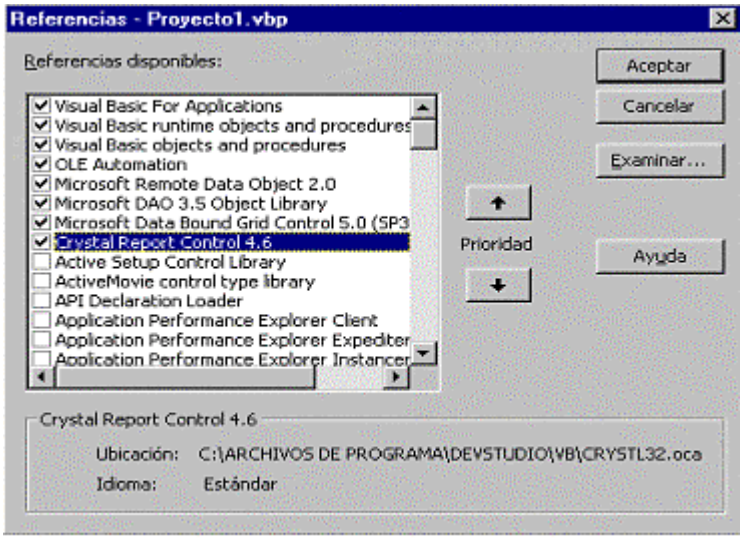


Figura 91. Ventana de Referencias de un proyecto.

Una vez que hemos creado una referencia hacia una librería de objetos, podemos utilizar los objetos contenidos en ella dentro de nuestro proyecto, visualizándose también en el Examinador de objetos.

El botón en la barra de herramientas es: 

- **Componentes.** Visualiza la ventana Componentes (figura 92), desde la que podemos insertar nuevos controles a la caja de herramientas, diseñadores u objetos insertables como una hoja de Excel. La combinación de teclado para esta opción es *Ctrl+T*.

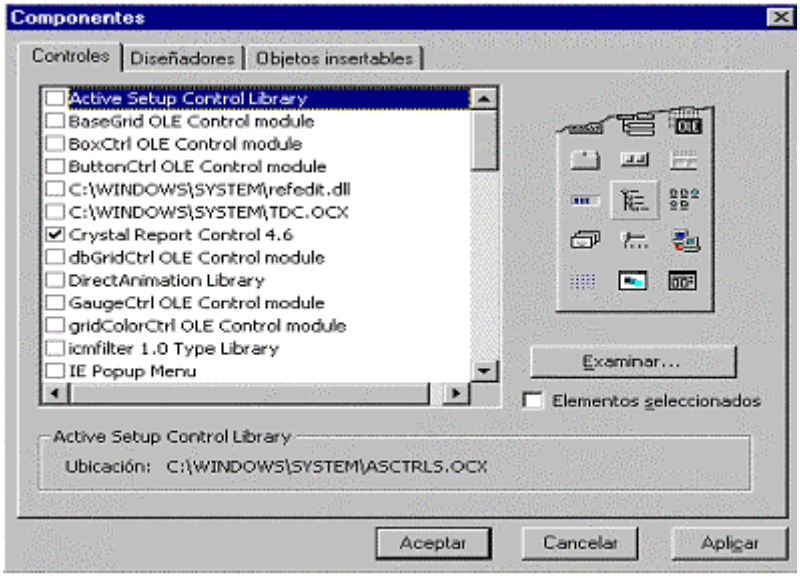


Figura 92. Ventana de Componentes de VB.

- **Propiedades de <NombreProyecto>.** Visualiza una caja de diálogo con las propiedades del actual proyecto (figura 93). Entre las propiedades modificables de las diferentes fichas de la caja están el nombre del proyecto, tipo, formulario o procedimiento de inicio, una breve descripción de la funcionalidad del proyecto, versión, icono, tipo de compilación, etc.

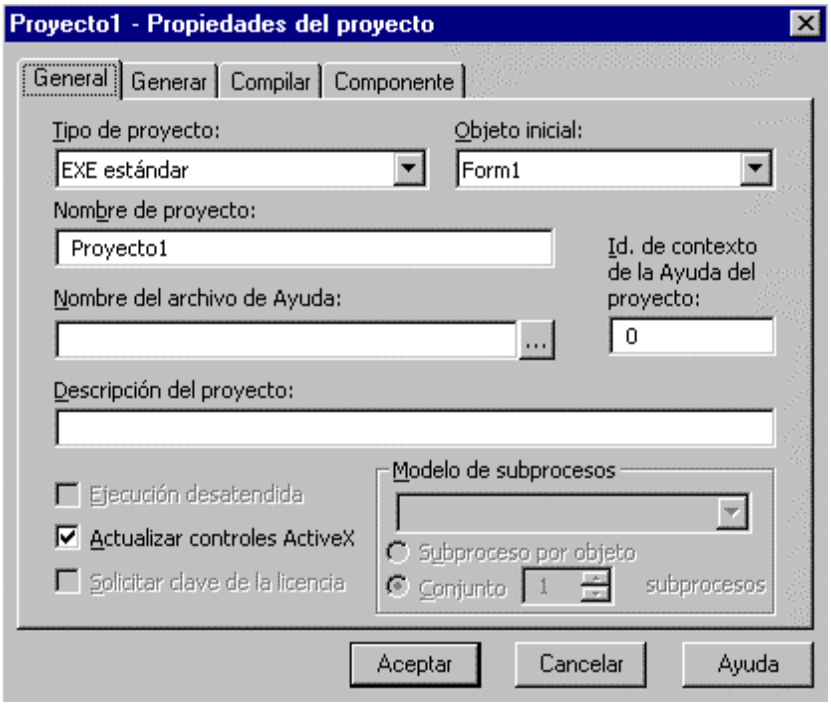


Figura 93. Propiedades de un proyecto.

Menú Formato

- Alinear. Abre un submenú con las siguientes opciones para alinear los controles seleccionados. El control que tiene los controladores de desplazamiento blancos se utilizará como referencia para alinear los controles seleccionados.

  - **Izquierda.** Sitúa horizontalmente los controles, alineados por el borde izquierdo.

    El botón en la barra de herramientas es: 

  - **Centro.** Sitúa horizontalmente los controles, alineados por el centro.

    El botón en la barra de herramientas es: 

  - **Derecha.** Sitúa horizontalmente los controles, alineados por el borde derecho.

    El botón en la barra de herramientas es: 

  - **Superior.** Sitúa verticalmente los controles, alineados por el lado superior.

    El botón en la barra de herramientas es: 

  - **Medio.** Sitúa verticalmente los controles, alineados por la mitad.

    El botón en la barra de herramientas es: 

  - **Inferior.** Sitúa verticalmente los controles, alineados por el lado inferior.

    El botón en la barra de herramientas es: 

  - **A la cuadrícula.** Sitúa los controles en la cuadrícula más próxima por el lado superior izquierdo.

    El botón en la barra de herramientas es: 

- **Igualar tamaño.** Tomando como referencia el último control seleccionado, adapta el tamaño del resto de los controles hasta igualarlo.

  - **Ancho.** Iguala el ancho.

    El botón en la barra de herramientas es: 

  - **Alto.** Iguala el alto.

    El botón en la barra de herramientas es: 

  - **Ambos.** Iguala ancho y alto.

    El botón en la barra de herramientas es: 

- **Ajustar tamaño a la cuadrícula.** Adapta el tamaño del control seleccionado a la cuadrícula más próxima.

El botón en la barra de herramientas es: 

- **Espacio horizontal.** Cambia el espacio horizontal entre los controles.

  - **Igualar.** Establece la misma distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 

  - **Aumentar.** Aumenta la distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 

  - **Disminuir.** Disminuye la distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 

  - **Quitar.** Elimina el espacio entre los controles de forma que quedan pegados.

    El botón en la barra de herramientas es: 

- **Espacio vertical.** Cambia el espacio vertical entre los controles.
  - **Igualar.** Establece la misma distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 
  - **Aumentar.** Aumenta la distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 
  - **Disminuir.** Disminuye la distancia entre el grupo de controles seleccionados.

    El botón en la barra de herramientas es: 
  - **Quitar.** Elimina el espacio entre los controles de forma que quedan pegados.

    El botón en la barra de herramientas es: 
- **Centrar en el formulario.** Centra los controles seleccionados en el formulario.
  - **Horizontalmente.**

    El botón en la barra de herramientas es: 
  - **Verticalmente.**

    El botón en la barra de herramientas es: 
- **Orden.** Cambia el orden de los controles seleccionados.
  - **Traer al frente.** Pone los controles por delante de los demás del formulario. La combinación de teclado para esta opción es *Ctrl+J*.

    El botón en la barra de herramientas es: 
  - **Enviar hacia atrás.** Pone los controles detrás de los demás del formulario. La combinación de teclado para esta opción es *Ctrl+K*.

    El botón en la barra de herramientas es: 
- **Bloquear controles.** Una vez que los controles están situados de forma definitiva, al activar esta opción impide que puedan moverse hasta que vuelva a desactivarse.

  El botón en la barra de herramientas es: 

Menú Depuración

- Paso a paso por instrucciones. Ejecuta el código del programa línea a línea, si llega a la llamada a un procedimiento, el depurador se introduce en el código de ese procedimiento. La combinación de teclado para esta opción es *F8.*

  El botón en la barra de herramientas es: 

- **Paso a paso por procedimientos.** Es similar a la anterior opción, se ejecuta cada instrucción del programa, pero si llega a la llamada a un procedimiento, el depurador ejecuta el procedimiento por completo y se sitúa en la siguiente línea después de la llamada al procedimiento. La combinación de teclado para esta opción es *Mayús+F8.*

  El botón en la barra de herramientas es: 

- **Paso a paso para salir.** Ejecuta las líneas del procedimiento actual hasta situarse en la primera línea posterior a la llamada del procedimiento en donde nos encontrábamos. La combinación de teclado para esta opción es *Ctrl+Mayús+F8.*

  El botón en la barra de herramientas es: 

- **Ejecutar hasta el cursor.** Ejecuta las líneas del programa hasta el punto donde está situado el cursor. Si hubiera un punto de ruptura antes de llegar, la ejecución se pararía en ese punto. La combinación de teclado para esta opción es *Ctrl+F8.*

- **Agregar inspección.** Muestra el cuadro de diálogo Agregar inspección para introducir una expresión, que al ser evaluada en el transcurso del programa, hará que este entre en modo de depuración según el tipo de inspección que se haya definido.

  El botón en la barra de herramientas es: 

- **Modificar inspección.** Muestra la ventana de Editar inspección en el que podemos modificar o eliminar la inspección. La combinación de teclado para esta opción es *Ctrl+W.*

  El botón en la barra de herramientas es: 

- **Inspección rápida.** Abre la caja de diálogo en la que podemos ver el valor de la variable o expresión seleccionada, pudiendo agregarla a la lista de expresiones ya definidas para el programa. La combinación de teclado para esta opción es *Mayús+F9.*

El botón en la barra de herramientas es: 

- **Alternar puntos de interrupción.** Inserta o elimina un punto de interrupción en la aplicación. Un punto de interrupción es una línea de código marcada para que al llegar la ejecución del programa a ella, dicho programa entre en modo de depuración.

  Los puntos de interrupción son reconocibles porque suelen tener un color resaltado del resto del código. Sólo pueden definirse estos puntos para líneas que contengan código ejecutable, no se pueden usar con líneas de comentarios o declaración de variables, por ejemplo. La combinación de teclado para esta opción es *F9.*

El botón en la barra de herramientas es: 

- **Borrar todos los puntos de interrupción.** Elimina todos los puntos de interrupción en la aplicación. La combinación de teclado para esta opción es *Ctrl+Mayús+F9.*

  El botón en la barra de herramientas es: 

- **Establecer instrucción siguiente.** Esta opción hace que la ejecución se desplace al punto seleccionado por el usuario, sin ejecutar las líneas intermedias que pudiera haber entre la línea activa en ese momento y la seleccionada como instrucción siguiente.

  Se puede establecer una línea posterior a la actual o anterior (que ya se haya ejecutado). Esta característica no está disponible entre rutinas diferentes. La combinación de teclado para esta opción es *Ctrl+F9.*

  El botón en la barra de herramientas es: 

- **Mostrar instrucción siguiente.** Muestra resaltada la siguiente línea a ejecutar.

  El botón en la barra de herramientas es: 

Menú Ejecutar

- Iniciar. Ejecuta la aplicación desde dentro del IDE. La combinación de teclado para esta opción es *F5.*

  El botón en la barra de herramientas es: 

- **Iniciar con compilación completa.** Compila el proyecto antes de ejecutarlo. La combinación de teclado para esta opción es *Ctrl+F5.*

- **Interrumpir.** Detiene la ejecución del programa entrando en modo de interrupción. La combinación de teclado para esta opción es *Ctrl+Interrupción.*

  El botón en la barra de herramientas es: 

- **Terminar.** Finaliza la ejecución de la aplicación.

  El botón en la barra de herramientas es: 

- **Reiniciar.** Estando situados en el modo de interrupción, esta opción ejecuta la aplicación desde el principio. La combinación de teclado para esta opción es *Mayús+F5.*

Menú Consulta

- Ejecutar. En una ventana Ejecutar tabla, abierta desde Vista Datos, que muestra el contenido de una tabla o consulta, esta opción vuelve a ejecutar la consulta en que se basa dicha ventana, mostrando de nuevo los registros.

- Borrar resultados. Limpia el contenido de una ventana Ejecutar tabla, pero no borra los registros de la tabla, que permanecen en la fuente de datos.

- Comprobar sintaxis SQL. Verifica la instrucción SQL en que se basa una ventana de tipo Ejecutar tabla.

- Agrupar por. Es posible construir una consulta de agrupación en una ventana Ejecutar tabla, con la ayuda de esta opción. Una vez que hayamos seleccionado las columnas a agrupar, ejecutaremos esta opción, que añadirá a la instrucción SQL, la parte correspondiente a la partícula GROUP BY; la figura 94 muestra un ejemplo.

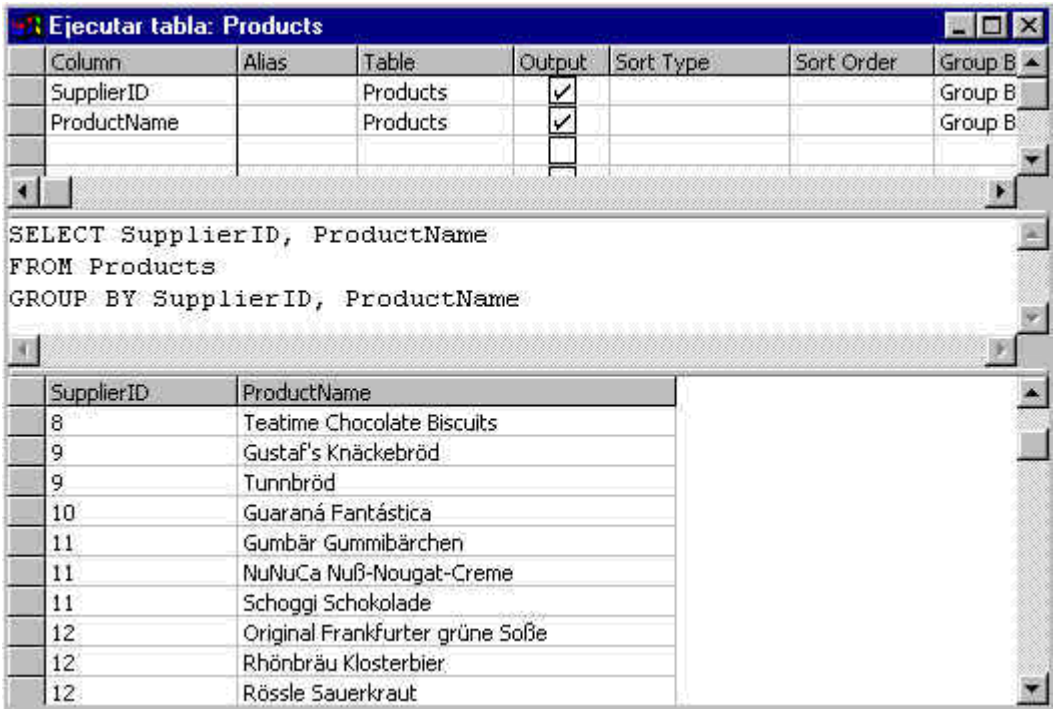


Figura 94. Crear un agrupado en una ventana Ejecutar tabla.

- **Cambiar tipo.** Este submenú modifica el tipo de instrucción incluida en una ventana Ejecutar tabla. Consta de las siguientes opciones:

    - **Seleccionar.** Proporciona una instrucción Select común para la tabla contenida en la ventana. Vemos un ejemplo en la figura 95.

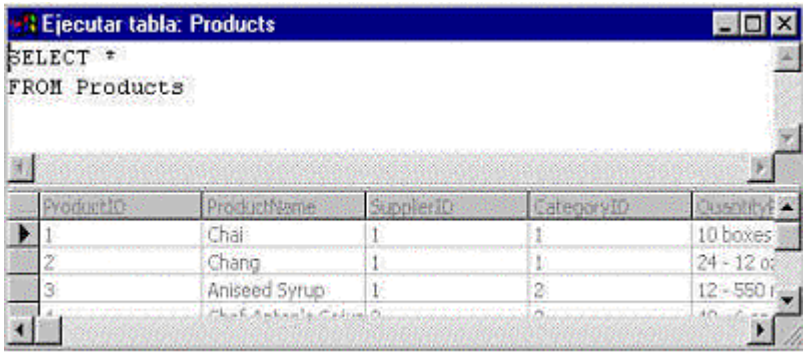


Figura 95. Crear una consulta de selección mediante la opción Cambiar tipo.

- **Insertar.** Cambia el tipo de instrucción a una de inserción en una tabla a partir de una selección de otra tabla. Vemos un ejemplo en la figura 96.

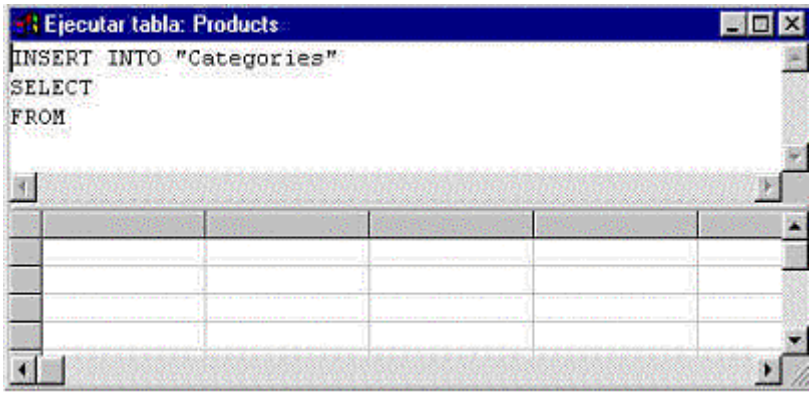


Figura 96. Crear una instrucción de inserción con la opción Cambiar tipo.

- **Insertar valores.** En este caso se crea una instrucción para añadir una fila a una tabla. Vemos un ejemplo en la figura 97.

Figura 97. Crear una instrucción para añadir una fila a una tabla.

- **Actualizar.** Aquí podemos modificar los valores de las columnas. Vemos un ejemplo en la figura 98.

Figura 98. Crear una instrucción para modificar las columnas de una tabla.

- **Eliminar.** Crea una sentencia SQL para borrar filas de la tabla. Vemos un ejemplo en la figura 99.

Figura 99. Crear una instrucción para eliminar filas de una tabla.

- **Crear tabla.** La sentencia generada en esta ocasión crea una tabla a partir de los datos de una existente en la base de datos. Vemos un ejemplo en la figura 100

Figura 100. Sentencia para crear una tabla a partir de otra existente.

- **Agregar resultados.** Actualiza el panel de resultados.
- **Orden ascendente, Orden descendente.** Ordena las filas en alguno de estos modos.
- **Quitar filtro.** Elimina el filtro que hubiera establecido sobre los registros.
- **Seleccionar todas las filas de <NombreTabla>.** Selecciona todos los registros de la tabla.

Menú Diagrama

- **Nueva anotación de texto.** Abre un cuadro en la ventana del diagrama de la base de datos para introducir una nota o comentario.

- **Establecer fuente de texto.** Muestra el cuadro de diálogo de selección de fuente para la nota de texto del diagrama que está seleccionada actualmente.

- **Agregar tablas relacionadas.** Visualiza en el caso de que no estén en el diagrama, las tablas relacionadas con la que está seleccionada actualmente.

- **Mostrar etiquetas de relaciones.** Junto al indicador gráfico de la relación, se muestra un texto con información sobre la relación.

- **Modificar vista personalizada.** Abre un cuadro de diálogo en el que se pueden establecer las columnas a mostrar en el diagrama para una tabla personalizada, como se muestra en la figura 101.

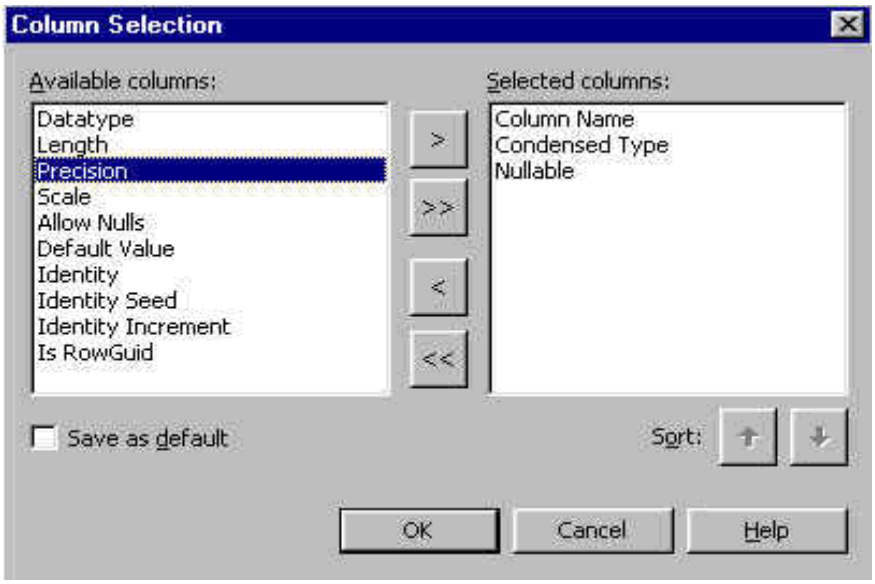


Figura 101. Seleccionar las columnas a mostrar en un diagrama de base de datos para una tabla.

- **Ver saltos de página.** Visualiza la disposición en páginas del diagrama.

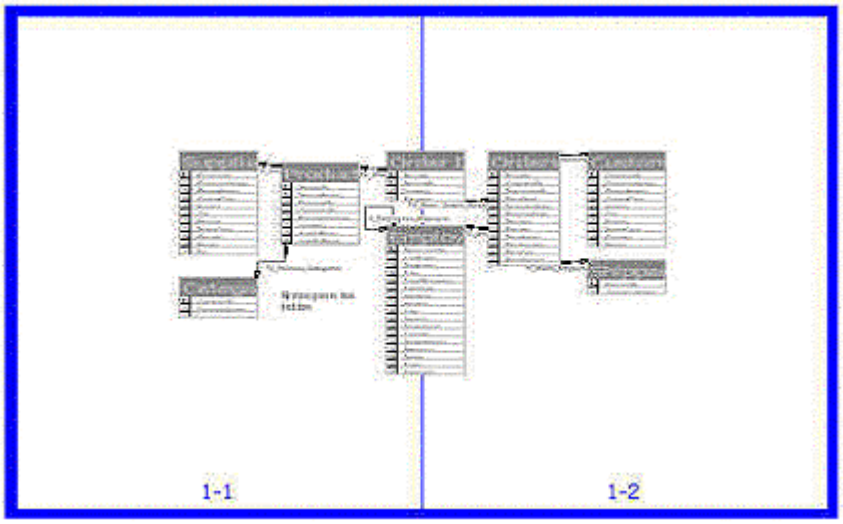


Figura 102. Diagrama de base de datos dispuesto en páginas.

- **Calcular saltos de página.** Distribuye los elementos del diagrama en diferentes páginas.
- **Organizar selección.** Ordena y sitúa las tablas seleccionadas en el diagrama.
- **Organizar tablas .** Ordena y sitúa todas las tablas del diagrama.
- **Ajustar automáticamente tablas seleccionadas.** Asigna un tamaño predeterminado a las tablas seleccionadas actualmente.

Menú Herramientas

- **Agregar procedimiento.** Añade en un módulo de código un procedimiento Sub, Function, Property o método de objeto. Esta opción construye el esqueleto del procedimiento, el añadir el código para que el procedimiento funcione de una u otra forma es tarea del programador.

  El botón en la barra de herramientas es: 

- **Atributos del procedimiento.** Visualiza un cuadro de diálogo para establecer diferentes características del procedimiento.
- **Editor de menús.** Abre la ventana del Editor de menús para poder crear o modificar el menú del formulario actual. El capítulo dedicado al diseño de menús, cubre todo lo referente al respecto. La combinación de teclado para esta opción es *Ctrl+E*.

  El botón en la barra de herramientas es: 

- **Opciones.** Muestra la ventana de Opciones, en la que se pueden configurar diversas características del entorno de programación. Las figuras 103 y 104 muestran esta ventana en los apartados correspondientes a Editor y Entorno respectivamente.

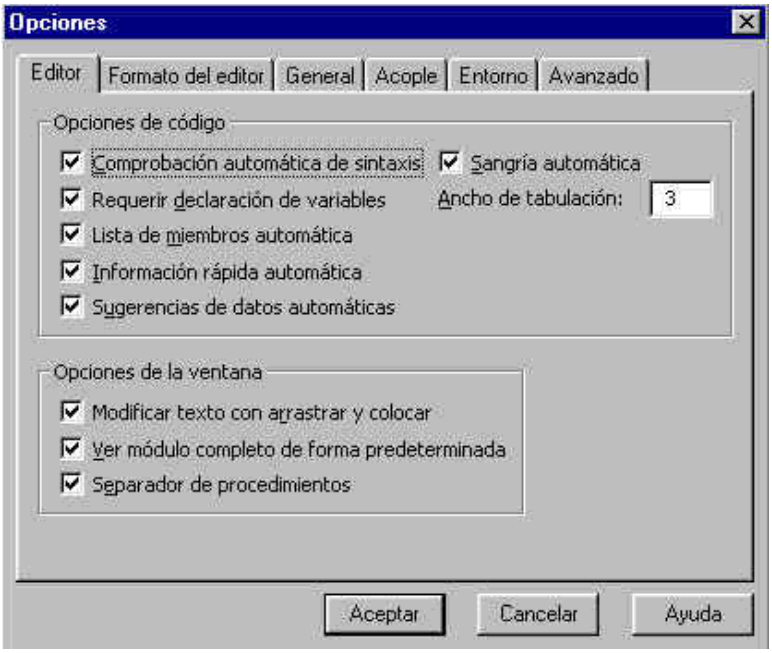


Figura 103. Opciones del IDE, pestaña Editor.

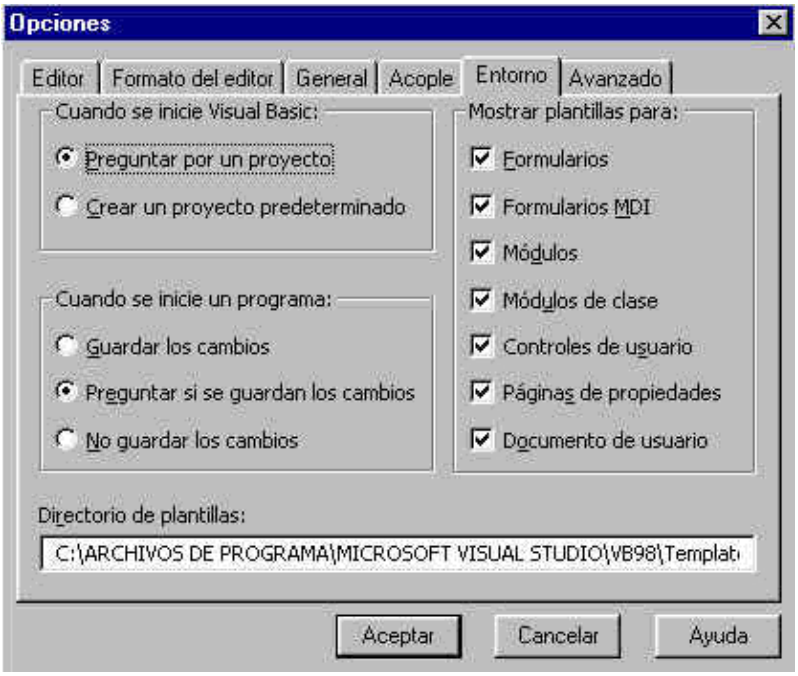


Figura 104. Opciones del IDE, pestaña Entorno.

Menú Complementos

- Administrador visual de datos. Inicia la aplicación VisData, que acompaña a Visual Basic 6 como complemento para el manejo de datos.

- Administrador de complementos. Abre el cuadro de diálogo que nos muestras los complementos instalados, y nos permite agregar o eliminar otros complementos.

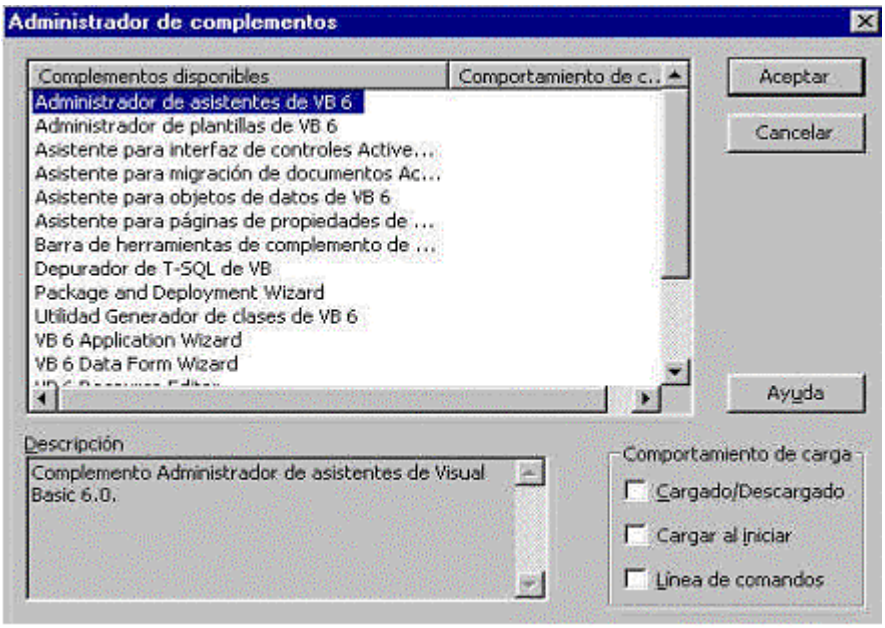


Figura 105. Administrador de complementos de VB.

Los complementos disponibles se encuentran en una lista, para incluirlos en el IDE, debemos seleccionar el componente que necesitemos y hacer clic en las casillas del Frame Comportamiento de carga; una vez cargado, aparecerá dicho componente añadido a las opciones de este menú. Vemos un ejemplo en la figura 106

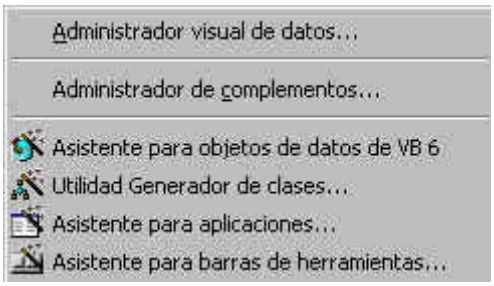


Figura 106 Menú Complementos con algunos complementos instalados en el entorno.

Menú Ventana

- Dividir. Separa mediante una barra, la ventana de edición de código en dos áreas, de forma que podemos ver dos procedimientos al mismo tiempo.

El botón en la barra de herramientas es: 

- **Mosaico horizontal.** Coloca las ventanas abiertas en el IDE en mosaico horizontal.

El botón en la barra de herramientas es: 

- **Mosaico vertical.** Coloca las ventanas abiertas en el IDE en mosaico vertical.

El botón en la barra de herramientas es: 

- **Cascada.** Coloca las ventanas abiertas en el IDE superpuestas en cascada.

  El botón en la barra de herramientas es: 

- **Organizar iconos.** Coloca las ventanas minimizadas en la parte inferior izquierda de la ventana del IDE.

- **Lista de ventanas.** Presenta los nombres de las ventanas abiertas.

Menú Ayuda

Visual Basic 6 incorpora para su sistema de ayuda la herramienta Microsoft Developer Network (MSDN), en sustitución de las clásicas ventanas de ayuda y Libros en pantalla. Esta herramienta está basada en HTML y proporciona características avanzadas y mejores opciones de búsqueda en la documentación disponible del producto.

- **Contenido.** Inicia MSDN y muestra su pantalla principal.

El botón en la barra de herramientas es: 

- **Índice.** Muestra la pestaña Índice en el panel de subconjuntos del MSDN, para realizar búsquedas por orden alfabético.

- Búsqueda. Muestra la pestaña Búsqueda en el panel de subconjuntos del MSDN, para realizar búsquedas en los temas que contengan la palabra que introduzcamos.

El botón en la barra de herramientas es: 

- **Soporte técnico.** Muestra los temas de ayuda relacionados con el soporte técnico del producto.

El botón en la barra de herramientas es: 

- **Microsoft en el Web.** Abre un submenú con direcciones de Internet donde existe información sobre Visual Basic en particular y Microsoft en general.

- **Acerca de Microsoft Visual Basic.** Abre la clásica caja de diálogo Acerca de..., con información breve sobre el producto.

Ventana de Proyecto

Esta ventana visualiza todos los componentes que forman parte del proyecto actual. Se puede trabajar con más de un proyecto al mismo tiempo con tan sólo agregarlo a esta ventana, de forma que pasaríamos a manejar un grupo de proyectos.

Los elementos del proyecto se clasifican dentro de una estructura jerárquica, en la que el nivel superior lo compone el proyecto, y los niveles inferiores están compuestos de carpetas que contienen los

formularios, clases, módulos de código, etc. Estas carpetas podemos abrirlas para mostrar su contenido o mantenerlas cerradas, como muestra la figura 107.

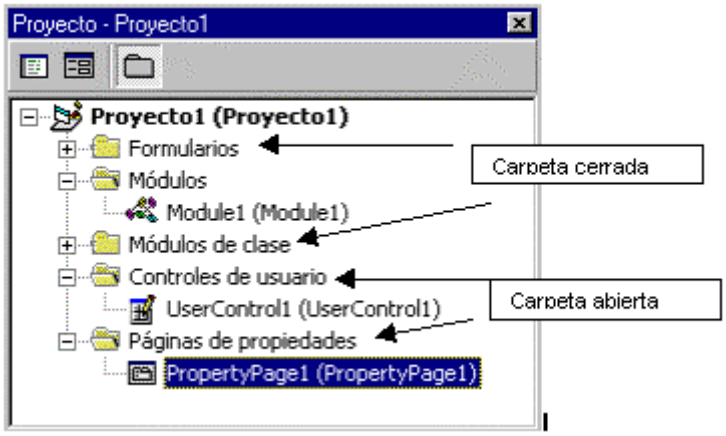


Figura 107. Ventana de Proyecto.

Aunque la organización por defecto es en carpetas, podemos mostrar los elementos del proyecto sin carpetas pulsando en el botón *Alternar carpetas*  de la barra de botones de la ventana.

Para ver la ventana de código del elemento seleccionado, pulsaremos el botón *Ver Código* 

Para ver la ventana de edición de objetos del elemento seleccionado, pulsaremos el botón *Ver Objeto* 

Dispone de un menú contextual al pulsar el botón derecho del ratón con opciones para agregar nuevos elementos al proyecto, guardar el objeto seleccionado en la ventana en disco, imprimir el proyecto, ver sus propiedades, etc.

Editor de Formularios

Esta ventana es la que utilizaremos para crear los formularios que servirán de interfaz entre el usuario y nuestra aplicación. Cada vez que incorporemos un nuevo formulario o seleccionemos uno existente desde la ventana de proyectos, se abrirá una nueva ventana de edición de formularios y dicho formulario se cargará en ella. Hecho esto, podemos modificar las propiedades del formulario, insertar un menú y controles en su área de trabajo.

Comencemos con el propio formulario; si queremos cambiar su tamaño, sólo hemos de hacer clic y arrastrar en alguno de los *controladores de desplazamiento*, dispuestos alrededor del objeto, como se muestra en la figura 108.

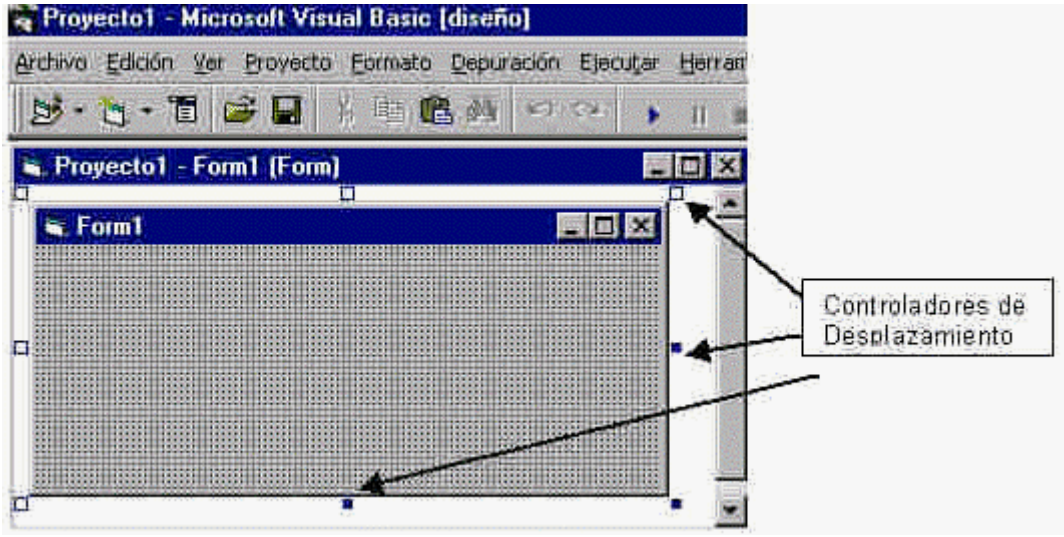


Figura 108. Editor de formularios con controladores de desplazamiento.

Abriendo su ventana de propiedades, podemos modificar entre otras las siguientes:

- Name. Nombre utilizado en el código para referenciar al formulario.
- Caption. Texto que aparece en la barra de título del formulario.
- Left, Top. Posición del formulario.
- Height, Width. Tamaño del formulario.
- ControlBox. Muestra/oculta el menú de control del formulario.
- MDIChild. En aplicaciones que disponen de una ventana MDI, esta propiedad indica si el formulario será mostrado como una ventana secundaria MDI, es decir, contenida dentro de la ventana principal MDI.
- BorderStyle. Selecciona el estilo del borde del formulario: fijo, dimensionable, ninguno, etc.

Insertar controles

Completado el diseño del formulario, abriremos el cuadro de herramientas para incluir algún control, cosa que podemos hacer de dos formas:

Haciendo doble clic en el control que necesitemos en el cuadro de herramientas, el control aparecerá en el centro del formulario. De esta manera, tenemos el inconveniente de que debemos situar el control en la posición correcta, ya que todos aparecen en el centro del formulario.

Hacer clic en el control del cuadro de herramientas y situar el cursor del ratón en la parte superior izquierda del formulario en donde queramos insertar el control; hacer clic en ese punto y arrastrar el ratón hacia la derecha y abajo para dibujar el control, cuando alcance el tamaño deseado, soltaremos el botón del ratón, con lo que aparecerá el control en la posición deseada. La excepción a esta regla reside en que algunos controles, al ser dibujados en el formulario toman un tamaño fijo que no puede ser variado; algunos de estos controles son Timer, CommonDialog, etc.

Una vez insertado el control, podemos abrir la ventana de propiedades para hacer cambios en las que sea necesario.

Modificar controles

Después de haber insertado un control, es posible que necesitemos cambiar el tamaño que originalmente le habíamos asignado. Para realizar esta labor, haremos clic en el control a cambiar, con lo que tomará el foco de la edición, podemos ver esto porque el control queda rodeado por sus controladores de desplazamiento, lo vemos en la figura 109. A partir de aquí, operaremos igual que con el formulario, haciendo clic en un controlador y arrastrando hasta conseguir el tamaño deseado, momento en el que liberaremos el botón del ratón.

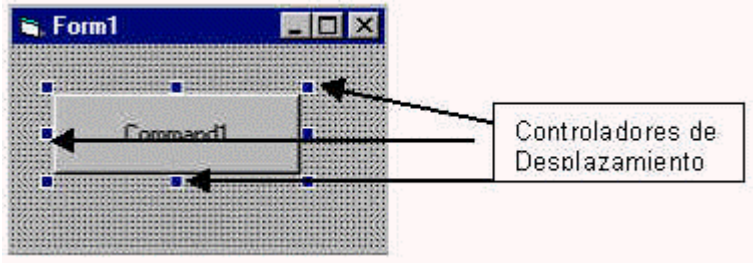


Figura 109. Control seleccionado.

Mover controles

Para cambiar de posición un control dentro del formulario, haremos clic en él y lo arrastraremos hasta su nueva situación, soltándolo en ese punto. También podemos hacerlo modificando sus propiedades Top y Left.

Seleccionar controles

Es recomendable seleccionar un grupo de controles cuando se tiene que realizar una operación común sobre ellos. Para seleccionar más de un control del formulario, mantendremos pulsada la tecla *Control* o *Mayúsculas* mientras hacemos clic en cada control a seleccionar, los controles seleccionados aparecerán señalados con sus controladores de desplazamiento en posición interior, como se puede apreciar en la figura 110.

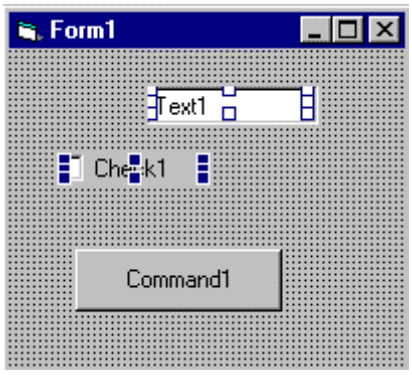


Figura 110. Formulario con algunos controles seleccionados.

Una vez terminada la selección, al hacer clic y arrastrar, se moverán todos en bloque. De igual forma, si abrimos la ventana de propiedades, sólo aparecerán las propiedades comunes a los controles seleccionados.

Cortar, Copiar y Pegar controles

Los controles seleccionados admiten las operaciones habituales de cortar, copiar y pegar. Por ejemplo, una vez copiados un conjunto de controles, podemos cambiar a otro formulario del proyecto y pegarlos allí.

Arrays de controles

Al copiar controles dentro de un mismo formulario, aparecerá un mensaje avisándonos que el control ya existe y que si deseamos crear un array de controles. Si respondemos afirmativamente se creará un array conteniendo un conjunto de controles del mismo tipo y nombre, diferenciados por su propiedad *Index*, que tendrá un número diferente para cada control. El código fuente 168 que vemos a continuación, cambia la propiedad Caption de un array de controles CommandButton.

```
Load frmControles
frmControles.cmdVarios(0).Caption = "el cero"
frmControles.cmdVarios(1).Caption = "el uno"
frmControles.cmdVarios(2).Caption = "el dos"
frmControles.Show
```

Código fuente 168

Manipulación global de controles

Bajo esta denominación, entendemos la capacidad de realizar una misma operación sobre un grupo de controles seleccionados. Si tenemos por ejemplo una ventana como la de la figura 111

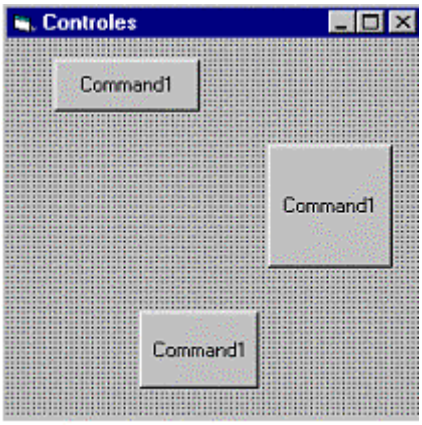


Figura 111. Controles para manipular de forma global.

entre las diversas operaciones a realizar, podemos seleccionar los controles del formulario y alinearlos a la izquierda del mismo, de forma que quedaran como en la figura 112.

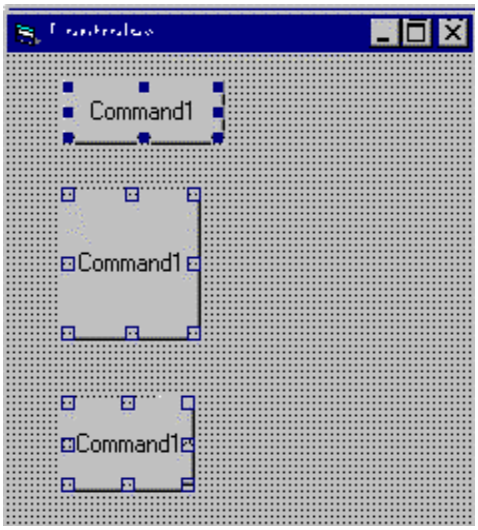


Figura 112. Controles alineados a la izquierda.

También podemos hacer que aumente el espacio vertical entre los controles, como aparece en la figura 113, o igualar el tamaño entre los controles, como muestra la figura 114.

Una vez que tenemos los controles terminados, podemos bloquearlos para no cambiar ninguna parte de su aspecto accidentalmente.

Para más información sobre las operaciones globales disponibles para controles seleccionados, recomendamos al lector consulte la parte dedicada al menú Formato, donde se describen las opciones existentes.

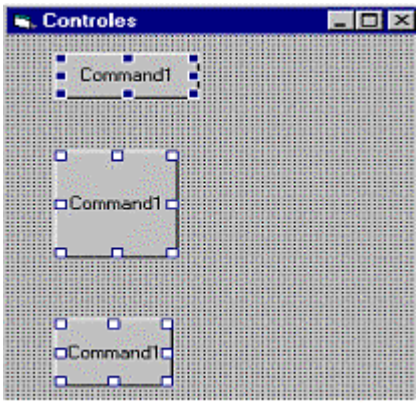


Figura 113. Controles con separación vertical.

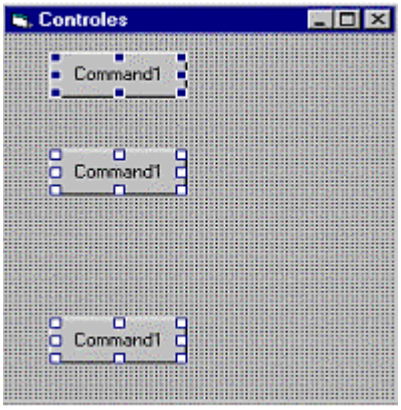


Figura 114. Controles con tamaño igualado.

Formularios MDI

Todo lo anteriormente comentado sobre formularios tiene una excepción en los formularios MDI (figura 115), ya que estos tienen alguna variación sobre los formularios estándar. Sólo puede existir un MDI por aplicación, que actúa como contenedor del resto de formularios de la aplicación, llamados secundarios MDI. Podemos distinguir un formulario MDI del resto en que su área de trabajo aparece en un tono diferente. En cuanto a los controles, sólo puede usar algunos como Menú, Timer y CommonDialog.

Figura 115. Editor de formularios con ventana MDI.

Editor de Código

En esta ventana podemos editar el código de los diferentes elementos del proyecto: eventos de formularios, controles, procedimientos, funciones, clases, etc. Es posible abrir tantas ventanas de código como sea necesario, pudiendo compartir código entre ellas mediante las acciones estándar de Windows de cortar, copiar y pegar.

En función del elemento que deseemos codificar, esta ventana tendrá unas particularidades determinadas; así un módulo de código de un formulario, contendrá los métodos para los eventos del objeto formulario y controles, además de los procedimientos normales de código; un módulo estándar de código tendrá sólo funciones y procedimientos; un módulo de clase tendrá procedimientos de propiedades, la definición de la clase, métodos, etc.

Podemos abrir el Editor de Código desde la Ventana de Proyecto, seleccionando un elemento del proyecto y pulsando en el botón *Ver código*, o mediante el menú contextual.

Si nos encontramos en el Editor de Formularios, haremos doble clic sobre el mismo o sobre un control, lo que abrirá el Editor de Código y además nos situará en el método por defecto correspondiente al objeto sobre el que hayamos pulsado.

Aparte del área de edición, como parte de la ventana donde escribimos el código de la aplicación, el Editor de Código está compuesto de las siguientes partes:

Lista de objetos

Muestra el objeto actualmente en edición; si desplegamos la lista, aparecerá una relación de los objetos disponibles en el módulo para ser codificados; en la figura 116 se muestra el módulo de código de un formulario con esta lista abierta y los objetos disponibles.

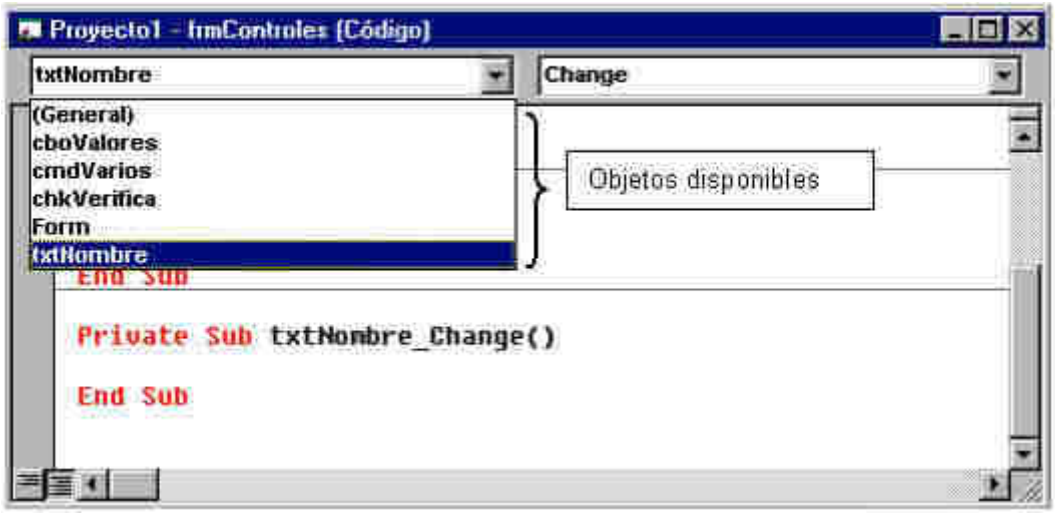


Figura 116. Editor de Código mostrando lista de objetos.

Lista de procedimientos

Muestra los procedimientos disponibles para el objeto que aparece actualmente en la lista de objetos. En la figura 117, se muestra esta lista abierta con algunos de los métodos para un control CommandButton.

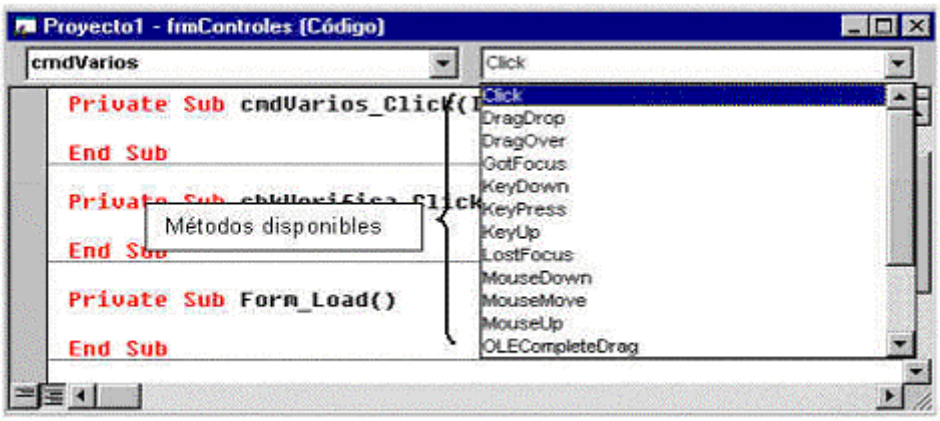


Figura 117. Editor de Código mostrando lista de procedimientos.

Para introducir código en un control hemos de tener claro en primer lugar que todos los objetos en Windows (formularios, controles, etc.) están conducidos por eventos. Cuando se produce un evento en un control, VB busca si para ese evento el objeto control dispone de un método codificado, en caso afirmativo, ejecuta el método, de otra forma efectúa las acciones por defecto que para ese control hay programadas en Windows; es decir, cuando necesitemos que un control realice una tarea bajo cierta circunstancia/evento, tendremos que buscar en la lista de procedimientos, el que se va a desencadenar por la circunstancia/evento antes mencionada y codificarlo.

Para acceder a las rutinas que no pertenecen propiamente a ningún control ni al formulario del módulo, seleccionaremos de la lista de objetos del módulo el elemento *General*; una vez hecho esto, al abrir la lista de procedimientos accederemos a las rutinas de uso común para el formulario o aplicación. Igualmente, si seleccionamos el elementos *Declaraciones* de la lista de procedimientos, tendremos acceso al código situado fuera de los procedimientos del módulo, tal como declaraciones de variables a nivel de módulo, tipos definidos por el usuario, etc., como vemos en la figura 118.

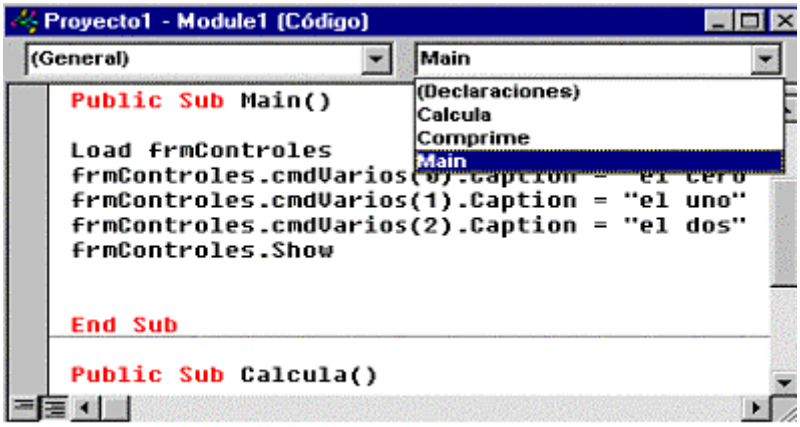


Figura 118. Módulo de código situado en la zona General

Esto último también es aplicable para los módulos de código estándar, puesto que sólo tienen un elemento General en su lista de objetos, perteneciendo a esta sección todos los procedimientos del módulo.

Barra de división

Separa la ventana de edición de código en dos áreas, de forma que podamos tener dos vistas del código que actualmente estamos editando, como aparece en la figura 119. En una podemos tener el procedimiento que estamos escribiendo, y en la otra podemos ver otro procedimiento del módulo, que necesitemos consultar a menudo.

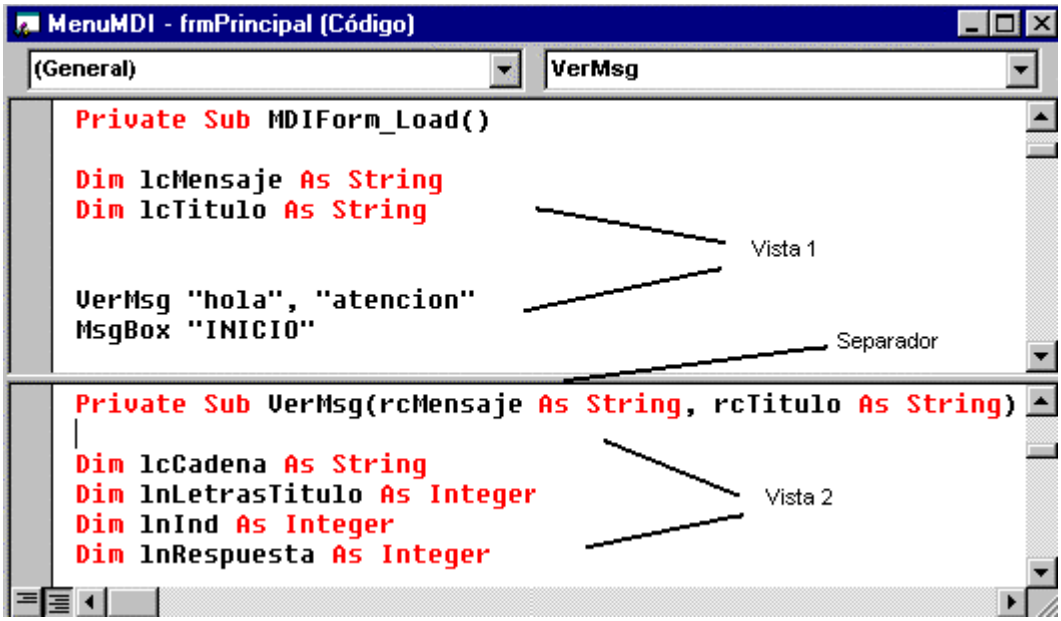


Figura 119. Ventana de Código con barra de división.

Visualizar el código

Si pulsamos el botón  situado en la parte inferior izquierda de la ventana, se mostrará un único procedimiento en la ventana. Si pulsamos en cambio el botón que se encuentra al lado  , se mostrará el código del módulo al completo.

Marcadores y barra de margen

El Editor de Código dispone en su parte izquierda de una barra indicadora en la que podemos situar marcas para seguir la pista de diferentes procesos. La tabla 45 muestra las marcas disponibles.

Los marcadores de línea son unos elementos muy útiles durante el proceso de escritura del código. Sirven para señalar puntos determinados dentro de un procedimiento muy largo o puntos diferentes en procedimientos y módulos distintos. De esta forma si estamos por ejemplo editando código en tres procedimientos distintos y en ciertas líneas de código de esos procedimientos, al tener marcadores para esas líneas, con los botones de la barra de Edición destinados al uso de marcadores, nos desplazaremos rápidamente entre esas líneas y procedimientos sin tener que buscar repetidamente entre todo el código de un módulo o en un procedimiento muy extenso. En la parte dedicada al menú Edición, opción Marcadores, de este mismo capítulo puede consultar el lector las opciones para situar y usar marcadores en el código del programa.

Marca	Denominación	Operación que realiza
	Punto de interrupción	Se puede insertar un punto de interrupción haciendo clic en la barra de margen.
	Línea de ejecución actual	Indica la línea de código que se ejecutará a continuación. También es posible arrastrar este marcador a una línea diferente de código.
	Marcador	Indica la situación de un marcador.
	Marcador de pila de llamadas	Informa de las líneas que se encuentran en la pila de llamadas. Este marcador sólo está disponible en modo de interrupción.

Tabla 45. Marcas disponibles para la ventana de código.

Configuración del Editor de Código

Mediante la ficha *Editor* del cuadro de diálogo *Opciones* (menú de VB H*erramientas+Opciones*), podemos configurar diversos aspectos de la edición del código y de la ventana del editor como muestra la figura 120.

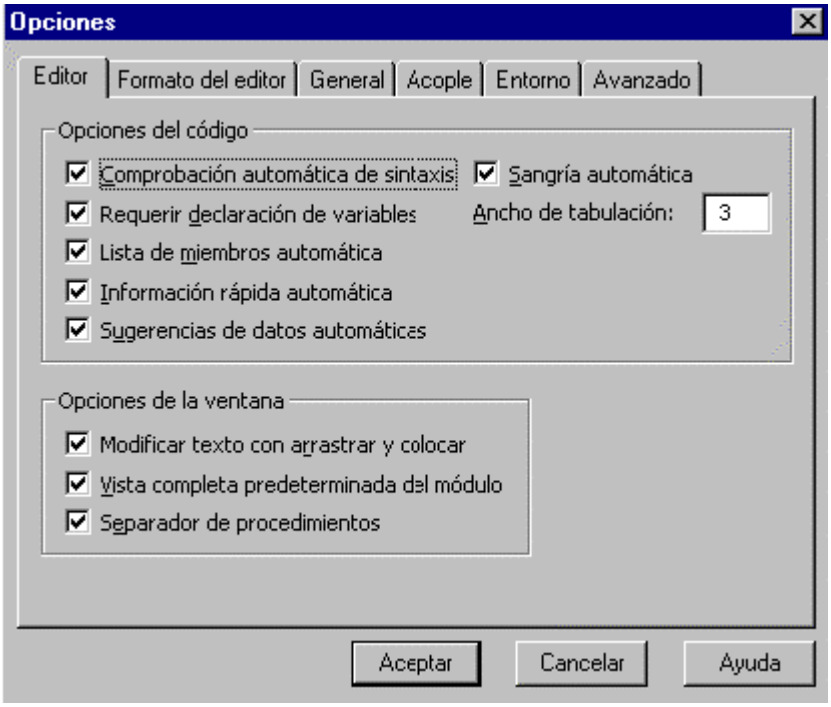


Figura 120. Opciones de configuración para el Editor de Código.

En lo que respecta al código, disponemos de las siguientes características:

- Comprobación automática de sintaxis. Comprobará después de escribir cada línea de código, avisándonos de la existencia de errores. En la figura 121 se muestra como en una línea

que se esperaba una asignación a una variable, se produce un error al pasar a la siguiente línea sin completar la anterior.

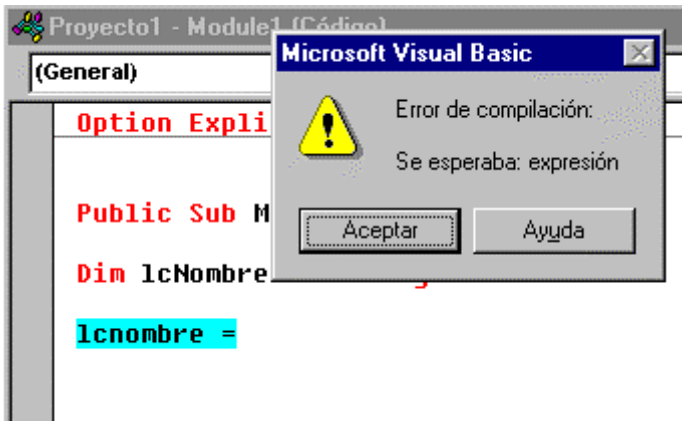


Figura 121. Error de sintaxis.

- Requerir la declaración de variables. Al crear un nuevo módulo, se agrega automáticamente en él, la instrucción Option Explicit.

- Lista de miembros automática. Visualiza una lista de miembros, constantes o valores disponibles para un parámetro de función, procedimiento, instrucción, etc. La figura 122 muestra la función MsgBox() con una lista de constantes disponibles para el parámetro de botones.

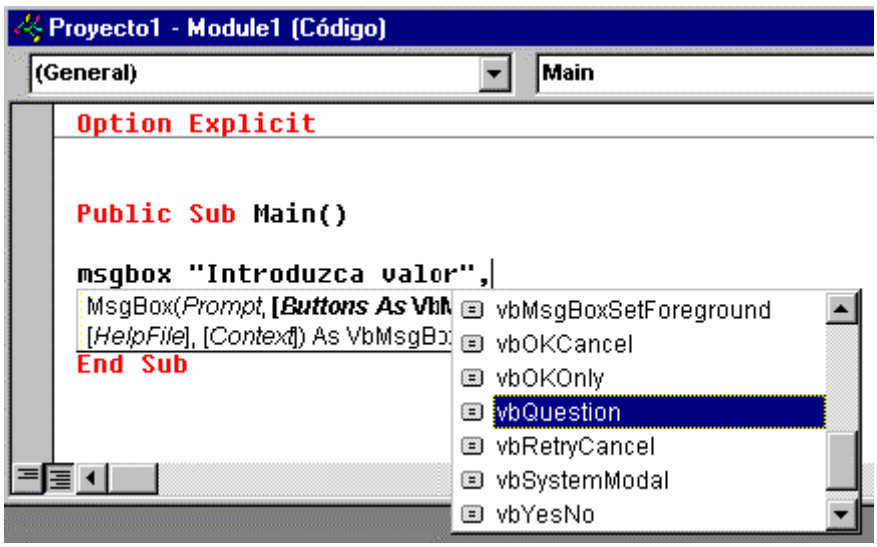


Figura 122. Función mostrando lista de miembros para un parámetro.

- Información rápida automática. Proporciona datos sobre los parámetros y orden de los mismos a utilizar en funciones y procedimientos. La figura 123 muestra los parámetros para la función InStr().

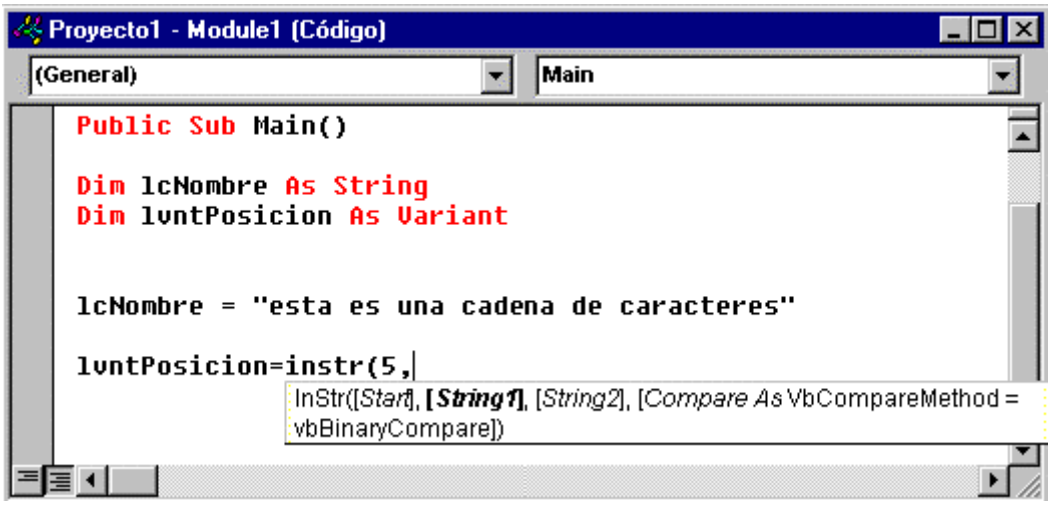


Figura 123. Información sobre la función InStr().

- Sugerencias de datos automáticas. Proporciona información sobre variables y expresiones. En modo de depuración, si colocamos el cursor sobre una variable, nos informará de su valor. Igualmente, si seleccionamos una expresión, esta será evaluada mostrando el resultado al situar el cursor sobre ella, todo ello mediante una pequeña etiqueta flotante, igual que la mostrada en la figura 124.

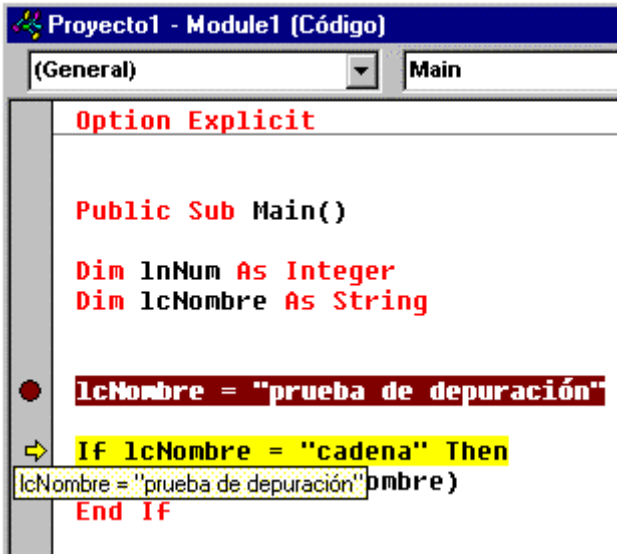


Figura 124. Información del contenido de una variable en modo de depuración.

- Sangría automática. Permite después de haber incluido una tabulación en una línea, que las líneas siguientes comiencen en esa posición.

- Ancho de tabulación. Se puede definir con intervalo de 1 a 32 espacios.

Como aspectos destacables para la ventana, tenemos los siguientes:

- Modificar texto con arrastrar y colocar. Permite transferir código a las ventanas Inmediato e Inspección.

- Vista completa predeterminada del módulo. Configura la ventana para poder ver todos los procedimientos del módulo seguidos o cada procedimiento por separado.

- Separador de procedimientos. Visualiza u oculta la línea de separación que divide el código de los procedimientos.

Configuración de formato del Editor de Código

Mediante la ficha *Formato del editor* del cuadro de diálogo *Opciones* (menú de VB *Herramientas+Opciones*), podemos configurar el aspecto que mostrará el código. En la figura 125 podemos ver esta ficha de configuración.

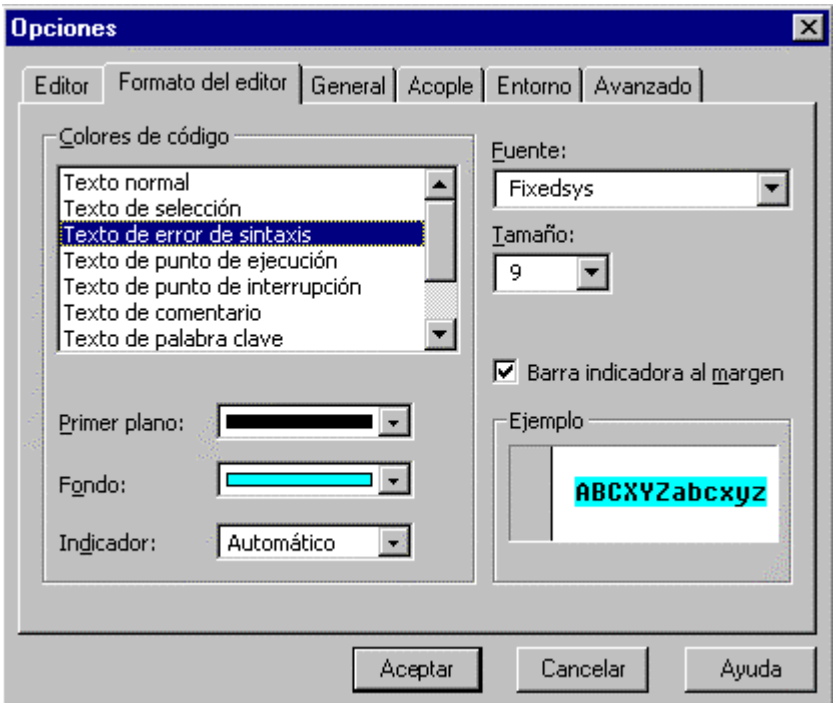


Figura 125. Opciones de configuración para el Formato del editor.

En el grupo *Colores de código* vemos una lista con los elementos del código a los que se puede cambiar el color, debajo están situadas unas listas desplegables para el color de primer plano, fondo y el indicador si es que dispone para el elemento de código seleccionado.

Por otro lado disponemos de otras dos listas desplegables para seleccionar el fuente y tamaño con el que será mostrado el código, y una casilla para mostrar u ocultar la barra de marcadores lateral en la ventana de código.

Ventanas complementarias

Podemos considerar como ventanas complementarias, las que sirven de apoyo dentro del entorno de VB a las ventanas principales de proyecto, edición de formulario y código fuente. Vamos a dar un repaso a estas ventanas, viendo algunas de sus principales características.

Ventana de Propiedades

Muestra una ventana con información sobre las propiedades del objeto actualmente seleccionado en el entorno de VB. Mientras esta ventana esté abierta, cada vez que se seleccione un elemento en el IDE, la Ventana de Propiedades reflejará las adecuadas para ese objeto, vemos un ejemplo en la figura 126.

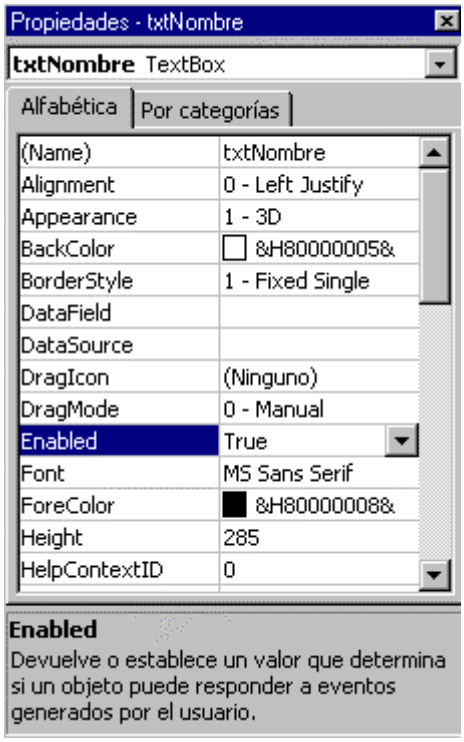


Figura 126. Propiedades de un control de formulario.

En la parte superior debajo del título, disponemos de una lista desplegable que contiene todos los objetos insertados en el formulario, de forma que podamos abrirla y seleccionar rápidamente uno.

Las propiedades pueden organizarse por orden alfabético o por categorías, pudiendo estas últimas comprimirse o expandirse para ver las propiedades contenidas en cada categoría. Para ver una u otra forma de organización sólo hemos de pulsar la ficha destinada a tal efecto.

La parte inferior de esta ventana puede mostrar una breve descripción de la propiedad seleccionada actualmente. Esta característica puede activarse mediante el menú contextual de la ventana.

Cuadro de herramientas

Visualiza una caja que contiene los controles disponibles para utilizar en los formularios de la aplicación, veámoslo en la figura 127.

Figura 127. Cuadro de herramientas.

Entre los controles más habituales en nuestro trabajo cotidiano con VB tenemos los siguientes:

- CommandButton. Representa un botón que produce el efecto de pulsado al hacer clic en él con el ratón (figura 128).

Figura 128. CommandButton

- Label. Este control no realiza ninguna acción, sirve para mostrar al usuario una cadena de caracteres, usualmente para informar de la función que desempeña otro control. En el cuadro de herramientas aparece como en la figura 129.

Figura 129. Label

- TextBox. Se trata de una caja donde se puede introducir texto, como se muestra en la figura 130.

Figura 130. TextBox

- CheckBox. Muestra una pequeña casilla en la que al hacer clic aparece una marca de verificación, figura 131, y al volver a pulsar desaparece la marca.

Figura 131. CheckBox

- OptionButton. Este botón tiene significado cuando se usa en conjunción con otros varios de su tipo. Al hacer clic en él, aparece o desaparece el punto de selección (figura 132). Su finalidad es dar al usuario la posibilidad de seleccionar una opción entre varias disponibles.

Figura 132. OptionButton

- ListBox. Muestra una lista de valores, de los que el usuario puede seleccionar uno, en el cuadro de herramientas aparece como en la figura 133.

Figura 133. ListBox

- ComboBox. Combinación de caja de texto y lista de valores, su apariencia se muestra en la figura 134.

Figura 134. ComboBox

- Data. Conecta y permite navegar por una tabla de una base de datos (figura 135).

Figura 135. Data

- DbGrid. Muestra el contenido la tabla conectada a un control Data del formulario (figura 136).

Figura 136. DbGrid

- OLE. Permite usar objetos de otras aplicaciones en nuestro proyecto en el cuadro de herramientas se muestra como en la figura 137.

Figura 137. OLE

Adicionalmente a todos los controles que vienen en esta caja, podemos incluir controles adicionales de la siguiente forma:

- Seleccionar la opción del menú de VB *Proyecto+Componentes*, que abrirá la ventana de Componentes, como se muestra en la figura 138.

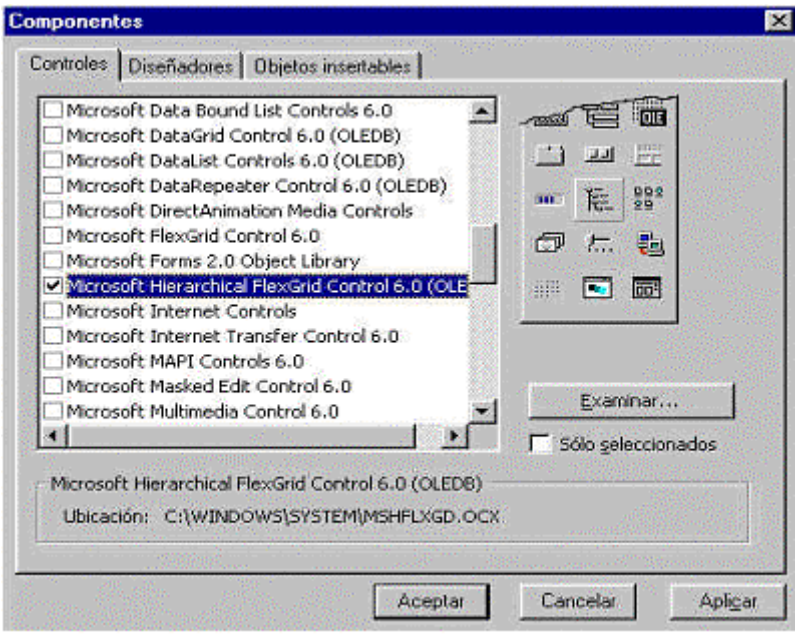


Figura 138. Selección de un componente para añadir a la caja de herramientas.

- Elegir un componente y marcarlo. En este caso vamos a seleccionar el control Microsoft Hierarchical FlexGrid.

- Pulsar Aceptar y ya lo tendremos incorporado a la caja de herramientas, como vemos en la figura 139.

Figura 139

Los controles de la caja vienen agrupados en *fichas*, siendo la ficha General la que aparece por defecto. Podemos crear nuevas fichas que contengan controles personalizados, o incluir en ellas parte de los controles existentes, si deseamos llevar otro tipo de organización de los mismos.

Para crear una ficha personalizada en la caja de herramientas, debemos seguir los siguientes pasos:

- Hacer clic en la caja de herramientas con el botón derecho del ratón. Se abrirá un menú contextual del que seleccionaremos la opción *Agregar ficha*.

- Se abrirá un cuadro de diálogo solicitando el nuevo nombre de la ficha, como aparece en la figura 140.

- Introducimos el nombre de la ficha y aceptamos la ventana, con lo que tendremos una nueva ficha en la caja de herramientas como se muestra en la figura 141.

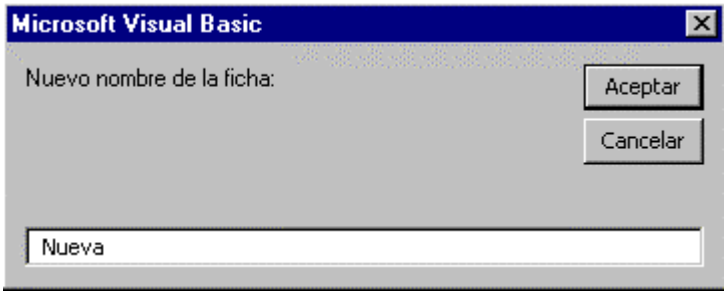


Figura 140. Agregar ficha a la caja de herramientas.

Figura 141. Nueva ficha creada en la caja de herramientas.

- Para abrir la ficha, sólo tenemos que hacer clic en su nombre.
- Por defecto, las nuevas fichas contendrán el icono del Puntero. Para incorporar nuevas herramientas en estas fichas, procederemos de la forma vista anteriormente o arrastrando y soltando herramientas de una ficha existente a la nueva.

Examinador de objetos

Esta ventana muestra todos los objetos que forman parte del proyecto y los que están disponibles en las librerías de Visual Basic para ser usados. La figura 142 muestra una imagen de esta ventana.

Los componentes de esta ventana son los siguientes:

- Lista Proyecto/Librería. Muestra el proyecto o grupo de proyectos actual y las librerías disponibles.
- Lista/Cuadro de Texto de búsquedas. Permite introducir un texto a buscar entre los objetos y muestra las últimas búsquedas realizadas.
- Retrocede o avanza en las búsquedas realizadas dentro del examinador. 
- Copia en el Portapapeles el elemento seleccionado actualmente en el examinador. 
- Abre la ventana de código fuente para ver el código del elemento seleccionado. 
- Muestra la ayuda si hay disponible para el elemento seleccionado. 

- Realiza una búsqueda del texto existente en el cuadro de texto de búsquedas, mostrando el resultado en la lista *Resultados de la búsqueda.*

- Muestra/oculta la lista de resultados de búsqueda.

- Lista Resultados de la búsqueda. Muestra la librería, clase y miembro correspondientes al elemento buscado.

- Lista de Clases. Visualiza todas las clases disponibles para el elemento que aparece en la lista Proyecto/Librería.

- Lista de Miembros. Visualiza los miembros que corresponden al elemento seleccionado en la lista Clases.

- Detalles. Muestra la definición del miembro actualmente seleccionado en la lista de miembros, permitiendo el salto a los miembros de la clase padre si existen.

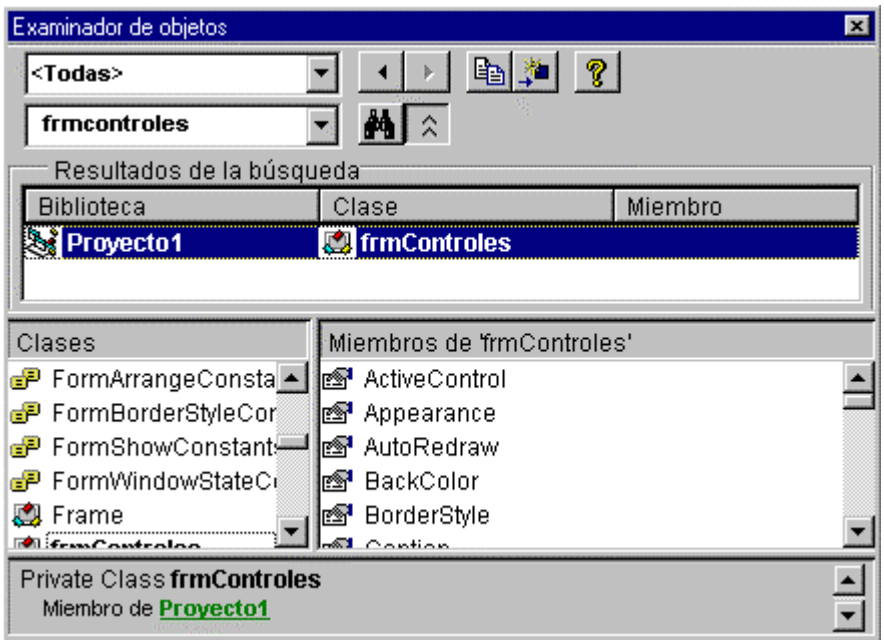


Figura 142. Examinador de objetos.

Una muestra de la utilidad del Examinador de objetos podría ser la siguiente:

Supongamos que necesitamos utilizar en un procedimiento el código de la tecla *Enter*, podemos utilizar su código numérico directamente que es el 13 o la constante que hay definida en VB, lo cual es mucho más recomendable. El inconveniente es que no recordamos muy bien como es esa constante, sólo sabemos que su nombre es algo parecido a *key*...

Abrimos entonces el Examinador de objetos, en la lista/cuadro de búsquedas tecleamos *key* y hacemos clic en el botón Búsqueda ; el Examinador buscará todas las referencias similares y presentará una ventana similar a la de la figura 143.

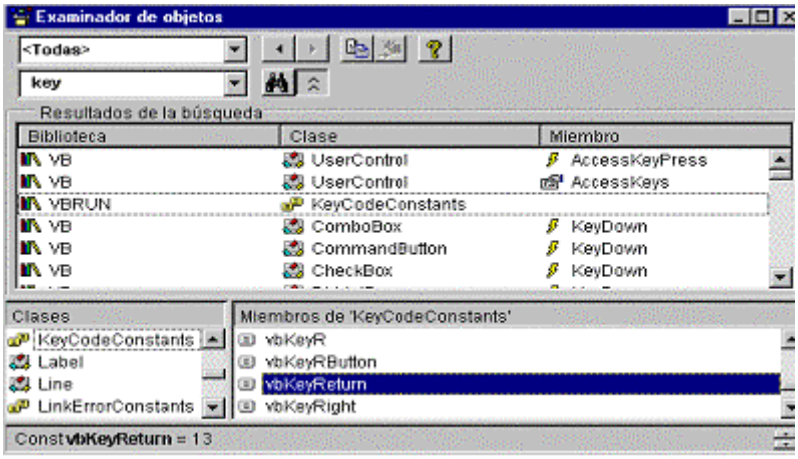


Figura 143. Resultado de una búsqueda en el Examinador de objetos.

En la lista *Resultados de la búsqueda* aparecerán todas las referencias a *key*, una de ellas es la biblioteca VBRUN que contiene la clase KeyCodeConstants. Al hacer clic en esta línea, aparecerá la clase en la lista Clases, y todos sus miembros en la lista Miembros, que son todas las constantes de teclado definidas por VB, buscamos vbKeyReturn que es la que corresponde a la tecla Enter, la copiamos al portapapeles y de aquí a nuestro programa.

Ventana de Posición del formulario

Permite situar de forma visual los formularios en la pantalla sin necesidad de ejecutar la aplicación repetidas veces y ajustar gradualmente las propiedades de posición de los formularios, como muestra la figura 144.

Figura 144. Ventana de Posición del formulario.

Para cambiar la posición de un formulario, sólo hemos de situar el cursor del ratón sobre el formulario a cambiar, hacer clic y arrastrar hasta la nueva posición.

Paleta de colores

Mediante esta ventana es posible cambiar la configuración de colores para un formulario o control, seleccionando el objeto a modificar y pulsando en alguno de los colores disponibles, como muestra la

figura 145. Podemos también devolver la configuración de colores original, pulsando el botón *Predeterminado.* Igualmente es posible crear colores personalizados pulsando el botón *Definir colores...* , el cual mostrará una ventana para realizar nuestras propias mezclas de color.

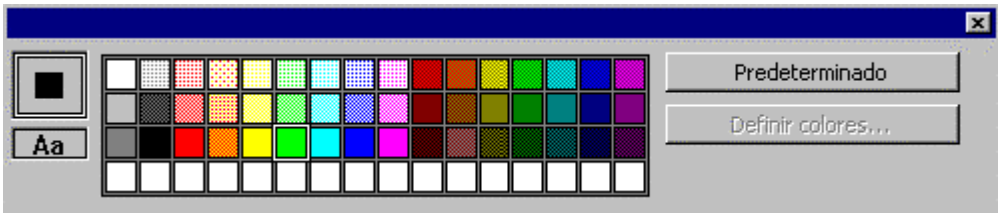


Figura 145. Paleta de colores de VB.

En el recuadro de muestra de colores, ver figura 146, se muestran los colores de primer plano y fondo para el objeto actualmente seleccionado. Si pulsamos el recuadro interior, los cambios de color se efectuarán en el color de primer plano, y pulsamos en el recuadro exterior, los colores afectados serán los de fondo.

Figura 146. Muestra de colores de la paleta.

Otra forma de definir los colores para los objetos es mediante las propiedades *BackColor* y *ForeColor* de la Ventana de Propiedades.

Anclaje de ventanas

Esta es la posibilidad de algunas de las ventanas del IDE para acoplarse a un lateral de la pantalla, quedando pegada o conectada a dicho lateral; tan solo hemos de arrastrarla hacia el lado en donde queramos acoplarla y soltar. Para reconocer una ventana con esta característica, hemos de abrir su menú contextual, si dicho menú dispone de la opción *Acoplable*, estamos ante una ventana de este tipo.

Igualmente las ventanas de este tipo pueden acoplarse entre sí, fundiéndose en una única ventana, como vemos en la figura 147.

Las barras de botones disponen de una cualidad similar, arrastrándolas hacia el menú principal de VB, se pegarán a este disponiéndose en filas.

Para despegarlas sólo hemos de hacer clic en la doble barra de su parte izquierda  y arrastrar, con lo nuevamente volverán a convertirse en una barra flotante.

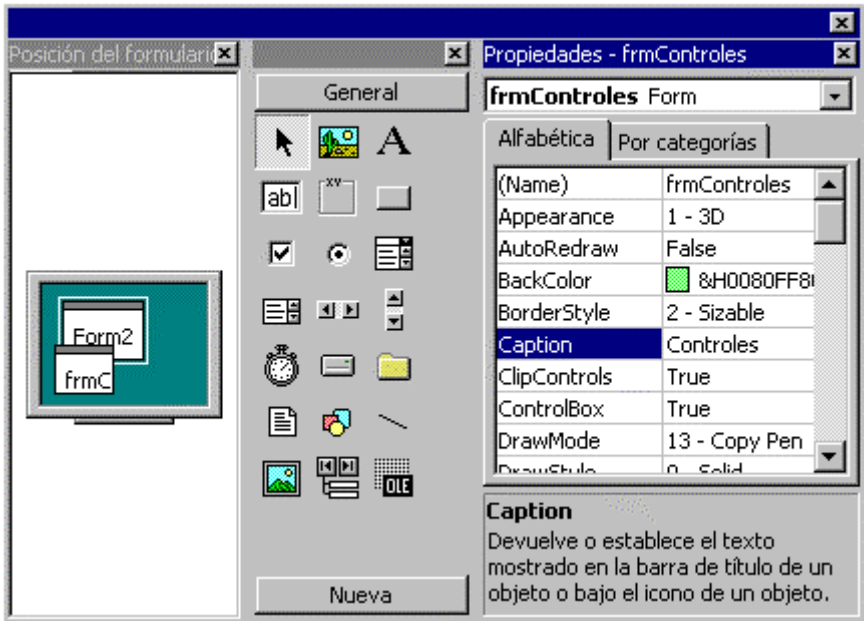


Figura 147. Ventanas de Posición de formulario, Herramientas y Propiedades acopladas.

Personalizar las barras de herramientas

Hemos visto en el apartado dedicado a los menús de VB, las diferentes barras de herramientas disponibles. Ahora vamos a ver como personalizarlas para adaptarlas mejor a nuestro modo de trabajo.

Seleccionando la opción de menú de VB *Ver+Barras de herramientas* aparecerá un submenú con las barras disponibles; nosotros elegiremos la opción *Personalizar* que abrirá una caja de diálogo para esta tarea como la de la mostrada en la figura 148.

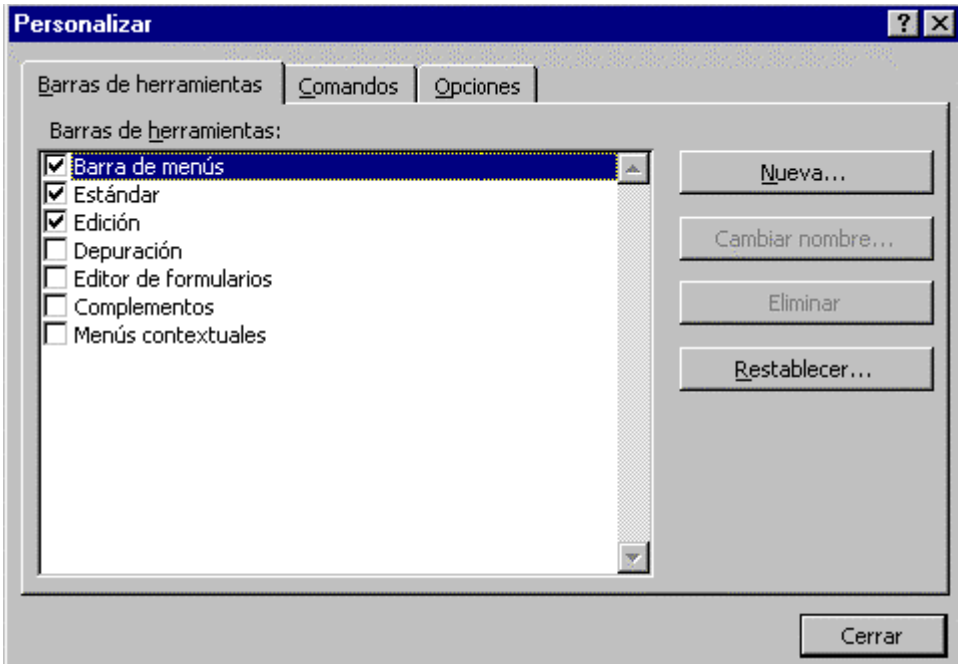


Figura 148. Cuadro de diálogo para personalizar las barras de herramientas.

A continuación pulsaremos en el botón *Nueva* para crear una barra de herramientas propia, en la que vamos a incluir algunas de las acciones que realizamos más habitualmente.

Se mostrará otra caja de diálogo para incluir el nombre de nuestra barra de herramientas, como muestra la figura 149.

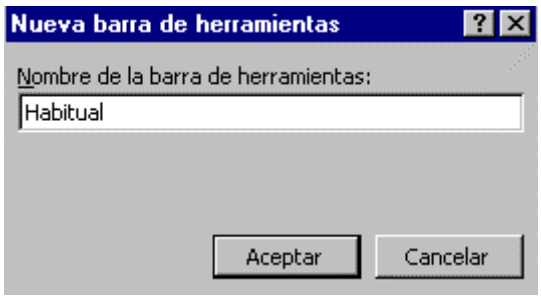


Figura 149. Caja de diálogo para nombrar una nueva barra de herramientas.

Aceptando esta última caja se creará una nueva barra de herramientas vacía. A continuación pulsaremos la ficha *Comandos* de la ventana *Personalizar*, que nos mostrará los comandos del IDE disponibles organizados por categorías, como aparece en la figura 150.

Hacemos clic en el comando *Bloque con comentarios* y lo arrastramos hasta la nueva barra de herramientas, con lo que ya tendremos un primer botón en esta barra. Repetimos la acción con algunos comandos más obteniendo como resultado la nueva barra personalizada (figura 151), que podremos utilizar como cualquiera de las que acompañan *de fábrica* a VB.

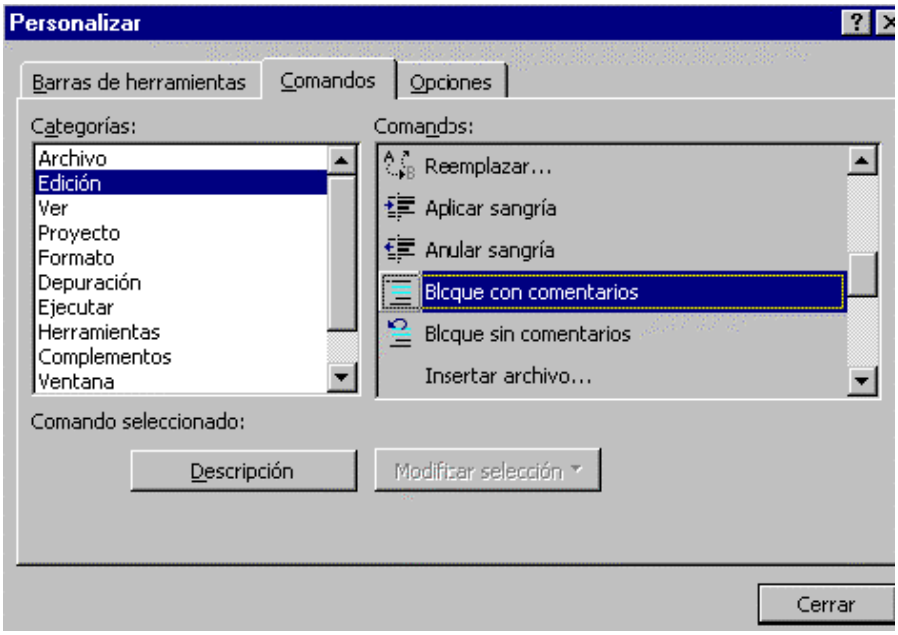


Figura 150. Comandos del tipo Edición.

Figura 151. Nueva barra de herramientas personalizada.

El botón *Guardar <fichero>*, situado en el lado derecho de la barra no tiene por defecto ninguna imagen asociada (podemos comprobarlo mirando esta misma opción en el menú *Archivo* de VB), pero una vez insertado en la barra, al seleccionarlo y abrir el menú contextual de la barra, dispondremos de una opción que permitirá añadirle una imagen de un conjunto disponible.

Durante esta fase de edición de los elementos de una barra de herramientas, podremos eliminarlos, cambiarlos de posición, mostrar el texto que acompaña al comando, etc.

En la ficha Opciones de esta ventana, podemos también hacer que los iconos se vean a un tamaño superior del habitual, mostrar sugerencias y combinaciones de teclado en las barras de herramientas; y hacer que los menús se abran mediante una animación.

5

Diseño de menús

Descripción de un menú

Un menú es uno de los componentes más habituales en los programas Windows. Se basa en un conjunto de opciones desplegables a partir de una serie de opciones principales, que permiten organizar el acceso a las diferentes partes de la aplicación. A efectos de programación, cada opción de menú es tratada y se maneja de la misma forma que cualquier control dentro del formulario.

Sin ser imprescindible su uso, si es útil a nivel organizativo, ya que evita la sobrecarga de controles en el formulario. Hemos de tener en cuenta que si no incorporamos un menú, tendremos que utilizar otro control, un botón de comando por ejemplo, por cada opción que queramos ejecutar dentro del formulario, lo que ocupa más espacio y resulta menos estético, debido a la sobrecarga de controles que se produciría dentro del formulario.

Lo anteriormente descrito se aplica en programas SDI, ya que si vamos a realizar un programa MDI, el uso de un menú para la ventana principal de la aplicación resulta obligatorio, siendo la forma que tiene la ventana MDI de invocar a las ventanas secundarias de la aplicación.

Características de un menú

Un menú está compuesto por un conjunto de opciones principales o nivel superior, que se disponen en la parte más próxima al título del formulario. De cada opción del nivel principal, se despliega un

conjunto de opciones o menú de nivel inferior dependientes del principal. A su vez, desde las opciones de este segundo nivel se pueden seguir abriendo sucesivamente niveles más inferiores.

Aunque la posibilidad de desplegar varios niveles de menús es muy interesante, de forma que podamos alcanzar un alto nivel de organización, no conviene, sin embargo, hacer un uso abusivo de esta cualidad, ya que un usuario puede tomar un mal concepto de una aplicación que le obliga a descender muchos niveles de menús para seleccionar una opción que es muy utilizada a lo largo de la aplicación.

Propiedades de un control Menú

Puesto que una opción de menú se maneja como un control más del formulario, podemos afirmar que una opción de menú es un objeto con sus propiedades y métodos. De las primeras vamos a ver una descripción:

- **Name.** Nombre del control.
- **Caption.** Texto que muestra la opción.
- **Enabled.** Valor lógico que habilita o deshabilita la opción. Cuando está deshabilitada, aparece en gris y el usuario no puede seleccionarla.
- **Checked.** Dato lógico que cuando es verdadero, muestra una marca en la opción de menú, para informar al usuario de que esa opción está activada.
- **Visible.** Si su valor es False, oculta la opción. Por defecto es True.
- **Index.** Permite definir una opción como parte de un array de controles u opciones de menú. Esta propiedad es útil para crear nuevas opciones de menú en tiempo de ejecución.
- **Shortcut.** Contiene el valor de una tecla de método abreviado o acelerador de teclado, como F5 o Ctrl+G. Cuando se define un acelerador, la combinación de teclas aparece junto a la descripción del menú. La ventaja de un acelerador es que no necesitamos navegar a través de las opciones del menú para llegar a una determinada. Si la que queremos tiene definido un acelerador, tecleándolo lanzaremos directamente la opción de ese menú.
- **WindowList.** En un formulario MDI, si damos el valor True a esta propiedad en una opción de nivel superior, el programa creará de forma transparente al usuario un submenú con las ventanas que tiene abiertas la aplicación, marcando la ventana activa.

En cuanto a métodos, el único disponible para el programador es Click(), que se dispara al seleccionar el usuario una opción del menú. Aquí deberemos incluir el código que se ejecutará para la opción de menú.

El Editor de menús

Para crear un menú en el formulario hemos de hacerlo usando la ventana de edición de menús, la cual abrimos de una de las siguientes maneras:

- Hacer clic con el botón derecho del ratón dentro del formulario, para visualizar el menú contextual del mismo. Dentro de este menú tenemos una opción para abrir el editor de menús.

- Pulsar el botón *Editor de menús* en la barra de herramientas de VB.
- Seleccionar el menú *Herramientas + Editor de menús* de VB.
- Usar la combinación de teclado *Ctrl+E.*

Al abrir el editor de menús por primera vez, obtenemos una ventana similar a la que aparece en la figura 152.

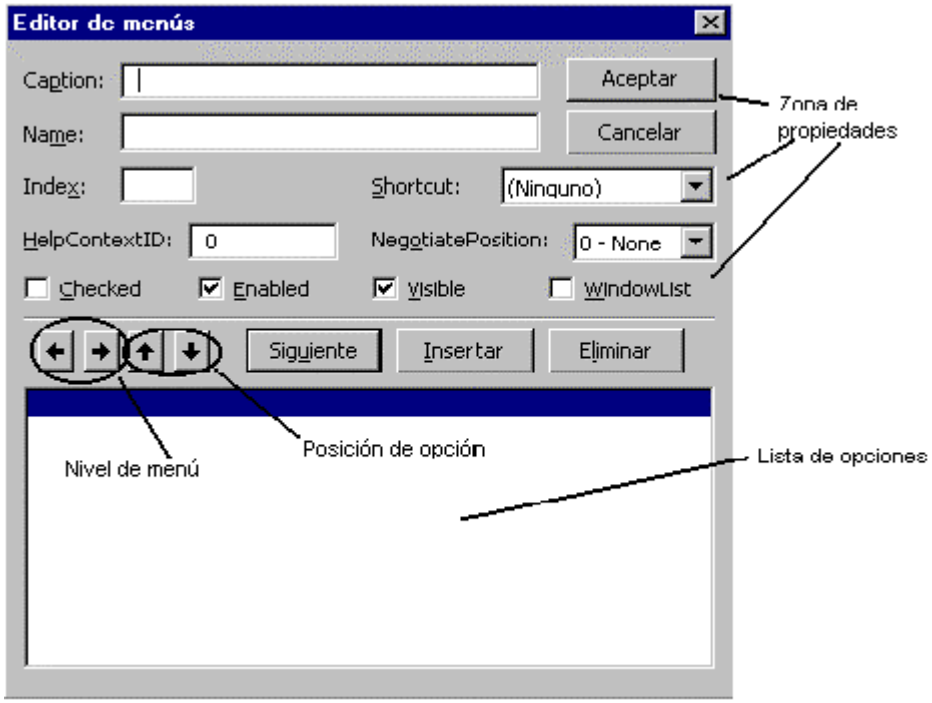


Figura 152. Editor de menús.

Los pasos para crear las opciones del menú son los siguientes:

- Introducir el Caption o título de la opción, que es la cadena de caracteres que aparecerá en el menú. Si deseamos que esta opción tenga una tecla de acceso rápido o hotkey, hemos de situar el carácter & precediendo a la letra que queramos actúe como hotkey. De esta forma, la letra aparecerá subrayada en tiempo de ejecución y podremos seleccionar la opción con la combinación de teclado *Alt+hotkey* si pertenece al nivel principal. Si es una opción incluida en un nivel inferior, debemos abrir primero el nivel y después pulsar la tecla de acceso rápido.
- Asignar un valor a Name, que es el nombre del control, y nos servirá para hacer referencia a él dentro del código.
- En este punto podemos asignar a la opción algún tipo de variante: habilitarla, marcarla, ocultarla, asignarle un acelerador de teclado o Shortcut, etc.
- Completadas todas las propiedades de la opción, pulsamos Siguiente, con lo que pasará a la parte inferior de la ventana, en la zona de opciones ya creadas

- Un aspecto muy importante a tener en cuenta es el nivel en el que vamos a situar la opción. Si deseamos que esté situada en un nivel inferior, hemos de hacer clic en el botón con la flecha a la derecha y un grupo de puntos aparecerá en la zona de opciones indicando que esa opción es de nivel inferior. Cuantos más grupos de puntos haya, mayor profundidad tendrá la opción. De forma inversa, si queremos que una opción suba de nivel, hemos de hacer clic en el botón con flecha a la izquierda. Si la opción no tiene puntos, es que pertenece al nivel principal.

- Es posible cambiar la posición de una opción, haciendo clic en los botones con las flechas arriba o abajo la desplazaremos en una u otra dirección.

- Pulsando el botón *Insertar* crearemos una nueva opción vacía entre las opciones ya creadas.

- Pulsando el botón *Eliminar* borraremos la opción actualmente resaltada.

- Podemos agrupar diferentes opciones dentro de un mismo menú usando separadores o líneas divisorias. Simplemente hemos de poner un guión "-" en el Caption de la opción de menú.

Añadir código a una opción de menú

Una vez terminado el diseño del menú, tenemos dos formas de incluir el código que debe ejecutar:

- Desde la ventana de diseño del formulario, seleccionar la opción de menú, lo que abrirá la ventana de código del formulario, situándonos en el método Click(), que como hemos visto anteriormente es el único disponible para este control.

- Abrir la ventana de código del formulario, y buscar nosotros directamente en la lista de objetos el control a codificar.

Creación de un menú

Vamos a ver una aplicación que contenga un menú en el que sus opciones muestren las propiedades descritas anteriormente.

Tomamos del grupo de ejemplos la aplicación llamada *Menu*. Una vez cargada en VB, abrimos su único formulario *frmMenu* que dispone de la siguiente estructura de menú:

```
Archivo
---Usuarios (Enabled: False)
---Entradas (Checked: True)
---Salir
Contabilidad
---Apuntes
---Consultas
------Cuenta
------Fecha
Clientes
---Altas
---Bajas
---(Separador)
---Enviando correo
Proveedores
```

```
---Altas (Shortcut: Ctrl+F)
---Bajas
---Consultas (Shortcut: Ctrl+Y)
Fuente (Visible: False / este será un menú emergente)
---Arial
---Courier
---Roman
Dinámico
---Dinámico1 (Index: 1)
```

Algunas de las propiedades ya están asignadas en tiempo de diseño, otras las modificaremos en ejecución. Así pues, ejecutamos la aplicación y en el primer menú podemos ver la opción *Usuarios* deshabilitada, aparece en gris, y *Enviando correo* que está marcada, como aparece en la figura 153.

Figura 153. Aplicación Menu. Menú Archivo.

Desde aquí podemos probar el uso de un hotkey. Si pulsamos por ejemplo la tecla "S", se ejecutará el código asociado a la opción *Salir*, que finalizará el programa descargando el formulario. Lo podemos ver en el código fuente 169.

```
Private Sub mnuArSalir_Click()
Unload Me
End Sub
```

Código fuente 169

Podemos cambiar el estado de la opción *Usuarios* pulsando el botón *Habilitar/Deshabilitar* del formulario, que tiene el código fuente 170.

```
Private Sub cmdHabilDeshab_Click()
If Me.mnuArUsuarios.Enabled Then
  Me.mnuArUsuarios.Enabled = False
Else
  Me.mnuArUsuarios.Enabled = True
End If
End Sub
```

Código fuente 170

De la misma forma, se puede cambiar la marca de la opción *Enviando correo*, pulsando el botón *Enviar correo*, lo vemos en el código fuente 171.

```
Private Sub cmdEnviar_Click()
If Me.mnuArEnvia.Checked Then
  Me.mnuArEnvia.Checked = False
Else
  Me.mnuArEnvia.Checked = True
End If
End Sub
```

Código fuente 171

Como muestra de menú con varios niveles de opciones, tenemos el menú *Contabilidad*, en el que la opción *Consultas* despliega a su vez otro submenú, como muestra la figura 154.

Figura 154. Aplicación Menu. Menú Contabilidad con todos los submenús desplegados.

En el menú Proveedores disponemos de dos opciones con acelerador de teclado, lo podemos ver en la figura 155.

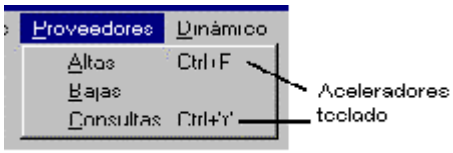


Figura 155. Aplicación Menu. Menú Proveedores.

Al pulsar alguna de estas combinaciones de teclado aparecerá un mensaje informándonos de que hemos seleccionado esa opción, veámoslo en el código fuente 172.

```
Private Sub mnuPrAltas_Click()
MsgBox "Menú Proveedores - Opción Altas"
End Sub
'-----------------------------------------
Private Sub mnuPrConsultas_Click()
MsgBox "Menú Proveedores - Opción Consultas"
End Sub
```

Código fuente 172

Durante el diseño del menú, a la opción de nivel superior *Fuente*, se le ha quitado la marca en la propiedad *Visible*. Esto provoca que al ejecutar el programa no aparezca en la barra de menús ya que *Visible* es *False*. La causa de hacer esto es conseguir un menú emergente de los que aparecen al pulsar el botón derecho del ratón, como aparece en la figura 156.

La forma de comprobar la pulsación del ratón en el formulario es mediante el método *MouseDown()*. El parámetro *Button* nos informa del botón pulsado, si ha sido el derecho, visualizamos el menú *Fuente* con el método *PopupMenu* del formulario, veámoslo en el código fuente 173.

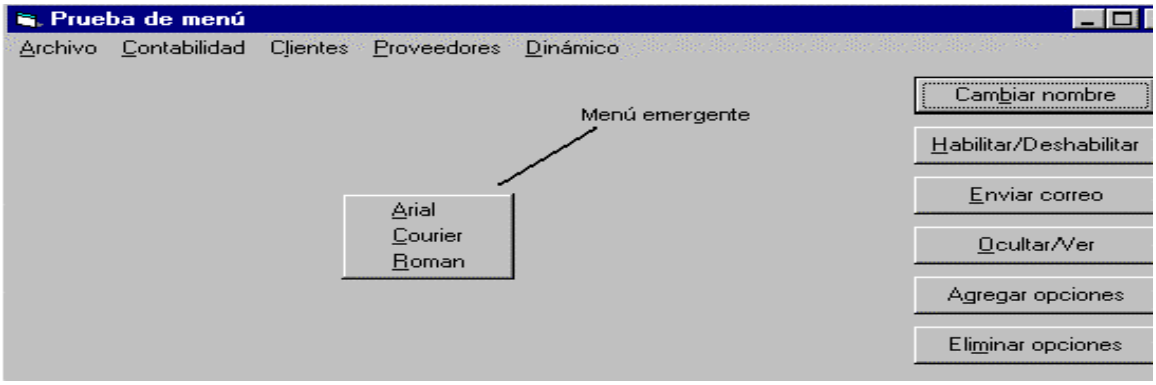


Figura 156. Aplicación Menu mostrando el menú emergente.

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As
Single)
If Button = vbRightButton Then
  PopupMenu Me.mnuFuente, vbPopupMenuRightButton, , , Me.mnuFuRoman
End If
End Sub
```

Código fuente 173

La sintaxis de PopupMenu es la siguiente:

```
PopupMenu cNombreMenu, xConstantes, x, y, mnuNegrita
```

- *cNombreMenu.* Nombre del menú que se va a presentar como emergente.
- *xConstantes.* Valores que especifican la ubicación y el comportamiento del menú. Los valores para la ubicación son los siguientes:
- *vbPopupMenuLeftAlign.* El lado izquierdo del menú se sitúa en x.
- *vbPopupMenuCenterAlign.* El menú se sitúa centrado.
- *vbPopupMenuRightAlign.* El lado derecho del menú se sitúa en x.

Los valores para el comportamiento son los siguientes:

- *vbPopupMenuLeftButton.* Sólo se pueden seleccionar las opciones del menú con el botón izquierdo del ratón; esta es la opción por defecto.
- *vbPopupMenuRightButton.* Se pueden seleccionar las opciones del menú también con el botón derecho del ratón.
- *x, y.* Coordenadas para situar el menú en el formulario.

- *mnuNegrita.* Nombre de una de las opciones del menú que deseamos que aparezca resaltada en negrita.

Es posible cambiar el título de una opción en tiempo de ejecución simplemente asignando una nueva cadena a su propiedad *Caption*. El botón *Cambiar nombre* cambia el título de la opción *Apuntes* del menú *Contabilidad* de la forma que indica el código fuente 174.

```
Private Sub cmdCambiaCaption_Click()
Me.mnuCoApuntes.Caption = "Nuevo nombre"
End Sub
```

Código fuente 174

Si queremos que una opción no sea visible para el usuario, ponemos a *False* su propiedad *Visible*, como hacemos en el botón *Ocultar/Ver*, como indica el código fuente 175.

```
Private Sub cmdOcultarVer_Click()
If Me.mnuPrBajas.Visible Then
  Me.mnuPrBajas.Visible = False
Else
  Me.mnuPrBajas.Visible = True
End If
End Sub
```

Código fuente 175

Con los botones *Agregar opciones* y *Eliminar opciones* añadimos o quitamos opciones al menú *Dinámico* en tiempo de ejecución. Para conseguir esta funcionalidad utilizamos un array de controles *Menu*, de forma que a la propiedad *Index* de la opción *Dinámico1* le damos el valor 1 (podríamos haberle puesto 0, pero por cuestiones de controlar el número de opciones creadas es más sencillo usar 1). También hemos definido la variable *mnOpcionesDinam* a nivel de formulario para que actúe como contador de las opciones que llevamos creadas.

Una vez hecho esto, cada vez que pulsemos el botón para agregar, utilizaremos la instrucción *Load* y el array de opciones con el nuevo número de opción a crear. Cuando pulsemos el botón para eliminar, usaremos la instrucción *Unload* y el array de controles con el número de opción a suprimir. El código fuente 176 contiene los métodos involucrados en estas operaciones.

```
‘ zona de declaraciones
Option Explicit
Private mnOpcionesDinam As Integer
‘---------------------------------
Private Sub Form_Load()
‘ inicializamos el contador
mnOpcionesDinam = 1
End Sub
‘---------------------------------
Private Sub cmdAgregar_Click()
‘ sumamos al contador
mnOpcionesDinam = mnOpcionesDinam + 1
‘ creamos una nueva opción de menú con la instrucción Load
Load Me.mnuDiDinam(mnOpcionesDinam)
Me.mnuDiDinam(mnOpcionesDinam).Caption = "Dinámico&" & CStr(mnOpcionesDinam)
```

```
End Sub
'---------------------------------
Private Sub cmdEliminar_Click()
' si el contador es mayor de 1 permitimos borrar opciones,
' en caso contrario no borramos ya que se produciría un error
If mnOpcionesDinam > 1 Then
  ' eliminamos una opción del menú descargándola
  Unload Me.mnuDiDinam(mnOpcionesDinam)
  ' ajustamos el contador
  mnOpcionesDinam = mnOpcionesDinam - 1
End If
End Sub
'---------------------------------
```

Código fuente 176

Menús en formularios MDI

Un menú en un formulario MDI funciona de la misma manera que en una aplicación SDI, la única salvedad reside en que al abrir una ventana secundaria MDI, si esta tiene su propio menú, dicho menú sustituirá al del formulario MDI mientras la ventana secundaria se encuentre en funcionamiento. Al descargarse volverá a verse el menú del formulario MDI.

La propiedad *WindowList* al ser usada en el menú de un formulario MDI nos permite definir un menú en el que sus opciones mostrarán los nombres de las ventanas secundarias abiertas.

En la aplicación de ejemplo *MenuMDI* se ha incorporado esta característica. Disponemos de dos formularios: *frmPrincipal* que es el formulario MDI y *frmDocumento* que es el formulario que va a actuar como documento dentro del MDI.

En el editor de menús de *frmPrincipal*, la opción *mnuVentana* tiene marcada la propiedad *WindowList*, con lo que estará activada la capacidad de mostrar en este menú los nombres de las ventanas secundarias abiertas dentro de la MDI, como muestra la figura 157.

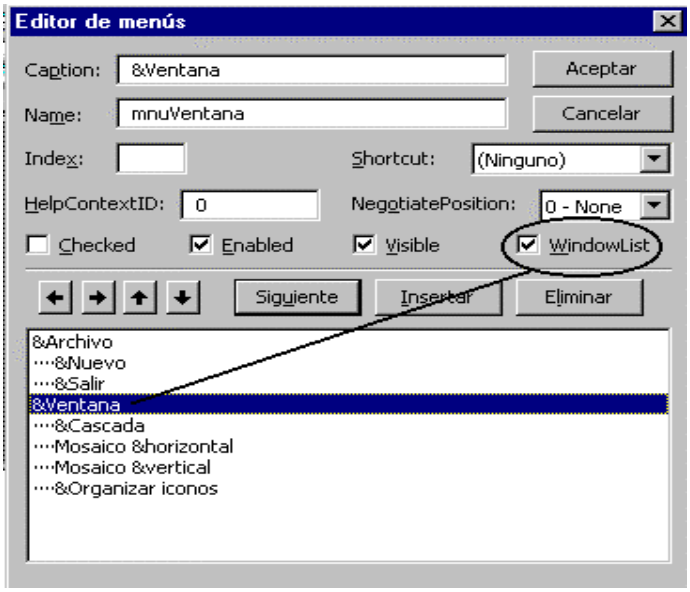


Figura 157. Editor de menú con propiedad WindowList activada.

Ejecutamos ahora la aplicación. Para trabajar con un nuevo documento seleccionamos la opción *Nuevo* del menú *Archivo* que crea una nueva ventana secundaria y controla su número gracias a un contador definido a nivel de módulo.

```
Private Sub mnuArNuevo_Click()
Dim frmDocum As frmDocumento
' instancia un nuevo objeto de la clase frmDocumento
Set frmDocum = New frmDocumento
' suma el contador
mnNumDocumento = mnNumDocumento + 1
frmDocum.Caption = "Documento" & mnNumDocumento
frmDocum.Show
End Sub
```

Código fuente 177

Abrimos varios documentos más con esta opción y pasamos al menú *Ventana*. Este menú es el típico de muchas aplicaciones MDI Windows, en el que las primeras opciones son para organizar las diferentes ventanas secundarias de la aplicación, y las restantes sirven para cambiar el foco entre esas ventanas secundarias. Para organizar las ventanas disponemos del método *Arrange* del formulario MDI. El código fuente 178 muestra las rutinas encargadas de la organización de ventanas.

```
Private Sub mnuVeCascada_Click()
' organiza las ventanas en cascada
Me.Arrange vbCascade
End Sub
'------------------------------------------
Private Sub mnuVeIconos_Click()
' organiza las ventanas iconizadas
Me.Arrange vbArrangeIcons
End Sub
'------------------------------------------
Private Sub mnuVeMosHoriz_Click()
' organiza las ventanas en mosaico horizontal
Me.Arrange vbTileHorizontal
End Sub
'------------------------------------------
Private Sub mnuVeMosVert_Click()
' organiza las ventanas en mosaico vertical
Me.Arrange vbTileVertical
End Sub
```

Código fuente 178

Figura 158. Menú de aplicación MDI con opciones de ventanas abiertas.

Las opciones del menú de la figura 158 corresponden cada una a una de las ventanas abiertas dentro del formulario MDI. Estas opciones son creadas, mantenidas y eliminadas por la aplicación sin que tengamos que escribir ni una sola línea de código para ello. Lo que supone una gran ventaja y ahorro de tiempo al programador.

6

El registro Windows

De los ficheros INI al Registro

Por norma general, los programas necesitan consultar cierta información adicional que no está incluida dentro del propio programa, como pueda ser la ruta de los ficheros de la aplicación, los últimos documentos abiertos, etc.

En Windows 3.1, esta información se guarda en los llamados ficheros de inicialización o INI. WIN.INI y SYSTEM.INI contienen la configuración general de los programas y el equipo en el que está instalado Windows; adicionalmente, cada programa proporciona al instalarse uno o más ficheros INI con la configuración específica para esa aplicación.

En Windows 95, los ficheros INI siguen estando disponibles, aunque se recomienda usarlos sólo a nivel de compatibilidad con las aplicaciones de 16 bits, y emplear para las nuevas aplicaciones de 32 bits el *Registro de configuraciones (Registry).*

El registro por dentro

El registro es una base de datos que almacena la configuración de los componentes físicos del equipo y aplicaciones instaladas en él. Para manipularlo, disponemos de la aplicación *Regedit* o *Editor del registro*, que podemos ejecutar abriendo el menú de inicio de Windows 95 y seleccionando la opción *Ejecutar*, escribiremos *Regedit* y pulsaremos *Aceptar* en dicha ventana, lo que abrirá el editor del registro, ver figura 159.

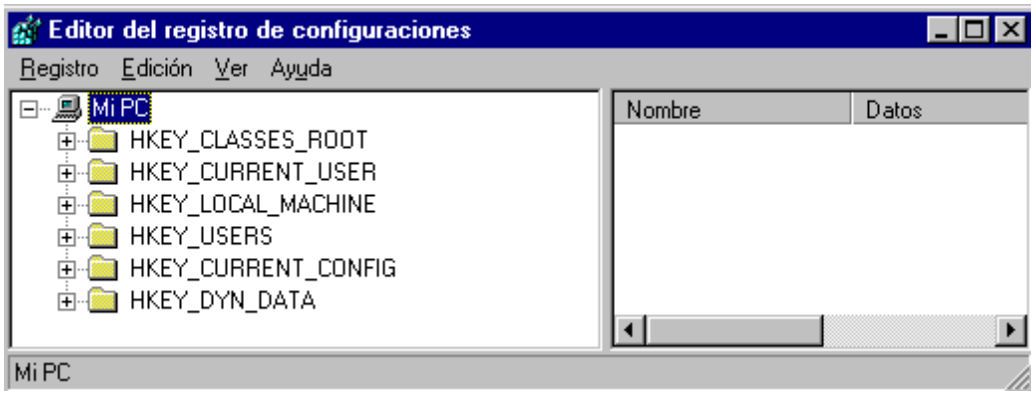


Figura 159. Ventana del editor del registro de Windows 95.

El registro consta de seis ramas principales que se detallan a continuación.

- HKEY_CLASSES_ROOT. Contiene las asociaciones entre extensiones de ficheros y aplicaciones, de forma que Windows sepa que aplicación ha de ejecutar al hacer doble clic o pulsar Enter sobre un fichero con una determinada extensión.
- HKEY_CURRENT_USER. Contiene la información para cada usuario.
- HKEY_LOCAL_MACHINE. Información sobre el equipo en el que está instalado el sistema operativo.
- HKEY_USERS. Contiene la información general de todos los usuarios del equipo.
- HKEY_CURRENT_CONFIG. Contiene la configuración del monitor e impresora.
- HKEY_DYN_DATA. Información dinámica sobre el sistema, rendimiento, etc.

A efectos de grabar la información de nuestras aplicaciones en el registro, la organización dentro del mismo podemos considerarla dispuesta en una serie de entradas que contienen el nombre de la aplicación u otro nombre descriptivo que sirva para reconocerla. Cada entrada contendrá a su vez una o más secciones, y cada sección contendrá una o más claves, estas últimas son las que guardan los diferentes valores de configuración para la aplicación.

Manipulación de las entradas del registro

Visual Basic proporciona las siguientes funciones para el manejo de las entradas del registro.

```
GetSetting(cNombreAplicación, cSección, cClave [, cValorDefecto])
```

Recupera el valor de una clave existente en el registro. Si no existe alguno de los parámetros en el registro, se devuelve el valor de cValorDefecto.

- cNombreAplicación. Cadena conteniendo el nombre de la aplicación, de la que vamos a tomar una clave.
- cSección. Cadena conteniendo el nombre de la sección donde se encuentra la clave.
- cClave. Cadena conteniendo el nombre de la clave a devolver.

- cValorDefecto. Opcional. Valor a devolver si no existe valor en la clave. Si se omite, el valor por defecto es una cadena de longitud cero ("").

```
SaveSetting cNombreAplicación, cSección, cClave, cValorClave
```

Crea una entrada para una aplicación en el registro de configuración de Windows, y graba en él un valor. Si la entrada ya existe, sólo se graba el nuevo valor. Si existe algún problema al guardar el valor, se producirá un error.

- cNombreAplicación. Cadena conteniendo el nombre de la aplicación, en el que se va a guardar un valor.
- cSección. Cadena conteniendo el nombre de la sección donde se va a guardar la clave.
- cClave. Cadena conteniendo el nombre de la clave a guardar.
- cValorClave. Valor que se va a guardar en cClave.

```
DeleteSetting cNombreAplicación, cSección [, cClave]
```

Borra una sección o una clave de una entrada del registro de Windows.

- cNombreAplicación. Cadena conteniendo el nombre de la aplicación de la que se va a eliminar un valor.
- cSección. Cadena conteniendo el nombre de la sección a borrar, o de la que se va a eliminar una clave. Si no se indica el parámetro cClave, se borrará la sección completa, con todas las claves contenidas.
- cClave. Opcional. Cadena conteniendo el nombre de la clave a borrar.

```
GetAllSettings(cNombreAplicación, cSección)
```

Devuelve un array bidimensional con las claves contenidas en la sección perteneciente a una aplicación incluida en el registro. Si no existen los parámetros registrados, se devuelve un valor Variant sin inicializar.

- cNombreAplicación. Cadena conteniendo el nombre de la aplicación de la que se va a recuperar valores.
- cSección. Cadena conteniendo el nombre de sección de la que se van a recuperar sus claves.

La aplicación de ejemplo PruRegis, contiene el formulario *frmRegistro*, que incluye un menú con una serie de operaciones para las entradas del registro, de manera que ilustren el manejo de las funciones disponibles para el mismo.

Comencemos con la carga del formulario, cuando esta se produce, se ejecuta si existe, el código contenido en el método Load(). En este caso se buscará la configuración de la dimensión del formulario en el registro, y se aplicará al formulario recién cargado. Si no existiera una o más claves, se tomará un valor por defecto.

```
Private Sub Form_Load()
' establece las dimensiones del formulario
' a las mismas que tenía al salir la última
' vez del programa
```

```
RestablecerConf
End Sub
' ------------------------------------------------
Private Sub RestablecerConf()
' toma los datos del registro sobre posición
' y tamaño de este formulario, y los asigna
' al objeto formulario actual
Me.Top = GetSetting("PruRegis", "FormDimension", "FrmTop", "100")
Me.Left = GetSetting("PruRegis", "FormDimension", "FrmLeft", "100")
Me.Height = GetSetting("PruRegis", "FormDimension", "FrmHeight", "100")
Me.Width = GetSetting("PruRegis", "FormDimension", "FrmWidth", "100")
End Sub
```

Código fuente 179

De forma similar a la carga, cuando se descarga el formulario, se ejecuta el método de evento Unload(), que aquí tiene el código fuente 180.

```
Private Sub Form_Unload(Cancel As Integer)
GuardarConf
End Sub
' ------------------------------------------------
Private Sub GuardarConf()
' guarda en el registro la posición
' y tamaño del formulario
SaveSetting "PruRegis", "FormDimension", "FrmTop", Me.Top
SaveSetting "PruRegis", "FormDimension", "FrmLeft", Me.Left
SaveSetting "PruRegis", "FormDimension", "FrmHeight", Me.Height
SaveSetting "PruRegis", "FormDimension", "FrmWidth", Me.Width
End Sub
```

Código fuente 180

Lo que se hace, es llamar a un procedimiento que guarda la dimensión de la ventana en el registro, de forma que al iniciar la próxima vez esta aplicación, la ventana aparezca en el mismo sitio y con el mismo tamaño que se dejó.

En lo que respecta al menú del formulario de esta aplicación, sus opciones realizan las operaciones que acabamos de ver, además de borrar los valores del registro para el programa, y mostrar cada una de las claves contenidas en el registro, como podemos ver en el código fuente 181 de estas opciones.

```
Private Sub mnuReBorrar_Click()
' borrar todos los valores del registro
' de la sección FormDimension.
DeleteSetting "PruRegis", "FormDimension"
End Sub
' ----------------------------
Private Sub mnuReVer_Click()
' muestra todos los valores del registro
' correspondientes a la sección FormDimension
Dim laDimensiones As Variant
Dim lnInd As Integer
laDimensiones = GetAllSettings("PruRegis", "FormDimension")
For lnInd = LBound(laDimensiones, 1) To UBound(laDimensiones, 1)
    MsgBox "Clave " & laDimensiones(lnInd, 0) & vbCrLf & _
    "Valor " & laDimensiones(lnInd, 1), , "Sección FormDimension"
Next
End Sub
```

Código Fuente 181

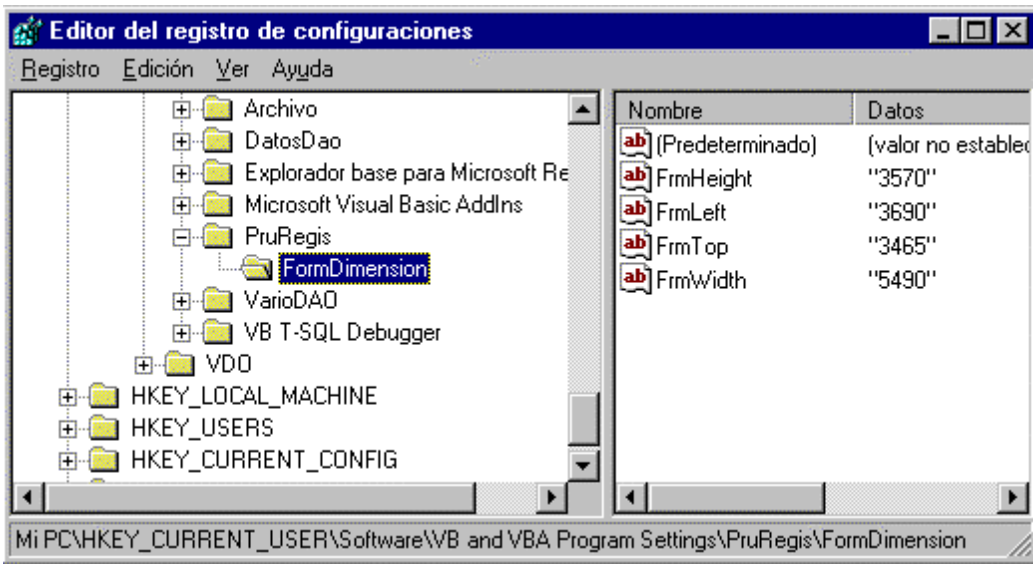


Figura 160. Configuración de la aplicación PruRegis.

En el procedimiento de borrado hemos eliminado la sección completa, pero si hubiésemos querido eliminar las claves por separado, podríamos haber empleado la siguiente sintaxis:

```
DeleteSetting "PruRegis", "FormDimension", "FrmTop"
DeleteSetting "PruRegis", "FormDimension", "FrmLeft"
DeleteSetting "PruRegis", "FormDimension", "FrmHeight"
DeleteSetting "PruRegis", "FormDimension", "FrmWidth"
```

Incluyendo como tercer parámetro la clave a eliminar de la sección. De esta manera podríamos haber borrado sólo algunas claves, sin necesidad de borrar la sección completa.

7

Tratamiento de datos con ActiveX Data Objects (ADO)

Introducción a la programación con bases de datos

Una base de datos es una estructura diseñada para el manejo de información. Dicha información se clasifica dentro de la base de datos en tablas. Una tabla es un grupo de datos que hacen referencia a un tema determinado como pueda ser la información de clientes de una empresa, las facturas emitidas por la empresa, etc. Dentro de la tabla, los datos se organizan en filas o registros; si la tabla contiene la información de clientes, una fila hará referencia a la información de uno de los clientes. A su vez las filas están compuestas por columnas o campos, que hacen referencia a un aspecto concreto de la fila; siguiendo con el ejemplo de los clientes, un campo de una fila de clientes puede ser el nombre del cliente, la dirección, el teléfono, etc.

La figura 161 nos muestra la representación gráfica de una tabla, en la que podemos ver sus filas y columnas.

Las bases de datos están contenidas en un *Sistema Gestor de Bases de Datos Relacionales (SGBDR)*, que es el que proporciona los mecanismos necesarios para realizar consultas de datos, mantener los niveles de integridad de la información, seguridad en el acceso, etc. A este conjunto de mecanismos se le denomina también *motor de datos.*

Según el motor de datos utilizado, el SGBDR proporcionará un conjunto de capacidades que en función de su número u optimización darán una mayor o menor potencia a la base de datos utilizada con respecto a otras

Figura 161. Tabla de Clientes.

Diseño de la base de datos

Esta es la primera tarea que hemos de acometer en el desarrollo de una aplicación que manipule una base de datos, los pasos a dar en esta etapa del proyecto son los siguientes:

Establecer el modelo de la aplicación

En este paso, debemos analizar lo que la aplicación debe hacer, recabando información del futuro usuario/s de la aplicación. En función de sus necesidades, comenzaremos a tomar idea de las tareas que ha de afrontar el programa. Una vez que tenemos una visión global del trabajo a realizar, debemos separarlo en varios procesos para facilitar su programación.

Analizar que información necesita la aplicación

En este punto, y una vez averiguadas las tareas a realizar por el programa, hemos de determinar que información tenemos que incluir en la base de datos y cual no, de manera que no sobrecarguemos las tablas con datos innecesarios que podrían retardar la ejecución del programa.

Un ejemplo lo tendríamos en la típica tabla de clientes para un comercio; si se desea enviar a cada cliente una felicitación por su cumpleaños, es necesario incluir un campo con su fecha de nacimiento, en caso contrario, no se incluirá este campo evitando sobrecargar la tabla.

Establecer la información en tablas

Para organizar los datos en una aplicación, comenzaremos agrupándolos por temas. De esta manera los datos que tengan que ver con clientes formarán la tabla Clientes; los que tengan que ver con proveedores la tabla Proveedores, y así sucesivamente, por cada tema crearemos una tabla.

Hemos de tener cuidado en esta fase de diseño, puesto que por ejemplo, la información de las tablas Clientes y Proveedores (Código, Nombre, Dirección, etc.) es muy similar y podríamos pensar en ponerla en una única tabla, pero en el caso de los clientes, tendríamos que incluir un campo con la

dirección del almacén de entrega de la mercancía, mientras que como a los proveedores no vamos a realizarles entregas, dicho campo no sería necesario, lo que ocasionaría muchas filas en la tabla con este campo inutilizado cuando se tratara de proveedores.

Normalización de la base de datos

Una vez creadas las tablas, el proceso de normalización consiste en eliminar la información redundante existente entre las tablas. Pongamos como ejemplo una tabla que contiene facturas, y cada fila de la tabla incluye los datos del cliente, como se muestra en la figura 162.

Figura 162. Tabla de facturas con información sin normalizar.

Si en lugar de incluir todos los datos del cliente, creamos una tabla para los clientes con su información, agregando un campo de código, y en la tabla de cabecera de facturas, sustituimos los campos de clientes por el código de cliente, ahorraremos un importante espacio en la tabla de facturas y seguiremos manteniendo la referencia al cliente que hemos facturado, ya que el código de cliente nos llevará a sus datos dentro de la tabla Clientes, como se muestra en la figura 163.

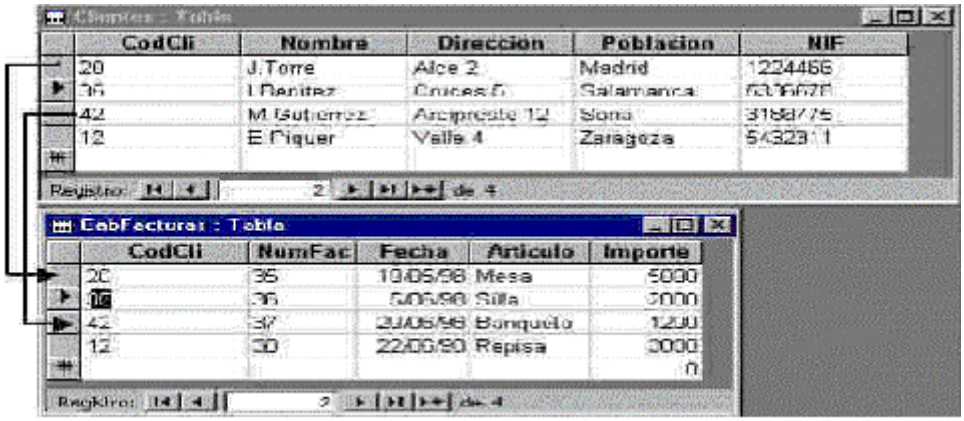


Figura 163. Información normalizada entre las tablas Clientes y CabFacturas.

Un factor muy importante a efectos del rendimiento de la aplicación, es realizar un estudio del tamaño necesario para los campos de las tablas, establecer por ejemplo, un tamaño de 50 caracteres para un campo de NIF es estar desperdiciando claramente espacio en el disco duro por un lado, y por otro, si tenemos varios campos con un desfase parecido, cuando la tabla comience a crecer, todo ese conjunto de pequeños detalles de diseño se volverá contra nosotros en forma de lentitud a la hora de hacer operaciones con la tabla.

Un elemento muy útil a la hora de aprovechar espacio lo constituyen las tablas de datos generales. Estas tablas contienen información que puede ser utilizada por todos los procesos de la aplicación, y sirven de apoyo a todas las tablas de la base de datos. Como ejemplo tenemos una tabla que guarde códigos postales, que puede ser utilizada tanto por una tabla de clientes como de proveedores, bancos, etc.

Relacionar las tablas

Eliminada toda información repetida o innecesaria de las tablas, necesitamos establecer relaciones entre las mismas. Una relación es un elemento por el cual dos tablas se vinculan mediante una clave, que pueden ser del siguiente tipo:

- Clave Primaria. Este tipo de clave consiste en que cada registro de la tabla es identificado de forma única mediante el valor de un campo o combinación de campos. Si intentamos dar de alta un registro con una clave ya existente, el motor de datos cancelará la operación provocando un error. Una clave primaria puede ser el código de un cliente, como vemos en la figura 164, o un código postal.

Figura 164. Tabla Clientes con clave primaria establecida en el campo CodCli.

- Clave Ajena. Esta clave se forma mediante el valor de un campo o combinación de campos de una tabla, y tiene como propósito establecer una relación con la clave primaria de otra tabla, pudiendo existir varios registros de la clave ajena que se relacionen con un único registro en la clave primaria de la tabla relacionada. En la figura 165, el campo CodCli de la tabla CabFacturas es clave ajena del campo del mismo nombre en la tabla Clientes, existiendo varias filas de CabFacturas con el mismo valor, que se corresponden con una única fila de Clientes.

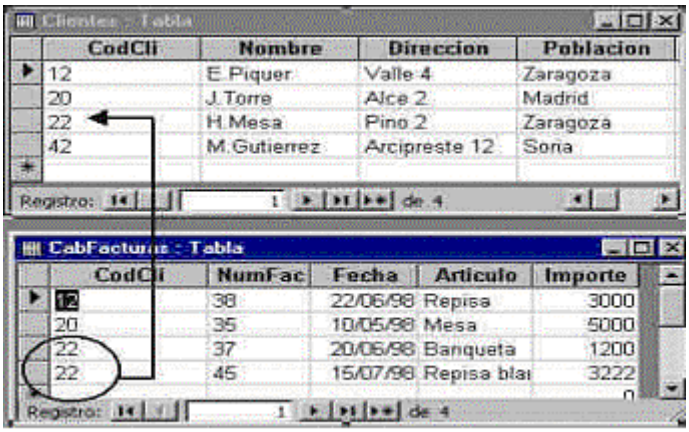


Figura 165. La tabla CabFacturas tiene una clave ajena en el campo CodCli.

En las relaciones por otra parte, tenemos los siguientes tipos:

- Relación de uno a varios. Este es el tipo de relación más común; en él, una fila de una tabla se relaciona con una o más filas de una segunda tabla, pero cada fila de la segunda tabla se relaciona sólo con una de la primera. Un ejemplo claro de esta situación lo tenemos en la figura anterior, cada fila de la tabla Clientes puede relacionarse con una o varias filas de la tabla CabFacturas por el campo CodCli; pero cada fila de la tabla CabFacturas se relaciona sólo con una fila de la tabla Clientes.

- Relación de varios a varios. Esta relación se produce, cuando cada fila de una tabla se relaciona con cada fila de una segunda tabla y viceversa. En esta situación debemos definir una tabla intermedia que tenga relaciones de uno a varios con las otras dos tablas. Pongamos como ejemplo un almacén de productos cosméticos que distribuye a diferentes perfumerías; cada perfumería puede adquirir varios tipos de cosméticos, e igualmente, cada cosmético puede ser adquirido por varias perfumerías. Al trasladar esta información a una base de datos podríamos hacerlo como muestra la figura 166.

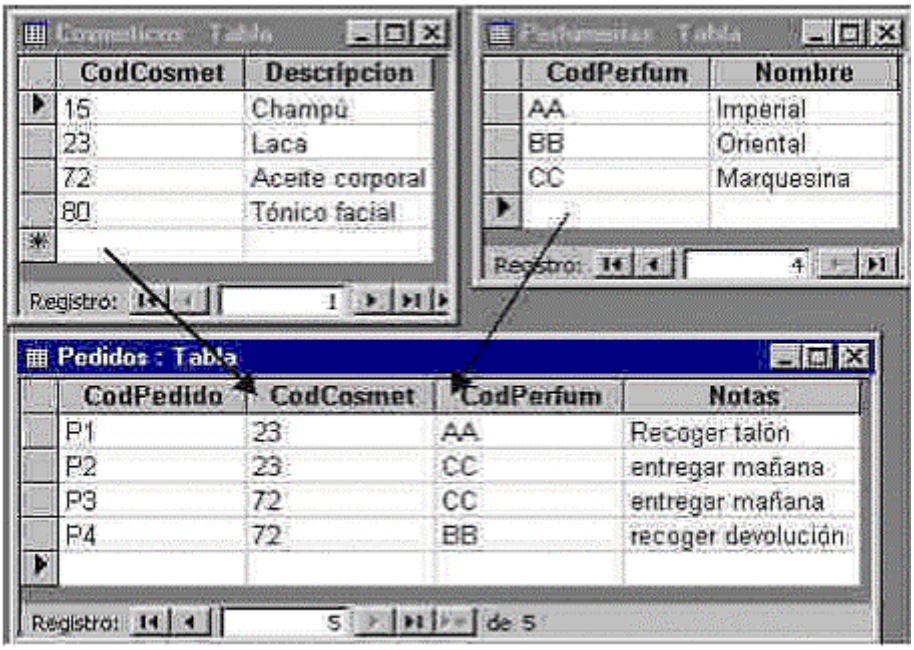


Figura 166. Relación de varios a varios resuelto con una tabla intermedia.

Establecer la integridad referencial

Una vez que se establece una relación entre dos tablas de la base de datos, el administrador de la base de datos debe decidir, en el caso de que el motor de datos lo permita, si se debe gestionar la integridad referencial.

La integridad referencial es la operación por la cual se debe mantener la consistencia entre la información de las tablas de la base de datos. Pensemos en el ejemplo anterior de las tablas Clientes y CabFacturas; si hemos establecido una relación entre las dos tablas a través del campo CodCli de ambas, y eliminamos en la tabla Clientes la fila que contiene el código de cliente 42, nos encontraremos con un problema de inconsistencia de información, ya que cuando desde la tabla CabFacturas nos posicionemos en la fila que contiene el código de cliente 42 y vayamos a recuperar

los datos de ese código de cliente a la tabla Clientes, nos encontraremos con que no existe ninguna fila con el código 42, en este caso, al registro no relacionado de la tabla CabFacturas se le denomina *registro huérfano*.

El tema de la integridad referencial es una decisión de diseño, puede exigirse o no; en el caso de que sea necesaria, será responsabilidad del programador mantenerla o del motor de datos en el caso de que este último la soporte. De cualquiera de las maneras, al exigir integridad referencial nos aseguramos que no quedarán registros huérfanos entre las tablas relacionadas, puesto que si intentamos modificar los valores que forman parte de la relación y no existe el nuevo valor introducido como parte de la clave, se producirá un error que no permitirá completar el proceso.

Adicionalmente se pueden establecer actualizaciones y eliminaciones en cascada, de forma que si se modifica o borra una fila en la primera tabla de la relación, también se modificarán o borrarán los registros relacionados en la segunda tabla.

Definir índices para las tablas

En este punto hemos de analizar que tablas y campos de la base de datos son susceptibles de ser consultados un mayor número de veces, y definir un índice para ellos, ya que al ser información que necesitaremos consultar con más frecuencia que el resto, también será necesaria recuperarla más rápidamente.

Un índice ordena las filas de una tabla en base a un criterio determinado. La clave del índice puede estar formada por uno o varios campos, de esta manera podemos encontrar rápidamente filas en la tabla, o mostrarla ordenada en base a la clave del índice.

Definir validaciones en la inserción de datos

La validación consiste en comprobar que los datos que vamos a incluir en la fila de una tabla son correctos. Podemos definir validaciones a nivel de motor, de forma que sea el propio motor de datos el que compruebe la información y la rechace en caso de no ser válida. Otra manera de introducir validaciones es mediante el código del programa, lo cual veremos cuando manejemos datos usando código.

Acceso a información bajo cualquier formato

El concepto de datos y bases de datos es algo que está cambiando rápidamente. Tradicionalmente la información residía en bases de datos que la organizaban en tablas y la ofrecían al usuario mediante ciertas utilidades o programas que accedían a ellas.

Sin embargo, el uso de aplicaciones que en un principio no estaban pensadas para organizar datos al uso y manera de una base de datos habitual (procesadores de texto, hojas de cálculo, páginas web, etc), y la continua expansión en la transferencia de información bajo estos formatos, sobre todo gracias a Internet, plantean el problema de convertir dicha información para que pueda ser leida desde una base de datos, o desarrollar un sistema que permita acceder y organizar dichos datos allá donde residan.

Hasta ahora, y debido a que el volumen de datos en este tipo de formato no era muy elevado, se incluían diferentes conversores en la base de datos para poder integrar el contenido de tales ficheros en tablas. Pero actualmente nos encontramos ante el problema del gran incremento de información a consultar bajo este tipo de ficheros, y la necesidad de realizar búsquedas y editar dicha información

directamente en el formato en que reside. Entre las diferentes soluciones planteadas, el Acceso Universal a Datos propuesto por Microsoft, es una de las principales opciones a tener en cuenta.

Acceso Universal a Datos (UDA)

El Acceso Universal a Datos (Universal Data Access), es una especificación establecida por Microsoft para el acceso a datos de distinta naturaleza, que se basa en la manipulación de los datos directamente en el formato en el que han sido creados. ¿Por qué emplear esfuerzos en realizar conversiones entre formatos?, lo lógico y natural es manipular los datos directamente allá donde residan. Pues bien, mediante este nuevo paradigma, y empleando OLE DB y ADO, podremos acceder de una forma sencilla a un gran espectro de tipos de datos, como muestra el esquema de la figura 167, sin perder un ápice de la potencia de que disponíamos usando DAO y RDO. Es más, mediante un modelo de objetos más simple y flexible, podremos obtener mejores resultados.

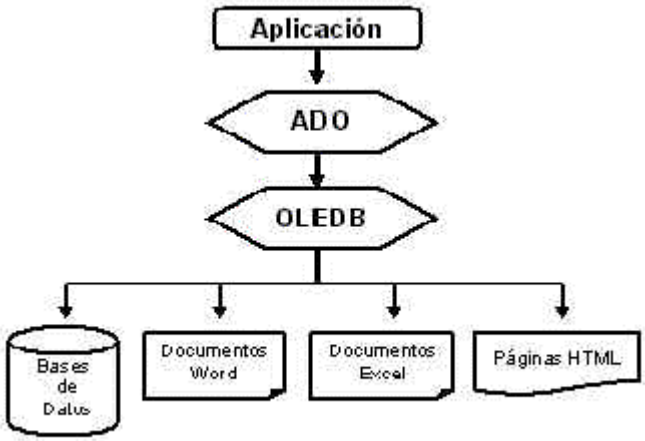


Figura 167. Acceso a datos bajo Universal Data Access.

OLE DB

OLE DB ha sido creado para sustituir progresivamente a ODBC. Los motivos principales son, por un lado, la ya comentada necesidad de acceder a información que reside en formatos no relacionales de datos. Por otra parte, cuando al navegar por Internet se realizan accesos a datos, la propia naturaleza de la Red no asegura que durante la consulta a una determinada página, la conexión a dichos datos vaya a mantenerse todo el tiempo que estamos conectados a dicha dirección. Finalmente está el objetivo de Microsoft, de que todo el acceso a objetos sea realizado mediante su especificación COM (Modelo de Objetos basado en Componentes).

ODBC está orientado a trabajar con información que reside en orígenes de datos tradicionales, y aunque se seguirá dando soporte a esta tecnología, los futuros objetivos van encaminados a utilizar OLE DB.

OLE DB se compone de un conjunto de interfaces de bajo nivel, que utilizan el modelo de componentes para proporcionar acceso a prácticamente cualquier tipo de formato de datos. Desde los relacionales, como puedan ser bases de datos Jet, SQL Server, Oracle; hasta no relacionales, como ficheros de correo electrónico, documentos Word, Excel, páginas web, etc.

La figura 168, muestra el modelo de objetos de la jerarquía OLE DB, seguido de una breve descripción de cada uno.

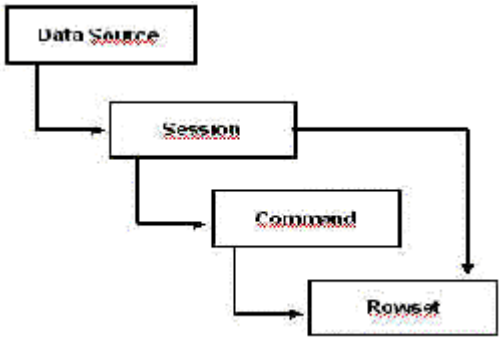


Figura 168. Modelo de objetos OLE DB.

- Data Source. Realiza las comprobaciones sobre el proveedor de datos a utilizar, los permisos de que dispone el usuario, e inicializa la conexión con la fuente de datos.

- Session. Contiene el código necesario para conectar con una fuente de datos.

- Command. Este objeto es el que permite definir los elementos que componen la base de datos y su manipulación, por ejemplo: la creación de tablas y ejecución de instrucciones; siempre y cuando el proveedor lo permita.

- Rowset. Los objetos de este tipo contienen un conjunto de filas, resultado de la ejecución de un objeto Command, o creados desde una sesión.

Al igual que en ODBC existen los conceptos de Cliente y Servidor, en OLE DB disponemos del Proveedor (Provider), o elemento que proporciona los datos; y el Consumidor (Consumer), o elemento que utiliza los datos que le facilita el proveedor. Entre los proveedores incluidos con la versión de OLE DB que acompaña a Visual Basic 6, destacamos los que aparecen en la tabla 46.

Proveedor	**Tipo de dato**
Microsoft Jet 3.51 OLE DB Provider.	Proporciona acceso a base de datos Jet.
Microsoft OLE DB Provider for ODBC Drivers.	Proporciona acceso a fuentes de datos ODBC.
Microsoft OLE DB Provider for Oracle.	Proporciona acceso a bases de datos Oracle.
Microsoft OLE DB Provider for SQL Server.	Proporciona acceso a bases de datos SQL Server.

Tabla 46. Proveedores OLE DB incluidos en Visual Basic.

Si nos encontramos ante el caso de tener que manipular un formato de datos aún no contemplado por ningún fabricante de proveedores OLE DB, mediante el OLE DB SDK, podemos desarrollar nuestros propios proveedores personalizados.

Debido a la complejidad de sus interfaces, la mejor forma de trabajar con OLE DB es utilizando objetos ADO, los cuales proporcionan un acceso simplificado a OLE DB, evitándonos el trabajo con

los aspectos más complicados del mismo. La figura 169, muestra el esquema de acceso a datos desde VB empleando OLE DB/ADO.

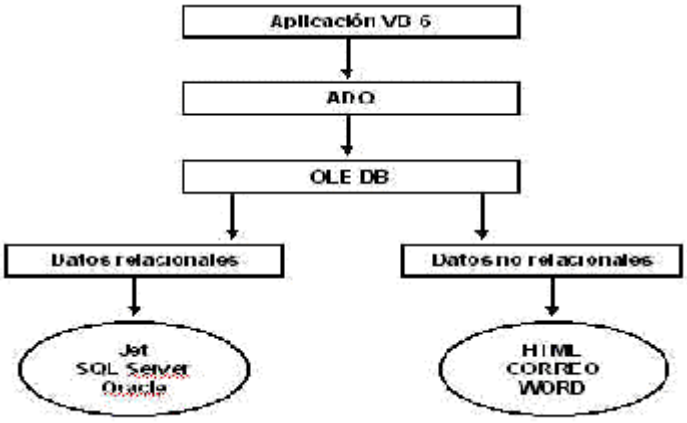


Figura 169. Modo de acceso a datos mediante OLE DB/ADO.

ActiveX Data Objects (ADO)

La jerarquía de objetos ADO, mediante un diseño más simple en comparación con las otras jerarquías de acceso a datos disponibles hasta la actualidad (DAO y RDO), y con un mayor nivel de rendimiento, se constituye a partir de Visual Basic 6, en el nuevo paradigma para la manipulación de datos facilitados por OLE DB. La propia Microsoft, recomienda encarecidamente emplear esta tecnología en todos los desarrollos que se inicien con la nueva versión de VB. Entre sus diferentes características, podemos destacar las siguientes:

- Posibilidad de crear objetos del modelo ADO, sin necesidad de navegar a través de la jerarquía de objetos. De esta forma, sólo utilizaremos los objetos realmente necesarios para nuestros propósitos.
- Independencia del lenguaje.
- Mínimos requerimientos de memoria y de espacio en disco.
- Gran velocidad de ejecución.
- Es posible llamar a procedimientos almacenados con parámetros de entrada/salida.
- Amplia variedad de tipos de cursor.
- Devolución de conjuntos de resultados múltiples desde una única consulta.
- Ejecución de consultas síncronas, asíncronas o basadas en eventos.
- Permite trabajar de forma flexible tanto con las bases de datos existentes como con los proveedores de OLE DB.
- Manejo de errores optimizado.

La figura 170 muestra el modelo de objetos ADO:

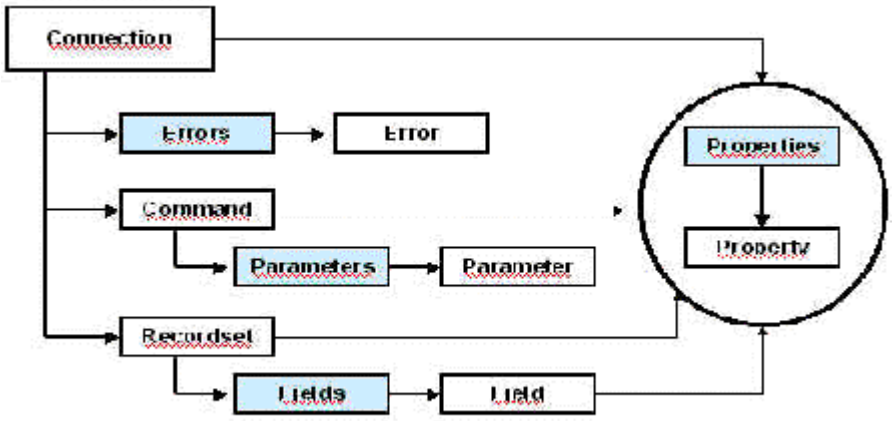


Figura 170. Jerarquía de objetos ADO.

En el caso de las aplicaciones existentes desarrolladas con DAO, y debido a que ADO tiene un caracter claramente más abierto en cuanto al tipo de datos a manejar, el aspecto de la migración de las aplicaciones creadas con DAO a ADO entraña ciertas dificultades que no permiten la simple recompilación del programa para convertirlo a esta nueva tecnología. Por esto se debe realizar un detallado análisis de la aplicación, para determinar si realmente es necesario o merece la pena el trabajo a dedicarle para convertir el código al nuevo modelo de objetos.

A continuación, se describen las principales características de cada uno de los objetos.

Connection

Un objeto Connection representa una conexión establecida con una base de datos. En función del tipo de fuente de datos al que nos conectemos, algunas propiedades o métodos de este objeto pueden no estar disponibles, por lo que recomendamos al lector que revise las especificaciones de la base de datos con la cual desea trabajar.

Propiedades

- Attributes. Constante de tipo Long, que indica las características de la conexión, según la tabla 47.

Constante	**Descripción**
adXactCommitRetaining	Al efectuar una llamada a CommitTrans, se inicia una nueva transacción.
adXactAbortRetaining	Al efectuar una llamada a RollbackTrans, se inicia una nueva transacción.

Tabla 47. Constantes para la propiedad Attributes.

- CommandTimeout. Numérico de tipo Long, que indica el número de segundos que un objeto Command intentará procesar un comando antes de generar un error. El valor por defecto es 30, pero si es cero, no habrá límite en el tiempo de espera.

- ConnectionString. Cadena con la información que utilizará el objeto para establecer la conexión. Dentro de esta cadena, se pueden pasar varios parámetros con sus correspondientes valores, siempre que tengan el formato: parámetro = valor, separados por punto y coma.

  Los parámetros empleados en esta cadena que pueden ser procesados por ADO se describen en la tabla 48.

Parámetro	**Descripción**
Provider	Nombre del proveedor con el que se efectúa la conexión.
File Name	Fichero que contiene información sobre la conexión a establecer. Debido a que con este parámetro se carga el proveedor asociado, no se puede utilizar conjuntamente con Provider.
Remote Provider	Nombre del proveedor que se usa al abrir una conexión del lado del cliente (sólo Remote Data Service).
Remote Server	Ruta de acceso del servidor que se usa al abrir una conexión del lado del cliente (sólo Remote Data Service).

Tabla 48. Valores para ConnectionString.

Cualquier otro parámetro es enviado directamente al proveedor OLE DB correspondiente para ser procesado.

- ConnectionTimeout. Valor de tipo Long, que se utiliza para establecer el tiempo que se intentará establecer la conexión antes de generar un error. El valor por defecto es 15, en el caso de que se utilice cero, no existirá límite de tiempo en el intento de conexión.

- CursorLocation. Permite establecer la situación del cursor disponible por el proveedor, del lado del cliente o del lado del proveedor. Las constantes para esta propiedad se pueden ver en la tabla 49.

Constante	**Descripción**
adUseNone	No se utilizan servicios de cursor. Este valor sólo se proporciona por compatibilidad con versiones anteriores.
adUseClient	Usa cursores del lado del cliente. También se puede utilizar la constante adUseClientBatch.
adUseServer	Predeterminado. Utiliza cursores del lado del servidor, suministrados por el controlador o por el proveedor de datos. Aunque este tipo de cursores suelen ser muy potentes y flexibles, puede que alguna característica no esté contemplada.

Tabla 49. Constantes para la propiedad CursorLocation.

- DefaultDatabase. Cadena con el nombre de la base de datos predeterminada para el objeto Connection.

- IsolationLevel. Constante que establece el nivel de aislamiento en las transacciones, según la tabla 50.

Constante	**Descripción**
adXactUnspecified	El proveedor está utilizando un nivel de aislamiento que no se puede determinar.
adXactChaos	Predeterminado. Impide sobrescribir cambios pendientes de transacciones con un nivel de aislamiento más alto.
adXactBrowse	Permite ver desde una transacción, cambios que no se han producido en otras transacciones.
adXactReadUncommitted	Realiza la misma función que adXactBrowse.
adXactCursorStability	Desde una transacción se podrán ver cambios en otras transacciones sólo después de que se hayan producido.
adXactReadCommitted	Igual que adXactCursorStability.
adXactRepeatableRead	Indica que desde una transacción no se pueden ver los cambios producidos en otras transacciones, pero que al volver a hacer una consulta se pueden obtener nuevos Recordsets.
adXactIsolated	Las transacciones se realizan aisladamente de otras transacciones.
adXactSerializable	Igual que adXactIsolated.

Tabla 50. Constantes para la propiedad IsolationLevel.

- Mode. Constante que indica los permisos de acceso a datos para una conexión, según la tabla 51.

Constante	Descripción
adModeUnknown	Predeterminado. Indica que los permisos no se han establecido aún o que no se pueden determinar.
adModeRead	Indica que son permisos de sólo lectura.
adModeWrite	Indica que son permisos de sólo escritura.
adModeReadWrite	Indica que son permisos de lectura/escritura.
adModeShareDenyRead	Impide que otros abran una conexión con permisos de lectura.

adModeShareDenyWrite	Impide que otros abran una conexión con permisos de escritura.
adModeShareExclusive	Impide que otros abran una conexión.
adModeShareDenyNone	Impide que otros abran una conexión con cualquier tipo de permiso.

Tabla 51. Permisos disponibles para un objeto Connection.

- Provider. Cadena con el nombre del proveedor de datos.
- State. Indica el estado del objeto: abierto o cerrado.

Constante	**Descripción**
adStateClosed	Predeterminado. El objeto está cerrado.
adStateOpen	El objeto está abierto.

Tabla 52. Constantes de estado para un objeto Connection.

- Version. Retorna una cadena con el número de versión de ADO.

Métodos

- BeginTrans(), CommitTrans(), RollbackTrans(). Gestionan las transacciones dentro de un objeto Connection. BeginTrans inicia una transacción; CommitTrans guarda en la base de datos los cambios realizados durante la transacción; RollbackTrans cancela las operaciones efectuadas durante la transacción y finaliza la misma.

  En el caso de BeginTrans, puede devolver un valor Long que indica el nivel de anidamiento de la transacción, utilizando la siguiente sintaxis:

  ```
  lNivel = oConnection.BeginTrans()
  ```

  Antes de intentar utilizar transacciones, es necesario comprobar que en la colección Properties del objeto, aparece la propiedad Transaction DDL, para asegurarnos que el proveedor las acepta.

- Cancel(). Cancela la ejecución de un método Execute u Open, cuando estos han sido llamados de forma asíncrona.

- Close(). Cierra un objeto Connection, liberando los recursos que estuviera utilizando. Este método no elimina el objeto de la memoria, ya que es posible modificar algunas de sus propiedades y volver a utilizarlo. Para eliminarlo por completo de la memoria es necesario asignarle Nothing.

- Execute(). Ejecuta una sentencia SQL, procedimiento almacenado, etc., sobre la base de datos que utiliza el objeto Connection.

- Sintaxis para un objeto que no devuelva filas:

```
oConnection.Execute cComando, lRegistrosAfectados, lOpciones
```

- Sintaxis para un objeto que devuelva filas. En este caso se obtiene un objeto Recordset de tipo Forward-only:

```
Set oRecordset = oConnection.Execute(cComando,
lRegistrosAfectados, lOpciones)
```

- Parámetros:

  - cComando. Cadena con la sentencia SQL, nombre de la tabla, procedimiento almacenado, etc.

  - lRegistrosAfectados. Opcional. Valor Long en la que el proveedor deposita el número de registros afectados por la operación

  - lOpciones. Opcional. Constante que indica al proveedor como debe evaluar el parámetro cCommandText. La tabla 53 muestra los valores disponibles.

Constante	**Descripción**
adCmdText	El parámetro cCommandText se evalúa como una instrucción SQL.
adCmdTable	ADO crea una consulta SQL que devuelve las filas de la tabla que se indica en cCommandText.
adCmdTableDirect	El resultado es una representación física de la tabla indicada en cCommandText.
adCmdStoredProc	El proveedor tiene que evaluar cCommandText como procedimiento almacenado.
adCmdUnknown	El tipo de comando en cCommandText es desconocido.
adExecuteAsync	Indica que el comando se tiene que ejecutar de forma asíncrona.
adFetchAsync	Indica que el resto de las filas siguientes a la cantidad inicial especificada en la propiedad CacheSize tiene que ser recuperada de forma asíncrona.

Tabla 53. Constantes utilizadas en el parámetro lOptions del método Execute.

Al finalizar la ejecución de este método, se genera un evento ExecuteComplete.

- Open(). Establece una conexión con una fuente de datos.

- Sintaxis:

```
oConnection.Open cCadenaConexion, cIDUsuario, cPassword,
xOpcionesApertura
```

- Parámetros:

  - cCadenaConexion. Opcional. Cadena con la información para establecer la conexión. El valor de este parámetro es igual que para la propiedad ConnectionString de este objeto.

  - cIDUsuario. Opcional. Cadena con el nombre de usuario que se va a conectar. Este valor tendrá prioridad en el caso de que también se utilice en cCadenaConexion.

  - cPassword. Opcional. Cadena con la clave de acceso del usuario. Este valor tendrá prioridad en el caso de que también se utilice en cCadenaConexion.

  - xOpcionesApertura. Opcional. Valor de tipo ConnectOptionEnum. Cuando este valor sea adConnectAsync, la conexión se abrirá de forma asíncrona.

Una vez se haya realizado la conexión, se generará un evento ConnectComplete.

- OpenSchema(). Proporciona información sobre el esquema o estructura interna de la base de datos sobre la que se ha establecido la conexión. Dicha información se devuelve en forma de recordset estático de sólo lectura.

  - Sintaxis:

```
Set oRecordset = oConnection.OpenSchema(xTipoConsulta,
cCriterio, IDEsquema)
```

  - Parámetros:

    - xTipoConsulta. Constante que indica el tipo de esquema a consultar.

    - cCriterio. Según la constante utilizada en xTipoConsulta, este parámetro contendrá un array con los valores disponibles para cada tipo de consulta. Los valores combinados de xTipoConsulta y cCriterio se muestran en la tabla 54.

Valores de QueryType	Valores de Criterio
adSchemaAsserts	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME
adSchemaCatalogs	CATALOG_NAME
adSchemaCharacterSets	CHARACTER_SET_CATALOG CHARACTER_SET_SCHEMA CHARACTER_SET_NAME
adSchemaCheckConstraints	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME

adSchemaCollations	COLLATION_CATALOG COLLATION_SCHEMA COLLATION_NAME
adSchemaColumnDomainUsage	DOMAIN_CATALOG DOMAIN_SCHEMA DOMAIN_NAME COLUMN_NAME
adSchemaColumnPrivileges	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME GRANTOR GRANTEE
adSchemaColumns	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
adSchemaConstraintColumnUsage	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
adSchemaConstraintTableUsage	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaForeignKeys	PK_TABLE_CATALOG PK_TABLE_SCHEMA PK_TABLE_NAME FK_TABLE_CATALOG FK_TABLE_SCHEMA FK_TABLE_NAME
adSchemaIndexes	TABLE_CATALOG TABLE_SCHEMA INDEX_NAME TYPE TABLE_NAME
adSchemaKeyColumnUsage	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
adSchemaPrimaryKeys	PK_TABLE_CATALOG PK_TABLE_SCHEMA PK_TABLE_NAME

adSchemaProcedureColumns	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME COLUMN_NAME
adSchemaProcedureParameters	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PARAMTER_NAME
adSchemaProcedures	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PARAMTER_TYPE
adSchemaProviderSpecific	Para proveedores no estándar.
adSchemaProviderTypes	DATA_TYPE BEST_MATCH
adSchemaReferentialConstraints	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME
adSchemaSchemata	CATALOG_NAME SCHEMA_NAME SCHEMA_OWNER
adSchemaSQLLanguages	<ninguno>
adSchemaStatistics	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaTableConstraints	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME CONSTRAINT_TYPE
adSchemaTablePrivileges	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME GRANTOR GRANTEE
adSchemaTables	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE
adSchemaTranslations	TRANSLATION_CATALOG TRANSLATION_SCHEMA TRANSLATION_NAME

adSchemaUsagePrivileges	OBJECT_CATALOG OBJECT_SCHEMA OBJECT_NAME OBJECT_TYPE GRANTOR GRANTEE
adSchemaViewColumnUsage	VIEW_CATALOG VIEW_SCHEMA VIEW_NAME
adSchemaViewTableUsage	VIEW_CATALOG VIEW_SCHEMA VIEW_NAME
AdSchemaViews	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME

Tabla 54. Valores para xTipoConsulta y cCriterio.

- IDEsquema. Identificador global para un esquema de un proveedor no definido en la especificación OLE DB. Este parámetro es necesario cuando el valor de xTipoConsulta sea adSchemaProviderSpecific.

Command

Este objeto representa un comando emitido hacia una fuente de datos para consulta, edición de registros, etc.

Propiedades

- ActiveConnection. Contiene el objeto Connection al que pertenece el Command
- CommandText. Cadena con una sentencia SQL, tabla o procedimiento almacenado a ejecutar.
- CommandTimeout. Numérico de tipo Long, que indica el número de segundos que un objeto Connection intentará procesar un comando antes de generar un error. El valor por defecto es 30, pero si es cero, no habrá límite en el tiempo de espera.
- CommandType. Establece el tipo de comando a ejecutar, optimizando la evaluación de la propiedad CommandText. Dentro de la tabla 55 se muestran las constantes disponibles para esta propiedad.

  Debemos ser cuidadosos a la hora de establecer esta propiedad, ya que si no guarda relación con el valor que contiene CommandText, se producirá un error al intentar ejecutar el objeto Command.

Constante	Descripción
AdCmdText	Evalúa CommandText como una instrucción SQL o una llamada a un procedimiento almacenado.
AdCmdTable	Crea una consulta SQL que devuelve las filas de la tabla que se indica en CommandText.
adCmdTableDirect	El resultado es una representación física de la tabla indicada en CommandText.
adCmdStoredProc	Evalúa CommandText como un procedimiento almacenado.
adCmdUnknown	Predeterminado. El tipo de comando de la propiedad CommandText es desconocido, por lo cual ADO debe realizar llamadas previas al proveedor para determinar el contenido de CommandText, y así, ejecutarlo adecuadamente. Por este motivo, el rendimiento en la ejecución se ve penalizado, razón por la que debemos evitar en lo posible el uso de este tipo de comando.
adCommandFile	Evalúa CommandText como el nombre de archivo de un valor Recordset persistente.
adExecuteNoRecords	Se utiliza cuando CommandText contiene una instrucción o procedimiento almacenado que no devuelve filas (como una modificación en el valor de una columna de una tabla). En el caso de recuperar alguna fila, no se devuelve. Siempre se combina con adCmdText o adCmdStoredProc. El uso de esta constante mejora el rendimiento en la ejecución del objeto Command, al procesarse de forma interna.

Tabla 55. Constantes para la propiedad CommandType.

- Prepared. Contiene un valor lógico que si es True, indica al proveedor que debe preparar/compilar la consulta contenida en el objeto antes de ejecutarla por primera vez. Esto causará una disminución de rendimiento en esa primera ejecución, pero al encontrarse compilado en las posteriores llamadas, se acelerará esa misma ejecución.

  Cuando el valor de esta propiedad sea False, se ejecutará el comando sin realizar su compilación.

  En el caso de que el proveedor no admita la compilación del comando, puede generar un error al intentar asignar el valor True a esta propiedad o simplemente no compilar el objeto.

- State. Indica el estado del objeto: abierto o cerrado. Consultar esta misma propiedad en el objeto Connection.

Métodos

- Cancel(). Cancela la ejecución de un método Execute, que ha sido llamado de forma asíncrona.

- CreateParameter(). Crea un objeto Parameter para un comando. Este método no agrega el nuevo parámetro a la colección Parameters del objeto Command. Para más información sobre el objeto Parameter, consulte el lector, el apartado dedicado a dicho objeto en este mismo tema.

  - Sintaxis:

    ```
    Set oParameter = oCommand.CreateParameter(cNombre, lTipo,
    lDireccion, lTamaño, lValor)
    ```

- Execute(). Ejecuta la consulta SQL o procedimiento almacenado contenido en la propiedad CommandText.

  - Sintaxis para un objeto que devuelva filas:

    ```
    Set oRecordset = oCommand.Execute(lRegistrosAfectados,
    aParametros, lOpciones)
    ```

  - Sintaxis para un objeto que no devuelva filas:

    ```
    oCommand.Execute lRegistrosAfectados, aParametros, lOpciones
    ```

  - Parámetros:

    - lRegistrosAfectados. Opcional. Valor Long en la que el proveedor deposita el número de registros afectados por la operación.

    - aParametros. Opcional. Array de tipo Variant, con los valores de los parámetros utilizados por la instrucción SQL del comando.

    - lOpciones. La funcionalidad es la misma que en el objeto Connection, consulte el lector este mismo método en dicho objeto.

  Una vez finalizada la ejecución de este método, se genera un evento ExecuteComplete.

Parameter

Este tipo de objeto, representa un parámetro asociado a un objeto Command de tipo consulta parametrizada o procedimiento almacenado, de manera que podamos ejecutar el mismo comando obteniendo diferentes resultados en función de los valores pasados en los parámetros.

Propiedades

- Attributes. Constante Long, que especifica las características del objeto según la tabla 56.

Constante	Descripción
adParamSigned	Predeterminado. El parámetro acepta valores firmados.
adParamNullable	Indica que el parámetro acepta valores Null.
adParamLong	Indica que el parámetro acepta datos binarios largos.

Tabla 56. Constantes para la propiedad Attributes.

- Direction. Constante que indica la dirección del objeto Parameter, en función de las constantes de la tabla 57.

Constante	Descripción
adParamUnknown	Indica que la dirección del parámetro es desconocida.
adParamInput	Predeterminado. El parámetro es de entrada.
adParamOutput	Indica un parámetro de salida.
adParamInputOutput	Indica un parámetro de entrada y otro de salida.
adParamReturnValue	Indica un valor devuelto.

Tabla 57. Constantes para la propiedad Direction.

- Name. Cadena con el nombre del objeto.
- NumericScale. Número de tipo Byte que indica la cantidad de posiciones decimales para los parámetros con valores numéricos.
- Precision. Dato Byte que indica la cantidad de dígitos usados para representar los parámetros con valores numéricos.
- Size. Número Long que establece el tamaño en bytes o caracteres del parámetro. Si el tipo de datos es de longitud variable, esta propiedad debe ser establecida antes de añadir el objeto a la colección Parameters, o se producirá un error.
- Type. Indica el tipo de datos para el objeto. La tabla 58 muestra las constantes para los tipos de datos disponibles en el modelo ADO. El tipo correspondiente para OLE DB se muestra entre paréntesis.

Constante	Descripción
adArray	Se une en una instrucción OR lógica con otro tipo para indicar que los datos son una matriz segura de ese tipo (DBTYPE_ARRAY).

adBigInt	Un entero con signo de 8 bytes (DBTYPE_I8).
adBinary	Un valor binario (DBTYPE_BYTES).
adBoolean	Un valor Boolean (DBTYPE_BOOL).
adByRef	Se une en una instrucción OR lógica con otro tipo para indicar que los datos son un puntero a los datos del otro tipo (DBTYPE_BYREF).
adBSTR	Una cadena de caracteres terminada en nulo (Unicode) (DBTYPE_BSTR).
adChar	Un valor de tipo String (DBTYPE_STR).
adCurrency	Un valor de tipo Currency (DBTYPE_CY). Un valor Currency es un número de coma fija con cuatro dígitos a la derecha del signo decimal. Se almacena en un entero con signo de 8 bytes en escala de 10.000.
adDate	Un valor de tipo Date (DBTYPE_DATE). Un valor Date se almacena como un valor de tipo Double; la parte entera es el número de días transcurridos desde el 30 de diciembre de 1899 y la parte fraccionaria es la fracción de un día.
adDBDate	Un valor de fecha (aaaammdd) (DBTYPE_DBDATE).
adDBTime	Un valor de hora (hhmmss) (DBTYPE_DBTIME).
adDBTimeStamp	Una marca de fecha y hora (aaaammddhhmmss más una fracción de miles de millones) (DBTYPE_DBTIMESTAMP).
adDecimal	Un valor numérico exacto con una precisión y una escala fijas (DBTYPE_DECIMAL).
adDouble	Un valor de coma flotante de doble precisión (DBTYPE_R8).
adEmpty	No se ha especificado ningún valor (DBTYPE_DBDATE)
adError	Un código de error de 32 bits (DBTYPE_ERROR).
adGUID	Un identificador único global (GUID) (DBTYPE_GUID).
adIDispatch	Un puntero a una interfaz Idispatch de un objeto OLE (DBTYPE_IDISPATCH).
adInteger	Un entero firmado de 4 bytes (DBTYPE_I4).

adIUnknown	Un puntero a una interfaz Iunknown de un objeto OLE (DBTYPE_IUNKNOWN).
adLongVarBinary	Un valor binario largo (sólo para el objeto Parameter).
adLongVarChar	Un valor largo de tipo String (sólo para el objeto Parameter).
adLongVarWChar	Un valor largo de tipo String terminado en nulo (sólo para el objeto Parameter).
adNumeric	Un valor numérico exacto con una precisión y una escala exactas (DBTYPE_NUMERIC).
adSingle	Un valor de coma flotante de simple precisión (DBTYPE_R4).
adSmallInt	Un entero con signo de 2 bytes (DBTYPE_I2).
adTinyInt	Un entero con signo de 1 byte (DBTYPE_I1).
adUnsignedBigInt	Un entero sin signo de 8 bytes (DBTYPE_UI8).
adUnsignedInt	Un entero sin signo de 4 bytes (DBTYPE_UI4).
adUnsignedSmallInt	Un entero sin signo de 2 bytes (DBTYPE_UI2).
adUnsignedTinyInt	Un entero sin signo de 1 byte (DBTYPE_UI1).
adUserDefined	Una variable definida por el usuario (DBTYPE_UDT).
adVarBinary	Un valor binario (sólo para el objeto Parameter).
adVarChar	Un valor de tipo String (sólo para el objeto Parameter).
adVariant	Un tipo Variant de automatización (DBTYPE_VARIANT).
adVector	Se une en una instrucción OR lógica con otro tipo para indicar que los datos son una estructura DBVECTOR, tal como está definida por OLE DB, que contiene un contador de elementos y un puntero a los datos del otro tipo (DBTYPE_VECTOR).
adVarWChar	Una cadena de caracteres Unicode terminada en nulo (sólo para el objeto Parameter).
adWChar	Una cadena de caracteres Unicode terminada en nulo (DBTYPE_WSTR).

Tabla 58. Tipos de datos de la jerarquía de objetos ADO.

- Value. Contiene un dato Variant con el valor asignado al objeto.

Métodos

- AppendChunk(). Permite añadir un texto de gran tamaño o datos binarios a un Parameter.
  - Sintaxis

    ```
    oParameter.AppendChunk vntDatos
    ```

  - Parámetros
    - vntDatos. Valor Variant que contiene la información a añadir.

  Si el bit adFldLong de la propiedad Attributes de un objeto Parameter tiene el valor True, se puede utilizar el método AppendChunk en dicho parámetro.

La primera vez que se llama a AppendChunk, se escriben los datos en el parámetro sobrescribiendo la información anterior. Las siguientes llamadas a este método, agregan los nuevos datos a los ya existentes. Si el valor pasado es nulo, se ignoran todos los datos del parámetro.

Recordset

Un objeto Recordset representa un conjunto de registros, bien puede ser la totalidad de registros de una tabla o un grupo más pequeño, resultado de la ejecución de una consulta SQL de selección, y que contenga registros de una o varias tablas.

Propiedades

- AbsolutePage. Informa de la página en la que está el registro actual. Las constantes que aparecen en la tabla 59 nos ayudan en la identificación de la página.

Constante	Descripción
adPosUnknown	El objeto Recordset está vacío, la posición actual se desconoce o el proveedor no admite la propiedad AbsolutePage.
adPosBOF	El puntero del registro actual está al comienzo del archivo (es decir, la propiedad BOF tiene el valor True).
adPosEOF	El puntero del registro actual está al final del archivo (es decir, la propiedad EOF tiene el valor True).

Tabla 59. Constantes para la propiedad AbsolutePage.

- AbsolutePosition. Valor Long que informa sobre el número que tiene un registro dentro de un Recordset, o bien puede ser una de las constantes ya especificadas en la propiedad AbsolutePage.

El uso de esta propiedad, aunque pueda ser de utilidad en casos muy determinados, no es en absoluto recomendable, ya los números de registro que maneja un objeto Recordset pueden cambiar bajo variadas circunstancias. Por este motivo, la forma más recomendable para guardar posiciones de registros siguen siendo los bookmarks.

- ActiveConnection. Contiene el objeto Connection al que pertenece el Recordset.
- BOF - EOF. Indican el comienzo y final del Recordset respectivamente.
- Bookmark. Valor que identifica un registro, de forma que podamos guardar su posición para volver a él en cualquier momento de la ejecución.
- CacheSize. Devuelve un número Long que indica la cantidad de registros que están en la memoria caché local.
- CursorLocation. Desempeña la misma función que en el objeto Connection. Consulte el lector el apartado sobre dicho objeto.
- CursorType. Establece el tipo de cursor para el Recordset, según las constantes que aparecen en la tabla 60.

Constante	**Descripción**
adOpenForwardOnly	Predeterminado. Cursor de tipo Forward-only. Idéntico a un cursor estático, excepto que sólo permite desplazarse hacia delante en los registros. Esto mejora el rendimiento en situaciones en las que sólo se quiere pasar una vez por cada registro.
adOpenKeyset	Cursor de conjunto de claves. Igual que un cursor dinámico, excepto que no se pueden ver los registros que agregan otros usuarios, aunque los registros que otros usuarios eliminan son inaccesibles desde su conjunto de registros. Los cambios que otros usuarios hacen en los datos permanecen visibles.
adOpenDynamic	Cursor dinámico. Las incorporaciones, cambios y eliminaciones que hacen otros usuarios permanecen visibles, y se admiten todo tipo de movimientos entre registros, a excepción de los marcadores si el proveedor no los admite.
adOpenStatic	Cursor estático. Una copia estática de un conjunto de registros que se puede usar para buscar datos o generar informes. Las incorporaciones, cambios o eliminaciones que hacen otros usuarios no son visibles.

Tabla 60. Constantes disponibles en CursorType.

- EditMode. Devuelve un valor que indica el estado de edición del registro actual. Las constantes asociadas aparecen en la tabla 61.

Constante	**Descripción**
adEditNone	Indica que no hay ninguna operación de modificación en ejecución.
adEditInProgress	Indica que los datos del registro actual se han modificado pero que no se han guardado aún.
adEditAdd	Indica que se ha invocado el método AddNew y que el registro situado actualmente en el búfer de copia es un nuevo registro que no se ha guardado en la base de datos.
adEditDelete	Indica que el registro actual se ha eliminado.

Tabla 61. Valores devueltos por la propiedad EditMode.

- Filter. Valor Variant, que establece un filtro para los registros de un Recordset. Puede estar compuesto por una o varias cadenas con el criterio de ordenación concatenadas mediante los operadores AND u OR; un array de Bookmarks; o una de las constantes que aparecen en la tabla 62.

Constante	**Descripción**
adFilterNone	Elimina el filtro actual y vuelve a poner todos los registros a la vista.
adFilterPendingRecords	Permite ver sólo los registros que han cambiado pero que no han sido enviados aún al servidor. Aplicable sólo para el modo de actualización por lotes.
adFilterAffectedRecords	Permite ver sólo los registros afectados por la última llamada a Delete, Resync, UpdateBatch o CancelBatch.
adFilterFetchedRecords	Permite ver los registros de la caché actual, es decir, los resultados de la última llamada para recuperar registros de la base de datos.
adFilterConflictingRecords	Permite ver los registros que fallaron en el último intento de actualización por lotes.

Tabla 62. Constantes a utilizar en la propiedad Filter.

- LockType. Indica el tipo de bloqueo para los registros durante su edición. Las constantes para esta propiedad se muestran en la tabla 63.

- MarshalOptions. Al utilizar recordsets del lado del cliente, esta propiedad mejora la comunicación entre esos tipos de recordset y el servidor, estableciendo que registros se devuelven al servidor (tabla 64).

Constante	Descripción
adLockReadOnly	Predeterminado. Sólo lectura.
adLockPessimistic	Bloqueo pesimista, registro a registro: el proveedor hace lo necesario para asegurar la modificación correcta de los registros, generalmente bloqueando registros en el origen de datos durante el proceso de modificación.
adLockOptimistic	Bloqueo optimista, registro a registro: el proveedor usa bloqueo optimista, bloqueando registros sólo cuando llama al método Update.
adLockBatchOptimistic	Actualizaciones optimistas por lotes: requerido para el modo de actualización por lotes como contraposición al modo de actualización inmediata.

Tabla 63. Tipos de bloqueo para un Recordset.

Constante	Descripción
adMarshalAll	Predeterminado. Todas las filas se devuelven al servidor.
adMarshalModifiedOnly	Sólo las filas modificadas se devuelven al servidor.

Tabla 64. Constantes para la propiedad MarshalOptions.

- MaxRecords. Establece el número máximo de registros que se devuelven a un Recordset
- PageCount. Dato Long que informa sobre el número de páginas de un Recordset.
- PageSize. Valor Long que contiene el número de registros que constituyen una página de un objeto Recordset.
- RecordCount. Indica el número de registros que contiene un Recordset.
- Sort. Ordena uno o varios campos del Recordset en forma ascendente, descendente o combinando ambas. Para ello debemos asignar una cadena a esta propiedad con el nombre o nombres de los campos a ordenar, separados por comas e indicar el modo de ordenación mediante las palabras clave ASC o DESC.
- Source. Devuelve una cadena con el tipo de origen de datos de un Recordset.
- State. Informa sobre el estado del objeto. La tabla 65 muestra las constantes que tiene esta propiedad.
- Status. Informa sobre el estado del registro actual. En la tabla 66 se muestran los valores para esta propiedad.

Constante	**Descripción**
AdStateClosed	Predeterminado. Indica que el objeto está cerrado.
adStateOpen	Indica que el objeto está abierto.
adStateConnecting	Indica que el objeto Recordset se está conectando.
adStateExecuting	Indica que el objeto Recordset está ejecutando un comando.
adStateFetching	Indica que se está obteniendo el conjunto de filas del objeto Recordset.

Tabla 65. Constantes para los estados del objeto Recordset.

Constante	**Descripción**
adRecOK	El registro se ha actualizado correctamente.
adRecNew	El registro es nuevo.
adRecModified	El registro se ha modificado.
adRecDeleted	El registro se ha eliminado.
adRecUnmodified	El registro no se ha modificado.
adRecInvalid	El registro no se ha guardado debido a que su marcador no es válido.
adRecMultipleChanges	El registro no se ha guardado debido a que habría afectado a múltiples registros.
adRecPendingChanges	El registro no se ha guardado debido a que hace referencia a una inserción pendiente.
adRecCanceled	El registro no se ha guardado debido a que la operación se ha cancelado.
adRecCantRelease	El nuevo registro no se ha guardado debido a los bloqueos de registro existentes.
AdRecConcurrencyViolation	El registro no se ha guardado debido a que la ejecución simultánea optimista estaba en uso.
adRecIntegrityViolation	El registro no se ha guardado debido a que el usuario ha infringido las restricciones de integridad.
adRecMaxChangesExceeded	El registro no se ha guardado debido a que había demasiados cambios pendientes.

adRecObjectOpen	El registro no se ha guardado debido a un conflicto con el objeto de almacenamiento abierto.
adRecOutOfMemory	El registro no se ha guardado debido a que el equipo se ha quedado sin memoria.
adRecPermissionDenied	El registro no se ha guardado debido a que los permisos del usuario eran insuficientes.
adRecSchemaViolation	El registro no se ha guardado debido a que infringe la estructura de la base de datos subyacente.
adRecDBDeleted	El registro ya se ha eliminado del origen de datos

Tabla 66. Valores para la propiedad Status.

Métodos

- AddNew(). Crea un nuevo registro en un Recordset.
  - Sintaxis

    ```
    oRecordset.AddNew aListaCampos, aValores
    ```

  - Parámetros
    - aListaCampos. El nombre de un campo o un array con los nombres de los campos a asignar valor.
    - aValores. Un valor o un array con los valores a asignar a los campos de ListaCampos.
- Cancel(). Cancela la ejecución asíncrona del método Open().
- CancelBatch(). Cancela una actualización por lotes pendiente de procesar.
  - Sintaxis:

    ```
    oRecordset.CancelBatch xRegistrosAfectados
    ```

  - Parámetros
    - xRegistrosAfectados. Opcional. Constante que indica los registros que resultarán afectados con este método. Lo vemos en la tabla 67.

Constante	**Descripción**
adAffectCurrent	Sólo cancela las actualizaciones del registro actual.

adAffectGroup	Cancela las actualizaciones pendientes de los registros que cumplen el valor actual de la propiedad Filter. Para poder utilizar esta opción tiene que establecer la propiedad Filter a una de las constantes predefinidas válidas.
adAffectAll	Predeterminado. Cancela las actualizaciones pendientes de todos los registros del objeto Recordset, incluyendo los no visibles debido al valor actual de la propiedad Filter

Tabla 67. Constantes para los registros afectados en el método CancelBatch().

- CancelUpdate(). Cancela la edición de un registro nuevo o existente antes de llamar al método Update().

- Clone(). Crea un Recordset duplicado. Este método sólo está disponible sobre un Recordset que admita marcadores.

  - Sintaxis:

    ```
    Set rsDuplicado = rsOriginal.Clone(xTipoBloqueo)
    ```

  - Parámetros

    - rsDuplicado. Nuevo Recordset que se va a crear a partir de rsOriginal.

    - rsOriginal. Recordset del cual se va a crear la copia.

    - xTipoBloqueo. Opcional. Constante que especifica el tipo de bloqueo de rsOriginal. En la tabla 68 vemos los valores para el tipo de bloqueo Clone()

Constante	**Descripción**
adLockUnspecified	Predeterminado. La copia se crea con el mismo tipo de bloqueo que el original.
adLockReadOnly	Se crea una copia de sólo lectura.

Tabla 68. Valores para el tipo de bloqueo en Clone().

El uso de Clone() es particularmente útil en situaciones en las que debamos disponer de información acerca de varios registros de un mismo Recordset simultáneamente, debido a que se obtiene un mayor rendimiento que abriendo diferentes Recordset con iguales características.

Las modificaciones efectuadas en uno de los objetos, serán visibles en el resto, excepto cuando se ejecute el método Requery() en el Recordset original.

Es posible cerrar cualquiera de los Recordsets, incluido el original, sin que esto conlleve al cierre de los demás.

- Delete(). Borra el registro actual del Recordset.
  - Sintaxis:

    ```
    oRecordset.Delete xRegistrosAfectados
    ```

  - Parámetros
    - xRegistrosAfectados. Opcional. Constante que determina cuantos registros serán afectados por el borrado. Veamos la tabla 69

Constante	Descripción
adAffectCurrent	Predeterminado. Sólo elimina el registro actual.
adAffectGroup	Elimina los registros que cumplen el valor de la propiedad Filter. Tiene que establecer la propiedad Filter a una de las constantes predefinidas válidas para poder utilizar esta opción.

Tabla 69. Constantes para indicar los registros afectados por Delete().

- Find(). Busca un registro en función de un criterio. Si el criterio se cumple, el registro localizado pasa a ser el actual, en caso contrario el Recordset se sitúa al final.
  - Sintaxis

    ```
    oRecordset.Find cCriterio, lModoComienzo, kDireccion,
    vntPosComienzo
    ```

  - Parámetros:
    - cCriterio. Cadena que contiene la expresión de búsqueda incluyendo el campo en el que realizará la búsqueda, el operador y el valor a buscar como en la siguiente línea de ejemplo:

      ```
      "Nombre >= 'Pedro'"
      ```

    - lModoComienzo. Opcional. Valor Long que por defecto es cero e indica que la búsqueda se realizará desde el registro actual o desde un marcador.
    - kDireccion. Opcional. Constante que indica si la búsqueda se realizará hacia adelante (adSearchForward) o hacia atrás (adSearchBackward).
    - vntPosComienzo. Opcional. Marcador que establece la posición de comienzo de la búsqueda.
- Move(). Desplaza la posición del registro actual en un Recordset a un nuevo registro.
  - Sintaxis

    ```
    oRecordset.Move lNumRegistros, vntComienzo
    ```

- Parámetros

  - lNumRegistros. Número Long con la cantidad de registros a desplazar.

  - vntComienzo. Cadena o Variant con un Bookmark a partir del que se realizará el desplazamiento. También puede ser una de las constantes que aparecen en la tabla 70.

Constante	**Descripción**
adBookmarkCurrent	Predeterminado. Empieza en el registro actual.
adBookmarkFirst	Empieza en el primer registro.
adBookmarkLast	Empieza en el último registro.

Tabla 70. Valores para el desplazamiento con Move().

- MoveFirst(), MoveLast(), MoveNext(), MovePrevious(). Desplazan la posición del registro actual de un Recordset al primer, último, siguiente y anterior registro respectivamente. El Recordset debe admitir movimiento hacia atrás o marcadores para poder utilizar estos métodos.

- NextRecordset(). Cuando se emplee una instrucción que englobe varios comandos o un procedimiento almacenado que devuelva varios resultados, este método recupera el siguiente Recordset del conjunto de resultados.

  - Sintaxis

```
Set rsSiguiente =
rsActual.NextRecordset(lRegistrosAfectados)
```

  - Parámetros

    - lRegistrosAfectados. Dato Long que utiliza el proveedor para informar del número de registros afectados por la ejecución del método.

Este método se emplea cuando se genera un Recordset al que se le pasa una cadena con una instrucción múltiple (por ejemplo: "SELECT * FROM Clientes; SELECT * FROM Cuentas")

- Open(). Abre un cursor.

  - Sintaxis:

```
oRecordset.Open vntOrigen, vntConexionActiva, xTipoCursor,
xTipoBloqueo, xOpciones
```

  - Parámetros

    - vntOrigen. Opcional. Variant que contiene un objeto Command, una sentencia SQL, el nombre de una tabla, procedimiento almacenado, o el nombre de fichero de un Recordset.

- vntConexionActiva. Opcional. Variant que contiene un objeto Conexion, o una cadena con parámetros para efectuar una conexión.

- xTipoCursor. Opcional. Constante que indica al proveedor el tipo de cursor a emplear. Consulte el lector la propiedad CursorType de este objeto.

- xTipoBloqueo. Opcional. Constante que determina el tipo de bloqueo. Consulte el lector la propiedad LockType de este objeto.

- xOpciones. Opcional. Constante que indica el modo de evaluación de vntOrigen. Consulte el lector la propiedad CommandType del objeto Command para un detalle sobre estas constantes.

- Requery(). Vuelve a ejecutar la consulta del Recordset para actualizar completamente su contenido. Este método es equivalente a llamar a los métodos Close() y Open() sucesivamente.

- Resync(). Actualiza la información de un Recordset desde la base de datos, refrescando los registros del Recordset. Este método no ejecuta la consulta en que está basado el Recordset, por lo que los nuevos registros no serán visibles.

  - Sintaxis

    ```
    oRecordset.Resync xRegistrosAfectados, xSobreescribirValores
    ```

  - Parámetros

    - xRegistrosAfectados. Constante que indica a los registros que afectará este método, (tabla 71).

Constante	Descripción
adAffectCurrent	Actualiza solamente el registro actual.
adAffectGroup	Actualiza los registros que cumplen el valor de la propiedad Filter actual. Debe establecer la propiedad Filter a una de las constantes válidas predefinidas para poder usar esta opción.
adAffectAll	Predeterminado. Actualiza todos los registros del objeto Recordset, incluyendo cualquier registro oculto por el valor de la propiedad Filter actual.

Tabla 71. Constantes para los registros afectados por Resync().

    - xSobreescribirValores. Valor que indica si se sobrescribirán los valores subyacentes, en la tabla 72 se muestran las constantes para sobrescribir los registros en Resync().

Constante	**Descripción**
adResyncAllValues	Predeterminado. Se sobrescriben los datos y se cancelan las actualizaciones pendientes.

adResyncUnderlyingValues	No se sobrescriben los datos ni se cancelan las actualizaciones pendientes.

Tabla 72.Constantes para sobrescribir los registros en Resync().

- Save(). Guarda el Recordset en un fichero.
  - Sintaxis

    ```
    oRecordset.Save cFichero, xFormato
    ```

  - Parámetros
    - cFichero. Ruta y nombre del fichero en el que se guardará el Recordset.
    - xFormato. Opcional. Formato en el que se guardará el objeto. El único valor disponible actualmente es adPersistADTG.
- Supports(). Informa de si un Recordset soporta cierta funcionalidad, retornando un valor lógico que indica si los modos de cursor pasados como parámetro son soportados.
  - Sintaxis

    ```
    bResultado = oRecordset.Supports(lModosCursor)
    ```

  - Parámetros
    - lModosCursor. Valor Long que puede especificarse mediante las constantes que aparecen en la tabla 73.

Constante	**Descripción**
adAddNew	Puede usar el método AddNew para agregar nuevos registros.
adApproxPosition	Puede leer y establecer las propiedades AbsolutePosition y AbsolutePage.
adBookmark	Puede usar la propiedad Bookmark para tener acceso a registros específicos.
adDelete	Puede usar el método Delete para eliminar registros.
adHoldRecords	Puede recuperar más registros o cambiar la posición de recuperación siguiente sin efectuar todos los cambios pendientes.
adMovePrevious	Puede usar los métodos MoveFirst y MovePrevious, y los métodos Move o GetRows para desplazar hacia atrás la posición del registro actual sin que se requiera marcadores.
adResync	Puede actualizar el cursor con los datos visibles en la base de datos subyacente, usando el método Resync.

adUpdate	Puede usar el método Update para modificar datos existentes.
adUpdateBatch	Puede usar actualización por lotes (métodos UpdateBatch y CancelBatch) para transmitir cambios al proveedor en grupos.

Tabla 73. Modos de cursor para el método Supports().

- Update(). Guarda los cambios efectuados en el registro actual de un Recordset.
  - Sintaxis

    ```
    oRecordset.Update aListaCampos, aValores
    ```

  - Parámetros

    Consulte el lector los parámetros empleados en el método AddNew().

- UpdateBatch(). Guarda los cambios realizados al Recordset en la base de datos a la que pertenece mediante un proceso por lotes. El Recordset debe admitir este tipo de actualizaciones.
  - Sintaxis

    ```
    oRecordset.UpdateBatch xRegistrosAfectados
    ```

  - Parámetros
    - xRegistrosAfectados. Opcional. Constante que indica los registros que serán afectados al ejecutar este método.

En la tabla 74 se muestran las constantes disponibles para los registros afectados en el método UpdateBach.

Constante	**Descripción**
adAffectCurrent	Escribe solamente los cambios pendientes en el registro actual.
adAffectGroup	Escribe los cambios pendientes en los registros que cumplen el valor de la propiedad Filter actual. Para poder usar esta opción, debe establecer la propiedad Filter a una de las constantes válidas predefinidas.
AdAffectAll	Predeterminado. Escribe los cambios pendientes en todos los registros del objeto Recordset, incluyendo aquellos que oculta el valor actual de la propiedad Filter.

Tabla 74. Constantes disponibles para los registros afectados en el método UpdateBatch().

Field

Objeto que representa a una columna de datos, en la colección Fields perteneciente a un objeto Recordset.

Propiedades

- ActualSize. Dato Long, que contiene la longitud actual del valor de un campo. No confundir con el tamaño definido para el campo. En este caso, si por ejemplo, un campo de una tabla se ha definido como de tipo texto para 50 caracteres, y en el registro actual sólo tiene una cadena con 10 caracteres, ActualSize devolverá el valor 10.

- Attributes. Constante que especifica las características de un objeto, veámoslo en la tabla 75.

Constante	**Descripción**
AdFldMayDefer	Indica que el campo se aplaza, es decir, los valores del campo no se recuperan del origen de datos con todo el registro, sino solamente cuando se tiene acceso explícito a los mismos.
adFldUpdatable	Indica que se puede escribir en el campo.
adFldUnknownUpdatable	Indica que el proveedor no puede determinar si se puede escribir en el campo.
adFldFixed	Indica que el campo contiene datos de longitud fija.
adFldIsNullable	Indica que el campo acepta valores Null.
adFldMayBeNull	Indica que se pueden leer valores Null del campo.
adFldLong	Indica que se trata de un campo binario largo. También indica que se pueden utilizar los métodos AppendChunk y GetChunk.
adFldRowID	Indica que el campo contiene un identificador de fila persistente en el que no se puede escribir y que no tiene ningún valor significativo excepto la identificación de la fila (como por ejemplo un número de registro, un identificador único, etc.).
adFldRowVersion	Indica que el campo contiene algún tipo de marca de hora o de fecha que se utiliza para efectuar actualizaciones.
adFldCacheDeferred	Indica que el proveedor almacena los valores del campo en la memoria caché y que las lecturas siguientes se efectúan en dicha memoria .

Tabla 75. Atributos para un objeto Field.

- DefinedSize. Número Long, con el tamaño definido para un campo.
- Name. Cadena con el nombre del objeto.
- NumericScale. Número de tipo Byte que indica la cantidad de posiciones decimales para los campos con valores numéricos.
- OriginalValue. Variant que contiene el valor del campo antes de que se efectuaran cambios sobre él.
- Precision. Dato Byte que indica la cantidad de dígitos usados para representar los campos con valores numéricos.
- Type. Tipo de datos para el campo. Consulte el lector la propiedad del mismo nombre en el objeto Parameter para una descripción detallada de los tipos disponibles.
- UnderlyingValue. Dato Variant, que contiene el valor del campo de la base de datos, este es el valor visible para una transacción y puede ser el resultado de una actualización efectuada por otra transacción reciente.
- Value. Mediante esta propiedad recuperamos o asignamos valor a un objeto Field.

Métodos

- AppendChunk. Permite añadir un texto de gran tamaño o datos binarios a un campo.
- GetChunk. Devuelve todo el contenido o parte de un objeto Field, que contiene datos binarios o un texto de gran tamaño. Este método es útil en situaciones en las que la memoria del sistema es limitada, y no podemos manejar un gran volumen de datos a un tiempo.
  - Sintaxis

    ```
    vntVariable = oField.GetChunk(lTamaño)
    ```

  - Parámetros
    - lTamaño. Número Long que establece la cantidad de información a recuperar.

Property

Este objeto representa una propiedad dinámica de un objeto ADO.

Las propiedades de los objetos ADO pueden ser intrínsecas y dinámicas. Las intrínsecas son aquellas que pueden ser manipulables mediante la sintaxis habitual Objeto.Propiedad. Las dinámicas vienen definidas por el proveedor de datos y son accesibles a través de la colección Properties del objeto ADO correspondiente.

Propiedades

- Attributes. Constante que especifica las características de un objeto, estos atributos aparecen en la tabla 76.

Constante	Descripción
adPropNotSupported	Indica que el proveedor no admite la propiedad.
adPropRequired	Indica que el usuario debe especificar un valor para esta propiedad antes de que se inicialice el origen de datos.
adPropOptional	Indica que el usuario no necesita especificar un valor para esta propiedad antes de que se inicialice el origen de datos.
AdPropRead	Indica que el usuario puede leer la propiedad.
AdPropWrite	Indica que el usuario puede establecer la propiedad.

Tabla 76. Atributos para un objeto Property.

- Name. Cadena con el nombre del objeto.
- Type. Tipo de datos para el objeto Property. Consulte el lector la propiedad del mismo nombre en el objeto Parameter para una descripción detallada de los tipos disponibles.
- Value. Mediante esta propiedad recuperamos o asignamos valor a un objeto propiedad.

Error

Este objeto nos informa de los errores producidos durante las operaciones de acceso a datos. Cada vez que se produce un error de este tipo, se crea un objeto Error, que se añade a la colección Errors del objeto Connection.

Propiedades

- Description. Cadena informativa con un breve comentario del error.
- NativeError. Número Long con el código de error específico del proveedor de datos con el que estamos trabajando.
- Number. Valor Long con el número exclusivo perteneciente a un objeto Error.
- Source. Cadena que contiene el nombre del objeto o aplicación que generó el error.

- SQLState. Cadena de cinco caracteres con el estado de SQL para un error, que sigue el estándar ANSI SQL.

Una vez vistos en detalle los objetos que componen el modelo ADO, pasemos a continuación a la parte práctica, en la que a través de diversos ejemplos, comprobaremos el modo de trabajo con cada uno de ellos.

El ADO Data Control

El medio más fácil y rápido para acceder a la información de una base de datos en Visual Basic, pasa por el empleo del control ADO Data, y controles enlazados a este.

El control ADO Data nos conectará a la base de datos mediante un proveedor OLE DB, y visualizará el contenido de los registros en los controles enlazados.

Para aquellos lectores que hayan programado utilizando DAO o RDO, el aspecto y modo de funcionamiento es igual que en las versiones de este control para tales modelos de datos, variando el apartado de configuración para la conexión a la base de datos y modo de recuperación de información de la misma.

A lo largo de este tema encontraremos ejemplos que utilizan una ruta de datos prefijada. En estos casos, el lector puede optar por crear dicha ruta igual que se describe en el ejemplo o utilizar una establecida por él. En este último caso también deberá, si es necesario, alterar las propiedades de los controles que componen el ejemplo, y que hacen referencia a la ruta en la que reside la base de datos utilizada, para evitar posibles errores. Por otro lado, veremos también ejemplos, que utilizando el Registro de Windows y mediante un procedimiento de lectura y escritura de información en el mismo, nos permitirán indicar la ruta de datos del programa, sin estar obligados a emplear una predeterminada.

El proyecto de ejemplo ADOCtrl, muestra como sin escribir ni una línea de código, podemos crear un mantenimiento para una tabla de una base de datos. Un mantenimiento limitado, eso sí, pero piense el lector por otra parte que todo se ha elaborado sin utilizar el editor de código, sólo empleando el editor de formularios, el cuadro de herramientas y la ventana de propiedades. Debemos tener en cuenta que la mayor potencia del control ADO Data reside en su sencillez de manejo. Gradualmente incorporaremos código para poder manejar los aspectos no accesibles directamente desde el control en modo de diseño.

En lo que respecta a las propiedades de los diferentes controles a emplear, asignaremos valor sólo a las que sea necesario, de forma que podamos comprobar cómo con un mínimo de configuración, podemos poner en marcha el programa.

En este mantenimiento abrimos la tabla Clientes del fichero Datos.MDB, visualizando su contenido en los controles de un formulario, desde el que podremos modificar y añadir nuevos registros. Los pasos a seguir para desarrollar este proyecto son los siguientes:

Crear un nuevo proyecto de tipo EXE Estándar y asignarle el nombre ADOCtrl. Seleccionar la opción del menú de VB Proyecto+Referencias, para poder incorporar en este proyecto las referencias necesarias hacia las librerías de ADO: Microsoft ActiveX Data Objects 2.0 Library y Microsoft Data Binding Collection, que nos permitirán trabajar con objetos ADO y realizar el enlace de los mismos con controles. La primera de estas referencias será obligatoria en toda aplicación que vaya a manipular datos utilizando ADO. En la figura 171 se muestran dichas referencias.

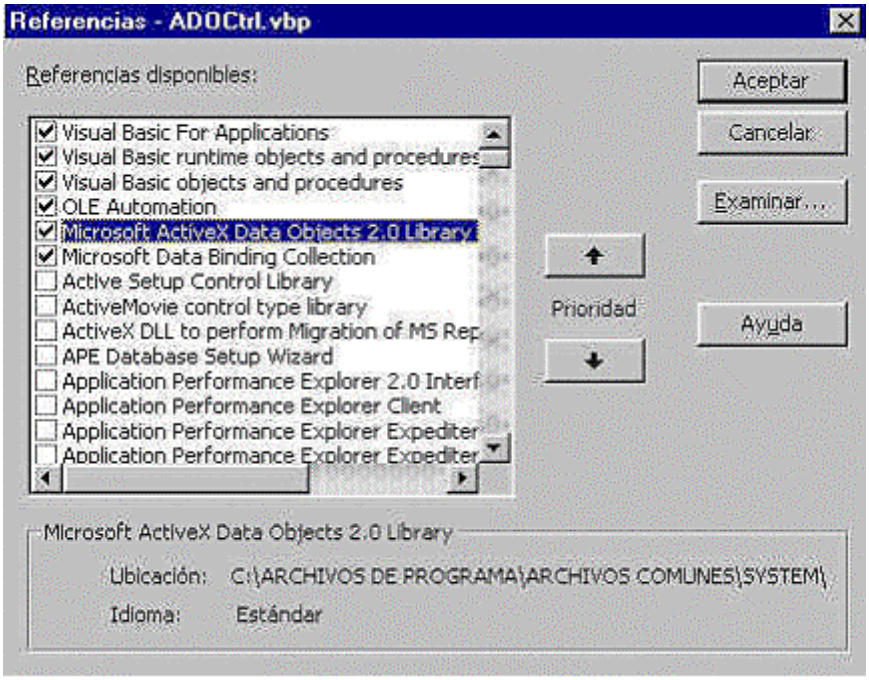


Figura 171. Establecimiento de referencias a las librerías de ADO en un proyecto.

Abrir el Cuadro de herramientas de VB, y pulsando sobre el mismo con el botón secundario del ratón, seleccionar de su menú contextual la opción Componentes, para agregar un nuevo control a este cuadro. Marcaremos el control Microsoft ADO Data Control 6.0 (OLEDB) tal y como muestra la figura 172.

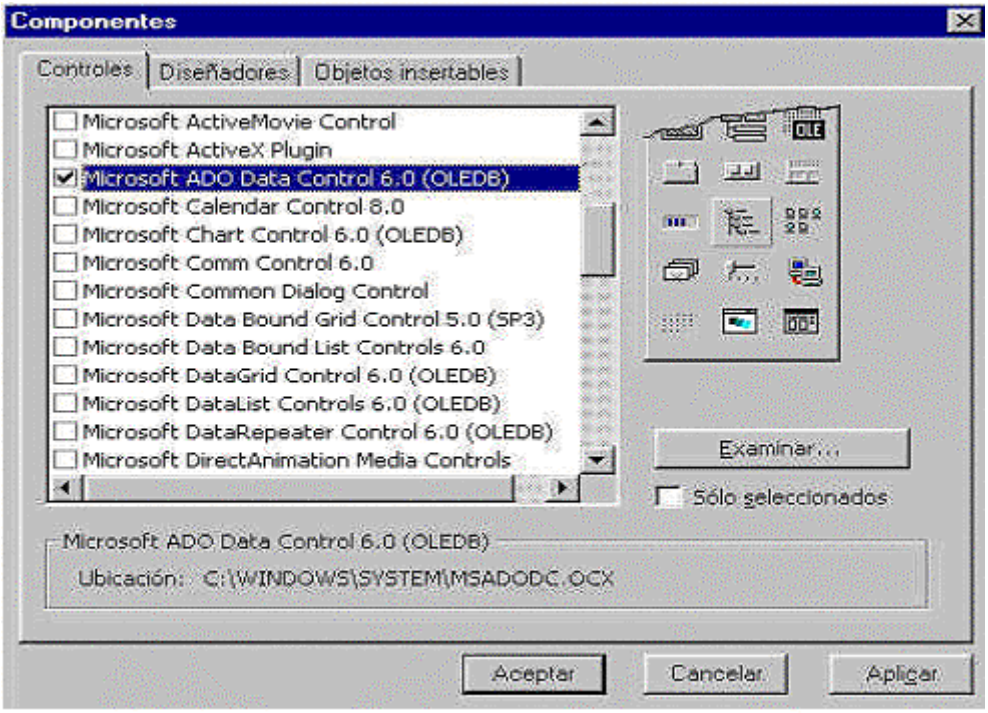


Figura 172. Selección del ADO Data Control para el Cuadro de herramientas.

Esta acción, incorporará el control en el Cuadro de herramientas, como se muestra en la figura 173.

Figura 173. ADO Data Control incluido en el Cuadro de herramientas.

Insertar un control de este tipo en el formulario que viene por defecto en la aplicación o en un nuevo formulario que agreguemos. La figura 174 muestra las operaciones de navegación por los registros efectuadas al pulsar los diferentes botones de este control.

Figura 174. Botones de navegación del control ADO Data.

Seleccionar el control insertado en el formulario, abrir su menú contextual y elegir la opción Propiedades de ADODC, que abrirá la ventana Páginas de propiedades para este control, como aparece en la figura 175.

En este punto debemos crear la conexión entre el control y la base de datos. Para ello, haremos clic sobre el OptionButton Usar cadena de conexión y pulsaremos el botón Generar para poder crear dicha cadena a través de un asistente que incorpora el control.

Se mostrará la ventana Propiedades de Data Link para crear la conexión. En su pestaña Proveedor deberemos seleccionar uno de los diferentes proveedores de OLE DB. En nuestro caso, ya que deseamos trabajar con una base de datos Access, elegiremos Microsoft Jet 3.51 OLE DB Provider, como se muestra en la figura 176, y pulsaremos el botón Siguiente >>, para pasar a la pestaña Conexión.

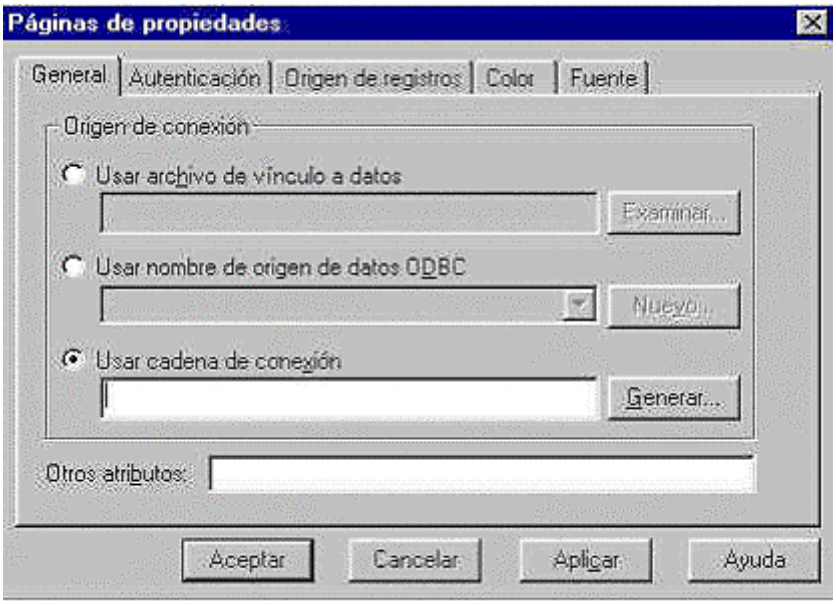


Figura 175. Páginas de propiedades del ADO Data Control.

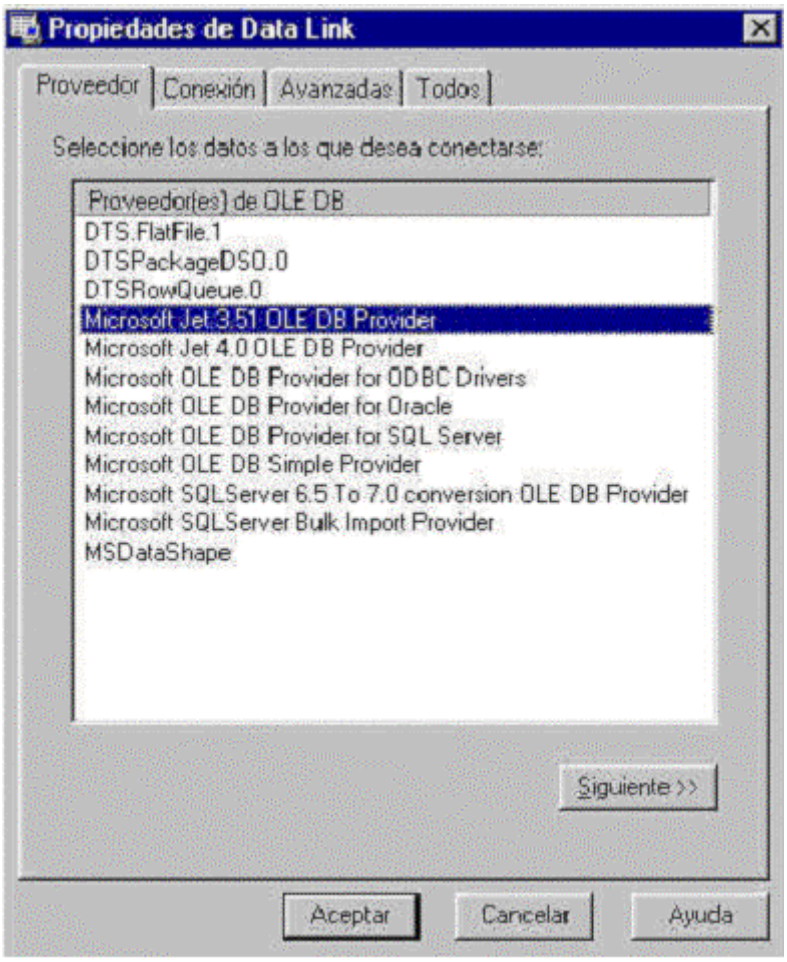


Figura 176. Selección del proveedor OLE DB para la conexión.

Después del proveedor debemos introducir la ruta y nombre del fichero de base de datos con el que vamos a trabajar, en el primer TextBox de esta pestaña, o bien buscarlo pulsando el botón que se encuentra junto a este control. Adicionalmente, puede establecer un usuario y clave de acceso a la base de datos, pero en este caso no será necesario.

La ruta empleada en la figura 177 es: C:\CursoVB6\Texto\DatosADO\ADOCtrl\Datos.mdb.

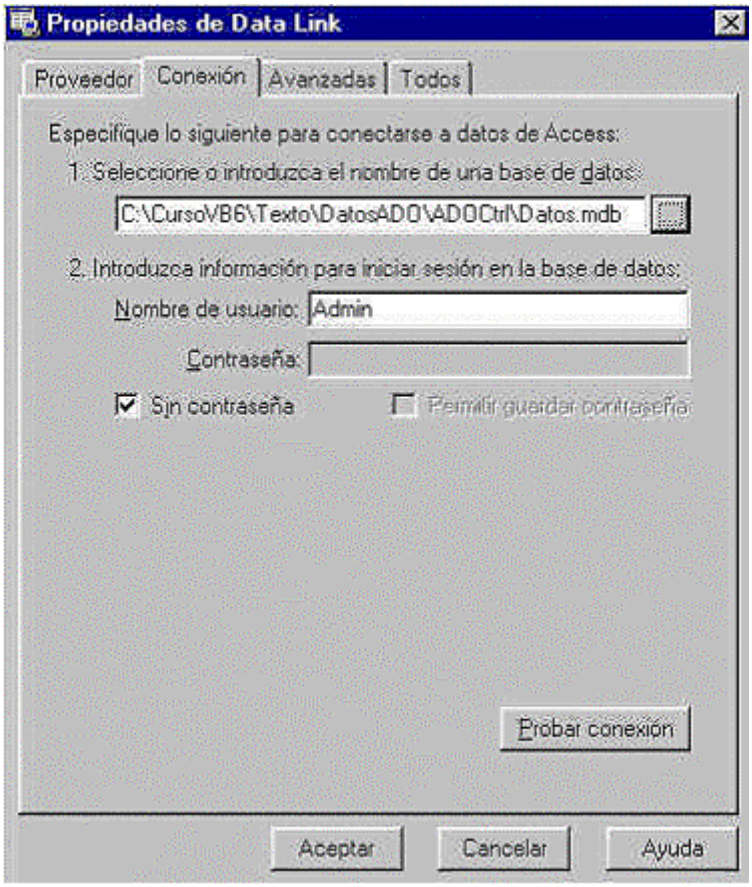


Figura 177. Ruta de la base de datos para la conexión.

Pulsando el botón Probar conexión, el lector podrá comprobar si la configuración de este paso es correcta y es posible establecer conexión con la base de datos, visualizando el resultado en una ventana de mensaje como la que aparece en la figura 178.

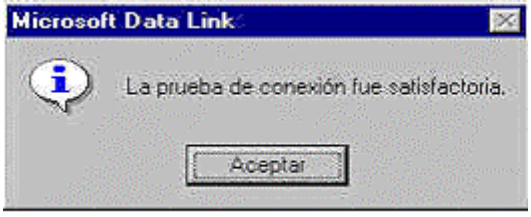


Figura 178. Mensaje de comprobación de conexión desde el control ADO Data.

Activando la pestaña Avanzadas estableceremos entre otros valores los permisos de acceso a datos. Dejaremos el valor por defecto Share Deny None (como podemos ver en la figura 179), para compartir los datos y permitir acceso a todos los usuarios.

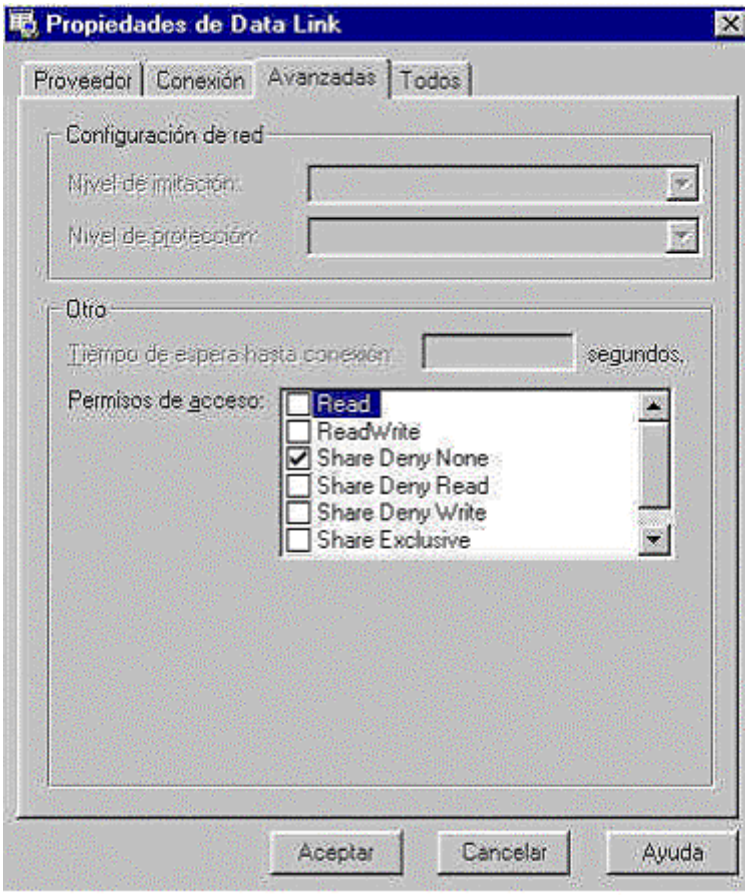


Figura 179. Permisos de acceso para la base de datos.

Finalmente, la pestaña Todos, como su nombre indica, nos permitirá visualizar al completo la totalidad de valores configurados para la conexión, modificando cualquiera de ellos al pulsar el botón Editar valor, como vemos en la figura 180.

Pulsando Aceptar, finalizará la creación de la cadena de conexión que será depositada en la ventana de propiedades del control ADO Data. Esta cadena presentará variaciones dependiendo del proveedor de datos utilizado, para este caso será algo parecido a la siguiente:

```
Provider=Microsoft.Jet.OLEDB.3.51;Persist Security Info=False;Data
Source=C:\CursoVB6\Texto\DatosADO\ADOCtrl\Datos.mdb
```

Continuando con las páginas de propiedades de este control, la pestaña Autenticación contiene la información de claves de acceso a la base de datos, que aquí estará vacía debido a que no vamos a solicitar dichas claves. Pasaremos pues a la pestaña Origen de registros (figura 181), en la cual estableceremos la forma y el contenido que vamos a solicitar de la base de datos.

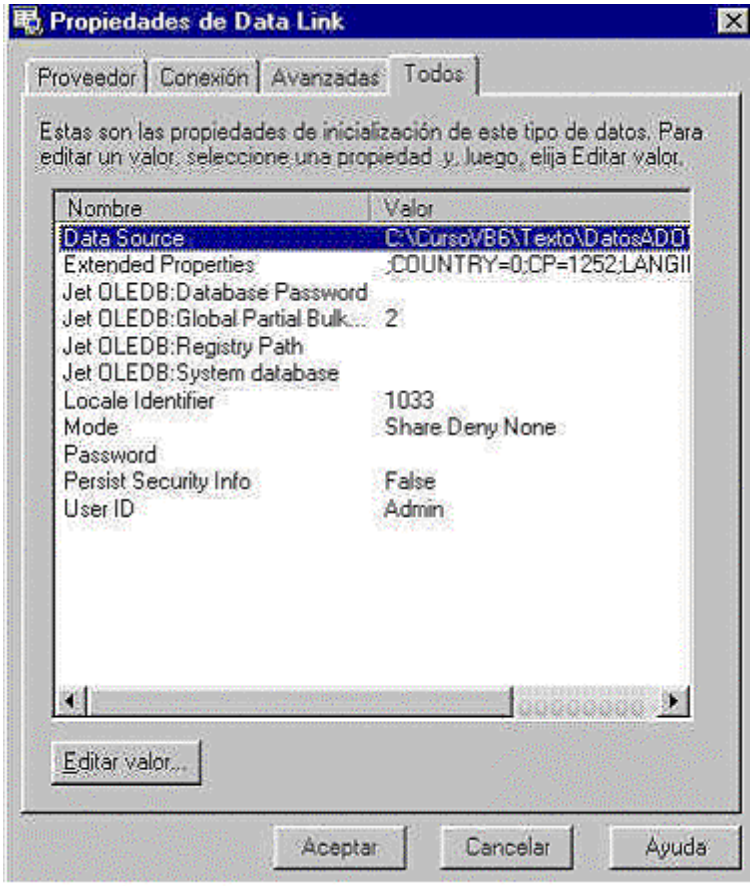


Figura 180. Valores al completo para la conexión con la base de datos.

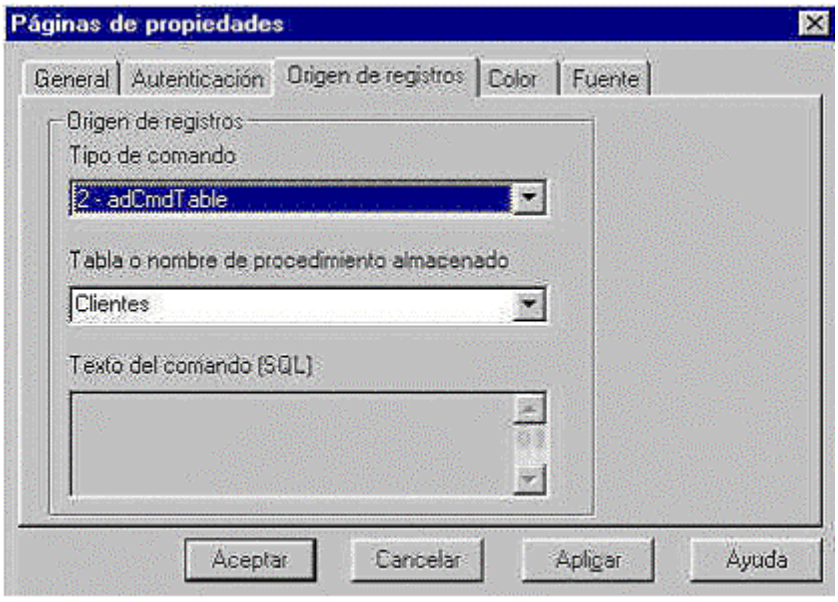


Figura 181. Establecer origen de registros para el ADO Data.

En primer lugar, debemos seleccionar de la lista desplegable Tipo de comando, el modo en que vamos a recuperar los datos. Seleccionaremos adCmdTable, ya que vamos a acceder a una tabla. En la siguiente lista, elegiremos la tabla o procedimiento almacenado a consultar: Clientes. Si necesitáramos ejecutar una sentencia SQL contra una o varias tablas de la base, emplearíamos el TextBox Texto del comando (SQL), en esta ocasión está deshabilitado ya que el tipo de comando que vamos a ejecutar, se basa en una tabla física y no una consulta de selección. Pulsando Aceptar, finalizaremos la configuración de la conexión a la tabla de datos.

Un control ADO Data al llegar al último de los registros, permanece por defecto en él aunque sigamos pulsando su botón de avance. Para poder agregar nuevos registros a la tabla, modificaremos la propiedad EOFAction del control en la ventana de propiedades, asignándole el valor adDoAddNew, como muestra la figura 182, que añadirá un nuevo registro a la tabla cuando al estar situados en el último registro, pulsemos de nuevo el botón de avance. Con esto finalizamos la configuración del control.

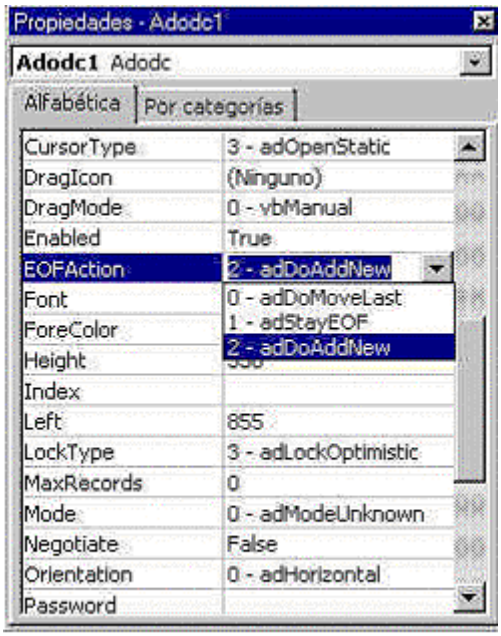


Figura 182. Valores disponibles para la propiedad EOFAction del ADO Data.

Ahora el control está listo para recuperar información de la tabla, pero por sí mismo no puede mostrar datos al usuario, por este motivo se hace necesario incluir controles que puedan enlazarse al ADO Data y mostrar la información que este contiene.

Incluiremos pues, desde el cuadro de herramientas, un control TextBox y un Label por cada campo de la tabla Clientes. En la propiedad Caption de los Label pondremos el nombre de cada campo y en los TextBox modificaremos las siguientes propiedades:

- Name. Un nombre identificativo del campo que va a mostrar el TextBox.
- DataSource. Esta propiedad abre una lista con todas las fuentes de datos definidas en el proyecto, en nuestro caso sólo disponemos de uno, que es el que seleccionamos.
- DataField. Esta propiedad abre una lista con los campos disponibles en la fuente de datos establecida en la propiedad DataSource del control enlazado. Lo que haremos aquí será seleccionar un campo diferente para cada TextBox.

- MaxLength. Aquí definimos el máximo número de caracteres que admite el TextBox, si no usáramos esta propiedad e intentáramos por ejemplo poner más de dos dígitos en el campo del código de cliente, VB nos avisaría que esto no es posible y no permitiría agregar el nuevo registro hasta que los datos fueran correctos.

Una vez hecho esto, ya hemos finalizado el trabajo, iniciamos la aplicación que tendrá un aspecto como el de la figura 183.

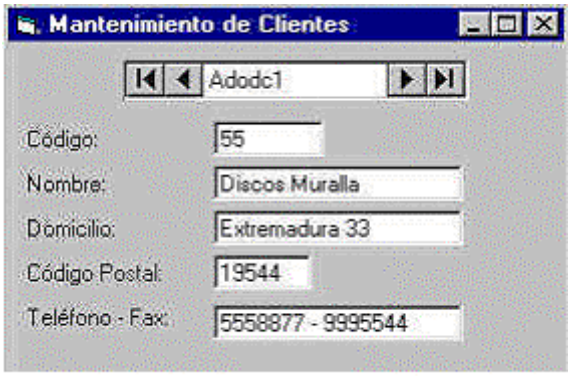


Figura 183. Mantenimiento de datos usando el control ADO Data.

Ahora podemos utilizar el control de datos para navegar por los registros de la tabla, modificar el registro que aparece en el formulario o añadir nuevos registros desde el final de la tabla. Los controles TextBox serán rellenados automáticamente con la información del registro actual, y todo esto se ha realizado utilizando básicamente el ratón. Por supuesto que una aplicación real, siempre requerirá un tipo de validación a la hora de editar datos, que nos obligue a utilizar algo más que el simple control, pero puede darse el caso de que necesitemos desarrollar un programa en el que sea necesaria la introducción inmediata de datos, mientras se desarrollan los procesos definitivos de entrada de datos. Podemos utilizar formularios de este estilo para que los usuarios puedan comenzar a introducir información, siempre que el diseño del programa y base de datos lo permitan.

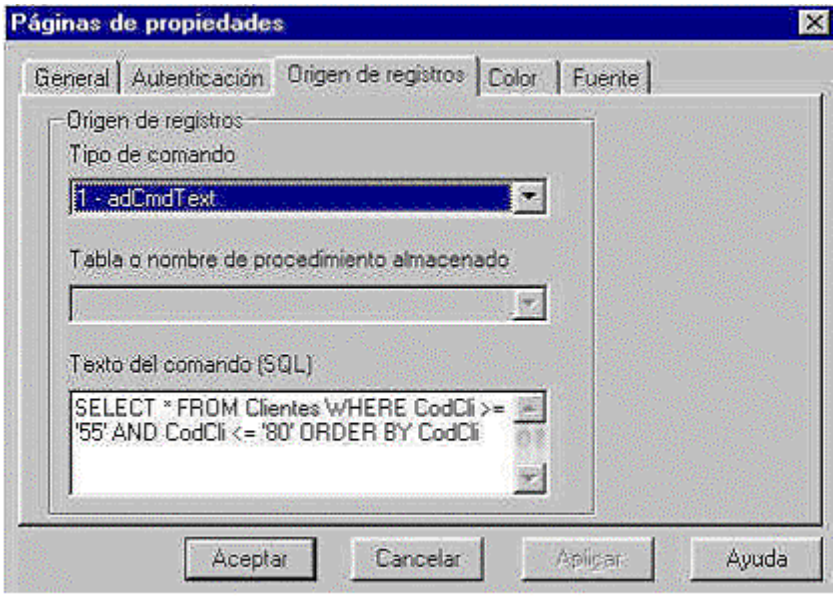


Figura 184. Seleccionar registros en el ADO Data mediante una instrucción SQL.

Para finalizar con este apartado, supongamos que en lugar de acceder a la totalidad de la tabla, sólo necesitamos manejar una selección de sus registros. Esto lo conseguiremos, abriendo la ventana de páginas de propiedades del control ADO Data, y en la pestaña Origen de registros, como tipo de comando seleccionar adCmdText, con lo que se deshabilitará esta vez la lista dedicada a la tabla o procedimiento almacenado y se habilitará el TextBox para introducir una instrucción SQL, como aparece en la figura 184, en el que podremos escribir un comando como el mostrado en la siguiente figura, que además de seleccionar un rango de registros, los ordenará en función a un campo.

Efectúe el lector estos cambios en este proyecto, para comprobar como el control sólo nos ofrece el subconjunto de registros indicados en la sentencia SQL.

Validaciones a nivel de motor

Pongámonos en el caso de que en el anterior programa no se debe dar un valor inferior a "10" para el campo código de cliente, si queremos conseguirlo sin tener que escribir una rutina de validación en el propio programa, podemos emplear una regla de validación a nivel de motor, que consiste en que el propio motor de datos, en este caso Access, se encarga de comprobar los valores a introducir en un registro, si no se cumple el criterio de validación establecido, no se grabará el registro.

Abrimos en Access la base de datos Datos.MDB y la tabla Clientes en modo diseño, nos situamos en el campo CodCli, e introducimos la validación tal y como se muestra en la figura 185.

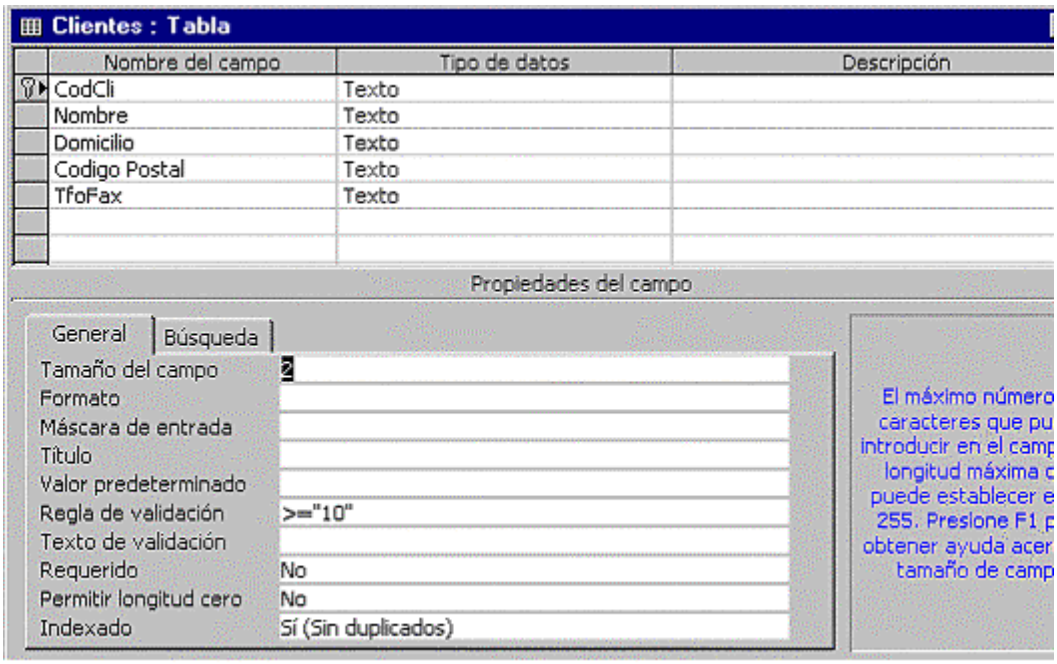


Figura 185. Regla de validación establecida en Access para un campo de una tabla.

Grabamos los cambios realizados en la tabla, cerramos Access y volvemos a ejecutar el ejemplo. Ahora si intentamos crear un nuevo registro, dando al campo de código el valor "06" por ejemplo, nos encontraremos con el aviso que muestra la figura 186.

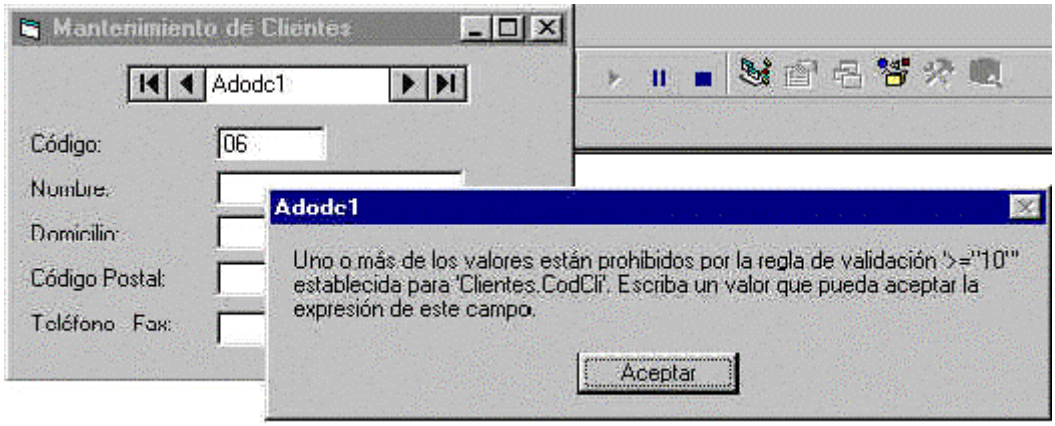


Figura 186. Aviso del motor de datos, por haber transgredido una regla de validación.

No podremos grabar el registro hasta que no cumplamos la regla de validación.

Es posible incluir reglas de validación para todos los campos de la tabla, y aunque la validación mediante una rutina de código siempre será más potente y flexible, este medio de controlar la grabación de los datos siempre puede sernos útil en validaciones sencillas.

El control DataGrid

El mantenimiento del ejemplo anterior tiene un defecto muy importante, y es que el usuario puede necesitar una visión global de los registros de la tabla y no tener que verlos uno a uno. Para solventar ese problema disponemos del control DataGrid o cuadrícula, que muestra los datos en forma de tabla, con lo cual podemos tener de un sólo vistazo, una mayor cantidad de información y navegar de forma más rápida por las filas.

La aplicación de ejemplo CtlDataGrid, proporciona un mantenimiento similar al visto en el apartado dedicado al ADO Data, pero utilizando un DataGrid en lugar de los TextBox para editar la información de los campos de un registro.

La ruta para la base de datos utilizada en este ejemplo es:

C:\CursoVB6\Texto\DatosADO\CtlDataGrid, por lo que el lector deberá de crear una igual o modificar este aspecto en el control ADO Data del ejemplo, en la parte de las propiedades que genera la cadena de conexión con la base de datos.

En el caso de que este control no se encuentre en el cuadro de herramientas, deberemos incorporarlo mediante la ventana Componentes, al igual que hacíamos en el apartado anterior con el control ADO Data, seleccionando en esta ocasión el control denominado *Microsoft DataGrid Control* 6.0 (OLEDB), como podemos ver en la figura 187.

Después de agregar este control, el cuadro de herramientas presentará un aspecto parecido al de la figura 188.

Los pasos para crear el formulario del programa y el control ADO Data, son iguales que en el anterior ejemplo, por lo que no los repetiremos aquí. Una vez llegados a este punto, insertaremos un control DataGrid de la misma forma que lo hacemos con otro control.

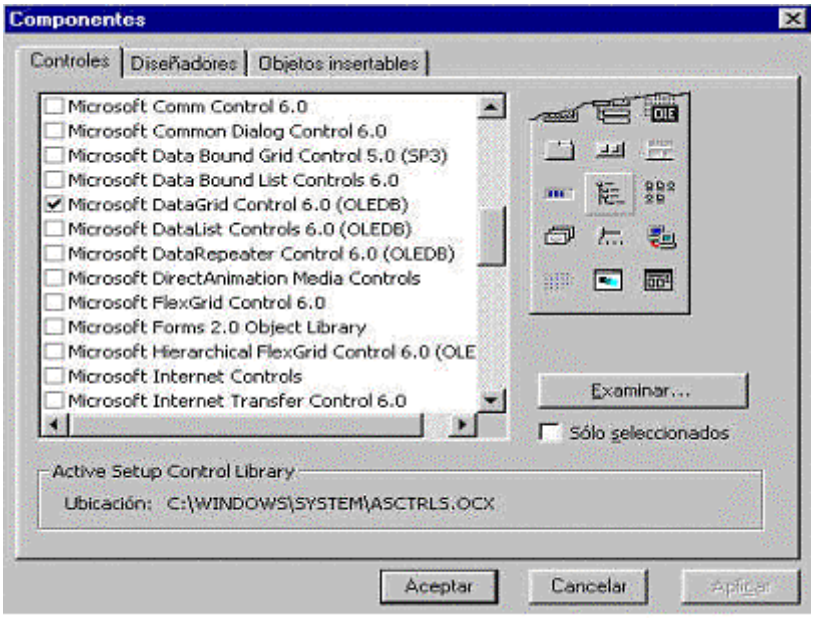


Figura 187. Selección del control DataGrid en la ventana Componentes.

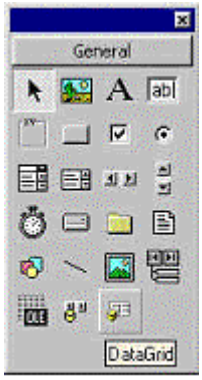


Figura 188. Control DataGrid incluido en el cuadro de herramientas.

Para obtener resultados inmediatos, sólo tenemos que asignar a la propiedad DataSource del DataGrid, el nombre del control ADO Data y ejecutar la aplicación, de esta manera se visualizarán los registros en el control DataGrid, como se puede ver en la figura 189.

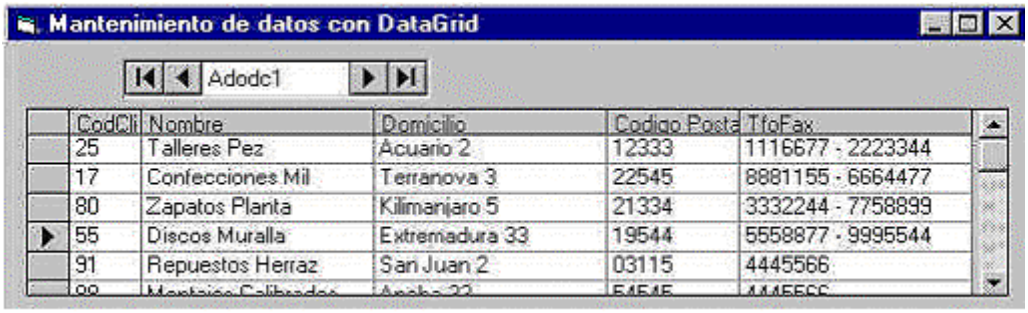


Figura 189. Aplicación de mantenimiento de datos con DataGrid.

Sin embargo, en muy pocas ocasiones nos bastará con mostrar este control con sus valores por defecto. Para alterar el aspecto de un DataGrid, podemos utilizar las ventanas de propiedades y páginas de propiedades de este control, o bien desde el código de la aplicación, manipular el aspecto del control antes de presentarlo al usuario. En este apartado lo haremos en modo de diseño.

En primer lugar, seleccionaremos el control DataGrid del formulario y abriremos su menú contextual, eligiendo la opción *Recuperar campos*; de esta forma, la información de los campos de la tabla que contiene el control ADO Data pasará a la cuadrícula y podrán ser modificados.

Nuevamente abriremos el menú contextual del DataGrid, seleccionando la opción *Propiedades*, que mostrará la ventana de páginas de propiedades con varias pestañas para configurar este control. Entre el numeroso grupo de propiedades, vamos a comentar algunas de las utilizadas en este ejemplo, agrupadas por la página que las contiene.

- General.
  - Caption. Cadena para establecer un título en el DataGrid.
  - RowDividerStyle. Constante con las diferentes formas de división para las filas del control. En este caso no se visualizarán líneas divisorias entre las filas al emplear la constante dbgNoDividers.
  - AllowAddNew. Valor lógico para permitir al usuario añadir nuevos registros directamente en el DataGrid.
  - AllowUpdate. Valor lógico para permitir al usuario modificar los registros existentes en el control.
  - ColumnHeaders. Valor lógico para mostrar u ocultar los títulos de las columnas.
- Teclado.
  - AllowArrows. Valor lógico para poder efectuar la navegación por las celdas del DataGrid utilizando las teclas de movimiento del cursor.
  - TabAction. Constante que establece el comportamiento de la tecla TAB cuando el DataGrid tiene el foco, para poder mover entre las celdas del control o pasar al resto de controles del formulario.
- Columnas.
  - Column. Columna del control a configurar.
  - Caption. Título de la columna.
  - DataField. Campo del control ADO Data que proporciona los datos y que se va a visualizar en el DataGrid.
- Diseño.
  - AllowSizing. Permite o impide variar el ancho de una columna.
  - Visible. Muestra u oculta una columna.

- Button. Muestra un botón en la parte derecha de la celda, para columnas en las que cada celda necesite seleccionar entre un conjunto de valores.
- Width. Establece el ancho de la columna.

- Divisiones.
  - AllowRowSizing. Permite modificar la altura de las filas.
  - RecordSelectors. Muestra la columna situada en primer lugar a la izquierda del control con los marcadores de fila.
  - MarqueeStyle. Permite establecer el aspecto de las filas normales y las que tienen el foco de edición.
- Formato.
  - Tipo de formato. Ofrece una lista de los formatos disponibles para aplicar a las diferentes columnas del control.

Después de algunos cambios en estas propiedades, ejecutaremos de nuevo el ejemplo, que nos mostrará de nuevo el DataGrid con un aspecto diferente de su configuración original (figura 190).

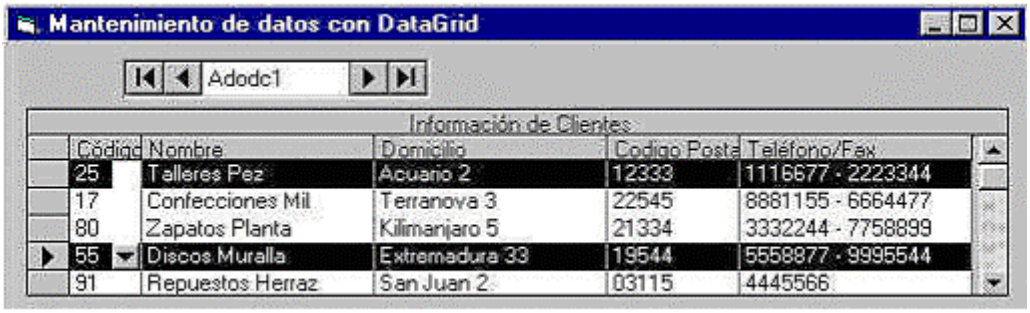


Figura 190. Formulario con DataGrid modificado.

Reflexiones sobre los controles enlazados a datos

Como acabamos de ver, el control ADO Data en combinación con el DataGrid o controles enlazados a datos, proporcionan un medio rápido y fácil de acceder y editar las filas de una tabla, pero toda esa sencillez se convierte en un problema cuando necesitamos que el programa haga ese *algo más* que siempre se le pide.

Entre el conjunto de operaciones que no son posibles de realizar directamente por el control ADO Data, y para las que tendremos que escribir código en el programa están el borrado de registros, búsquedas de registros, introducir nuevos registros desde cualquier punto de la tabla sin tener que ir hasta el último, y de ahí pasar a un registro vacío, etc.

En lo que respecta al DataGrid, puede que durante el transcurso del programa, necesitemos ocultar y mostrar una determinada columna, cambiar varias veces su título, habilitar y deshabilitar el control para permitir al usuario navegar por las filas, etc.

Otro problema añadido lo constituye el hecho de que en muchas aplicaciones no conocemos durante el desarrollo de las mismas la ubicación de la base de datos, ya que es durante la instalación del programa cuando el usuario proporciona dicha ubicación. En este tipo de casos, no podemos indicarle al control donde están los datos.

Cuando nos encontramos en esta situación, si utilizamos un DataGrid en un formulario, la mejor manera de configurar sus columnas es mediante código, de forma que una vez que sepamos la ruta de los datos y hayamos asignado un conjunto de registros al control ADO Data conectado al DataGrid, escribamos un procedimiento de configuración para el mismo con los títulos de columna, ancho, formato de salida de los datos, etc.

En definitiva, son muchas las situaciones en que el simple uso de los controles de datos no nos ayudará a resolver el problema, por lo cual podemos afirmar, que si realmente queremos potencia y flexibilidad en nuestros mantenimientos de datos, tendremos ineludiblemente que combinar controles de datos y la creación de un conjunto más o menos numeroso de procedimientos que hagan uso de los objetos de datos.

Controles enlazados y código, un punto intermedio

Antes de abordar el desarrollo de aplicaciones utilizando fundamentalmente código, vamos a usar una técnica situada a medio camino entre el empleo intensivo de controles o código. Se trata de combinar tanto controles de datos como código para manejar esos datos, de manera que sólo escribamos código allá donde sea absolutamente necesario, delegando en los controles el resto del trabajo. Aprovecharemos al mismo tiempo para presentar un nuevo control de datos: el control DataCombo.

La aplicación de ejemplo

El proyecto CtlEnlazCodigo, que emplearemos para realizar las pruebas, deberá ser incluido en la ruta C:\CursoVB6\Texto\DatosADO\CtlEnlazCodigo, esta es la ruta hacia la que estableceremos los controles enlazados.

Esta aplicación hace uso de la base de datos Mercanci.MDB, que contiene una tabla de productos y otra de envíos de esos productos a diferentes destinos. El objetivo de la aplicación es crear un formulario para introducir la información de cada envío, es decir, cada grupo de datos introducido correspondería al envío de un producto, lo que equivaldría a un registro en la tabla Envios.

Se utilizará un control ADO Data para la tabla Envios, enlazado a los controles TextBox que reflejan el registro actual. Para poder seleccionar los registros de la tabla Productos se usará otro control Data, pero este se enlazará a un control DataCombo, cuyas interesantes características veremos más adelante.

El formulario de datos

Una vez incluidos todos los controles en el formulario frmControlCodigo, este tendrá el aspecto que muestra la figura 191.

Para navegar por los registros de la tabla Envios, utilizaremos el control ADO Data adcEnvios, visualizándose el contenido del registro en los TextBox del formulario. Si queremos insertar un nuevo registro pulsaremos el botón cmdAgregar y escribiremos los valores en los campos correspondientes. Para seleccionar los productos utilizaremos la lista de productos incluida en el DataCombo.

Figura 191. Formulario frmControlCodigo.

El control DataCombo

Un control DataCombo es una combinación de cuadro de texto y lista de valores. En función de su estilo es posible seleccionar un valor de la lista desplegable o escribirlo en la zona de texto. Se diferencia del control ComboBox normal, en que es posible enlazarlo a un control ADO Data, de manera que la lista de valores se rellene automáticamente con los valores de uno de los campos del ADO Data. Pero cuando este control muestra todo su potencial es al enlazarlo a un segundo control ADO Data, de forma que se pueda traspasar el valor de un campo del primer ADO Data al segundo.

Situémonos en el ejemplo que ahora nos ocupa. Debemos crear registros para la tabla Envios y uno de los campos a completar es el código del producto que se va a enviar. ¿Pero qué es más fácil, recordar el código de cada uno de los productos o el nombre?, lógicamente el nombre nos será siempre más familiar, y puestos a pedir facilidades, si disponemos de una lista con los nombres de los productos a elegir, nos evitará tener que conocer todas las descripciones de los mismos.

Una vez configurado el DataCombo cboProductos la mecánica de este control es la siguiente: cada vez que seleccionemos el nombre de un producto, el código del mismo será pasado al TextBox del formulario destinado a ese valor. El usuario trabajará mejor empleando los nombres de los productos, mientras que el DataCombo se encargará internamente de usar los códigos.

Para conseguir que el control cboProductos se comporte de esta manera, las propiedades a configurar son las siguientes. Se indica entre paréntesis el valor utilizado para cada propiedad en este ejemplo.

- RowSource. Nombre del control ADO Data que se utiliza para llenar la lista (adcProductos).
- ListField. Nombre de un campo del Recordset contenido en RowSource con cuya información se llenará la lista del control (Descrip).
- DataSource. Nombre del control ADO Data al que se traspasa un valor cada vez que se selecciona un elemento de la lista (adcEnvios).
- DataField. Nombre de un campo del Recordset contenido en DataSource al que se traspasa el valor seleccionado en la lista (CodProd).
- BoundColumn. Nombre de un campo del Recordset incluido en RowSource cuyo valor se traspasa a DataField cuando se hace una selección en la lista (CodProd).

Abra el lector la ventana de propiedades del control cboProductos y comprobará que la configuración de las propiedades antes descritas para este control usan el campo Descrip para mostrar los nombres de los productos en la lista, pero el valor a traspasar entre los controles ADO Data es el del campo CodProd.

El código a incluir

Como hemos dicho, en esta aplicación se va a incluir sólo el código imprescindible allá donde los controles de datos no puedan llegar. Esto es una verdad sólo a medias, ya que en el caso del botón cmdAgregar se puede configurar el control Data para que inserte un nuevo registro al llegar al final del fichero, pero mediante el código fuente 182, en este CommandButton agregaremos el nuevo registro en cualquier momento.

```
Private Sub cmdAgregar_Click()
Me.adcEnvios.Recordset.AddNew
End Sub
```

Código fuente 182

Por su parte, el botón cmdEliminar ejecutará el código fuente 183 para borrar un registro y reposicionar el recordset al principio.

```
Private Sub cmdEliminar_Click()
Me.adcEnvios.Recordset.Delete
Me.adcEnvios.Recordset.MoveFirst
End Sub
```

Código fuente 183

Y como punto final tenemos el DataCombo cboProductos. En principio no es necesario incluir código, tal y como están configuradas sus propiedades, el traspaso de valores entre los recordsets de los controles ADO Data funciona perfectamente, pero existe un pequeño inconveniente: cada vez que se selecciona un producto de la lista, el TextBox que contiene su valor no refleja el código de producto seleccionado. Para esta situación disponemos del evento Click() en el DataCombo, que nos permite saber cuando un usuario ha pulsado con el ratón en este control y en que área del mismo lo ha hecho. De esta forma, cuando el usuario haga clic en un elemento de la lista, traspasaremos el código del producto al TextBox utilizando la propiedad BoundText del DataCombo, que contiene el valor del campo a traspasar.

```
Private Sub cboProductos_Click(Area As Integer)
Select Case Area
Case dbcAreaButton
   ' no traspasa valores al campo
Case dbcAreaEdit
   ' no traspasa valores al campo
Case dbcAreaList
   Me.txtCodProd.Text = Me.cboProductos.BoundText
End Select
End Sub
```

Código fuente 184

La aplicación en funcionamiento

Terminado el trabajo de creación del programa, podemos ponerlo en funcionamiento para comprobar los resultados.

El lector observará que según navega por los registros del control adcEnvios, el nombre del producto se va actualizando automáticamente en la parte de texto del DataCombo. De igual forma al crear o editar un registro, como hemos dicho anteriormente, cada vez que es seleccionado un nuevo valor en la lista del DataCombo, el código asociado al producto se sitúa en el TextBox establecido para dicho valor.

Como conclusión, podemos apuntar que el empleo de controles de datos y código, constituye una forma mejorada de manejar los datos que utilizando exclusivamente controles.

En cuando al control DataCombo, lo explicado en este apartado para dicho control es aplicable igualmente para el control DataList, con las particularidades propias de cada control, naturalmente.

Creación de objetos ADO

En este apartado, vamos a centrarnos exclusivamente en el uso de código para manipular objetos ADO. Empleando los controles de datos que hemos visto hasta el momento, también estamos utilizando objetos ADO, pero toda la fase de instanciación y manipulación, queda enmascarada por el propio control. Lo que a primera vista puede parecer una ventaja, ya que el control se encarga de la parte difícil al tratar directamente con ADO, cuando debamos de manipular ciertos aspectos resultará un inconveniente, ya que el control no nos dejará profundizar. En ese tipo de situaciones es cuando deberemos dejar de lado el control de datos y abordar el desarrollo utilizando fundamentalmente código.

ADO dispone de un modelo de datos mucho más simple y menos estricto de manejar que otras jerarquías ya conocidas como DAO o RDO. Si queremos, por ejemplo, crear un objeto Recordset de ADO, no es necesario instanciar los objetos de nivel superior para llegar a este, ya que es posible crear el Recordset directamente, mientras que los objetos superiores se crearán y usarán internamente sin que tengamos que preocuparnos de ellos.

En este apartado veremos algunos ejemplos de creación y manipulación de objetos ADO, contenidos en el proyecto ADOCodigo, que emplea la ruta C:\CursoVB6\Texto\DatosADO\ADOCodigo, en la que reside la base de datos Musica.MDB, utilizada por este proyecto, de modo que el lector pueda comprobar la flexibilidad ofrecida por este tipo de objetos.

Esta aplicación de ejemplo, utiliza el formulario MDI mdiInicio, desde el que podremos acceder a los diferentes ejemplos mediante las opciones del menú que contiene.

Utilizar los objetos principales del modelo

En este punto, vamos a realizar una primera aproximación a ADO al estilo tradicional, es decir, creando desde el objeto de mayor nivel hasta el Recordset, que es el que necesitamos para manejar registros. Si el lector ya ha programado con otras jerarquías de datos, por ejemplo DAO, este modo de trabajo le resultará familiar.

Seleccionaremos para ello la opción Archivo+Modelo completo, que contiene el código fuente 185, comentadas de forma que el lector pueda seguir cada paso de los realizados en el fuente.

```
Private Sub mnuArModeloCom_Click()

Dim lacnConexion As ADODB.Connection
Dim lacmComando As ADODB.Command
Dim larsRecordset As ADODB.Recordset
' instanciar conexion
Set lacnConexion = New ADODB.Connection
' establecer cadena de conexión
lacnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
' abrir conexión
lacnConexion.Open
' instanciar comando
Set lacmComando = New ADODB.Command
' establecer el objeto conexión en el comando,
' tipo de comando y cadena de consulta de registros
Set lacmComando.ActiveConnection = lacnConexion
lacmComando.CommandType = adCmdText
lacmComando.CommandText = "SELECT * FROM Grabaciones " & _
   "WHERE Titulo >= 'U' ORDER BY Titulo"
' instanciar recordset
Set larsRecordset = New ADODB.Recordset
' ejecutar el comando y asignar su resultado al recordset
Set larsRecordset = lacmComando.Execute
' recorrer el recordset y mostrar al usuario
' el valor de un campo de cada registro
Do While Not larsRecordset.EOF
   MsgBox "Titulo: " & larsRecordset("Titulo"), vbOKOnly, _
      "Valor del campo"
   larsRecordset.MoveNext
Loop
' cerrar objetos
larsRecordset.Close
lacnConexion.Close
' liberar recursos
Set larsRecordset = Nothing
Set lacmComando = Nothing
Set lacnConexion = Nothing

End Sub
```

Código fuente 185

Modificar la información a través del objeto Connection

Aparte de establecer una conexión con una fuente de datos, el objeto Connection mediante su método Execute() nos permite enviar instrucciones SQL para realizar operaciones de mantenimiento (agregar, modificar, eliminar, etc.) sobre los registros de la base de datos.

La opción de menú Archivo+Conexión con inserción del proyecto, insertará un nuevo registro en una de las tablas utilizando un objeto Connection, ahorrándonos el uso de un objeto Command o Recordset, que aunque son más adecuados para ello, al tratarse de una inserción simple, no son necesarios.

En las ocasiones que no debamos emplear procedimientos almacenados, ni realizar navegación por los registros de una tabla o consulta, el uso de Connection simplificará en gran medida nuestro trabajo.

El código fuente 186, contiene el código de esta opción.

```
Private Sub mnuArConexInserc_Click()
Dim lacnConexion As ADODB.Connection
Dim lcEjecutar As String
Dim llRegistros As Long
' crear objeto conexión, configurar y abrir
Set lacnConexion = New ADODB.Connection
lacnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnConexion.Open
' agregar un registro a una de las tablas de la base de datos
' desde el objeto conexión
lcEjecutar = "INSERT INTO Autores (CodAutor, Autor) " & _
   "VALUES ('056', 'THE SMASHING PUMPKINS')"
lacnConexion.Execute lcEjecutar, llRegistros, adCmdText
' cerrar conexión y eliminar de memoria
lacnConexion.Close
Set lacnConexion = Nothing
End Sub
```

Código fuente 186

Consultas de registros usando Connection

Al igual que en el caso anterior, si necesitamos realizar una consulta sencilla contra una base de datos, sin tener que emplear paso de parámetros ni navegación por los registros de la consulta, podemos utilizar el objeto Connection tal como muestra la opción de menú Archivo+Conexión.

```
Private Sub mnuArConexConsul_Click()
Dim lacnConexion As ADODB.Connection
Dim larsRecordset As ADODB.Recordset
Dim lfrmVerRecordset As frmVerRecordset
' crear objeto conexión, configurar y abrir
Set lacnConexion = New ADODB.Connection
lacnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnConexion.Open
' obtener un recordset a partir del objeto conexión
Set larsRecordset = lacnConexion.Execute("SELECT * FROM Autores")
' abrir el formulario en el que se
' visualizarán los datos del Recordset
Set lfrmVerRecordset = New frmVerRecordset
Load lfrmVerRecordset
lfrmVerRecordset.Caption = "Hacer consulta desde objeto Connection"
lfrmVerRecordset.Show
' recorrer el Recordset y rellenar
' el ListBox del formulario
Do While Not larsRecordset.EOF
   lfrmVerRecordset.lstDatos.AddItem larsRecordset("Autor")
   larsRecordset.MoveNext
Loop
' cerrar objetos y eliminarlos de memoria
larsRecordset.Close
lacnConexion.Close
Set larsRecordset = Nothing
Set lacnConexion = Nothing
End Sub
```

Código fuente 187

El recordset obtenido de esta manera es de tipo ForwardOnly, permitiéndonos únicamente avanzar registros, útil para consultas sencillas o listados. Si necesitamos un Recordset con un cursor más potente, debemos crear específicamente el Recordset. En este ejemplo lo usamos para rellenar un control ListBox de una ventana con los valores de un campo, como podemos ver en la figura 192.

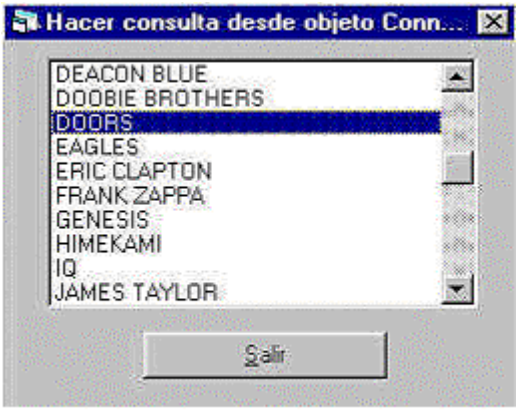


Figura 192. Valores del campo de un Recordset obtenido desde un objeto Connection.

Reserva de conexiones

Los objetos Connection tienen una característica denominada Connection Polling, que podemos traducir como reserva de conexiones, y que consiste en que una vez finalizado el trabajo con un objeto de este tipo, al destruirlo en el código, lo que realmente hace este objeto es pasar a un área especial o reserva de conexiones en la que permanecerá allí por un tiempo determinado.

Si en el transcurso de ese tiempo, la aplicación necesita una conexión que tenga las propiedades de uno de los objetos de esta reserva, se toma uno de estos objetos, evitando el consumo de tiempo que requiere el establecimiento de una nueva conexión.

Pasado un determinado tiempo, si el objeto de la reserva no ha sido utilizado, se destruirá definitivamente.

Procedimientos almacenados desde objetos Command

Cuando además de realizar consultas y alterar la información de la base de datos, necesitemos ejecutar un procedimiento almacenado con paso de parámetros, utilizaremos el objeto Command. Una vez ejecutado este objeto, si está basado en una consulta, nos devolverá un Recordset.

El código fuente 188, corresponde a la opción Archivo+Comando con procedimiento almacenado del programa de ejemplo, y en él podemos ver como ejecutar un procedimiento almacenado, creando un objeto Parameter y enviando un valor en dicho objeto al procedimiento para seleccionar un determinado registro.

Esta rutina hace uso de SQL Server y la base de datos Northwind incluida entre las diversas bases de datos de ejemplo de este gestor, por lo que el lector deberá disponer de SQL Server para poder ejecutar el código fuente 188.

```
Private Sub mnuArComando_Click()
Dim lacnConexion As ADODB.Connection
Dim lacmComando As ADODB.Command
Dim laprParametro As ADODB.Parameter
Dim larsRecordset As ADODB.Recordset
' crear objeto conexión, configurar y abrir
Set lacnConexion = New ADODB.Connection
lacnConexion.ConnectionString = "Provider=SQLOLEDB.1;" & _
   "Persist Security Info=False;User ID=sa;" & _
   "Initial Catalog=Northwind;Data Source=LMBLANCO"
lacnConexion.Open
' crear comando
Set lacmComando = New ADODB.Command
' crear parametro utilizando el Command
Set laprParametro = lacmComando.CreateParameter("CustomerID", _
   adChar, adParamInput, 5, "CACTU")
' agregar el parámetro a la colección del objeto comando
lacmComando.Parameters.Append laprParametro
' establecer en el comando, la conexión, nombre
' del procedimiento almacenado y tipo de comando
Set lacmComando.ActiveConnection = lacnConexion
lacmComando.CommandText = "DatosCliente"
lacmComando.CommandType = adCmdStoredProc
' ejecutar el comando, obteniendo un Recordset
' como resultado
Set larsRecordset = lacmComando.Execute
' mostrar los campos del Recordset (sólo tiene un registro)
MsgBox "Cliente: " & larsRecordset("CustomerID") & vbCrLf & _
   "Empresa: " & larsRecordset("CompanyName") & vbCrLf & _
   "Ciudad: " & larsRecordset("City"), , "Información del cliente"
' cerrar objetos y liberar memoria
lacnConexion.Close
larsRecordset.Close

Set lacnConexion = Nothing
Set lacmComando = Nothing
Set laprParametro = Nothing
Set larsRecordset = Nothing

End Sub
```

Código fuente 188

Aquí utilizamos directamente el valor del parámetro a buscar, pero en una aplicación real, dicho valor se solicitaría al usuario, de forma que ejecutando el mismo procedimiento almacenado, obtuviéramos distintos resultados en función del valor introducido por el usuario.

El resultado se muestra en la figura 193.

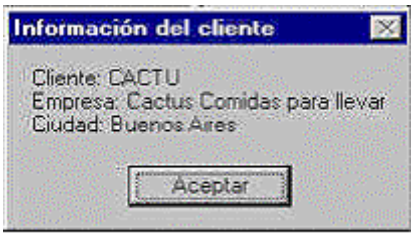


Figura 193. Resultado del procedimiento almacenado.

Modificar datos desde objetos Command

Al igual que con los objetos Connection, un objeto Command puede enviar una instrucción SQL de agregado, modificación, etc., de registros contra la base de datos sin la necesidad de recuperar un Recordset.

En la opción de menú Archivo+Comando con borrado del proyecto, se efectúa una eliminación de registros en una tabla de la base de datos incluida en este proyecto.

Es importante destacar también, que para ejecutar este comando no ha sido necesaria la creación previa de un objeto Connection, ya que todo se ha realizado desde el objeto Command, pasándole una cadena de conexión para que sea creada internamente, lo que simplifica nuestro código en gran medida, comprobémoslo en el código fuente 189.

```
Private Sub mnuArComandoBorrado_Click()
Dim lacmComando As ADODB.Command
' crear comando e información de conexión
Set lacmComando = New ADODB.Command
' asignar información de conexión al comando
lacmComando.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos

' establecer tipo de comando e instrucción SQL
lacmComando.CommandType = adCmdText
lacmComando.CommandText = "DELETE * FROM Grabaciones " & _
   "WHERE CodGrabacion >= '042' AND CodGrabacion <= '067'"
' ejecutar el comando
lacmComando.Execute

' liberar memoria usada por el objeto
Set lacmComando = Nothing
End Sub
```

Código fuente 189

Manejo de un Recordset sin usar objetos superiores

Como comentábamos al principio de este apartado, la sencillez y flexibilidad de ADO nos va a permitir instanciar y usar un objeto Recordset sin necesidad de pasar por los objetos de nivel superior de esta jerarquía.

La opción Archivo+Recordset del menú, creará un Recordset basado en una consulta SQL, volcando posteriormente un campo de los registros al ListBox de un formulario para mostrarlos al usuario.

```
Private Sub mnuArRecordset_Click()
Dim larsRecordset As ADODB.Recordset
Dim lfrmVerRecordset As frmVerRecordset
' crear recordset
Set larsRecordset = New ADODB.Recordset
' configurar conexión, datos a recuperar y abrir recordset
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
larsRecordset.Source = "SELECT * FROM Grabaciones " & _
   "WHERE CodGrabacion >= '060' AND CodGrabacion <= '090'" & _
   "ORDER BY Titulo"
```

```
larsRecordset.Open
' abrir formulario y mostrar en
' un ListBox los registros del Recordset
Set lfrmVerRecordset = New frmVerRecordset
Load lfrmVerRecordset
lfrmVerRecordset.Caption = "Valores del Recordset"
lfrmVerRecordset.Show
' pasar un campo de los registros al ListBox
Do While Not larsRecordset.EOF
   lfrmVerRecordset.lstDatos.AddItem larsRecordset("Titulo")
   larsRecordset.MoveNext
Loop
' cerrar y eliminar de memoria el recordset
larsRecordset.Close
Set larsRecordset = Nothing
End Sub
```

Código fuente 190

La figura 194 muestra el resultado de ejecutar este menú.

Figura 194. Valores del campo de un Recordset creado directamente.

Aunque esta práctica puede ser bastante socorrida en situaciones en las que debamos crear rápidamente un Recordset, debemos tener en cuenta que este modo de trabajo crea una conexión por cada Recordset abierto, lo cual consume una mayor cantidad de recursos.

Si necesitamos en un determinado procedimiento o módulo crear varios recordsets, es más recomendable crear en primer lugar un objeto Connection con ámbito de módulo o procedimiento y utilizarlo como conexión única para abrir todos los recordsets necesarios, nuestro código será más rápido y consumirá menos recursos.

Guardar la ruta de datos en el registro de Windows

Una cualidad muy apreciada por los usuarios y que hará ganar muchos puntos a la aplicación, es permitir a dicho usuario la elección de la ruta de los archivos de datos que componen la aplicación. Esto supone para el programador, el inconveniente de tener que desarrollar el programa, desconociendo a priori la ubicación de los ficheros de datos.

El proyecto de ejemplo RutaDatos, incluido en este apartado, muestra una posible solución a este problema, basada en el uso del Registro de Windows para guardar la ruta de la base de datos empleada por una aplicación. Una vez grabada dicha información, se recuperará cada vez que se inicie la aplicación.

Recomendamos al lector un repaso al tema *El Registro de Windows*, donde se detallan las técnicas para obtener y grabar valores en el registro de configuraciones del sistema.

La mecánica de este programa es muy simple. Se inicia desde un procedimiento Main() en el que se hace una llamada a la función RutaDatosRegistro(), encargada de buscar la ruta en el Registro de Windows, en función de los valores pasados como parámetro. Si no encuentra información, solicita la ruta al usuario para grabarla y utilizarla en las sucesivas ejecuciones, de manera que no haya que pedirla constantemente al usuario y tampoco necesitemos emplear una ruta prefijada por la aplicación.

El código fuente 191 corresponde al módulo de código General.BAS, que contiene el código tanto del procedimiento Main() como de la función que realiza el trabajo con el Registro.

```
Option Explicit
Public gcRutaDatos As String
**********************************
Public Sub Main()
Dim larsRecordset As ADODB.Recordset
Dim lfrmRutaDatos As frmRutaDatos
' llamar a la función que busca la ruta de
' los datos en el registro
gcRutaDatos = RutaDatosRegistro("RutaDatos", "ValoresApp", _
    "Ubicacion", "musica.mdb")
' si no existe la ruta en el registro, salir
If Len(gcRutaDatos) = 0 Then
   MsgBox "No existen o son incorrectos los datos del Registro", , "Error"
   Exit Sub
End If
' instanciar recordset
Set larsRecordset = New ADODB.Recordset
' establecer datos de conexión
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "musica.mdb"
' abrir recordset
larsRecordset.Open "Grabaciones", , adOpenStatic, adLockOptimistic, _
   adCmdTable
' abrir el formulario y asignar el recordset al DataGrid
Set lfrmRutaDatos = New frmRutaDatos
Load lfrmRutaDatos
Set lfrmRutaDatos.grdRuta.DataSource = larsRecordset
lfrmRutaDatos.Show
End Sub
**********************************
Function RutaDatosRegistro(ByVal vcNombreApp As String, _
   ByVal vcSeccion As String, ByVal vcClave As String, _
   ByVal vcFichDatos As String) As String

' tomar del registro la ruta de
' la base de datos. Si no existe
' información sobre esta aplicación
' en el registro, solicitarla al
' usuario
Dim lcRutaDatos As String
Dim lcFichero As String
Do While True
   ' inicializar valor de retorno
   RutaDatosRegistro = ""
   ' obtener del registro la ruta de la base de datos
   lcRutaDatos = GetSetting(vcNombreApp, vcSeccion, vcClave)

   ' si había algún valor en el registro
   ' comprobar si corresponde con una
   ' ruta válida de datos
```

```
   If Len(lcRutaDatos) > 0 Then
      lcFichero = Dir(lcRutaDatos & vcFichDatos)

      If Len(lcFichero) = 0 Then
         ' no es válida la ruta del fichero
         lcRutaDatos = ""
      Else
         ' si es válida la ruta del fichero,
         ' asignarla al valor de retorno
         RutaDatosRegistro = lcRutaDatos
         Exit Function
      End If
   End If

   ' si no hay en el registro una entrada
   ' para esta aplicación, crearla
   If Len(lcRutaDatos) = 0 Then
      lcRutaDatos = InputBox("Introducir la ruta del fichero de datos")

      ' si lcRutaDatos no tiene valor
      ' significa que el usuario no ha
      ' proporcionado la ruta de datos
      ' salir
      If Len(lcRutaDatos) = 0 Then
         RutaDatosRegistro = ""
         Exit Function
      Else

         If Right(lcRutaDatos, 1) <> "\" Then
            lcRutaDatos = lcRutaDatos & "\"
         End If

         ' si lcRutaDatos tiene valor,
         ' crear entrada en el registro
         SaveSetting vcNombreApp, vcSeccion, vcClave, lcRutaDatos
         lcRutaDatos = ""

         ' ahora volvemos al inicio del bucle
         ' para volver a recuperar el valor
         ' del registro y comprobar si la
         ' ruta de datos es correcta

      End If
   End If
Loop
End Function
```

Código fuente 191

Una vez obtenida la ruta de los datos de este programa, se utilizará para crear un Recordset y asignarlo al control DataGrid de un formulario en el que visualizaremos una de las tablas de la base de datos, tal como muestra la figura 195.

Figura 195. Formulario visualizando registros después de recuperar la ruta de los datos del Registro.

Si ejecutamos la aplicación RegEdit de Windows y buscamos alguno de los valores que componen la entrada que hemos creado en el Registro, nos encontraremos con una ventana similar a la figura 196.

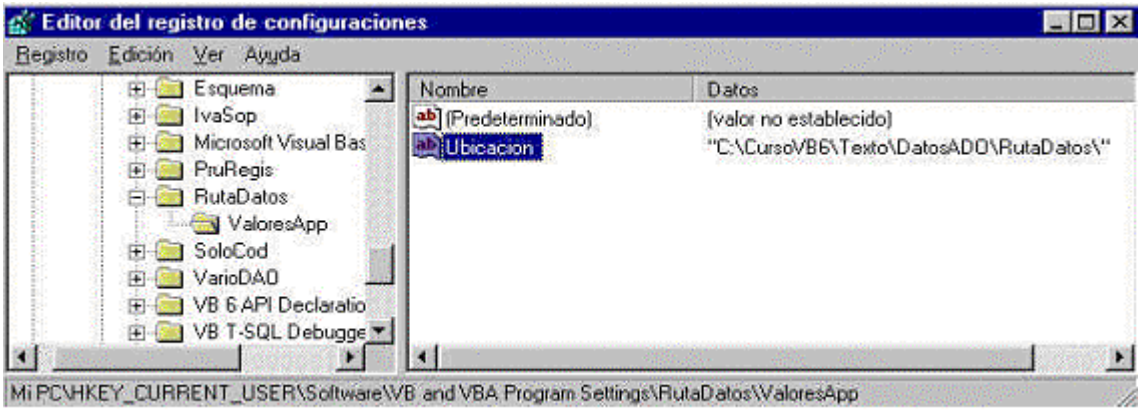


Figura 196. Registro del sistema mostrando los valores introducidos para la ruta de los datos de una aplicación.

Pruebe el lector a ejecutar este ejemplo introduciendo valores diferentes, buscándolos posteriormente en el Registro para comprobar como han quedado guardados.

Índices

Debido a que el diseño de ADO está orientado al manejo de datos en cualquier formato, el aspecto de la ordenación de registros mediante índices, si bien ya está implementado en esta versión, actualmente se sigue trabajando en la mejora del mismo, de modo que las sucesivas versiones de esta tecnología ofrezca mejores características en cuanto a la rapidez en el orden y búsqueda de los registros.

La forma de trabajar con índices en ADO, pasa por crear un índice para el campo que necesitemos ordenar, utilizando la propiedad Optimize de uno de los objetos Field contenidos en un Recordset. Dicho Recordset debe además tener establecido el cursor del lado del cliente mediante su propiedad CursorLocation. El código fuente 192 es una muestra de como crear un índice para un campo en un objeto Recordset.

```
' crear recordset
Set rsRecordset = New ADODB.Recordset
' establecer datos de conexión
rsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=C:\PRUEBAS\DATOS.MDB"
' cursor del lado cliente
rsRecordset.CursorLocation = adUseClient
' abrir recordset
rsRecordset.Open "SELECT * FROM Clientes", , adOpenStatic, _
   adLockOptimistic, adCmdText
' crear índice para un campo del recordset
marsRecordset("Nombre").Properties("Optimize") = True
```

Código fuente 192

Al asignar True a la propiedad Optimize se creará el índice. Para eliminarlo, basta con asignar a Optimize el valor False.

Una vez creado el índice de esta manera, su manejo correrá a cargo de ADO, no pudiendo el programador acceder a él de ninguna forma, ya que es de uso interno para estos objetos.

El uso de índices en ADO, optimiza el rendimiento del objeto Recordset al ordenar sus registros con la propiedad Sort, al establecer filtros con la propiedad Filter, y al realizar búsquedas mediante el método Find().

Órdenes

Tomando como base el apartado anterior, para ordenar uno o más campos de un Recordset, podemos hacerlo de dos maneras que vamos a ver en el programa de ejemplo OrdenSort.

En primer lugar, esta aplicación solicita al usuario en el procedimiento Main(), la ruta de la base de datos Datos.MDB, que se adjunta en este proyecto, para guardarla en el Registro de la manera que vimos en uno de los apartados anteriores. La función RutaDatosRegistro() es la misma que se usó en dicho apartado, por lo que evitaremos repetirla aquí.

```
Public Sub Main()
Dim lfrmOrden As frmOrden
' llamar a la función que busca la ruta de
' los datos en el registro
gcRutaDatos = RutaDatosRegistro("OrdenSort", "Informac", _
    "Ruta", "datos.mdb")
' si no existe la ruta en el registro, salir
If Len(gcRutaDatos) = 0 Then
   MsgBox "No existen o son incorrectos los datos del Registro", , "Error"
   Exit Sub
End If
gcRutaDatos = gcRutaDatos & "datos.mdb"
' abrir el formulario
Set lfrmOrden = New frmOrden
Load lfrmOrden
lfrmOrden.Show
End Sub
```

Código fuente 193

Este ejemplo utiliza el formulario de pruebas frmOrden, que creará un Recordset y lo visualizará en un DataGrid. El evento Load() muestra la creación inicial del Recordset. Observe el lector el establecimiento del cursor en el lado del cliente en el código fuente 194.

```
Private Sub Form_Load()
Dim larsRecordset As ADODB.Recordset
' crear recordset
Set larsRecordset = New ADODB.Recordset
' establecer datos de conexión
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
' cursor del lado cliente
larsRecordset.CursorLocation = adUseClient
' abrir recordset
```

```
larsRecordset.Open "SELECT * FROM Clientes", , adOpenStatic, _
   adLockOptimistic, adCmdText
Set Me.grdOrden.DataSource = larsRecordset
End Sub
```

Código fuente 194

Una vez abierto el formulario de pruebas, la primera manera de ordenar los registros se basa en una consulta SQL que incluya la cláusula ORDER BY. La opción Ordenar+Sentencia SQL en el menú del formulario creará un nuevo Recordset ordenado por un campo y lo asignará al DataGrid.

```
Private Sub mnuOrSQL_Click()
Dim larsRecordset As ADODB.Recordset
' crear recordset
Set larsRecordset = New ADODB.Recordset
' establecer datos de conexión
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
' cursor del lado cliente
larsRecordset.CursorLocation = adUseClient
' abrir recordset
larsRecordset.Open "SELECT * FROM Clientes ORDER BY Nombre", , _
   adOpenStatic, adLockOptimistic, adCmdText
' asignar recordset al datagrid
Set Me.grdOrden.DataSource = larsRecordset
End Sub
```

Código fuente 195

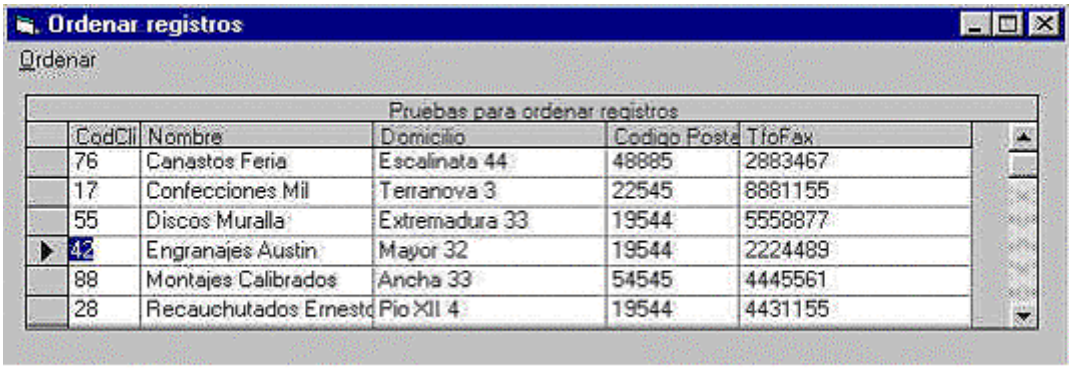


Figura 197. Registros ordenados por el campo Nombre mediante una instrucción SQL.

El otro modo de ordenar los registros consiste en utilizar la propiedad Sort del Recordset, asignándole una cadena con el nombre del campo a ordenar y el modo en que se ordenarán los registros, en ascendente o descendente.

La opción del menú Ordenar+Sort - un campo, cambiará el orden actual del Recordset por uno nuevo basado en el campo Domicilio. Previamente habrá creado el índice correspondiente para el campo con la propiedad Optimize del campo.

```
Private Sub mnuOrUn_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
```

```
Set larsRecordset = Me.grdOrden.DataSource
' crear índice para el campo a ordenar
larsRecordset("Domicilio").Properties("Optimize") = True
' establecer orden para un campo
larsRecordset.Sort = "Domicilio ASC"
End Sub
```

Código fuente 196

Pero no sólo podemos ordenar un campo a la vez, Sort nos permite ordenar varios campos y con distintos modos. Para finalizar con este ejemplo, la opción Ordenar+Sort - dos campos además de ordenar dos campos del Recordset, uno será ascendente y el otro descendente.

```
Private Sub mnuOrDos_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdOrden.DataSource
' crear índice para los campos a ordenar
larsRecordset("Codigo Postal").Properties("Optimize") = True
larsRecordset("TfoFax").Properties("Optimize") = True
' establecer orden para los campos
' combinando ordenación ascendente con descendente
' Los campos con nombres compuestos se ponen
' entre corchetes
larsRecordset.Sort = "[Codigo Postal] ASC, TfoFax DESC"
End Sub
```

Código fuente 197

Observe el lector que para los nombres de campos compuestos deberemos encerrar el nombre entre corchetes.

El resultado de esta opción lo vemos en la figura 198.

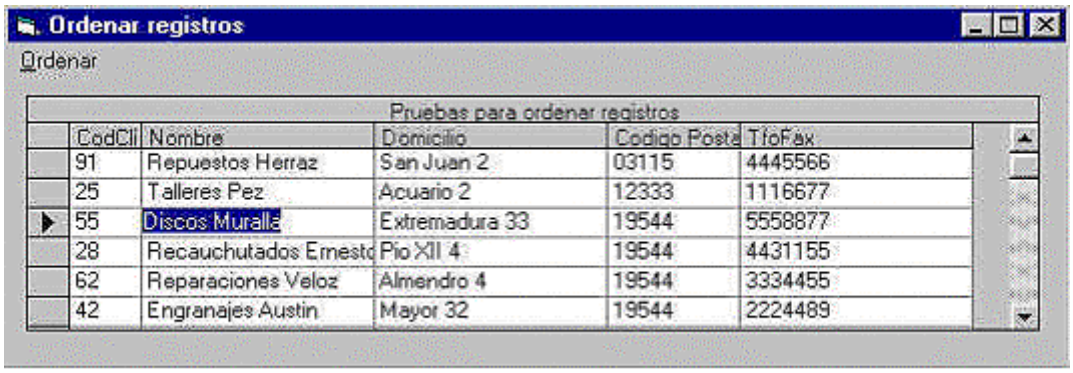


Figura 198. Registros ordenados por varios campos mediante la propiedad Sort.

Filtros

Al igual que sucedía en el apartado anterior, podemos establecer filtros en un Recordset utilizando una instrucción SQL o mediante la propiedad Filter del objeto.

El programa de ejemplo FiltroDatos dispone de una mecánica similar a la del ejemplo anterior: utiliza un formulario con un DataGrid para mostrar un Recordset al que se le aplicarán los filtros. Una vez cargado este formulario, podemos utilizar una sentencia SQL con la cláusula WHERE para mostrar los registros que cumplan determinada condición, tal como vemos en la opción Filtrar+Sentencia SQL del menú del formulario.

```
Private Sub mnuFiSQL_Click()
Dim larsRecordset As ADODB.Recordset
' crear recordset
Set larsRecordset = New ADODB.Recordset
' establecer datos de conexión
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
' cursor del lado cliente
larsRecordset.CursorLocation = adUseClient
' abrir recordset
larsRecordset.Open "SELECT * FROM Clientes " & _
   "WHERE CodCli > '30' AND CodCli < '80'", , _
   adOpenStatic, adLockOptimistic, adCmdText
' asignar recordset al datagrid
Set Me.grdFiltro.DataSource = larsRecordset
End Sub
```

Código fuente 198

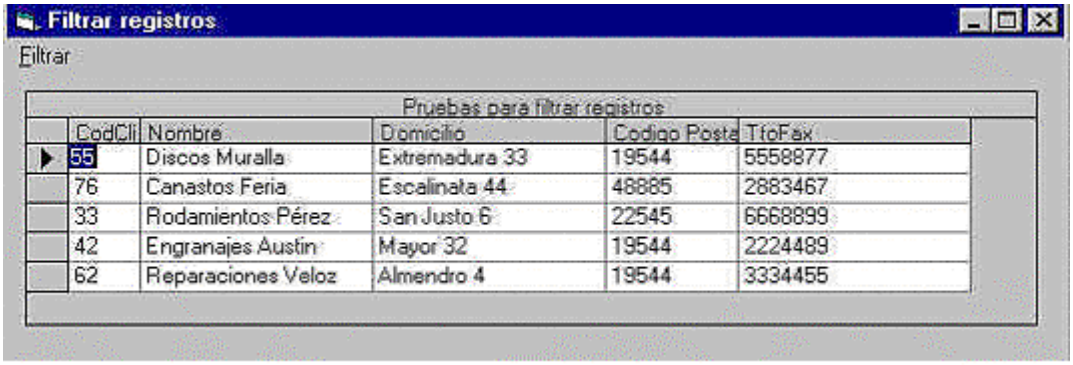


Figura 199. Registros resultantes de filtrar mediante una instrucción SQL.

Para establecer un filtro mediante la propiedad Filter, disponemos de la opción de menú Filtrar+Establecer filtro, que vemos en el código fuente 199 y en la figura 200.

```
Private Sub mnuFiEstablecer_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdFiltro.DataSource
' crear índice para el campo a filtrar
larsRecordset("Codigo Postal").Properties("Optimize") = True
' establecer un filtro sobre los registros
' existentes en el recordset
larsRecordset.Filter = "[Codigo Postal] > '10000' AND " & _
   "[Codigo Postal] < '20000'"
End Sub
```

Código fuente 199

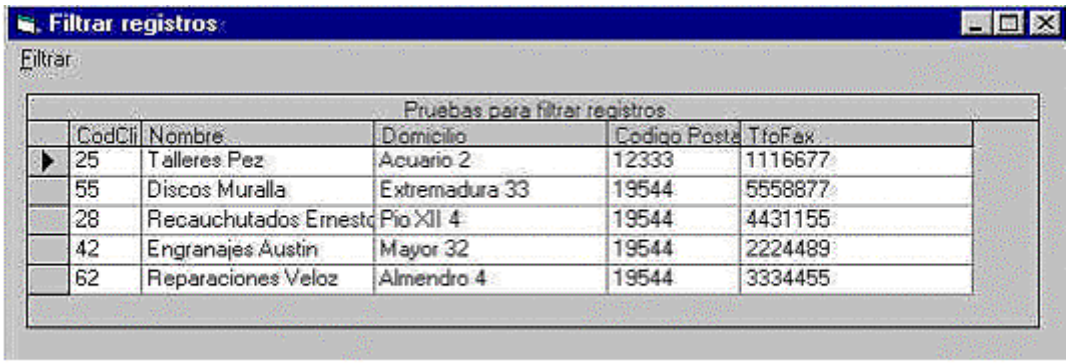


Figura 200. Registros obtenidos al aplicar la propiedad Filter.

Para eliminar el filtro y restablecer el Recordset a su estado original, tenemos la opción de menú Filtrar+Eliminar filtro. Observemos, en el código fuente 200, como después de eliminar el filtro, debemos llamar al método Requery() para volver a ejecutar la consulta en la que se basa el Recordset.

```
Private Sub mnuFiEliminar_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdFiltro.DataSource
' eliminar filtro actual y mostrar
' todos los registros
larsRecordset.Filter = adFilterNone
larsRecordset.Requery
End Sub
```

Código fuente 200

Búsqueda de registros

Para localizar un registro en un objeto Recordset disponemos del método Find(), que realiza una búsqueda en un campo del Recordset sobre la base de un criterio pasado como parámetro. En este apartado emplearemos la aplicación BuscarRegist que contiene diversos ejemplos de búsquedas en un Recordset que utiliza la misma base de datos que los anteriores apartados. Los resultados serán mostrados en el formulario frmBuscar mediante un control DataGrid. Comenzaremos por una búsqueda sencilla, seleccionando en el menú del formulario la opción Buscar+Normal, se ejecutará el código fuente 201.

```
Private Sub mnuBuNormal_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdBuscar.DataSource
' crear índice para el campo
' en el que vamos a buscar
larsRecordset("CodCli").Properties("Optimize") = True
' buscar valor en el campo
larsRecordset.Find "CodCli = '76'"
End Sub
```

Código fuente 201

Esta opción situará el puntero del Recordset en el registro indicado, tal y como vemos en la figura 201.

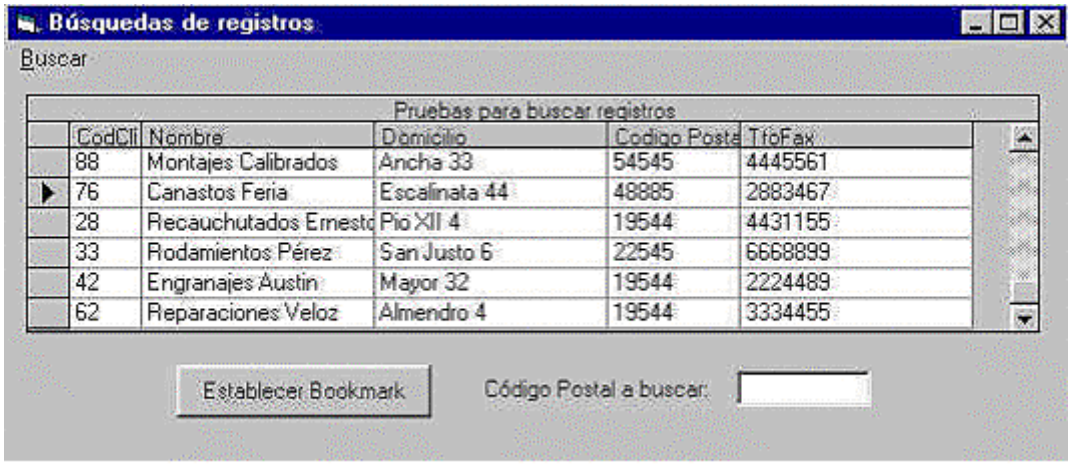


Figura 201. Búsqueda sencilla en un Recordset.

Una de las características de Find() es que nos permite realizar la búsqueda hacia los registros anteriores al que estamos posicionados. La opción Buscar+Hacia atrás del programa de ejemplo, posiciona el puntero del Recordset en un registro para luego realizar una búsqueda en los registros anteriores al actual. El código fuente 202 nos muestra dicha opción.

```
Private Sub mnuBuAtras_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdBuscar.DataSource
' crear índice para el campo
' en el que vamos a buscar
larsRecordset("Nombre").Properties("Optimize") = True
' situar el recordset por delante del
' registro que vamos a buscar
larsRecordset.Move 7
' buscar el registro hacia atrás
larsRecordset.Find "Nombre = 'Zapatos Planta'", , adSearchBackward
End Sub
```

Código Fuente 202

Para finalizar con los ejemplos sobre Find(), vamos a realizar una búsqueda a partir de un registro del Recordset especificado por un Bookmark. Aquí necesitaremos la ayuda del usuario, que pulsando el botón Establecer Bookmark, obtendrá el marcador del registro en que se encuentre actualmente el puntero del Recordset, asignándolo a una variable a nivel de módulo. Posteriormente teclearemos un valor en el TextBox que emplearemos para buscar por la columna de códigos postales y finalmente seleccionaremos la opción de menú Buscar+Desde Bookmark, que realizará la búsqueda como se muestra en el código fuente 203.

```
Private Sub mnuBuBookmark_Click()
Dim larsRecordset As ADODB.Recordset
' recuperar recordset del datagrid
Set larsRecordset = Me.grdBuscar.DataSource
```

```
' buscar campo partiendo desde
' un bookmark previamente definido
larsRecordset.Find "[Codigo Postal] = '" & Me.txtValor.Text & "'", _
   , , mvntBookmark
End Sub
```

Código fuente 203

Aunque en estos ejemplos no se ha incluido, debemos tener en cuenta que si la búsqueda no tiene éxito, el puntero del Recordset se situará en EOF o BOF dependiendo de la dirección en que se realice la búsqueda. Es por ello que ahora el Bookmark tiene mayor importancia, ya que si es soportado, podemos utilizarlo para guardar el registro en donde estábamos y volver a él en caso de no tener éxito la búsqueda.

La figura 202, muestra el resultado de una búsqueda empleando un Bookmark que se ha establecido al tercer registro del Recordset.

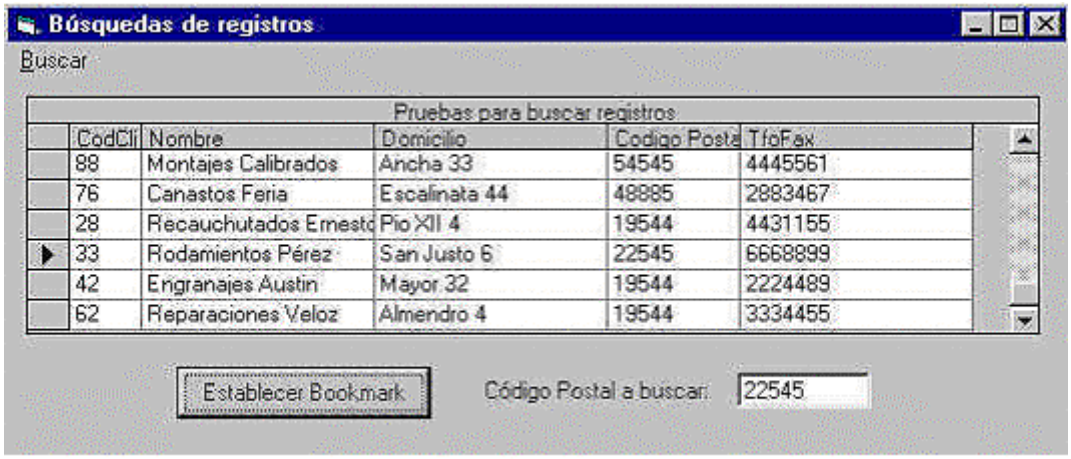


Figura 202. Búsqueda empleando un Bookmark.

Navegación por los registros

Para desplazarnos entre los registros de un Recordset, disponemos de los métodos *Move*, cuyo cometido puede consultarlo el lector en el apartado dedicado a los objetos ADO dentro de este mismo tema.

Las propiedades *BOF* y *EOF* del Recordset, devuelven un valor lógico que indican cuando se ha llegado al principio o final de los registros, respectivamente.

El programa de ejemplo NavegaRegist se encargará de abrir el formulario *frmNavegacion*, que nos mostrará de forma práctica todo este movimiento de registros en el interior del Recordset.

La mayoría de las operaciones realizadas por los botones son autoexplicativas. Podríamos destacar el control Label *lblNroRegistros,* que indica el número de registros de la tabla, usando la propiedad *RecordCount* del Recordset en el evento de carga del formulario que podemos ver en el código fuente 204.

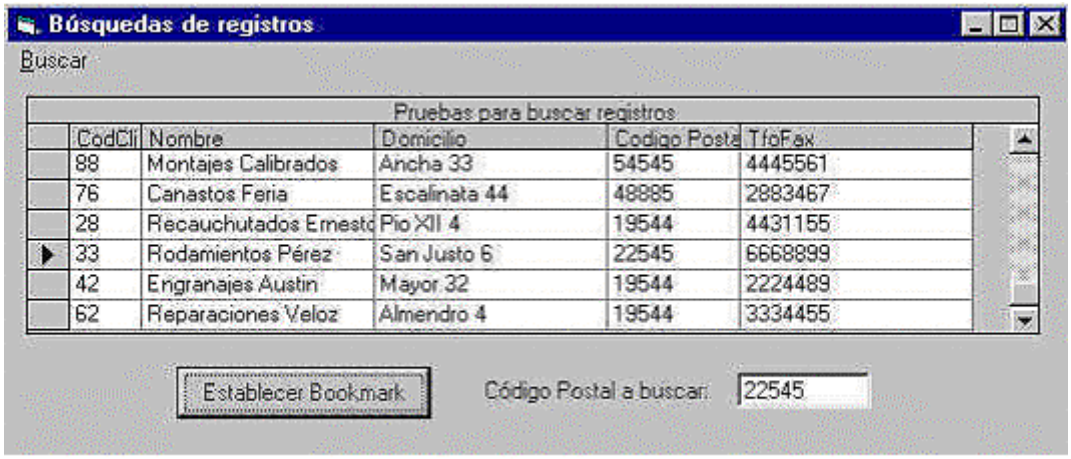


Figura 203. Ventana de desplazamiento entre los registros de un Recordset.

```
Private Sub Form_Load()
Dim larsRecordset As ADODB.Recordset

' crear recordset
Set larsRecordset = New ADODB.Recordset
' establecer datos de conexión
larsRecordset.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
' abrir recordset
larsRecordset.Open "Clientes", , adOpenStatic, adLockOptimistic, _
   adCmdTable
' ir al último registro y volver al primero
' para que la propiedad RecordCount refleje
' el valor adecuado
larsRecordset.MoveLast
larsRecordset.MoveFirst
Me.lblNroRegistros.Caption = larsRecordset.RecordCount
' asignar el recordset al control ADO
Set Me.adcClientes.Recordset = larsRecordset
End Sub
```

Código fuente 204

La propiedad *RecordCount* devuelve el número de registros que contiene el Recordset. Para asegurarnos que es un valor fiable, nada más crear el Recordset, es conveniente llamar a *MoveLast*, con lo cual se fuerza un recuento de registros y actualización de *RecordCount*. Después llamamos a *MoveFirst* con lo que volvemos de nuevo al registro inicial.

También es importante resaltar que al avanzar o retroceder filas en el Recordset, si llegamos al principio o final del mismo (propiedades BOF y EOF a True respectivamente), estaremos en una zona del objeto en la que no existe registro, por lo tanto hemos de reposicionar el puntero del Recordset en el primer o último registro, en función de donde nos encontremos.

```
Private Sub cmdAnterior_Click()
' retrocede un registro
Me.adcClientes.Recordset.MovePrevious
' si nos encontramos en el principio del
' recordset, reposicionar al primer registro
```

```
If Me.adcClientes.Recordset.BOF Then
   Me.adcClientes.Recordset.MoveFirst
   MsgBox "Primer registro", , "Movimiento de registros"
End If
End Sub
' ----------------------
Private Sub cmdSiguiente_Click()
' avanza al siguiente registro de la tabla
Me.adcClientes.Recordset.MoveNext
' si nos encontramos en el final del
' recordset, reposicionar al último registro
If Me.adcClientes.Recordset.EOF Then
   Me.adcClientes.Recordset.MoveLast
   MsgBox "Último registro", , "Movimiento de registros"
End If
End Sub
```

Código Fuente 205

Si en algún momento necesitamos guardar la posición de un registro del Recordset para después regresar a él, disponemos de la propiedad *Bookmark*, que devuelve un valor que identifica de forma única a un registro. Los botones *Marcar Registro* y *Volver a la marca, código fuente 206*, muestran como hacerlo.

```
' ----------------------
Private Sub cmdMarcar_Click()
' captura el valor del marcador de registros
' para poder volver a este registro posteriormente
mvntMarcador = Me.adcClientes.Recordset.Bookmark

End Sub
' ----------------------
Private Sub cmdVolver_Click()
' vuelve al registro marcado cuyo valor
' está en la variable mvntMarcador
Me.adcClientes.Recordset.Bookmark = mvntMarcador
End Sub
```

Código fuente 206

El *Bookmark* se guarda en la variable *mvntMarcador*, definida a nivel de módulo para que su valor sea visible desde todos los procedimientos de este formulario.

En el caso de que la aplicación use una base de datos distinta de Access, hay que tener la precaución de comprobar la propiedad *Bookmarkable* del Recordset creado, para ver si soporta esta característica.

Como curiosidad sobre el método *Move*, al introducir un valor en el TextBox *txtMoverReg* que hay dentro del Frame *Mover Registros* y pulsar Enter, el puntero se desplazará el número de registros indicado. Pero si además marcamos el CheckBox *Usando marcador*, el desplazamiento de registros se realizará a partir del registro indicado por el marcador que se haya definido al pulsar *Marcar Registro.*

Validación de controles

Hasta la llegada de Visual Basic 6, cuando en un formulario de entrada de datos era necesario comprobar que los valores introducidos en los respectivos controles de dicho formulario fueran

correctos, el programador debía recurrir a efectuar las validaciones oportunas en el evento LostFocus() del control que lo necesitara, o bien en el CommandButton u otro control encargado de aceptar el conjunto de todos los controles del formulario, o recurrir a otra técnica desarrollada por el programador para este cometido.

Sin embargo, el uso de LostFocus() entraña el problema de que es un evento producido cuando el control a validar ya ha perdido el foco, como indica su propio nombre, por lo que si su información no es adecuada, debemos volver a asignar el foco al control que lo tenía. Esto conlleva el problema añadido de que en ciertos contextos de validación, el flujo de la aplicación puede caer en un bucle infinito que bloquee el sistema.

Afortunadamente, a partir de VB 6 los controles incorporan la propiedad CausesValidation y el evento Validate(), que empleados conjuntamente, proporcionan al programador un medio más eficaz de realizar la validación del contenido de los controles de un formulario.

CausesValidation contiene un valor lógico que si es establecido a True en un primer control, hará que al intentar pasar el foco a un segundo control, y siempre que este segundo control también tenga True en esta propiedad, sea llamado antes de cambiar el foco el evento Validate() del primer control. Es en este evento donde el programador deberá escribir el código de validación para el control, y en el caso de que no sea correcto, establecer el parámetro Cancel de este evento a True, para forzar a mantener el foco en el primer control al no haber cumplido la regla de validación. De esta manera disponemos de un medio más flexible y sencillo de manejar las reglas de validación establecidas para los controles.

El esquema de funcionamiento de esta técnica de validaciones se muestra en la figura 204.

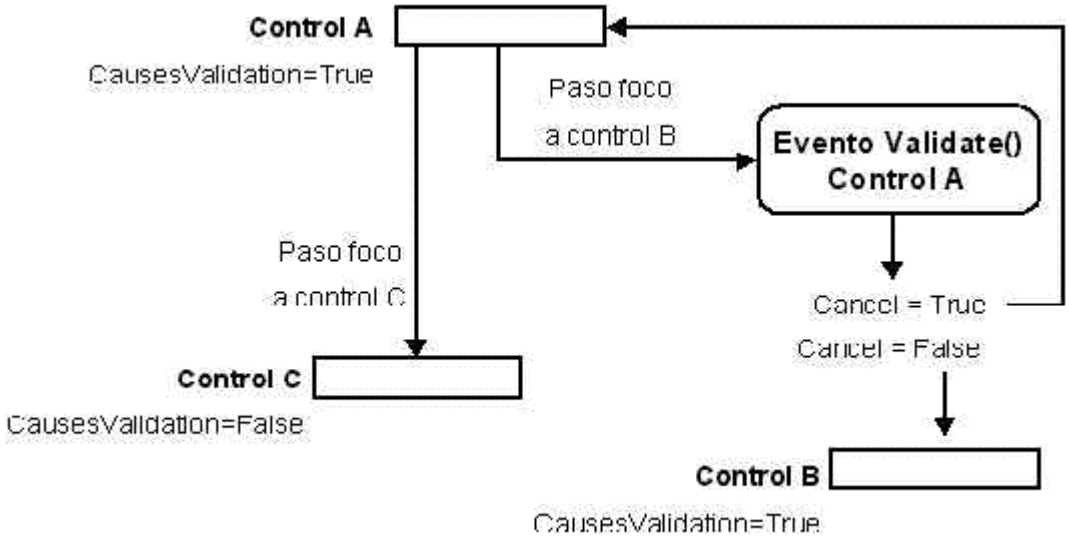


Figura 204. Esquema de funcionamiento de validaciones.

Según el esquema de funcionamiento mostrado en la anterior figura, al pasar del control A al B, se ejecuta el código de Validate(), si dentro de este evento no se cumplen las reglas de validación y se establece Cancel a True, el foco se devuelve al control A. En el caso de que todo sea correcto y Cancel valga False, el foco pasará al control B.

Para la situación de paso de foco desde el control A al C, no existirá impedimentos, ya que al tener el control C su propiedad CausesValidation establecida a False, no se ejecutará Validate() y siempre llegará el foco al control.

Sin embargo, lo que puede no llegar a ser tan intuitivo en este último caso es que una vez situados en el control C, si intentamos pasar el foco al control B no será posible ya que este si tiene el Valor True en CausesValidation, ejecutándose el Validate() del control A.

El ejemplo que el lector puede encontrar en ValidarCtls, consiste en un formulario de entrada de datos, en el que todos los controles tienen el valor True en CausesValidation excepto el botón cmdCancelar. Adicionalmente, en el TextBox txtCodigo se ha incluido el código fuente 207 en su método Validate(), de modo que el usuario sólo pueda introducir números.

```
Private Sub txtCodigo_Validate(Cancel As Boolean)
If Not IsNumeric(Me.txtCodigo.Text) Then
   MsgBox "Sólo se permiten valores numéricos en este campo"
   Cancel = True
End If
End Sub
```

Código fuente 207

Una vez en ejecución, si intentamos introducir un valor incorrecto en el control txtCodigo y cambiar a un control que no sea cmdCancelar, se mostrará el aviso que aparece en la figura 205.

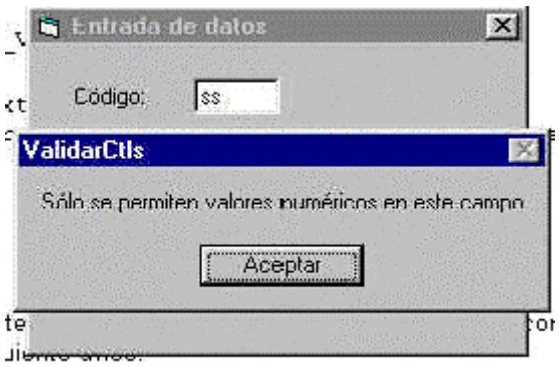


Figura 205. Aviso de error en la validación de los datos.

Crear controles en tiempo de ejecución

El modo tradicional de agregar un control al formulario durante la ejecución del programa era incluir dicho control en el formulario en tiempo de diseño y ocultarlo, para después agregar copias del mismo con la función Load().

Visual Basic nos ofrece a partir de ahora un nuevo medio para agregar controles a la aplicación en tiempo de ejecución, que nos va a proporcionar una mayor operatividad sobre los controles que incluyamos.

A la hora de añadir un control empleando únicamente código, debemos distinguir entre los controles referenciados, o aquellos que aparecen en la Caja de herramientas de VB y los no referenciados, o aquellos que aún estando instalados y registrados en el sistema, no tenemos un acceso directo a ellos porque no se han incorporado a la paleta de herramientas.

El programa de ejemplo CrearCtls, desde el menú de su ventana principal mdiInicio, nos mostrará la manera de crear controles en un formulario sólo con código, tanto si están referenciados, como si no lo están.

Para el primer caso, controles referenciados, la manera de añadirlos al formulario pasa por utilizar el método Add() de la colección Controls del formulario, que se muestra a continuación.

- Sintaxis

  ```
  Controls.Add(IDControl, cNombre, oContenedor)
  ```

- Parámetros

  - IDControl. Cadena con el identificador del control. Esta cadena se compone del nombre de la librería de objetos en que está contenido el control y la clase del control. Mediante el Examinador de objetos de VB es posible obtener esta información para emplear posteriormente en el programa.

  - cNombre. Cadena con el nombre que asignamos al control.

  - oContenedor. Opcional. Un objeto o control que actuará de contenedor para el control que vamos a crear. Si por ejemplo especificamos el nombre de un Frame que esté en el formulario, nuestro nuevo control se creará dentro de dicho Frame.

La opción de menú Archivo+Referenciados en el programa de ejemplo, muestra un formulario destinado a incorporar la información de nuevos usuarios a una aplicación.

En un principio, sólo disponemos de un TextBox en el que se deberá incluir una clave de acceso, si dicha clave pertenece al administrador del sistema, cuando pulsemos el botón Acceder se crearán dos TextBox con sus correspondientes Label para introducir el nombre y clave de acceso del usuario, así como un CommandButton para validar esta información.

En el código fuente 208 se muestra el código de este botón, incluyendo también la declaración de los controles a crear manualmente, que tienen un ámbito a nivel de módulo.

```
Option Explicit
' variables que contendrán los controles
Private mlblClave As Label
Private mlblNombre As Label
Private WithEvents mtxtClave As TextBox
Private mtxtNombre As TextBox
Private WithEvents mcmdGrabar As CommandButton
' ------------------------------------------
Private Sub cmdAcceder_Click()
' comprobar el nivel de acceso del usuario
If Me.txtNivel.Text <> "ADMINSYS" Then
   MsgBox "No tiene los permisos adecuados para entrar", , "Error"
   Exit Sub
End If
' crear los controles para el formulario
' utilizando código
Set mlblNombre = Me.Controls.Add("VB.Label", "lblNombre")
mlblNombre.Visible = True
mlblNombre.Top = 810
mlblNombre.Left = 225
mlblNombre.Height = 285
mlblNombre.Caption = "Nombre:"
```

```
Set mlblClave = Me.Controls.Add("VB.Label", "lblClave")
mlblClave.Visible = True
mlblClave.Top = 1215
mlblClave.Left = 225
mlblClave.Height = 285
mlblClave.Caption = "Clave:"
Set mtxtNombre = Me.Controls.Add("VB.TextBox", "txtNombre")
mtxtNombre.Visible = True
mtxtNombre.Top = 810
mtxtNombre.Left = 945
mtxtNombre.Height = 285
Set mtxtClave = Me.Controls.Add("VB.TextBox", "txtClave")
mtxtClave.Visible = True
mtxtClave.Top = 1215
mtxtClave.Left = 945
mtxtClave.Height = 285
Set mcmdGrabar = Me.Controls.Add("VB.CommandButton", "cmdGrabar")
mcmdGrabar.Visible = True
mcmdGrabar.Top = 900
mcmdGrabar.Left = 2430
mcmdGrabar.Height = 330
mcmdGrabar.Width = 1455
mcmdGrabar.Caption = "Grabar datos"
End Sub
```

Código fuente 208

El formulario con los nuevos controles se vería como en la figura 206.

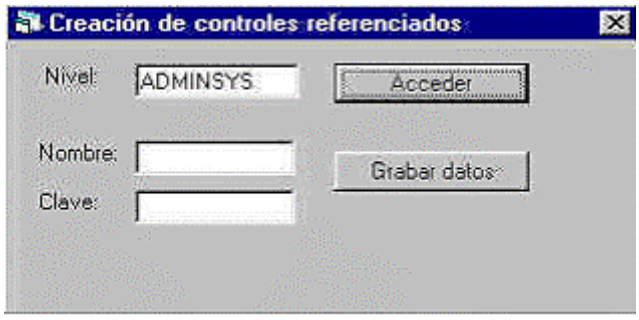


Figura 206. Formulario con controles agregados mediante código.

Todo esto está muy bien, pero el lector se estará preguntando en donde incluir el código de los controles creados en ejecución, para que dichos controles respondan a las acciones del usuario, por ejemplo al pulsar el CommandButton Grabar datos.

Para ello deberemos incluir la palabra clave WithEvents en la declaración de las variables de los controles. De esta manera accederemos a los eventos de esas variables, ya que al abrir la lista de objetos en el módulo de código, aparecerán también las variables declaradas de esta forma, y una vez seleccionada una de ellas, podremos escribir el código para sus eventos. En el código fuente 209, vamos a incluir código en el evento KeyPress() del control txtClave, para obligar al usuario a utilizar sólo números en la clave. También escribiremos código para el evento Click() del botón cmdGrabar, en el que mostraremos los datos del usuario y eliminaremos los controles que hemos creado anteriormente empleando el método Remove() de la colección Controls del formulario.

```
Private Sub mtxtClave_KeyPress(KeyAscii As Integer)
' comprobar la validez de los datos para la clave
```

```
If Not IsNumeric(Chr(KeyAscii)) Then
   KeyAscii = 0
   MsgBox "Sólo se permiten valores numéricos", , "Error"
End If
End Sub
' ------------------------------------------------------------
Private Sub mcmdGrabar_Click()
' después de introducir la información
' sobre el usuario, visualizar dichos datos
MsgBox "Nombre: " & mtxtNombre.Text & vbCrLf & _
   "Clave: " & mtxtClave.Text, , "Datos a grabar del usuario"
' eliminar los controles creados en ejecución
Me.Controls.Remove "cmdGrabar"
Me.Controls.Remove "txtNombre"
Me.Controls.Remove "txtClave"
Me.Controls.Remove "lblNombre"
Me.Controls.Remove "lblClave"
End Sub
```

Código fuente 209

En lo que respecta a los controles no referenciados, la opción de menú Archivo+No Referenciados, nos permitirá utilizar el control Slider sin haberlo incluido en la Caja de herramientas.

Para incorporar un control de este tipo a nuestro formulario, en primer lugar deberemos utilizar el método Add() de la colección Licenses, de forma que proporcionemos la clave de licencia del control a la aplicación para que pueda utilizarlo.

- Sintaxis

  `Licenses.Add(IDControl, cClaveLicencia)`

- Parámetros

  - IDControl. Cadena con el identificador del control del que obtendremos la clave de licencia.

  - cClaveLicencia. Opcional. Cadena con la clave de la licencia.

Una vez recuperada la clave de la licencia, agregaremos el control igual que en el anterior ejemplo, utilizando el método Add() de la colección Controls. En el formulario que agrega el control Slider, tenemos el botón Crear control, que ejecuta el código fuente 210.

```
Private Sub cmdCrearControl_Click()
' tomar clave de licencia del control
Licenses.Add "MSComctlLib.Slider"
' agregar un control de tipo Slider
Set moSlider = Me.Controls.Add("MSComctlLib.Slider", "sldDesplaza")
' configurar el control y mostrarlo
' en el formulario
moSlider.Visible = True
moSlider.Height = 650
moSlider.Left = 500
moSlider.Top = 800
moSlider.Width = 2715
End Sub
```

Código Fuente 210

El formulario tendría el aspecto que muestra la figura 207.

Figura 207. Formulario con control Slider agregado mediante código.

Para poder codificar los eventos de un control no referenciado, debemos primeramente declararlo a nivel de módulo, con eventos y de tipo VBControlExtender, que está especialmente preparado para manipular controles creados con código y no referenciados en la aplicación.

El control VBControlExtender dispone del evento ObjectEvent(), en el cual centralizaremos las acciones a realizar cuando se produzcan los diferentes eventos en nuestro control. El código fuente 211 muestra como manejar el evento Click() cuando se produce sobre el control Slider, mostrando un mensaje al usuario.

```
Option Explicit
Private WithEvents moSlider As VBControlExtender
' ------------------------------------------------
Private Sub moSlider_ObjectEvent(Info As EventInfo)

Select Case Info.Name
Case "Click"
   MsgBox "Se ha pulsado sobre el control Slider"

End Select

End Sub
```

Código fuente 211

Edición de registros

Por edición de registros entendemos todas aquellas operaciones que se encargan de crear nuevos registros, modificar la información de los existentes o eliminarlos de la fuente de datos.

En este apartado vamos a repasar las diferentes opciones de que disponemos para editar los registros de una fuente de datos utilizando objetos ADO, más concretamente los objetos Connection y Recordset. Emplearemos para ello la aplicación de ejemplo EdicRegistros, en la que a través de un formulario realizaremos las operaciones más comunes de edición sobre la tabla Clientes de la base de datos Datos.MDB, que acompaña a este proyecto.

A lo largo de la aplicación usaremos un objeto Connection y un Recordset, ambos creados en el procedimiento Main(), para efectuar la edición de los registros de una tabla. Mientras que con el Recordset, las operaciones a efectuar harán un mayor uso de los métodos y propiedades del objeto;

con el objeto Connection nos basaremos en instrucciones SQL enviadas mediante el método Execute(), para efectuar las labores de edición.

```
Option Explicit
Public gcRutaDatos As String
Public gacnDatos As ADODB.Connection
Public garsClientes As ADODB.Recordset
' ------------------------------------
Public Sub Main()
Dim lfrmVerDatos As frmVerDatos
' llamar a la función que busca la ruta de
' los datos en el registro
gcRutaDatos = RutaDatosRegistro("EdicRegistros", "ValoresApp", _
    "Ubicacion", "datos.mdb")
' si no existe la ruta en el registro, salir
If Len(gcRutaDatos) = 0 Then
   MsgBox "No existen o son incorrectos los datos del Registro", , "Error"
   Exit Sub
End If
' crear conexión
Set gacnDatos = New ADODB.Connection
' establecer cadena de conexión
gacnDatos.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "datos.mdb"
' abrir conexión
gacnDatos.Open
' crear recordset
Set garsClientes = New ADODB.Recordset
' abrir recordset
garsClientes.Open "Clientes", gacnDatos, adOpenStatic, adLockOptimistic, _
   adCmdTable
' abrir formulario
Set lfrmVerDatos = New frmVerDatos
Load lfrmVerDatos
lfrmVerDatos.Show
End Sub
```

Código fuente 212

Para obtener la ruta de la base de datos, se utiliza la técnica mostrada en el apartado Guardar la ruta de datos en el registro de Windows, por lo que sugerimos la consulta de dicho apartado para cualquier duda sobre este particular.

El formulario frmVerDatos (figura 208) será el que usaremos para mostrar las diversas operaciones de edición a través de su control DataGrid.

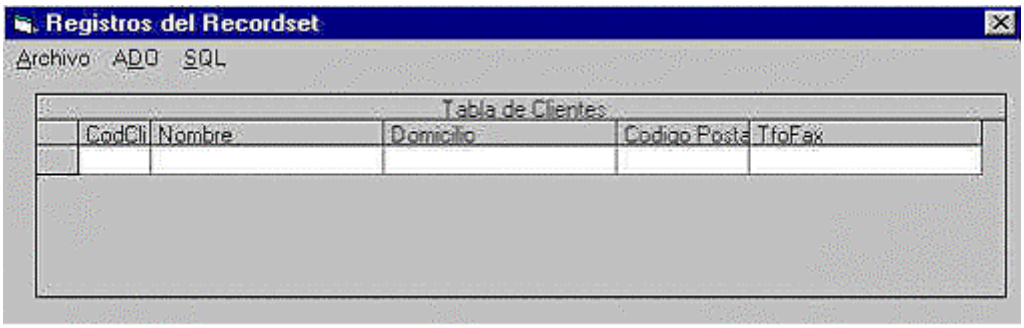


Figura 208. Formulario frmVerDatos, para mostrar la edición de registros.

El evento Load() de este formulario será el usado para asignar el Recordset al DataGrid, como vemos en el código fuente 213.

```
Private Sub Form_Load()
Set Me.grdClientes.DataSource = garsClientes
End Sub
```

Código fuente 213

Agregar con Recordset

La opción de menú ADO+Agregar nos muestra las diferentes formas de añadir nuevos registros a la tabla empleando un Recordset. Si el lector programa con DAO, observará que puede seguir utilizando los objetos de la misma manera, pero ahora disponemos de la ventaja de agregar el registro en una sola instrucción, pasando a AddNew() un array con los campos a asignar y otro con los valores.

```
Private Sub mnuADAgregar_Click()
Dim aCampos As Variant
On Error GoTo ControlError
' crear un array con los nombres de campos
aCampos = Array("CodCli", "Nombre", "Domicilio", _
   "Codigo Postal", "TfoFax")
' agregar registro usando un array con los
' nombres de los campos y otro con los valores
' a asignar
garsClientes.AddNew Array("CodCli", "Nombre", "Domicilio", _
   "Codigo Postal", "TfoFax"), _
   Array("55", "Elena", "Bosque, 10", "22445", "5554477")
' agregar registro usando el array con los nombres
' de campos creado anteriormente y otro array con
' los valores de los campos
garsClientes.AddNew aCampos, _
   Array("22", "Antonio", "Rosal, 42", "33445", "3335411")
' agregar registro y al mismo tiempo valor
' para un sólo campo
garsClientes.AddNew "CodCli", "92"
' agregar registro ejecutando primero AddNew()
' y asignando después los valores
garsClientes.AddNew
garsClientes("CodCli") = "21"
garsClientes("Nombre") = "Verónica"
garsClientes("Domicilio") = "Almendro, 6"
garsClientes("Codigo Postal") = "33124"
garsClientes("TfoFax") = "4345566"
garsClientes.Update
garsClientes.MoveFirst
Exit Sub
ControlError:
MsgBox Err.Description, , "Error"
Err.Clear
End Sub
```

Código Fuente 214

En el código fuente 214 se ha definido también un controlador de errores, para que puede comprobar el lector, el error generado en el caso de que intentemos ejecutar dos veces esta misma opción, ya que la tabla Clientes no permite claves duplicadas.

Agregar con Connection

Mediante la instrucción INSERT INTO de SQL, la opción de menú SQL+Agregar nos muestra como podemos añadir un nuevo registro a la tabla.

```
Private Sub mnuSQAgregar_Click()
Dim lcSQL As String
On Error GoTo ControlError
' crear instrucción de agregado
lcSQL = "INSERT INTO Clientes "
lcSQL = lcSQL & "(CodCli, Nombre, Domicilio, [Codigo Postal], TfoFax) "
lcSQL = lcSQL & "VALUES ('60', 'David', 'Mayor, 20', '28012', '5552133')"
' ejecutar instrucción
' desde el objeto conexión
gacnDatos.Execute lcSQL
garsClientes.Requery
Exit Sub
ControlError:
MsgBox Err.Description, , "Error"
Err.Clear
End Sub
```

Código fuente 215

No es un modo tan flexible como el que hemos visto con el objeto Recordset, pero habrá ocasiones en que también nos será de utilidad.

En las opciones de esta aplicación que utilizan sentencias SQL, ejecutamos finalmente el método Requery() del Recordset, de modo que el DataGrid del formulario refleje la información más actualizada de la tabla.

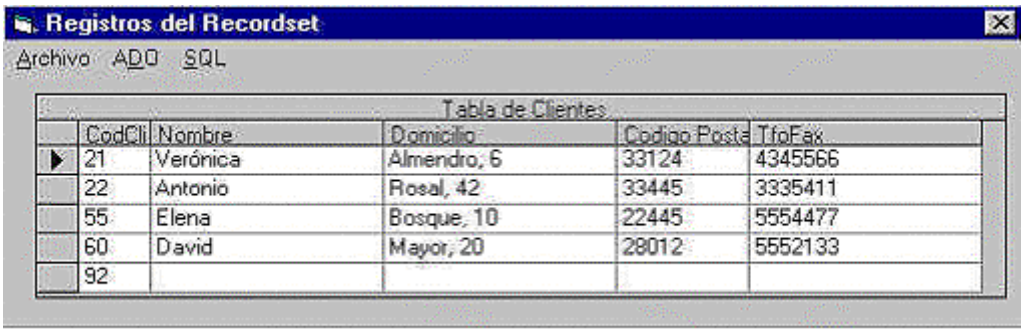


Figura 209. Registros agregados a la tabla Clientes

Modificar con Recordset

Para modificar los datos de un registro emplearemos el método Update(). Al igual que hacíamos para agregar, es posible utilizar arrays con los nombres de campo y valores a modificar, como podemos ver en la opción de menú ADO+Modificar del programa de ejemplo.

```
Private Sub mnuADModificar_Click()
' buscar registro a modificar
garsClientes.MoveFirst
garsClientes.Find "CodCli = '92'"
' si lo encontramos, modificar
If Not garsClientes.EOF Then
   ' modificar empleando un array con los campos a modificar
   ' y otro array con los nuevos valores para los campos
   garsClientes.Update Array("Nombre", "Codigo Postal"), _
      Array("Manuel", "12443")
End If
' buscar registro a modificar
garsClientes.MoveFirst
garsClientes.Find "CodCli = '22'"
' si lo encontramos, modificar
If Not garsClientes.EOF Then
   ' modificar campo a campo y llamar
   ' después al método Update()
   garsClientes("Domicilio") = "Everest, 8"
   garsClientes("Codigo Postal") = "41221"
   garsClientes("TfoFax") = "7778899"
   garsClientes.Update
End If
' buscar registro a modificar
garsClientes.MoveFirst
garsClientes.Find "CodCli = '92'"
' si lo encontramos, modificar
If Not garsClientes.EOF Then
   ' modificar un único campo
   ' usando el método Update()
   garsClientes.Update "Domicilio", "Valle,39"
End If
End Sub
```

Código fuente 216

Modificar con Connection

La instrucción UPDATE de SQL será la empleada en la opción de menú SQL+Modificar para cambiar algunos valores en los registros de la tabla. La ventaja en este caso del uso de SQL reside en que usando una sola instrucción podemos modificar en varios registros simultáneamente, lo vemos en el código fuente 217.

```
Private Sub mnuSQModificar_Click()
Dim lcSQL As String
' crear instrucción de modificación
lcSQL = "UPDATE Clientes "
lcSQL = lcSQL & "SET Nombre = 'Cambiado', TfoFax = '2221133' "
lcSQL = lcSQL & "WHERE TfoFax > '4000000'"
' ejecutar instrucción
' desde el objeto conexión
```

```
gacnDatos.Execute lcSQL
garsClientes.Requery
End Sub
```

Código fuente 217

La figura 210 el resultado de ejecutar esta opción.

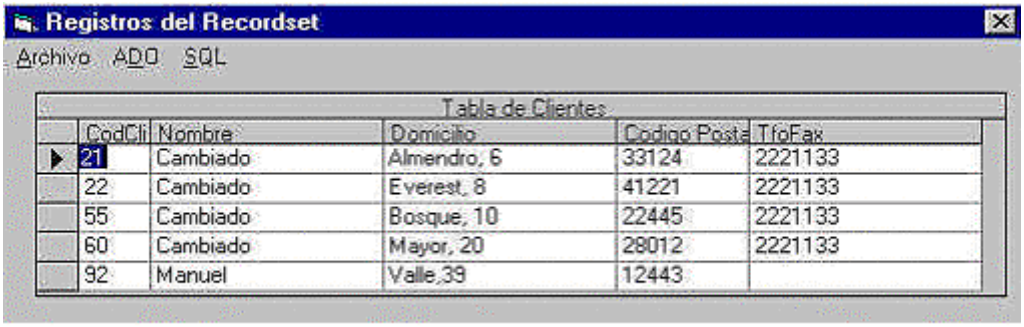


Figura 210. Registros resultantes después de modificarlos con una instrucción SQL.

Borrar con Recordset

El método Delete() del objeto Recordset elimina el registro actual o bien si se ha definido un filtro, podemos eliminar los registros que componen dicho filtro, pasando como parámetro la constante adecuada, que puede consultar el lector en el apartado dedicado a la definición de los objetos ADO en este mismo tema.

En este caso, la opción ADO+Borrar del formulario, buscará un registro y una vez posicionado en él lo eliminará, como muestra el código fuente 218.

```
Private Sub mnuADBorrar_Click()
' buscar registro
garsClientes.MoveFirst
garsClientes.Find "CodCli = '22'"
' borrarlo
garsClientes.Delete
End Sub
```

Código fuente 218

Si quisiéramos borrar un conjunto de registros, deberíamos establecer primero el filtro y luego borrarlos como indica el código fuente 219:

```
' buscar registro
garsRecordset.Filter = "CodCli > '20' AND CodCli < '80' "
' borrarlo
garsClientes.Delete adAffectGroup
```

Código Fuente 219

Borrar con Connection

La instrucción DELETE de SQL es la encargada de borrar registros. La ventaja con respecto al método de igual nombre en ADO es que en la misma instrucción SQL podemos indicar el rango de registros a borrar, como muestra la opción SQL+Borrar del formulario de ejemplo.

```
Private Sub mnuSQBorrar_Click()
Dim lcSQL As String
' crear instrucción de borrado
lcSQL = "DELETE * FROM Clientes WHERE CodCli > '55'"
' ejecutar instrucción
' desde el objeto conexión
gacnDatos.Execute lcSQL
garsClientes.Requery
End Sub
```

Código Fuente 220

Y hasta aquí hemos repasado las operaciones más comunes de edición de registros utilizando objetos ADO. En función de los requerimientos de cada desarrollo, el lector podrá decantarse por utilizar sólo propiedades y métodos o combinar instrucciones SQL.

Transacciones

En el ejemplo Transaccion, dedicado a estas operaciones, vamos a abrir una tabla, poner en marcha una transacción y realizar un borrado o modificación de registros. Pero antes de guardar definitivamente los cambios, pediremos confirmación de la operación, con lo que podremos dejar la base de datos tal y como estaba antes de empezar la transacción. El código fuente 221 corresponde al procedimiento Main() de inicio de este programa y al evento de carga del formulario en el que visualizaremos los resultados de las transacciones.

```
Public Sub Main()
Dim lfrmTransaccion As frmTransaccion
' llamar a la función que busca la ruta de
' los datos en el registro
gcRutaDatos = RutaDatosRegistro("Transaccion", "ValoresApp", _
    "Ubicacion", "transac.mdb")
' si no existe la ruta en el registro, salir
If Len(gcRutaDatos) = 0 Then
   MsgBox "No existen o son incorrectos los datos del Registro", , "Error"
   Exit Sub
End If
' abrir formulario
Set lfrmTransaccion = New frmTransaccion
Load lfrmTransaccion
lfrmTransaccion.Show
End Sub
'-----------------------------------
Private Sub Form_Load()
' crear conexión
Set macnTransac = New ADODB.Connection
' establecer cadena de conexión
macnTransac.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "transac.mdb"
```

```
' abrir conexión
macnTransac.Open
' crear recordset
Set marsProvincias = New ADODB.Recordset
' abrir recordset
marsProvincias.Open "Provincias", macnTransac, adOpenStatic, adLockOptimistic, _
   adCmdTable
' mostrar datos en el DataGrid
Set Me.grdProvincias.DataSource = marsProvincias
End Sub
```

Código fuente 221

Los objetos Connection y Recordset empleados para el manejo de los datos, se han definido a nivel de módulo en el formulario para estar accesibles desde cualquier procedimiento del mismo.

Figura 211. Ventana para realizar transacciones.

Al pulsar el botón *Borrar registros*, se ejecuta su método Click(), que contiene el código fuente 222.

```
Private Sub cmdBorrar_Click()
' borrar registros dentro de una transacción
Dim lnInd As Integer
Dim lnRespuesta As Integer
' iniciar transacción
macnTransac.BeginTrans
' borrar varios registros
marsProvincias.MoveFirst
For lnInd = 1 To 10
   marsProvincias.Delete
   marsProvincias.MoveNext
Next
' preguntar al usuario para
' confirmar borrado
lnRespuesta = MsgBox("¿Completar el borrado de registros?", _
   vbDefaultButton2 + vbYesNo, "Atención")
' según responda el usuario...
If lnRespuesta = vbYes Then
   ' confirmar el borrado de registros
   ' finalizando la transacción
   macnTransac.CommitTrans
```

```
Else
   ' deshacer el borrado dejando
   ' la tabla como estaba originalmente
   macnTransac.RollbackTrans
End If
' recuperar de nuevo los datos
' de la tabla para actualizar el DataGrid
marsProvincias.Requery
marsProvincias.MoveFirst
End Sub
```

Código fuente 222

Se inicia una transacción por medio del método BeginTrans del objeto Connection, seguida del borrado de una serie de registros. En la figura 212 se pide confirmación al usuario, y si es aceptada, se finaliza la transacción con CommitTrans; en el caso de que sea rechazada, se deja todo como estaba con RollbackTrans.

Figura 212. Transacción para el borrado de registros.

El trabajo del botón Modificar registros es similar al anterior. En este caso, modificamos el valor de varios registros y pedimos confirmación para grabar la modificación o para cancelarla.

```
Private Sub cmdModificar_Click()
' modificar registros dentro de una transacción
Dim lnInd As Integer
Dim lnRespuesta As Integer
' iniciar transacción
macnTransac.BeginTrans
' modificar varios registros
marsProvincias.MoveFirst
For lnInd = 1 To 10
   marsProvincias.Update "DsProv", "modificado"
```

```
   marsProvincias.MoveNext
Next
' preguntar al usuario para
' confirmar modificación
lnRespuesta = MsgBox("¿Completar la modificación de registros?", _
   vbDefaultButton2 + vbYesNo, "Atención")
' según responda el usuario...
If lnRespuesta = vbYes Then
   ' confirmar la modificación de registros
   ' finalizando la transacción
   macnTransac.CommitTrans
Else
   ' deshacer la modificación dejando
   ' la tabla como estaba originalmente
   macnTransac.RollbackTrans
End If
' recuperar de nuevo los datos
' de la tabla para actualizar el DataGrid
marsProvincias.Requery
marsProvincias.MoveFirst
End Sub
```

Código fuente 223

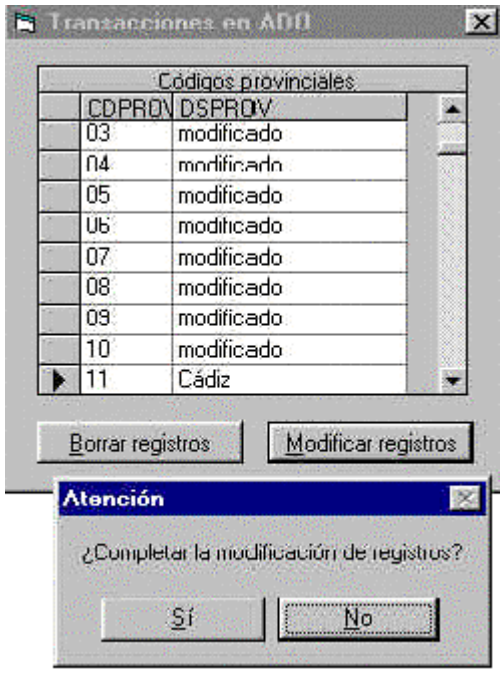


Figura 213. Transacción para la modificación de registros.

Los ejemplos mostrados aquí se basan en transacciones simples, pero es posible realizar transacciones anidadas, de forma que podamos tener al mismo tiempo varias transacciones y confirmar las operaciones realizadas por sólo algunas o todas. Para facilitar la labor de control de este tipo de transacciones, el método BeginTrans() devuelve un número que podemos recoger en el código de nuestra aplicación, correspondiente al nivel de anidamiento de la transacción que se acaba de poner en marcha. Es posible igualmente, configurar el objeto Connection de forma que inicie automáticamente las transacciones.

Asignando a la propiedad Attributes de dicho objeto la constante adXactCommitRetaining, se iniciará una nueva transacción después de realizar una llamada al método CommitTrans(). En el caso de que la

constante empleada sea adXactAbortRetaining, la transacción se iniciará automáticamente después de llamar a RollbackTrans().

Debido a que no todos los proveedores de datos admiten transacciones, es muy recomendable comprobar la propiedad Transaction DDL del proveedor, a través de la colección Properties del objeto Connection que vamos a utilizar para las transacciones.

Conectar mediante un origen de datos ODBC

Supongamos que debemos desarrollar una aplicación para manejar un formato de datos del que no existe un proveedor específico de OLE DB, por ejemplo ficheros .DBF. Si el lector ha programado con versiones anteriores de VB recordará que utilizando objetos DAO disponíamos de un modo de acceder a este y otros tipos de ficheros de datos que no eran los propios de Access, gracias a que DAO implementaba las características necesarias para poder acceder a tales bases de datos.

Sin embargo, esto no debe hacernos desistir en nuestra idea de migrar a ADO como modelo de objetos para desarrollar aplicaciones de datos, ya que OLE DB incorpora un proveedor para ODBC, a través del cuál disponemos de acceso a un variado número de bases de datos.

Siguiendo con la idea antes apuntada de acceder a ficheros dBase, la aplicación de ejemplo para este apartado, DBFODBC, nos guiará tanto en la configuración del controlador de ODBC para este tipo de datos como en el código a escribir en la aplicación para conectar con dicho controlador.

Esta aplicación contiene un formulario principal MDI llamado mdiInicio y varios formularios hijos en los que se mostrarán los resultados de las diferentes conexiones mediante ODBC a la base de datos Articulos.DBF, incluida también junto al proyecto.

El acceso a datos desde ADO utilizando el proveedor de OLE DB para ODBC puede realizarse de las formas descritas a continuación.

Nombre de fuente de datos o DSN de usuario (Data Source Name)

Un nombre de fuente de datos o DSN de usuario consiste en una conexión a una base de datos creada desde el administrador de ODBC, que contiene toda la información necesaria sobre el controlador de datos, base de datos, ubicación, y demás parámetros que variarán según sea el gestor de datos al que nos conectemos.

Lo que debemos hacer en primer lugar es crear el DSN, para ello abriremos el Panel de Control de Windows y ejecutaremos ODBC (32 bits), situándonos en la pestaña DSN de usuario y pulsando Agregar para crear un nuevo origen de datos.

Figura 214. Icono de la aplicación ODBC (32 bits)

Seguiremos seleccionando el controlador de base de datos a utilizar, en este caso será el correspondiente a dBase.

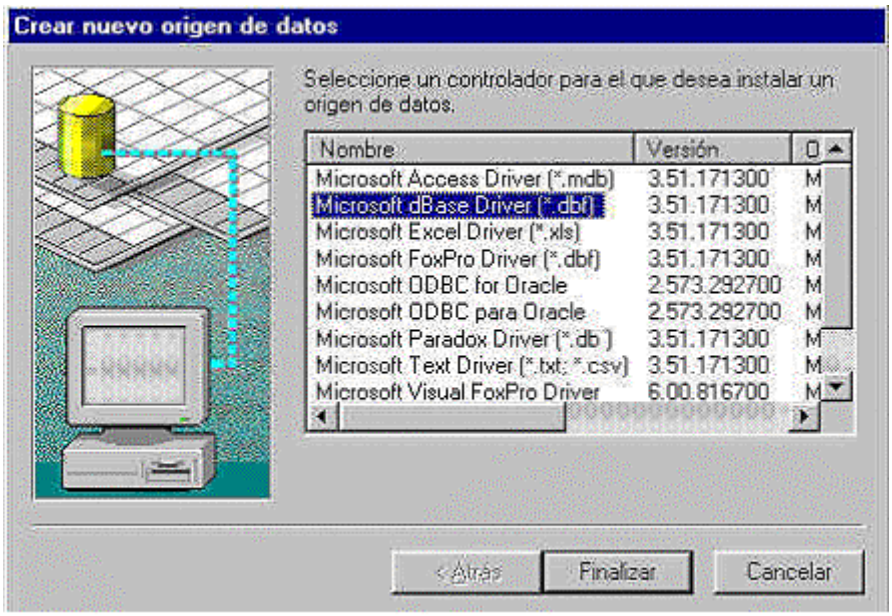


Figura 215. Selección del controlador de base de datos.

Pulsando el botón Finalizar aparecerá la ventana de instalación del controlador de dBase, en la que asignaremos un nombre para el DSN, elegiremos la versión correcta de dBase, y seleccionaremos el directorio en donde residen los ficheros .DBF a manejar. Sólo seleccionaremos el directorio y no el fichero; esto es así porque al ser los ficheros dBase la representación de una tabla, el directorio en donde están contenidos representa a la base de datos. La figura 216 muestra la configuración de la ventana para este ejemplo.

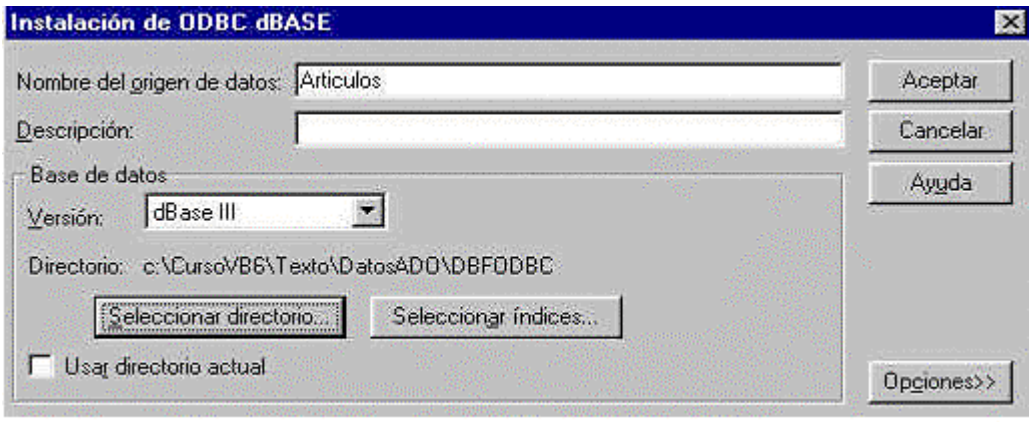


Figura 216. Valores para el DSN.

Al pulsar Aceptar, en la pestaña DSN de usuario del administrador de ODBC aparecerá el nuevo origen de datos creado.

Figura 217. Administrador de ODBC mostrando los orígenes de datos disponibles.

Finalizado el proceso de creación del DSN, pasaremos a la aplicación VB desde la que deberemos efectuar la conexión hacia el DSN recién creado. Esta labor podemos realizarla mediante un asistente en el caso de que utilicemos un control ADO Data para obtener la información, o bien desde el código de la aplicación.

- Conectar con un DSN de usuario desde el control ADO Data. Crearemos un formulario con el nombre frmDSN, en el que insertaremos un control ADO Data y abriremos su menú contextual seleccionando la opción Propiedades de ADODC, que mostrará la ventana de páginas de propiedades correspondiente a este control. En la pestaña General seleccionaremos el OptionButton Usar nombre de origen de datos ODBC y desplegando la lista asociada, elegiremos el nombre del DSN que acabamos de crear.

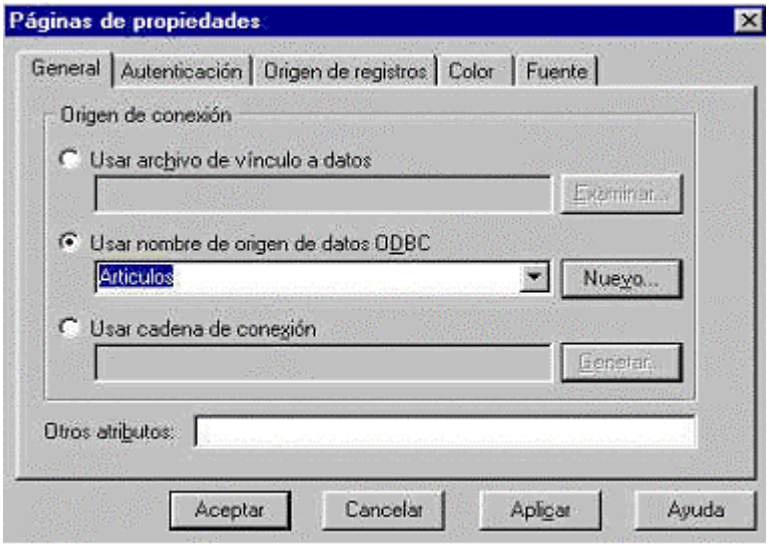


Figura 218. Selección del DSN en las propiedades del control ADO Data.

También es posible crear desde aquí un nuevo DSN si pulsamos el botón Nuevo de esta ventana, de forma que no necesitemos iniciar el administrador de ODBC, realizando todo el trabajo desde este asistente.

A continuación pasaremos a la pestaña Origen de registros, seleccionando adCmdTable de la lista de tipos de comando y el nombre de nuestro fichero de datos en la lista correspondiente a las tablas disponibles.

Daremos a este control el nombre adcArticulos y añadiremos un DataGrid, asignando a su propiedad DataSource el nombre del ADO Data que acabamos de crear, de forma que visualicemos los registros obtenidos del fichero de datos. La opción Conectar+DSN de usuario y control ADO Data del menú de mdiInicio abre este formulario, mostrando los registros.

Figura 219. Acceso a un fichero .DBF empleando ODBC.

- Conectar con un DSN de usuario desde el código. En esta ocasión, en lugar de utilizar los asistentes visuales, escribiremos el código necesario en el evento Load() del formulario frmDSNCodigo que va a mostrar los datos. El lugar principal en donde debemos fijar nuestra atención es en la propiedad ConnectionString del objeto Connection, que contiene el nombre del proveedor y el DSN.

```
Option Explicit
Private macnArticulos As ADODB.Connection
Private marsArticulos As ADODB.Recordset
' ------------------------------------------------------------
Private Sub Form_Load()
' crear objeto Connection
Set macnArticulos = New ADODB.Connection
macnArticulos.ConnectionString = "Provider=MSDASQL.1;DSN=Articulos"
macnArticulos.Open
' crear objeto Recordset
Set marsArticulos = New ADODB.Recordset
marsArticulos.Open "Articulos", macnArticulos, adOpenStatic, _
   adLockOptimistic, adCmdTable
' asignar el Recordset al DataGrid
Set Me.grdArticulos.DataSource = marsArticulos
End Sub
```

Código fuente 224

La opción de menú del formulario mdiInicio que abre esta ventaja hija es Conectar+DSN de usuario y código, y el resultado es el mismo que el de la figura 219, excepto que en esta ocasión no utilizamos un control ADO Data ya que asignamos directamente el Recordset creado en el código fuente 224 al DataGrid del formulario.

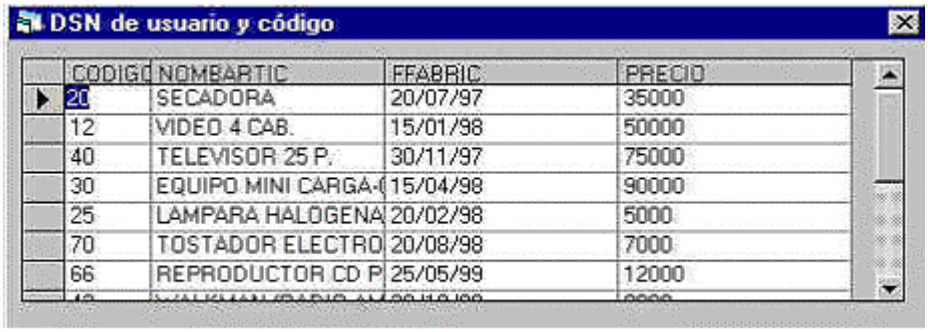


Figura 220. Apertura un DSN desde el código de la aplicación.

Fichero .DSN con la información de conexión

En este caso creamos desde el administrador de ODBC un fichero con extensión .DSN que contendrá toda la información para la conexión.

Para crear este archivo volveremos a ejecutar ODBC (32bits), pulsando esta vez la pestaña DSN de archivo, como vemos en la figura 221, y una vez situado en ella pulsaremos el botón Agregar.

Figura221. Creación de DSN en fichero.

Aparecerá a continuación la ventana de creación de un origen de datos. Seleccionaremos el correspondiente a dBase, como muestra la figura 222 y pulsaremos en el botón Avanzado, que nos mostrará una ventana con una o más de las líneas que se van a incluir en el fichero.

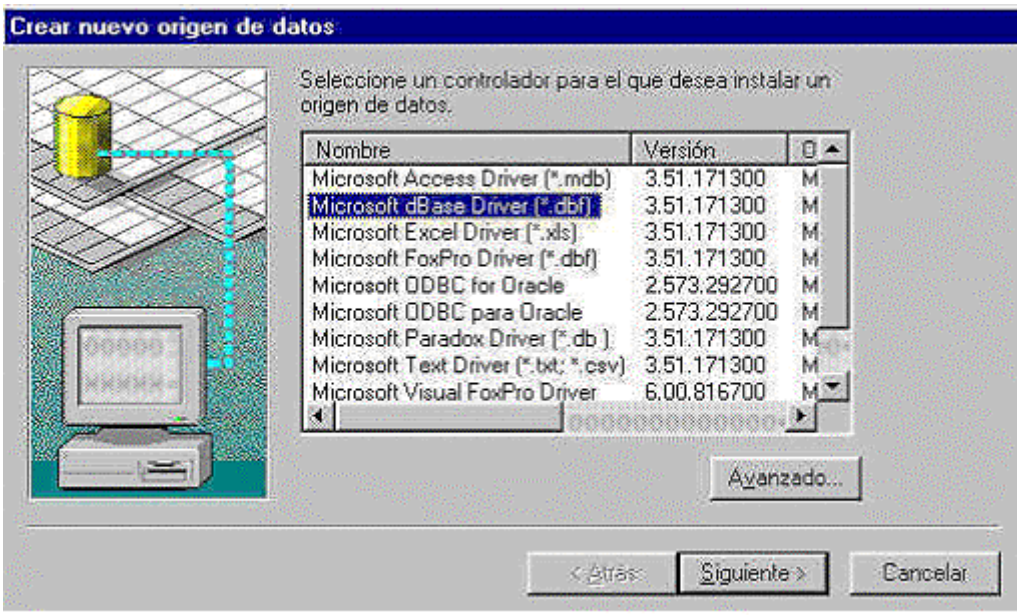


Figura 222. Creación de DSN en fichero.

Aquí y en función del tipo de base de datos, podremos añadir líneas para configurar la conexión. En este caso hemos agregado a la existente, una nueva con la ruta de la base de datos que el lector deberá de modificar en el caso de que no utilice la misma ruta por defecto de este proyecto.

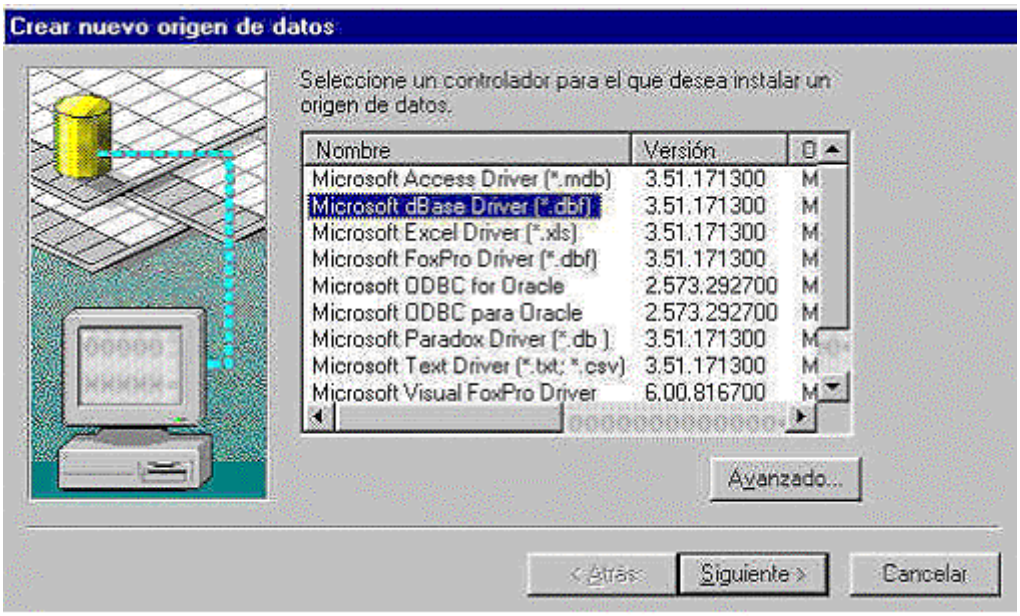


Figura 223. Información adicional para el fichero DSN.

Pulsaremos Aceptar volviendo a la ventana de creación del origen de datos, en la que al pulsar Siguiente deberemos indicar la ubicación y nombre del fichero .DSN que se va a crear.

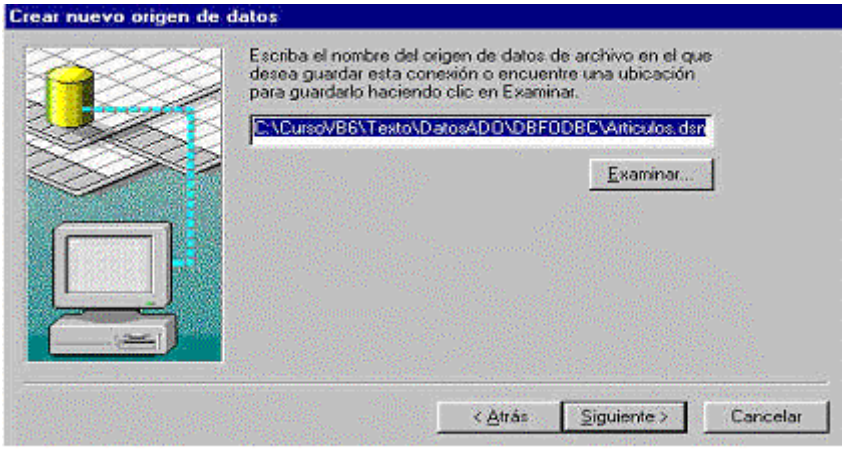


Figura 224. Ruta para crear el fichero DSN.

Al pulsar Siguiente se mostrará el contenido del fichero .DSN que está a punto de crearse. Pulsando el botón Finalizar, se finalizará la recopilación de datos para el fichero y aparecerá una ventana en la que podemos indicar algunos valores adicionales como por ejemplo la versión de dBase a emplear.

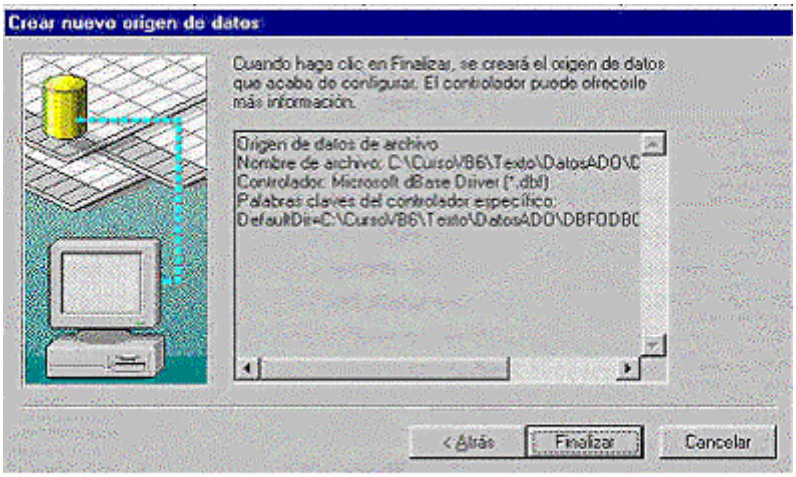


Figura 225. Paso previo a la creación del DSN.

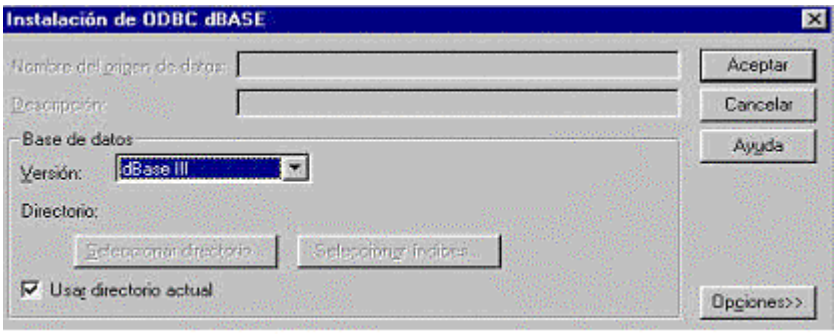


Figura 226. Configuración de la conexión a dBase.

Tras la anterior ventana se creará el fichero .DSN que emplearemos en la aplicación para conectar con la base de datos. Si vamos a utilizar un control ADO Data para recuperar la información, en su página de propiedades, pestaña General, debemos de seleccionar el OptionButton Usar archivo de vínculo a datos, escribiendo en el TextBox de este apartado la ruta y nombre del fichero DSN, o pulsar el botón Examinar para buscar el archivo. A partir de aquí, todo será igual que en el anterior ejemplo utilizando el control ADO Data.

Por ese motivo, lo que realmente nos interesa aquí es ver como se realiza la conexión mediante código. La opción Conectar+Fichero DSN del menú de frmInicio cargará el formulario frmDSNFichero, cuyo evento Load() realizará la conexión utilizando el fichero .DSN que hemos creado, como vemos en el código fuente 225.

```
Option Explicit
Private macnArticulos As ADODB.Connection
Private marsArticulos As ADODB.Recordset
'---------------------------------------
Private Sub Form_Load()
' crear objeto Connection
Set macnArticulos = New ADODB.Connection
macnArticulos.ConnectionString = "File Name=C:\CursoVB6\Texto\DatosADO\DBFODBC\" &
_
    "Articulos.DSN"
macnArticulos.Open
' crear objeto Recordset
Set marsArticulos = New ADODB.Recordset
marsArticulos.Open "Articulos", macnArticulos, adOpenStatic, _
   adLockOptimistic, adCmdTable
' asignar el Recordset al DataGrid
Set Me.grdArticulos.DataSource = marsArticulos
End Sub
```

Código fuente 225

Conectar sin DSN, sólo con código

Finalmente disponemos de la posibilidad de establecer una conexión mediante ODBC escribiendo el código de conexión directamente en la aplicación VB, sin utilizar un DSN previamente creado. Para ello seleccionaremos la opción de menú Conectar+Sólo código del formulario principal, que cargará la ventana frmSoloCodigo en cuyo evento Load() se establecerá este tipo de conexión.

```
Option Explicit
Private macnArticulos As ADODB.Connection
Private marsArticulos As ADODB.Recordset
'---------------------------------------
Private Sub Form_Load()
' crear objeto Connection
Set macnArticulos = New ADODB.Connection
macnArticulos.ConnectionString = "Provider=MSDASQL.1;" & _
   "Driver={Microsoft dBase Driver (*.dbf)};" & _
   "DefaultDir=C:\CursoVB6\Texto\DatosADO\DBFODBC;"
macnArticulos.Open
' crear objeto Recordset
Set marsArticulos = New ADODB.Recordset
marsArticulos.Open "Articulos", macnArticulos, adOpenStatic, _
   adLockOptimistic, adCmdTable
' asignar el Recordset al DataGrid
```

```
Set Me.grdArticulos.DataSource = marsArticulos
End Sub
```

Código fuente 226

Ejecutar un procedimiento almacenado

Gracias a ADO, es posible acceder a elementos de una base de datos, tales como: diagramas, vistas, tablas, procedimientos almacenados, etc. En estos últimos, los procedimientos almacenados, vamos a centrar la atención de este apartado.

El ejemplo contenido en la aplicación ProcAlm, nos muestra como conectar con una base de datos de gran envergadura como SQL Server y ejecutar uno de los procedimientos almacenados que contiene, con la ventaja de que la sencillez del código a emplear, es la misma que si tuviéramos que manipular datos de Access. También es cierto que el procedimiento almacenado que emplearemos en esta prueba es muy básico.

El ejemplo en cuestión, consta de un formulario que contiene un control ADO Data y un DataGrid, en los que únicamente se ha establecido su propiedad Name. Donde realmente reside el trabajo de conexión y apertura del procedimiento almacenado es en el evento Load() del formulario, la figura 227 muestra la instrucción SQL que contiene el procedimiento.

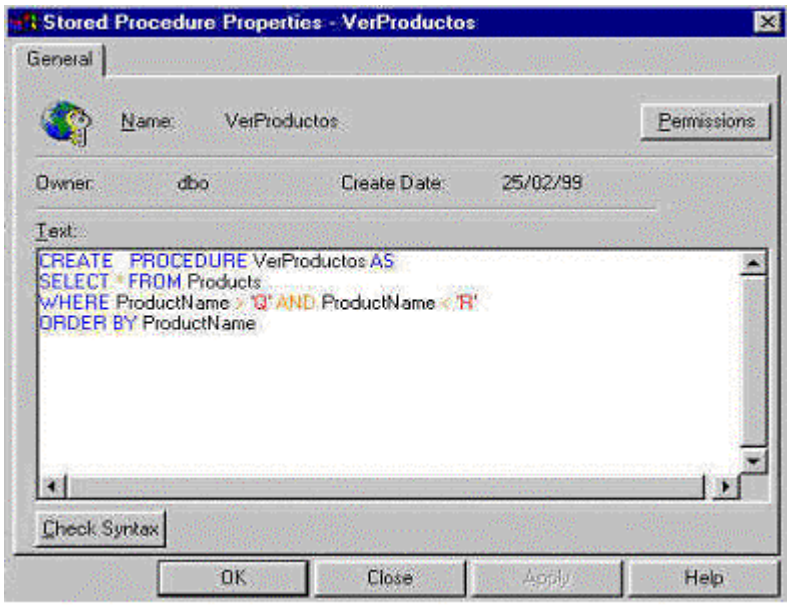


Figura 227. Instrucción SQL del procedimiento almacenado a ejecutar.

El código fuente 227 corresponde al evento Load() del formulario.

```
Private Sub Form_Load()
' establecer propiedades para el ADO Data
Me.adcVerProductos.UserName = "sa"
Me.adcVerProductos.ConnectionString = "Provider=SQLOLEDB.1;Persist Security
 Info=False;User ID=sa;Initial Catalog=Northwind;Data Source=LMBLANCO"
Me.adcVerProductos.CommandType = adCmdStoredProc
```

```
Me.adcVerProductos.RecordSource = "VerProductos"
' conectar con la base de datos Northwind
' y ejecutar el procedimiento almacenado
Me.adcVerProductos.Refresh
' visualizar el resultado en el grid
Set Me.grdVerProductos.DataSource = Me.adcVerProductos
End Sub
```

Código fuente 227

Finalmente, ejecutaremos la aplicación, que nos mostrará el formulario con las filas resultantes del procedimiento (figura 228).

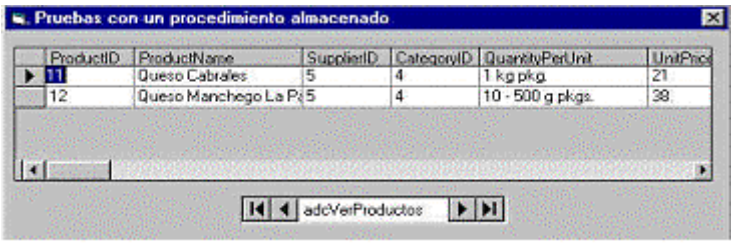


Figura 228. Aplicación ProcAlm en ejecución.

Esquemas

En términos ADO, un esquema no es otra cosa que la información interna de una fuente de datos, como puedan ser los nombre de tablas, campos, etc.

Si trabajando con objetos ADO, necesitamos durante el desarrollo de un programa, conocer cual es el esquema de una base de datos a la que nos hemos conectado, emplearemos el método OpenSchema() del objeto Connection, que nos devolverá un Recordset con este tipo de información. Debido a que ADO se comunica con OLE DB y este último es el encargado de realizar el trabajo pesado contra la fuente de datos, no importa cual sea el tipo de base de datos al que nos conectamos, la forma de obtener el esquema será igual para cualquier proveedor, ya que OLE DB se ocupa de traducir esta información y ponerla a disposición de ADO.

Para ejemplificar este tipo de operaciones, el lector dispone de la aplicación contenida en Esquema, que consta de un formulario, en el cual, al pulsar un CommandButton, crea una conexión, y a partir de esta, abre el esquema de la base de datos, rellenando con esta información un ListBox que también está situado en dicho formulario, tal como se puede ver en la figura 229.

Figura 229. Visualización del esquema de una base de datos.

En el código fuente 228 podemos ver el código del botón que realiza este trabajo.

```
Private Sub cmdVisualizar_Click()
Dim lacnConexion As ADODB.Connection
Dim larsEsquema As ADODB.Recordset
Dim lcTablaEnProceso As String
' instanciar conexion
Set lacnConexion = New ADODB.Connection
' establecer cadena de conexión
lacnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "musica.mdb"
' abrir conexión
lacnConexion.Open
' instanciar recordset
Set larsEsquema = New ADODB.Recordset
' abrir el esquema de la base de datos
' y pasarlo al recordset
Set larsEsquema = lacnConexion.OpenSchema(adSchemaColumns)
' llenar el listbox con la información
' de tablas y campos de la base de datos
Do While Not larsEsquema.EOF
   ' tomar nombre de la tabla
   If lcTablaEnProceso <> larsEsquema("TABLE_NAME") Then
      lcTablaEnProceso = larsEsquema("TABLE_NAME")
      Me.lstEsquema.AddItem lcTablaEnProceso
   End If

   ' tomar campos de la tabla
   Me.lstEsquema.AddItem Space(4) & larsEsquema("COLUMN_NAME")
   larsEsquema.MoveNext
Loop
' cerrar objetos
larsEsquema.Close
lacnConexion.Close
' liberar recursos
Set larsEsquema = Nothing
Set lacnConexion = Nothing
End Sub
```

Código fuente 228

La información obtenida a partir de OpenSchema() puede ser de muy diverso tipo, tal flexibilidad se debe a la constante pasada como parámetro en este método. Para una descripción detallada, consulte el lector, en el apartado ActiveX Data Objectos, el punto dedicado al objeto Connection.

Enlazar controles a datos en tiempo de ejecución

El enlace de controles a fuentes de datos en ADO, además de mantener todos los aspectos ya implementados en DAO, se refuerza mediante un conjunto de características que permiten un mayor nivel de control, incluso en tiempo de ejecución sobre fuentes de datos y controles enlazados a dichas fuentes.

Pongamos un ejemplo: cuando desarrollamos una aplicación con DAO, en un formulario que contenga un control Data y controles enlazados a este, el único medio de enlazar la información del Data a los controles, es en tiempo de diseño, mediante la ventana de propiedades de dichos controles.

ADO va más lejos y permite configurar un elevado número de aspectos de los controles de datos en tiempo de ejecución, de forma que podemos dotar a nuestras aplicaciones de una mayor versatilidad.

Para mostrar algunas de estas capacidades en la manipulación de datos y controles, se adjunta la aplicación EnlaceControl, que contiene dos ejemplos para recuperar información de una base de datos: utilizando un ADO Data Control enlazado a varios TextBox, que muestran el contenido de una tabla; y abriendo directamente la base de datos y conectando a controles sin necesidad de utilizar el ADO Data. Cada uno de estos ejemplos está contenido en un formulario distinto, a los que podemos acceder mediante el formulario mdiInicio, incluido en el programa.

Veamos en primer lugar, el formulario frmADOData: contiene un control ADO Data y dos TextBox que van a mostrar los registros de una tabla a la que se conectará el control de datos. Al ejecutar la aplicación, seleccionaremos la opción Archivo+ADO Data Control en la ventana mdiInicio, que nos mostrará este formulario. ¿Qué particularidad tiene esta ventana?, pues en primer lugar, toda la conexión del control ADO Data se va a realizar con código en tiempo de ejecución; y segundo, disponemos de dos OptionButton; en función del que esté pulsado, abriremos una base de datos distinta en cada caso. Al pulsar el botón cmdConectar, se ejecutará el código fuente 229, que conectará con la base de datos apropiada y mostrará los registros de una de sus tablas.

```
Private Sub cmdConectar_Click()
' si se ha marcado el OptionButton Cine,
' se conectará con esa base de datos y
' se mostrarán los datos de la tabla Actores
If Me.optCine.Value Then
   ' establecer cadena de conexión para
   ' el control ADO Data
   Me.adcDatos.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
      "Data Source=" & gcRutaDatos & "cine.mdb"

   ' indicar el modo en que se van a
   ' recuperar los datos y que tabla
   ' se va a consultar
   Me.adcDatos.CommandType = adCmdTable
   Me.adcDatos.RecordSource = "Actores"

   ' actualizar los datos del ADO Data
   Me.adcDatos.Refresh

   ' establecer en los controles TextBox
   ' la fuente de datos (el control ADO Data)
   ' campos a mostrar
   Me.txtCodigo.DataField = "CodActor"
   Set Me.txtCodigo.DataSource = Me.adcDatos

   Me.txtValor.DataField = "Actor"
   Set Me.txtValor.DataSource = Me.adcDatos

   ' asignar valor al Label que indica
   ' el campo de actor/autor
   Me.lblValor = "Actor"

Else
   ' si se ha marcado el OptionButton Música,
   ' se conectará con esa base de datos y
   ' se mostrarán los datos de la tabla Autores
   ' establecer cadena de conexión para
   ' el control ADO Data
   Me.adcDatos.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
      "Data Source=" & gcRutaDatos & "musica.mdb"

   ' indicar el modo en que se van a
```

```
    ' recuperar los datos y que tabla
    ' se va a consultar
    Me.adcDatos.CommandType = adCmdTable
    Me.adcDatos.RecordSource = "Autores"

    ' actualizar los datos del ADO Data
    Me.adcDatos.Refresh

    ' establecer en los controles TextBox
    ' la fuente de datos (el control ADO Data)
    ' campos a mostrar
    Me.txtCodigo.DataField = "CodAutor"
    Set Me.txtCodigo.DataSource = Me.adcDatos

    Me.txtValor.DataField = "Autor"
    Set Me.txtValor.DataSource = Me.adcDatos

    ' asignar valor al Label que indica
    ' el campo de actor/autor
    Me.lblValor = "Autor"

End If
End Sub
```

Código fuente 229

Es necesario resaltar en el código fuente 229 dos puntos importantes. Primero: después de haber establecido la información para las propiedades del control ADO, es necesario ejecutar su método Refresh(), para que actualice la información que guarda, estableciendo una conexión con la nueva base de datos. Segundo: el orden de asignación de las propiedades en los controles enlazados. Habrá notado el lector, que en primer lugar se asigna la propiedad DataField al TextBox, y después DataSource. Esto es muy importante, ya que si se hiciera al contrario, como en principio parece más lógico (establecer la fuente de datos y después el campo a obtener información), se produciría un error.

Al ejecutar el programa, este formulario presentará un aspecto como el de la figura 230

Figura 230. Formulario frmADOData en ejecución.

Como hemos podido comprobar, es posible asignar a la propiedad DataSource de un control enlazado, una fuente de datos (en este caso el ADO Data Control) en tiempo de ejecución, y establecer el campo que mostrará sobre la base de la consulta o tabla que se recuperará de dicho control de datos.

Empleando DAO, parte de este trabajo se debería haber realizado exclusivamente en la fase de diseño, con lo cual perdemos parte de la flexibilidad de que aquí disponemos.

Al otro ejemplo contenido en este programa se accede seleccionando la opción de menú Archivo+Sin control de datos de la ventana mdiInicio. Esta acción abrirá el formulario frmNoADOData, en el cual se establece también una conexión con datos y controles enlazados usando código, pero a diferencia del ejemplo anterior, aquí no se emplea el ADO Data Control y además, se muestran simultáneamente registros pertenecientes a tablas de bases de datos diferentes. La conexión se realiza en el evento Load() del formulario, lo vemos en el código fuente 230.

```
Private Sub Form_Load()

' variables para conectar y tomar datos
' de la base de datos Cine
Dim lacnCine As ADODB.Connection
Dim larsDirectores As ADODB.Recordset

' variables para conectar y tomar datos
' de la base de datos Musica
Dim lacnMusica As ADODB.Connection
Dim larsGrabaciones As ADODB.Recordset

' configurar conexión de datos para
' mostrar datos de la tabla Directores
' ------------------------------------
' crear y abrir conexión
Set lacnCine = New ADODB.Connection
lacnCine.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "cine.mdb"
lacnCine.Open

' crear y abrir el recordset
Set larsDirectores = New ADODB.Recordset
Set larsDirectores.ActiveConnection = lacnCine
larsDirectores.Source = "Directores"
larsDirectores.Open , , adOpenDynamic, adLockOptimistic, adCmdTable

' enlazar controles que mostrarán
' datos de la tabla Directores
Set Me.txtCodDirector.DataSource = larsDirectores
Me.txtCodDirector.DataField = "CodDirector"
Set Me.txtDirector.DataSource = larsDirectores
Me.txtDirector.DataField = "Director"

'*****************************************
' configurar conexión para mostrar información
' de la base de datos Musica
' ------------------------------------
' crear y abrir conexión
Set lacnMusica = New ADODB.Connection
lacnMusica.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "musica.mdb"
lacnMusica.Open

' crear y abrir el recordset,
' este recordset estará basado en una consulta SQL
' sobre más de una tabla
Set larsGrabaciones = New ADODB.Recordset
Set larsGrabaciones.ActiveConnection = lacnMusica
larsGrabaciones.Source = "SELECT Autores.Autor, Grabaciones.Titulo " & _
   "FROM Autores INNER JOIN Grabaciones " & _
   "ON Autores.CodAutor = Grabaciones.CodAutor " & _
   "ORDER BY Autores.Autor"
larsGrabaciones.Open , , adOpenDynamic, adLockOptimistic, adCmdText

' enlazar controles que mostrarán datos
```

```
Set Me.txtAutor.DataSource = larsGrabaciones
Me.txtAutor.DataField = "Autor"
Set Me.txtTitulo.DataSource = larsGrabaciones
Me.txtTitulo.DataField = "Titulo"
End Sub
```

Código fuente 230

No sólo atacamos a dos bases de datos diferentes al mismo tiempo, sino que mientras que en una de ellas, mostramos los registros en forma de tabla, en la otra se realiza una consulta SQL que recupera datos de dos tablas distintas. Así de potente es el trabajo con este tipo de objetos.

Es de destacar también, que sólo se emplean un objeto Connection y un Recordset, de esta forma se puede comprobar que no es necesario un objeto Command, sino que se pasa la conexión a una de las propiedades del Recordset, y este se encarga de la recuperación de registros. Estos mismos objetos, no se cierran con su método Clase(), como sería conveniente, al dejar de usarlos en el evento Load(), debido a que necesitarán estar activos para la navegación entre las filas de las tablas, como veremos seguidamente.

En este formulario, como la conexión sólo se establece una vez (al cargarlo), a diferencia del ejemplo anterior, no importa el orden en que asignemos las propiedades DataSource y DataField de los controles enlazados. La figura 231, muestra esta ventana en ejecución.

Figura 231. Formulario frmNoADOData en ejecución.

La navegación por los registros de los objetos Recordset se realiza mediante los botones situados en la parte inferior del formulario. Pero el lector probablemente se preguntará, si no se utilizara un control de datos, y los objetos ADO que se conectan con los datos en el evento Load() se han declarado en el ámbito local, ¿cómo conseguimos mover el Recordset al que están conectados los controles enlazados?. Pues precisamente, recuperando dicho Recordset de la propiedad DataSource en dichos controles TextBox. Como se muestra en el código fuente 231 correspondiente al botón que efectúa el avance al siguiente registro. Observe el lector, la forma de recuperar el Recordset y emplearlo para pasar a la siguiente fila.

```
Private Sub cmdAvanzar_Click()
' avanzar al siguiente registro de las tablas
Dim larsTablaDirectores As ADODB.Recordset
Dim larsTablaGrabaciones As ADODB.Recordset
' tomar el recordset de la tabla Directores de la
' propiedad correspondiente en uno de los TextBox
Set larsTablaDirectores = Me.txtDirector.DataSource
larsTablaDirectores.MoveNext
```

```
If larsTablaDirectores.EOF Then
   MsgBox "Último registro", , "Tabla Directores"
   larsTablaDirectores.MoveLast
End If
' tomar el recordset de la tabla Grabaciones de la
' propiedad correspondiente en uno de los TextBox
Set larsTablaGrabaciones = Me.txtTitulo.DataSource
larsTablaGrabaciones.MoveNext
If larsTablaGrabaciones.EOF Then
   MsgBox "Último registro", , "Tabla Grabaciones"
   larsTablaGrabaciones.MoveLast
End If
End Sub
```

Código fuente 231

El código del resto de botones de desplazamiento, se puede consultar cargando este proyecto en VB, ya que se basan en la misma técnica.

Enlazar controles de varios formularios a un mismo origen de datos

En versiones previas de VB, sólo estaba permitido el enlace de controles a fuentes de datos, dentro de un mismo formulario. VB 6 rompe con esta limitación, permitiéndonos enlazar controles en más de un formulario a un único origen de datos, gracias a las nuevas capacidades incorporadas de asignación de propiedades en tiempo de ejecución.

De esta forma, podremos desarrollar aplicaciones como las del ejemplo EnlazForms, que se acompaña, en el cual se crea una conexión hacia la base de datos Musica.MDB, ya conocida por el lector de anteriores ejemplos, y después de abrir su tabla Grabaciones con un Recordset, se asigna este a un formulario, que se encargará del desplazamiento de registros. Por otro lado, tendremos dos formularios con diversos TextBox enlazados al Recordset antes comentado, que se actualizarán desde el formulario que efectúa el movimiento por los registros.

Vayamos por partes, en primer lugar crearemos los formularios con los controles que se van a enlazar al Recordset, que tendrán los nombres frmGrabac1 y frmGrabac2, y podemos ver en las figuras 232 y 233.

Figura 232. Formulario frmGrabac1.

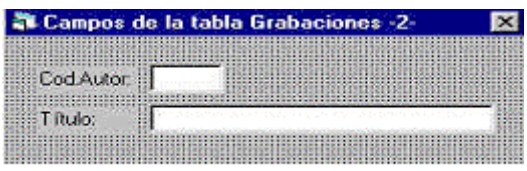


Figura 233. Formulario frmGrabac2.

Los TextBox en estos formularios, serán enlazados cada uno a un campo del Recordset.

En el siguiente paso, crearemos el formulario frmNavegar, que como su nombre indica, será el encargado del movimiento por los registros del Recordset que contiene la tabla a consultar.

Figura 234. Formulario frmNavegar.

En este formulario, se define a nivel de módulo, una variable que contendrá el Recordset de la tabla Grabaciones. Los CommandButton que incluye, serán los encargados de desplazarse al siguiente o anterior registro. El código fuente 232 , contiene parte del código de este formulario.

```
Option Explicit
Public marsGrabaciones As ADODB.Recordset
'---------------------------------------
Private Sub cmdAvanzar_Click()
' avanzar al siguiente registro de la tabla,
' los controles de los formularios conectados
' a este Recordset serán actualizados
marsGrabaciones.MoveNext
If marsGrabaciones.EOF Then
   marsGrabaciones.MoveLast
   MsgBox "Última fila de la tabla"
End If
End Sub
```

Código fuente 232

Y finalmente, llegamos al punto clave de este ejemplo, el formulario mdiInicio. Es aquí, a través de la opción Archivo+Enlazar formularios, donde se crean los objetos ADO, los formularios enlazados, y las conexiones entre controles y datos. Veamos el código fuente 233.

```
Private Sub mnuArEnlazar_Click()
Dim lfrmGrabac1 As frmGrabac1
Dim lfrmGrabac2 As frmGrabac2
Dim lfrmNavegar As frmNavegar
Dim lacnMusica As ADODB.Connection
Dim larsGrabaciones As ADODB.Recordset
' cargar formularios y mostrarlos
Set lfrmGrabac1 = New frmGrabac1
Set lfrmGrabac2 = New frmGrabac2
Set lfrmNavegar = New frmNavegar
Load lfrmGrabac1
Load lfrmGrabac2
Load lfrmNavegar
lfrmGrabac1.Show
lfrmGrabac2.Show
lfrmNavegar.Show
' crear y abrir conexión con la base de datos
Set lacnMusica = New ADODB.Connection
```

```
lacnMusica.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos & "musica.mdb"
lacnMusica.Open
' crear y abrir el Recordset, tabla Grabaciones
Set larsGrabaciones = New ADODB.Recordset
Set larsGrabaciones.ActiveConnection = lacnMusica
larsGrabaciones.Source = "Grabaciones"
larsGrabaciones.Open , , adOpenDynamic, adLockOptimistic, adCmdTable
' configurar los controles enlazados a datos de
' los formularios que hemos cargado y que van
' a mostrar los registros de la tabla
Set lfrmGrabac1.txtCodGrabacion.DataSource = larsGrabaciones
lfrmGrabac1.txtCodGrabacion.DataField = "CodGrabacion"
Set lfrmGrabac1.txtSoporte.DataSource = larsGrabaciones
lfrmGrabac1.txtSoporte.DataField = "Soporte"
Set lfrmGrabac2.txtCodAutor.DataSource = larsGrabaciones
lfrmGrabac2.txtCodAutor.DataField = "CodAutor"
Set lfrmGrabac2.txtTitulo.DataSource = larsGrabaciones
lfrmGrabac2.txtTitulo.DataField = "Titulo"
' asignar al formulario encargado de
' navegar por la tabla el Recordset
Set lfrmNavegar.marsGrabaciones = larsGrabaciones
End Sub
```

Código fuente 233

Con la ejecución de este código, obtendremos un resultado como el de la figura 235.

Figura 235. Enlace a datos de múltiples formularios.

Al ejecutar el lector este ejemplo, comprobará como al pulsar los botones Retroceder o Avanzar, se mueve el registro actual en la tabla, actualizando el valor de los controles en los otros formularios.

El modo de inicio de este programa es igual al de otros ejemplos de este tema, mediante un procedimiento Main() que busca la ruta de la base de datos en el Registro del sistema, por lo cual, no repetiremos de nuevo dicho código, ya que puede ser consultado por el lector desde el entorno de VB al cargar este proyecto.

El Entorno de Datos (Data Environment)

Esta utilidad, encuadrada dentro de lo que en VB se denominan diseñadores, nos permite crear visualmente los objetos para la manipulación de datos basados en OLE DB/ADO, más habituales en las aplicaciones: conexiones, comandos, procedimientos almacenados, campos calculados, agrupados, etc.

Con el fin de ilustrar estas y otras cualidades del Entorno de Datos, se acompaña la aplicación DataEnv, que contiene diversos ejemplos de uso, y que iremos viendo en los sucesivos puntos de este apartado. El formulario principal de este proyecto, mdiInicio, nos servirá para acceder a dichos ejemplos, a través de las opciones de su menú.

Agregar un Entorno de Datos

El primer paso a dar para trabajar con una utilidad de este tipo, es agregarla al proyecto. Podemos conseguirlo con la opción Proyecto+Más diseñadores ActiveX...+Data Environment del menú de VB, o bien desde el Explorador de proyectos, abriendo su menú contextual y seleccionando la opción Agregar+Más diseñadores ActiveX...+Data Environment, de manera que se incluirá este elemento en la aplicación, como vemos en la figura 236.

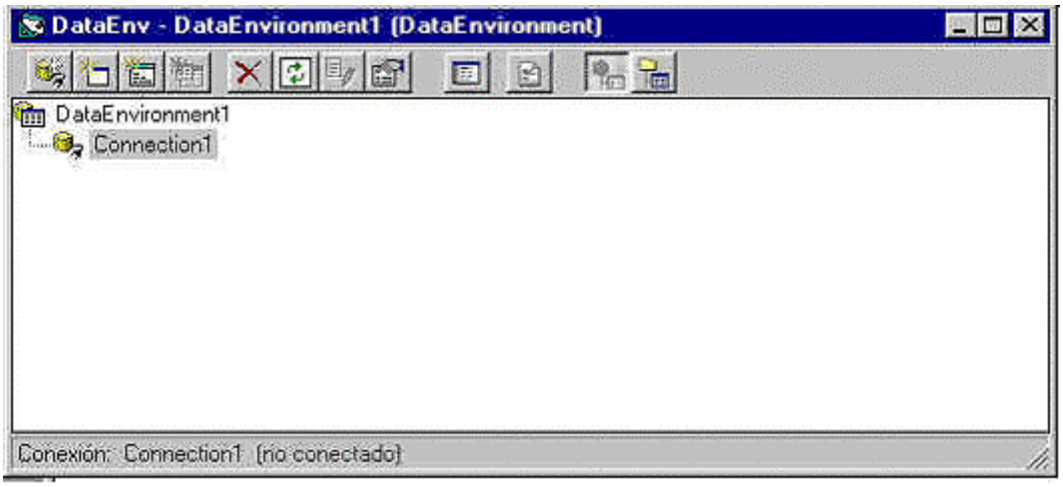


Figura 236. Ventana del Data Environment.

Es muy importante no confundir la ventana del diseñador del Entorno de Datos con el propio objeto Entorno de Datos. Al agregar un Entorno de Datos, la ventana que se incorpora al proyecto corresponde al diseñador, y el elemento que por defecto tiene el nombre DataEnvironment1 es el objeto que usaremos para establecer conexiones y ejecutar comandos. La ventana es un objeto contenedor de los objetos de datos.

Usaremos desde el Entorno de Datos, la ventana de propiedades estándar de VB, para asignar valores a las propiedades de los objetos contenidos en el diseñador. En este caso, cambiaremos el nombre que tiene por defecto el Entorno, por uno más adecuado al proyecto: dteEntorno.

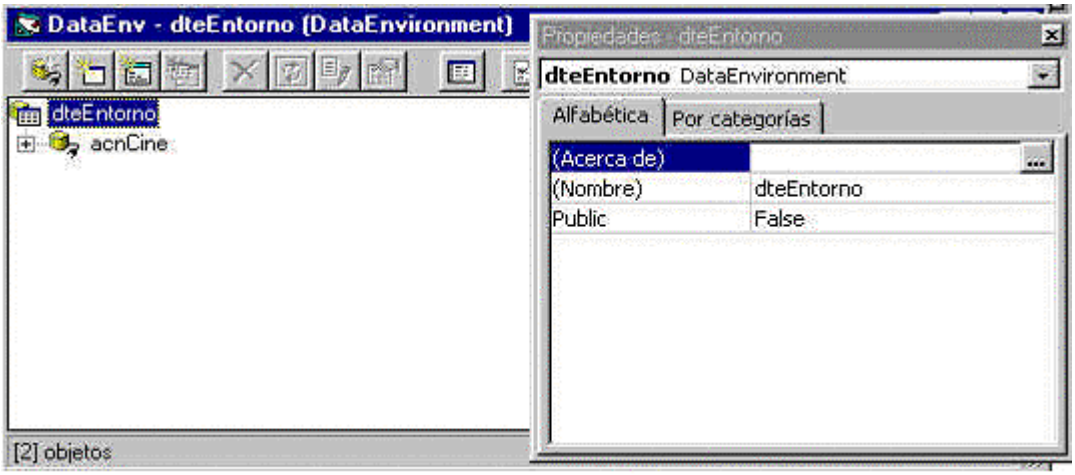


Figura 237. Propiedades del Entorno de Datos.

Crear una conexión

Creado el Entorno de Datos, el siguiente paso consiste en crear una conexión hacia una fuente de datos.

Mediante el botón Agregar conexión,  , de la barra de herramientas del Data Environment, o la opción del mismo nombre de su menú contextual, crearemos un objeto conexión, que configuraremos empleando la ventana de propiedades estándar o mediante un asistente similar al existente para el control ADO Data. Este asistente aparecerá al pulsar el botón Propiedades del Entorno de Datos.

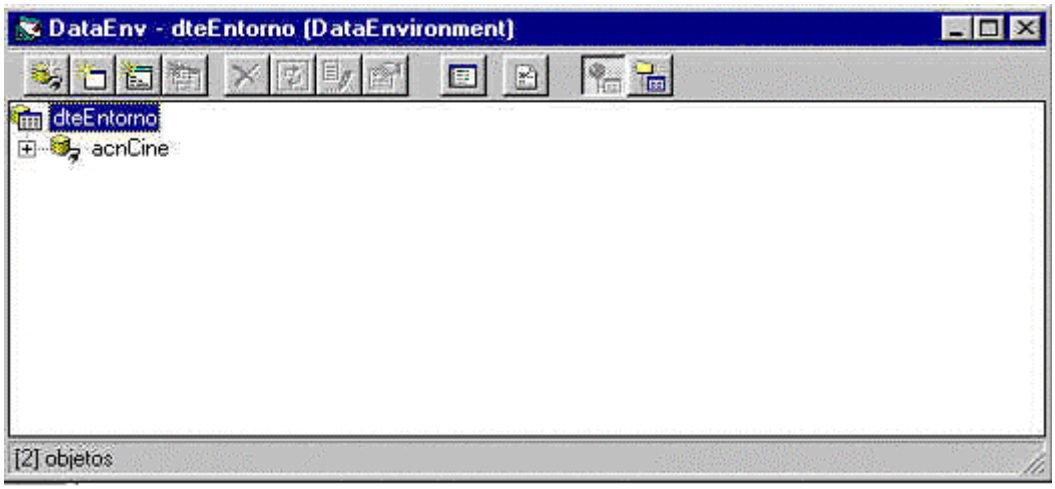


Figura 238. Conexión creada desde el Entorno de Datos.

Daremos a la conexión el nombre acnCine, ya que vamos a conectar con el fichero Cine.MDB, conocido por el lector de otros ejemplos. Las figuras 239 y 240 muestran los pasos del asistente para establecer las propiedades de conexión.

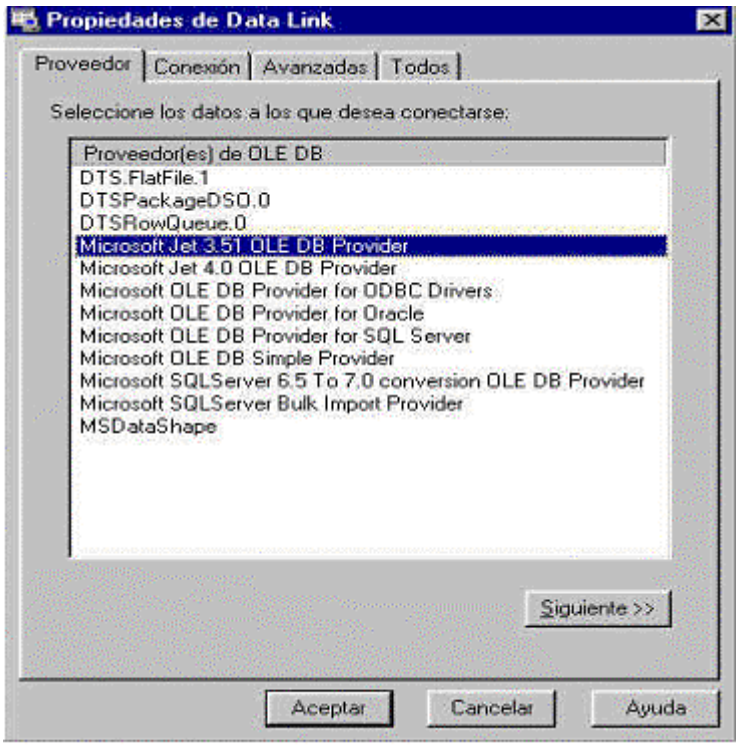


Figura 239. Propiedades para una conexión del Entorno; selección del proveedor de datos.

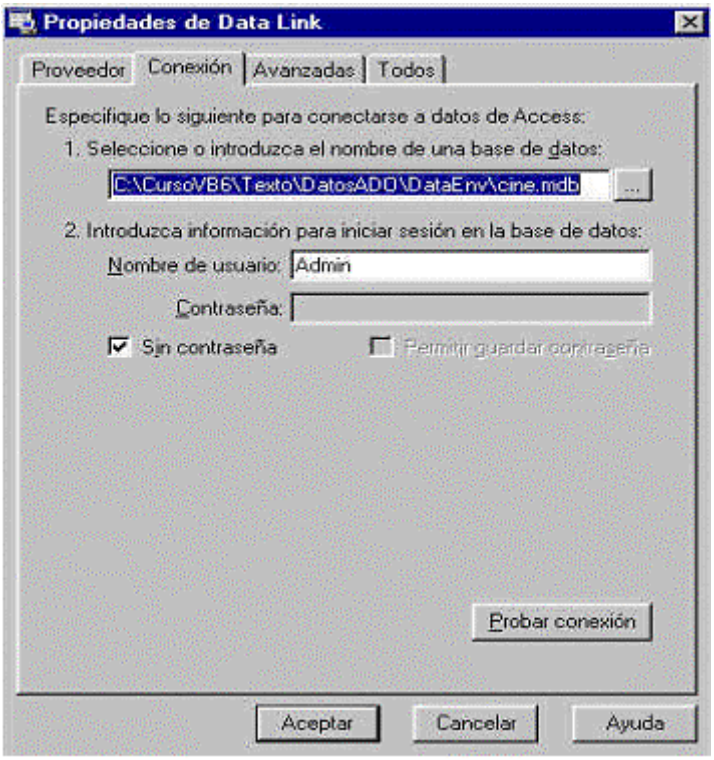


Figura 240. Propiedades para una conexión del Entorno; selección la base de datos.

Debido a las características de esta utilidad, estamos empleando una ruta de datos prefijada, por lo que el lector deberá crear esa misma ruta para poder probar estos ejemplos o alterarla en las propiedades de los objetos Connection correspondientes del diseñador.

En este punto, y después de haber comprobado que la conexión se establece con normalidad, podemos pulsar Aceptar, dejando en la pestaña Avanzadas las opciones por defecto referentes a los permisos de acceso para usuarios.

Crear un comando

Para poder acceder por fin a la base de datos, ya sólo nos queda crear un objeto Command a partir de una conexión. Seleccionaremos en el diseñador, el objeto Connection a partir del que vamos a crear un Command, y pulsaremos el botón Agregar comando,  , de la barra de herramientas del Entorno de Datos, o la opción del mismo nombre en el menú contextual del diseñador.

Para asignar las propiedades a un comando, podemos usar la ventana habitual de propiedades, aunque es más recomendable utilizar sus páginas de propiedades, que podemos abrir pulsando el botón Propiedades de la barra de herramientas del Entorno. Este medio nos permite manejar las propiedades de forma más organizada, disponiendo de controles más adecuados para establecer los diferentes valores. La figura 241, muestra esta ventana con las propiedades para el objeto Command acmPeliculas, que acabamos de crear, basado en la apertura de una tabla de la base de datos.

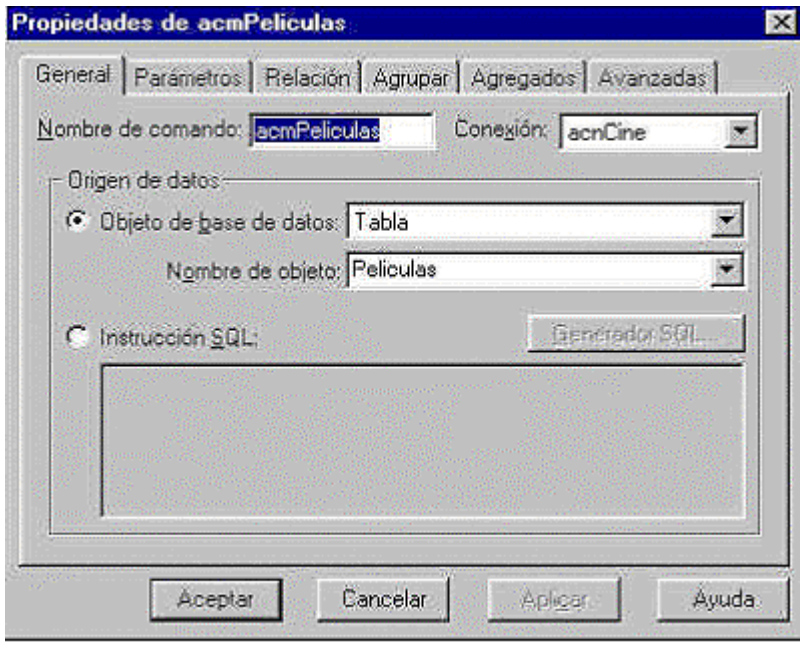


Figura 241. Páginas de propiedades para un objeto Command del Data Environment.

La ejecución de un objeto Command, permite la devolución de un Recordset que podemos configurar mediante la pestaña Avanzadas en las páginas de propiedades del comando, de manera que podemos establecer el tipo de bloqueo, cursor, etc.

Para comprobar el resultado de la ejecución de acmPeliculas, crearemos un formulario que llamaremos frmComando, y en el que insertaremos un control DataGrid. Al pulsar sobre la propiedad DataSource del DataGrid, aparecerá la lista de fuentes de datos disponibles en el proyecto, en nuestro caso será el Entorno de Datos dteEntorno, el cual seleccionaremos. A continuación haremos clic sobre la propiedad DataMember, e igualmente se abrirá una lista con los miembros de datos (objetos Command del Entorno) disponibles. Seleccionaremos acmPeliculas, y finalmente sólo quedará

ejecutar la aplicación, para comprobar como la tabla que contiene este objeto se visualiza en el formulario. La opción Archivo+Comando con formato tabla en el menú del formulario mdiInicio se encarga de abrir este formulario.

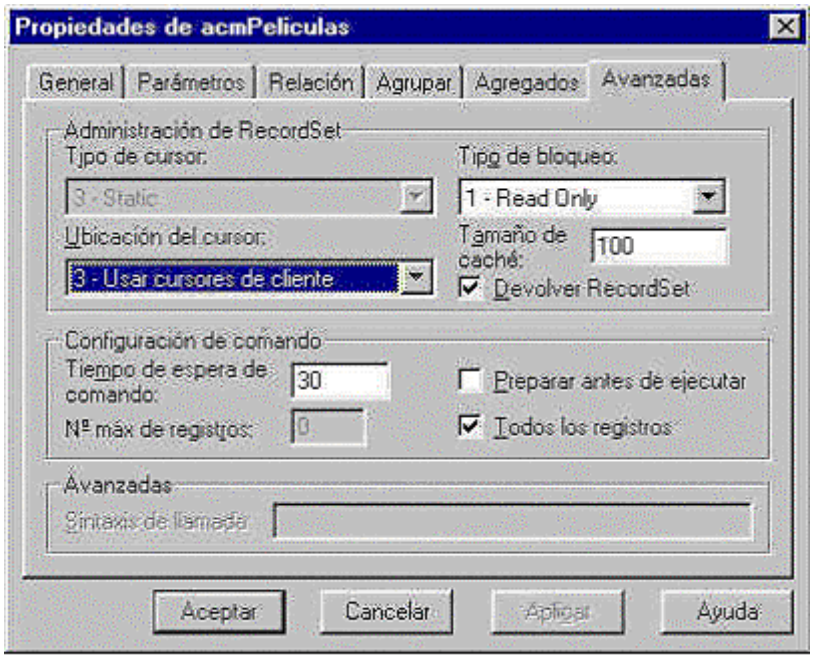


Figura 242. Propiedades para el Recordset devuelto por un objeto Command.

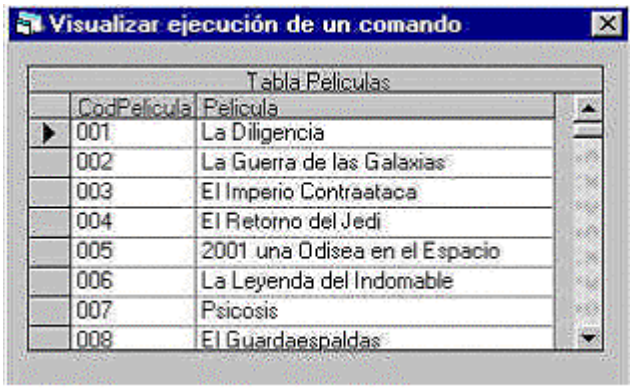


Figura 243. Ejecución del objeto comando contenido en un Entorno de Datos.

Agregar código a los objetos de datos

Seleccionando uno de los objetos en el Data Environment, y pulsando el botón Ver código, , en la barra de herramientas o la misma opción en el menú contextual, se abrirá la ventana de código del Entorno de Datos, para poder editar los eventos de los objetos creados hasta el momento.

En el ejemplo anterior, hemos introducido código para el objeto Connection acnCine, en concreto para los eventos que ocurren cuando se va a establecer la conexión y una vez establecida, como habrá podido comprobar el lector al ejecutarlo. A continuación se muestran las líneas de código para tales eventos.

```
Private Sub acnCine_WillConnect(ConnectionString As String,
 UserID As String, Password As String, Options As Long,
 adStatus As ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
MsgBox "Conectando a la base de datos..."
End Sub
' ------------------------------------------------
Private Sub acnCine_ConnectComplete(ByVal pError As ADODB.Error,
 adStatus As ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
MsgBox "Conexión efectuada"
End Sub
```

Código fuente 234

Otros tipos de comandos

Aparte de la presentación que acabamos de ver en forma de tabla, es posible crear un objeto Command basado en una consulta o vista de una base de datos. Para ello, crearemos una nueva conexión en el diseñador llamada acnMusica, dirigida hacia el fichero Musica.MDB, y crearemos también un nuevo comando llamado acmVariasGrab que abrirá la vista VariasGrab, contenida en esta base de datos. La figura 244 muestra las propiedades para este tipo de comando.

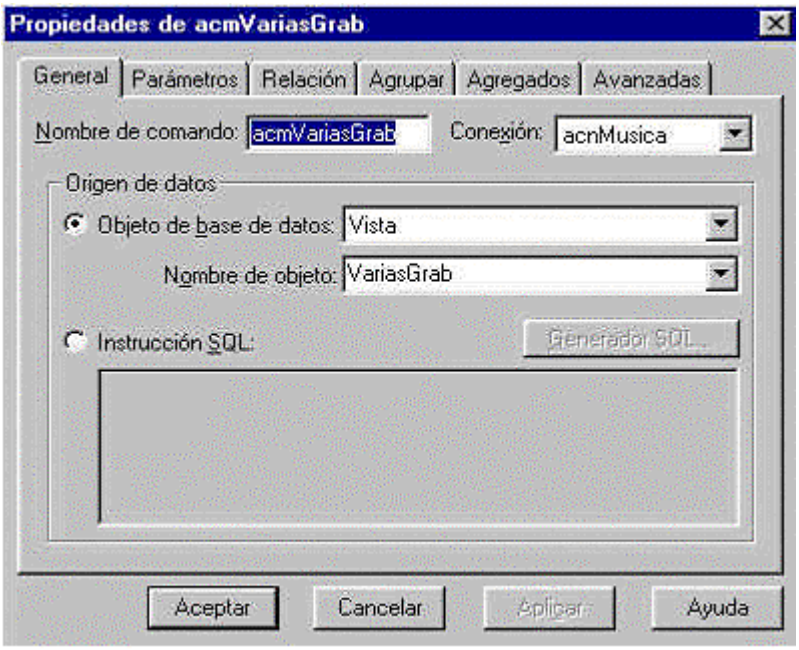


Figura 244. Propiedades para un objeto Command basado en una vista de la base de datos.

Igualmente, crearemos otro comando basado en una consulta SQL, al que llamaremos acmConsulta, basado en la instrucción que vemos en la figura 245

Comprobaremos el resultado de todos los objetos Command al mismo tiempo utilizando el formulario frmVerComandos, en el que se ha insertado un DataGrid por cada uno de los comandos que hemos incluido hasta el momento en el Data Environment.

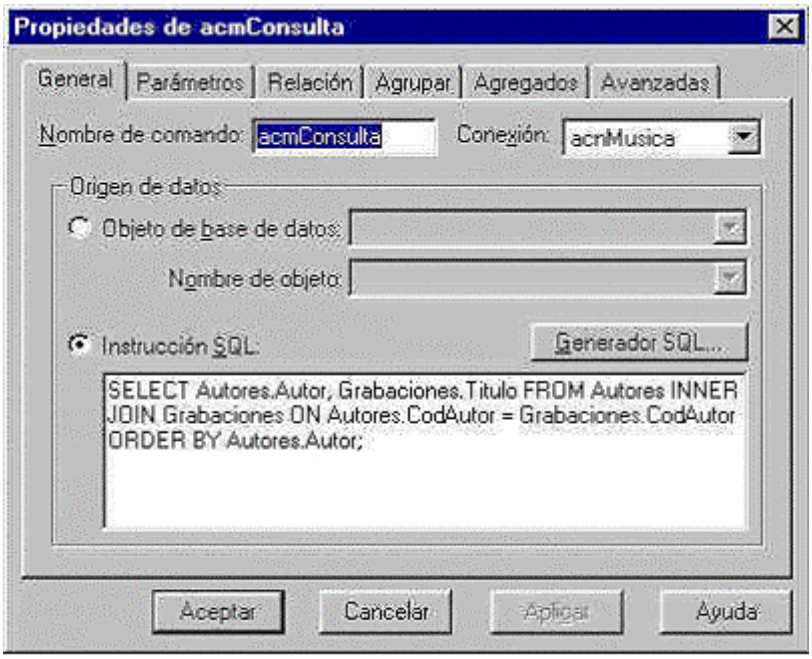


Figura 245. Propiedades para un objeto Command basado en una consulta SQL.

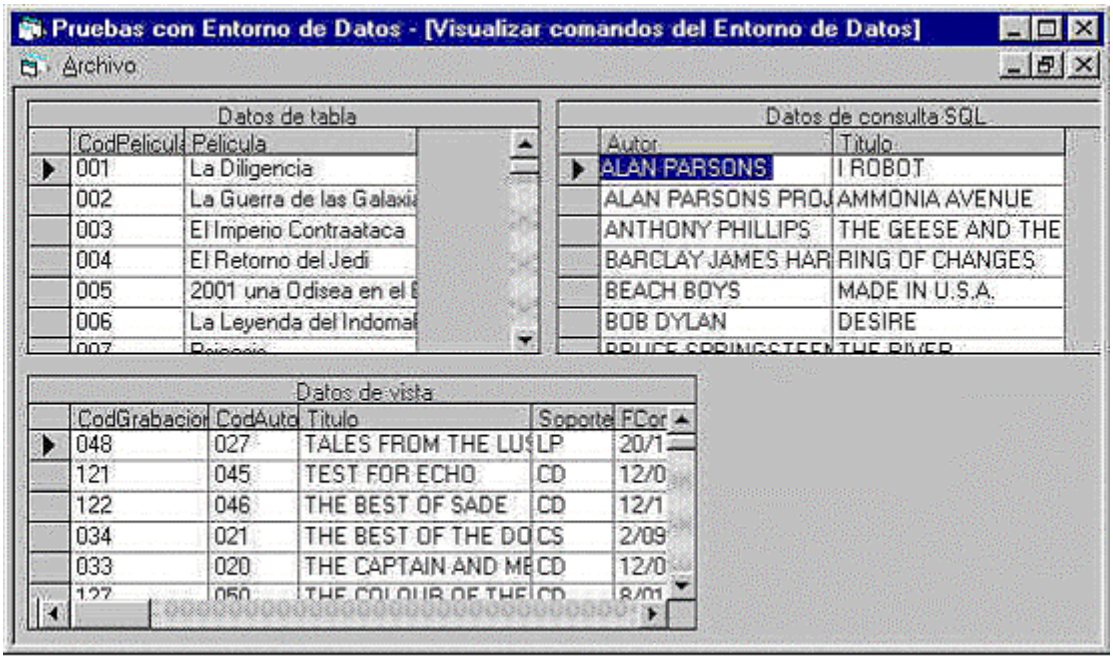


Figura 246. Ejecución simultánea de varios comandos del Entorno de Datos.

Todo el trabajo realizado hasta el momento, sólo ha requerido labor en el ámbito de diseño visual, sin necesidad de escribir código, lo que puede dar al lector una idea de la potencia y flexibilidad del Entorno de Datos para la presentación de datos al usuario.

Simplificar el desarrollo de un mantenimiento de datos

Desde el Entorno de Datos es posible seleccionar cualquier campo que forme parte de un objeto Command, y arrastrarlo hasta un formulario, de manera que al soltarlo, se genere automáticamente un control para editar dicho campo en el formulario.

Igualmente, si arrastramos y soltamos un objeto Command entero, se crearán tantos controles como campos tenga ese comando. Junto a cada control aparecerá un Label con el nombre del campo, para facilitar su reconocimiento. Esta clase de ayudas, facilitan en gran medida nuestro trabajo en los aspectos más rutinarios y repetitivos del desarrollo de la aplicación, en lo que a datos se refiere.

El tipo de control creado para editar cada campo, está especificado por defecto en el Entorno de Datos; por ejemplo: para un campo de una tabla, cuyo tipo de dato sea texto, se creará un control TextBox al arrastrarlo a un formulario, para un campo lógico se usará un CheckBox, etc. Sin embargo, el programador puede alterar este comportamiento, especificando un control diferente según sus necesidades.

Vamos a desarrollar un mantenimiento para el comando acmGrabaciones del Entorno de Datos de nuestro proyecto. Este comando muestra los registros de la tabla Grabaciones que hay en la base de datos Musica.MDB.

Si arrastramos y soltamos este objeto sobre el formulario, se crearán los correspondientes controles para cada campo. Todos serán TextBox a excepción del campo Oferta, que al ser de tipo lógico, tendrá asociado un CheckBox.

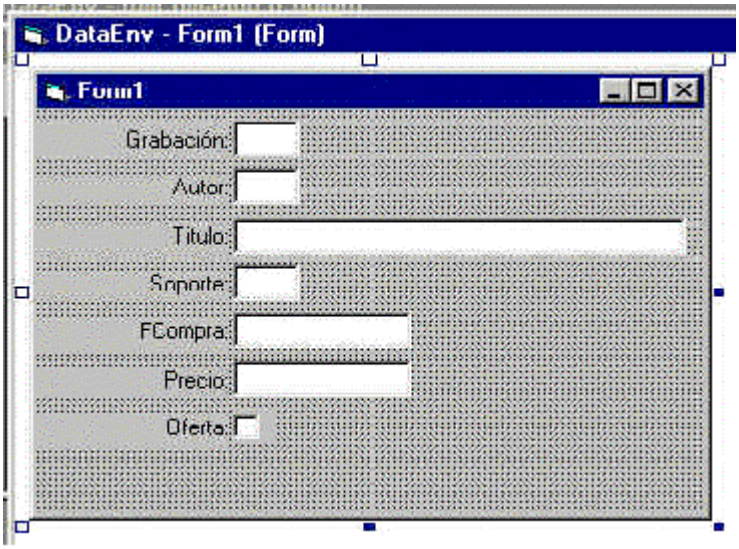


Figura 247. Controles de edición creados al soltar un Command sobre un formulario.

Pero observemos más detenidamente el campo FCompra. Aunque podamos manipularlo a través de un TextBox, VB dispone de controles más adecuados para el tratamiento de fechas, por ejemplo el control Microsoft Date and Time Picker, que tiene una apariencia similar a un ComboBox; en la zona de edición se introduce la fecha y al pulsar su botón, se despliega un calendario gráfico con el mes perteneciente a la fecha que aparece en el control.

La modificación del control asociado a este campo puede realizarse de dos formas:

- Seleccionar el comando que contiene este campo y hacer clic en el botón Opciones, del Data Environment,  . Se mostrará dicha ventana, que contiene un conjunto de ajustes para los objetos del diseñador. Al hacer clic en la pestaña Asignación de campos de esta ventana, podremos seleccionar uno de los tipos de datos y asignarlo a un control determinado o dejar el que tiene por defecto. En nuestro caso, pulsaremos sobre el tipo Date, y en la lista de controles elegiremos el control Microsoft Date and Time Picker, como muestra la figura 248.

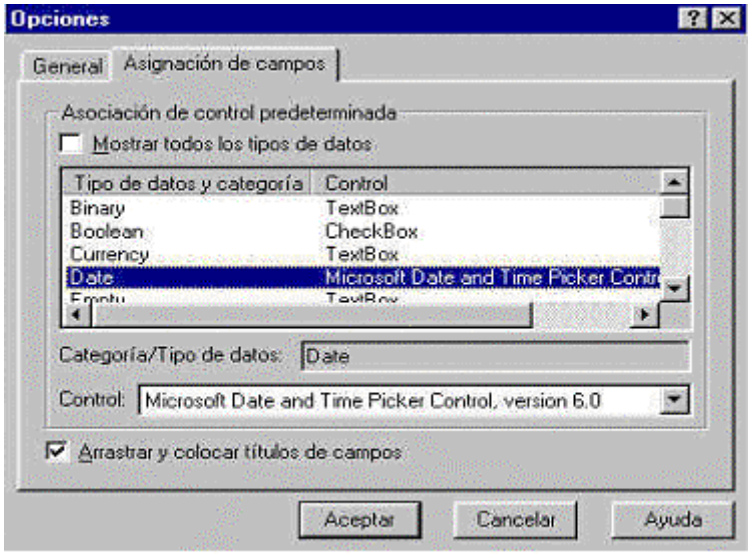


Figura 248. Asignación de controles a tipos de datos.

- El otro medio, consiste en seleccionar desde el diseñador, el campo al que vamos a asignar un nuevo control, y pulsar el botón Propiedades o elegir esta opción en su menú contextual. Esta acción nos mostrará una ventana informativa sólo para ese campo y la posibilidad de asignarle un control de edición diferente al que tiene por defecto.

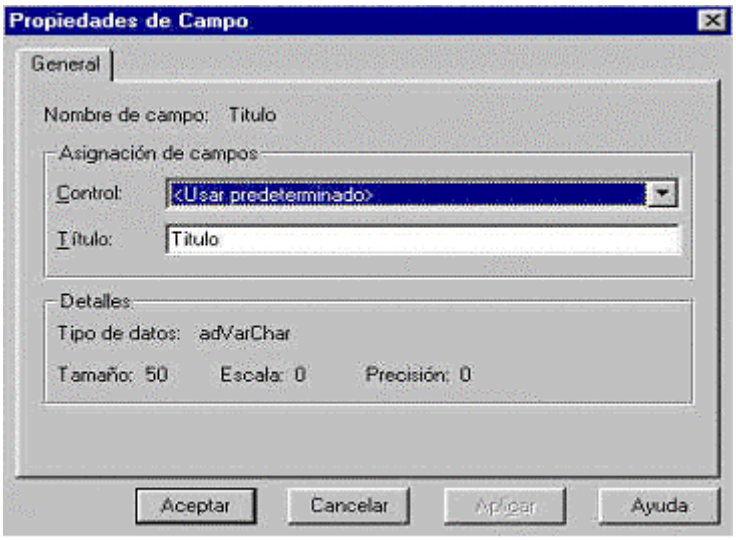


Figura 249. Asignación de control a un único campo.

La inserción automática de los controles de edición de campos, aunque supone una gran ayuda, no nos exime de trabajo, ya que debemos añadir los botones de edición y navegación, así como el código correspondiente para hacerlos funcionar. Sin embargo, al tener ya establecidas en el Entorno las propiedades de conexión y comandos, utilizaremos el propio objeto diseñador para realizar estas tareas.

Para manipular el Entorno de Datos en el código, usaremos directamente el nombre del diseñador sin necesidad de crear una instancia previa, ya que si intentáramos trabajar con una instancia del Data Environment, los controles enlazados al comando no responderían. Según esto, el código fuente 235 no funcionaría.

```
Private Sub cmdAvanzar_Click()
Dim ldteDatos As dteEntorno
Set ldteDatos = New dteEntorno
ldteDatos.rsacmGrabaciones.MoveNext
End Sub
```

Código fuente 235

Empleando el diseñador, el código fuente 235 se simplifica y conseguimos que funcione, veamos el código fuente 236

```
Private Sub cmdAvanzar_Click()
ldteEntorno.rsacmGrabaciones.MoveNext
End Sub
```

Código fuente 236

En la figura 250 se muestra el formulario conteniendo los controles resultado de depositar el objeto acmGrabaciones en su interior, y los diferentes botones para la navegación y edición del comando.

Figura 250. Formulario para la edición de un Command del Data Environment.

El código fuente 237, contiene las rutinas de desplazamiento por los registros del comando.

```
Private Sub cmdAvanzar_Click()
dteEntorno.rsacmGrabaciones.MoveNext
If dteEntorno.rsacmGrabaciones.EOF Then
   dteEntorno.rsacmGrabaciones.MoveLast
   MsgBox "Final de fichero", , "Aviso"
End If
End Sub
'--------------------------------------
Private Sub cmdRetroceder_Click()
dteEntorno.rsacmGrabaciones.MovePrevious
If dteEntorno.rsacmGrabaciones.BOF Then
   dteEntorno.rsacmGrabaciones.MoveFirst
   MsgBox "Principio de fichero", , "Aviso"
End If
End Sub
'--------------------------------------
Private Sub cmdPrincipio_Click()
dteEntorno.rsacmGrabaciones.MoveFirst
End Sub
'--------------------------------------
Private Sub cmdFinal_Click()
dteEntorno.rsacmGrabaciones.MoveLast
End Sub
```

Código fuente 237

Observe el lector, que para manejar los registros de un Command en el código, no utilizamos dicho objeto. Cada vez que en tiempo de diseño, creamos un nuevo comando, se genera al mismo tiempo un Recordset, con el mismo nombre que el comando, pero precedido del indicativo rs (rsacmGrabaciones), que será el que debamos emplear para la manipulación de los registros en el código.

En lo que respecta a la edición de registros, el código fuente 238 muestra el código de los botones que efectúan dicho trabajo, y los procedimientos que habilitan/deshabilitan los controles del formulario, en función del estado de edición del registro.

```
Private Sub cmdNuevo_Click()
dteEntorno.rsacmGrabaciones.AddNew
ControlesEditar
End Sub
'--------------------------------------
Private Sub cmdModificar_Click()

ControlesEditar

End Sub

'--------------------------------------
Private Sub cmdGrabar_Click()

dteEntorno.rsacmGrabaciones.Update
ControlesNoEditar

End Sub

'--------------------------------------
Private Sub cmdCancelEdit_Click()

dteEntorno.rsacmGrabaciones.CancelUpdate
```

```
dteEntorno.rsacmGrabaciones.Move 0
ControlesNoEditar

End Sub
'---------------------------------------
Private Sub ControlesEditar()
Me.cmdAvanzar.Enabled = False
Me.cmdRetroceder.Enabled = False
Me.cmdPrincipio.Enabled = False
Me.cmdFinal.Enabled = False
Me.cmdNuevo.Enabled = False
Me.cmdModificar.Enabled = False
Me.txtCodGrabacion.Enabled = True
Me.txtCodAutor.Enabled = True
Me.txtTitulo.Enabled = True
Me.txtSoporte.Enabled = True
Me.ctlFCompra.Enabled = True
Me.txtPrecio.Enabled = True
Me.chkOferta.Enabled = True
Me.cmdGrabar.Enabled = True
Me.cmdCancelEdit.Enabled = True
End Sub
'-------------------------------------
Private Sub ControlesNoEditar()
Me.cmdAvanzar.Enabled = True
Me.cmdRetroceder.Enabled = True
Me.cmdPrincipio.Enabled = True
Me.cmdFinal.Enabled = True
Me.cmdNuevo.Enabled = True
Me.cmdModificar.Enabled = True
Me.txtCodGrabacion.Enabled = False
Me.txtCodAutor.Enabled = False
Me.txtTitulo.Enabled = False
Me.txtSoporte.Enabled = False
Me.ctlFCompra.Enabled = False
Me.txtPrecio.Enabled = False
Me.chkOferta.Enabled = False
Me.cmdGrabar.Enabled = False
Me.cmdCancelEdit.Enabled = False
End Sub
```

Código fuente 238

Finalmente, sólo queda mostrar el resultado de la ejecución de este formulario, con el control especial para el campo de fecha. La opción de menú Archivo+Arrastrar y soltar campos de la ventana principal del proyecto, nos llevará a este formulario.

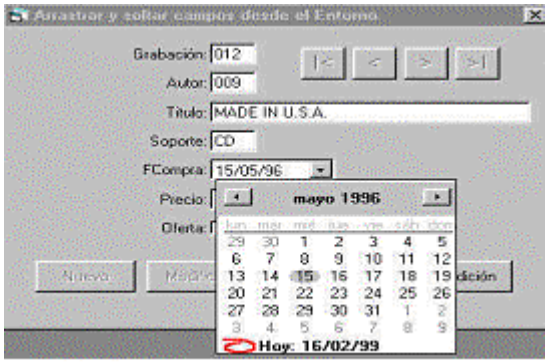


Figura 251. Mantenimiento de datos diseñado con el Data Environment.

Manipular los elementos de un Entorno de Datos en tiempo de ejecución

A pesar de que el medio indicado para modificar las conexiones y comandos que hayamos creado en un Entorno de Datos es el propio diseñador, puede darse el caso de que necesitemos realizar cambios en alguno de estos objetos, pero que no sea posible efectuarlos hasta que no esté la aplicación en ejecución. En la aplicación de ejemplo, disponemos del formulario frmModifCodigo, conteniendo el DataGrid grdModificar, que muestra los registros pertenecientes al comando acmConsulta, del Entorno de Datos de esta aplicación. Cuando pulsemos el botón cmdModificar, también en este formulario, se ejecutará el código fuente 239, que cambiará la conexión a una base de datos y comando diferentes, utilizando por consiguiente, un nuevo conjunto de registros.

```
Private Sub cmdModificar_Click()
Dim ldteEntorno As dteEntorno
Dim lacnConexion As ADODB.Connection
Dim lacmComando As ADODB.Command
' crear una instancia del Entorno de Datos
Set ldteEntorno = New dteEntorno
' tomar el objeto Connection de la colección
' de conexiones
Set lacnConexion = ldteEntorno.Connections(1)
' modificar la cadena de conexión del objeto
lacnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=C:\CursoVB6\Texto\DatosADO\DataEnv\factu.mdb"
' abrir la conexión
lacnConexion.Open
' tomar el comando que muestra los registros
Set lacmComando = ldteEntorno.Commands(1)
' el comando existente en diseño, contiene una consulta SQL,
' sin embargo este mostrará los registros como una tabla
lacmComando.CommandType = adCmdTable
' indicar la tabla a abrir
lacmComando.CommandText = "facturas"
' reconfigurar el DataGrid para que muestre
' los nuevos registros
Set Me.grdModificar.DataSource = ldteEntorno
Me.grdModificar.DataMember = "acmPeliculas"
End Sub
```

Código fuente 239

Las figuras 252 y 253, muestran el formulario de este ejemplo, antes y después de realizar la conexión con la nueva base de datos empleando código.

Figura 252. Formulario con los datos establecidos desde el Entorno de Datos.

Figura 253. Formulario que muestra el cambio de conexión mediante código en un Entorno de Datos.

Relacionar comandos del Entorno de Datos

El Data Environment nos permite establecer una relación entre dos tablas por un campo común, obteniendo un resultado de estilo maestro/detalle.

Como ejemplo, vamos a crear una relación entre las tablas Autores y Grabaciones de la base de datos Musica.MDB, por el campo CodAutor, mostrando los resultados en un formulario creado a tal efecto.

Los pasos a seguir para conseguir nuestro objetivo son los siguientes:

- Crear un nuevo objeto Command, dependiente de la conexión acnMusica, que abra la tabla Autores. El nombre de este objeto será acmAutor, como se muestra en la figura 254.

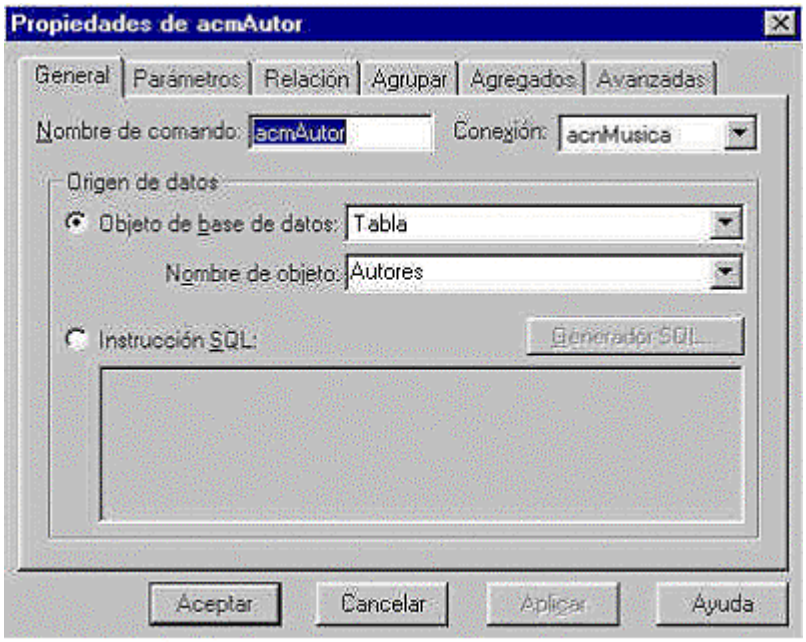


Figura 254. Comando primario de la relación.

- A partir del nuevo comando crearemos un comando secundario, mediante la opción Agregar comando secundario del menú contextual del objeto, o pulsando el botón de la

barra de herramientas  . Estableceremos las propiedades de este nuevo comando, para que abra la tabla Grabaciones, y le daremos el nombre acmGrabacion.

- Finalmente, en la ventana de páginas de propiedades de este último comando, haremos clic en la pestaña Relación, estableciendo el comando primario con el que se va a relacionar y los campos que componen la relación. La figura 255 muestra esta ventana.

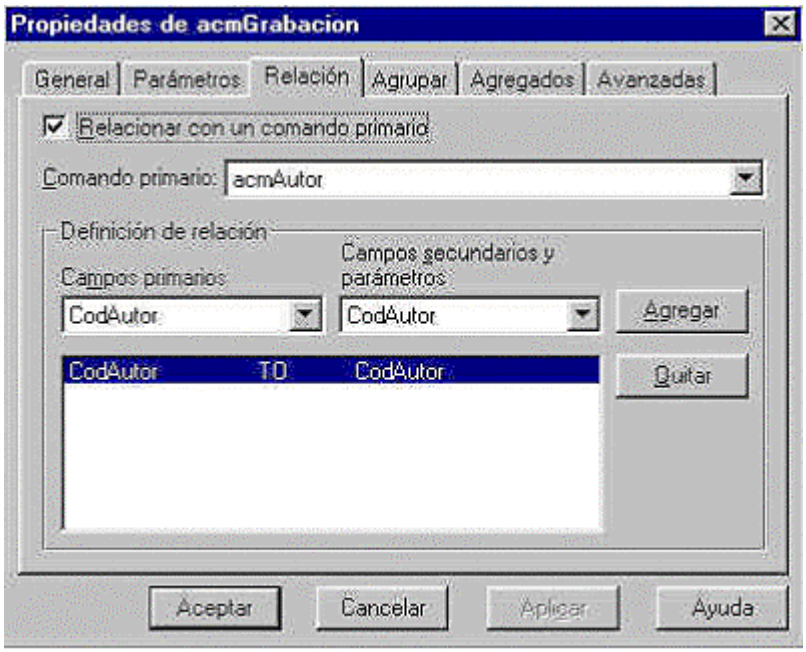


Figura 255. Propiedades del comando secundario de la relación.

Para mostrar el resultado al usuario, haremos uso del formulario frmRelacion, en el que se ha insertado el nuevo control Hierarchical FlexGrid. Este control, junto a las técnicas de creación de recordsets jerárquicos ya han sido comentados en el apartado dedicado al modelado de datos, y nos permite la presentación al usuario de los registros en formatos más complejos que en el habitual control DataGrid. Estableciendo su propiedad DataSource a dteEntorno, y la propiedad DataMember a acmAutor, los registros serán mostrados de forma similar a la figura 256.

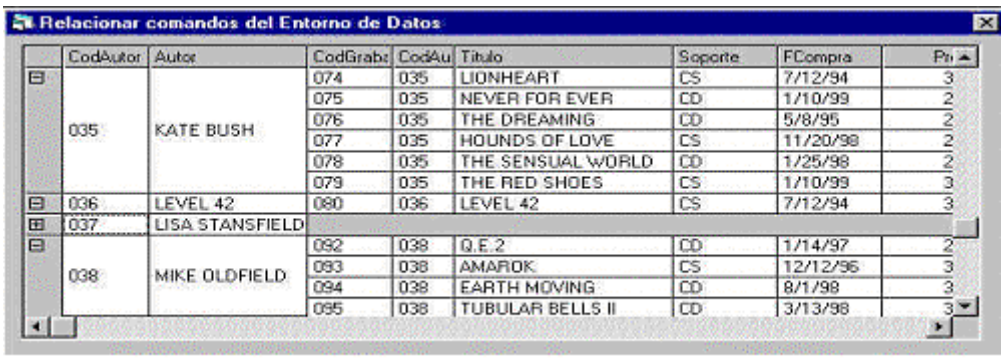


Figura 256. Formulario mostrando la relación entre comandos del Entorno de Datos mediante el control Hierarchical FlexGrid.

Haciendo clic en el bitmap existente en la primera columna, podemos expandir o contraer los registros relacionados en el comando secundario.

Agregados

Otra de las operaciones que podemos realizar desde el Data Environment, es la suma de los registros de un campo perteneciente a un objeto Command, sobre la base de una determinada condición. Como resultado obtendremos un nuevo objeto de tipo Field (Campo), al que se denomina agregado, y que se añadirá a un comando secundario.

Para demostrar este aspecto del Entorno de Datos, crearemos un ejemplo con un campo agregado, que sumará por cada autor existente, los valores del campo Precio en la tabla Grabaciones, visualizando el resultado en un control Hierarchical FlexGrid. A continuación vemos los pasos a seguir:

- Crear a partir de la conexión acnMusica, un nuevo comando que abra la tabla Autores, al que daremos el nombre acmSumaPrecio.

- Seguidamente crearemos el comando secundario acmGrab, que abrirá la tabla Grabaciones, y que relacionaremos con acmSumaPrecio por el campo CodAutor.

- Pasaremos nuevamente al comando acmSumaPrecio y pulsaremos la pestaña Agregados de su ventana de propiedades. Daremos el nombre TotalPrecios al agregado; estableceremos Suma como la operación a realizar; el campo resultante lo asignaremos al comando acmGrab; y el campo que sumaremos será Precio. La figura 257 muestra estas propiedades.

Figura 257. Propiedades para establecer un agregado dentro de un objeto Command.

Ya sólo resta, al igual que en el punto anterior, emplear un formulario con un control Hierarchical FlexGrid para visualizar el resultado. En la propiedad DataMember de este control, asignaremos el comando acmSumaPrecio, puesto que es el que contiene el campo agregado. La figura 258 muestra el resultado de este ejemplo. Observe el lector como pueden contraerse o expandirse las filas

pertenecientes a un mismo autor, y la suma efectuada igualmente del campo Precio, separada por autor, en la columna TotalPrecios.

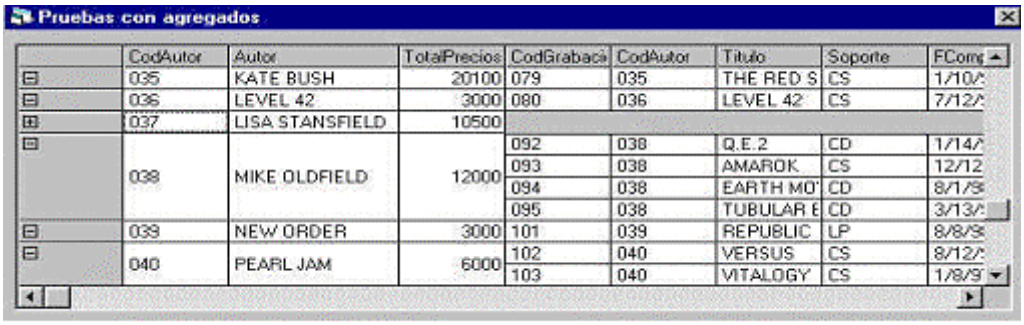


Figura 258. Resultado de la ejecución de un objeto Command conteniendo un campo agregado.

La opción Archivo+Agregado, en el formulario principal del programa, mostrará la columna TotalPrecios con el cálculo realizado.

Agrupar filas sobre la base de un campo

Tomando la tabla Grabaciones de nuevo, supongamos que debemos mostrar sus registros, de forma que se visualicen por grupos los que tengan el mismo soporte. A continuación se muestran las fases de esta operación.

- Crear el comando acmAgrupar, que abrirá la tabla Grabaciones.
- En la ventana de páginas de propiedades del comando, pulsar la pestaña Agrupar. En este grupo de propiedades, debemos seleccionar de una lista, el campo por el que se realizará la agrupación: Soporte. Esta operación creará un comando de agrupación al que daremos el nombre PorSoporte.

Figura 259. Propiedades para establecer una agrupación de registros en un objeto Command.

La ventana del Data Environment, presentará acmAgrupar de la siguiente manera:

Figura 260. Ventana del diseñador del Entorno de Datos mostrando el Command acmAgrupar.

Mostraremos el resultado al usuario de igual forma que en los últimos apartados. Cabe destacar, sin embargo, que al asignar en el DataGrid, la propiedad DataMember, debemos utilizar el miembro PorSoporte, en lugar de acmAgrupar como se podría pensar inicialmente. El motivo es que PorSoporte es el miembro que realmente hace referencia al agrupamiento efectuado en el objeto Command, y mostrará los registros de la forma que queremos.

Figura 261. Formulario mostrando los registros agrupados.

Al ejecutar la aplicación, la opción de menú Archivo+Agrupado será la que visualice este formulario.

Contar las filas de un campo agrupado

Basándonos en el punto anterior, es posible, a partir de un objeto Command de agrupamiento, contar las filas existentes dentro de cada grupo. Para ello crearemos el comando acmGrabSopor, que accederá a la tabla Grabaciones y agrupará los registros por el campo Soporte, sólo que en este caso, denominaremos PorSoporte2 al comando de agrupación, para que no exista conflicto de nombres con el anterior.

Hasta el momento todo igual, las filas aparecerán agrupadas por un campo. Si queremos además, mostrar una columna adicional con el número de filas que contiene cada grupo, deberemos ir a la pestaña Agregados de la ventana de propiedades del comando, y crear un nuevo campo, que llamaremos ContarGrabaciones. Como función a realizar, seleccionaremos Cuenta; para el modo de agregar seleccionaremos Grouping; mientras que el campo empleado para contar los registros será CodGrabacion. La figura 262 muestra estas propiedades.

Figura 262. Diseño de un campo de agregado para contar los registros de un comando agrupado.

En el proyecto emplearemos el formulario frmGrabSoporte, que ejecutaremos mediante la opción de menú Archivo+Contar Grabaciones por Soporte, para mostrar al usuario el resultado de este comando.

Figura 263. Formulario mostrando la ejecución del comando de agrupamiento que cuenta los registros de cada grupo.

Para poder visualizar los registros de esta manera, asignaremos el comando de agrupado PorSoporte2 a la propiedad DataMember en el control Hierarchical FlexGrid del formulario.

Vista Datos (Data View)

La ventana Vista Datos es una herramienta que nos permite tener acceso a las conexiones establecidas en los diferentes Entornos de Datos existentes en un proyecto, agregar nuevos diseñadores de Entorno de Datos a la aplicación, crear vínculos de datos, arrastrar y soltar elementos tales como tablas, vistas, procedimientos almacenados en un Entorno de Datos y ejecutarlos para ver su contenido.

Para comprobar algunas de estas características, disponemos del proyecto de ejemplo DataView, y debido a que el manejo de bases de datos con estas utilidades se realiza en tiempo de diseño, necesitaremos emplear una ruta previa que será: C:\CursoVB6\Texto\DatosADO\DataView, por lo que el lector deberá crear dicha carpeta en su equipo para estas pruebas.

Después de estas aclaraciones, veamos como se realiza el trabajo al desarrollar una aplicación empleando la ventana Vista Datos (figura 264).

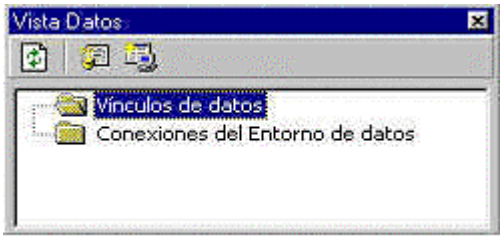


Figura 264. Ventana Vista Datos vacía.

Visualizar las conexiones del Entorno de Datos en Vista Datos

Cuando desde un Entorno de Datos, establecemos las propiedades para un objeto Connection, indicando la base de datos a la que se conecta, dicha información será actualizada al mismo tiempo en Vista Datos. La figura 265, muestra esta operación, al conectar con el fichero Cine.MDB en el Entorno dteCine del proyecto de ejemplo.

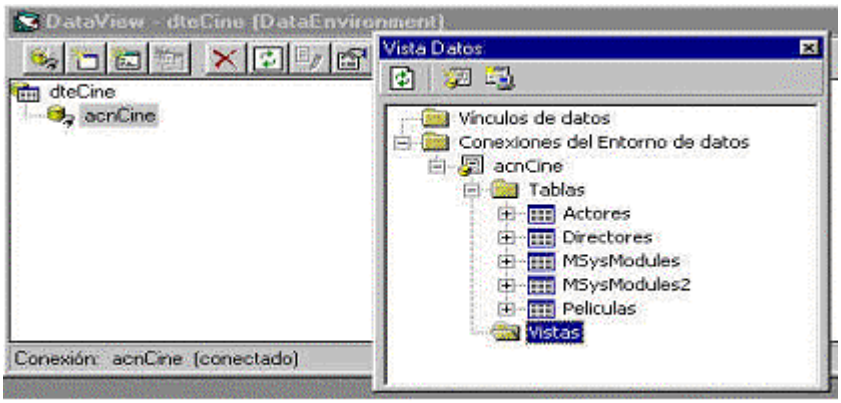


Figura 265. Visualización en Vista Datos de una conexión creada en el Entorno de Datos.

Lo que podemos ver en Vista Datos acerca de una conexión, son todos los elementos propios de la base de datos: tablas, vistas, procedimientos almacenados, etc.; mientras que los comandos, relaciones, agrupados, agregados, etc., sólo podrán ser visualizados en la ventana de Entorno de Datos, ya que son propios de este tipo de diseñador.

Crear un Entorno de Datos desde Vista Datos

Mediante el botón Agregar entorno de datos al proyecto actual  , de la barra de herramientas de Vista Datos, podremos añadir un nuevo diseñador de este tipo al programa que estamos desarrollando.

La figura 266, muestra un Entorno de Datos creado desde Vista Datos, y esta última ventana con las conexiones establecidas, pertenecientes a todos los Entornos del proyecto. Observe el lector, que en el Entorno se ha creado un objeto Command que abre la tabla Titles, pero dicho objeto no se refleja en Vista Datos.

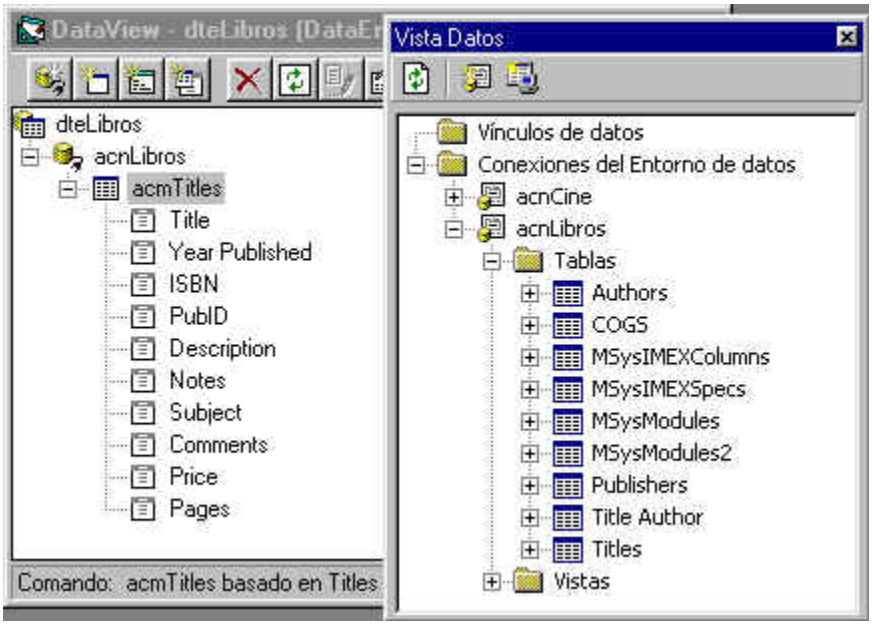


Figura 266. Vista Datos mostrando las conexiones del proyecto.

Crear un vínculo desde Vista Datos

La opción Agregar un vínculo de datos, del menú contextual de Vista Datos, o su botón de la barra de herramientas  , nos permite establecer un enlace con una base de datos. En este caso, vamos a conectar con una de las bases de datos de ejemplo, incluida con SQL Server. Una vez arrancada esta opción, se mostrará la ventana de propiedades del enlace de datos, que nos guiará a modo de asistente en el establecimiento de la conexión.

Después de seleccionar como proveedor, el correspondiente a SQL Server, pulsaremos el botón Siguiente>>, para establecer la base de datos a la que nos conectaremos, servidor, usuario, etc.

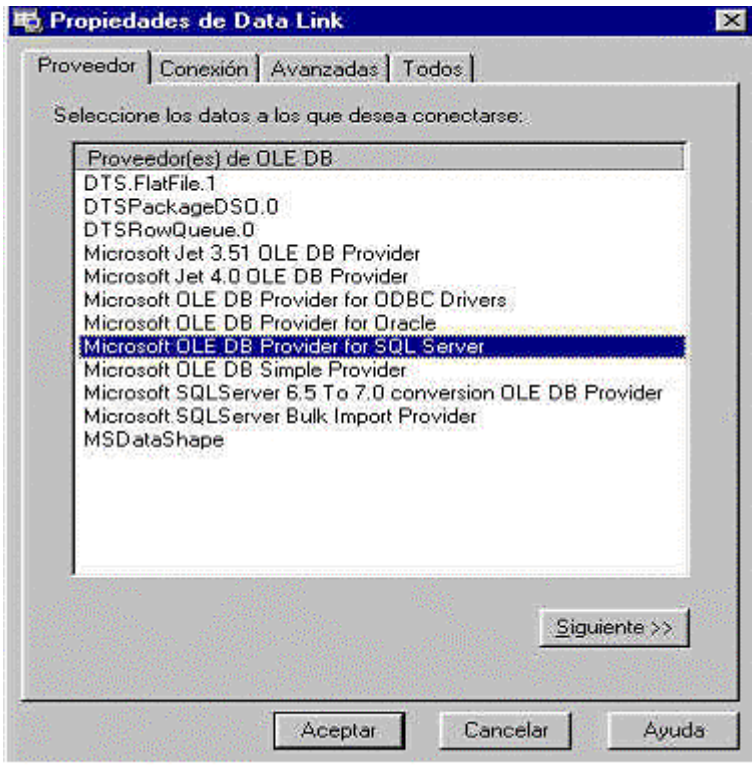


Figura 267. Vínculo de datos, establecimiento del proveedor.

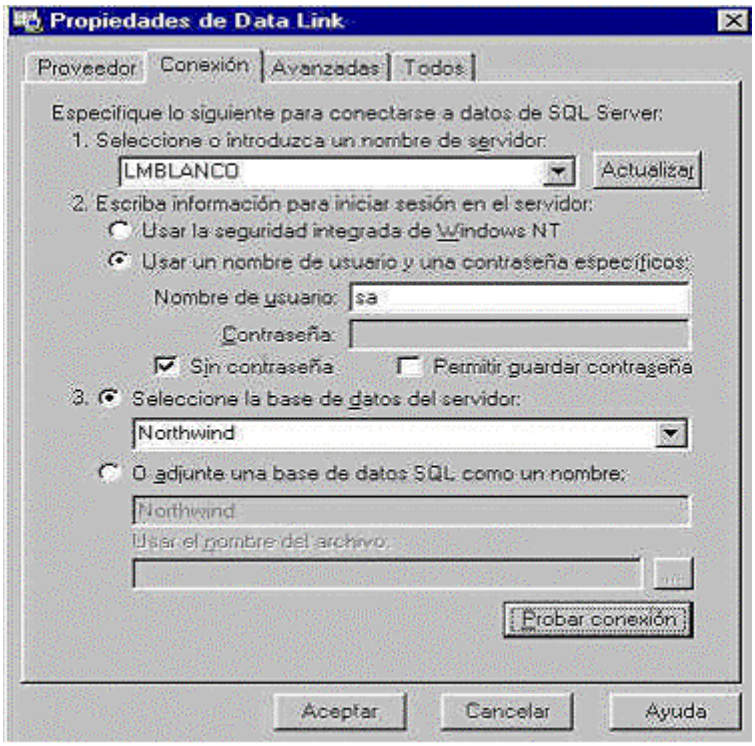


Figura 268. Vínculo de datos, propiedades de conexión.

Una vez probada la conexión, y si resulta satisfactoria, pulsamos Aceptar, con lo cual se añadirá este vínculo a Vista Datos con un nombre por defecto. Después de cambiarle dicho nombre por uno más significativo, por ejemplo Northwind, el mismo que tiene en SQL Server, tendremos a nuestra disposición esta base de datos con todas sus tablas, vistas, diagramas y procedimientos almacenados, como vemos en la figura 269.

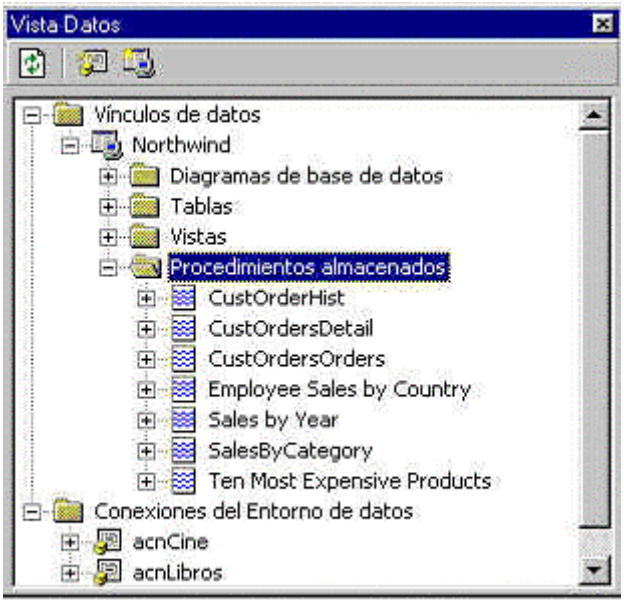


Figura 269. Vista Datos mostrando el vínculo creado.

Acceder a los elementos contenidos en Vista Datos

Todos los componentes de Vista Datos susceptibles de presentar resultados, podrán ser ejecutados para ver su contenido. Tomemos por ejemplo, la tabla Customers de la base de datos Northwind; al hacer doble clic sobre la misma, o elegir la opción Abrir de su menú contextual, se abrirá una ventana visualizando sus registros, como muestra la figura 270.

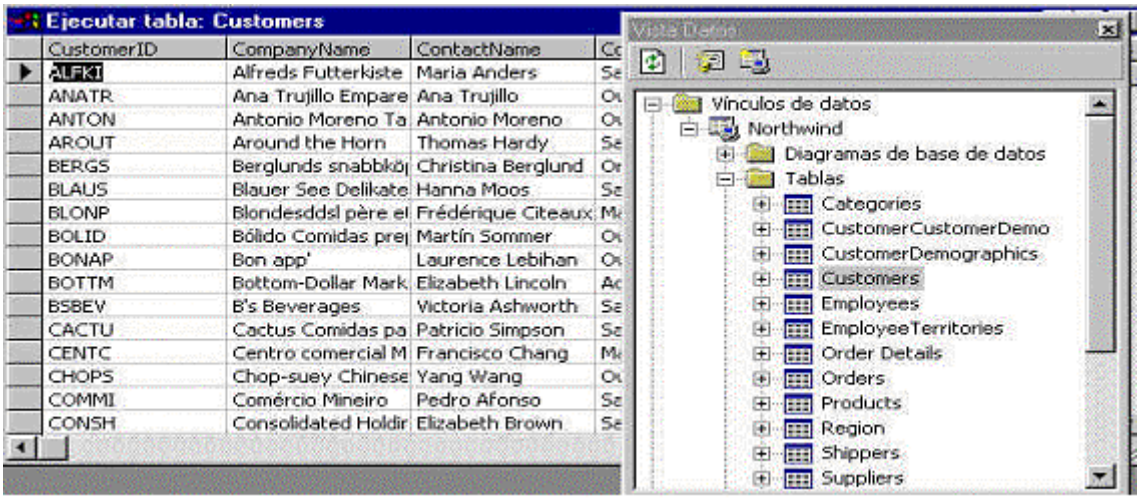


Figura 270. Ejecutar una tabla desde Vista Datos.

De igual modo, es posible seleccionar un procedimiento almacenado de la base de datos, en este caso Sales by Year, y abrir su ventana de edición seleccionando la opción Diseño en su menú contextual.

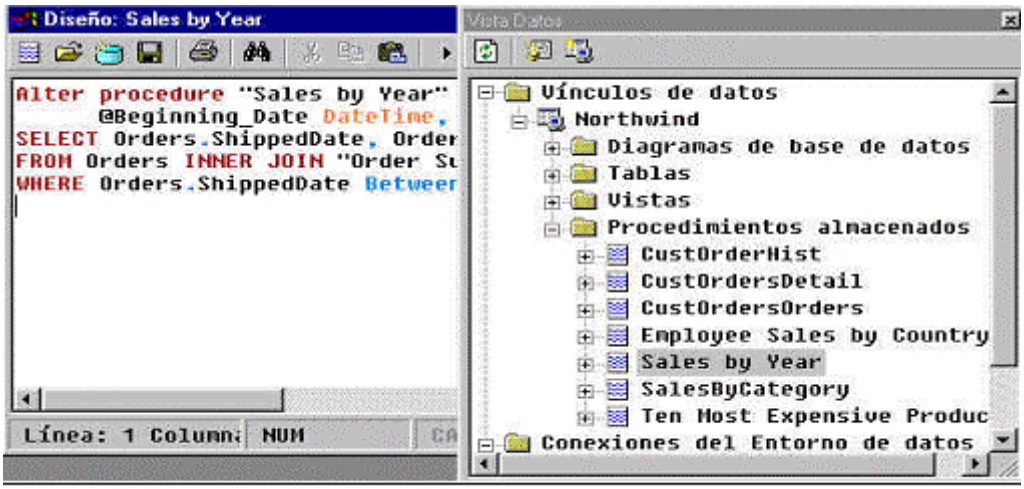


Figura 271. Editar un procedimiento almacenado desde Vista Datos.

Arrastrar y soltar desde Vista Datos

Vista Datos dispone de la posibilidad de tomar los elementos que contiene: tablas, vistas, procedimientos almacenados, etc., y mediante arrastrar y soltar, depositarlos en cualquier Entorno de Datos del proyecto.

En la figura 272, podemos observar como se ha tomado la tabla Categories y se ha soltado en el Entorno dteLibros. Al no existir una conexión previa con su base de datos, se crea una con el nombre predeterminado Connection1. Una vez en el Entorno de Datos, es posible visualizar el comando desde un formulario, de la misma manera que vimos en el apartado anterior.

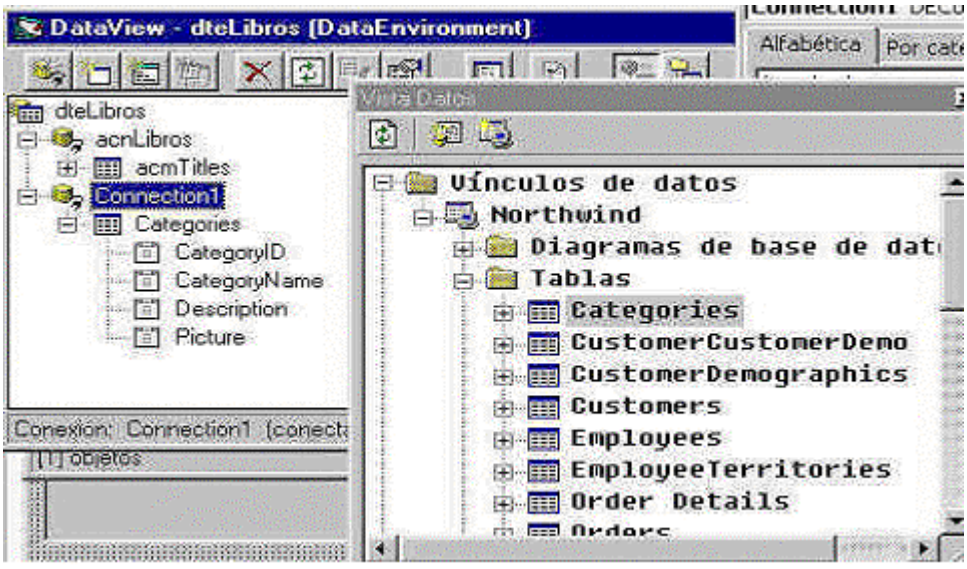


Figura 272. Command creado mediante arrastrar y soltar desde Vista Datos a un Entorno de datos.

Asistente para crear formularios de datos

Visual Basic proporciona entre sus múltiples asistentes, uno destinado a la creación de formularios para realizar mantenimientos de datos. A través de una serie de ventanas en las que tendremos que incluir la información de la base de datos a utilizar, tablas, etc., obtendremos una ventana de manipulación de datos que podremos utilizar directamente o retocar para adaptar a los requerimientos de la aplicación.

El proyecto de ejemplo *AsistDat*, incluye un formulario creado mediante este asistente utilizando la base de datos Almac.MDB, para obtener la información a manipular. La base de datos se incluye dentro de este proyecto y debido a que el asistente necesita una ruta donde localizarla, el lector ha de crearla y depositar en ella el proyecto para poder probarlo. La ruta en cuestión a crear es: C:\CursoVB6\Texto\DatosADO\AsistDat.

A continuación vamos a mostrar los pasos que han sido necesarios seguir para obtener el formulario de este proyecto.

Comenzaremos seleccionando la opción *Complementos+Asistente para formularios de datos...* del menú de VB, que pondrá en marcha dicho asistente. En la pantalla introductoria, podremos seleccionar una plantilla de un asistente con la información generada de una anterior sesión con este asistente; algo similar a los ficheros que guardan la información de instalación de un programa.

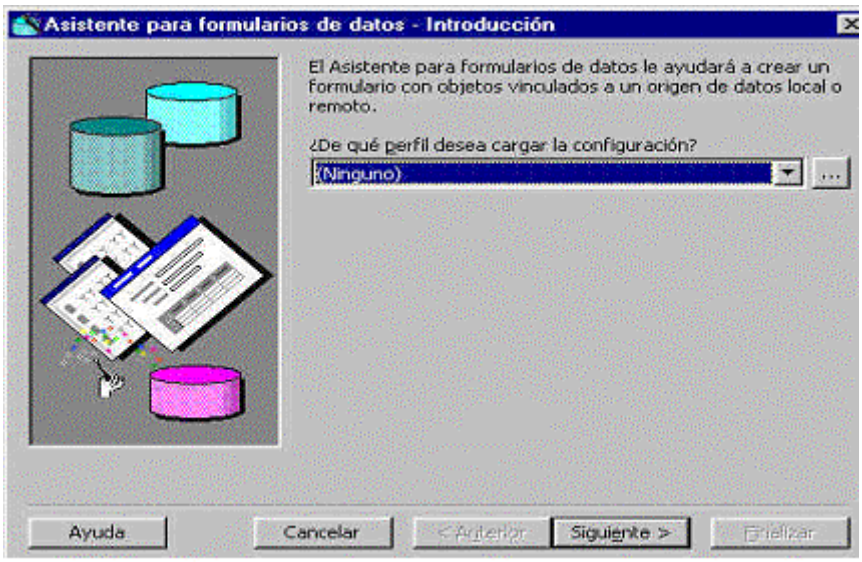


Figura 273. Inicio del asistente para formularios de datos.

A continuación, nos pedirá que seleccionemos el tipo de base de datos a utilizar, elegiremos Access, como podemos observar en la figura 274.

Pulsaremos el botón *Siguiente*, apareciendo la ventana que nos muestra la figura 276, en la que debemos indicar la ruta y nombre de la base de datos.

El siguiente paso consiste en asignar un nombre para el formulario y establecer el modo de edición de los datos. Podemos editar los registros uno a uno, visualizarlos todos en modo de tabla o en formato maestro/detalle, y también mediante los controles MS HFlexGrid o Chart. Para esta ocasión optaremos por la edición simple registro a registro (figura 276).

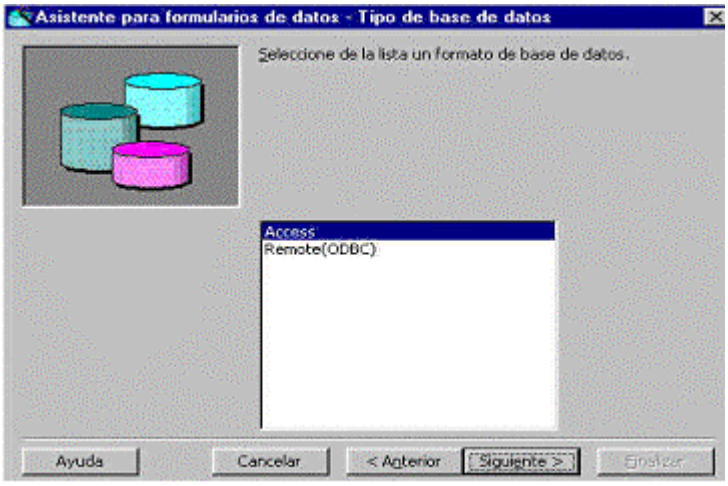


Figura 274. Selección del formato de datos para el asistente de formularios de datos.

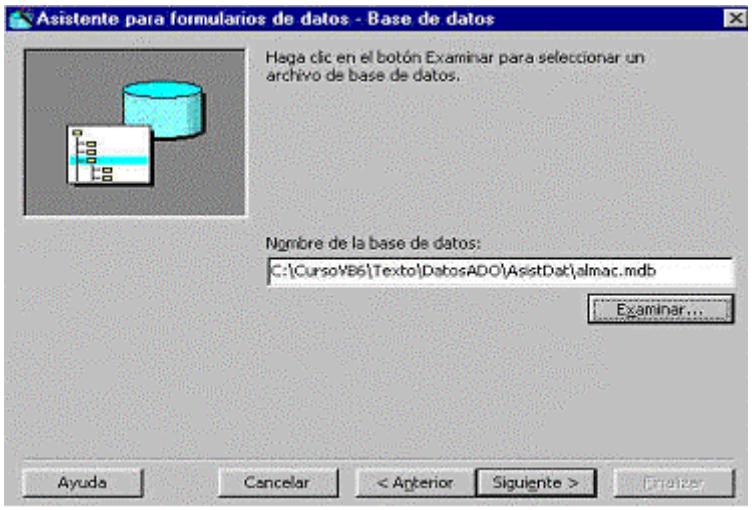


Figura 275. Selección de la base de datos a emplear.

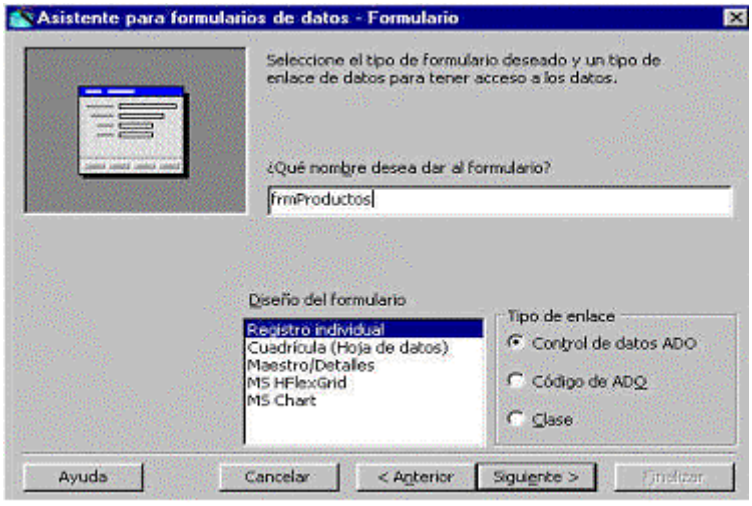


Figura 276. Estilo de edición de los registros para el formulario.

Continuaremos seleccionando una tabla de la base de datos, sobre la que realizaremos la edición de registros en el formulario creado por el asistente. En este ejemplo sólo disponemos de la tabla Productos, que será la utilizada por el formulario.

Después de la tabla, hemos de seleccionar que campos entre los disponibles en la tabla vamos a editar, y cual será el campo que sirva de orden de presentación para los registros.

Figura 277. Selección de los campos a editar en el formulario.

Este asistente insertará en el formulario un grupo de controles para realizar las operaciones más habituales de mantenimiento sobre la tabla. En el siguiente paso del asistente podemos elegir cuales de esos controles aparecerán en nuestro formulario.

Figura 278. Selección de controles a incluir en el formulario de datos.

Como último paso, podemos grabar los pasos dados en este asistente en un fichero, para posteriores usos. Al pulsar el botón *Finalizar* se creará dicho formulario incorporándose al proyecto.

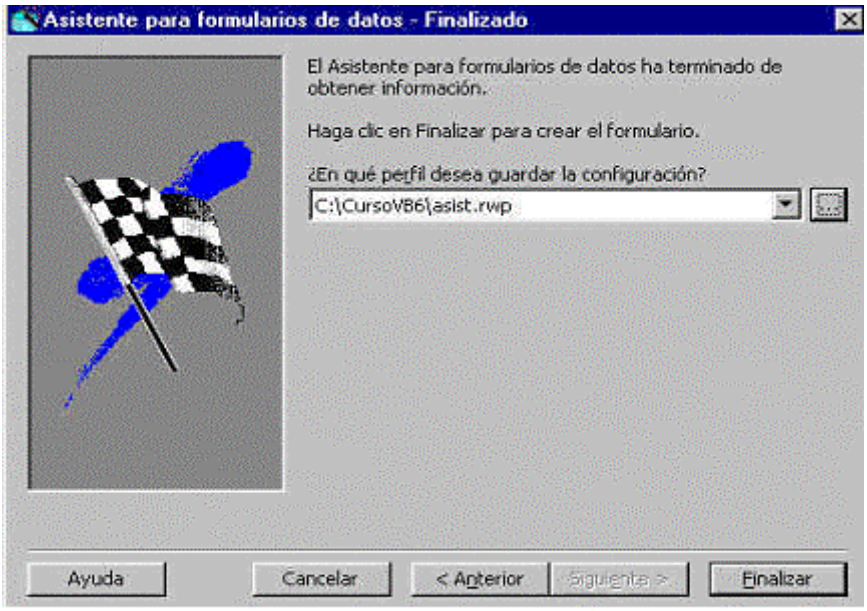


Figura 279. Asignar nombre al formulario y finalizar asistente.

El formulario resultante se muestra en la figura 280. El único paso restante es crear un procedimiento Main() que sirva de entrada a la aplicación y cargue esta ventana.

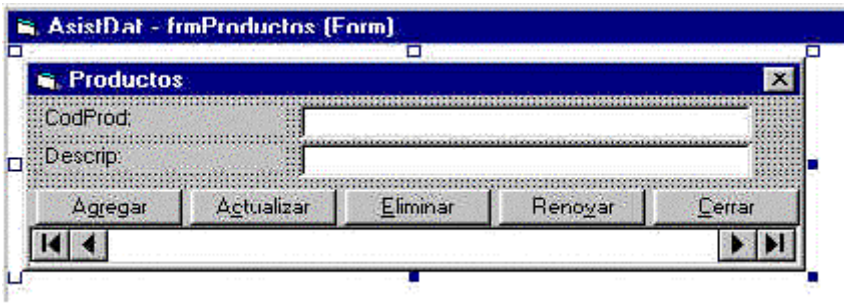


Figura 280. Formulario resultante del asistente.

El lector comprobará revisando el código fuente de este formulario, que se proporciona una funcionalidad muy básica, la cual será necesario completar la mayoría de las ocasiones. Sin embargo, en otras ocasiones puede ser suficiente para mantenimientos en tablas de datos generales, con lo que el ahorro en el tiempo de desarrollo es considerable.

8

Impresión

Técnicas de impresión en Visual Basic

Dejando aparte el uso de las funciones del API de Windows, la impresión desde VB, puede ser realizada mediante el empleo del objeto del sistema Printer, combinándolo con el uso de la colección Printers, para alternar entre las diversas impresoras disponibles, conectadas al ordenador o red.

Otra opción disponible, que será con toda probabilidad la más empleada, consiste en utilizar el diseñador Data Report, especialmente pensado para obtener los datos desde otro diseñador, el Data Environment, de forma flexible y en muy diversas formas.

Seguimos disponiendo de Crystal Reports, el tradicional generador de informes incluido en Visual Basic desde hace algunas versiones, de forma que la migración en nuestros hábitos de confeccionar listados no sea problemática y podamos seguir manteniendo los informes ya creados con esta utilidad, mientras que los nuevos podemos diseñarlos mediante Data Report

Impresión mediante objetos

El uso del objeto del sistema Printer, nos libera de tener que luchar a brazo partido con las funciones del API de Windows encargadas de imprimir. A través del conjunto de propiedades y métodos de Printer, podremos manejar los diferentes aspectos necesarios en la aplicación para imprimir texto, gráficos, uso de fuentes, etc. Esta técnica de trabajo con objetos, como hemos visto en otros temas de este mismo curso, supone una gran ventaja, ya que simplifica en gran medida el desarrollo.

La colección Printers

La colección Printers contiene todos los objetos Printer conectados al sistema. Esto nos permite seleccionar diferentes impresoras durante la ejecución de una aplicación, para imprimir los documentos usando la más apropiada, o bien, realizar una rutina de selección de impresoras, dejando al usuario, la tarea de elegir la que más se adapte al documento que necesita imprimir.

Para visualizar las impresoras conectadas al sistema, sólo hemos de abrir el menú Inicio, en la barra de tareas de Windows y seleccionar la opción Configuración+Impresoras, que nos mostrará la carpeta con las impresoras del sistema.

Figura 281. Carpeta de Impresoras.

El código fuente 240, muestra como podemos emplear la colección Printers, para visualizar las impresoras conectadas en nuestro sistema.

```
Public Sub Main()

Dim loImpresora As Printer
For Each loImpresora In Printers
   MsgBox "Impresora " & loImpresora.DeviceName
Next

End Sub
```

Código fuente 240

El objeto Printer

Como hemos visto anteriormente, este objeto del sistema, contiene la impresora predeterminada de la aplicación. Al iniciar el programa, Printer contiene la impresora predeterminada del sistema. Esta impresora es posible cambiarla durante la ejecución por cualquier otra del sistema, todas las veces que sea necesario. El código fuente 241, muestra una sencilla rutina para cambiar la impresora predeterminada de la aplicación.

```
Public Sub Main()

Dim loImpresora As Printer
Dim lnRespuesta As Integer

For Each loImpresora In Printers
   lnRespuesta = MsgBox("Impresora " & loImpresora.DeviceName & vbCrLf & _
      "¿Seleccionar?", _
      vbYesNo, "Selección de impresora")

   If lnRespuesta = vbYes Then
      Set Printer = loImpresora
   End If

Next

End Sub
```

Código fuente 241

La forma más sencilla de imprimir con el objeto Printer, es utilizando su método Print(), seguido del texto a imprimir. Usaremos tantas llamadas a este método, como líneas necesitemos imprimir. Por último, ejecutaremos el método EndDoc(), lo que dará por terminado la creación del documento a imprimir, y lo enviará a la impresora. El código fuente 242, muestra un ejemplo de esta forma de impresión.

```
Public Sub Main()

' comenzar tarea de impresión
Printer.Print "PRUEBAS CON OBJETO PRINTER"
Printer.Print "--------------------------"
Printer.Print "primera línea"
Printer.Print "segunda línea"
Printer.Print "tercera línea"
Printer.Print "fin de impresión"

' finalizar tarea de impresión,
' enviándola a la impresora
Printer.EndDoc

End Sub
```

Código fuente 242

Como acaba de comprobar el lector, el objeto Printer nos evita toda la complejidad que envuelve la tarea de imprimir en Windows. Para obtener el anterior resultado, VB se ha encargado de crear una tarea de impresión, que se trata de un espacio en memoria en el que se guarda el texto cuando usamos el método Print(). Este método, contrariamente a lo que podamos pensar, no imprime directamente el texto en la impresora, sino que se encarga de enviar dicho texto a la tarea de impresión, siendo el método EndDoc() el que finaliza la tarea de impresión, enviándola a la impresora.

Cambiar el tipo de fuente

El ejemplo anterior imprime el texto utilizando el fuente por defecto del objeto Printer. Sin embargo Printer dispone de la propiedad Font, que no es otra cosa que un objeto de la clase Font, el cual podemos configurar para que la impresión se realice con el fuente, tamaño y resto de características del tipo de letra que necesitemos.

Veamos, en le código fuente 243, como quedaría el ejemplo anterior, si previamente establecemos un tipo de fuente, tamaño y estilo cursiva.

```
Public Sub Main()
' cambiar propiedades del tipo de fuente
' para el objeto impresora
Printer.Font.Name = "Artistik"
Printer.Font.Size = 20
Printer.Font.Italic = True
' comenzar tarea de impresión
Printer.Print "PRUEBAS CON OBJETO PRINTER"
Printer.Print "-------------------------"
Printer.Print "primera línea"
Printer.Print "segunda línea"
Printer.Print "tercera línea"
Printer.Print "fin de impresión"
' finalizar tarea de impresión,
' enviándola a la impresora
Printer.EndDoc
End Sub
```

Código fuente243

Seleccionar la impresora y el tipo de letra mediante el control CommonDialog

Los medios de configurar la impresión empleados hasta el momento, si bien cumplen con su función didáctica, enseñando al lector la forma de cambiar la impresora predeterminada de la aplicación, y el tipo de fuente con el que vamos a imprimir, son claramente insuficientes de cara al desarrollo de un programa. ¿Que alternativa tenemos?, la primera que podríamos pensar, es construir los correspondientes formularios para que fuera el mismo usuario de nuestro programa el que los configurara a su gusto.

Sin embargo, este trabajo no va a ser necesario, ya que el control CommonDialog proporcionado en VB, permite configurar tanto estos, como otros aspectos generales para todas las aplicaciones Windows.

La aplicación Imprimir, contenida en los ejemplos, muestra la forma de utilizar este control para adaptarlo a las necesidades de impresión del programa.

El proyecto incluye un formulario denominado frmImprimir, que contiene un control CommonDialog, llamado dlgImprimir y tres CommandButton, encargados de establecer la impresora de la aplicación, el tipo de fuente e imprimir un texto, usando el objeto Printer, respectivamente. La figura 282, muestra este formulario en modo de diseño.

Figura 282. Formulario frmImprimir, para realizar pruebas de impresión.

El código fuente 244, contiene el código para cada uno de los botones del formulario. Observe el lector, como mediante la propiedad Flags, del control CommonDialog, adaptamos algunas características del cuadro estándar mostrado en cada ocasión.

```
Private Sub cmdFuente_Click()
' configurar el cuadro de diálogo estándar
' de selección de fuente:
' - muestra las fuentes disponibles para pantalla e impresora
' - muestra las opciones de efectos disponibles para
' la fuente
Me.dlgImprimir.Flags = cdlCFBoth Or cdlCFEffects
' mostrar cuadro de diálogo estándar
' para seleccionar fuentes
Me.dlgImprimir.ShowFont
' después de aceptar el cuadro de
' diálogo, tomar las características
' de la fuente seleccionada y pasarlas
' al objeto Printer
Printer.Font.Name = Me.dlgImprimir.FontName
Printer.Font.Size = Me.dlgImprimir.FontSize
Printer.Font.Bold = Me.dlgImprimir.FontBold
Printer.Font.Italic = Me.dlgImprimir.FontItalic
Printer.Font.Strikethrough = Me.dlgImprimir.FontStrikethru
Printer.Font.Underline = Me.dlgImprimir.FontUnderline
End Sub

' ----------------------------------
Private Sub cmdSeleccionar_Click()
' configurar el cuadro de diálogo estándar
' de selección de impresora:
' - oculta el CheckBox de envio a fichero la salida
' de la impresora
' - desactiva el OptionButton para imprimir la selección
' efectuada
Me.dlgImprimir.Flags = cdlPDHidePrintToFile Or cdlPDNoSelection
' mostrar cuadro de diálogo estándar
' para seleccionar una de las impresoras
' conectadas al sistema
Me.dlgImprimir.ShowPrinter
' después de aceptar este cuadro de
' diálogo, la impresora seleccionada
' es establecida automáticamente por VB
End Sub
```

```
' --------------------------------
Private Sub cmdImprimir_Click()
' comenzar tarea de impresión
Printer.Print "PRUEBAS CON OBJETO PRINTER"
Printer.Print "-------------------------"
Printer.Print "primera línea"
Printer.Print "segunda línea"
Printer.Print "tercera línea"
Printer.Print "fin de impresión"
' finalizar tarea de impresión,
' enviándola a la impresora
Printer.EndDoc
End Sub
```

Código Fuente 244

Otras propiedades y métodos del objeto Printer

Aparte de los métodos básicos del objeto Printer comentados hasta el momento, existe un conjunto más amplio de propiedades y métodos para este objeto, algunos de los cuales vamos a comentar brevemente a continuación. El lector puede recabar información adicional consultando la ayuda de VB referente a este objeto.

Propiedades

- Copies. Devuelve o establece en número de copias a imprimir del documento.
- Page. Devuelve el número de página actualmente en impresión.
- PaperBin. Devuelve o establece un valor, que informa de la bandeja de papel utilizada por la impresora. Es posible emplear las siguientes constantes mostradas en la tabla 77.

Constante	**Valor**	**Descripción**
VbPRBNUpper	1	Utilizar papel de la bandeja superior.
VbPRBNLower	2	Utilizar papel de la bandeja inferior.
VbPRBNMiddle	3	Utilizar papel de la bandeja intermedia.
VbPRBNManual	4	Esperar a la inserción manual de cada hoja.
VbPRBNEnvelope	5	Utilizar sobres del alimentador de sobres.
vbPRBNEnvManual	6	Utilizar sobres del alimentador de sobres, pero esperar a la inserción manual.
VbPRBNAuto	7	(Predeterminado) Utilizar papel de la bandeja predeterminada actual.
VbPRBNTractor	8	Utilizar papel del alimentador por tracción.

VbPRBNSmallFmt	9	Utilizar papel del alimentador de papel pequeño.
VbPRBNLargeFmt	10	Utilizar papel de la bandeja de papel grande.
vbPRBNLargeCapacity	11	Utilizar papel del alimentador de gran capacidad.
VbPRBNCassette	14	Utilizar papel del cartucho de cassette adjunto.

Tabla 77. Constantes disponibles para la propiedad PaperBin.

- PaperSize. Devuelve o establece el tamaño de papel que utilizará la impresora. La tabla 78 muestra las diferentes constantes disponibles para esta propiedad.

Constante	**Valor**	**Descripción**
VbPRPSLetter	1	Carta, 216 x 279 mm
VbPRPSLetterSmall	2	Carta pequeña, 216 x 279 mm
VbPRPSTabloid	3	Tabloide, 279 x 432 mm
VbPRPSLedger	4	Doble carta, 432 x 280 mm
VbPRPSLegal	5	Oficio, 216 x 356 mm
VbPRPSStatement	6	Estamento, 140 x 216 mm
VbPRPSExecutive	7	Ejecutivo, 190 x 254 mm
vbPRPSA3	8	A3, 297 x 420 mm
vbPRPSA4	9	A4, 210 x 297 mm
vbPRPSA4Small	10	A4 pequeño, 210 x 297 mm
vbPRPSA5	11	A5, 148 x 210 mm
vbPRPSB4	12	B4, 250 x 354 mm
vbPRPSB5	13	B5, 182 x 257 mm
VbPRPSFolio	14	Folio, 216 x 330 mm
VbPRPSQuarto	15	Cuarto, 215 x 275 mm
vbPRPS10x14	16	254 x 356 mm
vbPRPS11x17	17	279 x 431 mm
VbPRPSNote	18	Nota, 216 x 279 mm

VbPRPSEnv9	19	Sobre Nº 9, 98 x 225 mm
VbPRPSEnv10	20	Sobre Nº 10, 105 x 241 mm
VbPRPSEnv11	21	Sobre Nº 11, 114 x 264 mm
VbPRPSEnv12	22	Sobre Nº 12, 114 x 279 mm
VbPRPSEnv14	23	Sobre Nº 14, 127 x 292 mm
VbPRPSCSheet	24	Hoja tamaño C
VbPRPSDSheet	25	Hoja tamaño D
VbPRPSESheet	26	Hoja tamaño E
VbPRPSEnvDL	27	Sobre DL, 110 x 220 mm
vbPRPSEnvC3	29	Sobre C3, 324 x 458 mm
vbPRPSEnvC4	30	Sobre C4, 229 x 324 mm
vbPRPSEnvC5	28	Sobre C5, 162 x 229 mm
vbPRPSEnvC6	31	Sobre C6, 114 x 162 mm
vbPRPSEnvC65	32	Sobre C65, 114 x 229 mm
vbPRPSEnvB4	33	Sobre B4, 250 x 353 mm
vbPRPSEnvB5	34	Sobre B5, 176 x 250 mm
vbPRPSEnvB6	35	Sobre B6, 176 x 125 mm
VbPRPSEnvItaly	36	Sobre, 110 x 230 mm
VbPRPSEnvMonarch	37	Sobre monarca, 98 x 190 mm
VbPRPSEnvPersonal	38	Sobre, 6 3/4 (92 x165 mm).
VbPRPSFanfoldUS	39	Continuo USA estándar, 310 x 280 mm
VbPRPSFanfoldStdGerman	40	Continuo alemán estándar, 216 x 305 mm
VbPRPSFanfoldLglGerman	41	Continuo alemán oficio, 216 x 330 mm
VbPRPSUser	256	Definido por el usuario

Tabla 78. Constantes disponibles para la propiedad PaperSize.

- PrintQuality. Devuelve o establece el tipo de resolución para la impresora. La tabla 79 muestra la relación de constantes disponibles para esta propiedad.

Constante	Valor	Descripción
VbPRPQDraft	-1	Resolución borrador
VbPRPQLow	-2	Resolución baja
VbPRPQMedium	-3	Resolución media
vbPRPQHigh	-4	Resolución alta

Tabla 79. Constantes disponibles para la propiedad PrintQuality.

- TrackDefault. Valor lógico que indica, si es verdadero, que el objeto Printer cambia al seleccionar el usuario una nueva impresora desde el panel de configuración del sistema operativo. En el caso de que sea falso, el objeto Printer no cambiará, manteniendo el mismo valor de impresora.

Métodos

- KillDoc(). Finaliza la tarea de impresión inmediatamente, la impresora deja de recibir información, cancelando la impresión. En el caso de que el administrador de impresión no esté controlando la impresión en segundo plano, es posible que la impresora siga trabajando antes de poder cancelar su trabajo.

- NewPage(). Finaliza la página actualmente en impresión, avanzando hasta la nueva página de Printer, el valor de la propiedad Page se incrementa en uno.

- PaintPicture(). Envía un gráfico a la impresora.

Data Report (Diseñador de informes)

Como comentábamos al principio de este tema, Data Report es el diseñador de informes que se incluye en VB, siendo el medio recomendado para crear los listados de nuestras aplicaciones.

El diseñador Data Report nos proporciona un elevado grado de control sobre todos los aspectos relacionados con la confección de informes, liberándonos al mismo tiempo de las complejidades del uso del objeto Printer. Printer nos proporciona un control total sobre la impresión, pero ese mismo control, nos obliga también a realizar un esfuerzo extra para controlar hasta el más mínimo detalle en el trabajo de impresión.

Con el diseñador de informes, enfocamos nuestro trabajo en lo verdaderamente importante: el diseño del informe en cuanto a la distribución de la información obtenida de un origen de datos. Para ello, este diseñador se emplea conjuntamente con otro diseñador: Data Environment, que es el que proporciona la información al listado. El programador sólo debe ocuparse del manejo de los campos de las tablas, campos especiales, totales, secciones del informe, etc.; el resto de los detalles de

impresión, que empleando el objeto Printer deberíamos de controlar, es tarea que realiza internamente Data Report.

Elementos del diseñador de informes

El diseñador de informes está compuesto por varios objetos: el objeto principal, DataReport, que representa a la plantilla o base del informe, y contiene el resto de objetos involucrados en un listado. Un objeto DataReport, contiene a su vez varios objetos Section, que representan a las diferentes secciones del informe. Finalmente disponemos de varios controles especiales, diseñados para su uso exclusivo en el diseñador de informes, y que mostrarán los campos de la base de datos, texto normal, cálculos sobre los registros, imágenes, etc. Todos ellos, junto a una descripción de sus características, serán tratados en los siguientes apartados.

El objeto DataReport

Este es el objeto principal, que servirá como contenedor al resto de objetos que componen el diseñador. Se asemeja en gran medida a un formulario, ya que dispone de una parte visual, en la que el programador puede configurar los diferentes elementos que componen el informe; y un módulo de código, en el que es posible escribir acciones a realizar en respuesta a los distintos eventos de este objeto.

Propiedades

- AsyncCount. La impresión del informe es una tarea realizada de forma asíncrona, esta propiedad nos devuelve el número de operaciones pendientes de procesar por parte del informe, de forma que podamos optar por cancelar la impresión o continuar hasta que finalice.

- BottonMargin, TopMargin. Asigna o devuelve los márgenes superior e inferior del informe.

- DataMember. Asigna o devuelve el miembro de datos perteneciente a la fuente de datos establecida para el informe.

- DataSource. Asigna o devuelve la fuente de datos a la que está conectada el informe, que será un DataEnvironment existente en el proyecto.

- ExportFormats. Colección de objetos ExportFormat, que contienen la información de formato sobre un informe que se va a exportar.

- GridX, GridY. Establece el número de divisiones en la cuadrícula del diseñador para situar los controles. Cuanto mayor sea el número, dispondremos de una mayor precisión a la hora de colocar los controles en las diferentes secciones del informe.

- LeftMargin, RightMargin. Asigna o devuelve los márgenes izquierdo y derecho del informe.

- ReportWidth. Asigna o devuelve el ancho del informe.

- Sections. Colección de objetos Section, que contienen la información sobre las secciones del informe.

- Title. Asigna o devuelve el título del informe. Este título será utilizado por un control TextBox o Label del informe, que contenga la cadena "%i"; o en un informe exportado, en el que el valor del título sea la constante rptTagTitle.

Métodos

- ExportReport(). Exporta el contenido del informe a un formato determinado, sin incluir imágenes ni formas gráficas contenidas en el informe.

  Sintaxis.

  ```
  ExportReport(xFormatoExp, cFichero, bSobrescribir, bVerDlg,
  nIntervalo, nDesdePag, nHastaPág)
  ```

  - xFormatoExp. Un objeto ExportFormat o una constante indicando el tipo de formato, según la tabla 80.

Constante	Valor	Descripción
rptKeyHTML	key_def_HTML	Especifica el miembro predeterminado HTML de la colección ExportFormats.
rptKeyUnicodeHTML	key_def_Unicode HTML_UTF8	Especifica el miembro predeterminado HTML de Unicode de la colección ExportFormats.
rptKeyText	key_def_Text	Especifica el miembro predeterminado de texto de la colección ExportFormats.
rptKeyUnicodeText	key_def_Unicode Text	Especifica el miembro predeterminado de texto Unicode de la colección ExportFormats.

Tabla 80. Constantes disponibles para indicar el tipo de formato a emplear para exportar un informe.

  - cFichero. Nombre del fichero en el que se depositará el informe exportado.

  - bSobrescribir. Valor lógico que indica si se sobrescribirá el archivo en el caso de que exista (True) o no (False).

  - bVerDlg. Valor lógico que indica si se mostrará al usuario el cuadro de diálogo estándar Guardar como (True) o no (False). Si no se establecen valores en las propiedades xFormatoExp o cFichero, se mostrará el diálogo.

  - nIntervalo. Constante para indicar si se imprimirán todas las páginas del informe o sólo un determinado rango. Veamos la tabla 81

Constante	Valor	Descripción
rptRangeAllPages	0	(Predeterminado) Se imprimen todas las páginas.
rptRangeFromTo	1	Tan sólo se exporta el intervalo de páginas especificado.

Tabla 81. Constantes para la propiedad nIntervalo.

- nDesdePag, nHastaPag. Valores numéricos para establecer la página de comienzo y la de final para el informe.

- PrintReport(). Imprime el informe.

  Sintaxis.

```
PrintReport(bVerDlg, nIntervalo, nDesdePag, nHastaPag)
```

  - bVerDlg. Valor lógico que indica si se mostrará al usuario el cuadro de diálogo estándar Imprimir (True) o no (False).

  - nIntervalo. Constante para indicar si se imprimirán todas las páginas del informe o sólo un determinado rango. Consultar la propiedad del mismo nombre en el método ExportReport()

  - nDesdePag, nHastaPag. Valores numéricos para establecer la página de comienzo y la de final para el informe.

Eventos

- AsyncProgress(). Este evento tiene lugar cada vez que se procesa una página de datos.

  Sintaxis.

```
AsyncProgress(kTarea As AsyncTypeConstants, lCookie As Long,
lPagCompletada As Long, lTotalPag As Long)
```

Constante	Valor	Descripción
rptAsyncPreview	0	El informe está procesando una operación de presentación preliminar.
rptAsyncPrint	1	El informe está procesando una operación PrintReport.
rptAsyncReport	2	El informe está procesando una operación ExportReport.

Tabla 82. Constantes para la propiedad kTarea.

  - kTarea. Constante que indica el tipo de tarea realizada por el diseñador, según los valores de la tabla 82.

- lCookie. Un valor numérico que indica el identificador de la operación.
- lPagCompletada. Número de páginas completadas.
- lTotalPag. Número de páginas que se van a procesar.

- AsyncError(). Este evento sucede cuando se produce un error en la operación actual.

  Sintaxis.

  ```
  AsyncError(kTarea As AsyncTypeConstants, lCookie As Long,
  oErr As RptError, lMostrarError As Boolean)
  ```

  - kTarea. Constante que indica el tipo de tarea realizada por el diseñador. Consultar el evento AsyncProgress() para más información sobre las constantes de este parámetro.
  - lCookie. Un valor numérico que indica el identificador de la operación.
  - oErr. Un objeto Error con la información sobre el error ocurrido.
  - lMostrarError. Valor lógico que indica si se visualizará el mensaje de error (True) o no (False).

- ProcessingTimeout(). Sucede cada ciertos periodos de tiempo, y sirve para permitir al usuario cancelar una operación en curso.

  Sintaxis.

  ```
  ProcessingTimeout(nSegundos As Long, bCancelar As Boolean,
  kTarea As AsyncTypeConstants, lCookie As Long)
  ```

  - nSegundos. Número de segundos transcurridos desde el comienzo de la operación que se está llevando a cabo.
  - bCancelar. Valor lógico que indica si se cancela la operación (True) o no (False). El valor predeterminado es False.
  - kTarea. Constante que indica el tipo de tarea realizada por el diseñador. Consultar el evento AsyncProgress() para más información sobre las constantes de este parámetro.
  - lCookie. Un valor numérico que contiene el identificador de la operación.

El objeto Section

Un DataReport contiene a su vez varios objetos Section. Un objeto Section (sección del informe), es aquella parte del informe destinada a incluir una determinada clase de información. Los objetos Section que se incluyen por defecto en todo listado, son los siguientes:

- Encabezado de informe. Muestra texto que aparecerá una única vez en la ejecución del informe: al iniciarse el mismo.

- Encabezado de página. Muestra texto que aparecerá al imprimirse una nueva página del informe.
- Líneas de detalle. Se encarga de mostrar las filas o registros que componen el informe.
- Pie de página. Muestra información al final de cada página del informe.
- Pie de informe. Los datos de esta sección será mostrados al final de la impresión del informe.

Adicionalmente, podremos incluir parejas de secciones Encabezado/Pie de grupo, que estarán asociadas con un objeto Command del entorno de datos al que está conectado el diseñador de informes, y nos servirán para realizar agrupaciones de registros con el mismo valor en el informe.

Propiedades

- ForcePageBreak. Asigna o devuelve un valor que indica el modo en que se efectuará un salto de página. Podemos ver los valores disponibles en la tabla 83.

Constante	Valor	Descripción
rptPageBreakNone	0	(Predeterminado) No se produce ningún salto de página.
rptPageBreakBefore	1	Ocurre un salto de página antes de la sección actual.
rptPageBreakAfter	2	Ocurre un salto de página después de la sección actual.
rptPageBreakBeforeAndAfter	3	Ocurren saltos de página tanto antes como después de la sección actual.

Tabla 83. Constantes disponibles para la propiedad kTarea.

- KeepTogether. Asigna o devuelve un valor lógico que indica si el texto de una sección será dividido o no, al realizar un salto de página.

Controles del diseñador de informes

El diseñador de informes, dispone de un conjunto de controles expresamente diseñados para emplear en la creación de informes, no estando disponibles para utilizar en formularios. De la misma forma, los controles que habitualmente empleamos en formularios, estarán deshabilitados mientras nos encontremos en la fase de diseño de un informe. Los controles especiales para el diseñador de informes, se encuentran en la ficha DataReport del Cuadro de herramientas de VB, tal como muestra la figura 283.

Figura 283. Controles para el diseñador de informes.

A continuación vemos una breve descripción de cada control, en el orden en que aparecen en el Cuadro de herramientas.

- RptLabel. Al igual que el Label estándar, este control muestra un literal en el informe. Su uso más habitual es en la visualización de los nombres de las columnas del informe, número de página, título del informe, fecha, etc.

- RptTextBox. Se utiliza para mostrar los valores de los campos contenidos en el objeto Command al que está conectado el informe. A diferencia del TextBox empleado en formularios, no es posible la edición del valor de este control en tiempo de ejecución

- RptImage. Permite incluir en el informe, un fichero gráfico de alguno de los formatos de imágenes más habituales.

- RptLine. Dibuja una línea en el informe.

- RptShape. Dibuja una forma gráfica tal como un cuadrado, rectángulo, círculo, etc.

- RptFunction. Realiza un cálculo sobre un conjunto de registros del informe.

Entre las propiedades destacables de estos controles, mencionaremos las siguientes:

- CanGrow. Valor lógico que indica que el control puede aumentar su tamaño cuando el valor pasado es mayor que el tamaño definido en el informe (True), o que no puede aumentar su tamaño (False).

  Esta propiedad es aplicable en los controles RptFunction, RptLabel y RptTextBox.

- SizeMode. Constante que indica el modo de ajuste de una imagen dentro de un control RptImage, según los valores de la tabla 84.

Constante	**Valor**	**Descripción**
rptSizeClip	0	Se recortará la imagen.
rptSizeStretch	1	Se estirará la imagen para ajustarse al control.
rptSizeZoom	2	También se estira la imagen, pero la proporción X:Y de la imagen permanece constante.

Tabla 84. Constantes para la propiedad SizeMode.

- LineSlant. Constante que indica la orientación de un control RptLine contenido en un informe.

Constante	Valor	Descripción
rptSlantNWSE	0	La línea se inclina desde el ángulo superior izquierdo al inferior derecho (orientación noroeste-sureste).
rptSlantNESW	1	La línea se inclina desde el ángulo superior derecho al inferior izquierdo (orientación noreste-suroeste).

Tabla 85. Constantes para la propiedad LineSlant.

Una vez dibujado un control RptLine, el mismo tendrá siempre un grado de inclinación en función de esta propiedad. Para conseguir que la línea se muestre recta, existe un truco que consiste en asignar a la propiedad Height de este control, el valor 1, de forma que se mostrará totalmente recta.

Como nota negativa en cuanto a los controles que manipulan formas en un diseñador de informes, nos encontramos con el hecho de que no disponen de la propiedad BorderWidth, por lo que una vez incluidos en el informe, siempre mantendrán el mismo grosor, que en algunas ocasiones puede resultar insuficiente.

- Shape. Asigna o devuelve una constante que sirve para establecer la forma de un control RptShape

Constante	Valor	Descripción
vbShapeRectangle	0	(Predeterminado) Rectángulo
vbShapeSquare	1	Cuadrado
vbShapeOval	2	Elipse
vbShapeCircle	3	Círculo
vbShapeRoundedRectangle	4	Rectángulo redondeado
vbShapeRoundedSquare	5	Cuadrado redondeado

Tabla 86. Constantes para las formas disponibles de la propiedad Shape.

Crear un informe

La aplicación DataRep, será la encargada de ilustrar los diferentes aspectos en la creación de informes con el diseñador Data Report. Hace uso de la base de datos TelePlato.MDB, que contiene la

información que una supuesta empresa de comida rápida guarda sobre sus clientes, productos y los encargos que reciben. La figura 284, muestra el diagrama de esta base de datos.

Figura 284. Diagrama de la base de datos TelePlato.

Comenzaremos nuestra andadura por el diseñador de informes, confeccionando un sencillo informe que muestre los registros de la tabla Clientes. Para ello daremos los siguientes pasos:

Crear la fuente de datos

En primer lugar, crearemos un entorno de datos en el proyecto con el nombre dteTelePlato. Evitaremos repetir los pasos de creación de un objeto Command, puesto que este objeto y el diseñador Data Environment, fueron explicados en el tema Tratamiento de datos con ActiveX Data Objects, apartado El Entorno de Datos (Data Environment). Para cualquier duda sobre este particular, remitimos al lector a dicho tema.

Continuaremos agregando un objeto Connection (acnTelePlato) al entorno de datos, que configuraremos para que conecte con la base de datos antes mencionada. Finalizaremos la configuración de la fuente de datos agregando un objeto Command que abra la tabla Clientes.

Agregar un diseñador de informes al proyecto

Seleccionaremos la opción Proyecto+Agregar Data Report, en el menú de VB o en el menú contextual del Explorador de proyectos, la opción Agregar+Data Report, lo que incluirá un nuevo diseñador de informes en nuestro proyecto.

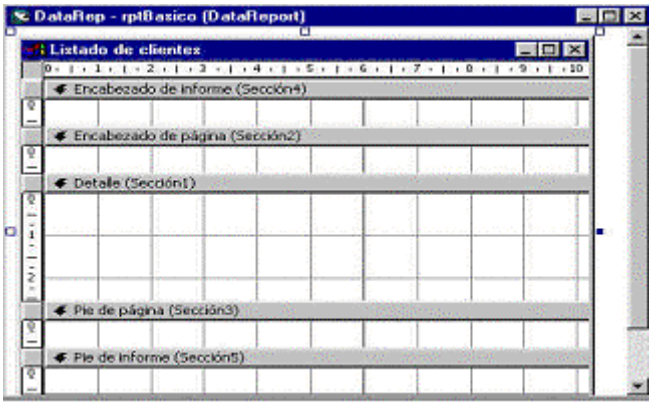


Figura 285. Nuevo diseñador de informes incorporado al proyecto.

Conectar el informe con una fuente de datos

Abriremos la ventana de propiedades del diseñador para asignarle el nombre rptBasico. En la propiedad DataSource asignaremos el entorno de datos creado anteriormente, dteTelePlato, y como DataMember estableceremos el comando que abre la tabla Cliente, acmClientes.

Una vez establecidas las propiedades referentes a los datos que mostrará el informe, abriremos su menú conextual, seleccionando la opción Obtener estructura, cuyo trabajo será tomar el objeto Command de la propiedad DataMember del diseñador de informes, incorporando su jerarquía de datos a las diferentes secciones del informe.

Antes de proceder a tomar la información del objeto Command, se nos avisará que la jerarquía de datos existente en el informe será eliminada. Después de responder afirmativamente, el diseñador tomará incorporará la jerarquía de datos del comando acmClientes, y asignará los nombres a las secciones del informe. Estos nombres son asignados por defecto, pudiendo modificarse en la ventana de propiedades del diseñador.

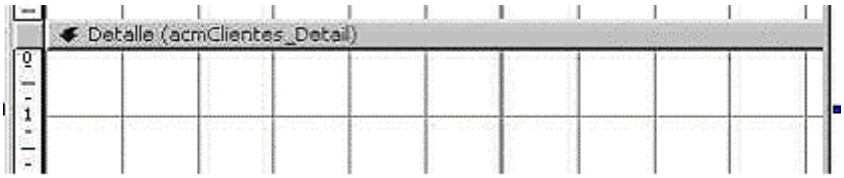


Figura 286. Sección de detalles del informe después de incorporar un comando.

El lector no apreciará por el momento ninguna ventaja especial en el uso de la opción Obtener estructura. Será en la creación de informes con registros agrupados, cuando valoraremos en su justa medida, las ventajas que nos ofrece esta opción.

Insertar campos en el informe

Esta labor es posible realizarla de dos maneras: manualmente o arrastrando y soltando desde el entorno de datos.

En primer lugar y para facilitar el trabajo de manipular los controles de forma precisa en el diseñador, asignaremos a las propiedades GridX y GridY, el valor 10. Con esto conseguimos que el tamaño de la cuadrícula para situar los controles sea más preciso, ya que por defecto el valor de estas propiedades es 1, lo que hace prácticamente imposible situar los controles de una forma óptima en el informe.

Para establecer los controles manualmente, tomaremos del cuadro de herramientas un RptTextBox y un RptLabel por cada campo de la base de datos a mostrar. El primero lo situaremos en la sección Detalles y el segundo en la sección Encabezado de página.

En cuanto a las propiedades de estos controles; para el RptTextBox, estableceremos el comando acmClientes en la propiedad DataMember, y el campo correspondiente de dicha tabla en la propiedad DataField. En el RptLabel asignaremos una cadena con el nombre del campo a mostrar en su propiedad Caption. La propiedad Nombre de ambos controles no es obligatorio asignarla, pero en el caso del RptTextBox sí es conveniente, de forma que si necesitamos acceder a este control en el código, podamos identificarlo más fácilmente por el campo del que presenta los datos.

La figura 287, muestra el resultado al incorporar dos campos manualmente.

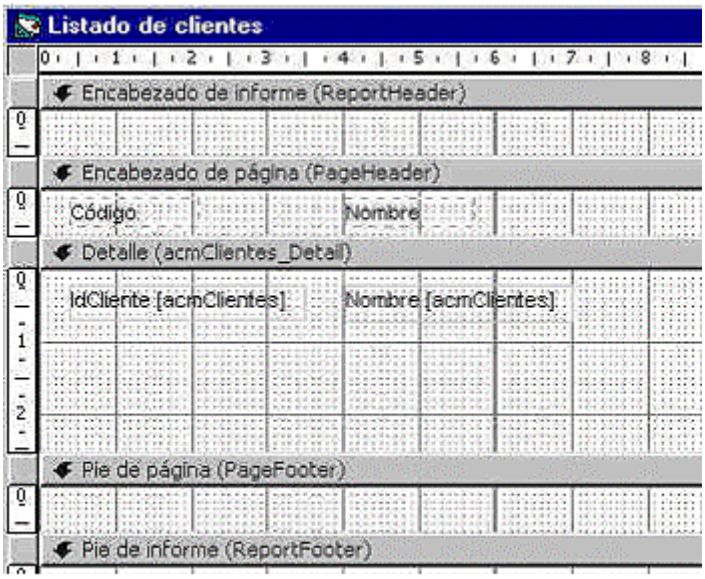


Figura 287. Diseño del informe con campos agregados manualmente.

Si empleamos la técnica de arrastrar y soltar, se facilita en mayor medida nuestro trabajo. En primer lugar seleccionaremos el comando acmClientes del Data Environment y lo arrastraremos hasta la sección Detalles del Data Report. Los controles necesarios serán creados automáticamente, pero es muy posible que necesitamos hacerles ciertos retoques, ya que el tamaño puede no ser adecuado; la disposición en el informe es en línea y debemos mostrarlos en columna; y todos los controles se depositan en la sección Detalles.

Por ejemplo, los RptLabel deberemos cambiarlos a la sección Encabezado de página. Igualmente necesitaremos cambiar su propiedad Caption por un nombre más adecuado, ya que por defecto los RptLabel muestran el nombre del campo de la tabla.

Previsualizar el informe

En este momento, nuestro informe ya está listo para ser impreso, aunque mientras estemos en su fase de diseño, emplearemos mayormente el modo de previsualización ejecutando el método Show(). El código fuente 245, pertenece a la opción Informes+Básico del menú, incluido en el programa de ejemplo, que instancia el objeto informe que hemos creado y lo muestra al usuario.

```
Private Sub mnuInBasico_Click()
Dim lrptBasico As rptBasico
Set lrptBasico = New rptBasico
Load lrptBasico
lrptBasico.Show
End Sub
```

Código fuente 245

Efectuada nuestra primera prueba con el informe, a pesar de haber sido satisfactoria, hemos de hacer algunas observaciones:

El espacio de separación entre cada línea del informe es muy grande. Esto lo solucionaremos, haciendo clic sobre el nombre de cada sección en el diseñador de informes, y desplazándola hasta reducirla u ocultarla en el caso de que dicha sección no muestre información. La figura 288 muestra como quedaría el informe, reduciendo las secciones a las estrictamente necesarias.

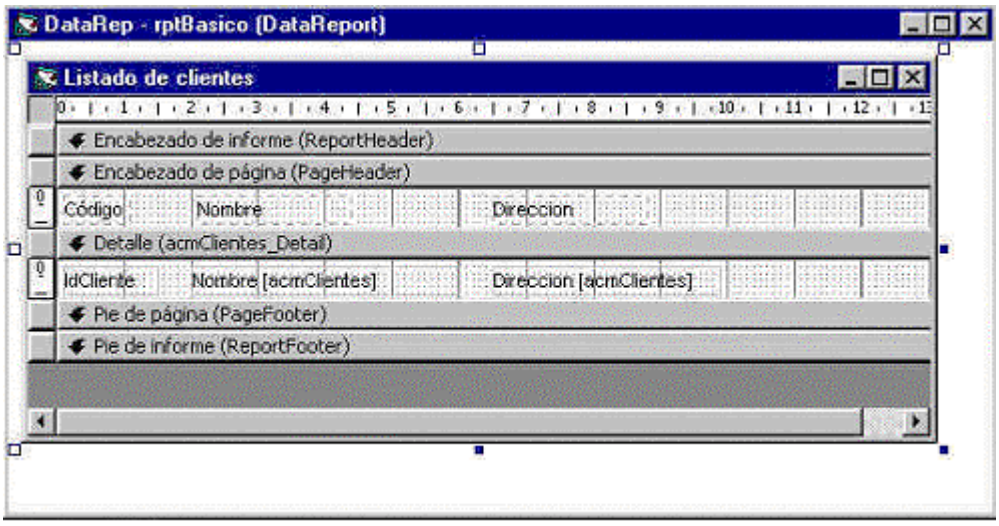


Figura 288. Diseño del informe con ajuste de secciones.

Por otro lado, ya que la ventana de previsualización del informe se comporta como un formulario cualquiera y dispone de las mismas propiedades. Deberemos asignar a su propiedad MDIChild, el valor True para que su comportamiento sea el de ventana secundaria de la ventana principal del programa de ejemplo. Ahora ya podemos volver a ejecutar el informe, que mostraría un aspecto similar al de la siguiente figura al ser previsualizado.

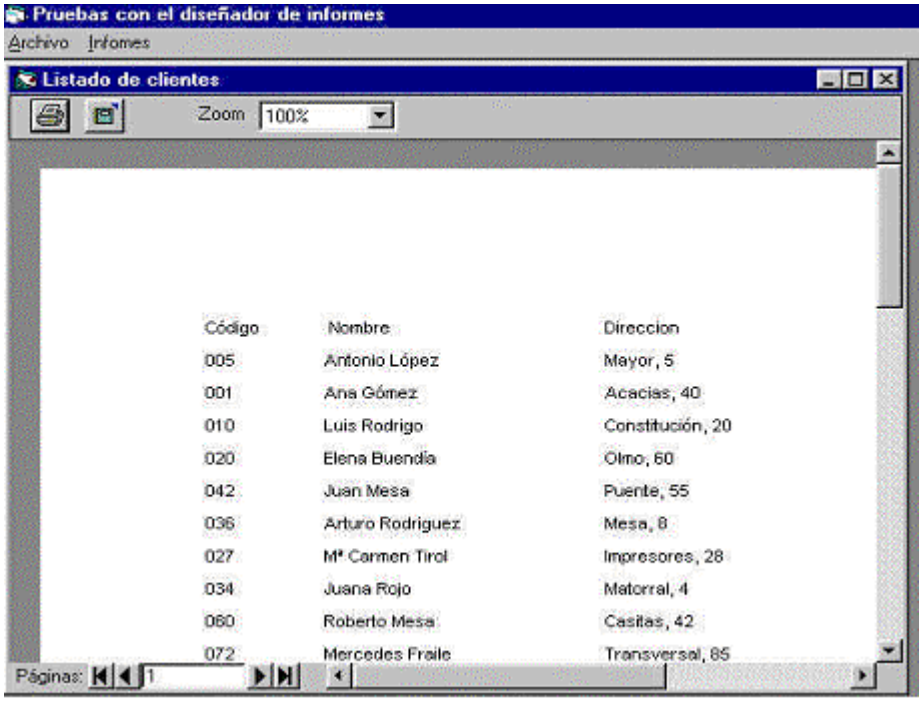


Figura 289. Informe básico en modo de previsualización.

Cancelar el proceso de impresión

Desde el momento que enviamos el informe a imprimir o previsualizar, en función del número de registros a procesar, dicho informe se demorará una cantidad determinada de tiempo. En informes especialmente largos, puede darse el caso de que en pleno proceso del informe, necesitemos variar alguno de sus parámetros para mostrar más registros, o mostrarlos ordenados en diferente forma que la originalmente pensada.

Esto no representa ningún problema debido a que el informe se procesa de forma asíncrona, y VB nos provee de los mecanismos necesarios para cancelarlo en cualquier momento. Concretamente, disponemos del evento ProcessingTimeout() en el objeto DataReport, cuya sintaxis fue explicada en el apartado dedicado a este objeto.

El código fuente 246, muestra un ejemplo de como superado un tiempo desde el inicio del proceso de impresión, se pregunta al usuario la posibilidad de seguir imprimiendo el informe o cancelarlo.

```
Private Sub DataReport_ProcessingTimeout(ByVal Seconds As Long,
 Cancel As Boolean, ByVal JobType As MSDataReportLib.AsyncTypeConstants,
 ByVal Cookie As Long)
Dim lnRespuesta As Integer
' superado el tiempo límite...
If Seconds > 3 Then
   ' preguntamos al usuario si quiere cancelar
   lnRespuesta = MsgBox("Listado en proceso, ¿desea cancelarlo?", _
      vbYesNo, "Preparando listado")

   ' asignando el valor al parámetro Cancel
   ' de este evento, cancelamos el listado
   If lnRespuesta = vbYes Then
      Cancel = True
```

```
   End If

End If
End Sub
```

Código Fuente 246

La ventana de previsualización

Esta ventana es la empleada cuando al desarrollar un informe, decidimos mostrar el resultado del mismo por pantalla antes de enviarlo a la impresora. Contiene los siguientes elementos:

- Datos del informe. Corresponde al área cliente de la ventana, que muestra la información de la base de datos conectada al informe.
- Imprimir. Pulsando este botón  , enviamos el informe a la impresora.
- Exportar. Pulsando este botón  , se muestra un cuadro de diálogo que nos permite guardar el informe en diversos formatos: texto, html, etc.
- Zoom. Mediante este ListBox  , podemos seleccionar el grado de proximidad con el que será mostrado el listado.
- Selector de página. Pulsando los botones de este control  , podremos desplazarlos por las diferentes páginas que componen el informe

Mejorar el aspecto del informe

El informe que acabamos de crear cumple perfectamente con su misión de extraer información de una base de datos, sin embargo, resulta un tanto escaso en lo que a información complementaria y estética se refiere. Vamos a darle unos toques de estilo, que si bien, no afectarán en absoluto a los datos proporcionados, si harán el informe más atractivo al usuario. Esto que puede parecer un aspecto banal a la hora de diseñar un listado, tiene mucha importancia, ya que un informe más atrayente al usuario, facilitará su trabajo en la asimilación de los datos que incluye.

Título

Para establecer el título de un informe, en primer lugar debemos asignar la cadena con dicho título a la propiedad Title del objeto DataReport, en el diseñador de informes. En este caso hemos establecido el literal "Listado de Clientes" a dicha propiedad.

Nos situaremos a continuación en la sección del informe en la que queramos insertar el título. Esta sección por lo general será Encabezado de informe, ya que no tendría sentido situarlo por ejemplo, en las líneas de detalle (aparecería un título por cada línea). Pulsando el botón secundario del ratón, abriremos el menú contextual del diseñador de informes, y seleccionaremos la opción Insertar control+Título del informe, que insertará en la sección en que estemos posicionados un control RptLabel cuya particularidad es que su propiedad Caption contiene el valor "%i; lo que hará que al ejecutarse el informe, dicha cadena sea sustituida por el valor de la propiedad Title del DataReport. La figura 290 muestra el informe con el título incorporado.

Figura 290. Informe mostrando el título.

Nuestro pensamiento más inmediato será: "Si puedo insertar manualmente en el informe un control RptLabel y asignarle allí mismo el título, ¿por qué tanto lío de insertar un control en la forma antes mencionada?". Bien, el motivo de haber establecido el título como se ha explicado anteriormente, es debido a que si necesitamos cambiar el título en tiempo de ejecución, sólo hemos de asignar nuevamente la propiedad Title del DataReport con un título diferente. El código fuente 247 corresponde a la opción del programa que carga el informe rptBasico, modificando el título del mismo.

```
Private Sub mnuInBasico_Click()
Dim lrptBasico As rptBasico
Set lrptBasico = New rptBasico
Load lrptBasico
lrptBasico.Title = "TelePlato, Comida rápida"
lrptBasico.Show
End Sub
```

Código fuente 247

También es cierto que podemos insertar un RptLabel directamente, sin dar los pasos antes explicados, y situarlo en la sección de cabecera del informe como título. El inconveniente que presenta es que si necesitamos cambiar el título, no disponemos de una vía tan directa como la anterior, ya que deberíamos pasar primero por la colección Sections del DataReport y a su vez, por la colección Controls de la sección a modificar; algo parecido a la línea de código que muestra el código fuente 248.

```
lrptBasico.Sections(1).Controls(1).Caption = "Nuevo título"
```

Código fuente 248

Definitivamente más complicado, ya que tenemos que averiguar la sección y el control a modificar, lo que nos empleará más tiempo de pruebas.

Si creamos un título que ocupa mucho espacio o simplemente, queremos mostrar sólo el título en la primera hoja del informe, sólo tenemos que asignar a la propiedad ForcePageBreak de la sección en que está el título, la constante rptPageBreakAfter.

Formas gráficas

Para separar o agrupar lógicamente los elementos de nuestro informe, disponemos de los controles RptLine y RptShape, que nos permiten dibujar líneas, cuadrados, círculos, etc., en el informe.

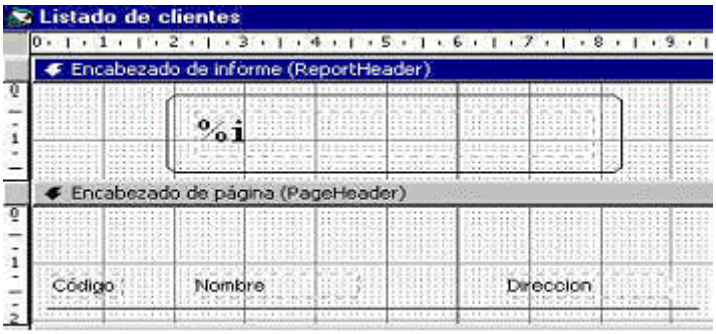


Figura 291. Informe con formas gráficas.

En el informe rptBasico hemos incluido una línea para separar los nombres de las columnas de las filas de datos, y un rectángulo con los bordes redondeados para el título, tal y como vemos en la figura 291.

Imágenes

Insertar un logotipo o imagen en el informe es muy sencillo mediante el control RptImage. Una vez dibujado el control en una sección del informe: encabezado de informe o página, por ejemplo; asignamos a la propiedad SizeMode la constante rptSizeStretch, para que la imagen al ser cargada en el informe no sobrepase el contorno del control, ajustándose a sus límites. Después pasamos a la propiedad Picture, que al ser pulsada, mostrará un cuadro de diálogo para seleccionar un fichero gráfico. Asignaremos el fichero Logo.WMF, que acompaña a esta aplicación, obteniendo el resultado de la figura 292.

Figura 292. Informe incluyendo una imagen.

Número de página

De igual forma que insertamos el título, podemos establecer un control para visualizar el número de página actual y el total de páginas del informe.

Después de situarnos en la sección elegida para albergar esta información, abriremos el menú contextual del diseñador y seleccionaremos la opción Insertar control+Número actual de página y/o Número total de páginas, que agregarán dos controles RptLabel, conteniendo los valores "%p" y "%P" en la propiedad Caption. Al ser ejecutado el informe, tales valores se traducirán en el número de página en curso y el número total de páginas del informe respectivamente.

Otra solución consiste en insertar manualmente un control RptLabel y asignarle a la propiedad Caption una cadena conteniendo los dos valores: "Página %p de un total de %P"

Fecha y hora

Al igual que en el punto anterior, la opción Insertar control del menú contextual del diseñador, nos permite mostrar la fecha y hora en nuestro informe en formato largo y corto.

El modo de operación es el mismo, se agrega un control RptLabel que contiene una cadena especial en su propiedad Caption, que será traducida al valor adecuado:

- %d. Fecha en formato corto.
- %D. Fecha en formato largo.
- %t. Hora en formato corto.
- %T. Hora en formato largo.

En esta ocasión podemos proceder de la misma forma que para el número de página. Insertaremos un RptLabel manualmente y le asignaremos en la propiedad Caption el siguiente valor para el formato largo: "Fecha: %d - Hora: %t". Este control lo situaremos en el encabezado el informe.

En la sección Pie de página del informe haremos igual, pero en este caso mostraremos el formato largo de fecha y hora: "Fecha: %D - Hora: %T".

Presentación de datos agrupados jerárquicamente

El informe creado en los apartados anteriores, rptBasico, en lo referente a manipulación de datos, se limita a realizar una presentación simple de los registros de una tabla. Sin embargo, habrá muchas otras ocasiones en que necesitemos diseñar listados que combinen registros de varias tablas de una base de datos, y a su vez, los registros con valores comunes deberán agruparse para no mostrar valores repetidos.

Tomemos un caso práctico, vamos a crear un informe que muestre los datos de los encargos realizados por los clientes de la base de datos TelePlato. La manera lógica de mostrar la información en un listado sería como se muestra en el esquema de la figura 293.

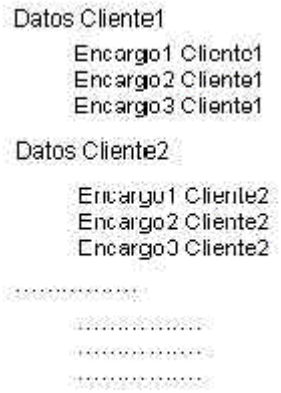


Figura 293. Esquema de organización de datos agrupados en un informe.

En primer lugar, y como vimos en el anterior informe, debemos crear el origen de datos en el Data Environment definido en el proyecto, sólo que esta vez debemos crear dos objetos Command, el primero, al que llamaremos acmGrpClientes obtendrá los registros de la tabla Clientes. A continuación crearemos un comando secundario a partir del que acabamos de crear, al que daremos el nombre acmGrpEncargos, que se relacionará con el primero a través del campo común IdCliente, y que se ocupará de extraer los registros de la tabla Encargos. Para más información de como crear objetos Command relacionados en un entorno de datos, consulte el lector el tema dedicado al tratamiento de datos, apartado El Entorno de Datos (Data Environment). En el entorno de datos del proyecto, dteTelePlato, esta jerarquía de comandos quedará según muestra la figura 294.

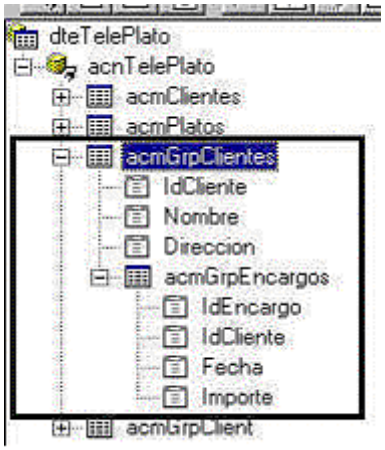


Figura 294. Objetos Command para las tablas Clientes y Encargos, agrupados jerárquicamente.

Seguidamente, agregaremos un diseñador de informes al proyecto, al que daremos el nombre rptAgrupados. Estableceremos sus propiedades de la forma vista anteriormente, pero teniendo en cuenta que al asignar la propiedad DataMember, debemos establecer el comando de más alto nivel de la jerarquía de objetos Command; es decir, si nuestra jerarquía la componen los comandos acmGrpClientes y acmGrpEncargos, elegiremos acmGrpClientes.

Pasaremos después a obtener la estructura de datos, empleando el menú contextual del diseñador, lo que creará nuevos encabezados y pies de grupo, en función del Command establecido como miembro de datos, y los ordenará según el anidamiento que presente dicho Command. La siguiente figura muestra la disposición de las secciones una vez obtenida la estructura.

Ahora sólo resta crear los controles para los campos y literales del informe, o tomar los campos directamente desde el entorno de datos y arrastrarlos hasta el diseñador, para posteriormente colocarlos en las secciones que corresponda. El diseño del informe quedaría como muestra la figura 296.

Para ejecutar este informe, debemos seleccionar la opción de menú Informes+Datos agrupados, que contiene el código fuente 249.

Al visualizar este informe en ejecución, el lector comprobará que los campos con datos de fecha y numéricos muestran un formato especial. Esto es debido a que al igual que sucede con el control TextBox de los formularios, el control RptTextBox permite especificar un formato para el contenido a mostrar, mediante su propiedad DataFormat.

Adicionalmente a este ejemplo, se acompaña en el proyecto el informe rptJerarqDetalle, consistente también en un listado de los registros agrupados en jerarquía, pero incluyendo un mayor nivel de anidamiento. Debido a que el modo de creación es el mismo que el explicado en este apartado, no volveremos a repetir dichos pasos, consulte el lector el diseñador cargando el proyecto en VB y ejecutándolo para verificar sus resultados. La siguiente imagen muestra este informe en ejecución, que está asociado a la opción de menú Informes+Agrupados en varios niveles del programa de ejemplo.

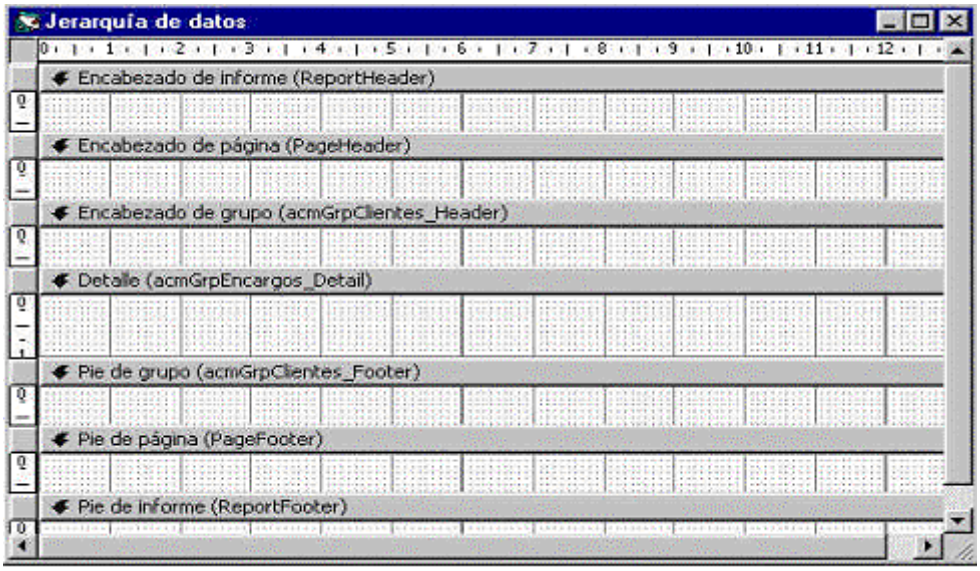


Figura 295. Secciones del informe después de obtener la estructura de datos.

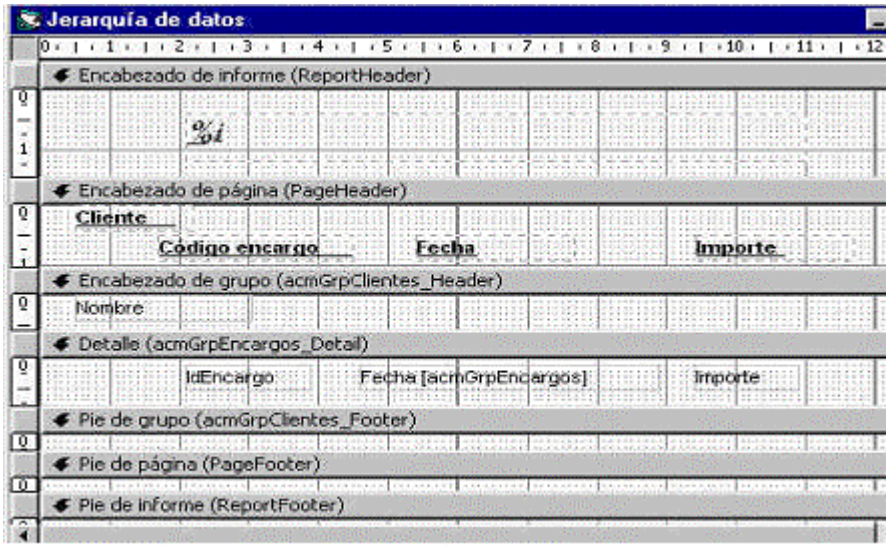


Figura 296. Diseño del informe con los registros agrupados.

```
Private Sub mnuInAgrupados_Click()
Dim lrptAgrupados As rptAgrupados
Set lrptAgrupados = New rptAgrupados
Load lrptAgrupados
lrptAgrupados.Show
End Sub
```

Código fuente 249

El control RptFunction

El control RptFunction nos permite realizar diversos tipos de cálculo sobre los registros del informe sobre la base de un campo. En función de la sección del diseñador donde sea situado, obtendremos el resultado sobre un grupo de registros o sobre la totalidad de los registros del listado. Si queremos obtener un resultado parcial, situaremos este control en el pie de grupo correspondiente a los registros a calcular; en cambio, si necesitamos un resultado global sobre todos los registros, deberemos colocar este control en la sección correspondiente al pie del informe.

Tomando el informe del apartado anterior, rptAgrupados, vamos a realizar una suma del campo Importe por cada grupo de registros correspondientes a un cliente del listado. Para ello insertaremos un control RptFunction en la sección Pie de grupo del comando acmGrpClientes. Asignaremos a la propiedad DataMember el comando que contiene la tabla Encargos: acmGrpEncargos, y a la propiedad DataField el campo Importe. Finalmente estableceremos en la propiedad FunctionType una constante con el tipo de cálculo a realizar, en concreto rptFuncSum. La tabla 87, muestra los valores disponibles para esta propiedad.

Constante	Valor	Descripción
rptFuncSum	0	(Predeterminado) Suma los valores del DataField.
rptFuncAve	1	Calcula el promedio de los valores del DataField.
rptFuncMin	2	Devuelve el valor mínimo del DataField.
rptFuncMax	3	Devuelve el valor máximo del DataField.
rptFuncRCnt	4	Cuenta las filas de la sección.
rptFuncVCnt	5	Cuenta los campos con valores no nulos.
rptFuncSDEV	6	Calcula la desviación estándar.
rptFuncSERR	7	Calcula el error estándar.

Tabla 87. Constantes disponibles para la propiedad FunctionType.

Aunque no sea obligatorio, añadiremos también un control RptLabel que indique el contenido del nuevo control RptFunction. La figura 297 muestra el informe en ejecución con el cálculo de suma sobre los registros.

Debido a que el control RptFunction calcula el resultado una vez que los registros se han procesado, si necesitamos por ejemplo, efectuar un cálculo para cada línea del informe, la solución pasa por crear un objeto Command basado en una instrucción SQL que contenga un campo calculado.

Para ilustrar este particular, crearemos el comando acmPlatos, que contiene una instrucción SQL, incluyendo un campo calculado para obtener el valor del IVA sobre cada precio. La figura 298, muestra la página de propiedades de este comando.

Continuaremos incluyendo un Data Report al que llamaremos rptCalcular, con la estructura del comando que acabamos de crear, insertándole además dos controles RptFunction en la sección

correspondiente al pie del informe, que realizarán la suma de las dos columnas con valores numéricos del informe. La figura 299, muestra el diseño del informe.

Encargos realizados por clientes

Cliente	Código encargo	Fecha	Importe
Antonio López			
	0120	martes, 16 de marzo de 1999	3000,000
	0099	domingo, 04 de abril de 1999	2500,000
	0072	jueves, 08 de julio de 1999	2100,000
		Total importes por cliente:	7600
Ana Gómez			
	0088	viernes, 08 de octubre de 1999	3800,000
		Total importes por cliente:	3800
Luis Rodrigo			
		Total importes por cliente:	

Figura 297. Informe que incorpora un control de función para sumar registros.

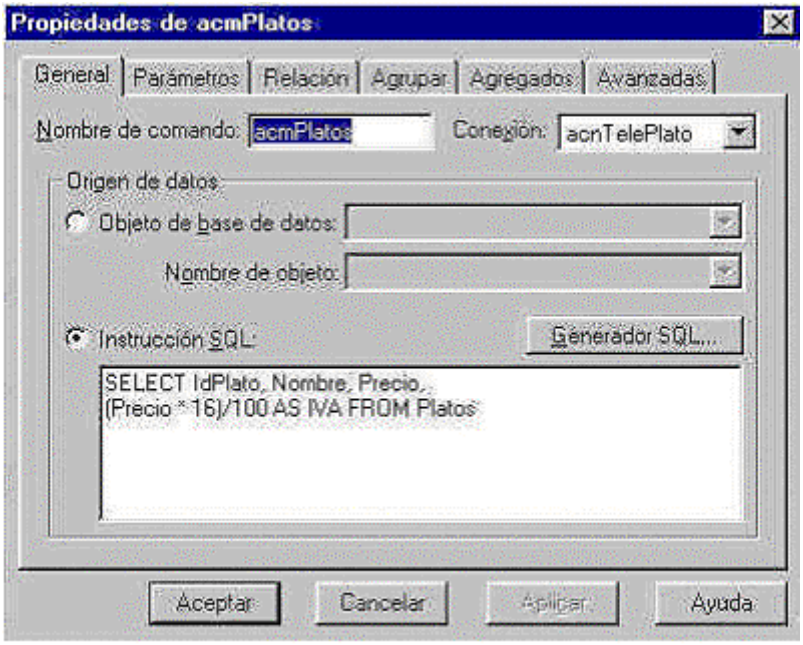


Figura 298. Comando basado en consulta SQL con campo calculado.

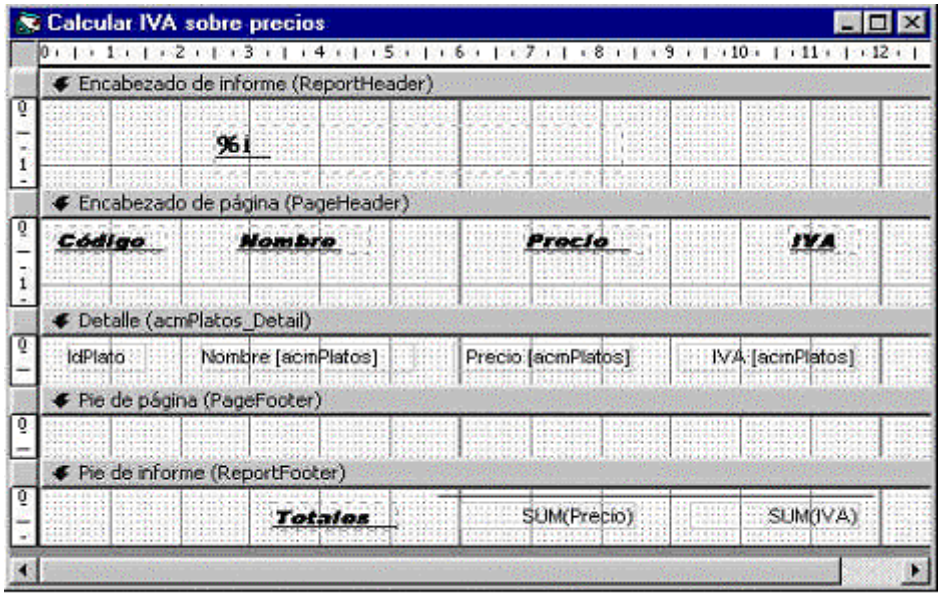


Figura 299. Diseñador de informe que incluye un campo calculado y totales generales.

Informes con selección de registros

Hasta el momento, todos los informes que hemos desarrollado obtienen la información de la base de datos de un modo genérico. Pero ¿qué debemos hacer si sólo tenemos que visualizar una parte de los registros en base a un valor proporcionado por el usuario?.

En este caso, más que un problema con el diseñador de informes, lo tenemos con el entorno de datos, ya que debemos modificar el objeto Command del DataEnvironment al que está conectado el informe, para que muestre los registros en un modo diferente al establecido originalmente.

Para mostrar este tipo de operación, crearemos un comando en el entorno de datos llamado acmSelecPlatos, que en principio obtendrá todos los registros de la tabla Platos de la base de datos TelePlato.MDB.

A continuación crearemos un nuevo informe para el proyecto, llamado rptSelecPlatos, en el que estableceremos las propiedades de datos para el nuevo objeto Command de igual forma que en los ejemplos de apartados anteriores.

Finalmente, crearemos el formulario frmSelecPlatos, desde el cual, el usuario introducirá un intervalo de valores para seleccionar un conjunto de registros de la tabla Platos.

Figura 300. Formulario para introducir los valores de selección de registros para un informe.

Este formulario se creará en la opción del programa Informes+Selección de registros. Una vez introducidos los valores a seleccionar, pulsaremos el botón Aceptar, que se encargará de crear una sentencia SQL y pasarla al objeto Command correspondiente del entorno de datos, de manera que la nueva consulta será la que se realice cuando el informe solicite los datos. En el código fuente 250 se muestran las líneas de código correspondientes a este botón del formulario.

```
Private Sub cmdAceptar_Click()
Dim ldteTelePlato As dteTelePlato
Dim lacmSelecPlatos As ADODB.Command
Dim lrptSelecPlatos As rptSelecPlatos
Dim lcSQL As String
Dim lnInd As Integer
' instanciar objetos informe y entorno de datos
Set lrptSelecPlatos = New rptSelecPlatos
Set ldteTelePlato = New dteTelePlato
' establecer títulos del informe
lrptSelecPlatos.Title = "Listado de platos desde el " & _
   Me.txtDesde.Text & _
   " hasta el " & Me.txtHasta.Text
' construir la consulta SQL
lcSQL = "SELECT * FROM Platos "
lcSQL = lcSQL & "WHERE IdPlato >= '" & Me.txtDesde.Text & "' "
lcSQL = lcSQL & "AND IdPlato <= '" & Me.txtHasta.Text & "' "
lcSQL = lcSQL & "ORDER BY IdPlato"
' buscar el objeto Command en la colección
' del entorno de datos
For Each lacmSelecPlatos In ldteTelePlato.Commands
   If lacmSelecPlatos.Name = "acmSelecPlatos" Then
      ' originalmente los datos se recuperaban
      ' como tabla, ahora serán mediante
      ' una instrucción SQL
      lacmSelecPlatos.CommandType = adCmdText

      ' asignar la cadena con la instrucción SQL
      lacmSelecPlatos.CommandText = lcSQL
   End If
Next
' asignar el entorno de datos al informe
' para que la conexión del informe con la
' base de datos sea actualizada
Set lrptSelecPlatos.DataSource = ldteTelePlato
' según el OptionButton que esté pulsado
' se previsualizará el informe o se
' enviará a la impresora
If Me.optPantalla.Value Then
   lrptSelecPlatos.Show
Else
   lrptSelecPlatos.Visible = False
   lrptSelecPlatos.PrintReport False, rptRangeAllPages
End If
Unload Me
End Sub
```

Código fuente 250

La figura 301 muestra este informe en ejecución, visualizando un rango de registros.

En definitiva, esta técnica nos permite crear informes en los que podremos mostrar información diversa, según los valores proporcionados por el usuario.

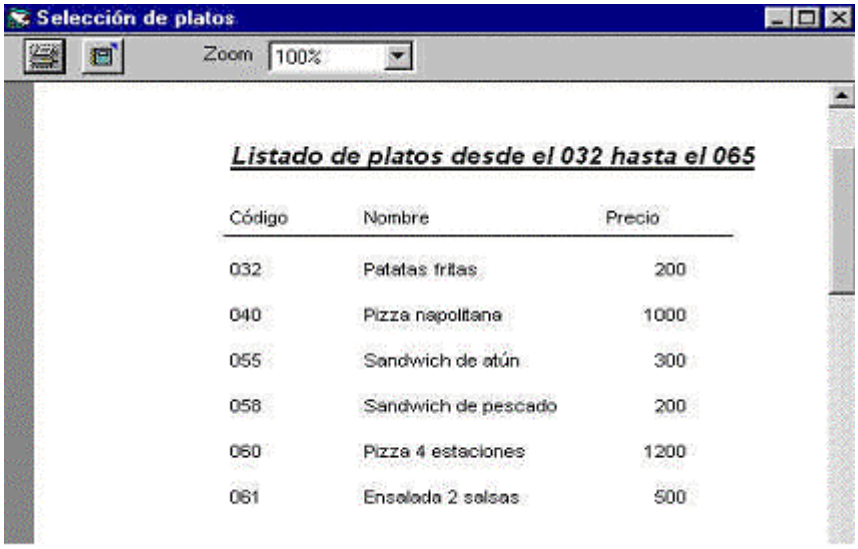


Figura 301. Informe de selección de registros en ejecución.

Exportar un informe

Mediante el método ExportReport() del objeto DataReport, es posible exportar un informe a un fichero en un formato predeterminado. El código fuente 251 exportaría un rango de páginas de un informe a un fichero, mostrando previamente al usuario el cuadro de diálogo Exportar para permitir variar alguno de los valores a exportar.

```
lrptBasico.ExportReport rptKeyHTML, "c:\correo\InfBasico", _
    True, True, rptRangeFromTo, 4, 7
```

Código fuente 251

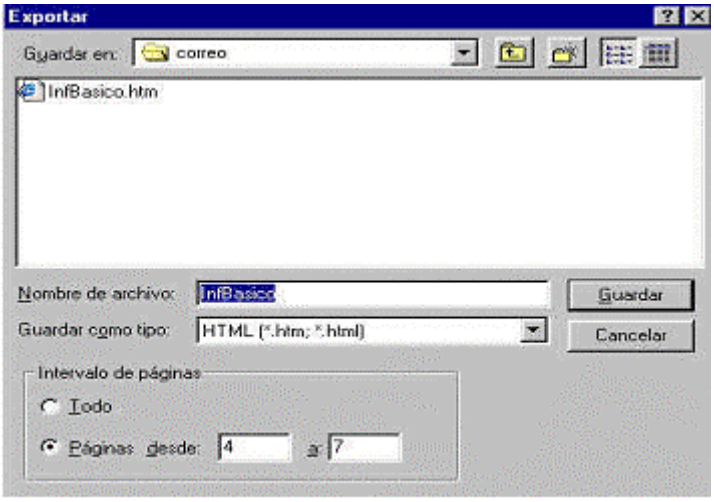


Figura 302. Cuadro de diálogo Exportar, para guardar informe en archivo.

Para mayor información sobre el método ExportReport() o el objeto DataReport, consulte el lector, el apartado dedicado a este objeto dentro de este mismo tema.

El encargado de establecer las características del informe que se va a exportar es el objeto ExportFormat.

Propiedades

- FileFilter. Cadena que contiene el tipo de fichero a guardar. Puede especificarse más de un fichero utilizando punto y coma.

```
oExportFormat.FileFilter = "*.htm; *.prn"
```

- FileFormatString. Cadena con una breve descripción del tipo de fichero al que se va a exportar el informe. Esta cadena aparecerá en el ListBox Guardar como tipo, del cuadro de diálogo Exportar.

```
oExportFormat.FileFormatString = "Informe para intranet"
```

- FormatType. Constante con el tipo de formato en el que se creará el informe, según la tabla 88.

Constante	Valor	Descripción
RptFmtHTML	0	HTML
RptFmtText	1	Texto
RptFmtUnicodeText	2	Unicode
rptFmtUnicodeHTML_UTF8	3	HTML codificado con el juego de caracteres universal (UTF – 8)

Tabla 88. Constantes para la propiedad FormatType.

- Key. Cadena para identificar de forma única al objeto dentro de la colección ExportFormats.

- Template. Contiene la cadena utilizada como plantilla para exportar el informe en formato HTML.

A pesar de que este es el objeto utilizado para especificar los atributos del informe a exportar, no debemos instanciarlo en nuestra aplicación. Lo que tendremos que hacer, es añadir un nuevo elemento a la colección ExportFormats del objeto DataReport, mediante el método Add() de dicha colección, pasando los valores para el formato en los parámetros de este método, que se encargará internamente de crear un objeto ExportFormat y agregarlo a la colección.

Las líneas de código mostradas en el código fuente 252, pertenecen a la opción de menú Informes+Exportar, de la aplicación de ejemplo, y se encargan de exportar uno de los informes del programa empleando la técnica antes descrita.

```
Private Sub mnuInExportar_Click()
Dim lrptBasico As rptBasico
Dim lcPlantilla As String
' crear la plantilla con el código html
lcPlantilla = "<HTML>" & vbCrLf & _
   "<HEAD>" & vbCrLf & _
   "<TITLE>" & "TelePlato: " & rptTagTitle & _
```

```
   "</TITLE>" & vbCrLf & _
   "TelePlato: Distribución en Internet" & vbCrLf & _
   rptTagTitle & _
   "<BODY>" & vbCrLf & _
   rptTagBody & vbCrLf & _
   "<BODY>" & vbCrLf & _
   "</HTML>"
' abrir el informe
Set lrptBasico = New rptBasico
Load lrptBasico
lrptBasico.Show
' añadir a la colección ExportFormats, la información
' para crear un nuevo formato
lrptBasico.ExportFormats.Add Key:="personalizado", _
   FormatType:=rptFmtHTML, _
   FileFormatString:="Informe personalizado", _
   FileFilter:="*.htm", _
   Template:=lcPlantilla
' exportar el informe utilizando
' el cuadro de diálogo Exportar
lrptBasico.ExportReport "personalizado", "c:\correo\prueba", _
   True, True, rptRangeAllPages
End Sub
```

Código fuente 252

Crystal Reports

Como comentábamos al principio de este tema, Crystal Reports es un generador de informes que se incluye en VB, y aunque el modo recomendado es mediante el uso del propio diseñador de informes de VB, todavía podemos usar esta utilidad para crear los listados de nuestras aplicaciones.

Al igual que ocurre con el Data Report, Crystal Reports (o CR, como también lo denominaremos a partir de ahora), nos permite ocuparnos de las labores de diseño y manejo de datos del informe, ocupándose internamente de los aspectos más áridos y de bajo nivel en la creación de informes.

Comenzando con Crystal Reports

Cuando desde VB seleccionemos la opción de menú Complementos+Diseñador de informes, se pondrá en marcha Crystal Reports, presentando su pantalla inicial, que podemos ver en la figura 303.

Aunque este generador dispone de un asistente para crear informes, no haremos uso por ahora del mismo, creando nuestros listados manualmente, esto nos servirá para familiarizarnos con las diferentes opciones de la herramienta.

Crystal Reports permite trabajar con muy diversos formatos de bases de datos. Para los ejemplos utilizados aquí, emplearemos Access como gestor de bases de datos, puesto que es de sobra conocido por el lector, al haber sido empleado en otros temas de este curso.

La base de datos Trafico.MDB, incluida como ejemplo, servirá para obtener la información que mostrarán los diferentes listados diseñados con Crystal Reports, durante los siguientes apartados. Esta base de datos contiene la tabla Llegadas, creada para ser usada por una compañía de transporte que necesita controlar diversos aspectos en la recepción de mercancía para sus clientes, como el código de transporte, fecha de llegada, importe a cobrar, etc.

Vamos a comenzar, creando un listado sencillo, que incluya todos los campos de la tabla Llegadas y sin ningún criterio especial en cuanto a presentación, selección, o agrupación de los registros.

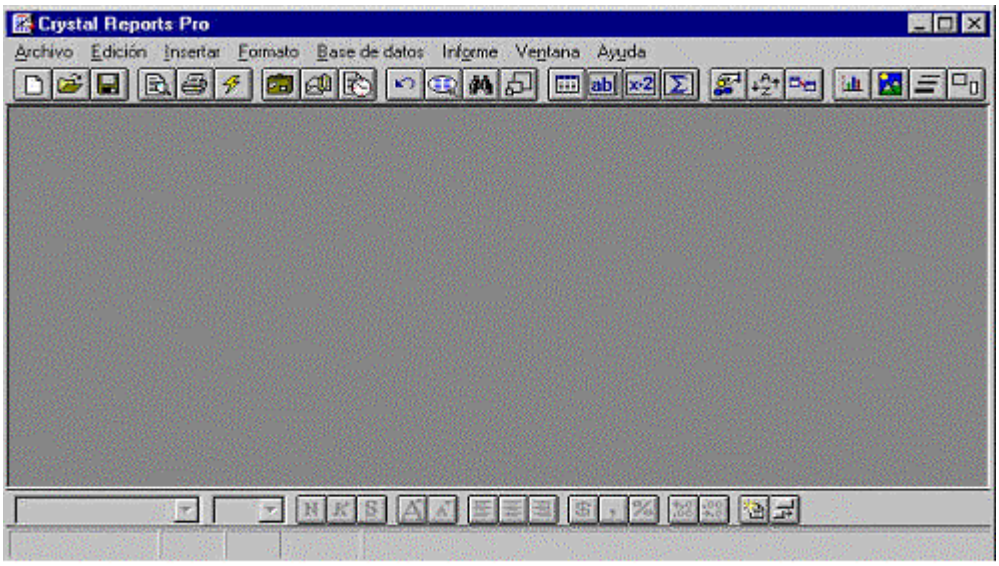


Figura 303. Pantalla inicial de Crystal Reports.

Creación de un nuevo informe

Seleccionaremos la opción Archivo+Nuevo, del menú de CR, para crear un nuevo informe, lo que nos presentará el cuadro de diálogo de selección de asistente. El botón de la barra de herramientas para esta opción es: 
Pulsaremos el botón Personalizado, de forma que se visualice la parte inferior de este diálogo, en donde podremos seleccionar un tipo de informe que no incluya asistente. Pulsaremos por tanto Informe personalizado.

Figura 304. Selección del modo de creación de un nuevo informe.

Selección de la base de datos

Acto seguido, pulsaremos el botón Archivo, en la anterior ventana, con lo cual, se mostrará un nuevo cuadro de diálogo (figura 305) para seleccionar el fichero que contiene los datos a utilizar en la creación del informe.

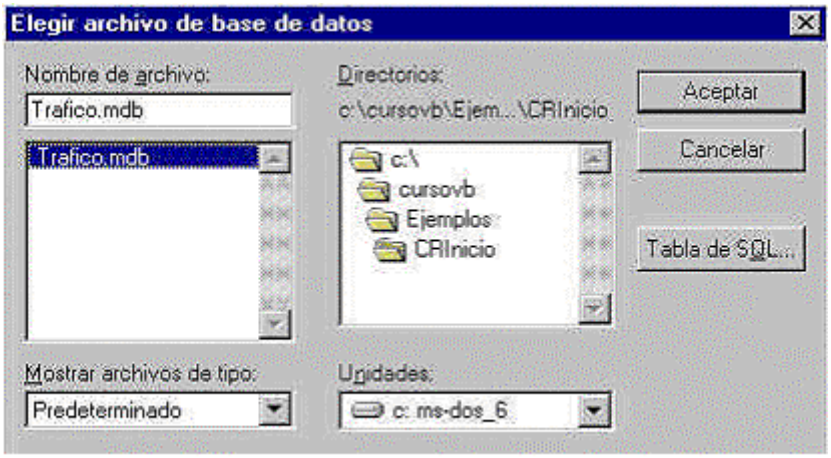


Figura 305. Selección del fichero que contiene el origen de datos para el informe.

Secciones del informe

Una vez seleccionada la base de datos, aparecerá en la ventana de Crystal Reports, en modo de diseño, el informe vacío, listo para que comencemos a crearlo.

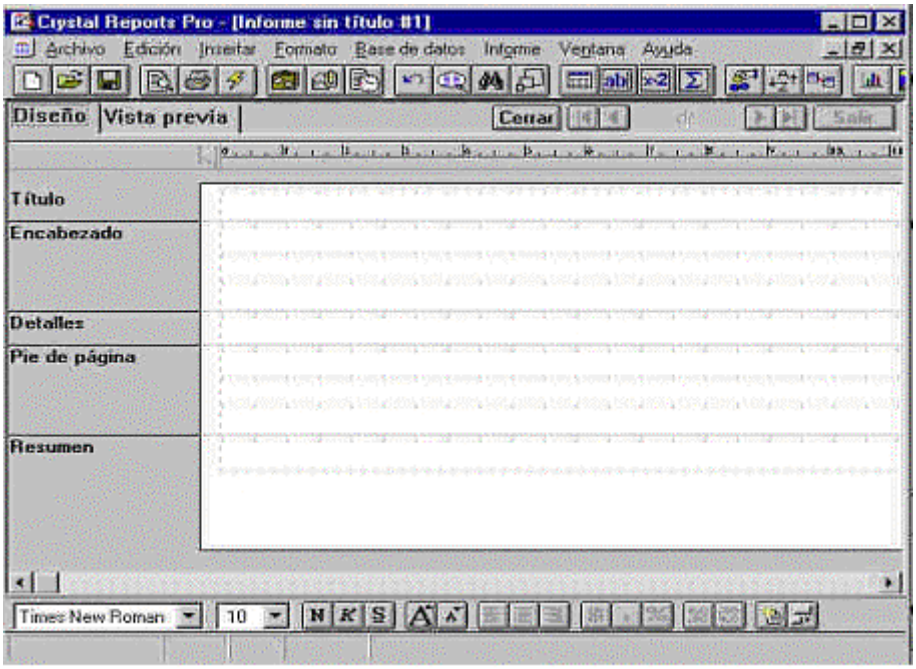


Figura 306. Secciones de un informe.

La plantilla del informe tiene las siguientes secciones:

- Título. Como su nombre indica, contiene el nombre que le daremos al listado. Esta descripción, sólo aparece al comienzo del informe.

- Encabezado. En esta sección, situaremos el texto que necesitemos mostrar como cabecera en cada nueva página del informe. Podemos incluir los títulos de las columnas o campos que se van a imprimir en Detalles, el número de página, la fecha del informe, etc.

- Detalles. Esta sección contiene los campos de los registros que componen el cuerpo del informe.

- Pie de página. Esta sección también se imprimirá una vez por página, al igual que Encabezado, pero con la diferencia de que los campos y el texto a imprimir se hará en la parte inferior de la hoja. Aquí podemos situar campos basados en cálculos que impriman el total por página, totales de columnas y también el número de página y fecha de listado al igual que en Encabezado.

- Resumen. Esta sección se imprime después de haber terminado la impresión de los registros del listado.

Insertar campos en el informe

Creado el nuevo informe vacío, junto a la plantilla de este, aparece la ventana Insertar campo de base de datos, la cual contiene las diferentes tablas y campos que componen la base de datos seleccionada. Esta ventana aparece por defecto al crear un nuevo informe, si posteriormente necesitamos agregar nuevos campos al informe, podemos invocarla mediante la opción Insertar+Campo de base de datos, del menú de Crystal Reports, que tiene el siguiente botón en la barra de herramientas: 

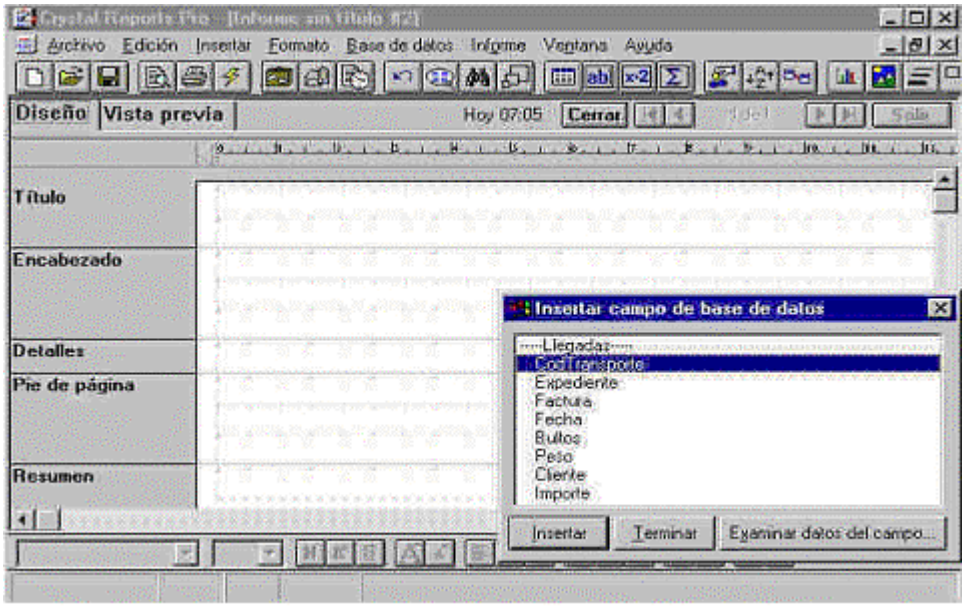


Figura 307. Nuevo informe vacío con ventana de selección de campos de la base de datos.

Para insertar un campo en el informe, hemos de seleccionarlo en la ventana de campos de la base de datos, y mediante arrastrar y soltar, situarlo en la sección del informe en la que queramos que aparezca dicho campo, esta sección será generalmente Detalles. Al incluir un campo en esta sección, automáticamente aparecerá el nombre del campo en la sección Encabezado, lo que nos ahorrará el trabajo de crear un campo de texto para el nombre de la columna.

También es posible, mediante las operaciones estándar de Windows (teclas Ctrl o Mayús y teclas de dirección), seleccionar un grupo de campos e insertarlos todos juntos en el informe.

El botón Examinar datos del campo, en la ventana de campos, visualiza los valores no repetidos, que contiene dicho campo en la tabla, como muestra la figura 308.

Figura 308. Valores del campo Cliente, de la tabla Llegadas.

Procederemos a insertar los campos en la manera que acabamos de ver, conservando el orden que mantienen en la tabla, quedando el diseño del informe según muestra la figura 309.

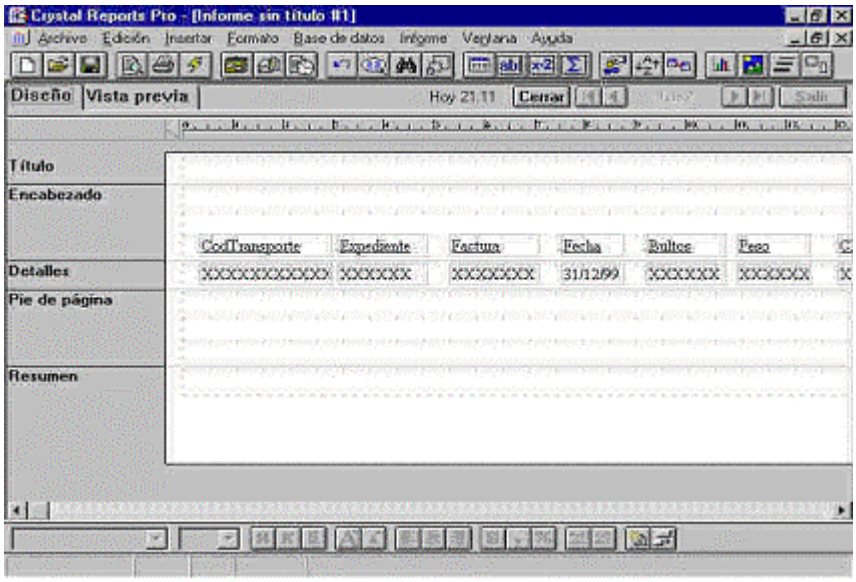


Figura 309. Informe con los campos de la tabla insertados.

Para saber con que campo estamos trabajando en todo momento, cada vez que seleccionemos uno del informe, se visualizará su nombre en la parte inferior de la ventana del informe.

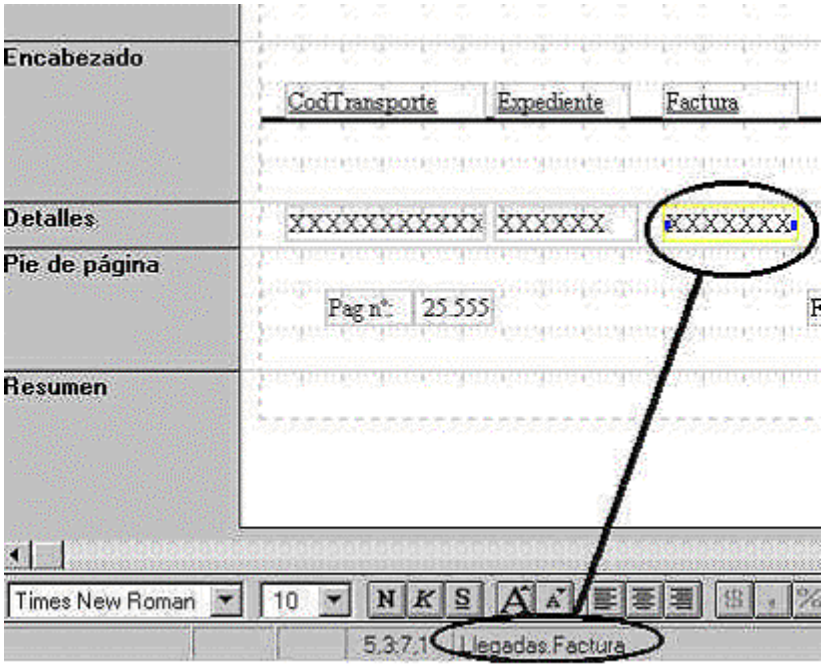


Figura 310. Indicador del campo seleccionado.

Con los pasos dados hasta ahora ya hemos creado un informe, excesivamente simple, eso sí, pero funcional. Procederemos a grabarlo mediante la opción Archivo+Guardar del menú de CR, que tiene el siguiente botón en la barra de herramientas:  .

Le daremos el nombre Básico. La extensión .RPT, utilizada por Crystal Reports, la asignará la propia herramienta en el caso de que no lo hagamos nosotros.

Previsualizar el informe

Si para cada mínimo cambio que realizamos en el informe, necesitáramos imprimir una prueba en papel, haríamos un flaco servicio a los árboles del planeta. Por este motivo, Crystal Reports incluye una opción de previsualización, que muestra por pantalla el resultado del informe sin necesidad de obtenerlo en papel. De esta forma, haremos nuestra pequeña aportación ecológica y ahorraremos tiempo en diseño, ya que la previsualización es más rápida que la impresora.

Para ello, sólo tenemos que hacer clic en la pestaña Vista Previa, situada en la parte superior izquierda de la plantilla del informe. La figura 311, muestra el resultado de la previsualización del informe.

Los botones de desplazamiento situados en la parte derecha a la altura de esta pestaña, nos permitirán la navegación por las páginas del informe, en el caso de que este tenga más de una página.

Por supuesto, habrá ocasiones en que para realizar ciertos ajustes, no quede más remedio que utilizar la impresora, pero el uso que haremos de ella será mucho menor, si utilizamos esta opción.

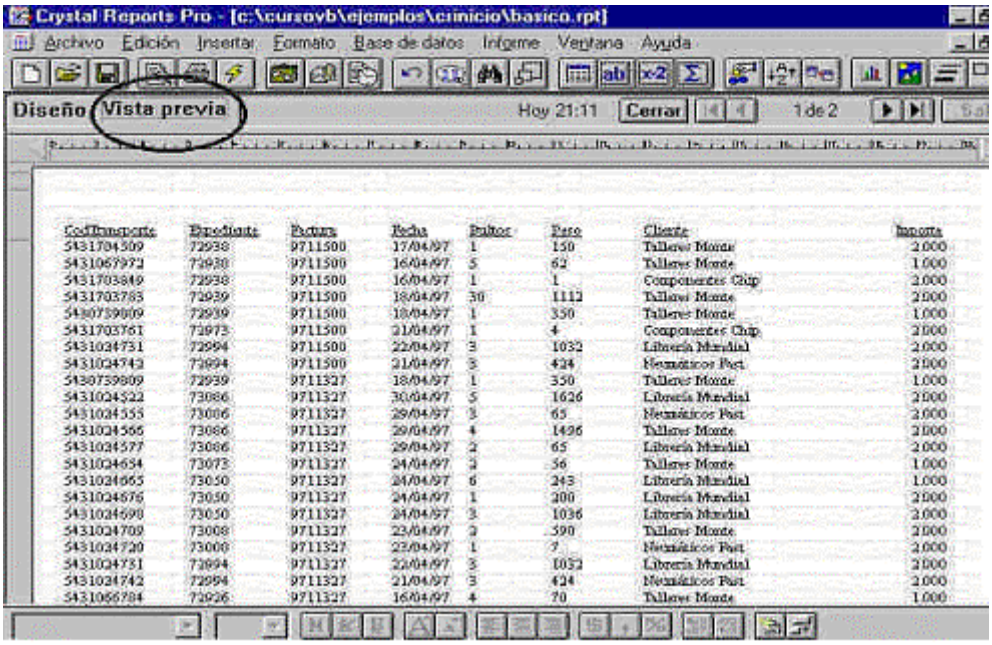


Figura 311. Vista previa del informe.

Otros modos de realizar esta acción, son usar la opción de menú de CR: Archivo+Vista Previa, o el botón de la barra de herramientas: .

La opción de previsualización, tiene como ventaja añadida, la posibilidad de obtener diferentes vistas del informe. Mediante la característica de zoom, es posible visualizar el informe ajustado a la ventana, al ancho y a tamaño completo. El comando del menú de CR que ejecuta el zoom es Informe+Zoom, y su botón en la barra de herramientas: .

Para volver al modo de creación del informe, pulsaremos en la pestaña Diseño, de la plantilla del informe.

Completando el informe

El informe que acabamos de crear: Básico.RPT, como ya hemos comentado, proporciona la información suficiente, pero queda bastante pobre en cuanto a aspecto. Vamos a darle en este apartado, algunos retoques, que sin alterar la información que proporciona, si mejorará su imagen global, lo que hará que gane muchos puntos respecto a su estado original.

Título

Insertaremos un título en el informe, de manera que el usuario sepa cual es el contenido del mismo. Para ello utilizaremos un campo de texto, cuya creación vemos seguidamente.

Para incluir un campo de texto en el informe, seleccionaremos la opción de menú Insertar+Campo de texto, que tiene el siguiente botón en la barra de herramientas: .

Aparecerá la ventana Editar campo de texto, en la que escribiremos el texto que deseamos que aparezca en el informe. Pulsaremos Aceptar, con lo que el campo se situará en el informe, pudiéndolo desplazar hasta la sección que queramos, en nuestro caso Título

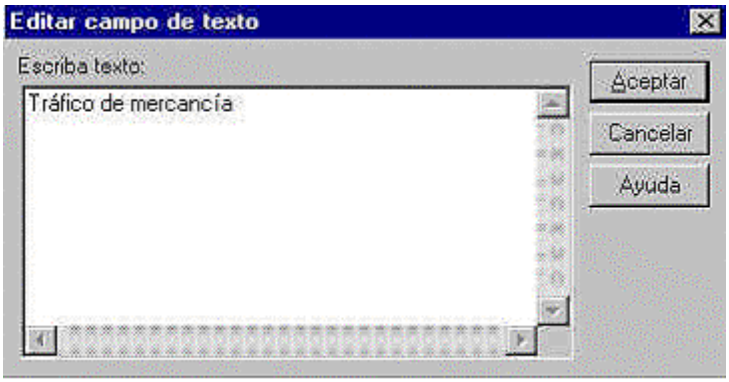


Figura 312. Creación de campo de texto para un informe.

Una vez situado el campo de texto, es posible editarlo mediante la opción de menú Edición+Campo de texto. También se puede editar, al seleccionar el campo en el informe y abrir su menú contextual haciendo clic con el botón secundario del ratón, que incluirá la misma opción de edición para el campo.

El mismo menú contextual, permite además, modificar el tipo de fuente del campo, o utilizar la barra de herramientas situada en la parte inferior de CR.

Figura 313. Barra de herramientas de formato y estilo.

El campo recién incluido en el título, quedaría como muestra la figura 314.

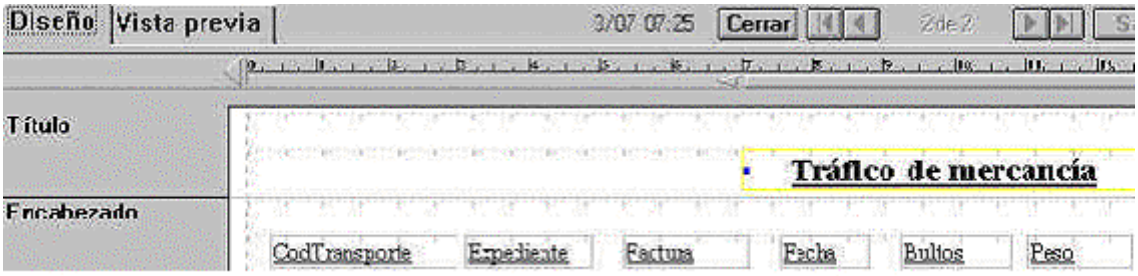


Figura 314. Campo de texto en la sección Título del informe.

Dibujo

Utilizando las opciones Insertar+Línea o Insertar+Cuadro, del menú de CR, podremos dibujar en cualquier parte del informe, una de estas formas, que podremos usar para separar elementos del informe o para dar un mejor acabado al mismo.

Una vez seleccionada cualquiera de estas herramientas, aparecerá un puntero del ratón en forma de lapicero, acto seguido debemos hacer clic y arrastrar creando el dibujo, soltando al completarlo. Los botones de la barra de herramienta son:  .

La figura 315, muestra como hemos dibujado una línea, para separar los nombres de las columnas de los datos del informe, y un rectángulo para realzar el título.

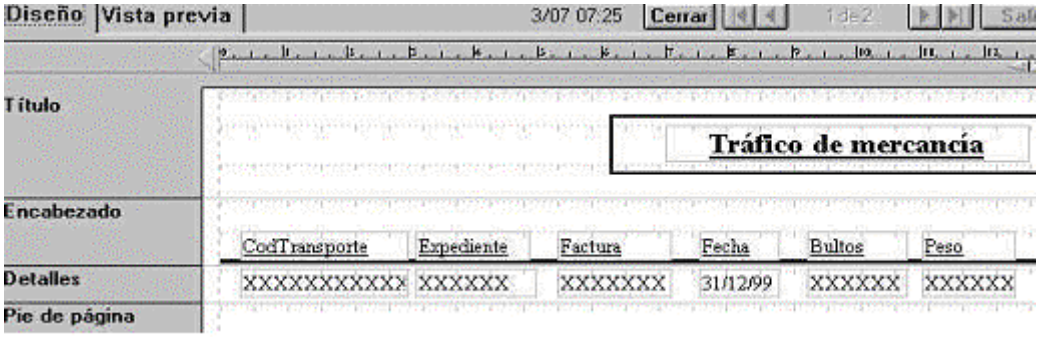


Figura 315. Línea y rectángulo dibujados en el informe.

Imagen

Mediante la opción Insertar+Imagen, es posible incluir un fichero gráfico en el informe, que puede contener el logotipo de la empresa o cualquier otro gráfico. Al seleccionar esta opción, aparecerá un cuadro de diálogo para elegir el fichero que contiene la imagen. Una vez seleccionada, situaremos la imagen en el informe, de igual forma que hacemos con los campos. El botón de la barra de herramientas es:  .

En nuestro informe, hemos insertado el fichero Finish.BMP, situándolo en la sección de Título, tal y como muestra la figura 316.

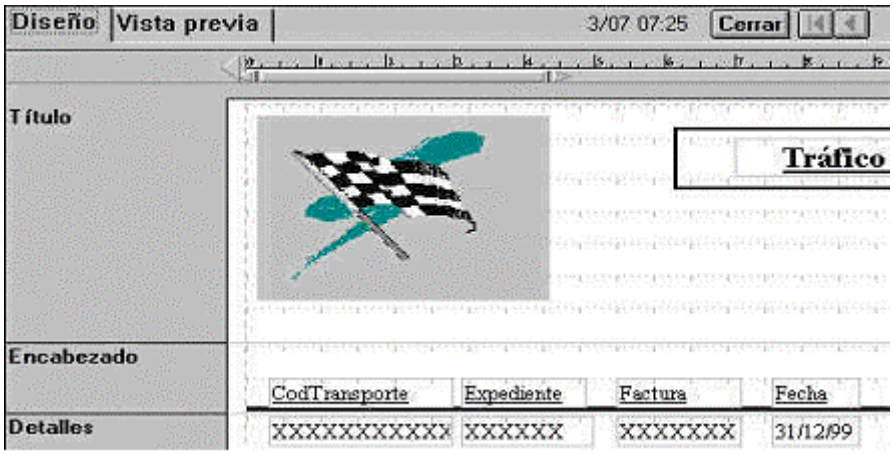


Figura 316. Imagen insertada en el informe.

Una vez insertada la imagen, es posible seleccionarla y volver a cambiar su posición, o modificar su tamaño, utilizando el ratón.

Ajustar sección

Al insertar una imagen, hemos podido comprobar, que si esta era mayor que la altura de la sección en la que se debía insertar, el tamaño de la sección cambiaba automáticamente para adaptarse a la altura de la imagen. Si necesitamos volver a reajustar la altura de la sección, hemos de situar el puntero del ratón en la línea divisoria de la sección, dentro del informe. Cuando cambie de forma, haremos clic y arrastraremos, dando mayor o menor altura a la sección, en función de los requerimientos de la misma. La figura 317, muestra un cambio en la sección Encabezado, a la que se ha dado más altura, de forma que exista una mayor separación entre los títulos de las columnas y el comienzo del detalle de las mismas.

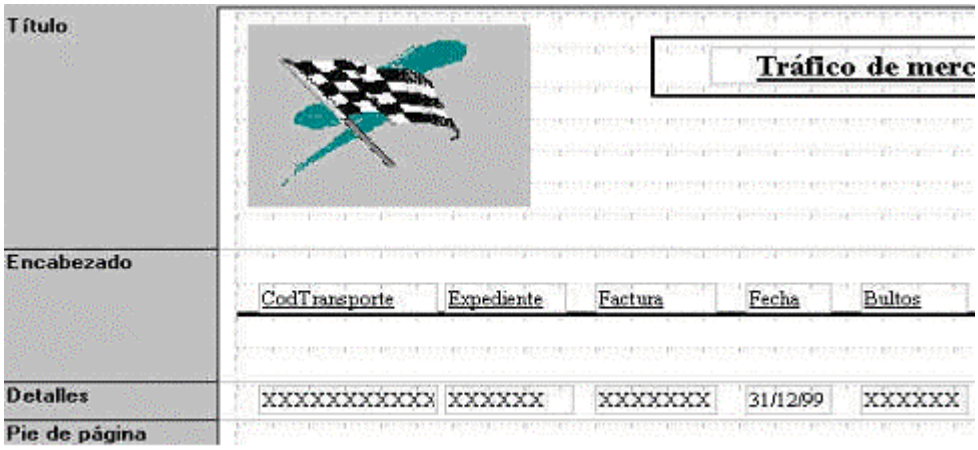


Figura 317. Sección Encabezado con cambio de tamaño.

Número de página y fecha del informe

La opción del menú de CR Insertar+Campo especial, abre un submenú, que nos permite elegir una serie de campos predefinidos. Estos campos suelen ser de uso generalizado para la mayoría de los informes, de esta manera, el generador nos ahorra el trabajo de tener que calcularlos manualmente. Dos de estos campos de uso común son el número de página y la fecha del informe, que se insertan en el mismo de la forma vista anteriormente para el resto de los campos.

La figura 318, muestra estos dos campos, situados en la sección Pie de página del informe, de manera que serán visualizados en la parte inferior de cada página.

Por supuesto, los campos de texto Pag nº y Fecha Informe, utilizados para identificar a los anteriores campos especiales, debemos establecerlos aparte, ya que no se incluyen automáticamente.

Por defecto, el formato de un campo de fecha es dd/mm/aa, pero mediante la opción Formato+Campo, o el menú contextual de dicho campo, es posible cambiar el formato de presentación de la fecha, a través del cuadro de diálogo que aparece en la figura 319.

Como puede comprobar el lector, se ha modificado el formato de la fecha para que el mes se muestre en letra. Este es uno de los diferentes aspectos en que puede cambiarse la presentación de un campo de este tipo.

En el caso de campos numéricos, también es posible modificar su formato mediante la misma opción de menú, que mostrará el diálogo que aparece en la figura 320.

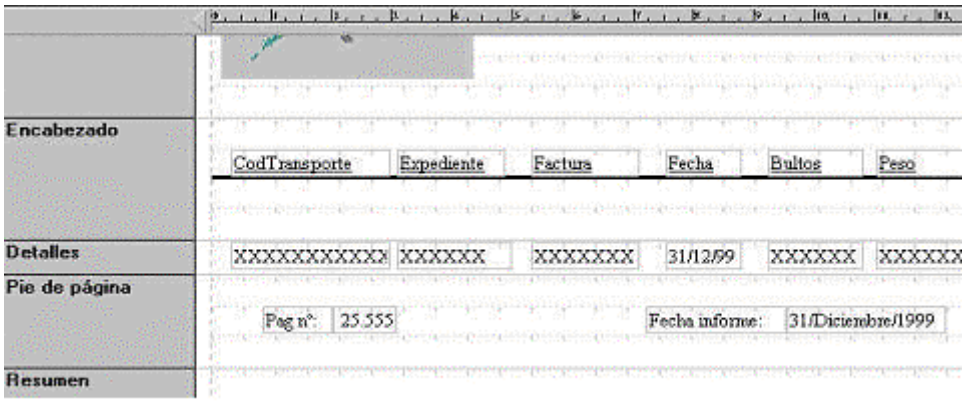


Figura 318. Informe con campos de número de página y fecha.

Figura 319. Cuadro de diálogo para modificar el formato de un campo de fecha.

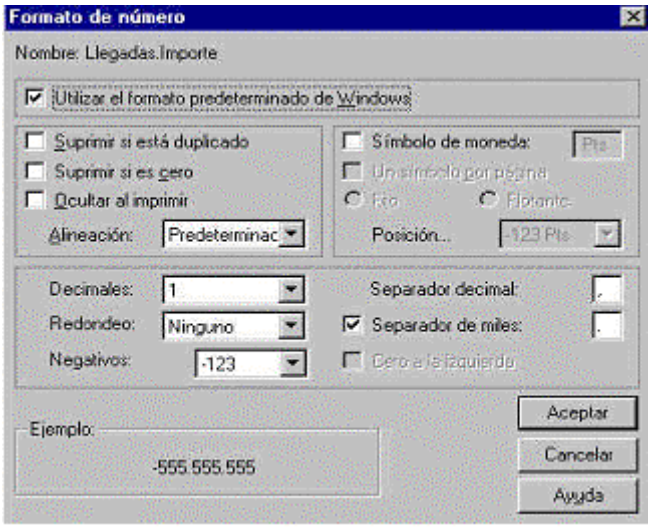


Figura 320. Ventana de configuración para un campo numérico.

Configuración del informe

Todos los campos que hemos insertado hasta el momento en el informe, tienen un tipo de letra y tamaño determinado. Pero ¿qué ocurre si necesitamos que tengan un formato diferente, y que además, dicho formato se establezca por defecto?. La solución está en la opción Archivo+Opciones o Archivo+Opciones de informe, que permiten establecer las opciones predeterminadas para todos los informes de Crystal Reports o sólo para el informe actual, como muestran las figuras 321 y 322.

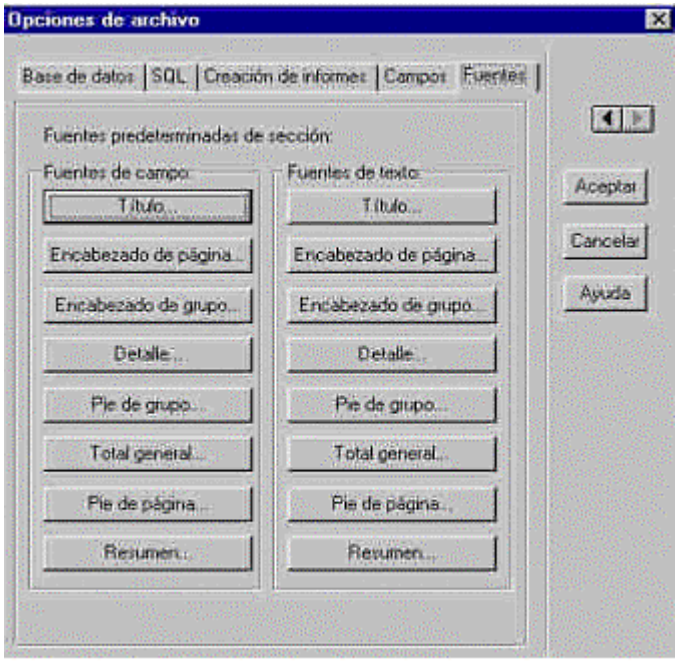


Figura 321. Opciones generales para Crystal Reports.

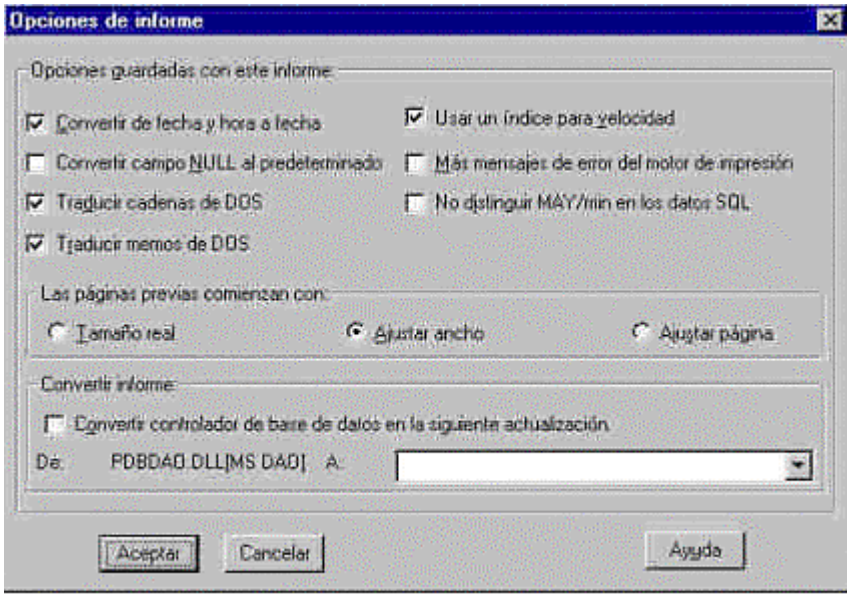


Figura 322. Opciones de configuración para el informe actual.

Mediante estos cuadros de diálogos, podremos configurar todos los aspectos referentes a la creación de los informes.

Manipulación de la base de datos conectada al informe

Las siguientes acciones, están disponibles para el manejo de la base de datos enlazada al informe sobre el que trabajamos.

Establecer ubicación

Cuando indicamos la ruta de la base de datos al crear el informe, se conserva dicha ruta en el interior del mismo, aún cuando cambiemos la ubicación tanto de la base de datos, como del informe. Mediante la opción Base de datos+Establecer ubicación, se abrirá un cuadro de diálogo para indicar la nueva ruta de la base de datos, o lo que es mejor, podemos pulsar el botón Igual al informe, lo que hará que Crystal Reports siempre busque la base de datos en el mismo directorio en el que está situado el informe.

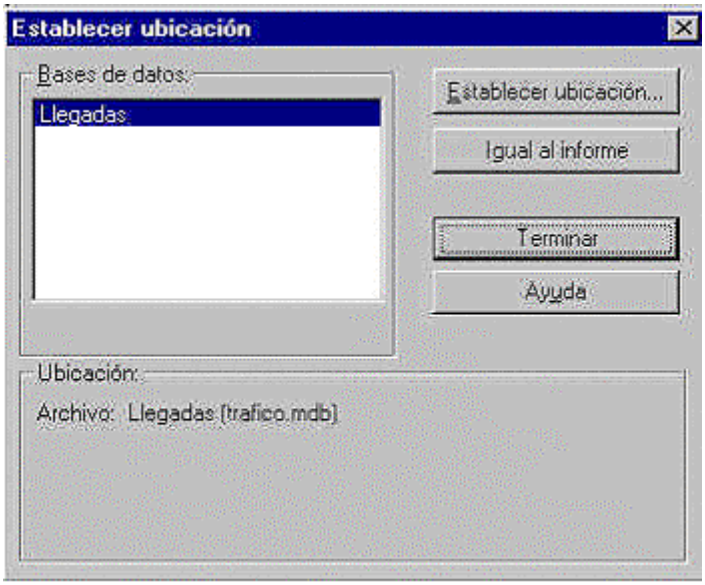


Figura 323. Ventana de ubicación de la base de datos.

Comprobar la base de datos

Es posible que durante la creación del informe, debamos realizar modificaciones en una o varias tablas de la base de datos. Esto puede provocar errores al ejecutar el informe, ya que los cambios en la base de datos, no han sido notificados al informe.

Por tal motivo, la opción Base de datos+Comprobar base de datos de CR, permite actualizar la información que mantiene el informe, de la base de datos conectada al mismo, de forma que no se produzcan errores.

Selección de registros

Para informes basados en tablas muy extensas, lo normal es que no sea necesario listar todos los registros que contiene la tabla, sino una selección de los mismos.

La opción Informe+Asistente de selección de registros, nos ayuda en el filtrado de los datos que debemos mostrar en el informe, construyendo una formula de selección de registros, basada en uno o más campos de una o varias tablas. El botón en la barra de herramientas es:  .

Al invocar esta opción, se mostrará la ventana de dicho asistente, como vemos en la figura 324.

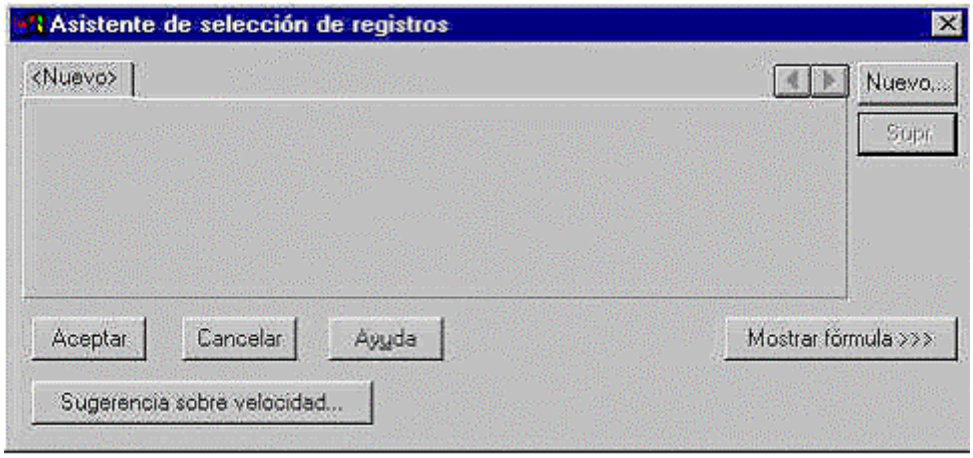


Figura 324. Asistente de selección de registros

Cada vez que pulsemos en la pestaña Nuevo, se podrá elegir un campo de una tabla. Una vez hecha esta selección, se creará una nueva pestaña para el campo elegido. En ella, indicaremos el modo de selección de registros para el campo, y el valor que debe tener el campo para ser seleccionado entre el resto de registros, como muestra la figura 325.

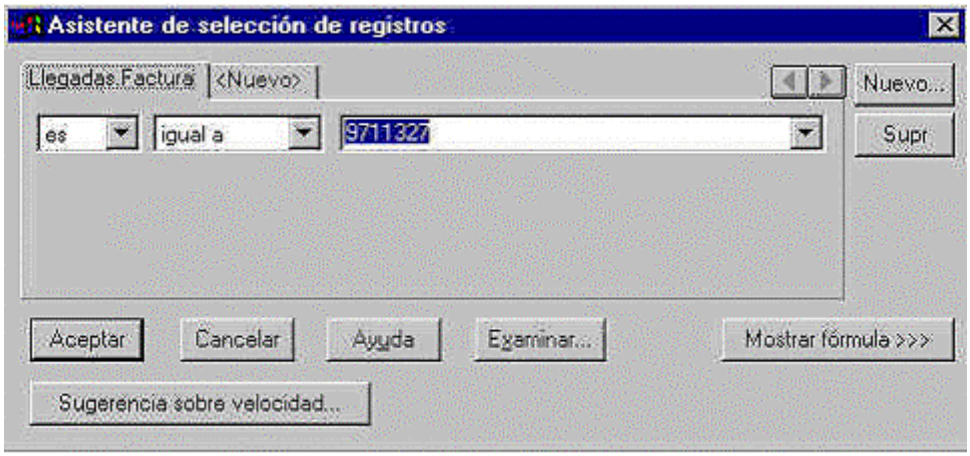


Figura 325. Modo de selección de registros para uno de los campos.

Para el informe Basico.RPT, que estamos usando como ejemplo, se ha establecido una selección de registros en la que el campo Factura debe ser igual a "9711327", y el campo Expediente debe ser igual a "72926" como podemos ver en la figura 326.

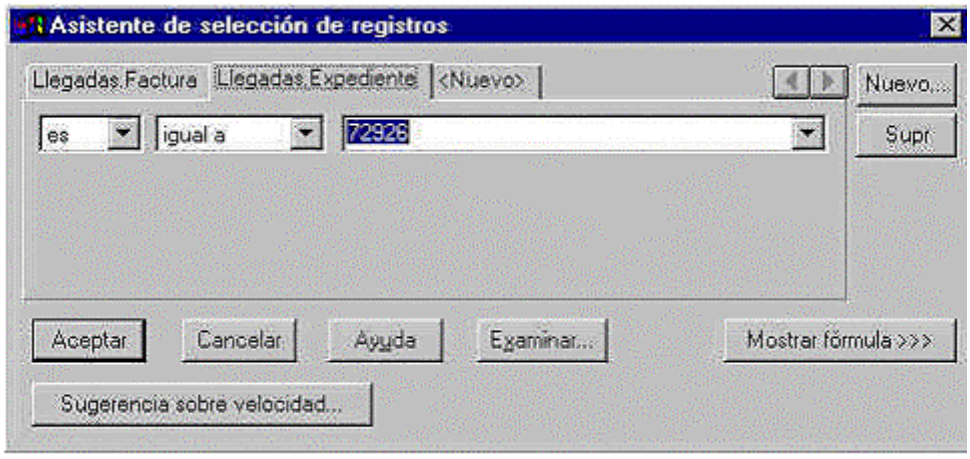


Figura 326. Asistente para filtrado de datos en el informe.

Si pulsamos el botón Mostrar fórmula, se visualizará el código de la fórmula creado por el asistente, para conseguir la selección de registros, basados en las condiciones establecidas para cada campo.

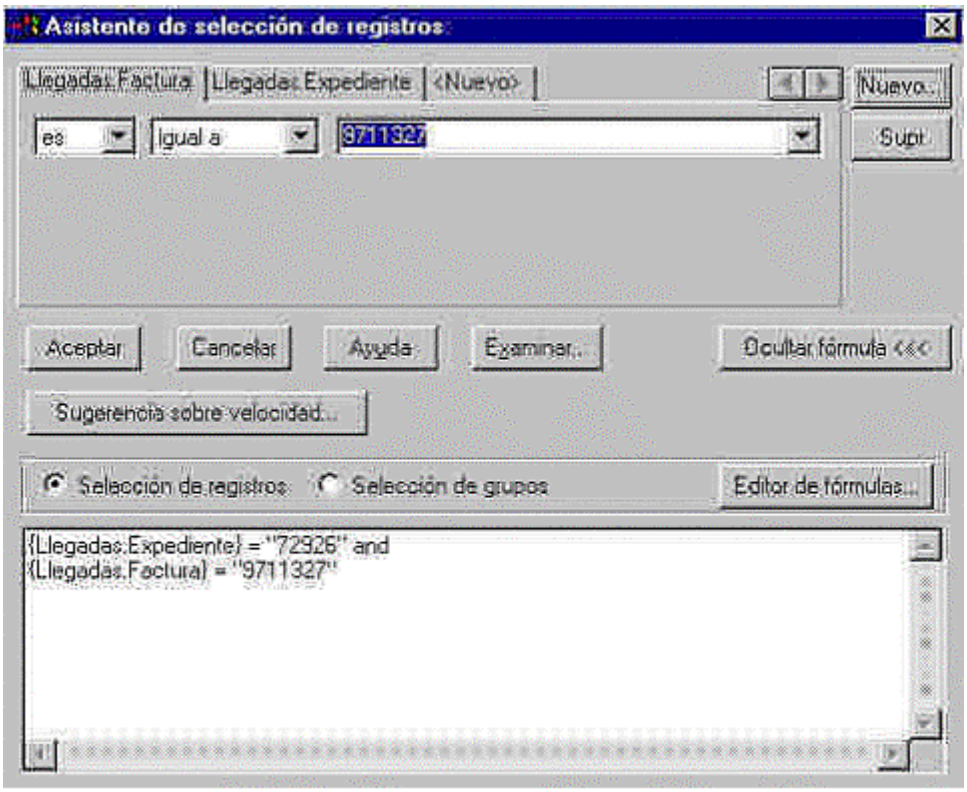


Figura 327. Asistente de selección mostrando la fórmula de selección de registros.

La creación de fórmulas será tratada más adelante.

Finalizada la selección de registros, podemos realizar una vista previa del informe, para comprobar que ahora sólo se mostrarán los registros que hemos indicado en el asistente de selección.

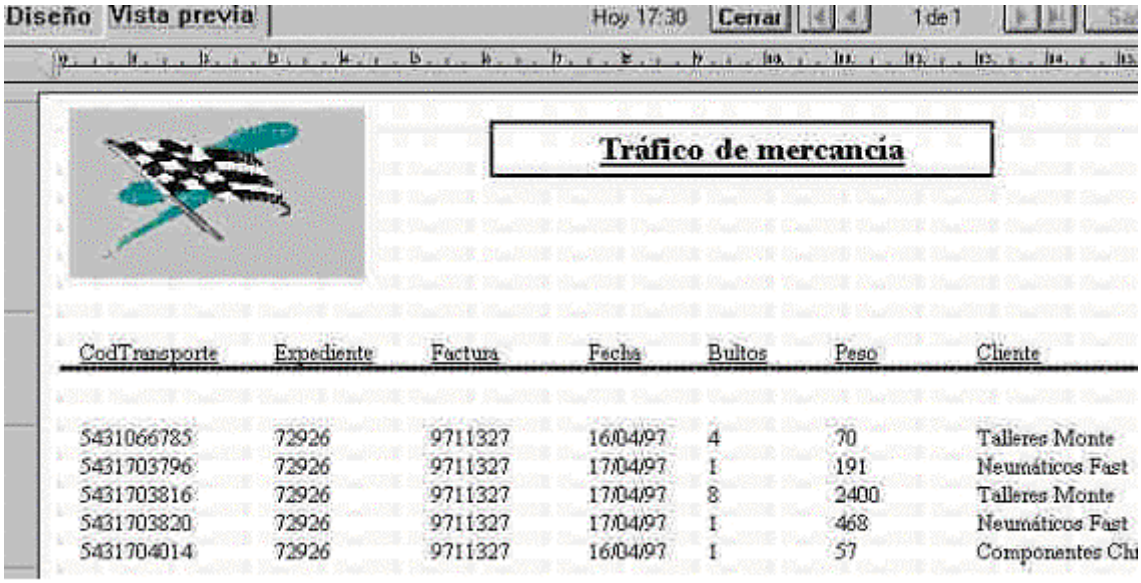


Figura 328. Listado de los registros seleccionados.

Orden de los registros

Además de la selección, el mostrar los registros del informe ordenados sobre la base de un determinado campo, es otra de las tareas más habituales en el trabajo con listados. La opción de CR Informe+Ordenar registros, nos muestra el cuadro de diálogo para establecer el criterio en el que se basa el orden para el listado. El botón en la barra de herramientas es:  .

En el informe con el que estamos trabajando, vamos a realizar una ordenación sobre la base del campo CodTransporte, en la que los registros se mostrarán ordenados en forma descendente. Para ello, realizaremos las selecciones en la ventana de establecimiento de orden (figura 329).

Figura 329. Estableciendo un criterio de ordenación.

Compruebe el lector, la forma en que resultan ordenados los registros, tras las acciones anteriores.

Figura 330. Resultado del informe, después del establecimiento de un orden para los registros.

Selección de campos

Para las situaciones en que debamos efectuar alguna operación global sobre un conjunto de campos del informe, como por ejemplo, cambiar el tamaño o tipo de letra, es posible seleccionar dicho grupo de campos mediante la opción del menú de CR Edición+Seleccionar campos. La forma de selección es igual a la selección de un conjunto de controles en un formulario: hacer clic, y arrastrar, abarcando los campos necesarios, que quedarán marcados y listos para su modificación o desplazamiento a través del informe.

El botón de la barra de herramientas para esta opción es: 

Fórmulas

Una fórmula, es un pequeño programa, escrito en código de Crystal Reports, en el cual se realizan una serie de cálculos, que dan un resultado que puede ser empleado en el informe actual, en forma de campo. Para que el lector identifique más fácilmente este concepto, se asemeja a una macro de las aplicaciones de Office.

Para crear un campo de fórmula, debemos utilizar la opción Insertar+Campo de fórmula, del menú de CR. El botón de la barra de herramientas es: 

La anterior acción, mostrará un cuadro de diálogo que contendrá una lista con las fórmulas existentes para el informe y un TextBox, en el que deberemos introducir el nombre del campo de fórmula a crear, como vemos en la figura 331.

Si ya hubiera fórmulas creadas, la seleccionaríamos de la lista, y al pulsar Aceptar, el campo de fórmula seleccionado se insertaría en el informe, listo para ser situado donde el programador lo necesitara. Si escribimos el nombre de una fórmula nueva, al pulsar Aceptar, se abrirá la ventana del editor de fórmulas (figura 332), para introducir el código de la nueva fórmula.

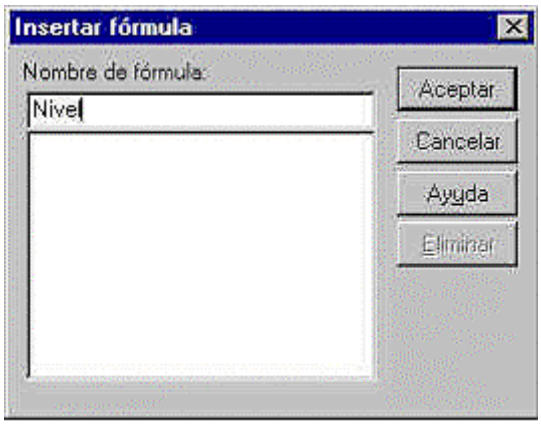


Figura 331. Diálogo para crear una nueva fórmula.

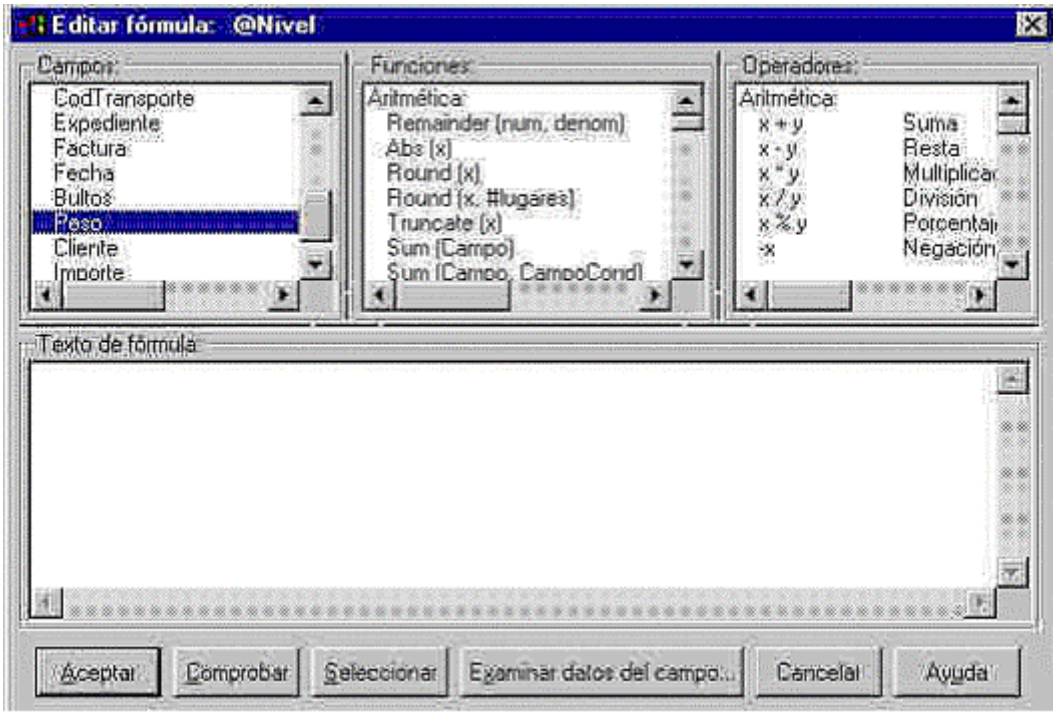


Figura 332. Editor de fórmulas.

Los componentes de esta ventana son los siguientes:

- Campos. Lista con los campos disponibles para ser insertados en el informe. Esto incluye tanto a los campos de las tablas de la base de datos, como a los diferentes campos de fórmulas, resumen, totales, etc., creados por el programador para el informe.

- Funciones. Lista con todas las funciones disponibles en CR, clasificadas por categorías.

- Operadores. Lista que incluye organizados por categorías, los diferentes operadores que se pueden utilizar en CR para construir expresiones.

- Texto de fórmula. Cuadro de texto, en el cual se escribirá el código de la fórmula.

- Aceptar. Botón que se utiliza para guardar la nueva fórmula o los cambios realizados en una fórmula existente. Si el código de la fórmula no estuviera correctamente escrito, se mostraría un mensaje de aviso.

- Comprobar. Este botón, sirve para efectuar una verificación del código de la fórmula, mostrando un mensaje informativo si existen errores o está correcta.

- Seleccionar. Incluye en el informe, el elemento actualmente seleccionado en los controles de lista Campos, Funciones u Operadores.

- Examinar datos del campo. Muestra los valores disponibles para el campo seleccionado. Efectúa la misma tarea que la opción Examinar datos del campo, del menú Edición.

- Cancelar. Suspende la creación o edición del campo de fórmula.

- Ayuda. Muestra la ayuda disponible para la creación de fórmulas.

Combinando los elementos de los controles de lista: Campos, Funciones y Operadores, del editor de fórmulas, e incluyéndolos en el control Texto de fórmula, de este editor, crearemos el campo de fórmula adaptado al cálculo que necesitemos realizar.

Como ejemplo de creación de fórmulas, se adjunta el informe Formula.RPT, que está basado en el informe que hasta ahora nos había servido de ejemplo: Basico.RPT, y que contiene un campo de fórmula, llamado Nivel, el cual comprueba si el valor del campo Peso de la tabla Llegadas es mayor de 1000, previa conversión de dicho campo a número. Si es cierto, Nivel devolverá la cadena "MAYOR", en caso contrario devolverá "MENOR". La fórmula empleada para crear este campo podemos verla en la figura 333.

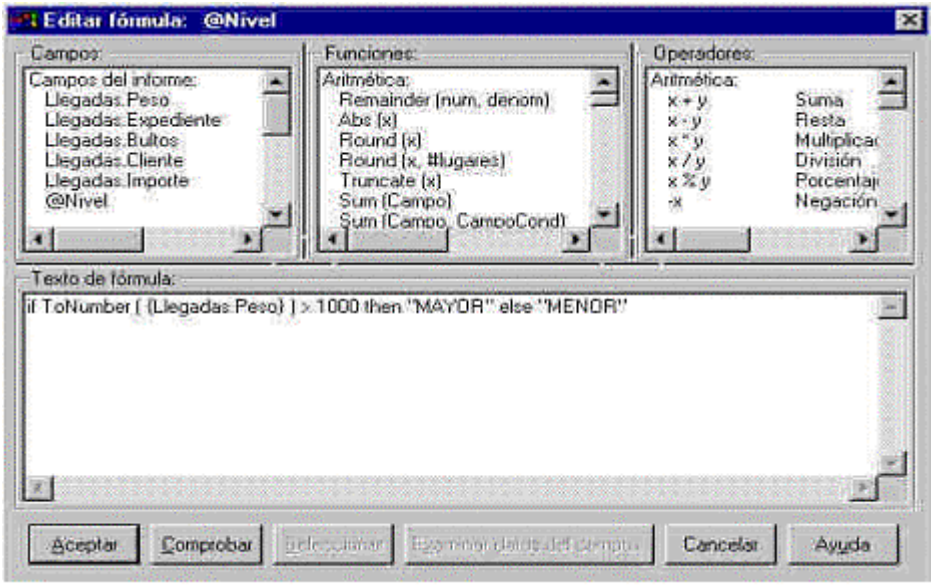


Figura 333. Fórmula Nivel para el informe Formula.RPT.

Dentro del informe, podemos reconocer un campo de fórmula por el caracter "@", que CR adjunta al nombre del campo y que podemos ver en el indicador de campo del informe, en este caso será: @Nivel.

Pulsaremos Aceptar en el editor de fórmulas, e incluiremos el campo de fórmula recién creado en el informe, según muestra la figura 334.

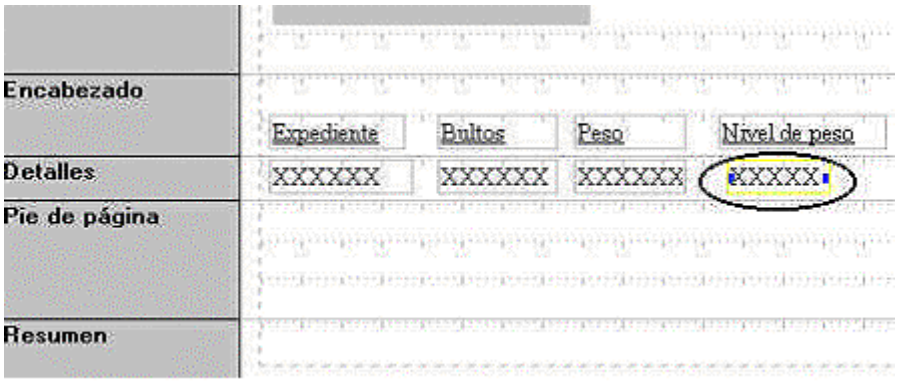


Figura 334. Campo de fórmula Nivel, incluido en el informe.

Al ejecutar el informe, comprobaremos como según el valor del campo Peso, el campo Nivel, muestra una de las dos cadenas disponibles.

Creación de un resumen

Al confeccionar listados, se puede dar el caso de que ciertas columnas repitan un mismo valor a lo largo de un conjunto de filas. Una de las soluciones a esa repetición del valor de un campo para una serie de filas, es agrupar las filas sobre la base del valor de dicho campo y listarlas, situando el valor común del campo al principio o final del grupo de filas.

En Crystal Reports, este problema se puede resolver de una forma muy sencilla creando un resumen para el campo repetido. Vamos a usar como ejemplo, un informe llamado Resum.RPT, que parte del creado en Basico.RPT, incluyendo todos los registros, sin ningún tipo de selección ni orden previos de los registros.

La situación es la siguiente: el lector puede observar que en este informe, el valor del campo Expediente, se repite en un elevado número de filas. Para ello vamos a crear un resumen basado en este campo, de forma que se agruparán las filas por dicho campo y se eliminarán los valores repetidos, indicando el número de expediente sólo una vez para cada grupo de filas.

En primer lugar, seleccionaremos el campo Expediente del informe, accediendo a continuación, a la opción Insertar+Resumen, del menú de CR, que mostrará el cuadro de diálogo, que aparece en la figura 335, para realizar esta acción.

El único control de esta ventana que cambiaremos, será el que indica el nombre del campo por el cual se ordenarán y agruparán los registros, seleccionando el campo Expediente. Seguidamente aceptaremos la ventana, tras lo cual, se creará un nuevo grupo o sección, que agrupará los registros según la información proporcionada en la ventana anterior y un nuevo campo situado en el pie del grupo, que mostrará el valor del campo Expediente.

Puesto que ya tenemos un campo que muestra el número de expediente para un grupo de filas con ese mismo número, no tiene sentido mantener el campo Expediente en la sección Detalles del informe, por lo que procederemos a eliminarlo.

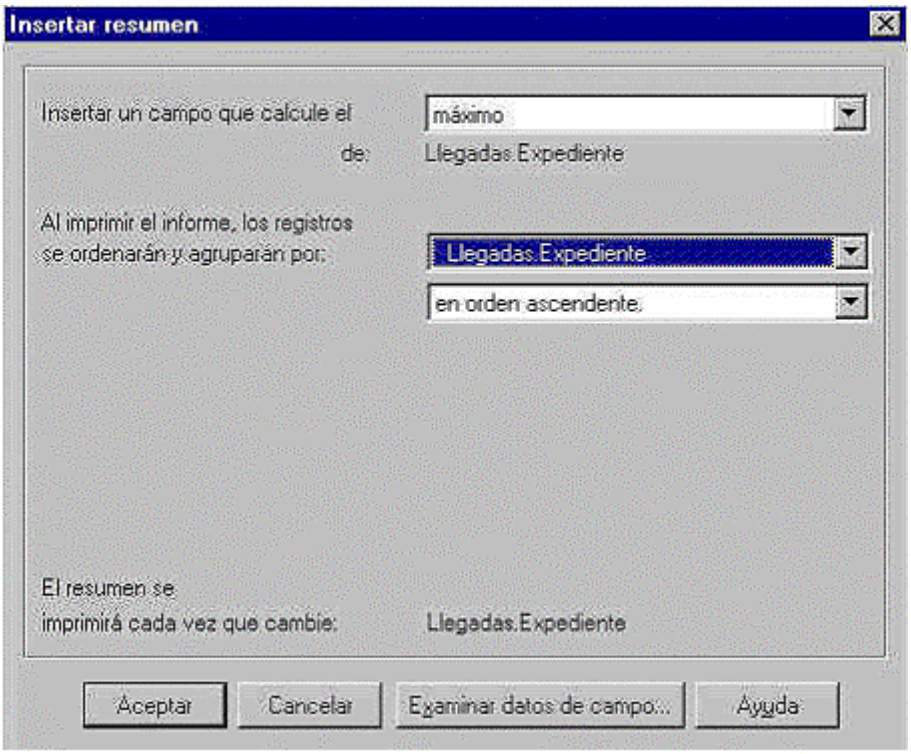


Figura 335. Ventana para insertar resumen.

Por otro lado, el campo creado por el resumen no proporciona un efecto muy estético, por ello lo situaremos en el encabezado del grupo y le aplicaremos unos cambios de formato, para que destaque del resto de líneas de detalle del informe, como puede verse en la figura 336.

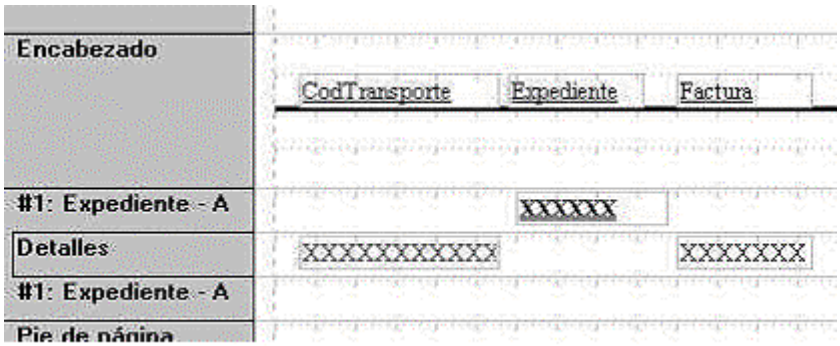


Figura 336. Campo de resumen situado en el encabezado de su grupo.

Al ejecutar el informe, el lector podrá comprobar el efecto de agrupamiento de líneas.

Insertar un Subtotal

Terminada la creación del resumen, si observamos detenidamente el informe, la lógica nos indica que ya que hemos agrupado las filas por número de expediente, lo normal sería crear un subtotal que sumara los importes por cada grupo de expedientes.

La opción de menú Insertar+Subtotal, nos permitirá, una vez hayamos seleccionado un campo numérico, crear un campo de subtotal a partir del campo seleccionado en el informe. En este caso, después de seleccionar el campo Importe y llamar a esta opción de CR, aparecerá la ventana que muestra la figura 337.

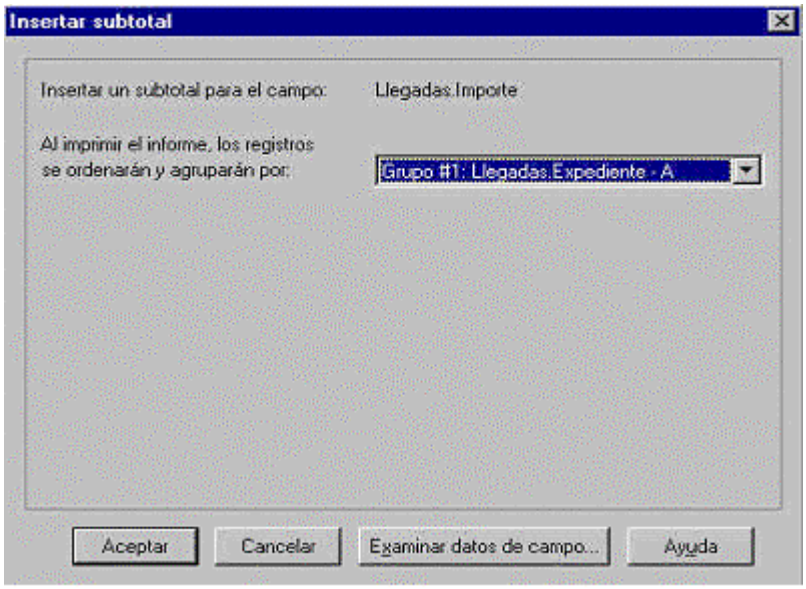


Figura 337. Inserción de un subtotal en el informe.

Después de aceptar esta ventana, se creará el campo de subtotal y se situará por defecto en el pie del grupo, por lo que no necesitaremos moverlo, ya que por norma general, es el mejor lugar para ese tipo de campo.

Si necesitamos crear un campo de total general para el informe, seleccionaremos en este caso la opción Insertar+Total general, procediendo de igual forma que para la creación del subtotal.

Una vez creados los dos campos, situaremos junto a cada uno de ellos, un campo de texto que indique su contenido.

Figura 338. Campos de subtotal y total general para el informe.

Asistentes para la creación de informes

Estos asistentes, facilitan la creación de los tipos más comunes de informes. Al iniciar un nuevo informe, se presentará el diálogo de creación de informes con los asistentes disponibles.

Figura 339. Asistentes para la creación de un informe.

Vamos a crear a continuación un informe, basándonos en las indicaciones de uno de estos asistentes, para explicar su modo de funcionamiento.

Seleccionaremos para este ejemplo, el asistente Referencia, que crea un informe basado en una referencia cruzada. Este tipo de informe, obtiene el valor para un campo, sobre la base de la intersección de valores de otros dos campos.

Usaremos la base de datos Trafico.MDB, que es la que estamos empleando para todos estos ejemplos, y su tabla Llegadas. El cruce se realizará mediante los campos Fecha y Cliente. El primero definirá las filas del informe y el segundo las columnas. El resultante de la intersección, será la suma del campo Importe para los registros que tengan la misma Fecha y el mismo Cliente.

Para ilustrar mejor este concepto, pasemos pues a crear el informe, al final del cual, quedará más claro en que consiste una referencia cruzada.

Después de pulsar el botón correspondiente al asistente Referencia, aparecerá la ventana correspondiente a dicho asistente, como muestra la figura 340.

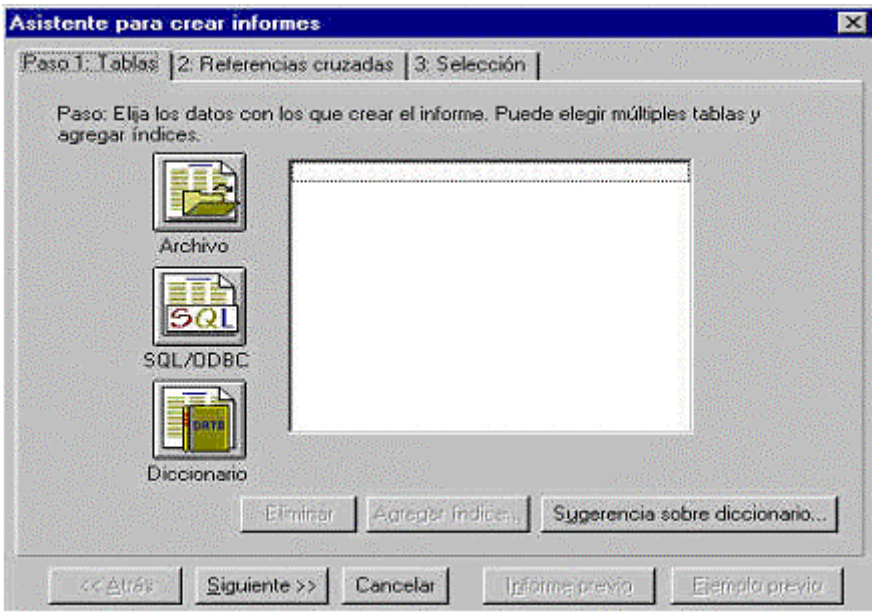


Figura 340. Ventana del asistente para crear informes.

En este primer paso, pulsaremos el botón Archivo, para seleccionar la base de datos con la que vamos a trabajar, operación que ya conocemos de anteriores apartados. Una vez seleccionado el fichero Trafico.MDB, pulsaremos el botón Siguiente, para continuar con el siguiente paso.

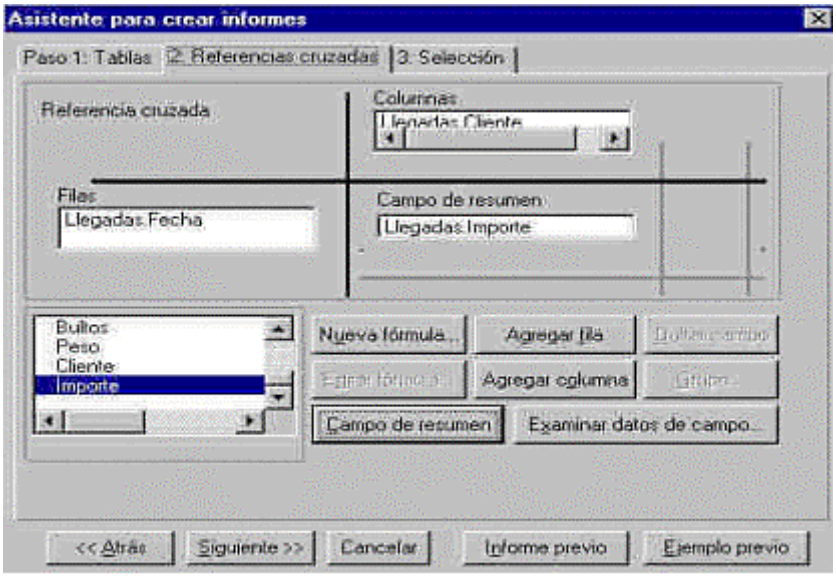


Figura 341. Creación de la referencia cruzada para el informe.

En este paso, debemos establecer los campos que van a componer el informe. En el control de texto perteneciente a las filas, seleccionaremos el campo Fecha y pulsaremos el botón Agregar fila. Para el control que contiene las columnas, seleccionaremos el campo Cliente y pulsaremos el botón Agregar columna. Finalmente, para el Campo de resumen, que proporciona el valor de la referencia cruzada, seleccionaremos el campo Importe y pulsaremos el botón Campo de resumen.

Pulsaremos de nuevo el botón Siguiente, en el caso de que necesitemos seleccionar los registros a listar, lo que mostrará la siguiente pestaña del asistente.

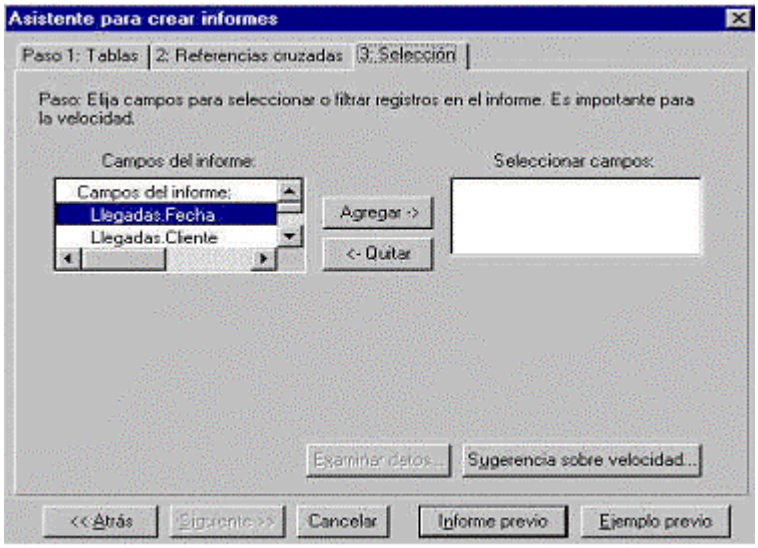


Figura 342. Selección de registros para el informe.

En el caso de que no necesitemos realizar una selección de registros, pulsaremos directamente el botón Informe previo, que completará la creación del informe y lo ejecutará, para poder ver el resultado. Deberemos pasar al modo de diseño, para retocar el ancho de los campos y hacerlos un poco más pequeños, de forma que queden como muestra la figura 343, permitiendo la visualización de toda la información en la página.

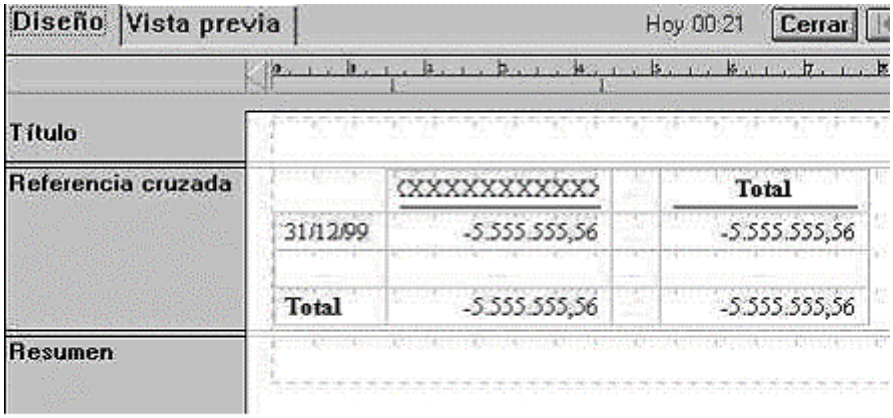


Figura 343. Modo de diseño del informe para referencia cruzada.

A partir de aquí, el programador puede añadir cuantos elementos requiera el informe: título, número de página, fecha, etc., para mejorar su aspecto. Por último, grabaremos el informe con el nombre Asist.RPT.

Como acabamos de comprobar, el empleo de los asistentes incluidos en CR, nos permitirá crear los más variados tipos de informe con el mínimo esfuerzo.

El control Crystal Report

De nada serviría tener el generador de informes más potente, si no dispusiéramos de un medio de ejecutar dichos informes desde nuestras aplicaciones VB. Para ello tenemos el control Crystal Reports, que puede ser incluido en los formularios desde los que se precise ejecutar informes, actuando de puente entre la aplicación y el generador. La figura 344 muestra el icono perteneciente a este control en el cuadro de herramientas.

Figura 344. Control Crystal Report.

El modo de empleo de este control es igual que para cualquiera de los ya existentes, podemos establecer sus propiedades en modo de diseño o en tiempo de ejecución, de una forma fácil y flexible. En este último caso, las propiedades se modificarán mediante código, ya que al ejecutar la aplicación, el control permanece oculto. Algunas de las propiedades más comunes se describen seguidamente.

- Destination. Indica el dispositivo que se va a utilizar para mostrar el informe. Se puede utilizar una de las siguientes constantes:
    - crptMapi. Envía el informe por correo electrónico.

  - crptToFile. Envía el informe a un fichero.
  - crptToPrinter. Lista el informe por la impresora.
  - crptToWindow. Muestra el informe por una ventana.

- CopiesToPrinter. Valor numérico con el número de copias que se van a imprimir del informe.
- ReportFileName. Cadena que indica la ruta y nombre del fichero que contiene el informe.
- SelectionFormula. Cadena con la fórmula de selección de registros para el informe. Es muy importante que esta cadena comience y termine por comillas dobles. Las comillas simples, si son necesarias, se usarán dentro del cuerpo de la fórmula, en caso contrario, los resultados del informe no serán los esperados. Esta opción es de gran utilidad para establecer los registros a listar en tiempo de ejecución. Una forma de utilizar esta propiedad sería como se muestra en el código fuente 253.

```
Informe.SelectionFormula = "{Datos.Nombre} = 'Pedro' "
```

Código fuente 253

Existe un truco para facilitar la creación de la fórmula de selección, en el caso de que esta sea muy compleja, que consiste en crear dicha fórmula desde el generador de informes, empleando el asistente de selección de registros (ver apartado Selección de registros en este mismo tema) y visualizar el código de la fórmula creado por el asistente. Copiaremos el código creado por el asistente y lo asignaremos a esta propiedad del control, modificando allí donde se necesite parametrizar los valores a enviar al informe. Por último eliminaremos del informe la selección de registros creada por el asistente, o de lo contrario, interferirá con los valores que le enviemos desde el control.

- SortFields. Mediante esta propiedad, podemos establecer uno o varios campos para componer el criterio de ordenación de los registros en el informe. La sintaxis para establecer un valor en esta propiedad es la siguiente:

```
Informe.SortFields(ArrayCampos) = "+-{Tabla.Campo}"
```

Esta propiedad contiene un array que almacena los campos a ordenar. Si necesitamos una ordenación ascendente, usaremos el signo más "+" junto al nombre del campo, en caso contrario usaremos el signo menos " -". Cuando el criterio de orden deba estar compuesto por más de un campo, realizaremos tantas asignaciones a esta propiedad como campos compongan la clave del orden, igual que el código fuente 254.

```
Informe.SortFields(0) = "+{Datos.Codigo}"
Informe.SortFields(1) = "+{Datos.Nombre}"
```

Código fuente 254

- DataFiles. Permite especificar una ruta para la base de datos, distinta a la indicada en el diseño del informe. Esta propiedad, al igual que la anterior, se basa en un array de valores, en el que podemos incluir varias rutas de base de datos. La sintaxis es la siguiente:

```
Informe.DataFiles(ArrayFicheros) = "RutaBaseDatos"
```

El uso de esta propiedad, no cambia la ruta de la base de datos establecida en el diseño del informe, sólo afecta durante la ejecución del informe.

```
Informe.DataFiles(0) = "C:\APPLIC\RESULTA.MDB"
```

- Formulas. Esta propiedad permite modificar el contenido de las fórmulas incluidas en el informe. Como las anteriores propiedades, esta también consta de un array en el cual se irán

depositando las fórmulas modificadas. La sintaxis para modificar una fórmula de un informe es la siguiente:

```
Informe.Formulas(ArrayFormulas) = "NombreFormula= NuevoValor "
```

Es muy importante ser cuidadoso con la sintaxis para esta propiedad, ya que en caso contrario, la modificación para la fórmula podría no tener efecto. Por otra parte, no se debe incluir el signo "@" al indicar el nombre de la fórmula.

- WindowXXX. Las propiedades que comienzan por la palabra Window, permiten configurar todos los aspectos de la ventana que mostrará el informe al visualizarlo por pantalla: título, ancho, alto, posición, etc.

Para que el lector pueda comprobar ambos tipos de trabajo, se adjunta la aplicación de ejemplo Imprimir, que pasamos a describir a continuación.

En primer lugar, disponemos de un procedimiento Main() para iniciar la aplicación, que se ocupa de cargar la ventana MDI mdiInicio, desde la cual podremos acceder a las ventanas que ejecutan los informes. El código de este procedimiento, lo podemos ver en el código fuente 255.

```
Public Sub Main()
Dim lmdiInicio As mdiInicio
gcRuta = "C:\CursoVB\Ejemplos\CRInicio\"
Set lmdiInicio = New mdiInicio
Load lmdiInicio
lmdiInicio.Show
End Sub
```

Código Fuente 255

Como punto destacable aquí, debemos señalar la variable gcRuta, definida a nivel global, que contendrá la ruta en la que residen los informes. Si el lector utiliza un directorio diferente, deberá modificar el valor de esta variable.

Desde el menú del formulario mdiInicio, accederemos con la opción Archivo+Informes prediseñados, al formulario frmPredis, que vemos en la figura 345.

Lo que hemos hecho en este formulario, es simplemente insertar un control Crystal Reports, al que llamamos rptInforme y un conjunto de OptionButton, para seleccionar uno de los informes que hemos creado en los anteriores ejemplos de este tema: Basico.RPT, Formula.RPT y Resum.RPT

Al pulsar el botón Imprimir, se comprobará el tipo de informe seleccionado y se ejecutará usando el método PrintReport() del control Crystal Reports, que envía el informe en este caso a una ventana. El código fuente 256, contiene el código para este fuente.

Figura 345. Formulario frmPredis.

```
Private Sub cmdImprimir_Click()
' comprobar el botón de opción pulsado, y listar el
' informe correspondiente
If Me.optBasico.Value Then
   Me.rptInforme.ReportFileName = gcRuta & "Basico.rpt"
   Me.rptInforme.WindowTitle = "Informe básico"
End If
If Me.optFormula.Value Then
   Me.rptInforme.ReportFileName = gcRuta & "Formula.rpt"
   Me.rptInforme.WindowTitle = "Informe con fórmula"
End If
If Me.optResumen.Value Then
   Me.rptInforme.ReportFileName = gcRuta & "Resum.rpt"
   Me.rptInforme.WindowTitle = "Informe resumen"
End If
' enviar el informe a una ventana
Me.rptInforme.PrintReport
End Sub
```

Código fuente 256

Volviendo a la ventana mdiInicio, al seleccionar la opción Archivo+Informes con parámetros, ejecutaremos el formulario frmParam, en el que hace un mayor uso de las propiedades pertenecientes al control CR que incorpora, empleando el informe Param.RPT para listar los registros. La figura 346, muestra este formulario.

Como habrá podido comprobar el lector, el nivel de configuración por parte del usuario, de los diferentes aspectos del informe, es bastante mayor en este formulario. Por un lado, se debe indicar el valor para el campo Expediente de la tabla Llegadas, de forma que se puedan seleccionar los registros a listar, en base a dicho valor. También es posible enviar el listado a una ventana o impresora, indicando el número de copias a imprimir en este último caso. Finalmente, podemos elegir el orden de los registros, en función de varios campos de la tabla. Todo esto ocurrirá al pulsar el botón Imprimir, cuyo fuente vemos en el código fuente 257.

Para cualquier propiedad del control no incluida en este ejemplo, recomendamos al lector, la consulta de la ayuda en VB referente a dicho control, en donde podrá obtener una mayor información sobre todos sus aspectos.

Figura 346. Formulario frmParam.

```
Private Sub cmdImprimir_Click()
' comprobar que se ha introducido el expediente
If Len(Me.txtExpediente.Text) = 0 Then
   MsgBox "Se requiere un número de expediente", , "Error"
   Exit Sub
End If
' establecer el informe a imprimir
Me.rptInforme.ReportFileName = gcRuta & "Param.rpt"
' seleccionar los registros a imprimir
Me.rptInforme.SelectionFormula = "{Llegadas.Expediente} = '" & _
   Me.txtExpediente.Text & "' "
' establecer el órden de los registros
If Me.optCodTransporte.Value Then
   Me.rptInforme.SortFields(0) = "+{Llegadas.CodTransporte}"
End If
If Me.optFecha.Value Then
   Me.rptInforme.SortFields(0) = "+{Llegadas.Fecha}"
End If
If Me.optCliente.Value Then
   Me.rptInforme.SortFields(0) = "+{Llegadas.Cliente}"
End If
' comprobar el destino del informe,
If Me.optVentana.Value Then
   ' si es hacia una ventana
   Me.rptInforme.Destination = crptToWindow

   ' asignar título a la ventana de impresión
   Me.rptInforme.WindowTitle = "Imprimir registros del expediente " & _
      Me.txtExpediente.Text

Else
   ' si se envía a la impresora
   Me.rptInforme.Destination = crptToPrinter
   Me.rptInforme.CopiesToPrinter = Me.txtCopias.Text
End If
' imprimir el informe
Me.rptInforme.PrintReport
End Sub
```

Código fuente 257

Obtención de los datos desde una tabla intermedia

Una técnica alternativa en la selección de datos para un informe, consiste en crear una tabla intermedia, temporal o no, en la base de datos con la que trabajemos. La estructura de dicha tabla contendrá los campos que va a tener nuestro informe y será la aplicación Visual Basic la encargada, en función de las necesidades del usuario, de realizar el volcado de los registros que necesitemos listar.

En este caso, desde Crystal Reports, los campos que deberemos insertar en el informe, serán los correspondientes a la tabla intermedia en lugar de los campos de las tablas normales.

¿Por qué emplear una tabla intermedia?, entre los diversos motivos existentes, podemos argumentar dos principales:

- Cuando la consulta de selección de registros, sea demasiado compleja y estimemos que vamos a tardar un tiempo excesivo en crearla empleando Crystal Reports.

  En este tipo de casos, hemos de ser ante todo prácticos. Crearemos una o más tablas intermedias con los campos necesarios para el informe y como dijimos anteriormente, el informe lo diseñaremos incluyendo los campos de las tablas intermedias en el informe, encargándose la aplicación, de seleccionar la información a listar en las tablas intermedias.

- Cuando la tabla o tablas que contienen la información a imprimir, tengan un elevado número de registros.

  Cuando Crystal Reports imprime un informe en el que hay definida una fórmula de selección de registros, para poder seleccionar los registros en los que se basa la fórmula, debe leer previamente todos los registros de todas las tablas implicadas en el informe, antes de poder imprimirlo.

  Si tenemos una tabla con 100.000 registros, dará igual que tengamos que listar 1.000 que 50.000, el tiempo de preparación del informe será el mismo, ya que en ambos casos, se deberá leer todos los registros de la tabla. Como es natural, a la hora de imprimir, tardará menos el de 1.000 registros que el de 50.000.

  Sin embargo, si desde la aplicación VB, realizamos la selección de registros, traspasándolos a una tabla intermedia, siendo esta tabla la empleada por el informe para imprimir los registros, el ahorro de tiempo será considerable, al realizar la selección desde el programa, ya que sólo se empleará el tiempo necesario en los registros que necesitemos imprimir. En lo que respecta a CR, el tiempo empleado en imprimir, dependerá del número de registros que contenga la tabla intermedia.

  La técnica de impresión desarrollada para la aplicación IVASop, se basa en el empleo de tablas intermedias. Como recordará el lector, en este programa disponíamos del formulario frmImprimir, que vemos en la figura 347, para que el usuario introdujera los valores del trimestre y ejercicio que necesitaba imprimir.

  En la base de datos IVASop.MDB, se han definido las tablas Lineas y Resumen, destinadas a contener los datos para las líneas de detalle e importes agrupados por tipo de IVA respectivamente.

  Al pulsar el botón Aceptar, se ejecutará código del código fuente 258, que se encargará de traspasar los registros de las tablas de trabajo habituales a las tablas intermedias. Una vez realizada esta tarea, el control CR rptIvaSop, se encargará de ejecutar los informes

Lineas.RPT, que imprimirá los registros con los apuntes de IVA y Resumen.RPT, que contendrá los totales separados por tipo de IVA.

Figura 347. Formulario para seleccionar valores de impresión.

```
Private Sub cmdAceptar_Click()
Dim lRCDEjercicio As Recordset
Dim lRCTLineas As Recordset
Dim lRCTTipoIva As Recordset
Dim lRCDSumLineas As Recordset
Dim lRCTResumen As Recordset
Dim lnRespuesta As Integer
Dim lcSQL As String
On Error Resume Next
' comprobar que los campos tienen valor correcto
If Len(Me.txtEjercicio.Text) = 0 Then
   MsgBox "El Ejercicio ha de tener valor", , "Error"
   Exit Sub
End If
If Len(Me.cboTrimestre.Text) = 0 Then
   MsgBox "El Trimestre ha de tener valor", , "Error"
   Exit Sub
End If
If Not IsNumeric(Me.txtEjercicio.Text) Then
   MsgBox "El Ejercicio ha de ser un valor numérico", , "Error"
   Exit Sub
End If
' abrir tabla TipoIVA
Set lRCTTipoIva = gdbIvaSop.OpenRecordset("TipoIva", dbOpenTable)
' seleccionar los registros del ejercicio y trimestre
lcSQL = "SELECT DISTINCT Trimestre FROM EJ" & Trim(Me.txtEjercicio.Text) & "
"
lcSQL = lcSQL & "WHERE Trimestre = '" & cboTrimestre.Text & "' "
' intentar abrir tabla del ejercicio correspondiente
Do While True
   Set lRCDEjercicio = gdbIvaSop.OpenRecordset(lcSQL)
   ' si existe la tabla y no ha habido error, salir de bucle
   If Err.Number = 0 Then
      Exit Do
   End If
   ' si no existe la tabla del ejercicio, terminar...
   If Err.Number > 0 Then
      Err.Clear

      MsgBox "No existen datos del ejercicio " & _
         Trim(Me.txtEjercicio.Text), , "Error"
      Exit Sub
   End If
Loop
If lRCDEjercicio.BOF And lRCDEjercicio.EOF Then
   MsgBox "No existen datos del Ejercicio - Trimestre ", , "Error"
   Exit Sub
```

```
End If
' borrar registros de tabla Lineas
gdbIvaSop.Execute "DELETE * FROM Lineas"
' borrar registros de tabla Resumen
gdbIvaSop.Execute "DELETE * FROM Resumen"
' volcar registros de la tabla del ejercicio
' seleccionado a la tabla intermedia Lineas
lcSQL = "INSERT INTO Lineas "
lcSQL = lcSQL & "SELECT EJ" & Trim(Me.txtEjercicio.Text) & ".*, "
lcSQL = lcSQL & "NIF.NOMBRE FROM NIF INNER JOIN "
lcSQL = lcSQL & "EJ" & Trim(Me.txtEjercicio.Text) & " "
lcSQL = lcSQL & "ON NIF.NIF = EJ" & Trim(Me.txtEjercicio.Text) & ".NIF "
lcSQL = lcSQL & "WHERE Trimestre = '" & Me.cboTrimestre.Text & "' "
gdbIvaSop.Execute lcSQL
' abrir tabla Resumen
Set lRCTResumen = gdbIvaSop.OpenRecordset("Resumen", dbOpenTable)
' sumar los registros de la tabla Lineas,
' agruparlos por tipo de iva y
' pasar el resultado a la tabla Resumen
lRCTTipoIva.MoveFirst
Do While Not lRCTTipoIva.EOF
   lcSQL = "SELECT SUM(BaseImponible) AS XBaseImponible, "
   lcSQL = lcSQL & "SUM(BaseExenta) AS XBaseExenta, "
   lcSQL = lcSQL & "SUM(CuotaIva) AS XCuotaIva, "
   lcSQL = lcSQL & "SUM(TotalFactura) AS XTotalFactura "
   lcSQL = lcSQL & "FROM Lineas "
   lcSQL = lcSQL & "WHERE TipoIva = " & lRCTTipoIva("TipoIva") & " "
   Set lRCDSumLineas = gdbIvaSop.OpenRecordset(lcSQL)
   ' si hay importe en total factura...
   If lRCDSumLineas("XTotalFactura") > 0 Then
      ' agregar registro a tabla RESUMEN
      lRCTResumen.AddNew
      lRCTResumen("Ejercicio") = txtEjercicio.Text
      lRCTResumen("Trimestre") = cboTrimestre.Text
      lRCTResumen("BaseImponible") = lRCDSumLineas("XBaseImponible")
      lRCTResumen("TipoIva") = lRCTTipoIva("TipoIva")
      lRCTResumen("CuotaIva") = lRCDSumLineas("XCuotaIva")
      lRCTResumen("BaseExenta") = lRCDSumLineas("XBaseExenta")
      lRCTResumen("TotalFactura") = lRCDSumLineas("XTotalFactura")
      lRCTResumen.Update
   End If
   lRCTTipoIva.MoveNext
Loop
' enviar listado a destino
If optPantalla.Value Then
   ' listar por pantalla
   rptIvaSop.Destination = crptToWindow
Else
   ' listar por impresora
   rptIvaSop.Destination = crptToPrinter
End If
' indicar la ruta de la base de datos al informe
rptIvaSop.DataFiles(0) = gcRutaDatos & "IVASOP.MDB"
' imprimir informe de líneas de factura
rptIvaSop.ReportFileName = gcRutaDatos & "LINEAS.RPT"
rptIvaSop.PrintReport
' imprimir informe de resumen de iva
rptIvaSop.ReportFileName = gcRutaDatos & "RESUMEN.RPT"
rptIvaSop.PrintReport
End Sub
```

Código Fuente 258

El empleo de tablas intermedias no es absolutamente necesario para resolver el problema de la impresión en este programa, sin embargo, se ha realizado de esta manera para que el lector disponga de un caso práctico con el que comprobar como se aborda este tipo de situaciones.

9

El depurador

¿Qué es un depurador?

Un depurador, es una utilidad o conjunto de utilidades que se suministran junto a una herramienta de programación, para permitir al programador, realizar un seguimiento exhaustivo, de todos los aspectos del código existentes en su programa, de forma que pueda detectar y corregir los posibles errores producidos durante la fase de desarrollo de la aplicación.

El depurador es un elemento tan importante como pueda serlo el compilador, y de su versatilidad a la hora de tratar el código, mediante puntos de interrupción, evaluación de expresiones y otros aspectos propios de este tipo de componente, depende que podamos desarrollar más eficiente y rápidamente nuestro trabajo.

Visual Basic incorpora un excelente conjunto de herramientas de depuración, que nos permitirán abordar la ejecución del código durante la fase de desarrollo de muy distintas formas, ayudándonos en la eliminación de todos los errores de programación existentes.

Para un mayor detalle sobre el lugar en el que se encuentran las diferentes órdenes y opciones del depurador, recomendamos al lector consultar el tema El Entorno de Desarrollo (IDE). No obstante, se indicarán el botón de la barra de herramientas y combinación de teclado para aquellos elementos accesibles mediante el menú de VB.

La aplicación Depurar, se incluye como ejemplo para que el lector pueda realizar las pruebas necesarias con este elemento de VB.

Acciones básicas

Estas acciones proporcionan el funcionamiento mínimo para el depurador. Veamos una breve descripción de las mismas, y a continuación su forma de uso.

- Iniciar/Continuar. Con esta acción, comenzaremos la ejecución del programa desde el entorno de desarrollo de VB, o reanudaremos la ejecución, si sxe encontraba interrumpida.  (*F5*)
- Interrumpir. Para detener momentáneamente la ejecución de la aplicación y entrar en el modo de interrupción, utilizaremos este comando.  (*Ctrl+Interrupción)*
- Terminar. Esta acción, finaliza la ejecución del programa cuando se realiza desde el entorno de VB. 
- Reiniciar. Si la ejecución del programa está interrumpida, el uso de este comando hace que comience de nuevo la ejecución, de igual forma que si hubiéramos pulsado el botón *Iniciar*. (*Mayús+F5*).

Para probar estas órdenes, insertemos un nuevo formulario en un proyecto, asignándole un tamaño que nos permita acceder a las opciones del entorno de VB e incluyendo un control CheckBox y un OptionButton. Pulsemos acto seguido, el botón Iniciar, de forma que el formulario se ponga en ejecución. Si a continuación pulsamos Interrumpir, entraremos en el modo de interrupción, pasando el foco a la ventana *Inmediato* del entorno de VB, que veremos posteriormente.

Pulsando nuevamente Iniciar, se reanudará la ejecución del formulario. Ahora haremos clic en los dos controles del formulario y seleccionaremos la orden Reiniciar del depurador. Lo que pasará a continuación es que, al ejecutarse el formulario de nuevo desde el comienzo los controles aparecerán vacíos, en su estado inicial.

Modo de interrupción

Se puede definir como una parada momentánea cuando se ejecuta un programa desde el entorno de VB. Las formas de invocar este modo son las siguientes:

- Al llegar la ejecución del programa a un punto de interrupción.
- Introduciendo en el código una instrucción *Stop*.
- Al pulsar la combinación de teclado *Ctrl+Enter*.
- Al producirse un error de ejecución.
- Cuando una inspección se define para que entre en este modo, al devolver un valor verdadero cuando es evaluada.

Puntos de Interrupción

Un punto de interrupción, es una línea de código con una marca especial, que produce la detención del programa, al llegar la ejecución a ella. Es posible establecer puntos de interrupción en cualquier línea

ejecutable de la aplicación. No están permitidos en líneas de declaración o comentarios. El comando Alternar puntos de interrupción de VB, nos permite establecer/quitar un punto de interrupción en el código de la aplicación,  (F9). También es posible insertar un punto de interrupción, haciendo clic en el margen, junto a la línea que deba provocar la interrupción. La figura 348 muestra la ventana de código con un punto de interrupción establecido.

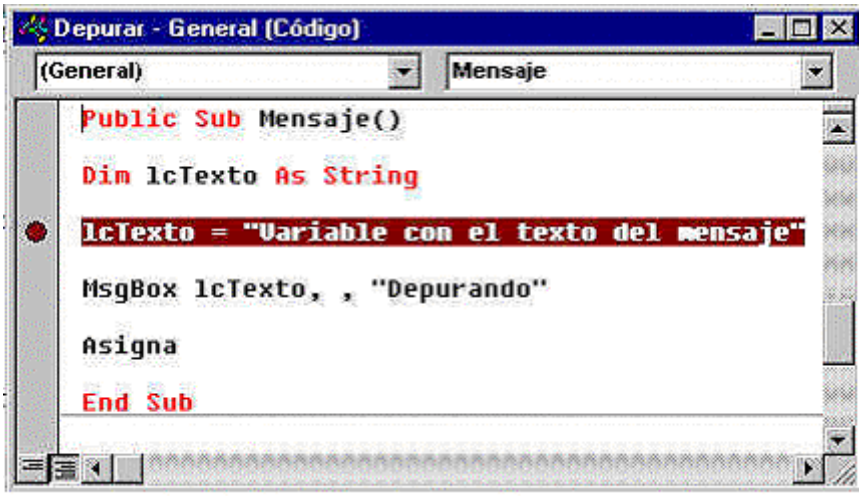


Figura 348. Ventana de código con punto de interrupción.

Una vez que hayamos establecido uno o más puntos de interrupción en el código del programa, lo iniciaremos. Si durante la ejecución, el flujo de la aplicación pasa por alguna de las líneas de código marcadas como de interrupción, el programa se detendrá provisionalmente, pasando el foco a la ventana de código de VB que contiene la línea definida como punto de interrupción, este tipo de ejecución se conoce como modo de interrupción.

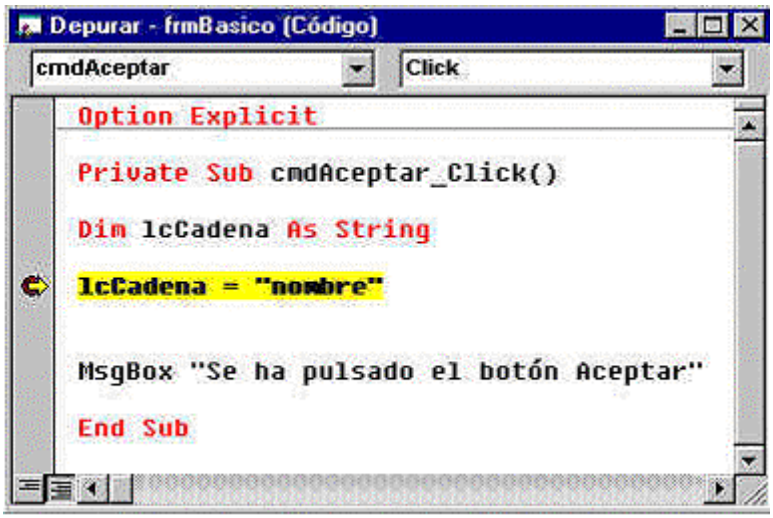


Figura 349. Ejecución detenida en un punto de interrupción.

Aquí podemos comprobar el estado de las expresiones, variables, ejecutar paso a paso, etc., o volver a la ejecución normal pulsando Continuar.

Para eliminar un punto de interrupción utilizaremos el mismo botón de la barra de herramientas o combinación de teclado usada para establecerlo. Si queremos eliminar todos los puntos de interrupción del programa, usaremos el comando Borrar todos los puntos de interrupción  (*Ctrl+Mayús+F9*), de VB.

Ejecución paso a paso

Una vez detenido el programa en un punto de interrupción, la ejecución paso a paso nos permite controlar la ejecución del código situado en el depurador. Las diferentes formas de ejecución paso a paso son las siguientes:

- **Paso a paso por instrucciones.** Ejecuta el código del programa línea a línea, si llega a la llamada a un procedimiento, el depurador se introduce en el código de ese procedimiento, procediendo a ejecutarlo también línea a línea.

   (F8)

- **Paso a paso por procedimientos.** Es similar a paso por instrucciones, que ejecuta cada instrucción del programa. Pero en este caso, si llega a la llamada a un procedimiento, el depurador ejecuta el procedimiento por completo sin mostrarlo al programador, y se sitúa en la siguiente línea después de la llamada al procedimiento.

   (*Mayús+F8)*

- **Paso a paso para salir.** Ejecuta las líneas del procedimiento actual hasta situarse en la primera línea posterior a la llamada del procedimiento en donde nos encontrábamos.

   (*Ctrl+Mayús+F8*)

- **Ejecutar hasta el cursor.** Ejecuta las líneas del programa hasta el punto donde está situado el cursor. Si hubiera un punto de ruptura antes de llegar, la ejecución se pararía en ese punto. (*Ctrl+F8).*

Adicionalmente, en este modo de ejecución por líneas, disponemos de las siguientes órdenes:

- **Establecer instrucción siguiente.** Esta opción hace que la ejecución se desplace al punto seleccionado por el usuario, sin ejecutar las líneas intermedias que pudiera haber entre la línea activa en ese momento y la seleccionada como instrucción siguiente. Es posible establecer, una línea anterior o posterior a la que se ha ejecutado. Esta característica no está disponible entre procedimientos diferentes.

   (*Ctrl+F9)*

- **Mostrar instrucción siguiente.** Sitúa el cursor en la siguiente línea a ejecutar. Esta instrucción es muy útil, si estamos revisando el código y no recordamos en que lugar se encuentra la siguiente instrucción del programa a ejecutar.

Inspecciones

Una inspección, es una expresión definida por el programador, que será evaluada dentro de un contexto, ámbito o zona de alcance durante la ejecución del programa. En función del resultado de dicha evaluación, se realizará una acción determinada por el programador.

Supongamos que disponemos de un formulario con un CommandButton cmdAceptar. Al pulsar dicho botón, se ejecutará el código fuente 259.

```
Private Sub cmdAceptar_Click()

Dim lcCadena As String
Dim lnNumero As Integer
lcCadena = "agregar cadena"
lnNumero = 2350
lnNumero = lnNumero * 2
lcCadena = lcCadena & " al nombre"

End Sub
```

Código fuente 259

Lo que necesitamos en este código, es controlar cuando el valor de la variable lnNumero es superior a 3000. Para ello, vamos a emplear una expresión de inspección.

Emplearemos la opción Depuración+Agregar inspección, del menú de VB para añadir nuevas inspecciones, lo que mostrará la ventana Agregar inspección, en donde introduciremos la expresión que queremos controlar.

Figura 350. Ventana de edición de inspecciones.

Los elementos de esta ventana son los siguientes:

- Expresión. Permite introducir la variable o expresión que será evaluada.

- Procedimiento. En este control podemos seleccionar el procedimiento en el que se realizará la inspección, o todos los procedimientos del módulo.

- Módulo. Aquí podemos indicar un módulo en el que se realizará la inspección, o todos los módulos del proyecto.

  Aunque es posible indicar que la inspección se realice en todos los procedimientos y todos los módulos del proyecto, lo más recomendable es acotar dicha inspección a un procedimiento y módulo si es posible. La razón de esto, reside en que cuando la aplicación entra en un procedimiento en el que hay definida una inspección, dicha inspección es evaluada tras la ejecución de cada línea de código, lo cual nos permite deducir que cuantos más procedimientos deba comprobar una inspección, más lento se ejecutará el programa.

- Expresión de inspección. Si estamos situados en un procedimiento con una inspección definida, se muestra su valor en la ventana Inspección.

- Modo de interrupción cuando el valor sea Verdadero. Cuando durante la ejecución de la aplicación, la evaluación de la expresión a inspeccionar de un resultado verdadero, se interrumpirá la ejecución de igual forma que si se hubiera definido un punto de interrupción en el código.

- Modo de interrupción cuando el valor cambie. Es igual que el anterior, excepto en que entramos en el modo de interrupción cuando cambia el valor de la variable a inspeccionar.

Cuando sea necesario modificar alguno de los valores de una inspección ya establecida, emplearemos la opción Editar inspección (Ctrl+W), del mismo menú Depuración. Esta orden mostrará, la misma ventana usada para agregar inspecciones, pero con la inspección seleccionada, lista para editar.

La ventana de Inspección

Esta ventana es la empleada cuando entramos en el modo de depuración, para comprobar las diferentes inspecciones que hemos definido en la aplicación.

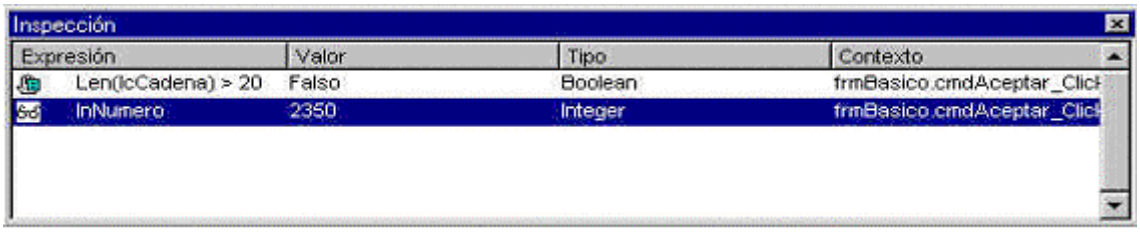


Figura 351. Ventana Inspección visualizando expresiones durante la depuración de un programa.

Las inspecciones son organizadas en forma de filas y columnas. Cada fila o expresión, se compone de la propia expresión a evaluar, el valor actual, resultado de haberla evaluado, el tipo de valor resultante de la expresión y el contexto o lugar del código en que está definida.

Según vayamos ejecutando la aplicación paso a paso, esta ventana reflejará los cambios que se produzcan en las inspecciones, de forma que podamos localizar comportamientos extraños en el código.

Inspección rápida

Esta opción, nos permite ver en modo de interrupción, el resultado de una expresión para la cual no se ha definido una inspección. Para ello, sólo debemos marcar la expresión a evaluar y seleccionar este comando en el menú de VB.

 *(Mayús+F9)*

La figura 352, nos muestra la ventana de código en modo interrupción, con una expresión seleccionada, sobre la que se efectúa una inspección rápida.

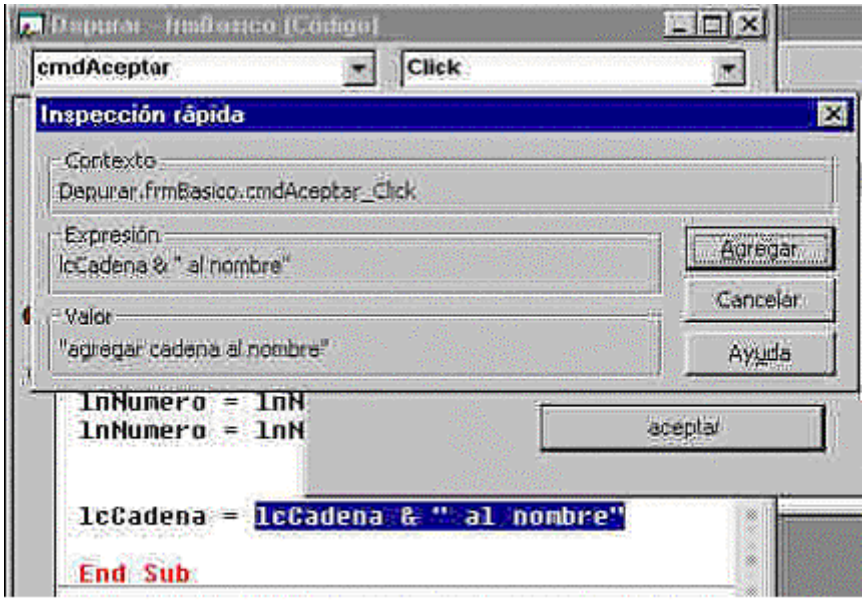


Figura 352. Inspección rápida sobre una expresión.

Como podrá apreciar el lector, la información aquí es muy similar a la ofrecida por la ventana Inspección. Disponemos del contexto en el que se ejecuta la expresión, la expresión seleccionada y el resultado de la misma, existiendo la posibilidad de agregar la expresión a la ventana Inspección.

Si intentamos ver los resultados de las inspecciones cuando no se está ejecutando la aplicación, tanto la ventana de Inspección como la de Inspección rápida no proporcionarán valores, indicándonos que la inspección está fuera de contexto.

La ventana Inmediato

Esta ventana se abre automáticamente, desde el momento en que ejecutamos una aplicación en el entorno de desarrollo de VB, aunque si la aplicación ocupa toda la pantalla, no la veremos a no ser que entremos en el modo de interrupción o cambiemos la tarea activa al entorno de VB.

Si una vez en VB, esta ventana no está disponible, podemos abrirla mediante la combinación de teclado Ctrl+G, o su botón de la barra de herramientas  .

Como su propio nombre indica, podemos ver de forma inmediata, el resultado de expresiones del código que está en ese momento en ejecución, y hacer asignaciones de valores a variables y objetos, siempre que el ámbito de los mismos esté accesible.

La figura 353, muestra esta ventana durante la ejecución de un programa, en concreto el procedimiento de pulsación de un CommandButton. Si queremos ver el valor de una variable o una expresión, debemos escribir la instrucción Print, o el símbolo ?, y a continuación la variable o expresión a comprobar. Después pulsaremos Enter, obteniendo el resultado en la línea siguiente de la ventana. Para ejecutar una sentencia desde esta ventana, la escribiremos directamente y pulsaremos Enter. El código a evaluar en esta ventana, también podemos copiarlo y pegarlo desde la ventana de código en ejecución.

El formulario frmBasico, al que pertenece el código que estamos depurando, y todos sus controles también pueden ser analizados en esta ventana. Como puede comprobar el lector en la figura, estamos visualizando el Caption del formulario, para después cambiar dicha propiedad y el ancho.

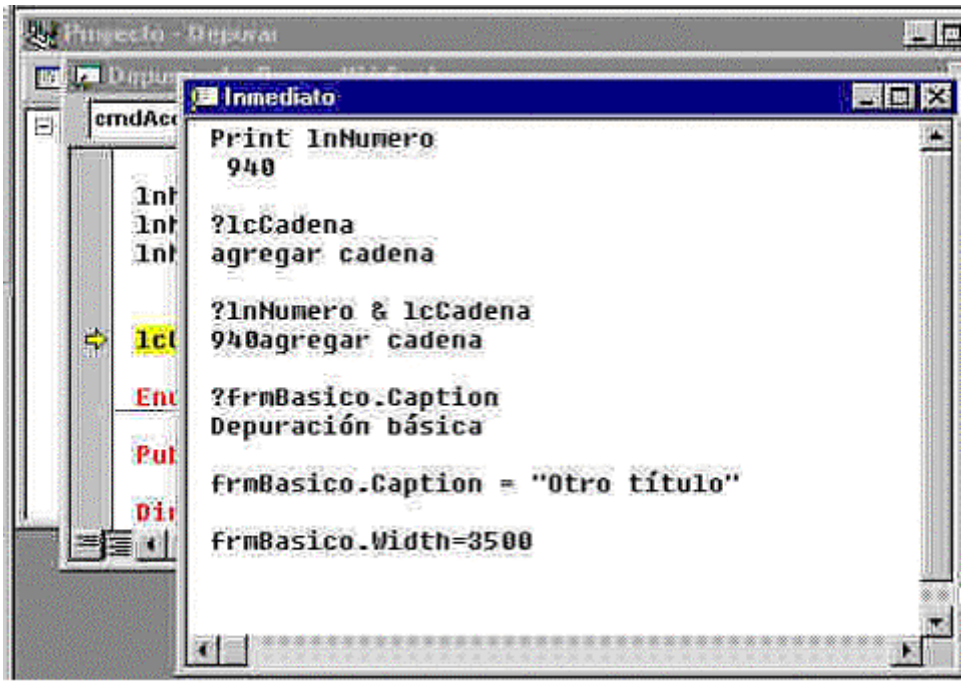


Figura 353. Ventana Inmediato durante la ejecución del programa.

La ventana Locales

Esta ventana visualiza las variables locales declaradas en el procedimiento actual.

La figura 354, muestra esta ventana, cuyos elementos describimos a continuación:

- Contexto. Situado debajo del título de esta ventana, este elemento nos informa del punto del código que se está ejecutando.

- Pila de llamadas. Muestra la ventana de pila de llamadas, que veremos después de este apartado.

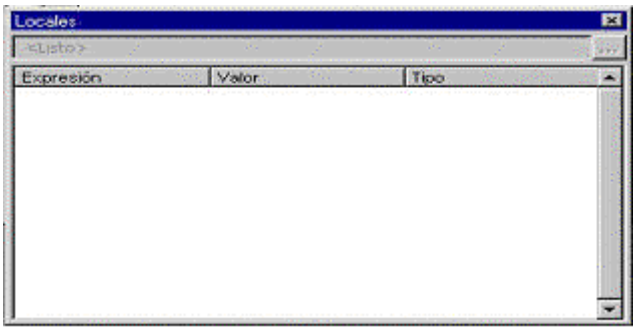


Figura 354. Ventana de información de variables locales.

- Expresión. Esta columna muestra la lista de variables definidas en el procedimiento. Si estamos situados en un módulo de clase, la palabra Me hará referencia al objeto en ejecución de dicha clase. Consulte el lector, el apartado La palabra clave Me, incluido en el tema Programación orientada a objeto. En el caso de estar en un módulo de código estándar, la primera variable será el nombre del módulo.

  Tanto la palabra Me, como el nombre de un módulo estándar, y las variables que contengan objetos, irán acompañadas de los iconos y , que al ser pulsados, desplegarán y contraerán respectivamente, la lista de propiedades y variables a nivel de módulo, que contiene dicho elemento. Esta ventana no permite el acceso a variables globales

- Valor. Muestra el valor de la variable. Es posible modificar el valor visualizado, haciendo clic sobre él, y tecleando el nuevo valor. Una vez modificado, pulsaremos Enter, Flecha Arriba o Fecha Abajo, para aceptar el nuevo valor. Si dicho valor no fuera correcto, el valor seguiría en modo de edición. Para cancelar la edición pulsaremos Esc.

- Tipo. Muestra el tipo de la variable.

La figura 355, muestra esta ventana durante la ejecución de un programa, una vez que se ha entrado en el modo de interrupción, para depurar el código. Los valores reflejados se irán actualizando según vayamos ejecutando el código línea a línea.

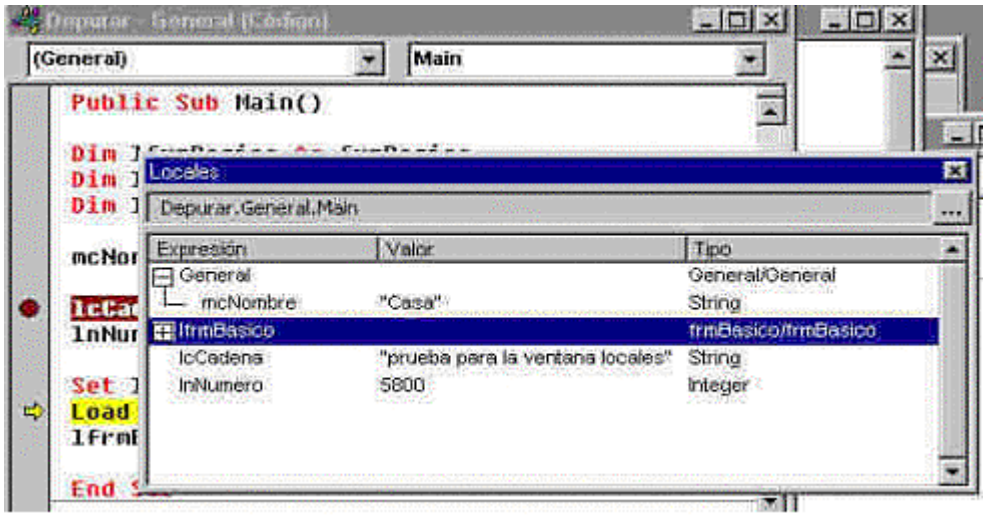


Figura 355. Ventana Locales, mostrando el valor de las variables del procedimiento Main().

Pila de llamadas

Durante la ejecución de un programa, el procedimiento activo es situado en una lista que controla las llamadas entre procedimientos. Si dicho procedimiento, llama a su vez a otro, el nuevo es agregado a la lista pasando a ocupar el primer lugar y así sucesivamente. La ventana Pila de llamadas, muestra esta lista de procedimientos apilados, de forma que el programador pueda comprobar la relación de llamadas efectuadas entre sí por los procedimientos en ejecución.

El código fuente 260, muestra la entrada a una aplicación mediante un procedimiento Main(), este hace una llamada a otro procedimiento Segundo(), y este a su vez, a otro llamado Tercero().

```
Public Sub Main()
MsgBox "Este es el primer procedimiento de la pila"
Segundo
End Sub
' -------------------
Public Sub Segundo()
MsgBox "Este es el segundo procedimiento en la pila"
Tercero
End Sub
' -------------------
Public Sub Tercero()
MsgBox "Este es el tercer procedimiento en la pila"
End Sub
```

Código fuente 260

Si realizamos una ejecución paso a paso desde el modo de interrupción, y al llegar al procedimiento Tercero(), abrimos la ventana de la pila de llamadas (Ctrl+L), se mostrará igual que la siguiente figura.

Figura 356. Ventana Pila de llamadas durante la ejecución de un programa.

Como podemos observar, cada llamada de la ventana, muestra en este orden: el proyecto en ejecución, el módulo y la función, procedimiento, método, etc., que se está ejecutando. La primera línea indica el procedimiento que actualmente se está ejecutando. Si seleccionamos cualquier otra línea, y pulsamos el botón Mostrar, se mostrará la ventana de código de ese procedimiento, en el punto de llamada a uno de los procedimientos de la pila, indicado por la flecha verde.

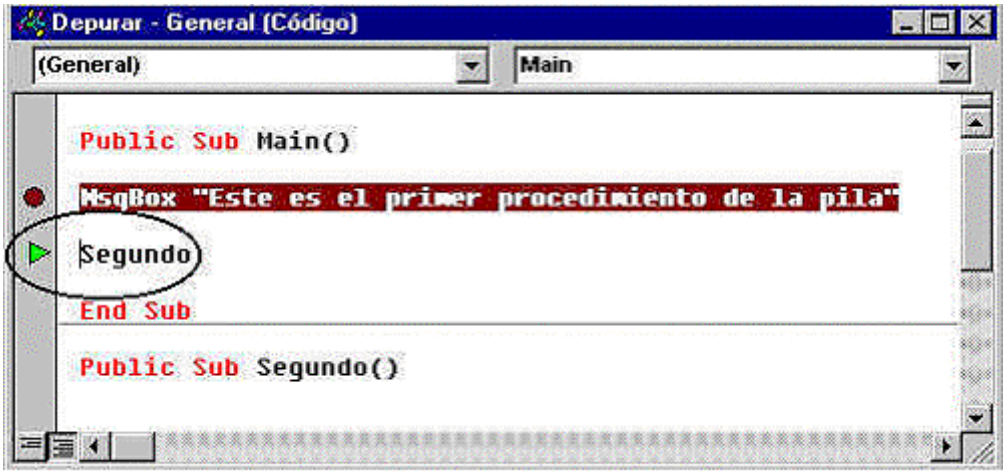


Figura 357. Resultado de la pulsación del botón Mostrar en la ventana Pila de llamadas.

10

Programación orientada a objeto (OOP)

¿Qué ventajas aporta la programación orientada a objetos?

La programación orientada a objeto aparece como una forma de acercar las técnicas de programación al modo en que la mente humana trabaja para resolver los problemas. Tradicionalmente, durante el proceso de creación de una aplicación, la atención del programador se centraba en la forma de resolver las tareas (procesos), y no los elementos (objetos) involucrados en esas tareas.

Pongamos por ejemplo un programa de facturación. Mediante programación procedural, podríamos emplear un código similar al código fuente 261 para crear una factura (el fuente de este ejemplo se basa en pseudocódigo).

```
LOCAL hFact AS Handle
LOCAL nNumFact AS Number
LOCAL cNombre AS Character
LOCAL nImporte AS Number
// abrir el fichero de facturas, la variable
// hFact hace las veces de manejador
// del fichero
hFact = AbrirFichero("Facturas")
// en las siguientes líneas,
// el usuario introduce valores
// en las variables que contendrán
// los datos de la factura
TomarValores(nNumFact, cNombre, nImporte)
// grabar los valores en el fichero de facturas
```

```
GrabarFactura(hFact, nNumFact, cNombre, nImporte)
// cerrar el fichero de facturas
CerrarFichero(hFact)
```

Código fuente 261

Como acabamos de ver, hacemos uso de un conjunto de rutinas o funciones aisladas e inconexas para resolver problemas puntuales, que si analizamos detenidamente, tienen un elemento común denominador, la factura.

Utilizando OOP (Oriented Object Programming), pensamos en primer lugar en la factura como un *objeto*, que tiene un número, fecha, importe, etc., (propiedades), que podemos visualizar, grabar, etc., (métodos). Estamos enfocando nuestra atención en los objetos de la aplicación y no en los procesos que manejan los objetos. Esto no significa que las rutinas que manejan los objetos sean menos importantes, sino que al abordar el diseño de esta manera, la factura pasa a convertirse en un componente, una entidad dentro de la aplicación y no un conjunto de funciones sin nexo común.

Con lo anteriormente expuesto, la grabación de una factura empleando técnicas OOP sería algo similar a lo que aparece en el código fuente 262.

```
' declarar una variable que contendrá el objeto factura
Dim oFactura As Factura
' instanciar un objeto de la clase factura,
' y asignarlo a la variable de objeto
Set oFactura = New Factura
' ejecutar el método de captura de
' datos para la factura
oFactura.TomarValores
' grabar los datos de la factura a la tabla
oFactura.Grabar
' eliminar la referencia al objeto
Set oFactura = Nothing
```

Código fuente 262

El lector se estará preguntando dónde se encuentra el código que abre el fichero de datos en donde se graban las facturas, la inicialización de propiedades del objeto, etc. Bien, pues esta es una de las ventajas del uso de objetos, todo ese trabajo se encuentra encapsulado dentro del código del objeto, y si este se encuentra bien diseñado y construido, no necesitaremos saber nada en lo que respecta a esos puntos, ya que al crearse el objeto, en los diferentes métodos del mismo se realizará todo ese trabajo.

Objetos

Visto lo anterior, podemos decir que un objeto es un conjunto de código creado para acometer una tarea concreta. Que dispone de información o propiedades para consultar y modificar su estado, y de rutinas o métodos que definen su comportamiento.

Clases

Una clase es un conjunto de objetos que tienen características comunes. También se podría decir que es un grupo de especificaciones a partir del cual se crean los objetos, como una especie de plantilla o plano.

Imaginemos una plancha utilizada en una imprenta para fabricar impresos, la plancha se puede identificar con la *Clase Impreso*, y cada impreso que se obtiene a partir de ella es un *Objeto Impreso*.

Una clase es una entidad abstracta, puesto que nosotros manejamos objetos, no clases, pero nos ayuda a mantener una organización. La clase Coche es algo intangible, mientras que un objeto Coche podemos tocarlo y usarlo. Es esta misma abstracción la que nos permite identificar a que clase pertenecen los objetos, ¿por qué motivo?, pues porque tomando el ejemplo de la clase coche, todos los objetos de esta clase no son exactamente iguales.

Un coche deportivo tiene una línea más aerodinámica, un motor más potente...; un coche familiar es más grande que un deportivo, su maletero es mayor, aunque el motor es menos potente...; una furgoneta sustituye el maletero por una zona de carga, puede soportar un mayor peso, etc. Pero todos los objetos de esta clase tienen una serie de características (propiedades) comunes: volante, ruedas, motor, etc., y un comportamiento (métodos) igual: arrancar, parar, girar, acelerar, etc., que nos permiten reconocerlos entre los objetos de otras clases. La capacidad de reconocer la clase a la que pertenecen los objetos es la abstracción.

Las diferencias entre objetos de una misma clase vienen dadas por las propiedades particulares de cada objeto. Continuando con el ejemplo de los coches, dos objetos de la clase coche familiar, pueden ser creados incluyendo en uno la propiedad asiento de cuero, y en el otro no. De igual manera, durante el periodo de duración del coche, su dueño puede cambiarle la propiedad radio dejándola vacía, o por otra más moderna. Los dos objetos seguirán perteneciendo a la clase coche familiar, pero existirán diferencias entre ambos impuestas por los distintos valores que contienen sus propiedades.

La organización de las clases se realiza mediante jerarquías, en las cuales, los objetos de clase inferior o clase hija heredan las características de la clase superior o clase padre. La figura 358 muestra como se organiza la jerarquía de medios de transporte. La clase superior define a los objetos más genéricos, y según vamos descendiendo de nivel encontramos objetos más específicos.

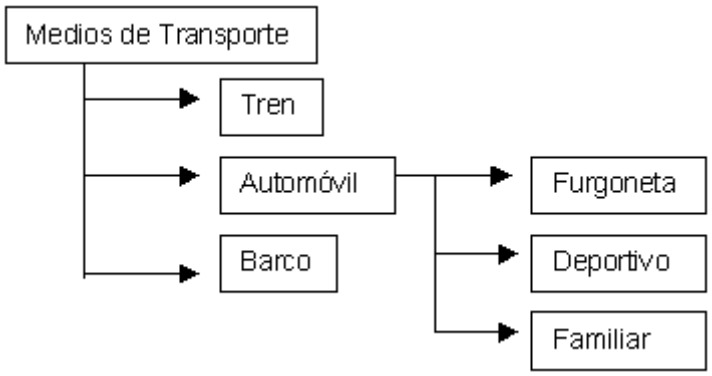


Figura 358. Jerarquía de la clase Medios de Transporte.

Clases abstractas

Una clase abstracta es aquella de la que no se pueden crear objetos, estando definida únicamente a efectos organizativos dentro de la jerarquía.

Tomemos de la jerarquía vista en el apartado anterior y veamos en la figura 359 la rama Automóvil.

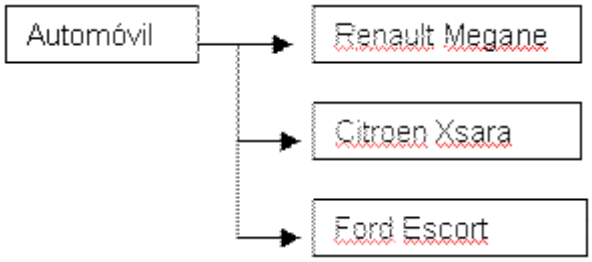


Figura 359. Rama Automóvil de la jerarquía de la clase Medios de Transporte.

Mientras que no podemos crear un objeto directamente de la clase Automóvil, si podemos hacerlo de cualquiera de las clases que dependen de ella. Esto es debido a que Automóvil como objeto no puede existir, es una entidad genérica, sienta las bases y especificaciones para que sus clases hijas puedan crear objetos automóviles *pero* de una determinada marca y modelo, que son los que sí podemos tocar y manejar.

Otro ejemplo de clase abstracta lo podría constituir la anterior clase Impreso, de la cual heredarían las clases Albarán, Factura y Carta. La clase Impreso no puede crear objetos, no existen objetos impreso, pero sí existen impresos de factura, carta, recibo, aviso, etc.

Relaciones entre objetos

Los objetos dentro de una aplicación se comunican entre sí mediante una serie de relaciones, a continuación hacemos un breve repaso de las mismas.

Herencia

En ocasiones, los objetos de una determinada clase, heredan a su vez de otra clase que ocupa un nivel superior en la jerarquía. Por ejemplo, un objeto de la clase Factura hereda de la clase Impreso, un objeto Albarán también hereda de la clase Impreso. A estas clases heredadas también se las conoce como subclases.

Cuando se crea una nueva clase heredada, la nueva clase dispone de todas las propiedades y métodos de la clase padre, mas el código implementado dentro de la propia clase.

Una forma de reconocer este tipo de relaciones, es realizar un análisis sintáctico de la misma usando las partículas "hereda de" o "es un", por ejemplo: "La clase Factura hereda de la clase Impreso" o "Un objeto Factura es un Impreso".

Pertenencia

Los objetos pueden contener o estar formados a su vez por otros objetos. Un objeto coche está formado por los objetos motor, volante, rueda, asiento, etc. Se dice en este caso que hay una relación de pertenencia, ya que existe un conjunto de objetos que pertenecen a otro objeto o se unen para formar otro objeto. A este tipo de relación se le denomina también Contenedora.

La forma de reconocer este tipo de relaciones, es realizar un análisis sintáctico de la misma usando la partícula "tiene un", por ejemplo: Un objeto de la clase Coche tiene un objeto de la clase Volante.

Utilización

Hay situaciones en que un objeto utiliza a otro para realizar una determinada tarea, sin que ello suponga la existencia de una relación de pertenencia entre dichos objetos.

Por ejemplo, un objeto Formulario puede utilizar un objeto Cliente para mostrar al usuario las propiedades del cliente, sin la necesidad de que el objeto cliente sea una propiedad del objeto formulario.

Nótese la importante diferencia entre esta relación y la anterior, ya que aquí, el objeto formulario a través de código, creará o le será pasado como parámetro un objeto cliente, para poder mostrarlo en el área de la ventana.

La forma de reconocer este tipo de relación, es realizar un análisis sintáctico de la misma, usando la partícula "usa un". Por ejemplo: Un objeto de la clase Form usa un objeto de la clase Cliente.

Elementos básicos de un sistema OOP

Encapsulación

Esta característica nos indica que las propiedades de un objeto sólo pueden ser manejadas por el propio objeto. De esta manera, la información del objeto permanece libre de modificaciones exteriores que pudieran hacer que el objeto se comportara de forma errónea.

Para recuperar/asignar el valor de una propiedad del objeto, se recomienda escribir métodos que comprueben el valor antes de asignarlo a la propiedad, y que realicen un formateo del valor a mostrar en el caso del acceso a la propiedad.

En VB disponemos de los métodos o procedimientos Property para lograr encapsulación, evitando el manejo directo de las propiedades del objeto.

Polimorfismo

Según este principio, dos métodos con el mismo nombre pueden realizar diferentes operaciones en función del objeto sobre el que se aplique. La ventaja de esta característica es que el programador no necesita preocuparse de cómo funciona el método en su interior, sólo ha de utilizarlo.

De esta forma, un método llamado Abrir, tendría un comportamiento diferente al ser usado con un objeto Ventana o con un objeto Fichero.

Para conseguir polimorfismo en VB, disponemos de los llamados *Interfaces*, que veremos a lo largo de este tema,.

Herencia

Esta es la característica más importante de la OOP mediante la cual cuando definimos una clase hija que hereda de una superior, los objetos creados a partir de la clase hija, por el mero hecho de haber establecido la relación de herencia y sin tener que escribir ni una sola línea de código, contendrán todas las propiedades y métodos de la clase padre. Naturalmente, podremos añadir propiedades y métodos específicos para la clase hija que sirvan para completar su funcionalidad.

Tomemos como ejemplo una clase *Ventana*, que tiene como propiedades las coordenadas de la ventana, y un método *Mover()* para cambiarla de posición en pantalla. Ahora podemos crear una nueva clase llamada *VentNombre* que herede de *Ventana* y agregarle la propiedad *Nombre* para mostrar un título en la ventana, y el método *CambiarTam()* para redimensionarla, con lo que aprovecharemos todo el código de la clase padre y sólo tendremos que ocuparnos de crear la nueva funcionalidad para esta clase.

VB no permite en la actual versión una herencia *pura*, por decirlo de alguna manera. Podemos lograr una cierta comunicación entre las clases mediante el uso de Interfaces, aunque no será el tipo de herencia al que estamos acostumbrados si hemos trabajado con otras herramientas que proporcionan herencia auténtica.

Reutilización

Este es uno de los objetivos perseguidos por la OOP. Un objeto bien diseñado, puede ser utilizado en otra aplicación, bien directamente o heredando una clase hija de él, que aproveche todo lo que tiene y a la que añadiríamos nuevas funcionalidades. Todo esto supone un considerable ahorro en el tiempo de desarrollo de la aplicación.

Creación de clases

Después de las anteriores nociones teóricas, veamos a continuación los pasos necesarios para construir una clase en VB. El lector dispone en la aplicación OOClien, de una clase básica como ejemplo, con sus propiedades y métodos.

Crear un módulo de clase

En primer lugar, debemos abrir desde el IDE de VB un nuevo proyecto, o uno existente. Seguidamente, mediante la opción de menú *Proyecto+Agregar módulo de clase*, insertaremos un módulo de clase en el proyecto. Este tipo de módulo contiene la definición y el código para una clase, por lo que deberemos añadir un módulo de este tipo por cada clase que incorporemos al proyecto.

Los módulos de clase se guardan en ficheros con extensión .CLS, en nuestro ejemplo será Cliente.CLS

En la figura 360 utilizaremos la ventana de propiedades para asignar en la propiedad Name el nombre de la clase; el resto de propiedades de la clase se verán en el tema dedicado al tratamiento de datos con ADO.

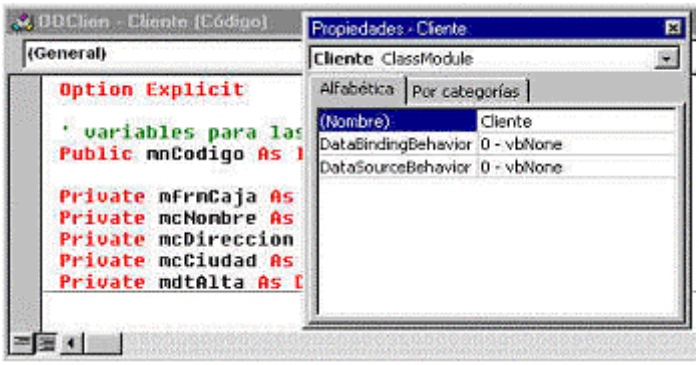


Figura 360. Ventana de propiedades para asignar el nombre de una clase.

Definir las propiedades de la clase

Definir una propiedad para una clase, es tan sencillo como declarar una variable pública o privada a nivel del módulo. Veamos un ejemplo de cada una, en el código fuente 263.

```
Public mnCodigo As Integer
Private mcNombre As String
```

Código fuente 263

Estas propiedades, como se ha comentado anteriormente, constituyen la información que contendrá cada objeto creado a partir de la clase.

Acceder a las propiedades de la clase

Podemos acceder a las propiedades de una clase, en función del ámbito (privado o público) de la propiedad.

Si la propiedad se ha declarado como pública, la forma de uso variará según el lugar desde el que accedamos a ella. Desde el propio módulo de clase, como se muestra en el código fuente 264.

```
Dim lnValor As Integer
........
........
Me.mnCodigo = 521 ' para asignar un valor
lnValor = Me.mnCodigo ' para recuperar el valor
........
........
```

Código fuente 264

La palabra clave Me, así como la forma de crear objetos de una clase, se comentan en sucesivos apartados.

Desde otro módulo del proyecto, como muestra el código fuente 265.

```
Dim lnValor As Integer
Dim oCliente As Cliente ' declarar una variable de objeto
Set oCliente = New Cliente ' crear un objeto y asignarlo a la variable
........
........
oCliente.mnCodigo = 521 ' para asignar un valor a una propiedad
lnValor = oCliente.mnCodigo ' para recuperar el valor de una propiedad
........
........
```

Código fuente 265

El lector habrá notado que la manipulación de las propiedades de una clase, es igual aquí que para un módulo de formulario, de hecho, si hemos trabajado anteriormente con VB y formularios, sin emplear módulos de clase, ya hemos trabajado con clases y objetos. Esto es debido a que un módulo de formulario contiene la definición de una clase que hereda de la clase Form. La propiedad Name del formulario identifica a la clase, de igual forma que en un módulo de clase. La ventaja que tenemos, es que todo el mecanismo fundamental de funcionamiento del formulario: propiedades, métodos, eventos, etc., nos viene ya implementado, siendo posible añadir cuanto código adicional necesitemos.

Si la propiedad se ha declarado como privada, que es lo más conveniente, haremos uso de los llamados procedimientos Property, para acceder a sus valores.

¿Qué son los procedimientos Property?

Los procedimientos Property se utilizan para proporcionar acceso al valor de las propiedades privadas definidas dentro de un módulo de clase.

Este tipo de procedimientos se declaran públicos por defecto, siendo esta la manera en que el código externo a la clase puede acceder a sus propiedades, ya que al haberse declarado las propiedades como privadas, nos será imposible acceder a ellas desde el exterior. Visto lo anterior, podemos afirmar que los procedimientos Property hacen de puente entre la aplicación y el contenido de la clase.

Aunque un procedimiento Property se declare público por defecto, también es posible declararlo privado, de modo que sólo la clase pueda acceder a la propiedad,

En otras herramientas de programación con capacidades de orientación a objeto, a este tipo de procedimientos se les conoce como *métodos de Acceso/Asignación*, lo cual es lógico, si pensamos que no dejan de ser métodos, aunque con una finalidad especial, que es la de tratar con las propiedades de la clase.

Tipos de procedimientos Property

Disponemos de tres modos de comunicarnos con las propiedades de una clase.

- **Property Get.** Retorna el valor de una propiedad de la clase. Adicionalmente, este procedimiento puede contener el código para devolver el valor de la propiedad formateada,

como pueda ser incluir el año completo para las fechas, puntos decimales en propiedades numéricas, etc. Este tipo de procedimiento es similar a Function, ya que devuelve un valor asignado al propio nombre del procedimiento.

En el código fuente 266, podemos ver este procedimiento aplicado a la propiedad Nombre de la clase Cliente, en su modo más sencillo de uso, asignando el valor de la propiedad a la denominación del procedimiento.

```
Public Property Get Nombre() As String
Nombre = mcNombre
End Property
```

Código fuente 266

Dentro de una aplicación, lo usaríamos como indica el código fuente 267.

```
' declarar variables para el objeto
' y para guardar un valor del objeto
Dim loCliente As Cliente
Dim lcDato As String
' crear un objeto
Set oCliente = New Cliente
..........
..........
' recuperar el valor de una propiedad del objeto
' y asignarlo a una variable
lcDato = oCliente.Nombre
..........
..........
```

Código fuente 267

- **Property Let.** Asigna un valor a una propiedad de la clase. Gracias a este tipo de procedimientos, podemos comprobar que los valores para la propiedad son correctos antes de asignarlos definitivamente, así como realizar operaciones de formato con dichos valores.

  En el código fuente 268, vemos un procedimiento de este tipo para la propiedad Nombre de la clase Cliente. Como podrá comprobar el lector, la variable de propiedad mcNombre, es la que guarda el valor asignado al procedimiento Property.

```
Public Property Let Nombre(ByVal vcNombre As String)
mcNombre = vcNombre
End Property
```

Código fuente 268

Dentro de una aplicación, lo usaríamos como se muestra en el código fuente 269.

```
' declarar variables para el objeto
Dim loCliente As Cliente
' crear un objeto
Set oCliente = New Cliente
..........
```

```
..........
' asignar valor a una propiedad del objeto
oCliente.Nombre = "Luis Palermo"
..........
..........
```

Código fuente 269

En la propiedad Alta, queda más patente la forma de realizar comprobaciones sobre el valor que se asigna a la propiedad. Si el mes de la fecha que asignamos coincide con el de la fecha actual, se pasa el valor a la propiedad, en caso contrario, se avisa al usuario, sin modificar el valor que la propiedad tuviera.

```
Public Property Let Alta(ByVal vdtAlta As Date)
Dim lnMesActual As Integer
Dim lnMesAlta As Integer
lnMesAlta = Month(vdtAlta)
lnMesActual = Month(Date)
If lnMesAlta = lnMesActual Then
   mdtAlta = vdtAlta
Else
   MsgBox "El mes del alta ha de ser el actual", , "Error"
End If
End Property
```

Código fuente 270

El paso de parámetros a los procedimientos Property es igual que en el resto de procedimientos en VB, se puede hacer por valor o referencia. Lo más recomendable, si queremos mantener el objeto lo más aislado posible y conseguir el máximo rendimiento, es utilizar el paso por valor, excepto cuando se trate de arrays de tipo Variant o cadenas de caracteres de gran tamaño, en que se deberá usar el paso por referencia.

- **Property Set.** Este procedimiento es una variante de Let, ya que se utiliza para asignar un valor a una propiedad, con la particularidad de que el valor a asignar ha de ser un objeto.

  Supongamos que en la aplicación de ejemplo, hemos añadido la propiedad mfrmCaja a la clase Cliente. La nueva propiedad contendrá un objeto de tipo formulario llamado frmDatos, como vemos en el código fuente 271.

```
Private mfrmCaja As frmDatos
```

Código fuente 271

Como la propiedad es privada, proporcionamos su acceso creando el oportuno par de procedimientos Property, pero esta vez serán Get/Set en vez de Get/Let, debido a que la nueva propiedad contiene un objeto.

```
Public Property Set Caja(ByVal vfrmCaja As frmDatos)
Set mfrmCaja = vfrmCaja
End Property
```

```
' ------------------------------------
Public Property Get Caja() As frmDatos
Set Caja = mfrmCaja
End Property
```

Código fuente 272

Dentro de una aplicación, lo usaríamos como se indica en el código fuente 273

```
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' declarar una variable para contener
' un objeto de la clase frmDatos
Dim lfrmDatos As frmDatos
' instanciar un objeto de la clase Cliente
Set loCliente = New Cliente
' instanciar un objeto de la clase frmDatos
Set lfrmDatos = New frmDatos
' asignar el objeto lfrmDatos a la propiedad
' Caja del objeto loCliente
Set loCliente.Caja = lfrmDatos
```

Código fuente 273

Tener que utilizar un método o procedimiento para acceder a una propiedad, puede resultar un poco chocante, ya que habituados a la forma de manejar las variables en el código, nuestra tendencia es utilizar las propiedades de la misma forma. Sin embargo, una de las normas de la OOP es no permitir *nunca* acceso directo a las propiedades de un objeto desde el exterior del mismo (la encapsulación no tendría sentido en ese caso), por este motivo se proporcionan los procedimientos Property en VB, y los métodos de Acceso/Asignación en otros lenguajes. A este respecto, un punto interesante se encuentra en el hecho de que las propiedades públicas de una clase, son convertidas internamente por VB en pares de procedimientos Get/Let, de forma que el lector puede hacerse una idea de la importancia que se le otorga a este aspecto en VB.

Insistiendo en la importancia que los procedimientos Property tienen en VB, también debemos comentar que para esta herramienta, las propiedades son identificadas sólo mediante este tipo de procedimiento. Como ejemplo, tenemos los asistentes que manipulan clases, al visualizar las propiedades de una clase, muestran los nombres de los procedimientos Property. Nosotros pensamos que una propiedad, es realmente el elemento de código que contiene el valor de esa propiedad. En el caso de VB, sería la variable de propiedad definida en el módulo de clase, mientras que los procedimientos Property cumplen la tarea de dar acceso al valor de la variable de propiedad desde el exterior de la clase, y formatear o validar el valor a establecer en la variable de propiedad.

Debido a esta estrecha relación entre las variables de propiedad y los procedimientos Property, emplearemos el término *propiedad*, para referirnos tanto a uno como otro elemento de una clase.

Ventajas de los procedimientos Property

Aunque el empleo de propiedades públicas en la clase, es la manera más fácil y rápida de definir propiedades, también es la menos recomendable. Siempre que sea posible es mejor utilizar la combinación, propiedades privadas – procedimientos Property. Algunas de las ventajas, se enumeran a continuación.

- **Encapsulación.** Supongamos que tenemos una cadena que debe ser convertida a mayúsculas antes de asignarla a la propiedad de un objeto. Empleando un método Property Let para esa propiedad, podemos hacer la conversión dentro del procedimiento en vez de realizar una conversión previa en cada punto del código antes de asignar la cadena a la propiedad. Con esta forma de trabajo, por un lado nos despreocupamos de hacer la conversión en múltiples puntos a lo largo del código; y por otro al tener el código de conversión centralizado en un solo lugar, si necesitáramos hacerle un retoque en cualquier momento, sólo debemos modificar una vez. De la otra forma tendríamos que estar buscando en todo el proyecto los lugares en los que se realiza la conversión, y modificar las correspondientes líneas de código.

- **Hacer que la propiedad sea de *Sólo lectura* o *Sólo asignación*.** Puede darse el caso de que al crear un objeto, este tenga una propiedad que guarde la fecha y hora de creación para mantener cualquier tipo de estadística. Si necesitamos acceder a dicha propiedad, dispondremos del correspondiente Property Get para visualizar el contenido de la propiedad. Dado que no es conveniente modificar ese valor, para no influir en el resultado de la estadística, podemos hacer dos cosas:

  - No escribir el procedimiento Property Let para esa propiedad. De esta forma, al ser la propiedad privada, no habrá medio de que desde el exterior del objeto pueda alterarse la propiedad que contiene la fecha.

    El uso de procedimientos Property, no obliga a crear un conjunto Get/Let o Get/Set. Es perfectamente válido escribir sólo uno de ellos, y no crear el otro si no es necesario.

  - Escribir el procedimiento Property Let declarándolo privado. Así, sólo el código de la propia clase podrá asignar valor a la propiedad, y como es natural, el programador ya se encargará de que el código de la clase no altere erróneamente los valores de la propiedad, y la estadística no resulte inexacta.

  Piense el lector, que en este anterior caso, si la propiedad se hubiese declarado pública, cualquier elemento de la aplicación podría alterar su valor accidentalmente, con los problemas que ello acarrearía.

- **Crear Propiedades Virtuales.** Mediante los procedimientos Property, es posible acceder a valores que no se corresponden directamente con una propiedad declarada en la clase, sino que son el resultado de las operaciones efectuadas entre varias propiedades.

  Supongamos que en la clase Cliente necesitamos saber los días transcurridos entre la fecha de alta del cliente (propiedad Alta) y la fecha actual del sistema. La opción más lógica es declarar una nueva propiedad en la clase para que contenga este valor, siendo esto perfectamente válido. Pero existe otra vía para lograr este fin: crear un procedimiento Property Get, que realice el cálculo y devuelva el resultado. Veamos un ejemplo en el código fuente 274.

```
Public Property Get DiasAlta() As Long
DiasAlta = DateDiff("d", Me.Alta, Date)
End Property
```

Código fuente 274

La forma de utilizarlo en el programa, sería igual que para el resto de los procedimientos Property que sí se corresponden con una propiedad de la clase.

```
' declarar variables
Dim loCliente As Cliente
Dim llResultado As Long
' crear objeto Cliente
Set loCliente = New Cliente
............
............
' asignar la fecha 10-5-98 a la propiedad Alta
loCliente.Alta = #5/10/98#
' tomamos el valor de la propiedad DiasAlta,
' que en realidad lo que hace es ejecutar el
' procedimiento DiasAlta,
' suponiendo que la fecha actual sea 15-5-98
' el valor de llResultado será 5
llResultado = loCliente.DiasAlta
```

Código fuente 275

- **Empleo de nombres de propiedad más naturales.** Ya hemos comentado los beneficios de utilizar unas normas de notación para las variables, aspecto al que las propiedades de una clase no pueden permanecer ajenas.

  En la clase Cliente del ejemplo, hemos declarado una propiedad mnCodigo pública, mientras que el resto se han declarado privadas. Cuando en la aplicación utilicemos un objeto de esta clase para asignarle valores, lo haremos según el código fuente 276.

```
' declarar una variable y asignarle un objeto
Dim loCliente As Cliente
Set loCliente = New Cliente
' asignar valores a algunas propiedades del objeto
loCliente.mnCodigo = 286
.............
.............
loCliente.Ciudad = "Venecia"
.............
.............
```

Código fuente 276

Como la propiedad mnCodigo se ha declarado pública, al utilizarla en un objeto hemos de emplear, como es lógico, el mismo nombre para acceder a su contenido, lo que resulta un tanto incómodo al tener que recordar dentro del nombre, las partículas que indican el tipo de dato que contiene y el ámbito.

Para la propiedad mcCiudad, la situación es muy diferente. Al haberse declarado privada y disponiendo de procedimientos Property para acceder a su valor, emplearemos el nombre de tales procedimientos, en lugar usar directamente el nombre de la propiedad.

Ambos modos de trabajo son válidos, pero resulta evidente que el uso de los procedimientos Property para Ciudad, proporcionan un modo mucho más natural de interactuar con las propiedades de un objeto, constituyendo el mejor medio de comunicación entre el

programador y la información del objeto. Este es otro de los argumentos que podemos apuntar a favor de los procedimientos Property, como ventaja frente a las propiedades públicas.

Una de las normas de la OOP es proporcionar modelos más naturales de trabajo. Cuando anotamos en un papel los datos de un cliente ponemos: "Codigo: 286, Ciudad: Venecia" y no "mnCodigo: 286, mcCiudad: Venecia". Del mismo modo, cuando accedamos a las propiedades de un objeto en VB, los nombres de esas propiedades, deberían estar lo más próximas al lenguaje habitual que sea posible.

Establecer una propiedad como predeterminada

Cuando usamos un control Label, es posible asignar un valor a su propiedad Caption en la forma mostrada a continuación.

```
lblValor = "coche"
```

El hecho de no indicar la propiedad Caption en la anterior línea, se debe a que esta es la propiedad predeterminada del control/objeto, por lo que no será necesario ponerla para asignar/recuperar el valor que contiene.

De igual forma, si analizamos cual va a ser la propiedad o método que más se va a utilizar en una clase, podemos establecer esa propiedad o método como predeterminado, de manera que no sea necesario escribirlo en el código para usarlo. Sólo es posible definir una propiedad o método como predeterminado para cada clase. En la clase Cliente tomamos la propiedad Nombre como predeterminada.

Para ello, nos situaremos en el código del método o en el procedimiento Property Get de la propiedad a convertir en predeterminada y seleccionaremos la opción *Herramientas+Atributos del procedimiento* del menú de VB, que nos mostrará la ventana de la figura 361.

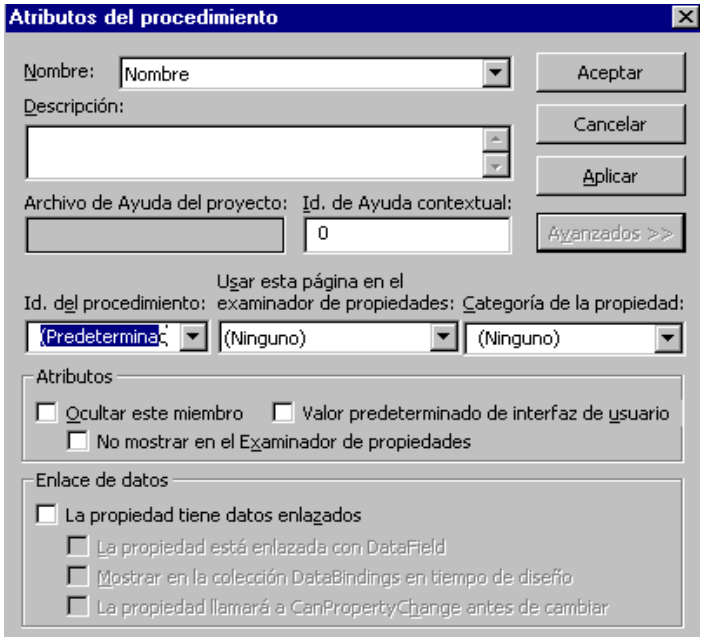


Figura 361. Establecer un elemento de la clase como predeterminado.

El siguiente paso consiste en seleccionar el valor Predeterminado en el ListBox *Id.del procedimiento*. Pulsaremos *Aceptar* para guardar los cambios realizados, con lo que a partir de ahora, para acceder a los valores de la propiedad Nombre de la clase Cliente podemos utilizar o no el nombre de la propiedad según se muestra en el código fuente 277.

```
Dim loCliente As Cliente
Set loCliente = New Cliente
...........
...........
' accedemos a la propiedad Nombre
' de la siguiente forma
loCliente = "José Ramirez"
...........
...........
' o bien de esta otra
loCliente.Nombre = "José Ramirez"
```

Código fuente 277

Un inconveniente de las propiedades predeterminadas es que hemos de recordar en todo momento cual es, para poder asignarle el valor adecuado. En la clase Cliente por ejemplo, hemos de tener la precaución de no intentar asignar a la propiedad Nombre el valor del código.

Crear los métodos de la clase

Los métodos de una clase, definen la forma en que un objeto creado a partir de una clase se comportará y las acciones que llevará a cabo.

La forma de crear los métodos en VB es a través del uso de procedimientos Sub o Function, dependiendo de si el método a codificar deberá devolver o no un valor. La única diferencia existente es en el ámbito conceptual, a un procedimiento creado dentro de un módulo de clase se le denomina método, mientras que los creados en módulos estándar son simplemente procedimientos.

En lo que respecta al ámbito de los métodos, se aplicarán iguales normas que para las propiedades. Si queremos que un método sea accesible desde el exterior de la clase, se declarará público, si sólo es necesario en el ámbito interno, se declarará privado. Declarando un método privado, conseguimos que el código de ese método no sea ejecutado por procesos externos a la clase, que pudieran afectar negativamente al objeto al que pertenece.

La clase Cliente dispone del método NombreMay(), cuyo cometido es pasar a mayúsculas la cadena contenida en la propiedad Nombre del objeto.

```
Public Sub NombreMay()
Me.Nombre = UCase(Me.Nombre)
End Sub
```

Código fuente 278

El código fuente 279, muestra como se emplea el método dentro del programa.

```
' declarar las variables
Dim loCliente As Cliente
Dim lcNuevoNombre As String
' crear un objeto Cliente
Set loCliente = New Cliente
..............
..............
loCliente.Nombre = "Luis Palermo"
..............
..............
' llamar al método NombreMay del objeto
loCliente.NombreMay
' el valor de la propiedad nombre será "LUIS PALERMO"
lcNuevoNombre = loCliente.Nombre
```

Código fuente 279

¿Cuándo crear un procedimiento Property o un método?

Cuando queramos definir las características o atributos de un objeto, emplearemos un procedimiento Property. Por ejemplo, para manejar la *ciudad* de un objeto cliente crearemos el Property Ciudad.

Cuando se trate de definir el comportamiento o acciones de un objeto, utilizaremos métodos. Por ejemplo, la acción de *mostrar los datos* de un cliente puede perfectamente englobarse en un método.

Pese a lo anteriormente dicho, existen ocasiones en que es difícil determinar cuando emplear uno u otro sistema, por lo cual el empleo de un procedimiento Property o un método debe ser algo flexible y no basarse en unas normas rígidas. Para averiguar por ejemplo, los días que lleva dado de alta un cliente, podemos crear un procedimiento Property, ya que se puede considerar perfectamente como una propiedad del objeto. Sin embargo, también es factible definirlo como un método, ya que para obtener el número de días es necesario realizar una operación. Como podemos comprobar en este caso, el empleo de uno de los tipos de procedimiento queda en manos del criterio del programador.

Como ayuda ante estas situaciones, debemos analizar lo que necesitamos definir para el objeto, si tiene que ver con un nombre (Ciudad) crearemos procedimientos Property, si es con un verbo (MostrarDatos) crearemos un método.

Crear objetos de una clase

Una vez que hemos terminado de desarrollar una clase, para poder usarla hemos de crear *instancias* de la misma, ya que la clase no puede ser utilizada directamente.

Para comprender este concepto un poco mejor, volvamos al ejemplo de la clase Impreso, visto al comienzo del tema. La plancha de la imprenta, que representa a la clase Impreso, no puede ser utilizada directamente para rellenar los datos de un albarán o factura. Debemos utilizar la plancha (clase) en la imprenta, para crear (instanciar) una factura (objeto) de papel, que será lo que rellenemos y enviemos al cliente.

Por instanciar entendemos pues, la operación de crear un objeto a partir de la definición de una clase. Dichos objetos, serán los que utilicemos en el código del programa.

El código fuente 280, muestra un procedimiento Main() con los pasos básicos para crear y utilizar un objeto de la clase Cliente.

```
Public Sub Main()
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' instanciar un objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores a las propiedades del objeto
loCliente.mnCodigo = 286
loCliente.Nombre = "Juan Palermo"
loCliente.Direccion = "Escalinata 20"
loCliente.Ciudad = "Salamanca"
' llamar a los métodos del objeto
loCliente.MostrarDatos
' liberar los recursos utilizados por el objeto
Set loCliente = Nothing
End Sub
```

Código fuente 280

Tipificación de variables

Por este concepto entendemos la asignación de un tipo de dato a una variable en el momento en que es declarada, con dicha característica conseguiremos que el compilador genere un ejecutable más veloz, ya que las referencias a los tipos de las variables serán resueltas en tiempo de compilación y no de ejecución.

Si aplicamos la anterior descripción a variables que van a contener objetos, la vía más óptima de hacerlo, es indicando el nombre de la clase de la cual vamos a instanciar un objeto. Esta característica se denomina *early binding* o *enlace temprano*. Mediante este tipo de enlace, las referencias a métodos y propiedades para la variable del objeto se establecen en tiempo de compilación, de forma que durante la ejecución no es necesario buscarlas a través del código, mejorando el rendimiento de la aplicación.

Una declaración early binding podría tener la siguiente forma.

```
Dim loCliente As Cliente
```

Una variante a este tipo de declaración, es el uso de la palabra clave New al declarar el objeto. Con este tipo de declaración no será necesario instanciar posteriormente el objeto con la instrucción Set, se creará una instancia automática de él la primera vez que se utilice una de sus propiedades o métodos, con lo que ahorramos el paso de instanciarlo.

```
' declarar el objeto con New
Dim loCliente As New Cliente
' en esta línea, al mismo tiempo que
' se crea una instancia del objeto
' de forma automática, se asigna un
' valor a una de sus propiedades
loCliente.mnCodigo = 286
```

Código fuente 281

Este tipo de declaración puede influir negativamente en el rendimiento del programa, debido a que cada vez que se utilice el objeto para llamar a un método o propiedad, VB ha de estar comprobando si ya se ha creado o no, una instancia de la clase para la variable.

Otra forma de declaración se basa en el uso de la palabra *Object* para indicar el tipo de variable.

```
Dim loCliente As Object
```

Esta tipificación es conocida como late binding o enlace tardío, en ella, al no conocerse el tipo concreto de objeto con el que vamos a trabajar, las referencias que la variable del objeto realiza a los métodos y propiedades, no se pueden establecer en tiempo de compilación. Por dicho motivo, es durante la ejecución del programa, cuando se han de estar buscando las referencias según se van produciendo, de forma que el rendimiento de la aplicación se ve afectado negativamente.

Visto lo anterior, podemos deducir que empleando variables de objeto, y siempre que conozcamos la clase de objeto que contendrá la variable, debemos declararla indicando el nombre de dicha clase

Existe una tercera forma de declaración, no indicando el tipo de dato para la variable, que es la que arrastra un mayor grado de penalización en el rendimiento, por lo que debe evitarse siempre que sea posible.

```
Dim loCliente
```

La línea de código anterior no proporciona al compilador información alguna sobre el tipo de dato que debe contener la variable, creándose Variant por defecto, con lo cual, el compilador ha de encargarse de hacer las conversiones de tipos, y en el caso de objetos, el inconveniente añadido del enlace tardío.

Instanciar un objeto

La manera más habitual de crear una instancia de una clase y asignarla a una variable es mediante la instrucción *Set*, empleando la palabra clave *New*.

```
Set loCliente = New Cliente
```

La línea de código anterior crea una nueva instancia de la clase Cliente y la asigna a la variable loCliente.

El motivo de emplear Set para asignar un objeto a una variable, a diferencia de las variables comunes que no necesitan utilizar dicha instrucción, reside en el hecho de que una variable de objeto contiene una referencia a un objeto y una variable común contiene un valor, de ahí el empleo de una sintaxis especial.

Si la declaración de la variable incluye la palabra clave New, se instanciará un objeto la primera vez que utilicemos la variable sin necesidad de usar Set (consultar el apartado anterior, en lo referente a la declaración con New). Veamos un ejemplo en el código fuente 282.

```
' declarar el objeto con New
' pasando directamente a utilizarlo
Dim loCliente As New Cliente
loCliente.mnCodigo = 286
```

Código fuente 282

Uso de propiedades y métodos

Una vez que el objeto se ha creado y asignado a una variable, esta se utilizará para manipular las propiedades del objeto y hacer llamadas a sus métodos, empleando el llamado en los lenguajes OOP *operador de envío* o *Send*, que en VB se representa con el signo de un punto ".".

Figura 362. Formato de uso para una variable de objeto.

```
' manipular una propiedad del objeto
loCliente.Ciudad = "Salamanca"
' llamar a un método del objeto
loCliente.MostrarDatos
```

Código fuente 283.

Un consejo para aumentar el rendimiento de la aplicación, es emplear en lo posible sentencias cortas que hagan uso de un único objeto, en lugar de sentencias que impliquen múltiples referencias a varios objetos. Para explicar esta cuestión un poco mejor, supongamos que tenemos un objeto empresa, que contiene como propiedad otro objeto de tipo proveedor, y este último a su vez contiene una propiedad teléfono. Si queremos acceder al teléfono del objeto proveedor en una sola línea de código, podemos hacer lo que aparece en el código fuente 284.

```
Dim cTelf As String
Do While <expr>
  ........
  ........
  cTelf = oEmpresa.Proveedor.Telefono
  ........
  ........
Loop
```

Código fuente 284

Con este modo de trabajo, cada vez que recuperamos la propiedad teléfono, se efectúan dos búsquedas en objetos, una para el objeto empresa y otra para el objeto proveedor. Si el bucle Do While debe realizar muchas iteraciones, esto supone una pérdida de rendimiento. Sin embargo, si empleamos la técnica mostrada en el código fuente 285, conseguiremos evitar una de las operaciones con los objetos, recuperando el valor únicamente de donde nos interesa.

```
Dim cTelf As String
Dim oProv As Proveedor
oProv = oEmpresa.Proveedor
Do While <expr>
  ........
  ........
  cTelf = oProv.Telefono
  ........
  ........
Loop
```

Código fuente 285

Por una razón similar, además de un ahorro y claridad en el código, cuando debamos utilizar un mismo objeto en varias líneas de código seguidas, es conveniente incluirlas en una estructura With...End With, como vemos en el código fuente 286, para una mayor información sobre esta estructura de control, consulte el lector el tema *El Lenguaje*, apartado *Estructuras de control*.

```
With loCliente
  .Codigo = 286
  .Nombre = "Juan Palermo"
  .Direccion = "Escalinata 20"
  .Ciudad = "Salamanca"
  .MostrarDatos
End With
```

Código fuente 286

Eliminar el objeto

Un objeto utiliza los recursos del equipo, por lo que cuando no siga siendo necesario su uso, es necesario eliminar la referencia que mantiene la variable hacia el objeto, asignando a la variable la palabra clave *Nothing*.

```
' liberar los recursos utilizados por el objeto
Set loCliente = Nothing
```

Código Fuente 287

Si hemos definido en un procedimiento, una variable local para un objeto, la referencia al objeto contenida en la variable será eliminada al finalizar el procedimiento.

La palabra clave Me

La palabra Me, se utiliza para establecer una referencia a un objeto, desde dentro del código del propio objeto. Este concepto, que puede parecer un rompecabezas, podemos entenderlo mejor con el siguiente ejemplo. En el módulo Codigo.BAS de la aplicación OOMe, incluida con los ejemplos, hemos creado el procedimiento EliminarCiudad(), que recibe como parámetro, un objeto de la clase Cliente y asigna una cadena vacía a su propiedad Ciudad.

```
Public Sub EliminarCiudad(ByVal voCliente As Cliente)
voCliente.Ciudad = ""
End Sub
```

Código fuente 288.

En el código fuente 289, en el procedimiento Main() de la aplicación, creamos dos objetos de la clase Cliente, les asignamos valores a su propiedad Ciudad, y mediante el anterior procedimiento, borramos el contenido de la propiedad de uno de ellos.

```
Public Sub Main()
' declarar variables de objeto y
' asignarles valor
Dim loUnCliente As Cliente
Dim loOtroCliente As Cliente
Set loUnCliente = New Cliente
loUnCliente.Ciudad = "Roma"
Set loOtroCliente = New Cliente
loOtroCliente.Ciudad = "Estocolmo"
' eliminar valor de la propiedad Ciudad
' en uno de los objetos
EliminarCiudad loUnCliente
' mostrar contenido de la propiedad Ciudad de cada objeto
MsgBox "Ciudad del objeto loUnCliente: " & loUnCliente.Ciudad
MsgBox "Ciudad del objeto loOtroCliente: " & loOtroCliente.Ciudad
End Sub
```

Código fuente 289

Hasta ahora todo normal, pero llegados a este punto, vamos a crear un método en la clase Cliente, llamado QuitaCiudad(), que hará una llamada al procedimiento EliminarCiudad() para borrar el valor de la propiedad Ciudad del objeto. El procedimiento Main(), quedaría ahora como se muestra en el código fuente 290.

```
Public Sub Main()
Dim loUnCliente As Cliente
Dim loOtroCliente As Cliente
Set loUnCliente = New Cliente
loUnCliente.Ciudad = "Roma"
Set loOtroCliente = New Cliente
loOtroCliente.Ciudad = "Estocolmo"
' primera forma de eliminar la propiedad
' deshabilitada para que no interfiera con
' la siguiente
'EliminarCiudad loUnCliente
' siguiente forma de eliminar la propiedad
loUnCliente.QuitaCiudad
MsgBox "Ciudad del objeto loUnCliente: " & loUnCliente.Ciudad
MsgBox "Ciudad del objeto loOtroCliente: " & loOtroCliente.Ciudad
End Sub
```

Código fuente 290

Cuando el objeto loUnCliente llama al método QuitaCiudad, se ejecuta el código fuente 291.

```
Public Sub QuitaCiudad()
EliminarCiudad Me
End Sub
```

Código fuente 291

En este método se llama al procedimiento EliminarCiudad, que recibe como ya vimos, un objeto de la clase Cliente, por lo tanto, ¿cómo podemos referirnos al objeto desde dentro de su propio código?. La respuesta es *Me*, que es pasado como parámetro a EliminarCiudad.

Usando Me, el objeto sabrá que se está haciendo uso de una de sus propiedades o métodos desde dentro de su propio código, o que se está pasando la referencia del objeto como parámetro a una rutina, este es el caso en el método QuitaCiudad().

Adicionalmente, el lector habrá comprobado que se están utilizando dos variables con objetos de la misma clase, pero sólo se borra la propiedad de uno de los objetos. Ya que los objetos de una misma clase comparten el código de sus métodos, ¿cómo sabe el código de la clase, el objeto sobre el que está actuando?.

La respuesta a la anterior pregunta es la siguiente: cuando un objeto ejecuta uno de sus métodos, internamente se establece una referencia a las propiedades que contiene ese objeto, de forma que siempre se sabe sobre que objeto actuar. Para el programador, dicha referencia está representada en Me.

Para terminar con las ventajas del uso de Me, diremos que proporciona una mayor legibilidad al código de una clase, ya que nos permite comprobar más fácilmente cuales son las propiedades y métodos pertenecientes a la clase, como se muestra en el código fuente 292.

```
Public Sub ComprobarValores()
Dim Nombre As String
' podemos utilizar una variable con la misma
' denominación que una propiedad, ya que al usar Me
' se sabe cuando se quiere usar la variable
' y cuando la propiedad
Nombre = "valor cualquiera"    ' aquí usamos la variable
MsgBox "Contenido de la propiedad: " & Me.Nombre  ' aquí usamos la propiedad
' ahora llamamos a un método de esta clase,
' lo identificamos rápidamente por el uso de Me
Me.CalculaTotales
' a continuación llamamos a un procedimiento
' de la aplicación. Una de las formas de saberlo
' es teniendo la costumbre de utilizar siempre Me
' en las llamadas a los métodos de la propia clase,
' aunque hemos de tener en cuenta que si ponemos el
' nombre de un método sin Me, también será admitido
VerFechaActual
End Sub
```

Código fuente 292

La palabra Me en VB equivale a Self o This en otros lenguajes orientados a objeto.

Emplear valores constantes en una clase

La definición de constantes en una clase, plantea el problema de que no pueden utilizarse desde el exterior de la misma. Si definimos dentro de un módulo de clase varias constantes.

```
Const Volum_Bajo = 1
Const Volum_Medio = 2
Const Volum_Alto = 3
```

Código fuente 293

Y tenemos un método o propiedad que recibe un valor relacionado con las constantes. Dentro del propio código de la clase podemos emplear las constantes, como vemos en el código fuente 294.

```
Me.Volumen = Volum_Medio
```

Código fuente 294

Pero si intentamos utilizarlas desde un objeto fuera de la clase, se producirá un error (código fuente 295).

```
loCliente.Volumen = Volum_Medio
' esta línea provocará un error
```

Código fuente 295

Para solventar este problema, VB proporciona los nuevos tipos enumerados, que definidos como públicos dentro de un módulo de clase, serán visibles desde el exterior de la misma. Consulte el lector el tema *El Lenguaje*, apartado *El tipo Enumerado*, para una descripción detallada de los mismos.

La aplicación OOEnum proporcionada como ejemplo, muestra la ya conocida clase Cliente, con las adaptaciones necesarias para emplear tipos enumerados. Dichos cambios se basan en la creación de una nueva propiedad que muestra el volumen de compras que realiza el cliente.

En primer lugar, declaramos en el módulo de la clase Cliente la nueva propiedad, que guardará el volumen de compras, código fuente 296.

```
Private mvntVolumen As Variant
```

Código fuente 296

En el código fuente 297, definimos en la zona de declaraciones de la clase, el tipo enumerado VolumCompras, que contendrá los valores permitidos para la propiedad mvntVolumen.

```
' crear valores constantes para la clase
' a través de un tipo enumerado
```

```
Public Enum VolumCompras
   Bajo = 1
   Medio = 2
   Alto = 3
End Enum
```

Código fuente 297

En el código fuente 298, creamos los procedimientos Property Volumen, de acceso/asignación para la propiedad mvntVolumen.

```
Public Property Let Volumen(ByVal vvntVolumen As Variant)
mvntVolumen = vvntVolumen
End Property
' ------------------------------------
Public Property Get Volumen() As Variant
Select Case mvntVolumen
Case VolumCompras.Alto
   Volumen = "Alto"
Case VolumCompras.Medio
   Volumen = "Medio"
Case VolumCompras.Bajo
   Volumen = "Bajo"
End Select
End Property
```

Código fuente 298

Y por fin, para utilizar esta nueva cualidad en un objeto cliente, lo haremos según se muestra en el procedimiento Main() de esta aplicación.

```
Public Sub Main()
' declarar variable de objeto
Dim loCliente As Cliente
Set loCliente = New Cliente
............
............
loCliente.Volumen = VolumCompras.Alto  ' utilizamos el tipo enumerado
............
............
loCliente.MostrarDatos
End Sub
```

Código fuente 299

Como habrá observado el lector, la causa de declarar como Variant la propiedad mvntVolumen se debe a que al asignarle valores, utilizamos el tipo enumerado VolumCompras, pero al recuperar el valor de dicha propiedad con el método MostrarDatos(), necesitamos disponer de una cadena de caracteres que muestre al usuario la descripción del volumen de compras en lugar de un número que proporciona menos información.

También podíamos haber declarado mvntVolumen como Long para emplear el mismo valor que VolumCompras, realizando la conversión a cadena en el método MostrarDatos(). Todo depende de los requerimientos de la aplicación y el enfoque del programador.

Eventos predefinidos para una clase

Un evento es un mensaje que un objeto envía cuando se produce una situación determinada sobre dicho objeto, y que podrá ser respondido o no por el código de la aplicación.

Una clase dispone de dos eventos predefinidos, que se producen al crear y destruir el objeto respectivamente.

El primero de estos eventos es *Initialize*, que se genera al instanciar un objeto de la clase. Si el lector conoce algún otro lenguaje OOP, este evento es similar al Constructor de la clase. La diferencia entre este y los constructores de otros lenguajes, es que Initialize no puede recibir parámetros.

El otro evento es *Terminate*, que se produce al eliminar la referencia que el objeto mantiene con la aplicación. Para eliminar dicha referencia podemos asignar Nothing al objeto, o se puede producir al finalizar el procedimiento en donde está contenido el objeto, si se ha declarado como local. Este evento es similar al Destructor de la clase en otros lenguajes OOP.

Para comprobar el trabajo realizado en estos eventos, disponemos de la aplicación de ejemplo EventCls, que contiene la clase Cliente, utilizada en el apartado anterior, y a la que se le ha incorporado el código fuente 300 en los métodos de evento predefinidos.

```
Private Sub Class_Initialize()
MsgBox "Se crea un objeto de la clase Cliente"
End Sub
' ---------------------------------------
Private Sub Class_Terminate()
MsgBox "Se elimina el objeto de la clase Cliente"
End Sub
' ---------------------------------------
Public Sub Main()
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' instanciar un objeto de la clase Cliente
' aquí se produce el evento Initialize
Set loCliente = New Cliente
............
............
' eliminar el objeto
' aquí se produce el evento Terminate
Set loCliente = Nothing
```

Código fuente 300

Colecciones

Un objeto Collection contiene un grupo de elementos de uno o varios tipos de datos, que pueden ordenarse basándose en una clave.

En la aplicación de ejemplo *Coleccio*, vamos a crear un objeto Collection y varios objetos de la clase Cliente que guardaremos en la colección, de forma que ilustremos la creación y manejo de este nuevo tipo de objeto.

Cuando agreguemos objetos a una colección, hemos de tener presente que no se añade el objeto por completo, sino una referencia al mismo, por lo que aunque durante el manejo de la colección en este ejemplo, tengamos la sensación de que la colección alberga objetos, lo que realmente tiene y manejamos en el programa, son referencias a dichos objetos.

Aunque no es necesario, si es una buena costumbre que cada objeto colección que creemos para una aplicación, se defina como privado en un módulo de clase independiente, lo cual nos ayudará a mantener encapsulado el código de la colección. La forma de acceder al contenido de la colección será creando métodos públicos para que el código externo a la colección pueda acceder a ella.

Puesto que vamos a utilizar objetos de la ya conocida clase Cliente, no repetiremos de nuevo los pasos de creación de dicha clase, pasando directamente a los pasos de creación de la clase que albergará la colección Clientes.

Insertar un módulo de clase

En primer lugar, agregaremos al proyecto un nuevo módulo de clase, en el que incluiremos todo el código para manejar la colección.

Los objetos creados a partir de esta clase, serán colecciones destinadas a contener objetos de la clase Cliente, por este motivo llamaremos a la clase por el plural de su contenido: Clientes.

En la zona de declaraciones del módulo, definiremos una variable privada que será la que contenga el objeto Collection, y que sólo será accesible desde el interior del módulo.

```
Private mcllClientes As Collection
```

Eventos de creación/destrucción del objeto Collection

Cuando creemos un objeto de la clase Clientes, necesitaremos también que dentro del objeto exista una instancia creada de la colección mcllClientes, pero si dicha colección es una variable privada y no es posible acceder a ella ni siquiera a través de procedimientos Property, para conseguir la mayor encapsulación, ¿cómo podemos instanciarla?. La solución está en los eventos predefinidos de la clase, que utilizaremos para crear y destruir un objeto Collection y asignarlo a mcllClientes, según se indica en el código fuente 301.

```
Private Sub Class_Initialize()
Set mcllClientes = New Collection
End Sub
' ------------------------------
Private Sub Class_Terminate()
Set mcllClientes = Nothing
End Sub
```

Código fuente 301

Procedimientos Property para la colección

Los objetos Collection disponen de la propiedad *Count*, que devuelve un número Long con la cantidad de elementos contenidos en la colección. Esta propiedad es de sólo lectura.

Para conseguir recuperar el valor de Count del objeto mcllClientes, sólo hemos de escribir un procedimiento público Property Get que acceda a dicha propiedad como indica el código fuente 302.

```
Public Property Get Count() As Long
Count = mcllClientes.Count
End Property
```

Código fuente 302

Métodos para la colección

Al igual que para las propiedades, hemos de escribir los oportunos métodos que accedan a la colección mcllClientes del módulo de clase, y que ejecuten los métodos de dicha colección. Dichos métodos, creados por nosotros, mantendrán los mismos nombres que los del objeto colección, pero podríamos darles un nombre cualquiera.

- **Add.** Este método, añade un nuevo elemento a la colección, su sintaxis es la siguiente:

  ```
  Add xElemento, cClave, nAntes, nDespues
  ```

  - xElemento. Un valor u objeto que se va a agregar a la colección.
  - cClave. Cadena de caracteres con un valor para que el elemento a agregar se ordene dentro de la colección.
  - nAntes. Expresión numérica que indica que xElemento se situará antes del valor nAntes.
  - nDespues. Expresión numérica que indica que xElemento se situará después del valor nDespues.

  El valor de los parámetros nAntes y nDespues debe ser un número o expresión numérica comprendida entre 1 y el valor de la propiedad Count de la colección. Es posible utilizar uno u otro, pero nunca ambos a la vez. También es importante saber, que el uso de estos parámetros puede tener un efecto perjudicial en el rendimiento de la aplicación.

  El método a codificar para Add, estará parcialmente modificado, ya que en este caso en concreto, no vamos a hacer uso de los parámetros nAntes ni nDespues, y el valor que actuará como clave para la colección, será el propio código del cliente, que será convertido a cadena dentro del método.

  ```
  Public Sub Add(ByVal voCliente As Cliente)
  mcllClientes.Add voCliente, CStr(voCliente.Codigo)
  End Sub
  ```

  Código fuente 303

- **Remove.** Este método, elimina un elemento de la colección, su sintaxis es la siguiente:

  ```
  Remove xIndice
  ```

- xIndice. Expresión que identifica la posición de la colección a eliminar. Si es un valor numérico, debe estar entre 1 y el contenido de la propiedad Count. Si es una cadena, el valor será el asignado como clave al añadir el elemento a la colección.

```
Public Sub Remove(ByVal vnElemento As Variant)
mcllClientes.Remove vnElemento
End Sub
```

Código fuente 304

La razón de declarar el parámetro de este método como Variant, es que podemos borrar el elemento de la colección por su posición dentro de la colección, o por la cadena pasada como clave al agregar el elemento a la colección.

- **Item.** Devuelve un elemento de una colección, pasando como parámetro su posición o clave, su sintaxis es la siguiente:

  `Item(xIndice)`

  - xIndice. Expresión que identifica el elemento de la colección a recuperar. Si es un valor numérico, debe estar entre 1 y el contenido de la propiedad Count. Si es una cadena, el valor será el asignado como clave al añadir el elemento a la colección.

  Este método se define como Function al tener que devolver un valor. Como la colección contiene elementos de tipo objeto, hacemos uso de Set para asignar el objeto de la colección al nombre del método.

```
Public Function Item(ByVal vnElemento As Variant) As Cliente
Set Item = mcllClientes.Item(vnElemento)
End Function
```

Código fuente 305

Al igual que en los objetos Collection habituales, este método se ha definido como predeterminado para que no sea necesario indicarlo en el código al recuperar uno de sus elementos.

```
' recuperar uno de los elementos de la colección
' utilizando un método predeterminado del objeto
Set loCliMostrar = lcllClientes(2)
```

Código fuente 306

Como hemos podido comprobar, no es necesario emplear Item() para tomar uno de los objetos cliente de la colección.

Incorporar nuevas funcionalidades a la colección

Hasta aquí, hemos implementado los métodos y propiedades de que dispone un objeto Collection en nuestro módulo de clase. Pero otra de las ventajas de incluir el manejo de la colección en un módulo de clase, aparte de la encapsulación, es que podemos añadir nuevas características a la colección.

Los objetos Collection no disponen de un método para eliminar su contenido dejándolo vacío. Vamos a crear pues, un método llamado EliminarTodo() que haga esta labor. De esta forma, nuestras colecciones de objetos cliente, además de hacer todo lo que hace un objeto colección normal, podrán eliminar su contenido mediante la llamada a uno de sus métodos.

Lo único que tenemos que hacer es insertar una línea de código que instancie un nuevo objeto Collection y lo asigne a la variable mcllClientes del módulo de clase que contiene el código de la colección, como se muestra en el código fuente 307.

```
Public Sub EliminarTodo()
Set mcllClientes = New Collection
End Sub
```

Código fuente 307

Recorrer la colección mediante *For Each...Next*

Para recorrer los elementos de un array o colección, podemos emplear la estructura de control For...Next, como se indica en el código fuente 308.

```
For lnIndice = 1 To lcllClientes.Count
MsgBox "Nombre del cliente: " & lcllClientes.Item(lnIndice).Nombre
Next
```

Código fuente 308

Sin embargo, la mejor forma de recorrer los elementos de un array o colección, es el empleo de la estructura de control For Each...Next, ya que está especialmente diseñada para esta tarea.

En la clase Clientes, hemos creado el método VerNombres(), que haciendo uso de For Each...Next, muestra la propiedad Nombre de cada objeto cliente contenido en la colección.

```
Public Sub VerNombres()
Dim loCliente As Cliente
For Each loCliente In mcllClientes
    MsgBox "Nombre del cliente: " & loCliente.Nombre
Next
End Sub
```

Código fuente 309

Como podemos comprobar, desde la propia clase podemos utilizar perfectamente For Each...Next. El problema aparece cuando queremos utilizar esta estructura de control desde el exterior de la clase Clientes, sobre un objeto creado a partir de la clase Clientes. Como ya sabemos, la colección mcllClientes es privada, y por lo tanto inaccesible desde fuera del módulo de clase.

Para solventar este inconveniente existe un pequeño truco que consiste en exponer la enumeración del objeto colección mcllClientes en el módulo de clase. De esta forma se podrá aplicar For Each...Next a la colección mcllClientes desde cualquier punto de la aplicación.

Para conseguir este propósito, debemos crear un método llamado NewEnum() en el módulo de la clase Clientes. Este método será el que proporcione acceso al enumerador de la colección.

```
Public Function NewEnum() As IUnknown
Set NewEnum = mcllClientes.[_NewEnum]
End Function
```

Código fuente 310

Lo que hace este método es acceder al método oculto _NewEnum que contiene cualquier objeto Collection, en nuestro caso la colección es mcllClientes. La llamada a _NewEnum devuelve el enumerador de la colección, más concretamente el interface IUnknown de un objeto enumerador. Mediante este interface, accederemos al contenido de la colección.

La especial forma de notación empleada al obtener el interface, se debe a que el método _NewEnum es oculto (los métodos que comienzan por un guión bajo son ocultos por convención). El uso de corchetes en el nombre del método es necesario ya que el guión bajo no es un carácter válido para emplear en el nombre de un método.

A continuación abrimos la ventana *Atributos del procedimiento* para NewEnum() (figura 363). En el campo *Id. del procedimiento* tecleamos "-4", marcamos el CheckBox *Ocultar este miembro* del marco *Atributos* y aceptamos los valores de la ventana.

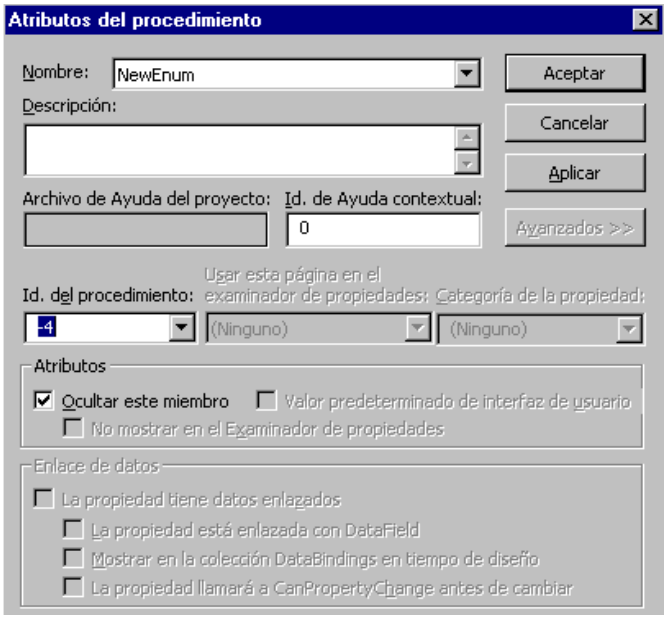


Figura 363. Atributos para el método NewEnum().

Por último sólo queda comprobar que en la ventana *Referencias* del proyecto se ha marcado la referencia *OLE Automation* (StdOle2.tlb), para evitar errores del compilador (figura 364).

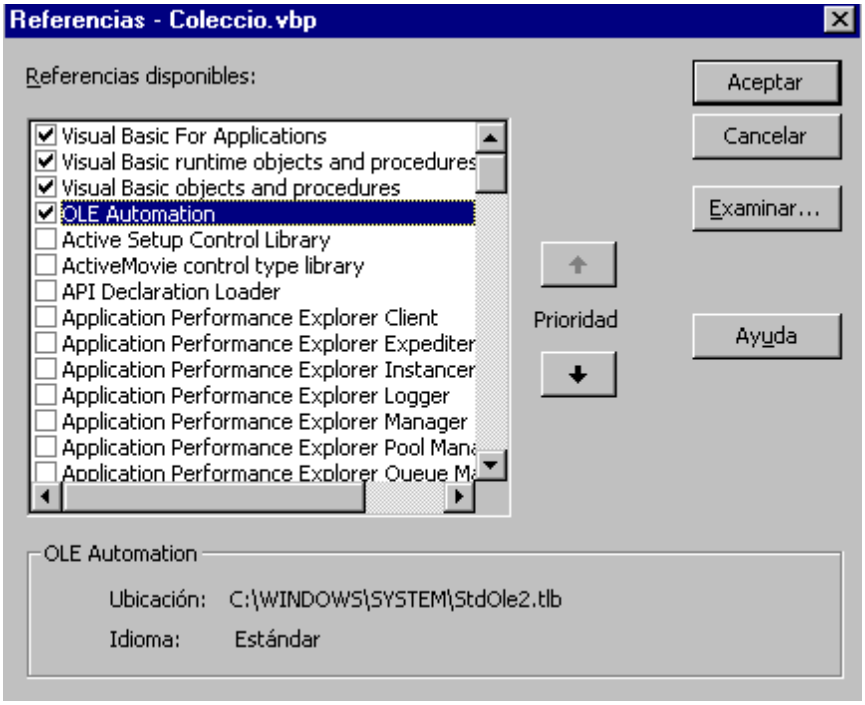


Figura 364. Referencia en el proyecto a OLE Automation.

A partir de ahora, podemos utilizar la estructura For Each...Next para acceder de una forma sencilla a todos los elementos contenidos en un objeto de nuestra clase Clientes.

En el procedimiento Main() de esta aplicación podemos ver como crear y utilizar un objeto de la clase Clientes.

```
Public Sub Main()
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' declarar variable de tipo Cliente
' para utilizar en bucle For Each...Next
Dim loClienteVer As Cliente
' declarar una variable para recuperar
' un elemento de la colección y
' visualizarlo
Dim loCliMostrar As Cliente
' declarar una variable para contener
' un objeto de la clase Clientes,
' que internamente contiene una colección
Dim lcllClientes As Clientes
' instanciar el objeto Clientes
Set lcllClientes = New Clientes
' instanciar tres objetos de la clase Cliente
' y agregarlos a la colección Clientes
' ------------------------------------------
' instanciar primer objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores a las propiedades del objeto
```

```
loCliente.Codigo = 751
loCliente.Nombre = "Enrique Rubio"
loCliente.Direccion = "Real 5"
loCliente.Ciudad = "Soria"
' agregar el objeto a la colección
lcllClientes.Add loCliente
' instanciar segundo objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores a las propiedades del objeto
loCliente.Codigo = 228
loCliente.Nombre = "Ana Rojo"
loCliente.Direccion = "Peral 32"
loCliente.Ciudad = "Burgos"
' agregar el objeto a la colección
lcllClientes.Add loCliente
' instanciar tercer objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores a las propiedades del objeto
loCliente.Codigo = 576
loCliente.Nombre = "Ramiro Blas"
loCliente.Direccion = "Balanza 6"
loCliente.Ciudad = "Teruel"
' agregar el objeto a la colección
lcllClientes.Add loCliente
' el siguiente método utiliza dentro
' de su código, en el módulo de clase
' un bucle For Each...Next
' para recorrer el contenido de la
' colección
lcllClientes.VerNombres
' mostrar la propiedad Nombre de cada
' uno de los objetos contenidos en
' la colección mediante un bucle For Each...Next
For Each loClienteVer In lcllClientes
   MsgBox "Ciudad del cliente: " & loClienteVer.Ciudad
Next
' eliminar uno de los elementos de la colección
' empleando la clave del elemento
lcllClientes.Remove "228"
' recuperar uno de los elementos de la colección
' utilizando un método predeterminado del objeto
Set loCliMostrar = lcllClientes(2)
loCliMostrar.MostrarDatos
' eliminar todos los elementos
' contenidos en el objeto
lcllClientes.EliminarTodo
' mostrar como ha quedado el contenido
' del objeto
MsgBox "Número de clientes en la colección " & _
   lcllClientes.Count
End Sub
```

Código fuente 311

El objeto Dictionary

Los objetos Dictionary, al igual que los Collection, se encargan de almacenar un conjunto de elementos. Sin embargo, los Dictionary están diseñados de forma que su uso sea más sencillo y flexible para el programador, evitando algunos de los errores producidos en ejecución por las colecciones.

En un diccionario, cada elemento se compone de una clave y el valor que vamos a asignar a ese elemento. De esta manera, se facilita el acceso a los valores del objeto mediante su clave.

El proyecto de ejemplo Dicciona, contiene algunas muestras del modo de manejo de este tipo de objetos. En primer lugar para poder utilizar objetos Dictionary en una aplicación, debemos establecer la correspondiente referencia a la librería Microsoft Scripting Runtime, como se muestra en la figura 365.

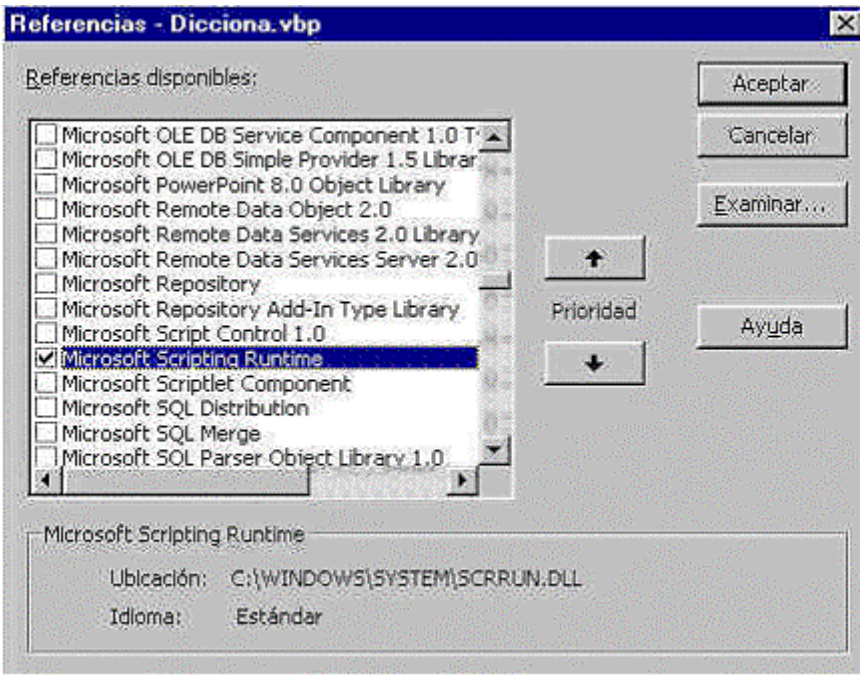


Figura 365. Establecimiento de la referencia a la librería que contiene los objetos Dictionary.

Establecida la referencia, en este programa se utiliza un formulario con el nombre frmDiccionario, que contiene el código para manejar el objeto Dictionary dicDatos, declarado a nivel de módulo para que sea visible desde cualquier punto de esta ventana.

Una vez abierto este formulario desde el procedimiento Main() del proyecto, a través de las opciones de su menú, realizaremos diferentes operaciones con el objeto diccionario.

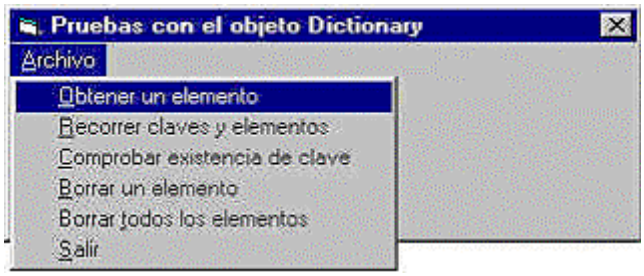


Figura 366. Formulario frmDiccionario para las pruebas con el objeto Dictionary.

Agregar elementos al diccionario

En el evento Load() de frmDiccionario, se emplea el método Add() con el que añadiremos un grupo de elementos al objeto para nuestras pruebas.

- Sintaxis:

  ```
  Add( xClave, xElemento)
  ```

  - xClave. El dato por el que se buscará en el objeto un determinado elemento, puede ser de cualquier tipo de datos excepto array.

  - xElemento. El valor que se asigna al elemento del diccionario, puede ser de cualquier tipo de dato.

El código de Load() quedará de la siguiente forma.

```
Private Sub Form_Load()
' crear objeto diccionario
Set mdicDatos = New Dictionary
' asignar datos al diccionario
mdicDatos.Add "quique", "ENRIQUE JUAN"
mdicDatos.Add "feli", "MIGUEL FELIPE"
mdicDatos.Add "pepe", "JOSE ALBERTO"
mdicDatos.Add "nando", "FERNANDO GABRIEL"
mdicDatos.Add "paco", "FRANCISCO JOSE"
End Sub
```

Código fuente 312

Obtener un elementos del diccionario

Mediante la propiedad Item de este objeto, pasando como parámetro el nombre de una clave, recuperaremos el valor de un elemento del diccionario. Esta operación se efectúa seleccionando en el menú del formulario, la opción Archivo+Obtener un elemento, cuyo fuente vemos en el código fuente 313.

```
Private Sub mnuArObtener_Click()
Dim lcNombreClave As String
' introducir un nombre para buscar en el objeto
lcNombreClave = InputBox("Introducir nombre de la clave", _
   "Buscar elemento")
' mostrar el elemento del diccionario
MsgBox "Elemento del diccionario: " & mdicDatos.Item(lcNombreClave)
End Sub
```

Código fuente 313

Recorrer la información del diccionario

Los métodos Keys() e Items() de un objeto Dictionary, devuelven un array conteniendo las claves y los elementos del objeto. El código fuente 314 corresponde a la opción Archivo+Recorrer claves y elementos del formulario de ejemplo, e ilustra el modo de recuperar esta información y visualizarla.

```
Private Sub mnuArRecorrer_Click()
Dim lvntClaves As Variant
Dim lvntElementos As Variant
Dim lnInd As Integer
' obtener las claves
lvntClaves = mdicDatos.Keys
' mostrar las claves
For lnInd = 0 To UBound(lvntClaves)
   MsgBox "Clave " & lnInd & " del diccionario: " & lvntClaves(lnInd)
Next
' obtener los elementos
lvntElementos = mdicDatos.Items
' mostrar los elementos
For lnInd = 0 To UBound(lvntElementos)
   MsgBox "Elemento " & lnInd & " del diccionario: " & lvntElementos(lnInd)
Next
End Sub
```

Código fuente 314

Comprobar existencia de elementos

El método Exists() comprueba la existencia de un elemento en un Dictionary. Recibe como parámetro el valor de la clave a buscar, retornando un valor lógico verdadero si la clave existe, y falso en el caso de que no haya una clave con ese valor. La opción de menú Archivo+Comprobar existencia de clave, en el ejemplo, muestra como utilizar este método.

```
Private Sub mnuArComprobar_Click()
Dim lcNombreClave As String
' introducir el nombre de una clave
' a buscar en el diccionario
lcNombreClave = InputBox("Introducir el nombre de la clave", _
   "Buscar elemento por clave")
If mdicDatos.Exists(lcNombreClave) Then
   MsgBox "Se ha encontrado la clave: " & lcNombreClave & vbCrLf & _
   "Elemento: " & mdicDatos(lcNombreClave)
Else
   MsgBox "No se ha encontrado la clave: " & lcNombreClave
End If
End Sub
```

Código fuente 315

Borrar un elemento del diccionario

Emplearemos el método Remove() para eliminar un elemento de este objeto; dicho método recibe como parámetro el nombre de la clave a borrar. El código de la opción Archivo+Borrar un elemento del menú del ejemplo nos muestra este aspecto del objeto.

```
Private Sub mnuArBorrar_Click()
Dim lcNombreClave As String
' introducir el nombre de una clave
' a borrar del diccionario
lcNombreClave = InputBox("Introducir el nombre de la clave", _
   "Borrar elemento")
' si existe la clave, borrar el elemento
If mdicDatos.Exists(lcNombreClave) Then
   mdicDatos.Remove lcNombreClave
   MsgBox "Clave borrada"
Else
   MsgBox "No existe la clave"
End If
End Sub
```

Código fuente 316

Borrar todos los elementos del diccionario

Esta operación es la más sencilla de todas, ya que sólo hemos de llamar al método RemoveAll() para borrar todos los elementos del diccionario. Corresponde a la opción Archivo+Borrar todos los elementos, del menú de ejemplo.

```
Private Sub mnuArBorrarTodos_Click()
' eliminar todos los elementos del objeto
mdicDatos.RemoveAll
End Sub
```

Código fuente 317

Los ejemplos que acabamos de ver, utilizan un diccionario para manipular cadenas de caracteres, pero es perfectamente posible emplear este tipo de objetos para almacenar otros objetos. De igual forma que vimos anteriormente con las colecciones, podemos utilizar un Dictionary para guardar objetos de la clase Cliente, que el lector ya conoce de anteriores ejemplos.

Eventos personalizados

Un evento, como se indicaba en una descripción anterior, es un mensaje que envía un objeto, para que pueda ser respondido por otro objeto dentro de la aplicación. Al objeto que desencadena el evento se le conoce como *emisor de eventos*, mientras que el objeto que responde al evento enviado es conocido como *receptor de eventos*.

Supongamos que hemos creado un formulario y hemos insertado en él un objeto CommandButton llamado cmdBoton. Cada vez que pulsemos con el ratón sobre dicho objeto, este creará un evento Click que será enviado para ser respondido por cualquier objeto de la aplicación que se haya definido

como receptor de eventos. En nuestro caso, el objeto receptor será el formulario que contiene el CommandButton. El formulario al recibir el evento buscará en su módulo de código el procedimiento definido para el evento Click del botón, si existe código, lo ejecutará, entendiéndose esto como la respuesta al evento lanzada por el botón.

El motivo de incluir el código de respuesta a la pulsación del botón en el formulario, se debe en primer lugar a que no es posible acceder al código de la clase CommandButton, y en segundo, a que de esta forma se organiza mejor el código.

Cuando un objeto emisor envía un evento, no lo dirige hacia un objeto en concreto, sólo se encarga de lanzar ese evento. Si existen en la aplicación uno o más objetos receptores de ese evento y además se han configurado para responder a ese tipo de eventos, ya se encargarán los receptores de capturar dicho evento y tratarlo.

Aparte de los eventos predeterminados vistos anteriormente, VB permite al programador la creación de eventos propios para poder controlar diversas situaciones a lo largo del programa.

Para ilustrar este aspecto, utilizaremos la aplicación de ejemplo *EventPers*, que se basa en las ya conocidas clases Cliente y Clientes, añadiendo la nueva clase TomarDatos. La clase Clientes actuará como emisor de eventos, y la clase TomarDatos como receptor. Sigamos los pasos necesarios para crear un evento personalizado en esta aplicación.

Primeramente se debe declarar en el módulo de la clase Clientes o emisora de eventos (zona de declaraciones), el evento a generar mediante la instrucción Event(), el cual será público por defecto, utilizando la siguiente sintaxis:

```
Event NombreEvento([Parámetros])
```

El evento que deseamos crear, y al que llamaremos *Agregado*, debe producirse cada vez que se incorpore un nuevo elemento (objeto Cliente) a un objeto-colección Clientes. Para lo cual, se declarará el evento en el código de la clase Clientes como indica el código fuente 318.

```
' declaración de eventos
Public Event Agregado()
```

Código fuente 318

Después pasaremos al método Add() de la clase Clientes, aquí desencadenaremos el evento con la instrucción *RaiseEvent*, Que tiene la siguiente sintaxis:

```
RaiseEvent NombreEvento[(Parámetros)]
Public Sub Add(ByVal voCliente As Cliente)
mcllClientes.Add voCliente, CStr(voCliente.Codigo)
' cada vez que se añada un elemento
' a la colección, desencadenar el
' evento Agregado
RaiseEvent Agregado
End Sub
```

Terminado de crear el código para el emisor de eventos, pasaremos a continuación a introducir el código correspondiente a la clase TomarDatos, que es el receptor de eventos, para que pueda responder a los eventos generados por la clase Clientes.

La primera tarea a realizar en TomarDatos, es declarar como propiedad un objeto de la clase Clientes como muestra el código fuente 319.

```
Private WithEvents moClientes As Clientes
```

Código fuente 319

El empleo de la palabra clave WithEvents al declarar la propiedad moClientes, significa que se habilita al código de la clase TomarDatos para recibir y responder a los eventos generados por el objeto moClientes. De esta forma, el objeto moClientes pasa a formar parte de los objetos disponibles en el módulo de código de la clase TomarDatos, creándose de forma automática un procedimiento de evento vacío para el evento Agregado de la clase Clientes.

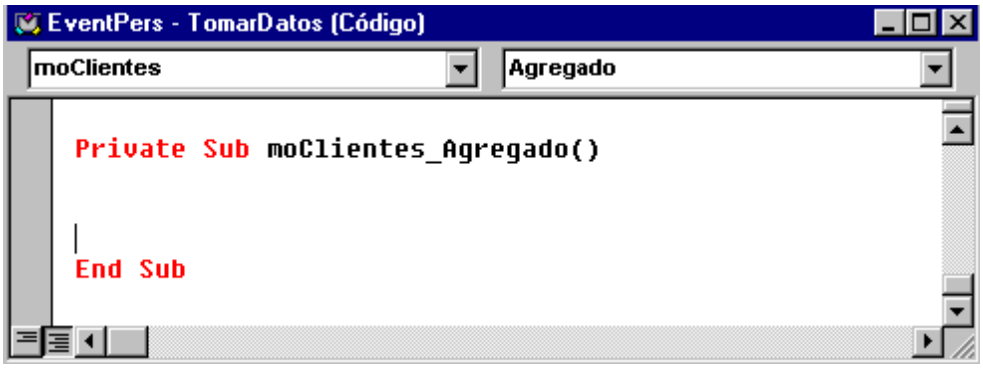


Figura 367. Procedimiento de respuesta a evento en la clase TomarDatos.

Ahora es necesario escribir el código para esta clase. Comenzaremos creando el método AddCliente(), que recibirá un conjunto de valores con los que crear un objeto cliente, para después incorporarlo a la colección moClientes, como se muestra en el código 320.

```
Public Sub AddCliente(ByVal vnCodigo As Integer, _
   ByVal vcNombre As String, ByVal vcDireccion As String, _
   ByVal vcCiudad As String)

' agregar los valores pasados como parámetro a
' un objeto Cliente
' el objeto Cliente se agregará a su vez
' al objeto colección Clientes de esta clase
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' instanciar el objeto Clientes
Set loCliente = New Cliente
' asignar valores a las propiedades del objeto
loCliente.Codigo = vnCodigo
loCliente.Nombre = vcNombre
loCliente.Direccion = vcDireccion
loCliente.Ciudad = vcCiudad
' añadir el cliente a la colección moClientes
moClientes.Add loCliente
End Sub
```

Código fuente 320

Cuando en la última línea de este método se agrega el objeto loCliente a la colección moClientes, este último objeto desencadena el evento Agregado al ejecutar su método Add(). Esto causa que la aplicación busque en la propiedad moClientes si existe código en el procedimiento de evento Agregado(), en caso afirmativo, se ejecutará dicho código como respuesta al evento generado por el objeto moClientes.

```
Private Sub moClientes_Agregado()
Dim loCliente As Cliente
Set loCliente = moClientes(moClientes.Count)
MsgBox "Se acaba de incorporar el cliente " & loCliente & vbCrLf & _
    "Total de clientes agregados: " & moClientes.Count, , _
    "Información"
End Sub
```

Código fuente 321

En concreto, lo que hace este procedimiento es informar al usuario de que se acaba de incorporar un nuevo elemento al objeto moClientes, y el número de elementos que contiene.

Un objeto sólo podrá declararse para notificar eventos (WithEvents) dentro de un módulo de clase, nunca en un módulo de código estándar.

Interfaces

Un *interfaz* es un conjunto de propiedades y métodos de un objeto relacionados con una característica determinada de ese objeto.

Por ejemplo, para la clase Cliente, las propiedades que guardan el código, nombre, dirección, ciudad, y el método que las visualiza forman el llamado *interfaz Predeterminado*, que está formado por los elementos básicos de la clase. Si posteriormente necesitamos disponer de información contable sobre un objeto cliente, podemos crear un *interfaz Contable* con una propiedad para la cuenta contable del cliente, otra para el saldo, y un método para mostrar el saldo al usuario. Más adelante, puede darse el caso de que necesitemos mandar circulares a los clientes, para lo cual crearíamos el *interfaz Circular* con una propiedad para el texto de la circular y métodos para visualizar dicho texto y enviar la circular al cliente, y así sucesivamente. Vemos el esquema de los interfaces para la clase cliente en la figura 368.

Si como vimos al principio de este tema, la programación orientada a objetos nos ayuda a enfocar los problemas de una forma más natural, los interfaces nos ayudan a organizar en el ámbito declarativo los objetos creados para resolver los problemas.

¿Qué significa eso de organizar en el ámbito declarativo?, Pues se trata de que un interfaz contiene la declaración de las propiedades (procedimientos Property) y métodos que van a componer ese interfaz, pero no la implementación del código para los mismos. Aunque pueda resultar extraño, mediante esta forma de hacer las cosas conseguiremos que las clases desarrolladas dispongan de características tan importantes como polimorfismo y un cierto nivel de herencia, teniendo siempre presente que la herencia, tal y como se entiende para los lenguajes orientados a objeto, no se proporciona por el momento en la versión actual de VB.

Todos los objetos que hemos utilizado hasta el momento, no disponían de una organización interna a base de interfaces, ya que todas las propiedades y métodos eran incluidos en el interfaz

predeterminado. El uso de interfaces nos permite dividir en partes lógicas las diferentes funcionalidades de un objeto, aportándonos diversas ventajas.

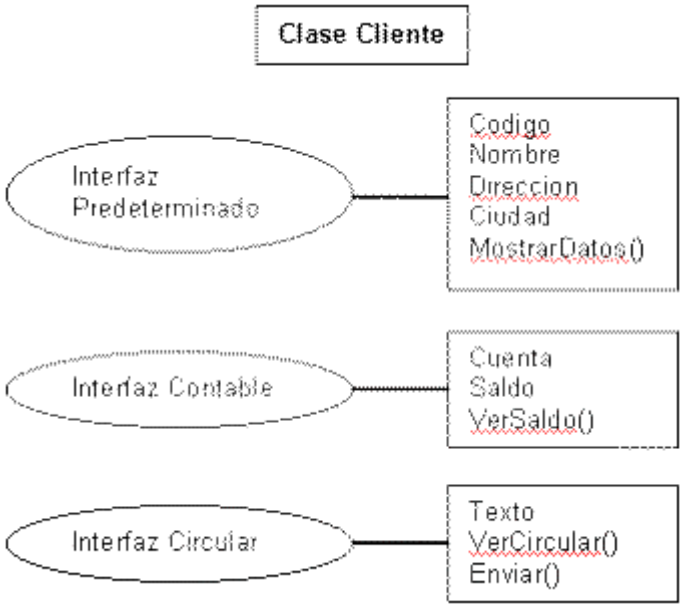


Figura 368. Esquema de los interfaces de la clase Cliente.

Creación de un interfaz

Para ilustrar este punto, nos ayudaremos de la aplicación de ejemplo Interface, que contiene la clase Cliente, ya conocida por el lector. Para esta clase desarrollaremos un interfaz que controle sus aspectos contables.

El primer paso a dar es el diseño del interfaz. Este es un punto muy importante, ya que una vez creado el interfaz, si intentamos cambiarlo, las clases que hagan uso de él se verán afectadas negativamente, al no disponer de las modificaciones introducidas al interfaz. Por este motivo es muy importante que el diseño se haga cuidadosamente, ya que una vez terminado no debemos realizar modificaciones.

Nuestro interfaz contable deberá disponer de una propiedad para contener la cuenta contable del cliente, otra para el saldo pendiente, y un método para mostrar al usuario dicho saldo.

El siguiente paso consiste en añadir un nuevo módulo de clase al proyecto, que será el que contenga la definición del interfaz. El nombre que daremos a este módulo y por consiguiente al interfaz es *IContable*. La "I" al comienzo del nombre nos sirve para saber que este módulo contiene la definición de un interfaz.

A continuación insertaremos la declaración de los procedimientos (Property y métodos) del interfaz sin incluir código en el interior de los mismos. Como hemos dicho, un interfaz sólo tiene efecto en el ámbito declarativo, por lo que cualquier declaración de variables de módulo y código serán ignorados. El contenido del interfaz Contable quedaría como se muestra en el código fuente 322.

```
Public Property Let Cuenta(ByVal vcCuenta As String)
End Property
Public Property Get Cuenta() As String
End Property
```

```
' ------------------------------------------------
Public Property Let Saldo(ByVal vlSaldo As Long)
End Property
Public Property Get Saldo() As Long
End Property
' ------------------------------------------------
Public Sub VerSaldo()
End Sub
```

Código fuente 322

Implementar el interfaz

Una vez creado el interfaz, hemos de implementarlo (incluirlo) en el módulo de clase que vaya a usarlo, en este caso la clase Cliente.

En primer lugar, hemos de declarar el interfaz en la sección de declaraciones de la clase Cliente mediante la instrucción *Implements*, como vemos en el código fuente 323.

```
' implementar el interface IContable
Implements Icontable
```

Código fuente 323

Una vez declarado el interfaz en la clase, el objeto IContable aparecerá en la lista de objetos de la ventana de código del módulo Cliente, y todas las propiedades y métodos declaradas en el interfaz, aparecerán en la lista de procedimientos, como vemos en las figuras 369 y 370.

Como este interfaz va a manejar dos propiedades (procedimientos Property) que son la cuenta contable y el saldo, hemos de declarar las variables a nivel de módulo que contendrán el valor de tales propiedades (código fuente 324).

```
' variable para propiedades del interface IContable
Private mcCuenta As String
Private mlSaldo As Long
```

código fuente 324

El siguiente paso es incluir el código en los procedimientos del interfaz para dotarle de funcionalidad. Es obligatorio incluir dicho código, o de lo contrario aparecerán errores al intentar ejecutar la aplicación.

Si es totalmente necesario dejar sin codificar alguno de los procedimientos del interfaz, hemos de generar un error en el código de este interfaz con el número &H80004001, para el cual podemos declarar la constante E_NOTIMPL. De esta forma, el compilador sabrá que no se ha codificado ese procedimiento.

Por defecto, al implementar el interfaz en una clase, se da el nombre *RHS* al parámetro utilizado por los procedimientos Property Let, que significa *Right Hand Side* y contiene el valor a asignar a la propiedad, como vemos en el código fuente 325. Una vez implementado el interfaz, el programador puede cambiar este nombre por otro más significativo dentro del contexto del código.

```
Private Property Let IContable_Cuenta(ByVal RHS As String)
End Property
```

Código fuente 325

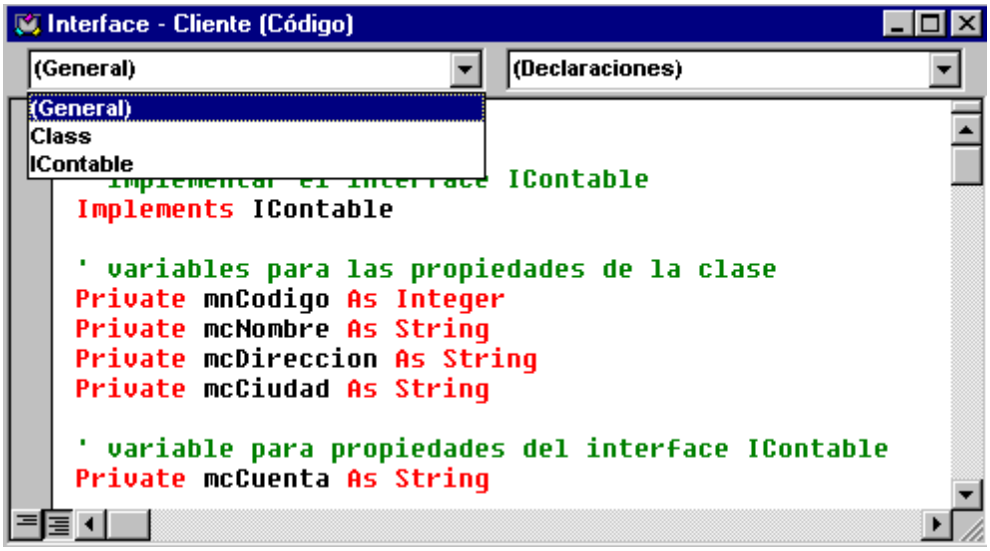


Figura 369. El interfaz IContable aparece en la lista de objetos de la clase que lo utiliza.

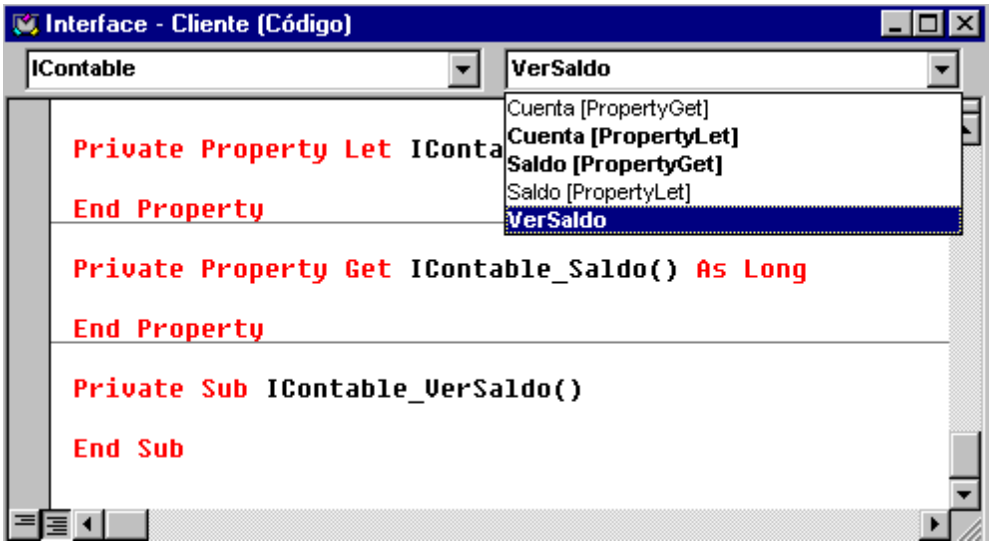


Figura 370. Procedimientos del interfaz incluidos en la clase que lo utiliza.

Una vez incluido el código para el interfaz, los procedimientos del mismo quedarán como muestra el código fuente 326.

```
Private Property Get IContable_Cuenta() As String
IContable_Cuenta = mcCuenta
End Property
Private Property Let IContable_Cuenta(ByVal vcCuenta As String)
mcCuenta = vcCuenta
```

```
End Property

' ------------------------------------------------

Private Property Let IContable_Saldo(ByVal vlSaldo As Long)
mlSaldo = vlSaldo
End Property
Private Property Get IContable_Saldo() As Long
IContable_Saldo = mlSaldo
End Property

' ------------------------------------------------

Private Sub IContable_VerSaldo()
MsgBox "El cliente " & Me.Nombre & vbCrLf & _
   "tiene un saldo pendiente de " & _
   Format(IContable_Saldo, "#,###,###")
End Sub
```

Código fuente 326

Utilizar un interfaz implementado en una clase

Si el lector revisa los procedimientos de la clase Cliente correspondientes al interfaz IContable, notará que estos se han declarado privados. En ese caso, ¿cómo podemos acceder al interfaz implementado en la clase Cliente?. La respuesta es mediante una variable del tipo del interfaz. Sí, ha entendido bien el lector, del tipo del interfaz y no de la clase que contiene el interfaz, este punto es muy importante, veámoslo con detenimiento.

Utilizaremos la variable de interfaz para llamar a uno de sus procedimientos, al ser públicos, accederemos a ellos sin problema. ¿Qué ocurre?, que los procedimientos del interfaz no contienen código, pero como el interfaz está implementado en una clase, y a estos procedimientos aunque son privados, el interfaz si puede acceder, hace una llamada al que tiene el mismo nombre dentro de la clase y lo ejecuta.

Ya que esta forma de comunicación con el código parece un tanto compleja, veamos en la figura 371 un esquema el proceso seguido.

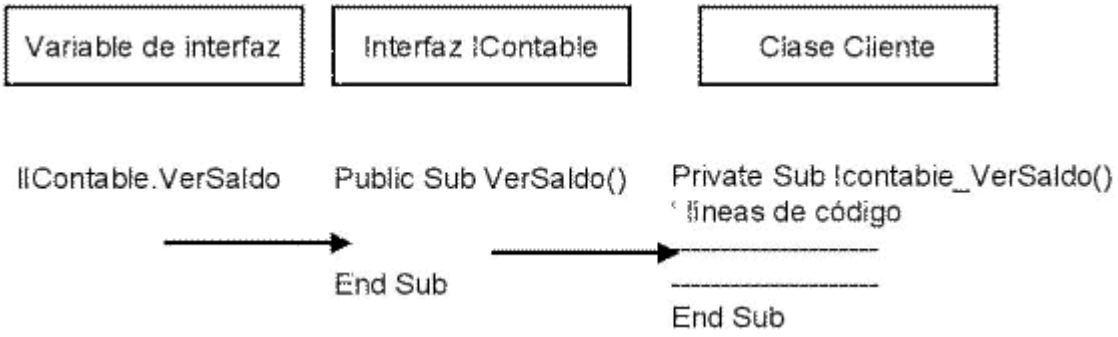


Figura 371. Sistema de llamadas en un interfaz.

Ahora veamos en el código fuente 327 los dos mecanismos disponibles para utilizar un interfaz. En uno de ellos creamos una variable de objeto Cliente y una de interfaz IContable. Inicializamos ambas y manejamos el objeto Cliente como habitualmente lo hemos hecho. Pero para poder utilizar las características del interfaz, asignamos el objeto Cliente a la variable de interfaz. Sólo a partir de ese momento es cuando podremos utilizar el interfaz.

```
Public Sub Main()
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' declarar una variable para contener
' un interfaz IContable
Dim lIContable As IContable
' instanciar un objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores al interfaz predeterminado
loCliente.Codigo = 286
loCliente.Nombre = "Juan Palermo"
loCliente.Direccion = "Escalinata 20"
loCliente.Ciudad = "Salamanca"
' llamar a los métodos del objeto
loCliente.MostrarDatos
' asignar el objeto al interfaz IContable
' para poder manipular el contenido
' del interfaz
Set lIContable = loCliente
lIContable.Cuenta = "4300920"
lIContable.Saldo = 140555
lIContable.VerSaldo
' liberar los recursos utilizados por el objeto
Set loCliente = Nothing
End Sub
```

Código fuente 327

Observe el lector que para el interfaz IContable no es necesario crear ninguna instancia, ya que el interfaz en sí mismo no es un objeto, se le asigna un objeto para manejar el contenido del interfaz. Bien puede ser un objeto ya creado contenido en una variable, o crear un objeto al mismo tiempo que lo asignamos al interfaz, en el caso de que no vayamos a usar el interfaz predeterminado del objeto.

```
Set lIContable = New Cliente
```

Aunque un interfaz se define en un módulo de clase, no es posible instanciar objetos de dicho módulo, ya que un interfaz no es un objeto. El módulo de clase sólo sirve de contenedor para la declaración del interfaz, pero no se ejecuta código en su interior como ya hemos visto.

La otra forma de utilizar un interfaz, pasa por crear un procedimiento que tenga como parámetro un valor del interfaz a emplear. Al hacer la llamada el procedimiento, se pasaría como parámetro el objeto del que hemos de utilizar el interfaz, y una vez dentro del procedimiento, al estar declarado el parámetro como del tipo de interfaz, sólo se utilizaría este. De esta manera, el código que llama al procedimiento no necesitaría declarar variables de interfaz ni asignar el objeto al interfaz, todo se haría en el procedimiento que maneja el interfaz.

Observemos en el código fuente 328, como quedaría nuestro ejemplo de la clase Cliente utilizando esta técnica.

```
Public Sub Main()
' declarar una variable para contener
' un objeto de la clase Cliente
Dim loCliente As Cliente
' instanciar un objeto de la clase Cliente
Set loCliente = New Cliente
' asignar valores al interfaz predeterminado
loCliente.Codigo = 286
```

```
loCliente.Nombre = "Juan Palermo"
loCliente.Direccion = "Escalinata 20"
loCliente.Ciudad = "Salamanca"
' llamar a los métodos del objeto
loCliente.MostrarDatos
' llamar al procedimiento de manejo
' del interfaz pasándole el objeto
' cliente como parámetro
'DatosContab loCliente
' liberar los recursos utilizados por el objeto
Set loCliente = Nothing
End Sub
' ------------------------------------------------
Public Sub DatosContab(ByVal oDatos As IContable)
' este procedimiento al recibir el objeto
' en el parámetro, sólo utilizará el interfaz
' IContable, al estar declarado el parámetro
' de este tipo
oDatos.Cuenta = "4300234"
oDatos.Saldo = 55874
oDatos.VerSaldo
End Sub
```

Código fuente 328

Esta técnica puede resultar un poco complicada de manejar, si el objeto que estamos utilizando tiene más de dos interfaces implementados o existe más de un objeto que implemente el interfaz, por lo que se recomienda la primera forma de uso de interfaces.

Después de todo lo visto sobre el uso de interfaces el lector puede pensar: "Todo esto está muy bien, pero ¿qué gano yo utilizando interfaces, aparte de añadir complejidad a mi código?"

Entre las ventajas que proporciona el uso de interfaces, podemos apuntar las siguientes:

- Facilitar el diseño de los objetos. En clases que contengan muchas propiedades y métodos, el uso de interfaces ayuda a organizar internamente dichas clases en partes o temas lógicos.

- Reutilizar las características de un interfaz. Si disponemos de un interfaz para una clase y creamos una nueva clase que necesita las características del interfaz de la primera, podemos reutilizar el interfaz implementándolo en la nueva clase.

- Incorporar nuevas características a la clase sin afectar al código existente. Si después de usar una clase en varias aplicaciones, debemos incorporar a la misma alguna característica que en un principio no estaba prevista, creando un nuevo interfaz e implementándolo en la clase evitará tener que modificar el código ya escrito y posibles errores causados por la modificación de alguna rutina existente en la clase.

Polimorfismo mediante Interfaces

El polimorfismo según vimos al comienzo de este tema, se basa en la posibilidad de que dos objetos tengan un comportamiento distinto al aplicar sobre ellos el mismo método.

Si tenemos un objeto vaso de cristal y otro objeto pelota de goma, está claro que el comportamiento de ambos será muy diferente cuando apliquemos sobre ellos el método Tirar. Mientras que el vaso se romperá, la pelota rebotará en el suelo sin romperse. Mismo método, diferente comportamiento.

Para mostrar como funciona el polimorfismo mediante interfaces, emplearemos la aplicación de ejemplo *Polimorf*. En ella disponemos de la clase Cliente utilizada anteriormente y de la nueva clase Proveedor, la cual dispone de un interfaz predeterminado y en la que queremos incorporar la funcionalidad contable de que dispone la clase Cliente.

La forma de implementar el interfaz IContable en la clase Proveedor es la misma que la vista en el anterior apartado para la clase Cliente. La diferencia reside al mostrar el saldo con el método VerSaldo(). Mientras que en la clase Cliente, informamos al usuario que el saldo es el que nos debe el cliente, en la clase Proveedor indicaremos que el saldo es el que tenemos que pagar al proveedor. Iguales métodos, distinto comportamiento.

De esta forma el método VerSaldo() del interfaz IContable implementado en la clase Proveedor, quedaría como se muestra en el código fuente 329.

```
Private Sub IContable_VerSaldo()
MsgBox "Al proveedor " & Me.Nombre & vbCrLf & _
    "debemos pagarle un importe de " & _
    Format(IContable_Saldo, "#,###,###")
End Sub
```

Código fuente 329

En el procedimiento Main() de esta aplicación, se declaran dos variables para objetos Cliente y Proveedor respectivamente. Después de utilizar su interfaz predeterminado, se asigna a la variable de interfaz IContable para que el objeto pueda manipular los datos de este interfaz.

```
Public Sub Main()
' declarar variables para objetos
' Cliente y Proveedor
Dim loCliente As Cliente
Dim loProveedor As Proveedor
' declarar variable para
' interfaz IContable
Dim lIContable As IContable
' instanciar un objeto de la clase Cliente
Set loCliente = New Cliente
' manejar el interfaz predeterminado
loCliente.Codigo = 286
loCliente.Nombre = "Juan Palermo"
loCliente.Direccion = "Escalinata 20"
loCliente.Ciudad = "Salamanca"
loCliente.MostrarDatos
' asignar el objeto cliente al
' interfaz IContable para que el
' objeto manipule este interfaz
Set lIContable = loCliente
lIContable.Cuenta = "4300920"
lIContable.Saldo = 140555
lIContable.VerSaldo
' ----------------------------------------
' instanciar un objeto de la clase Proveedor
Set loProveedor = New Proveedor
' manejar el interfaz predeterminado
loProveedor.Codigo = 523
loProveedor.Nombre = "Transportes Veloz"
loProveedor.VerProveedor
' asignar el objeto proveedor al
' interfaz IContable para que el
```

```
' objeto manipule este interfaz
Set lIContable = loProveedor
lIContable.Cuenta = "4000153"
lIContable.Saldo = 60541
lIContable.VerSaldo
' eliminar los objetos
Set loCliente = Nothing
Set loProveedor = Nothing
End Sub
```

Código fuente 330

Aunque asignamos los objetos a la misma variable de interfaz, dicha variable sabe a que código de la clase dirigir la llamada a las propiedades y métodos, ya que internamente es el objeto asignado y no el interfaz, el que se utiliza.

Herencia mediante Interfaces

Como ya comentamos anteriormente, mediante la herencia podemos crear una nueva clase o clase hija a partir de una clase existente o clase padre. Por el mero hecho de hacer que la clase hija herede de la padre, esta clase hija contendrá todas las propiedades y métodos de la clase padre más los nuevos que desarrollemos en ella. Más concretamente, los objetos creados a partir de la clase hija, podrán acceder al código contenido en la clase padre.

Ya que actualmente VB no proporciona auténtica herencia, la forma de conseguir una pseudo-herencia, por decirlo de alguna manera, es mediante los siguientes pasos que se indican a continuación, y que serán explicados después con más detalle:

- Crear la clase padre en el caso de que no dispongamos ya de ella.
- Implementar la clase padre dentro del código de la clase hija.
- Declarar en la clase hija, una variable a nivel de módulo para contener un objeto de la clase padre.
- Instanciar dentro de la clase hija, un objeto de la clase padre.
- Delegar dentro de la clase hija, las funcionalidades correspondientes a la clase padre.

Para mostrar como conseguir un cierto grado de herencia en VB, utilizaremos la aplicación *Herencia* incluida como ejemplo.

En esta aplicación disponemos de la anteriormente utilizada clase Cliente, pero con algunas propiedades menos para no hacer este ejemplo demasiado pesado.

La situación que se nos plantea es la siguiente: necesitamos una clase que tenga toda la funcionalidad de que dispone la clase Cliente, pero aplicada a clientes a los que debemos cobrar unos gastos financieros. La clase Cliente actuaría como padre, haciéndose necesario la creación de una clase hija a la que llamaremos CliFinanciar, que posea las nuevas características.

Omitiremos el código fuente de la clase Cliente por ser de sobra conocido por el lector, pasando directamente a la creación de la clase hija CliFinanciar. Esta clase dispone de propiedades para el tipo de interés a aplicar, la base de cálculo del interés y los gastos financieros resultantes. Dispone también

de un método que es el que realiza el cálculo para obtener los gastos financieros. El código fuente 331 muestra todo este código.

```
' declarar propiedades de esta clase
Private mnTipo As Integer
Private mdBaseFinanc As Double
Private mdGastosFinanc As Double
' ------------------------------------------------
Public Sub CalcularFinanc()
Me.GastosFinanc = (Me.BaseFinanc * Me.Tipo) / 100
End Sub
' ------------------------------------------------
Public Sub MostrarCalculo()
MsgBox "Resultado del cálculo" & vbCrLf & vbCrLf & _
   "Importe a financiar: " & Format(Me.BaseFinanc, "#,###,###") & vbCrLf & _
   "Tipo aplicado: " & Me.Tipo & vbCrLf & _
   "Gastos resultantes: " & Format(Me.GastosFinanc, "#,###,###")

End Sub
' ------------------------------------------------
Public Property Let Tipo(ByVal vnTipo As Integer)
mnTipo = vnTipo
End Property
Public Property Get Tipo() As Integer
Tipo = mnTipo
End Property
' ------------------------------------------------
Public Property Let BaseFinanc(ByVal vdBaseFinanc As Double)
mdBaseFinanc = vdBaseFinanc
End Property
Public Property Get BaseFinanc() As Double
BaseFinanc = mdBaseFinanc
End Property
' ------------------------------------------------
Public Property Let GastosFinanc(ByVal vdGastosFinanc As Double)
mdGastosFinanc = vdGastosFinanc
End Property
Public Property Get GastosFinanc() As Double
GastosFinanc = mdGastosFinanc
End Property
```

Código fuente 331

El siguiente paso a dar es implementar la clase padre en la sección de declaraciones de la clase CliFinanciar. Esta operación como vimos al explicar la creación de interfaces, tomará las definiciones de los procedimientos de la clase Cliente y los depositará en la clase hija.

```
Implements Cliente
' ------------------------------------------------
Private Property Let Cliente_Codigo(ByVal RHS As Integer)
End Property
Private Property Get Cliente_Codigo() As Integer
End Property
' ------------------------------------------------
Private Sub Cliente_MostrarDatos()
End Sub
' ------------------------------------------------
Private Property Let Cliente_Nombre(ByVal RHS As String)
End Property
Private Property Get Cliente_Nombre() As String
End Property
```

Código fuente 332

Continuaremos declarando un objeto de la clase padre dentro del código de la clase hija, este objeto será utilizado sólo por el código de la clase hija, no pudiendo acceder a él desde el exterior. Se creará al instanciar un objeto de clase hija y se destruirá al eliminarlo.

```
' declarar objeto de clase padre
Private mCliente As Cliente
' ------------------------------
Private Sub Class_Initialize()
Set mCliente = New Cliente
End Sub
' ------------------------------
Private Sub Class_Terminate()
Set mCliente = Nothing
End Sub
```

Código fuente 334

Una vez que la clase padre está implementada dentro de la hija, en vez de repetir todo el código de la clase Cliente dentro de los procedimientos correspondientes de la clase CliFinanciar, lo que haremos será que desde dichos procedimientos se llame al método con el que guarda correspondencia en la clase padre. Esta operación se conoce como *delegación*, y en el código fuente 334 se muestran algunos de los procedimientos en la clase CliFinanciar que delegan en sus homónimos de la clase padre Cliente.

```
Private Property Let Cliente_Codigo(ByVal vnCodigo As Integer)
mCliente.Codigo = vnCodigo
End Property
' ---------------------------------------------
Private Property Get Cliente_Codigo() As Integer
Cliente_Codigo = mCliente.Codigo
End Property
' ---------------------------------------------
Private Sub Cliente_MostrarDatos()
mCliente.MostrarDatos
End Sub
```

Código fuente 334

Como se puede observar, se delega en el objeto cliente mCliente, que se encarga de llamar al procedimiento que tiene el mismo nombre en la clase padre.

Uno de los inconvenientes de esta forma de crear herencia, es que en el código del programa hemos de trabajar con variables declaradas tanto para la clase hija como la padre. En una herencia más real, sólo necesitaríamos declarar la variable de la clase hija, y a través de esta variable haríamos llamadas a los métodos contenidos en la clase padre y accederíamos a sus propiedades.

El procedimiento Main(), del código fuente 335, de esta aplicación, utiliza un objeto de la clase hija y accede a la clase padre mediante la técnica de interfaces.

```
Public Sub Main()
' declarar un objeto de la clase
```

```
' cliente que actúe como interfaz,
' es decir, de este objeto no se
' van a crear instancias
' ** clase padre **
Dim lICliente As Cliente
' declarar objeto CliFinanciar
' ** clase hija **
Dim loCliFinanciar As CliFinanciar
' instanciar objeto de la clase hija
Set loCliFinanciar = New CliFinanciar
' utilizar los métodos propios de la
' clase hija
loCliFinanciar.BaseFinanc = 170000
loCliFinanciar.Tipo = 6
loCliFinanciar.CalcularFinanc
loCliFinanciar.MostrarCalculo
' asignar el objeto a la variable
' de la clase padre que hará las
' veces de interfaz
Set lICliente = loCliFinanciar
' utilizar las propiedades y métodos
' de la clase padre
lICliente.Codigo = 443
lICliente.Nombre = "Emilio Manzano"
lICliente.MostrarDatos
Set loCliFinanciar = Nothing
End Sub
```

Código fuente 335

El lector habrá comprobado que aunque la variable lICliente se declara de tipo Cliente, no se crea ninguna instancia de ella, se le asigna un objeto ya creado de su clase hija y se accede al código de la clase Cliente vía interfaz, de ahí el haberle incluido la "I" en el nombre, para saber que trabajamos con el interfaz.

Herencia sin utilizar interfaces en la aplicación

La técnica de utilizar interfaces en la aplicación para acceder al código de la clase padre es la forma habitual de trabajo. Existe, sin embargo, un medio de hacer llamadas al código de la clase padre sin tener que utilizar un interfaz de dicha clase. Es necesario añadir código adicional en la clase hija, pero el acceso a la clase padre queda oculto simulando una herencia más real.

Cuando implementamos la clase padre dentro de la clase hija y delegamos en la clase padre las llamadas a sus procedimientos, podemos crear procedimientos Property en la clase hija con los nombres de las propiedades de la clase padre. En dichos procedimientos nos limitaremos a hacer una llamada al procedimiento de la clase padre que está implementado en la clase hija.

Como la anterior explicación es un tanto rebuscada y liosa, lo mejor es exponer esta técnica mediante un ejemplo. Para ello usaremos la aplicación de ejemplo *Herenci2*, que se basa en la aplicación *Herencia*, pero con algunos retoques en el código para conseguir una herencia mejorada.

Tomemos la propiedad Código de la clase Cliente. Una vez implementada en la clase CliFinanciar, y añadido el código necesario para delegar en la clase Cliente, sus procedimientos Property quedarían como se muestra en el código fuente 336.

```
Private Property Let Cliente_Codigo(ByVal vnCodigo As Integer)
```

```
mCliente.Codigo = vnCodigo
End Property
' ------------------------------------------------
Private Property Get Cliente_Codigo() As Integer
Cliente_Codigo = mCliente.Codigo
End Property
```

Código fuente 336

Hasta ahora nada nuevo, debemos acceder a la clase Cliente mediante un interfaz de dicha clase. Pero si creamos en la clase CliFinanciar dos procedimientos Property públicos llamados Código, que llamen a los anteriores como se indica en el código fuente 337.

```
Public Property Let Codigo(ByVal vnCodigo As Integer)
Cliente_Codigo = vnCodigo
End Property
' ------------------------------------------------
Public Property Get Codigo() As Integer
Codigo = Cliente_Codigo
End Property
```

Código fuente 337

Podremos acceder desde el exterior de la clase mediante una serie de intermediarios a la propiedad Código de la clase Cliente, sin necesidad de crear un interfaz para la misma, como muestra el esquema de la figura 372.

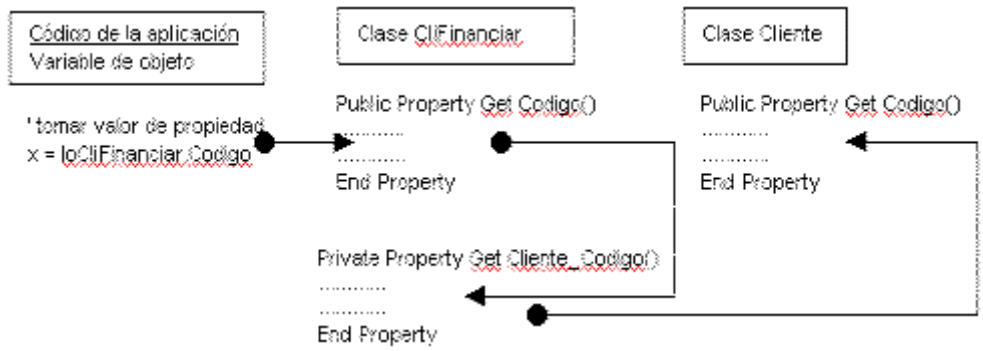


Figura 372. Esquema de llamadas en herencia sin interfaces.

Repetimos la misma operación para crear la propiedad Nombre y el método MostrarDatos() en la clase CliFinanciar (código fuente 338).

```
Public Sub MostrarDatos()
Cliente_MostrarDatos
End Sub
' ------------------------------------------------
Public Property Get Nombre() As String
Nombre = Cliente_Nombre
End Property
```

```
Public Property Let Nombre(ByVal vcNombre As String)
Cliente_Nombre = vcNombre
End Property
```

Código fuente 338

El código del procedimiento Main() en esta aplicación sólo necesitará emplear una variable de objeto instanciada de la clase CliFinanciar. Al ser esta clase hija de la clase Cliente, utilizaremos las propiedades y métodos de Cliente como si estuvieran escritos en la propia clase CliFinanciar, veámoslo en el código fuente 339.

```
Public Sub Main()
' declarar objeto CliFinanciar,
' esta clase hereda de Cliente
Dim loCliFinanciar As CliFinanciar
' ******************
' no es necesario declarar una variable
' de tipo Cliente para que actúe como
' interfaz
' ******************
' instanciar objeto
Set loCliFinanciar = New CliFinanciar
' utilizar los métodos propios de la
' clase hija
loCliFinanciar.BaseFinanc = 170000
loCliFinanciar.Tipo = 6
loCliFinanciar.CalcularFinanc
loCliFinanciar.MostrarCalculo
' utilizar métodos de la clase padre
' pero a través de la clase hija
loCliFinanciar.Codigo = 443
loCliFinanciar.Nombre = "Emilio Manzano"
loCliFinanciar.MostrarDatos
Set loCliFinanciar = Nothing
End Sub
```

Código fuente 339

Con esta técnica ya nos aproximamos un poco más al estilo de herencia de los lenguajes OOP.

Procedimientos Friend

Al declarar dentro del código de una clase, una propiedad o método público, este puede ser accedido por cualquier elemento de la aplicación. Si posteriormente convertimos la aplicación en un componente ActiveX, y la referida clase dentro del componente es definida como pública, todas las propiedades y métodos públicos de la clase podrán ser accedidos no sólo desde el propio componente (aplicación) ActiveX, sino desde cualquier aplicación cliente que haga uso de dicho componente ActiveX, lo cual puede representar un riesgo en el caso de elementos de la clase que estén diseñados exclusivamente para funcionamiento interno de la misma. Consulte el lector el tema dedicado a ActiveX para una mayor información sobre estos componentes.

La solución a este inconveniente pasa por declarar como Friend aquellos procedimientos de la clase que queramos poner a disposición de otras clases dentro de la misma aplicación, pero que no deban ser utilizados por una aplicación cliente.

En el código fuente 340 al declararse Friend, sólo podrá ser utilizado por la propia clase que lo define u otra siempre dentro aplicación que lo contiene.

```
Friend Sub NombreMay()
Me.Nombre = UCase(Me.Nombre)
End Sub
```

Código fuente 340

Manipulación indirecta de objetos; la función CallByName()

Existe una técnica para el manejo de objetos, que si bien no es la habitual, puede ser útil para emplear en determinadas ocasiones. Se basa en el uso de la función CallByName(), cuya sintaxis es la siguiente:

```
CallByName(oObjeto, cMetodoProp, kTipoLlamada [, aParametros()])
```

- oObjeto. Nombre del objeto sobre el que se va a realizar la operación.
- cMetodoProp. Cadena con el nombre de la propiedad o método a llamar.
- kTipoLlamada. Constante con el tipo de operación a efectuar, según la tabla 89, correspondiente al tipo enumerado vbCallType, predefinido por VB.

Constante	Valor	**Descripción**
VbMethod	1	Ejecuta un método.
VbGet	2	Obtiene el valor de una propiedad.
VbLet	4	Asigna un valor a una propiedad.
VbSet	8	Asigna un objeto a una propiedad.

Tabla 89

- aParametros(). Array con los parámetros necesarios para ejecutar el método.

Supongamos que nos encontramos ante la circunstancia de desarrollar una aplicación, en la cuál debemos proporcionar al usuario, un medio de introducir el nombre de un método o propiedad de un objeto para que la aplicación lo ejecute.

Está claro que el elemento que hará de interfaz entre el usuario y el objeto, será un formulario. Pero, ¿como hemos de escribir el código, para que al introducir por ejemplo el nombre de un método en un TextBox, dicho método sea asociado al objeto y ejecutado?. Es evidente que el código fuente 341 no funcionaría.

```
' declarar variable de objeto
Dim loDatos As Datos

' instanciar un objeto de la clase Datos
Set loDatos = New Datos

' ejecutar el nombre del método contenido
' en el control txtMetodo
lodatos.Me.txtMetodo.Text
```

Código fuente 341

La última línea del anterior fuente provocará un error, puesto que el compilador intentará buscar una propiedad o método del objeto loDatos con el nombre Me; como es natural, no puede adivinar que lo que intentamos hacer, es extraer el nombre contenido en el TextBox y ejecutarlo como uno de sus métodos.

Ante tales situaciones, debemos hacer uso de CallByName(), que nos permitirá acceder al contenido del objeto. La aplicación de ejemplo PorNombre, muestra un caso de este tipo: disponemos de un formulario en el cuál introducir los valores para las propiedades de un objeto y el método a ejecutar. Si pulsamos el botón Ejecutar CallByName, se creará un objeto, y utilizando dicha función, se le pasarán los valores de las propiedades y ejecutará el método que aparece en uno de los TextBox del formulario. En el código fuente 342, se muestra el código de dicho botón.

```
Private Sub cmdEjecutar_Click()
' declarar variable de objeto
Dim loDatos As Datos
' instanciar un objeto de la clase Datos
Set loDatos = New Datos
' manipularlo mediante la función CallByName()
' y el contenido de los controles del formulario
CallByName loDatos, "Clave", VbLet, Me.txtClave.Text
CallByName loDatos, "Nombre", VbLet, Me.txtNombre.Text
CallByName loDatos, Me.txtMetodo.Text, VbMethod
End Sub
```

Código fuente 342

Para que el lector aprecie la diferencia entre las dos técnicas, el otro botón del formulario: Técnica habitual, muestra el modo normal de manejo de objetos, que salvo excepciones como la mostrada, es el que debemos de utilizar. El código de este control, puede ser consultado cargando este ejemplo en el entorno de VB.

Formularios con "Clase"

Nuestro trabajo con clases hasta el momento, ha consistido en la creación y manipulación de objetos de forma interna a la aplicación, sin ofrecer posibilidad de acción al usuario. La clase Cliente que nos viene acompañando desde los comienzos de este tema ha cumplido su tarea a la perfección. Pero ahora se nos plantea un problema, ¿qué debemos hacer si el usuario necesita manipular las propiedades de los objetos de la clase Cliente?.

La respuesta es utilizando un formulario, pero no de la forma que un principio se podría pensar, creando el formulario por una parte, el objeto cliente por otra y traspasando información desde el formulario al objeto; el propio formulario será un objeto cliente. Esto es posible debido a que un formulario es una clase que proporciona una interfaz de usuario. Una clase un tanto especial, ya que no tenemos que construirla partiendo de cero, pero es una clase al fin y al cabo. Podemos crear nuestras propiedades, métodos y eventos, además de utilizar los que vienen por defecto con el formulario. Esto último es una gran ventaja, ya que no debemos emplear tiempo en desarrollar el funcionamiento básico del interfaz, puesto que viene incluido.

En base a esta explicación, el lector podría estar preguntándose en que ocasiones es conveniente crear una clase normal y en cuales desarrollarla en un formulario. Podemos decir sin que suponga una norma estricta, que siempre que una clase necesite de alguna acción por parte del usuario, hemos de desarrollarla en un módulo de formulario. Cuando el cometido de la clase no necesite la aportación del usuario, deberá desarrollarse en un módulo de clase.

En este apartado utilizaremos como ejemplo la aplicación *FrmCls*, para mostrar la manera de agregar un componente visual a nuestras clases.

Comencemos con la clase Cliente. Para adaptarla a un formulario, insertaremos un módulo de este tipo en el proyecto, al que llamaremos frmCliente, y en donde situaremos el código que contenía la anterior clase Cliente con algunas variaciones. Desaparecen algunos métodos, se incorporan otros nuevos y la propiedad Operación, que contiene una cadena que indica cuando se está creando un nuevo objeto frmCliente o visualizando uno existente. En el código fuente 343 se muestran las novedades más significativas de este módulo. El lector puede consultarlo al completo en el proyecto desde el entorno de VB.

```
' declaraciones de la clase
' declarar eventos
Public Event AceptarDatos()
' declarar variables de propiedad
Private mcOperacion As String
Private mcCodigo As String
Private mcNombre As String
Private mcDireccion As String
Private mcCiudad As String
' ------------------------------------------
' propiedad Operacion
Public Property Let Operacion(ByVal vcOperacion As String)
mcOperacion = vcOperacion
End Property
Public Property Get Operacion() As String
Operacion = mcOperacion
End Property
' ------------------------------------------
Public Sub CargarDatos()
' cargar las propiedades del objeto
' actual en los controles del formulario
Me.txtCodigo.Text = Me.Codigo
Me.txtNombre.Text = Me.Nombre
Me.txtDireccion.Text = Me.Direccion
Me.txtCiudad.Text = Me.Ciudad
End Sub
' ------------------------------------------
Public Sub Ver()
' mostrar este objeto
' se visualiza en estilo modal
' para que el usuario no pueda
' cambiar al objeto frmClientes
' hasta que no termine de utilizar
```

```
' este objeto
Me.Show vbModal
End Sub
' ------------------------------------------
Public Sub Quitar()
' descargar este objeto
Unload Me
End Sub
' ------------------------------------------
Private Sub cmdAceptar_Click()
' pasar el contenido de los controles
' del formulario a las propiedades
Me.Codigo = Me.txtCodigo.Text
Me.Nombre = Me.txtNombre.Text
Me.Direccion = Me.txtDireccion.Text
Me.Ciudad = Me.txtCiudad.Text
' generar un evento que será
' tomado por el objeto frmClientes
RaiseEvent AceptarDatos
End Sub
```

Código fuente 343

La representación visual de esta clase quedaría como muestra la figura 373.

Figura 373. Formulario de la clase frmCliente.

Al terminar de crear un nuevo objeto frmCliente pulsando en su botón cmdAceptar, el objeto frmCliente genera un evento notificando de esta situación, de forma que el objeto frmClientes pueda ser consciente de esta situación, agregando el objeto recién creado a su colección. Más adelante veremos como la clase frmClientes captura el evento generado.

En lo que respecta a la clase Clientes, crearemos el formulario frmClientes, que contendrá la clase Clientes con algunas variaciones, orientadas también a proporcionar una participación del usuario en la manipulación de la clase.

Para el formulario se ha incluido un ComboBox que se encargará de mostrar los objetos frmCliente que contiene la colección del objeto frmClientes. También se incluyen varios CommandButton para las diferentes acciones a realizar sobre los objetos contenidos en la colección, como se muestra en la figura 374.

Figura 374. Formulario de la clase frmClientes.

El código de esta clase ha sufrido un importante cambio con respecto a la clase Clientes. Se sigue usando la variable de módulo mcllClientes como Collection para contener los objetos frmCliente, pero se han suprimido algunos métodos y otros como Add() han variado su contenido para adaptarse a las necesidades del formulario.

Para poder captar los eventos que genere un objeto frmCliente, se ha declarado a nivel de módulo la variable mfrmCliente, con la cláusula WithEvents. De esta forma se crea el procedimiento de evento mfrmCliente_AceptarDatos(), para saber cuando el usuario ha terminado de usar el objeto frmCliente. El código fuente 344, muestra el código de la clase frmClientes.

```
' declarar variable de objeto de clase frmCliente
' se declara para poder capturar sus eventos
Private WithEvents mfrmCliente As frmCliente
' declarar variables de módulo
' esta variable contendrá los objetos
' frmCliente que se vayan creando
Private mcllClientes As Collection
' ------------------------------------------
Public Sub Add()
' agregar el objeto frmCliente actual
mcllClientes.Add mfrmCliente
' incorporar el valor de la propiedad Codigo
' del objeto frmCliente al ComboBox
Me.cboClientes.AddItem mfrmCliente.Codigo
Me.cboClientes.Text = mfrmCliente.Codigo
' ocultar el objeto frmCliente y
' eliminar su referencia
mfrmCliente.Quitar
Set mfrmCliente = Nothing
End Sub
' ------------------------------------------
Public Sub FinVisualizar()
' al terminar la consulta del objeto frmCliente
' ocultarlo y eliminar su referencia
mfrmCliente.Quitar
Set mfrmCliente = Nothing
End Sub
' ------------------------------------------
Public Sub Ver()
' visualizar el objeto de esta clase
Me.Show
End Sub
' ------------------------------------------
Private Sub cmdAgregar_Click()
' instanciar un objeto de la clase frmCliente
```

```
Set mfrmCliente = New frmCliente
' dar valor a sus propiedades y mostrarlo
mfrmCliente.Operacion = "AGREGAR"
mfrmCliente.Caption = "Agregar"
mfrmCliente.Ver
End Sub
' ------------------------------------------
Private Sub cmdEliminar_Click()
Dim lfrmCliente As frmCliente
Dim lnInd As Integer
Dim lcValorEliminar As String
' tomar el valor actual que aparece en el
' ComboBox y que usaremos para eliminar un
' objeto frmCliente de la colección contenida
' en este objeto frmClientes
lcValorEliminar = Me.cboClientes.Text
For Each lfrmCliente In mcllClientes
   lnInd = lnInd + 1

   ' si encontramos el valor
   If lcValorEliminar = lfrmCliente.Codigo Then
      ' eliminar el objeto de la colección
      mcllClientes.Remove (lnInd)
      Exit For
   End If
Next
' eliminar también la cadena que muestra el código
' de cliente en el ComboBox para que no se produzcan
' inconsistencias entre los objetos de la colección
' y los valores del ComboBox
For lnInd = 0 To (Me.cboClientes.ListCount - 1)
   ' si encontramos el elemento del Combo, eliminarlo
   ' y dejar como elemento actual el último de la lista
   ' del combo
   If Me.cboClientes.List(lnInd) = lcValorEliminar Then
      Me.cboClientes.RemoveItem (lnInd)
      Me.cboClientes.Text = _
         Me.cboClientes.List(Me.cboClientes.ListCount - 1)
      Exit For
   End If
Next
End Sub
' ------------------------------------------
Private Sub cmdVisualizar_Click()
Dim lfrmCliente As frmCliente
' buscar en la colección un objeto
' con el mismo código que el mostrado
' actualmente en el ComboBox
For Each lfrmCliente In mcllClientes
   If Me.cboClientes.Text = lfrmCliente.Codigo Then
      ' si se encuentra el objeto frmCliente
      ' asignarlo a la variable, configurar
      ' su modo de uso y mostrarlo
      Set mfrmCliente = lfrmCliente
      mfrmCliente.Operacion = "VISUALIZAR"
      mfrmCliente.Caption = "Visualizar"
      mfrmCliente.CargarDatos
      mfrmCliente.Ver
      Exit For
   End If
Next
End Sub
' ------------------------------------------
Private Sub Form_Initialize()
' al inicializar este objeto frmClientes
' instanciar un nuevo objeto Collection
Set mcllClientes = New Collection
```

```
End Sub
' ------------------------------------------
Private Sub Form_Terminate()
' eliminar el objeto colección
Set mcllClientes = Nothing
End Sub
' ------------------------------------------
Private Sub mfrmCliente_AceptarDatos()
' cuando el objeto frmCliente genere
' el evento AceptarDatos, el flujo del
' programa pasará por este procedimiento
' preparado para ese evento, en función de
' la operación realizada por el objeto
' frmCliente se hará la oportuna llamada
' a un método de esta clase --frmClientes--
Select Case mfrmCliente.Operacion
Case "AGREGAR"
   Me.Add

Case "VISUALIZAR"
   Me.FinVisualizar
End Select
End Sub
```

Código Fuente 344

Invitamos al lector a que ejecute la aplicación y pruebe a crear nuevos objetos frmCliente que serán incluidos en la colección; a visualizarlos después de creados, y finalmente eliminarlos.

Variables globales de formulario

En todos los ejemplos de este curso que utilizan formularios, la forma de crearlos es como hemos mostrado para cualquier objeto de una clase: se declara una variable de la clase del formulario y se crea una nueva instancia de esa clase en forma de objeto, asignándola a la variable, como muestra el código fuente 345.

```
Dim lfrmDatos As frmDatos
Set lfrmDatos = New frmDatos
```

Código fuente 345

Existe otra forma de crear y manejar instancias de un formulario, más fácil que la anterior aunque más alejada también de las maneras y modos habituales de las técnicas OOP, que consiste en emplear el nombre del formulario directamente sin haber creado previamente una variable de su clase ni haberlo instanciado.

```
frmDatos.Caption = "Información"
frmDatos.Show
```

Código fuente 346

En el código fuente 346 se asignaría el título a un formulario de una aplicación y se mostraría al usuario, sin necesidad de haberlo declarado ni instanciado en primer lugar. Esto es posible gracias a las llamadas *variables globales ocultas de formulario*. La primera vez que se referencia a un formulario por su nombre en el código de la aplicación, VB define una de estas variables y le asigna una instancia de ese formulario. Toda esta operación es totalmente transparente de cara al programador, dando la impresión de estar trabajando directamente con ese formulario sin necesidad de emplear variables, aunque como hemos dicho, todo ese trabajo con variables sí se hace, pero de manera oculta, al igual que las llamadas a los métodos Initialize() y Load(), que crean la instancia y cargan el formulario; todo se hace de forma automática.

Esta es una de las diferencias entre los módulos de clase y formularios, ya que utilizando una clase normal, siempre hemos de declarar la correspondiente variable y crear una instancia de la clase, asignando el objeto a la variable.

Una vez que hayamos terminado de usar el formulario, para eliminar la referencia al mismo y el consiguiente consumo de recursos, sólo hemos de asignarle Nothing a su variable de formulario.

```
Set frmDatos = Nothing
```

Utilidades y herramientas para el manejo de clases

Generador de clases (Class Wizard)

Esta utilidad nos permite diseñar las clases o jerarquía de las mismas para una aplicación, proporcionándonos ayuda en la creación de las propiedades, métodos, eventos y colecciones de las clases, pero sólo a nivel de diseño, sin incorporar código. Una vez realizada la plantilla de la clase, el generador creará las definiciones para los diferentes elementos de la misma, debiendo encargarse el programador de escribir el código de los procedimientos.

La aplicación *GenClase* muestra un ejemplo de una clase creada utilizando esta herramienta.

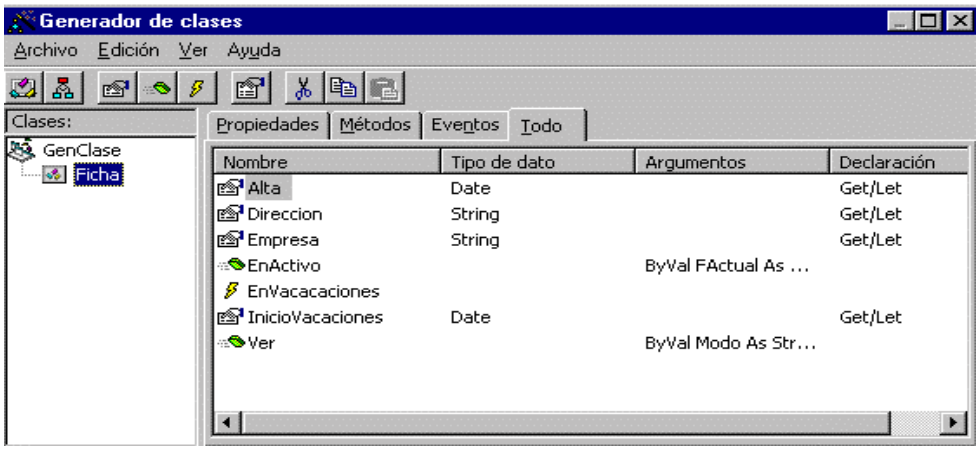


Figura 375. Generador de clases.

En el panel izquierdo de esta ventana disponemos de las clases y colecciones incluidas en el proyecto actual. El panel derecho muestra los diferentes elementos de la clase o colección seleccionada.

Podemos ver todos los elementos de la clase o según su categoría, en función de la ficha seleccionada en la ventana.

En la barra de herramientas disponemos de las opciones principales para trabajar con este asistente.

-  **Agregar nueva clase.** Incorpora una nueva plantilla de clase al proyecto. Mediante esta ventana de diálogo damos nombre a la nueva clase. Es posible además documentar la clase incorporando información en la ficha Atributos de esta ventana. Esta ficha estará presente en la mayoría de las ventanas de creación de elementos de una clase, facilitándonos la tarea de proporcionar información sobre métodos, propiedades, etc.

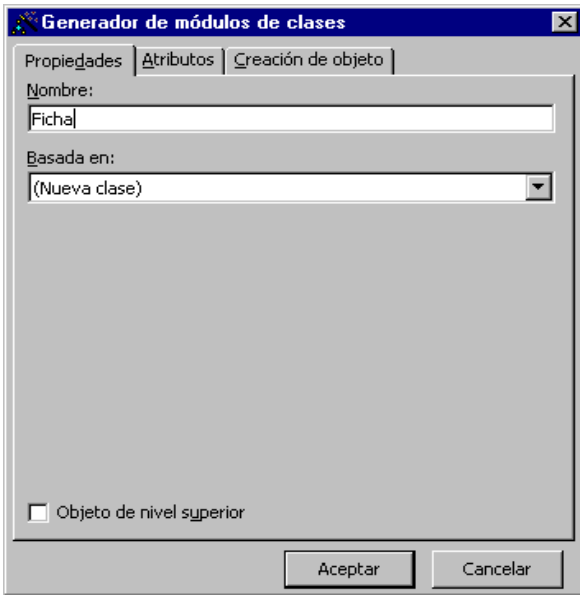


Figura 376. Ventana para agregar clase.

-  **Agregar nueva colección.** Crea una colección para guardar objetos de una clase existente en el proyecto o a partir de una nueva clase creada al mismo tiempo. La figura 377 muestra la ventana para crear colecciones.

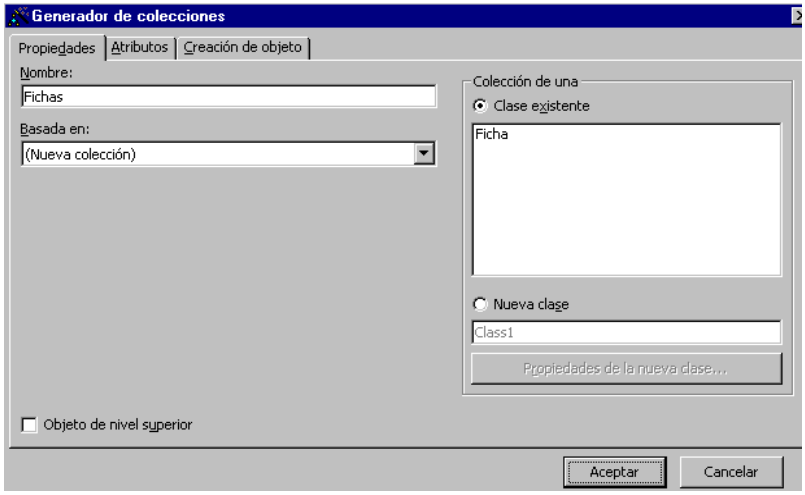


Figura 377. Ventana para agregar una colección.

- **Agregar propiedad.** Esta opción permite incorporar una nueva propiedad a la clase, e indicar su tipo mediante la ventana de la figura 378.

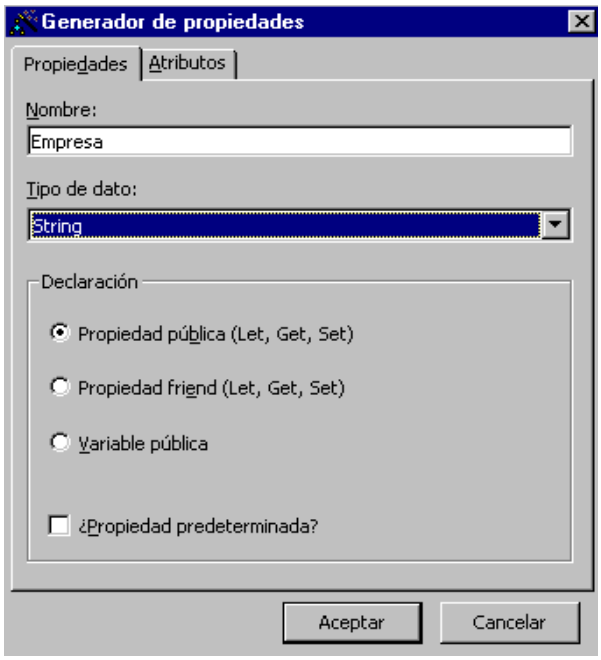


Figura 378. Ventana de creación de propiedades.

- **Agregar método.** Mediante esta opción especificamos las características de un método para la clase: nombre, parámetros, tipo, etc.

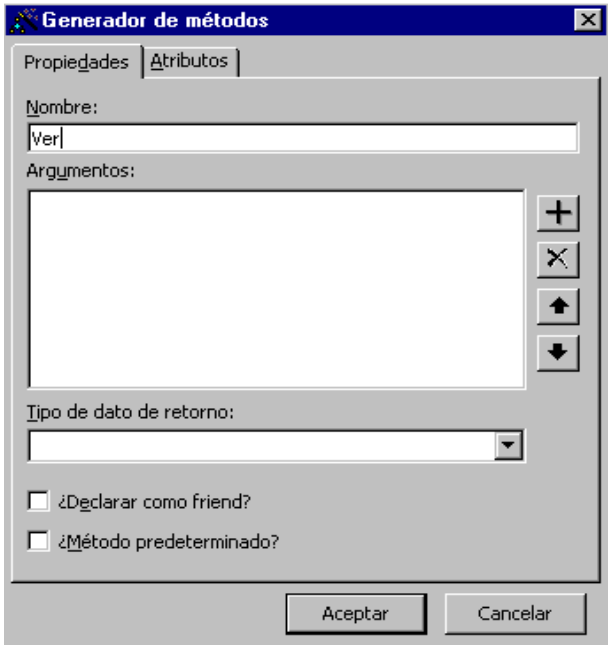


Figura 379. Ventana de creación de métodos.

- **Agregar evento.** Con esta opción creamos la declaración de un evento para la clase.

Figura 380. Ventana de definición de eventos.

Para no repetir todo el código de la clase en este tema, el lector puede consultar el resultado de la creación de la clase Ficha para este ejemplo, cargándola en VB y visualizando su módulo de código. Como podrá comprobar, las definiciones resultantes de métodos, propiedades, etc., nos evitan una buena cantidad de trabajo a la hora de escribir código. El único inconveniente que podemos encontrar es que si estamos acostumbrados a una notación determinada en el código, es posible que la proporcionada por el generador no se adapte a la nuestra, lo que nos obligará a retocar parte del código de la clase.

Examinador de objetos (Object Browser)

Mediante esta estupenda herramienta, podemos visualizar las clases contenidas en las librerías estándar de objetos de Visual Basic, las que tengan establecida una referencia en el proyecto actualmente en desarrollo, y las clases propias de dicho proyecto. Adicionalmente es posible acceder desde el examinador al código de las clases propias del proyecto.

Supongamos que desde la aplicación de ejemplo OOCliente, arrancamos el examinador de objetos y queremos que nos muestre todas las referencias a la palabra "ciudad" que existen en las clases de dicha aplicación. Para ello teclearemos dicha palabra en el ComboBox *Texto de búsqueda* y pulsaremos *Enter* o el botón de *Búsqueda*; el resultado nos mostrará cada una de las clases que contienen un miembro formado por esa palabra y los miembros correspondientes, teniendo en cuenta que el examinador de objetos también considera como clase a los módulos de código estándar, y como miembros a los procedimientos, métodos, propiedades, etc. Véase un ejemplo en la figura 381.

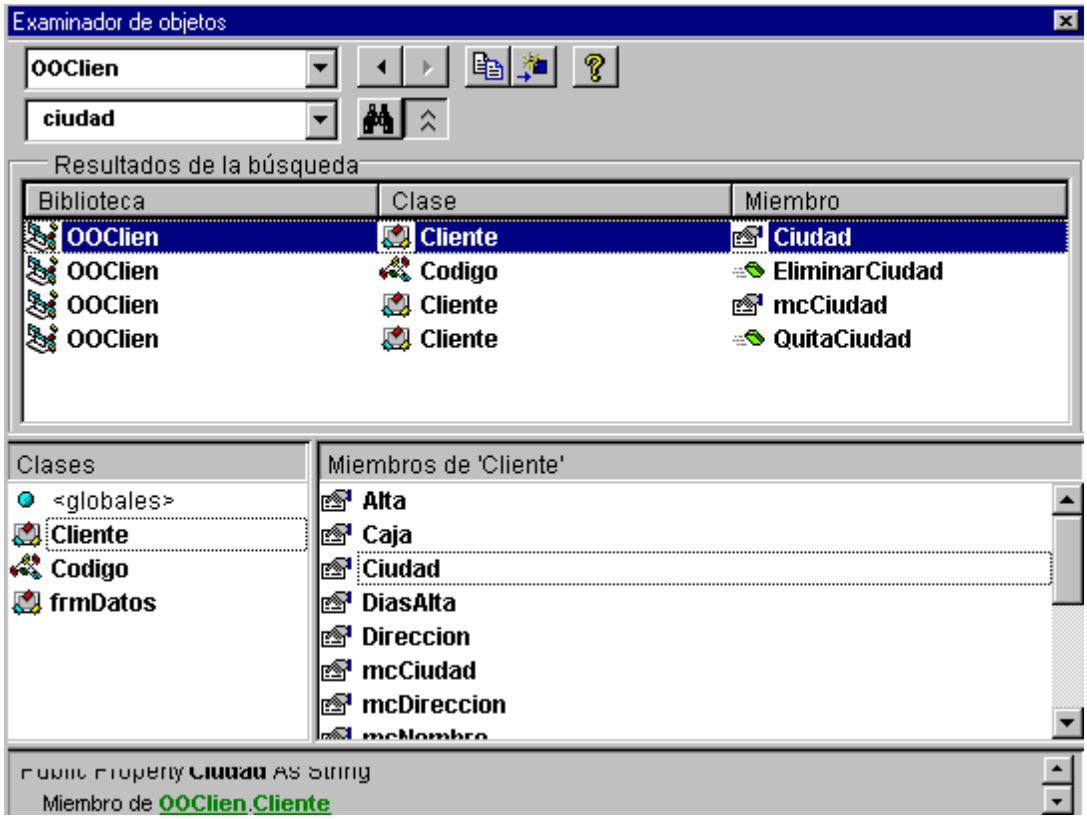


Figura 381. Examinador de objetos mostrando el resultado de una búsqueda.

Para información adicional sobre el examinador de objetos, el lector puede consultar el tema *El Entorno de Desarrollo*, apartado *Ventanas complementarias*.

Objetos del sistema

Visual Basic dispone de un conjunto de objetos, que son creados automáticamente cada vez que ejecutamos una aplicación, y destruidos al finalizar su ejecución. Estos objetos tienen un ámbito global, pueden usarse desde cualquier punto del programa, y su uso nos puede servir de apoyo en la programación de aspectos generales de nuestra aplicación, son los denominados *objetos del sistema*. A continuación, pasaremos a realizar una breve descripción de cada uno. Si el lector precisa de información más detallada, puede consultar la ayuda de VB, donde encontrará amplia información de cada uno de estos objetos.

- **App.** Proporciona información genérica sobre la aplicación que estamos ejecutando, tal como el nombre de su fichero ejecutable, título, versión, si es un ejecutable independiente o componente ActiveX, etc.

- **Clipboard.** Permite acceder y manipular el Portapapeles de Windows, realizando las acciones habituales del mismo: copiar, cortar y pegar.

  Siempre que copiemos información al Portapapeles, hemos de borrar el contenido anterior utilizando el método *Clear()* de este objeto.

Es posible tener simultáneamente información de diferente tipo, como texto y gráficos, y recuperarla por separado. Pero debemos tener en cuenta, que cada vez que copiemos información en este objeto, desaparecerá la que hubiera anteriormente con el mismo tipo.

- **Debug.** Este objeto se emplea para enviar información a la ventana de Depuración de VB, estando disponible únicamente en tiempo de desarrollo de la aplicación.

- **Err.** Este objeto proporciona información sobre los errores producidos en tiempo de ejecución, de forma que puedan ser tratados por la aplicación. El programador también puede utilizar este objeto para provocar errores, mediante su método Raise(), ante ciertas situaciones durante el transcurso del programa.

- **Printer.** Este objeto representa a una de las impresoras del sistema. Si disponemos de varias, hemos de emplear la colección *Printers*, para seleccionar una.

  Un objeto Printer nos permite seleccionar el número de copias a imprimir de un documento, el tipo de fuente, tamaño del texto, etc.

- **Screen.** Emplearemos este objeto cuando necesitemos tener información sobre los formularios que se encuentran en pantalla, su tamaño, formulario y control activo, tipo de puntero del ratón, etc.

11

ActiveX

¿Qué es ActiveX?

La creciente complejidad en el desarrollo de software, motivada por las necesidades de los usuarios de disponer de aplicaciones cada vez más potentes y rápidas, que sean capaces de afrontar los nuevos problemas que se plantean en el campo de la informática (léase Internet y todo lo que le rodea), han dado como resultado el *modelo de desarrollo basado en componentes*, como respuesta a los retos que han de resolver las aplicaciones actuales.

Según este modelo de desarrollo, una aplicación puede ser dividida en varias partes o componentes, cada uno de estos componentes podrá ser compilado por separado y disponer de funcionalidad propia, de forma que sea posible integrarlo en otra aplicación, a la manera de una pieza de maquinaria. Esto incrementa considerablemente la capacidad de desarrollo, puesto que en buena parte el programador necesita emplear menos tiempo en desarrollar, limitándose a ensamblar componentes.

La forma en que los componentes se conectan y comunican entre sí, viene dada por la especificación COM (Component Object Model ó Modelo de Objetos basado en Componentes). Aquí es donde entra en juego ActiveX, ya que es una tecnología basada en COM.

ActiveX se puede definir como una tecnología que permite la comunicación entre componentes compilados por separado y con funcionalidad independiente, que pueden haber sido desarrollados utilizando diferentes herramientas que la soporten: Visual Basic, Visual C++, Java, Delphi, etc.

ActiveX es la denominación para la tecnología antes conocida como OLE. El cambio de nombre no es sino un enfoque comercial hacia Internet, pero el contenido es el mismo. Recientemente se han incorporado nuevas características como la programación por hebras, generación de eventos, uso de procedimientos Friend, creación de formularios no-modales, etc.

Antes de comenzar a programar utilizando esta técnica, debemos conocer una serie de conceptos y elementos que conforman los pilares teóricos en los que se basa ActiveX.

Componentes ActiveX

Un componente ActiveX se puede definir como un fragmento de código compilado basado en la especificación COM/DCOM, que encapsula un conjunto de reglas de negocio, entendiendo estas reglas como parte de la arquitectura de tres capas utilizada en aplicaciones cliente/servidor, y que se ajustan al siguiente modelo:

- Reglas de interfaz de usuario.
- Reglas de negocio.
- Tratamiento de datos.

Dicho componente ActiveX está formado por una o más clases, y su cometido es crear objetos de las mismas, proporcionándolos a las aplicaciones cliente que lo solicitan, mediante el interfaz de automatización ActiveX, que no es otra cosa, sino la definición de los interfaces que componen los objetos contenidos en el componente.

En versiones anteriores, a estos componentes se les denominaba servidores OLE, actualmente se denominan servidores ActiveX o clientes ActiveX, en función de si proporcionan o solicitan objetos respectivamente. Sin embargo, ya que según la tarea a realizar, un mismo componente puede actuar como cliente y servidor, la denominación más común es *componente ActiveX* o *componente de código ActiveX* cuando hablamos de componentes ActiveX DLL o EXE. El término *código* se emplea para diferenciarlos del resto de componentes ActiveX que tienen una mayor orientación visual, como los controles y documentos ActiveX.

ActiveX DLL

Un ActiveX DLL es un tipo de componente compilado en formato de librería (DLL), de manera que la única forma de acceder a su interior es instanciando objetos de las clases que contiene. Entre sus ventajas figura el hecho de que se ejecuta en el mismo espacio de proceso en el que se ejecuta la aplicación cliente que solicita los servicios del ActiveX DLL, lo cual proporciona un mejor rendimiento en la ejecución al realizarse las llamadas dentro del mismo proceso.

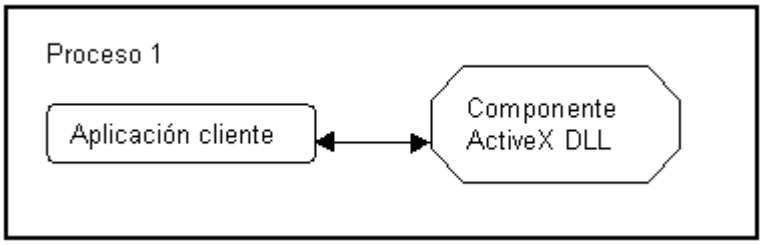


Figura 382. Comunicación entre aplicación cliente y componente ActiveX DLL.

Una de las principales diferencias entre este tipo de componentes y las tradicionales DLL's, reside en la forma de comunicación con las mismas. Mientras que para una DLL normal, debemos declarar la

función a utilizar, accediendo a dicha función mediante un puntero de entrada, con un ActiveX DLL utilizamos objetos mediante los interfaces proporcionados por el componente.

ActiveX EXE

Un ActiveX EXE es un componente compilado en formato ejecutable (EXE). Al igual que en un ActiveX DLL, es posible crear objetos de las clases contenidas en este componente, existiendo la ventaja de poder ejecutarlo como un programa independiente.

Un ejemplo de ActiveX EXE lo constituye Excel. Por una parte es posible ejecutarlo como una aplicación normal, por otro lado puede utilizarse como servidor, solicitándole objetos desde una aplicación Visual Basic. Por último podemos crear un componente ActiveX desde VB que proporcione objetos a Excel, en este caso Excel actuaría como cliente, y nuestro componente como servidor.

Este tipo de componente requiere un espacio de proceso propio para ejecutarse, por lo que si es solicitado por una aplicación cliente, esta última debe traspasar su propia zona de proceso y conectar con la zona de proceso del ActiveX EXE para comunicarse con dicho componente. Toda esta interconexión entre procesos se traduce en una disminución de rendimiento con respecto a la obtenida al trabajar con ActiveX DLL.

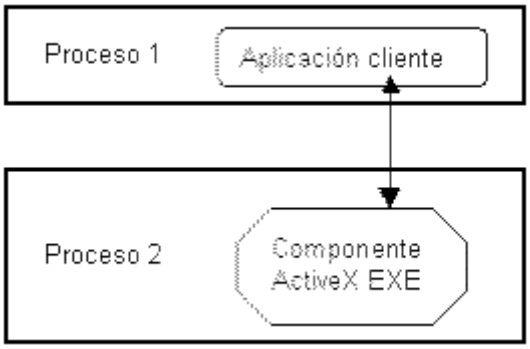


Figura 383. Comunicación entre aplicación cliente y componente ActiveX EXE.

Como contrapartida, el uso de componentes ActiveX EXE proporciona las siguientes ventajas:

- Las aplicaciones de 16 bits pueden acceder a un componente ActiveX EXE.
- Si se utiliza DCOM para acceder a un componente, este debe ser un ActiveX EXE.
- Si se produce un error durante la ejecución del ActiveX EXE, la aplicación cliente puede evitar verse afectada por el error en función de la situación en la que se encuentre en ese momento con respecto al componente. Dicho de otro modo, si el servidor ActiveX cae, no tiene por qué arrastrar obligatoriamente a las aplicaciones cliente conectadas a él.

Control ActiveX

Un control ActiveX es un elemento generalmente visual, creado por el programador y almacenado en formato OCX, que puede ser mostrado en aplicaciones que soporten este tipo de controles, desde programas estándar hasta páginas Web.

Una de las características que lo distinguen de otro tipo de componentes ActiveX, es que normalmente los controles tienen algún elemento visual como puede ser un CheckBox, TextBox, OptionButton, etc.

Una vez creado, un control ActiveX puede ser incorporado a la caja de herramientas de VB y utilizado como uno más de los controles estándar.

Durante la creación de un componente ActiveX, una recomendación a seguir es definir las reglas de negocio en componentes ActiveX DLL o EXE y no en controles ActiveX, dejando a estos últimos sólo las labores de interfaz con el usuario.

Documento ActiveX

Un documento ActiveX se puede definir como un formulario que puede ser visualizado en una aplicación contenedora de documentos ActiveX tal como Internet Explorer. Para poder crear este tipo de componente, hemos de crear previamente un servidor de documentos ActiveX. Un servidor de documentos ActiveX es el componente que proporciona los documentos ActiveX a la aplicación contenedora que se encargará de visualizarlos.

DCOM

El modelo DCOM (Distributed Component Object Model) no es otra cosa que COM distribuido. ¿Qué quiere decir esto?, pues que la comunicación entre componentes se realiza en una red. La ventaja de esta técnica reside en que no es necesario enviar una copia del componente a cada máquina que vaya a utilizarlo. Sólo es necesaria una única copia del componente funcionando en un puesto de la red, para que pueda ser accesible por todos los demás puestos que lo soliciten. Al trabajar con una sola copia del componente, tenemos la ventaja añadida de que las labores de mantenimiento y actualización del mismo se simplifican enormemente.

Automatización Remota

Este es el sistema de comunicación entre componentes proporcionado en Visual Basic 4. Ha sido sustituido por DCOM, aunque todavía se mantiene por razones de compatibilidad con anteriores versiones y para permitir la comunicación con aplicaciones de 16 bits.

Creación de componentes ActiveX DLL y EXE

Los siguientes puntos de este apartado nos mostrarán los pasos a seguir para crear un componente de código ActiveX.

Establecer el tipo de componente

En primer lugar, debemos seleccionar la opción *Archivo+Nuevo Proyecto* del menú de VB, que mostrará la ventana para la creación de un nuevo proyecto, como vemos en la figura 384.

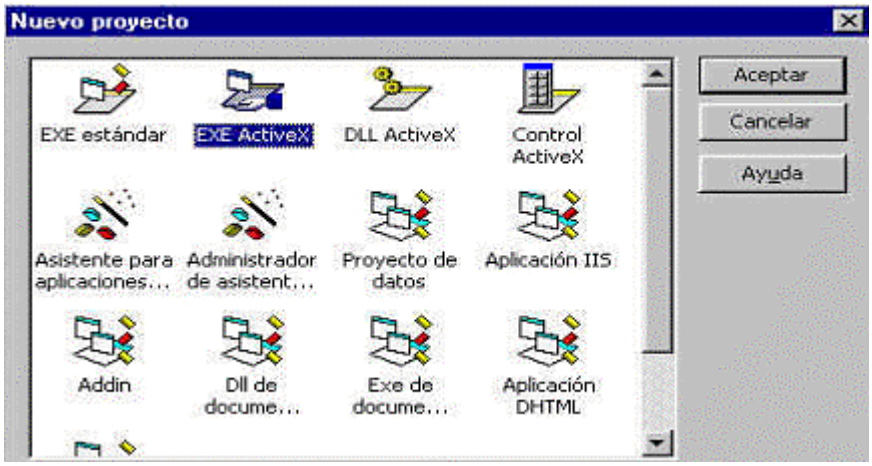


Figura 384. Iniciar un nuevo proyecto.

Hemos de elegir el tipo de componente a desarrollar según la funcionalidad que vaya a desempeñar. Si vamos a utilizar el componente para instanciar objetos de sus clases desde una aplicación cliente, usaremos un ActiveX DLL. Si además de instanciar objetos del componente necesitamos que él mismo se ejecute como una aplicación independiente, deberemos decantarnos por crear un ActiveX EXE.

Se recomienda siempre que sea posible, el uso de un componente ActiveX DLL, debido a que se ejecuta en el mismo proceso que la aplicación cliente, proporcionando un mejor rendimiento.

Propiedades del componente

Después de iniciar un nuevo proyecto, hemos de establecer una serie de propiedades en el mismo, para el correcto funcionamiento del componente a desarrollar.

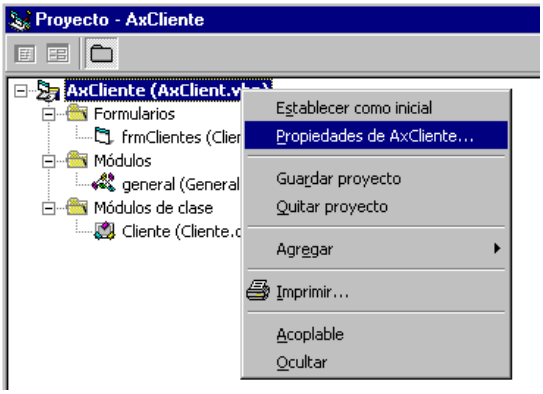


Figura 385. Accediendo a las propiedades desde la ventana de proyecto.

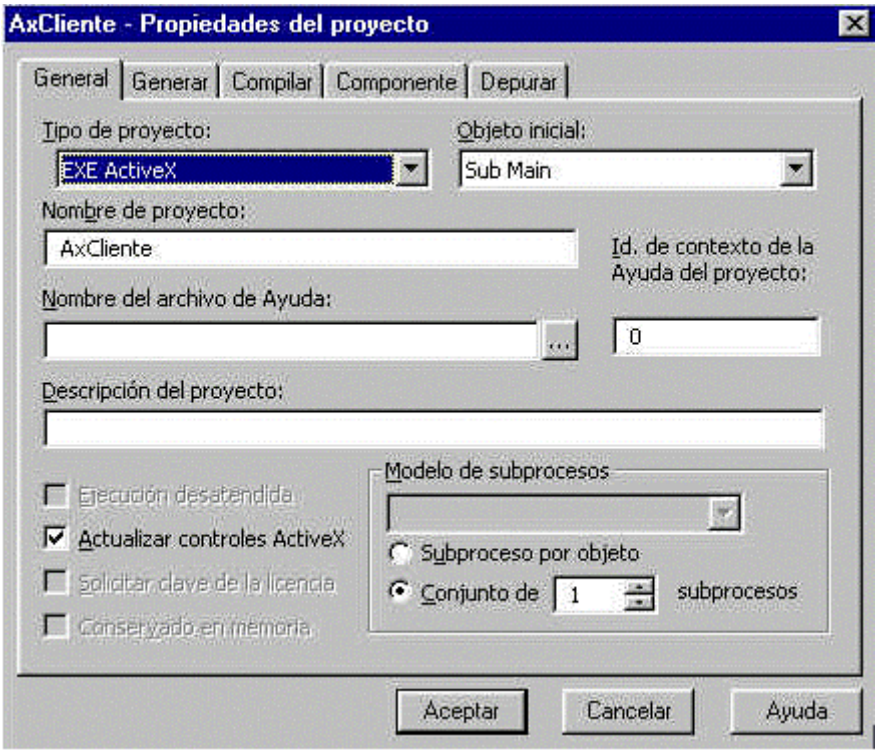


Figura 386. Propiedades generales para un proyecto conteniendo un componente ActiveX

Seleccionaremos el nombre del proyecto en la ventana del explorador de proyectos y abriremos su menú contextual (figura 385). Seleccionando la opción *Propiedades de NombreProyecto...*, que mostrará la ventana de propiedades del proyecto (figura 386).

Para la ficha *General* las propiedades a establecer son las siguientes:

- **Tipo de proyecto.** Este valor se establece al crear el proyecto. Los valores a seleccionar serán DLL ActiveX o EXE ActiveX.

- **Nombre de proyecto.** Es el nombre con el que identificaremos al componente. Este nombre será utilizado en el explorador de objetos y en el Registro de Windows.

- **Objeto inicial.** Si el componente necesita algún tipo de inicialización, indicaremos *Sub Main* para esta propiedad, en caso contrario elegiremos *Ninguno*.

- **Nombre del archivo de Ayuda.** Fichero de ayuda para este proyecto

- **Id. de contexto de la Ayuda del proyecto.** Identificador asociado en la ayuda de este proyecto que se utilizará en el Examinador de objetos para mostrar la ayuda sobre un tema determinado.

- **Descripción del proyecto.** Texto que se muestra en el Examinador de objetos, y que ofrece una breve información sobre el proyecto.

- **Ejecución desatendida.** Marcando esta propiedad, el componente se ejecuta sin la interacción del usuario. Esta opción sólo está disponible si en el proyecto no existen formularios. Si por cualquier causa, el proyecto muestra algún tipo de mensaje con funciones del estilo MsgBox(), dicha ventana no se visualiza y su contenido se escribe en un fichero de log.

- **Actualizar controles ActiveX.** Permite actualizar los controles ActiveX.

- **Solicitar clave de la licencia.** Permite establecer un código de licencia para los componentes ActiveX.

- **Modelo de subprocesos.** Permite establecer la configuración de las hebras (subprocesos) en el proyecto.

    - Subproceso por objeto. Cada nueva instancia de una clase que tenga en su propiedad Instancing el valor Multiuse, se creará en una nueva hebra.

    - Conjunto de n subprocesos. Cada nueva instancia de una clase que tenga en su propiedad Instancing el valor Multiuse, se creará en una hebra del conjunto de hebras indicado en el control de texto asociado a esta propiedad.

Para la ficha *Componente* las propiedades a establecer son las siguientes:

- **Modo de Inicio.** Si el componente es ActiveX EXE y es necesario que funcione como un ejecutable, se marcará el valor *Independiente*, en el resto de los casos seleccionaremos *Componente ActiveX*. Este ajuste referente al modo de inicio del componente, sólo es válido durante la fase de diseño del mismo, ignorándose una vez que el componente es compilado. A partir de ese momento, si el componente se ejecuta directamente haciendo doble clic en su icono o desde el explorador de archivos (ejecutando su fichero EXE), se iniciará en modo ejecutable; si es una aplicación cliente la que pone en marcha el componente, este se iniciará como un servidor ActiveX.

- **Servidor remoto.** Al compilar el componente con esta opción marcada, se crea un fichero de registro (.VBR). La información de este fichero es la que necesita el registro de Windows para ejecutar el componente en una máquina remota. Esta característica sólo está disponible en la edición empresarial.

- **Compatibilidad de la versión.** Mediante esta característica, las modificaciones efectuadas a un componente, no obligarán a recompilar todas las aplicaciones cliente que hagan uso de él, sino sólo las que necesiten hacer uso de las nuevas funcionalidades que aporte la última versión del componente. Este aspecto se verá en detalle en el siguiente apartado.

Clases del componente

En esta etapa, desarrollaremos las clases que necesitará el componente para proporcionar objetos a la aplicación cliente o para su propio uso. Una buena costumbre a la hora de escribir el código de las clases, consiste en incluir las clases con características comunes en un solo componente, definiendo una jerarquía para ese conjunto de clases. Esta práctica facilita el mantenimiento del código del componente y el uso de las clases que contiene, al estar organizadas en una jerarquía.

Creación de formularios en el componente

Si un componente ActiveX necesita comunicarse con el usuario, puede hacerlo empleando formularios. El estilo del formulario puede ser modal o no modal, y su comportamiento dependerá del tipo de componente.

En el caso de componentes ActiveX EXE que proporcionen formularios a una aplicación cliente, debido a que el componente se ejecuta en un proceso propio, el formulario al visualizarse puede aparecer por detrás de la aplicación cliente, dando la impresión de que la aplicación se ha bloqueado, aunque en realidad no sea así.

Para los componentes ActiveX DLL esta situación mejora, ya que al ejecutarse en el mismo proceso que la aplicación cliente, los formularios mostrados por el componente se visualizarán correctamente.

Procedimiento Main de entrada

Cuando necesitemos incorporar código de inicialización para el componente, ya sea ActiveX EXE o DLL, podemos incluir en un módulo de código estándar un procedimiento Main(). En el caso de ActiveX EXE, debemos comprobar el modo de ejecución: componente ActiveX o ejecutable independiente. Este valor se comprobará utilizando la propiedad *StartMode* del objeto *App*, que contiene un valor numérico indicando el modo de ejecución.

A diferencia de la anterior versión de VB, que requería obligatoriamente un procedimiento Main() para inicializar el componente, en esta versión no es necesario salvo en los casos que se necesite inicializar expresamente.

De igual manera, si planeamos desarrollar una jerarquía de clases en un componente, la forma más recomendable de inicializar dicha jerarquía, es en el evento Initialize() de la clase situada en el nivel más alto de la misma.

Compatibilidad de versiones

Un componente puede ser utilizado por una o varias aplicaciones cliente. Si una nueva aplicación cliente o una ya existente, requiere un nuevo tipo de funcionalidad no contemplada actualmente por el componente, esta propiedad permite incorporar el nuevo código al componente y que sólo sea necesario recompilar las aplicaciones que van a utilizar dicha funcionalidad, sin afectar al resto, que seguirán empleando el componente igual que hasta ahora.

Como vimos en el apartado anterior, para acceder a esta propiedad, debemos abrir la ventana de propiedades del proyecto, pestaña Componente, y seleccionar alguno de los tipos de compatibilidad disponibles, como muestra la figura 387.

Los distintos tipos de compatibilidad que se pueden establecer, son los siguientes:

- Sin compatibilidad. No se fuerza la compatibilidad del componente. Esta opción deberá usarse la primera vez que se cree el componente y durante su fase de pruebas, antes de dejarlo disponible para ser usado por aplicaciones cliente.

- Compatibilidad del proyecto. Esta opción deberá utilizarse al compilar el componente, para comenzar a ser utilizado por aplicaciones cliente en modo de prueba. En el TextBox situado en el recuadro de esta propiedad, deberá indicarse la ruta completa del componente compilado. Esta acción, permite mantener la referencia al componente, aunque se realicen cambios en él. De esta manera, una aplicación cliente que haga uso del componente, no estará perdiendo constantemente la referencia al mismo, por los cambios que en él se realicen.

Cada vez que se compila el componente, se crea una nueva librería de tipos (fichero .TLB), que es utilizada por las aplicaciones cliente, para acceder a las clases del componente. El identificador asociado a dicha librería se mantiene igual, de forma que no sea necesario actualizar las referencias hacia el componente que tiene las aplicaciones cliente.

- Compatibilidad binaria. Una vez que se ha terminado de desarrollar el componente, marcaremos esta opción e indicaremos en el TextBox de esta propiedad, la ruta definitiva para el componente.

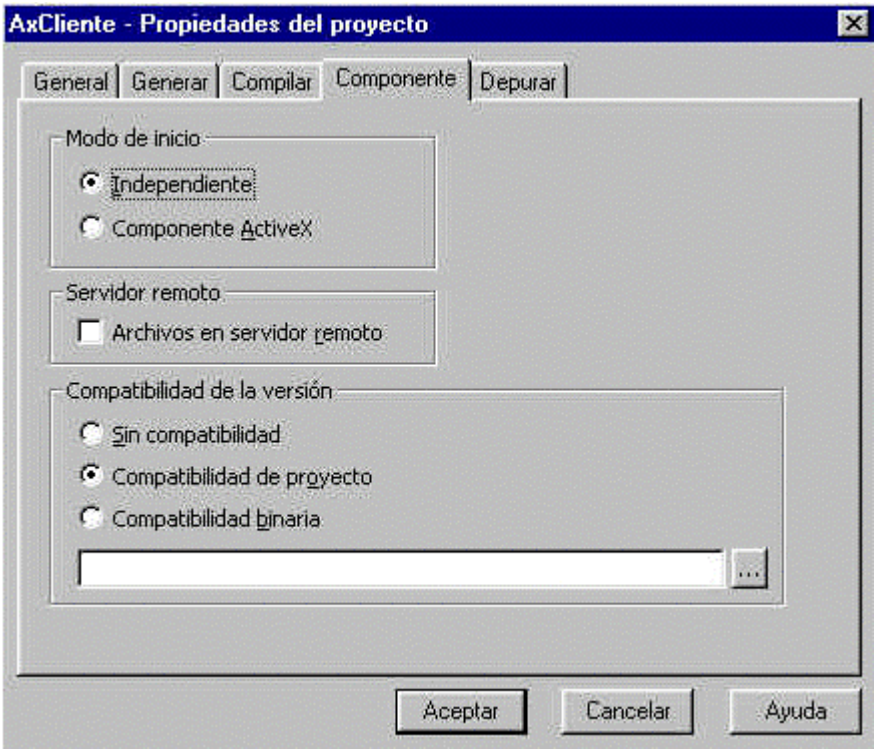


Figura 387. Propiedades de un proyecto para un componente ActiveX.

Si posteriormente necesitamos realizar cambios en el componente: añadir clases, métodos, interfaces, etc., la compatibilidad binaria comprobará que dichos cambios sean compatibles con la versión existente del componente y con todas las aplicaciones cliente que hacen uso de él. Si por algún motivo, el código a añadir es incompatible, seremos notificados de tal situación, siendo conveniente que el código que puede hacer incompatible el componente, sea incluido en una nueva clase o interfaz, para evitar conflictos con el código existente y mantener la compatibilidad.

Desarrollo de un componente ActiveX EXE

Una vez vistas las principales fases en el desarrollo de un componente de código, pasemos a continuación al ejemplo práctico de creación de componente ActiveX EXE incluido en la aplicación *AXCliente*. Para facilitar la tarea del lector de adaptación a los nuevos conceptos sobre componentes, el incluido en este ejemplo se ha diseñado para su uso como ejecutable independiente, que es el tipo de aplicación al que el lector está más acostumbrado hasta el momento.

El objetivo de esta aplicación será crear un mantenimiento de datos para la tabla de clientes, perteneciente a la base de datos de esta aplicación: AxClient.MDB. Siguiendo los pasos descritos

anteriormente, después de insertar y configurar las propiedades del proyecto, se han desarrollado los siguientes elementos para el componente:

Clase Cliente

Ya que el objeto con el que vamos a trabajar es un cliente, lo mejor es diseñar una clase para este elemento, de forma que su manejo se simplifique al utilizarlo como un objeto.

Las propiedades a establecer para un módulo de clase perteneciente a un componente de código ActiveX serán comentadas más adelante. De momento para el caso que nos ocupa, basta saber que la propiedad *Instancing* se ha establecido a *MultiUse*.

Cada vez que instanciemos un objeto de esta clase, se abrirá su base de datos y tabla correspondiente, solicitando al usuario la ruta del fichero de datos en el caso de no encontrarlo, para poder incluirlo en el Registro de Windows.

El código fuente 347 muestra las variables de propiedad definidas para esta clase.

```
' definición de propiedades
' propiedades para los campos del registro
Private mcCodCli As String
Private mcNombre As String
Private mcDireccion As String
Private mcCiudad As String
Private mcNIF As String
' propiedad para el modo de edición del registro
Private mcOperacion As String
' propiedad para buscar registros
Private mcCampoBuscar As String
' propiedades para la base de datos - tabla
Private macnAxClient As ADODB.Connection
Private marsClientes As ADODB.Recordset
Private mcRuta As String
```

Código fuente 347

Para las propiedades que necesiten ser manipuladas desde el exterior de la clase, se definirán los correspondientes procedimientos Property, que el lector puede consultar cargando este proyecto en VB.

En cuanto a los métodos, se pueden dividir en categorías según la función a desempeñar. Disponemos por una parte de los métodos que se encargan de la apertura y configuración de la tabla.

```
Public Function AbrirDatos() As Boolean
' comprobar la ruta de la base
' de datos y abrir la tabla de Clientes
AbrirDatos = False
If Not Me.RutaDatosRegistro("AxClient", "Valores", _
   "RutaDatos", _
   "AxClient.mdb") Then
   Exit Function
End If
' crear conexión
Set macnAxClient = New ADODB.Connection
```

```
macnAxClient.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & Me.Ruta & "AxClient.mdb"
macnAxClient.Open
' crear recordset
Set marsClientes = New ADODB.Recordset
marsClientes.CursorLocation = adUseClient
marsClientes.Open "Clientes", macnAxClient, adOpenDynamic, _
   adLockOptimistic, adCmdTable
Me.Ordenar "PRINCIPAL"
AbrirDatos = True
End Function

' ---------------------------------------

Friend Function RutaDatosRegistro(ByVal vcNombreApp As String, _
   ByVal vcSeccion As String, ByVal vcClave As String, _
   ByVal vcFichDatos As String) As Boolean
' tomar del registro la ruta de
' la base de datos. Si no existe
' información sobre esta aplicación
' en el registro, solicitarla al
' usuario
Dim lcRutaDatos As String
Dim lcFichero As String
Do While True
   ' inicializar valor de retorno
   RutaDatosRegistro = False

   ' obtener del registro la ruta de la base de datos
   lcRutaDatos = GetSetting(vcNombreApp, vcSeccion, vcClave)
   ' si había algún valor en el registro
   ' comprobar si corresponde con una
   ' ruta válida de datos
   If Len(lcRutaDatos) > 0 Then
      lcFichero = Dir(lcRutaDatos & vcFichDatos)
      If Len(lcFichero) = 0 Then
         ' no es válida la ruta del fichero
         lcRutaDatos = ""
      Else
         ' si es válida la ruta del fichero,
         ' asignarla al valor de retorno
         ' del método y finalizarlo
         Me.Ruta = lcRutaDatos
         RutaDatosRegistro = True
         Exit Function
      End If
   End If
   ' si no hay en el registro una entrada
   ' para esta aplicación, crearla
   If Len(lcRutaDatos) = 0 Then
      lcRutaDatos = InputBox("Introducir la ruta del fichero de datos")
      ' si lcRutaDatos no tiene valor
      ' significa que el usuario no ha
      ' proporcionado la ruta de datos
      ' salir del método
      If Len(lcRutaDatos) = 0 Then
         Me.Ruta = ""
         RutaDatosRegistro = False
         Exit Function
      Else
         If Right(lcRutaDatos, 1) <> "\" Then
            lcRutaDatos = lcRutaDatos & "\"
         End If
         ' si lcRutaDatos tiene valor,
         ' crear entrada en el registro
         SaveSetting vcNombreApp, vcSeccion, vcClave, lcRutaDatos
```

```
         lcRutaDatos = ""
         ' ahora volvemos al inicio del bucle
         ' para volver a recuperar el valor
         ' del registro y comprobar si la
         ' ruta de datos es correcta
      End If
   End If
Loop
End Function
```

Código fuente 348

Habrá observado el lector que este último método se ha declarado como Friend. Esto quiere decir que si una aplicación cliente accede a este componente ActiveX EXE, y crea un objeto de la clase Cliente, podrá utilizar todos sus métodos excepto este, ya que al haberse declarado Friend, sólo será accesible por las clases contenidas dentro del componente.

El código fuente 349 hace referencia a la edición de registros.

```
Public Function Buscar(ByVal vcValor As String) As Boolean
' buscar un registro según el valor
' pasado como parámetro al método
Dim lvntMarca As Variant
lvntMarca = marsClientes.Bookmark
Buscar = True
marsClientes.Find Me.CampoBuscar & " = '" & vcValor & "'"
If marsClientes.EOF Then
   marsClientes.Bookmark = lvntMarca
   Buscar = False
End If
End Function
' ----------------------------------------
Public Sub Ordenar(ByVal vcOrden As String)
' establecer el índice para la tabla
Select Case vcOrden
Case "PRINCIPAL"
   marsClientes("CodCli").Properties("Optimize") = True
   marsClientes.Sort = "CodCli ASC"
   Me.CampoBuscar = "CodCli"

Case "NOMBRE"
   marsClientes("Nombre").Properties("Optimize") = True
   marsClientes.Sort = "Nombre ASC"
   Me.CampoBuscar = "Nombre"
End Select
End Sub
' ----------------------------------------
Public Sub Cargar()
' traspasar el contenido del registro
' de la tabla a las propiedades del objeto
Me.CodCli = marsClientes("CodCli")
Me.Nombre = marsClientes("Nombre")
Me.Direccion = marsClientes("Direccion")
Me.Ciudad = marsClientes("Ciudad")
Me.NIF = marsClientes("NIF")
Me.Operacion = "MODIFICAR"
End Sub
' ----------------------------------------
Public Sub Grabar()
' según el tipo de operación del objeto
' agregar o editar el registro, y traspasar
```

```
' el contenido de las propiedades a los
' campos
If Me.Operacion = "INSERTAR" Then
   marsClientes.AddNew
End If
marsClientes("CodCli") = Me.CodCli
marsClientes("Nombre") = Me.Nombre
marsClientes("Direccion") = Me.Direccion
marsClientes("Ciudad") = Me.Ciudad
marsClientes("NIF") = Me.NIF
marsClientes.Update
End Sub
' ----------------------------------------
Public Sub Nuevo()
' preparar el objeto para crear un
' nuevo registro en la tabla
Me.CodCli = ""
Me.Nombre = ""
Me.Direccion = ""
Me.Ciudad = ""
Me.NIF = ""
Me.Operacion = "INSERTAR"
End Sub
```

Código fuente 349

Finalmente tenemos los métodos encargados de la navegación entre los registros de la tabla.

```
Public Sub Avanzar()
' avanzar al siguiente registro de la tabla
marsClientes.MoveNext
If marsClientes.EOF Then
   MsgBox "Último Registro"
   marsClientes.MovePrevious
End If
End Sub

' ----------------------------------------

Public Sub Retroceder()
' retroceder al anterior registro de la tabla
marsClientes.MovePrevious
If marsClientes.BOF Then
   MsgBox "Primer Registro"
   marsClientes.MoveNext
End If
End Sub

' ----------------------------------------

Public Sub IrPrimero()
' situarse en el primer registro de la tabla
marsClientes.MoveFirst
End Sub

' ----------------------------------------

Public Sub IrUltimo()
' situarse en el último registro de la tabla
marsClientes.MoveLast
End Sub
```

Código fuente 350

Formulario frmClientes

Este formulario proporcionará al usuario un interfaz visual mediante el cual podrá realizar la edición de los registros de la tabla de clientes empleando un objeto de la clase Cliente contenido en la variable *moCliente*, declarada a nivel de módulo, que se instanciará al cargar el formulario.

El formulario dispone de controles para editar el contenido de un objeto Cliente, navegar por los clientes de la tabla y buscar un cliente determinado. Observe el lector, que mediante este formulario estamos manipulando un objeto de la clase Cliente, y no los registros de la tabla Clientes. La variable moCliente, que contiene el objeto, es la encargada de realizar la navegación por los registros de la tabla, editar un registro existente, crear uno nuevo, grabarlo en la tabla, etc. El formulario, a diferencia de lo que podamos estar acostumbrados mediante las técnicas habituales de programación, no es el encargado de realizar todo el trabajo directamente con la base de datos. En esta ocasión existe un claro reparto de las tareas según el modelo de tres capas comentado al comienzo de este tema:

1) El formulario se encargará de las reglas de interfaz de usuario.

2) El objeto de la clase Cliente se ocupará de las reglas de negocio.

3) La base de datos se ocupará del tratamiento de los mismos.

Figura 388. Formulario para la edición de un objeto Cliente.

Como podemos comprobar, el empleo de objetos en lugar de procedimientos sin conexión, nos ayuda a conseguir una aplicación mejor diseñada, muestra de ello es el código fuente 351, perteneciente al botón *Grabar* del formulario.

```
Private Sub cmdGrabar_Click()
' llamamos al procedimiento que pasa los valores
' de los controles al objeto cliente
VolcarObj
End Sub
' ------------------------------------
Private Sub VolcarObj()
' volcado del contenido de los controles
' del formulario a las propiedades
```

```
' del objeto cliente
moCliente.CodCli = Me.txtCodCli.Text
moCliente.Nombre = Me.txtNombre.Text
moCliente.Direccion = Me.txtDireccion.Text
moCliente.Ciudad = Me.txtCiudad.Text
moCliente.NIF = Me.txtNIF.Text
' grabar las propiedades del objeto
' a la tabla de la base de datos
moCliente.Grabar
Me.cmdPrincipio.Value = True
End Sub
```

Código fuente 351

Para no repetir todo el código del componente en este texto, el lector puede cargar el proyecto en Visual Basic y consultarlo desde allí.

Procedimiento Main

Al usar este componente como ejecutable, utilizamos este procedimiento como punto de entrada, comprobando el modo de ejecución como hemos visto en la explicación de las etapas de creación de un componente.

```
Public Sub Main()
Dim lfrmClientes As frmClientes
' si el modo de inicio del componente
' es independiente, utilizar el formulario
If App.StartMode = vbSModeStandalone Then
   ' instanciar un objeto de frmClientes
   Set lfrmClientes = New frmClientes
   Load lfrmClientes

   ' si ha habido un problema al cargar
   ' el objeto, salir de la ejecución
   If Not lfrmClientes.InicioOK Then
      Unload lfrmClientes
      Exit Sub
   Else
      lfrmClientes.Show
   End If

End If
End Sub
```

Código fuente 352

Con esto terminamos la creación de código para el componente. Ahora sólo queda que el lector lo ejecute y compruebe su funcionamiento, navegando entre los registros de la tabla y editándolos.

Uso de un componente ActiveX EXE desde una aplicación cliente

Terminado el desarrollo del componente ActiveX EXE y probado su funcionamiento en modo independiente, pasaremos en este apartado a verificar el comportamiento de ese mismo componente cuando es utilizado por una aplicación cliente.

Debido a que un ActiveX EXE se ejecuta en un proceso distinto a la aplicación cliente, la mejor forma de realizar pruebas de ejecución con este tipo de componente es iniciar dos instancias de Visual Basic, ejecutando el componente en una de ellas y la aplicación cliente en la otra. A continuación se detallan los pasos para este proceso:

- Nos situaremos en primer lugar en el componente ActiveX EXE y abriremos la ventana de propiedades del proyecto. Estableceremos la propiedad *Modo de inicio* a *Componente ActiveX*.

- Seguidamente debemos ejecutar el componente. Como su modo de ejecución se ha establecido a componente, se mantendrá en ejecución de forma que podamos hacer pruebas con una aplicación cliente.

- Ahora pasaremos a la otra copia de VB y crearemos un nuevo proyecto de tipo ejecutable estándar, que será el que actúe como cliente. El ejemplo proporcionado para este caso es la aplicación *LlamarAx*, que puede incluirse en el mismo directorio del componente o en otro distinto.

- Después de establecer las propiedades básicas en la aplicación cliente, abriremos la ventana *Referencias* (figura 389) del proyecto mediante la opción *Proyecto+Referencias* del menú de VB, y marcaremos el nombre del componente ActiveX que hemos puesto en ejecución.

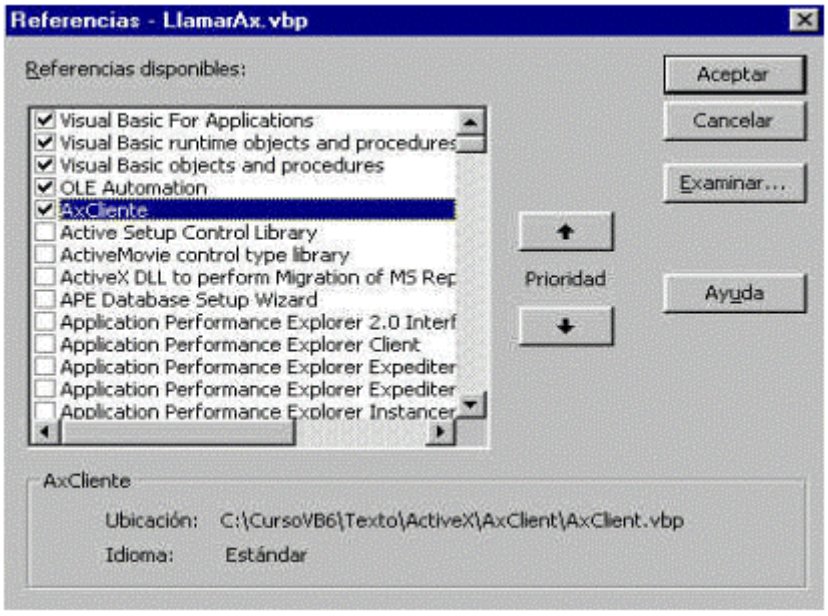


Figura 389. Establecimiento desde la aplicación cliente de la referencia hacia el componente ActiveX.

La ruta del fichero de proyecto que está ejecutando el componente deberá aparecer en la parte inferior de la ventana de referencias.

Una vez establecida la referencia, las clases públicas contenidas en el componente podrán ser visualizadas en el Examinador de Objetos. De esta manera podremos consultar el contenido del componente que vayamos a utilizar desde la aplicación cliente. De igual modo, en la ventana de código aparecerán las referencias a las clases del componente, métodos, propiedades, etc.

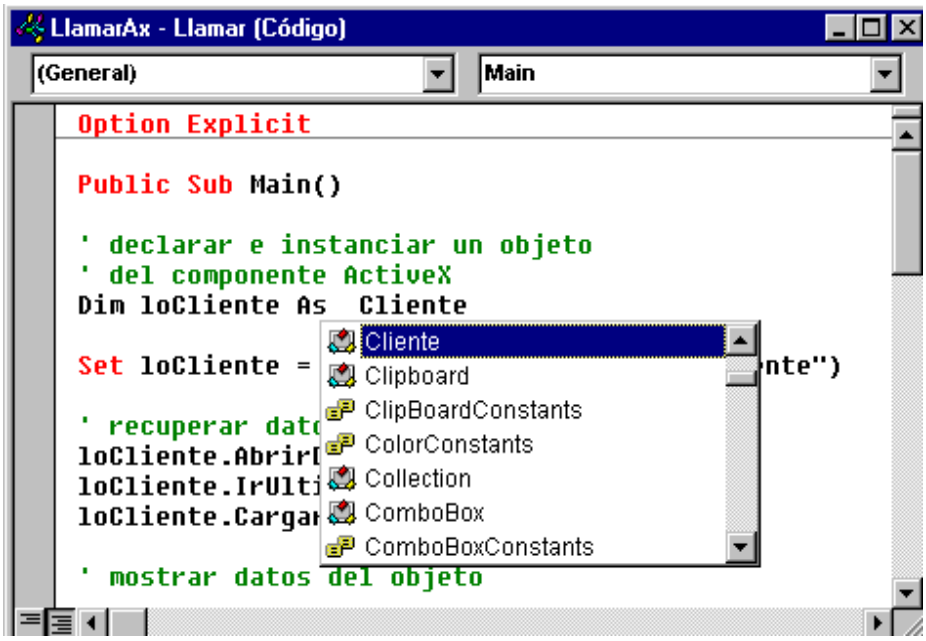


Figura 390. Lista de elementos disponibles para usar en la aplicación, entre los que figura la clase del componente ActiveX.

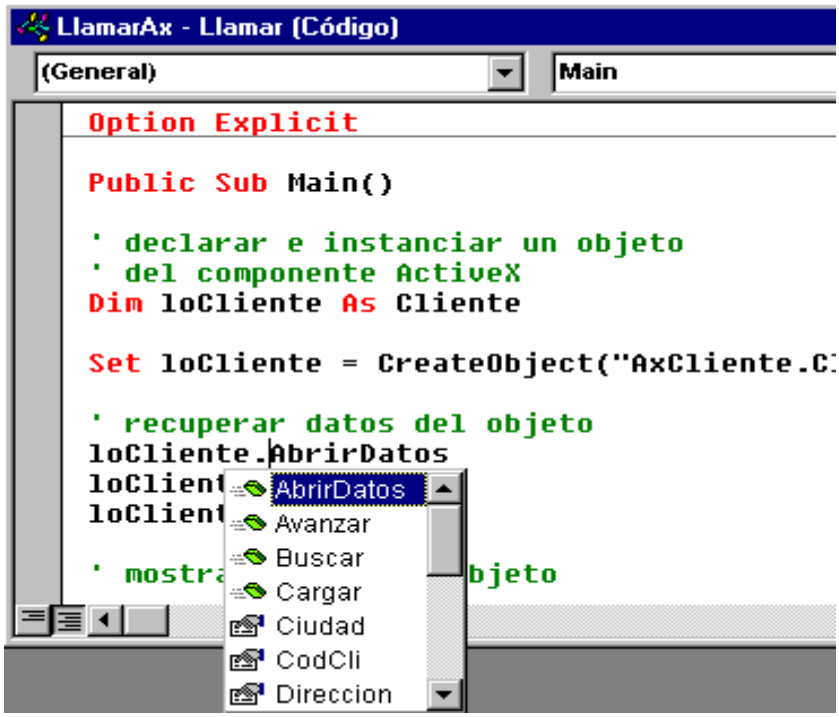


Figura 391. Lista de miembros pertenecientes a la clase del componente ActiveX.

Finalmente, escribiremos el código para esta aplicación cliente y la ejecutaremos comprobando la conexión con el componente ActiveX.

Durante las pruebas con el componente y la aplicación cliente, si hemos realizado algún cambio en el código del ActiveX, se nos podría presentar un problema de conexión entre el cliente y el componente. Para evitarlo, hemos de realizar lo siguiente:

- Compilar el componente.
- Establecer en el componente la propiedad *Compatibilidad de la versión* a *Compatibilidad del proyecto.*
- Incluir el nombre del componente en el TextBox que aparece en la propiedad de compatibilidad.

Si no realizamos las acciones anteriores y se produce un error, deberemos establecer de nuevo la referencia que la aplicación cliente mantiene con el componente.

En el ejemplo de la aplicación cliente LlamarAx partimos del siguiente supuesto: disponemos en el proyecto AxCliente de un componente ActiveX EXE, y necesitamos instanciar del mismo un objeto de la clase Cliente para añadir datos a la tabla de clientes, pero sin usar las características de interfaz de usuario que proporciona el componente. Para ello, crearemos en un módulo de código un procedimiento Main() que se encargue de crear un objeto cliente como se muestra en el código fuente 353.

```
Public Sub Main()
' declarar e instanciar un objeto
' del componente ActiveX
Dim loCliente As Cliente
Set loCliente = CreateObject("AxCliente.Cliente")
' recuperar datos del objeto
loCliente.AbrirDatos
loCliente.IrUltimo
loCliente.Cargar
' mostrar datos del objeto
MostrarObjeto loCliente
' crear desde esta aplicación cliente
' un nuevo objeto cliente y guardarlo
' en la base de datos
loCliente.Nuevo
loCliente.CodCli = "60"
loCliente.Nombre = "Reparaciones La General"
loCliente.Direccion = "Torre 4"
loCliente.Ciudad = "Hamburgo"
loCliente.NIF = "A66887222"
loCliente.Grabar

' visualizar los datos del objeto creado
MostrarObjeto loCliente
' buscar un cliente existente y mostrarlo
If loCliente.Buscar("40") Then
   loCliente.Cargar
   MostrarObjeto loCliente
End If
' eliminar las referencias al objeto
Set loCliente = Nothing
End Sub
```

```
' ------------------------------------------------------------

Public Sub MostrarObjeto(ByVal voCliente As Cliente)
MsgBox "Código: " & voCliente.CodCli & vbCrLf & _
   "Nombre: " & voCliente.Nombre & vbCrLf & _
   "Dirección: " & voCliente.Direccion & vbCrLf & _
   "Ciudad: " & voCliente.Ciudad & vbCrLf & _
   "NIF: " & voCliente.NIF, , "Datos del cliente actual"
End Sub
```

Código fuente 353

Si el lector sigue la ejecución con el depurador, comprobará el salto al código del componente cuando se produce una llamada al mismo.

Hasta aquí hemos visto todo lo referente en cuanto a la creación y ejecución de un componente ActiveX y una aplicación cliente. Existen sin embargo, ciertos aspectos en el desarrollo de componentes en cuanto a configuración y optimización, que es necesario conocer antes de seguir con la creación del siguiente tipo de componente. Tales características se describen en los siguientes apartados.

CreateObject()

El ejemplo anterior utiliza una nueva función: *CreateObject()*, que debemos usar en lugar de New, para instanciar objetos de clases contenidas en componentes ActiveX. Su sintaxis es la siguiente:

```
CreateObject(cClase [, cServidor])
```

- cClase. Cadena que consta de dos partes separadas por un punto. La primera indica el nombre de la aplicación o componente en la que se halla la clase de la que crear un objeto. La segunda indica el nombre de la clase a partir de la cual vamos a instanciar un objeto.

- cServidor. Cadena con el nombre del servidor, en el cuál el objeto será creado; si dicho servidor no está disponible, se producirá un error.

Ejemplo

```
Set loCliente = CreateObject("AxCliente.Cliente")
```

Reducción de la comunicación entre procesos

Al emplear un componente ActiveX EXE, hemos de diseñarlo de forma que al utilizar sus servicios desde una aplicación cliente, el número de veces que la aplicación cliente debe realizar llamadas al componente sea el menor posible. De esta forma se reducirá el número de ocasiones que se debe traspasar el proceso de la aplicación para llegar al proceso del componente.

Si tomamos el procedimiento Main() del ejemplo anterior, al crear un nuevo cliente y asignar valores a sus propiedades, cada asignación a una propiedad supone una llamada al componente, como vemos en el código fuente 354.

```
.........
loCliente.CodCli = "60"
loCliente.Nombre = "Reparaciones La General"
loCliente.Direccion = "Torre 4"
loCliente.Ciudad = "Hamburgo"
loCliente.NIF = "A66887222"
.........
```

Código fuente 354

Si la clase Cliente dispusiera de un método para hacer todas las asignaciones pasando los valores como parámetro, sólo sería necesario traspasar la zona de proceso entre aplicaciones una única vez, como vemos en el código fuente 355.

```
Public Sub AsignaDatos(ByVal vcCodCli As String, ByVal vcNombre As String, ByVal
vcDireccion As String, ByVal vcCiudad As String, ByVal vcNIF As String)
' traspasar el valor de los parámetros
' a las propiedades del objeto
Me.CodCli = vcCodCli
Me.Nombre = vcNombre
Me.Direccion = vcDireccion
Me.Ciudad = vcCiudad
Me.NIF = vcNIF
End Sub
```

Código fuente 355

Por ello es muy importante que al diseñar una clase, sobre todo si va a incluirse en un ActiveX EXE, se tenga este aspecto muy en cuenta, a fin de mejorar el rendimiento tanto del componente como del cliente que se conecta a él.

Mejorar el *guiado* de datos entre procesos

El sistema de transferencia de datos entre los procesos de un componente ActiveX y una aplicación cliente se denomina *guiado* (marshaling). Aunque VB se ocupa de todo este proceso automáticamente, el programador puede ayudar a que este mecanismo funcione mejor empleando unas sencillas técnicas al pasar parámetros a los componentes ActiveX.

Como norma general, cuando pasemos valores de un objeto a un componente, siempre que sea posible pasaremos las propiedades de ese objeto en lugar del objeto completo. Los valores aislados de las propiedades son más fáciles de manejar por el componente. El código fuente 356 pasa como parámetro a un objeto contenido en un componente ActiveX, otro objeto para que desde el código del componente se tomen los valores necesarios de ese objeto. Este modo de trabajo supone una pérdida de rendimiento en la aplicación.

```
Public Sub Main()
' declarar variable para objeto de componente ActiveX
Dim loAxObj As AxObj
' declarar variable para objeto de la aplicación cliente
Dim loFicha As Ficha
' instanciar objetos
Set loAxObj = CreateObject("Comp.AxObj")
```

```
Set loFicha = New Ficha
' dar valores a las propiedades del objeto
' de la aplicación cliente
loFicha.Nombre = "Datos"
loFicha.Telef = "4443322"
' ahora pasamos el objeto Ficha como parámetro en un
' método del objeto AxObj
loAxObj.TomarValores loFicha
End Sub
' ------------------------------------------------
' método en el código del componente que recibe el objeto
Public Sub TomarValores(ByVal vFicha As Object)
Me.Descrip = vFicha.Nombre
Me.NumTelf = vFicha.Telef
End Sub
```

Código fuente 356

Si por el contrario, pasamos los valores de las propiedades del objeto creado en la aplicación cliente, a las propiedades del objeto que está en el componente, no se producirá el mencionado problema de pérdida en el rendimiento. Esta técnica se muestra en el código fuente 357.

```
Public Sub Main()
' declarar variable para objeto de componente ActiveX
Dim loAxObj As AxObj
' declarar variable para objeto de la aplicación cliente
Dim loFicha As Ficha
' instanciar objetos
Set loAxObj = CreateObject("Comp.AxObj")
Set loFicha = New Ficha
' dar valores a las propiedades del objeto
' de la aplicación cliente
loFicha.Nombre = "Datos"
loFicha.Telef = "4443322"
' ahora pasamos a las propiedades del objeto AxObj
' las propiedades correspondientes del objeto Ficha
loAxObj.Descrip = loFicha.Nombre
loAxObj.NumTelf = loFicha.Telef
End Sub
```

Código fuente 357

En lo que se refiere a los componentes ActiveX DLL, las normas generales de paso de parámetros se pueden aplicar aquí para mejorar el sistema de guiado: pasar los parámetros por valor (ByVal) excepto cuando se trate de arrays de tipo Variant y cadenas grandes de caracteres, en cuyo caso lo haremos por referencia (ByRef).

Para los componentes ActiveX EXE hemos de pasar los parámetros siempre por valor excepto cuando el componente deba cambiar el contenido del parámetro, en este caso lo haremos por referencia. El paso por referencia sin embargo, tiene un gran problema en este tipo de componente, ya que el parámetro debe ser copiado en primer lugar al espacio de direcciones del componente, y después el puntero a la copia local de los datos se envía al componente. Finalizada la operación, el parámetro se vuelve a copiar al espacio de direcciones de la aplicación cliente. Esta última copia de retorno es la que podemos ahorrar si utilizamos el paso por valor siempre que sea posible.

La Propiedad Instancing

Las clases desarrolladas en componentes ActiveX disponen de dos propiedades: Name e Instancing. La primera ya la conocemos, sirve para dar un nombre a la clase. En cuanto a Instancing (creación de instancias), su propósito es establecer el modo de acceso a las clases del componente por parte de una aplicación cliente. Los diferentes valores para esta propiedad, se detallan en la figura 392.

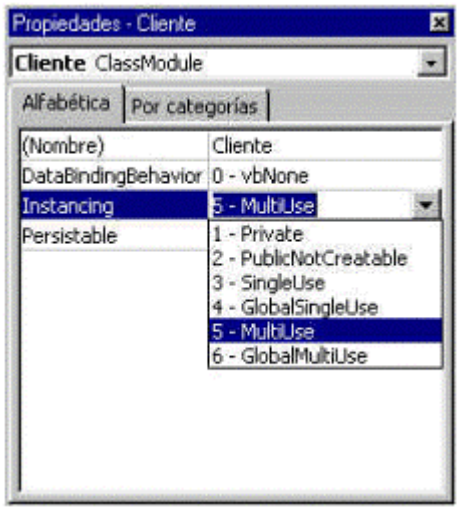


Figura 392. Valores disponibles para la propiedad Instancing.

Private

Las aplicaciones cliente no podrán acceder de ninguna forma a este tipo de clase. Una clase Private se suele utilizar dentro de un componente, para proporcionar servicios a otras clases del componente.

PublicNotCreatable

Los objetos creados con este valor de instancia sólo pueden ser creados por el componente, pero pueden ser usados tanto por el componente como por una aplicación cliente. A este tipo de objetos se les conoce como *objetos dependientes*, y aunque una aplicación cliente no pueda instanciarlos, si puede ver sus interfaces mediante el Examinador de objetos, declarar variables del tipo de objeto dependiente y manejar objetos de este tipo ya instanciados por el componente.

Supongamos un componente que contiene una clase Agenda y una clase Ficha. La clase Agenda se ha definido como MultiUse, y la clase Ficha como PublicNotCreatable. Los objetos agenda pueden crear dentro del código de sus métodos uno o más objetos ficha. Una aplicación cliente no puede crear instancias de la clase Ficha, debido al valor que esta tiene en su propiedad Instancing. Sin embargo hay un medio de que el cliente pueda manejar este tipo de objetos: instanciar un objeto de la clase Agenda y a través de él, tomar el objeto ficha que contiene, pasándolo a una variable declarada de tipo Ficha; a partir de ese momento podrá usarlo como un objeto más.

SingleUse

Las aplicaciones cliente podrán instanciar objetos de una clase contenida en el componente. Pero cada nuevo objeto creado iniciará una nueva instancia del componente. Este valor sólo está disponible para componentes ActiveX EXE.

Los objetos creados con este tipo de instancia dentro del componente tienen un comportamiento especial. Si se emplea la instrucción New para crearlos, se ignorará este valor de instancia y los nuevos objetos serán todos creados en la misma instancia del componente. Sin embargo si se usa la función CreateObject(), los nuevos objetos serán creados en instancias distintas del componente.

GlobalSingleUse

Es similar a SingleUse excepto en que las llamadas a propiedades y métodos de la clase se pueden hacer como si fueran funciones globales, la instancia de la clase se creará automáticamente sin necesidad de hacerlo en el código. El inconveniente de este modo es que se pierde legibilidad en el código. Sólo está disponible para componentes ActiveX EXE.

MultiUse

Permite con una sola instancia del componente, proporcionar objetos a las aplicaciones cliente que lo demanden.

Si se trata de un ActiveX EXE, que se ejecuta en su propio proceso, este componente proporcionará múltiples objetos a las diferentes aplicaciones cliente que se ejecutan fuera de su proceso, y que demanden sus servicios.

Si se trata de un ActiveX DLL, que se ejecuta en el mismo proceso que el cliente, este componente proporcionará múltiples objetos a su aplicación cliente.

Este tipo de instancia es más recomendable en cuanto al uso de recursos del sistema, ya que una sola instancia del componente proporciona múltiples objetos.

GlobalMultiUse

Es similar a MultiUse excepto en que las llamadas a propiedades y métodos de la clase se pueden hacer como si fueran funciones globales. El inconveniente de este modo al igual que en GlobalSingleUse es que se pierde legibilidad en el código

Programación multihebra

Las aplicaciones en Visual Basic se ejecutan por lo general en una única hebra o subproceso. Cuando se desarrollan componentes, estos pueden definir más de una hebra de ejecución, de forma que se puedan ejecutar simultáneamente varios objetos.

El modelo de hebra utilizado en VB es el de *apartamento*. Cada hebra puede considerarse como un apartamento en donde se ejecutan los objetos definidos en su interior, sin tener conocimiento del resto

de objetos que se encuentran en los demás apartamentos o hebras. Los datos definidos como globales en módulos estándar del componente no se comparten, ya que existe una copia de esta información global para cada hebra.

Los componentes ActiveX DLL, al ejecutarse en el mismo proceso que la aplicación cliente, están restringidos a las hebras creadas por el cliente, por lo cual no pueden crear hebras por sí mismos. Ello hace que para este tipo de componente, las capacidades multihebra estén deshabilitadas.

Los componentes ActiveX EXE son los que proporcionan capacidades multihebra. Para ello, se debe marcar la propiedad *Ejecución desatendida* en la ventana de propiedades del componente, teniendo en cuenta que esta acción hará que todos los elementos con interfaz de usuario que tenga el componente: formularios, mensajes, etc., sean deshabilitados.

Una vez seleccionada esta propiedad, la gestión de hebras para el componente se puede realizar de una de las siguientes formas:

- **Subproceso (hebra) por objeto.** Cada objeto del componente se creará en una nueva hebra. Cada una de estas hebras tendrá su propia copia de los datos globales del componente.

- **Conjunto de *n* subprocesos (hebras).** Cada objeto del componente se creará en una hebra de un número predefinido de hebras posibles. Si se crean más objetos que hebras, habrá objetos que compartirán alguna de las hebras definidas en el conjunto. Los datos globales del componente son compartidos por todas las hebras del conjunto. Esta es la opción predeterminada en VB, ya que proporciona una mayor fiabilidad en componentes de versiones anteriores.

Desarrollo de un componente ActiveX DLL

Después de comentar algunos aspectos importantes para la creación de componentes de código, pasemos ahora a desarrollar un componente de código ActiveX DLL.

Como hemos comentado anteriormente, este tipo de componente proporciona objetos a una aplicación cliente, ejecutándose en el mismo proceso que esta última, por lo que el rendimiento de ejecución es mayor. Los pasos para la creación de este tipo de componente ya han sido comentados con anterioridad en el apartado *Creación de componentes ActiveX DLL y EXE* en este mismo tema, por lo que sugerimos al lector que repase dicho apartado en lo que a este particular se refiere. Aquí vamos a centrar nuestra atención en el desarrollo del código, el uso de interfaces y la generación de eventos desde el componente.

Situémonos en el problema a resolver: debemos confeccionar un pequeño programa de facturas, grabando los datos de las facturas y clientes en las tablas del mismo nombre que se encuentran en la base de datos AxFact.MDB. Crearemos una clase para el mantenimiento y manejo de los datos de cada tabla, así como un formulario que se ocupe del aspecto visual del manejo de los datos de cara al usuario. Como punto de partida de la aplicación, usaremos un formulario MDI desde el cual se llevarán a cabo todos los procesos.

En lo que respecta a las clases, se desarrollarán en un componente ActiveX DLL al que llamaremos *AxMant*. La parte que se ocupa del interfaz de usuario se creará en un proyecto estándar llamado *AxClient*, que actuará como aplicación cliente del componente. Todo ello se integrará en un grupo de proyectos denominado *AxGrupo*, que nos va a facilitar la parte referente a las pruebas durante el desarrollo del proyecto y que se incluye como ejemplo para que el lector pueda seguir su creación.

Diseño de las clases

Como dijimos antes, las clases que forman el componente están incluidas en el proyecto *AxMant*. Para la gestión de los clientes, hemos desarrollado la clase Cliente, que se basa en la misma clase utilizada para el ejemplo con el componente ActiveX EXE, pero con algunas variaciones. Esta clase se ocupará de la apertura de la base de datos y tabla de clientes, así como de la edición y navegación en los registros de dicha tabla. En lo que respecta al manejo de facturas, crearemos la clase Factura, que deberá proporcionar una funcionalidad similar a los clientes en cuanto a la edición y navegación de los registros

Puesto que ambas clases deben realizar operaciones similares de mantenimiento y consulta de datos, vamos a crear los interfaces necesarios que agrupen cada tipo de operación. Para detalles sobre la creación de interfaces, consulte el lector el apartado *Interfaces*, del tema *Programación Orientada a Objeto*.

Por un lado crearemos un interfaz que aglutine todas las tareas de edición de los datos y al que llamaremos IEditar. La tabla 90 muestra los elementos que forman el interfaz y una descripción de cada uno.

Elemento	Tipo	Descripción
Operación	Propiedad	Contiene una cadena que identifica si se está insertando un nuevo registro o modificando uno existente
AbrirDatos	Método	Abre la base de datos, tabla correspondiente a la clase y crea un Recordset para manejar los registros
Buscar	"	Buscar un registro en función de un valor pasado como parámetro
Cargar	"	Traspasa el valor de los campos del registro actual a las propiedades del objeto
Grabar	"	Según el tipo de operación, crea un nuevo registro o modifica el actual y pasa el contenido de las propiedades del objeto al registro
Nuevo	"	Inicializa las propiedades del objeto y lo prepara para crear un nuevo registro

Tabla 90. Elementos que componen el interfaz IEditar.

En lo que respecta al desplazamiento por los registros de la tabla, se creará el interfaz INavegar, cuya composición vemos en la tabla 91.

Elemento	Tipo	Descripción
Avanzar	Método	Se desplaza al siguiente registro de la tabla

Retroceder	"	Se desplaza al anterior registro de la tabla
IrPrimero	"	Se desplaza al primer registro de la tabla
IrUltimo	"	Se desplaza al último registro de la tabla

Tabla 91. Elementos que componen el interfaz INavegar.

Como ya sabemos, los módulos donde se han definido los interfaces no contienen código, sólo la definición de métodos y propiedades. Lo que debemos hacer para utilizar estos interfaces es implementarlos en las clases que vayan a usarlos, escribiendo el código fuente 358 en la zona de declaraciones de cada clase.

```
' implementar interfaces
Implements IEditar
Implements Inavegar
```

Código fuente 358

Con esto aparecerán los interfaces en la lista de objetos de la ventana de código de la clase. El siguiente paso consistirá en añadir el código necesario para los procedimientos de cada interfaz en cada una de las clases. El código fuente 359 el método *Grabar()*, del interfaz IEditar, desarrollado para la clase Cliente y Factura, el resto del código puede consultarlo el lector cargando el proyecto en VB.

```
' clase Cliente
Private Sub IEditar_Grabar()
If Me.Operacion = "INSERTAR" Then
   marsClientes.AddNew
End If
marsClientes("CodCli") = Me.CodCli
marsClientes("Nombre") = Me.Nombre
marsClientes("Direccion") = Me.Direccion
marsClientes("Ciudad") = Me.Ciudad
marsClientes("NIF") = Me.NIF
marsClientes.Update
End Sub
' ----------------------------
' clase Factura
Private Sub IEditar_Grabar()

If Me.Operacion = "INSERTAR" Then
   marsFacturas.AddNew
End If

marsFacturas("Numero") = Me.Numero
marsFacturas("Fecha") = Me.Fecha
marsFacturas("CodCli") = Me.CodCli
marsFacturas("Articulo") = Me.Articulo
marsFacturas("Importe") = Me.Importe

marsFacturas.Update

End Sub
```

Código fuente 359

Una vez desarrollado en las clases el código correspondiente a los procedimientos de cada interfaz implementado en dichas clases, podríamos comenzar a utilizar ya estas clases en una aplicación cliente, pero con la incomodidad de tener que estar trabajando al mismo tiempo con variables de la clase y del interfaz implementado para poder acceder a dicho interfaz.

Para evitar este problema, vamos a aplicar en el código de las clases la técnica explicada en el punto *Herencia sin utilizar interfaces* del apartado *Interfaces*, incluido en el tema dedicado a la programación con objetos. En este punto solucionábamos el inconveniente de declarar variables de interfaz, creando procedimientos públicos dentro de la clase, que proporcionaran acceso desde el exterior de la clase a los procedimientos privados del interfaz. Aquí emplearemos esa misma técnica, con lo que la aplicación cliente sólo tendrá que declarar una variable para la clase, sin preocuparse de cuantos interfaces haya implementados en su interior. El código fuente 360 muestra el empleo de este sistema para el método Grabar() de cada clase.

```
' clase Cliente
Public Sub Grabar()
IEditar_Grabar
End Sub
' ----------------------------
' clase Factura
Public Sub Grabar()
IEditar_Grabar
End Sub
```

Código fuente 360

En cuanto a los métodos de desplazamiento de registros, necesitamos informar si cuando se produce un avance o retroceso de registro en la tabla, se ha llegado al final o principio de la misma respectivamente. Lo mas apropiado en este aspecto, es definir un evento para cada una de las situaciones, generándolo cuando se produzcan. El código fuente 361 muestra el código desarrollado para resolver este problema en la clase Cliente.

```
' declaración de eventos en la
' sección de declaraciones del módulo de clase
Public Event PrimerRegistro()
Public Event UltimoRegistro()
' ----------------------------
Private Sub INavegar_Avanzar()

marsClientes.MoveNext

If marsClientes.EOF Then
   marsClientes.MovePrevious
   RaiseEvent UltimoRegistro
End If

End Sub
' ----------------------------
Private Sub INavegar_Retroceder()

marsClientes.MovePrevious

If marsClientes.BOF Then
   marsClientes.MoveNext
```

```
   RaiseEvent PrimerRegistro
End If

End Sub
```

Código fuente 361

En lo que concierne a la clase Factura, se aplica el mismo código para resolver esta situación. La forma de tratar este evento se verá en la aplicación cliente que hace uso del componente.

Finalmente tenemos la clase *RutaDatos* que será utilizada por las clases Cliente y Factura para establecer/obtener la ruta de la base de datos del Registro de Windows. El trabajo de esta clase ya estaba implementado en el ejemplo sobre ActiveX EXE, pero debido a que las operaciones de manipulación del Registro son necesarias para las dos clases principales del componente actual, se obtiene una mayor organización del código creando una clase aparte para esta tarea.

El método *AbrirDatos()* del interfaz IEditar es el utilizado para crear un objeto de la clase RutaDatos que tome la clave del registro correspondiente a la aplicación. En el código fuente 362 vemos la apertura de datos para la clase Cliente y el método *RutaDatosRegistro()* de la clase RutaDatos que toma o asigna el valor del registro.

```
' clase Cliente
Private Function IEditar_AbrirDatos() As Boolean

Dim lcRuta As String
Dim loRutaDatos As RutaDatos

Set loRutaDatos = New RutaDatos

IEditar_AbrirDatos = False

If Not loRutaDatos.RutaDatosRegistro("AxMant", "Valores", _
   "RutaDatos", _
   "AxFact.mdb") Then

   Exit Function
End If

lcRuta = loRutaDatos.Ruta

' crear conexión
Set macnAxFact = New ADODB.Connection
macnAxFact.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & lcRuta & "AxFact.mdb"
macnAxFact.Open

' crear recordset
Set marsClientes = New ADODB.Recordset
marsClientes.CursorLocation = adUseClient
marsClientes.Open "Clientes", macnAxFact, adOpenDynamic, _
   adLockOptimistic, adCmdTable

' establecer orden
marsClientes("CodCli").Properties("Optimize") = True
marsClientes.Sort = "CodCli ASC"

IEditar_AbrirDatos = True

End Function
```

```
' ---------------------------------------------
' clase RutaDatos
Public Function RutaDatosRegistro(ByVal vcNombreApp As String, _
   ByVal vcSeccion As String, ByVal vcClave As String, _
   ByVal vcFichDatos As String) As Boolean

Dim lcRutaDatos As String
Dim lcFichero As String
Do While True
   ' inicializar valor de retorno
   RutaDatosRegistro = False

   ' obtener del registro la ruta de la base de datos
   lcRutaDatos = GetSetting(vcNombreApp, vcSeccion, vcClave)

   ' si había algún valor en el registro
   ' comprobar si corresponde con una
   ' ruta válida de datos
   If Len(lcRutaDatos) > 0 Then
      lcFichero = Dir(lcRutaDatos & vcFichDatos)

      If Len(lcFichero) = 0 Then
         ' no es válida la ruta del fichero
         lcRutaDatos = ""
      Else
         ' si es válida la ruta del fichero,
         ' asignarla al valor de retorno
         ' de la función y finalizar función
         Me.Ruta = lcRutaDatos
         RutaDatosRegistro = True
         Exit Function
      End If
   End If

   ' si no hay en el registro una entrada
   ' para esta aplicación, crearla
   If Len(lcRutaDatos) = 0 Then
      lcRutaDatos = InputBox("Introducir la ruta del fichero de datos")

      ' si lcRutaDatos no tiene valor
      ' significa que el usuario no ha
      ' proporcionado la ruta de datos
      ' salir de la función
      If Len(lcRutaDatos) = 0 Then
         Me.Ruta = ""
         RutaDatosRegistro = False
         Exit Function
      Else

         If Right(lcRutaDatos, 1) <> "\" Then
            lcRutaDatos = lcRutaDatos & "\"
         End If

         ' si lcRutaDatos tiene valor,
         ' crear entrada en el registro
         SaveSetting vcNombreApp, vcSeccion, vcClave, lcRutaDatos
         lcRutaDatos = ""

         ' ahora volvemos al inicio del bucle
         ' para volver a recuperar el valor
         ' del registro y comprobar si la
         ' ruta de datos es correcta

      End If
   End If
```

```
Loop
End Function
```

Código fuente 362

La propiedad Instancing de la clase RutaDatos se ha declarado como Private, ya que sólo las clases del componente van a instanciar objetos de tipo RutaDatos. Para el resto de las clases de este componente dicha propiedad será MultiUse, de forma que estén accesibles a las aplicaciones cliente.

Con esto podemos dar por terminado la creación de las clases para el componente ActiveX DLL AxMant, cuyo proyecto tendrá el aspecto de la figura 393.

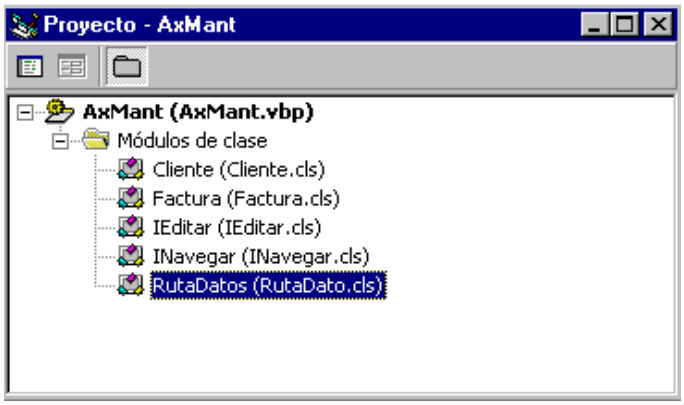


Figura 393. Ventana de proyecto del componente AxMant.

Trabajar con un grupo de proyectos

Finalizado el proyecto que contiene el componente, necesitamos desarrollar una aplicación cliente que pruebe las clases contenidas en el componente. Cuando estamos trabajando con ActiveX DLL, el aspecto de las pruebas se simplifica bastante, puesto que la actual versión de Visual Basic permite trabajar con varios proyectos simultáneamente, formando lo que se denomina un *grupo de proyectos*. De esta manera usaremos la opción de menú *Archivo+Agregar proyecto...*, para incorporar un proyecto de tipo estándar al proyecto existente ActiveX DLL. Al guardar el grupo de proyectos se hará en un fichero VBG.

Figura 394. Grupo de proyectos para realizar pruebas con un componente ActiveX DLL.

Una vez incorporado el nuevo proyecto, después de asignarle un nombre e indicar que su inicio se realizará mediante un procedimiento Main(), seleccionaremos dicho proyecto y abriendo su menú contextual, seleccionaremos la opción *Establecer como inicial*, con lo que el nombre del proyecto cambiará a negrita. Esta acción indicará a VB que la ejecución del grupo de proyectos comenzará por el indicado como inicial.

La figura 394 muestra el Explorador de proyectos después de haber incluido el proyecto *AxClient*, que se ejecutará como aplicación cliente del ActiveX DLL.

Desarrollo de la aplicación cliente para un componente ActiveX DLL

En primer lugar hemos de establecer desde el proyecto cliente la referencia hacia el componente ActiveX DLL (figura 395), de forma que la aplicación cliente tenga acceso a las clases del componente. Esta tarea se realiza de la misma forma que vimos en el ejemplo con ActiveX EXE.

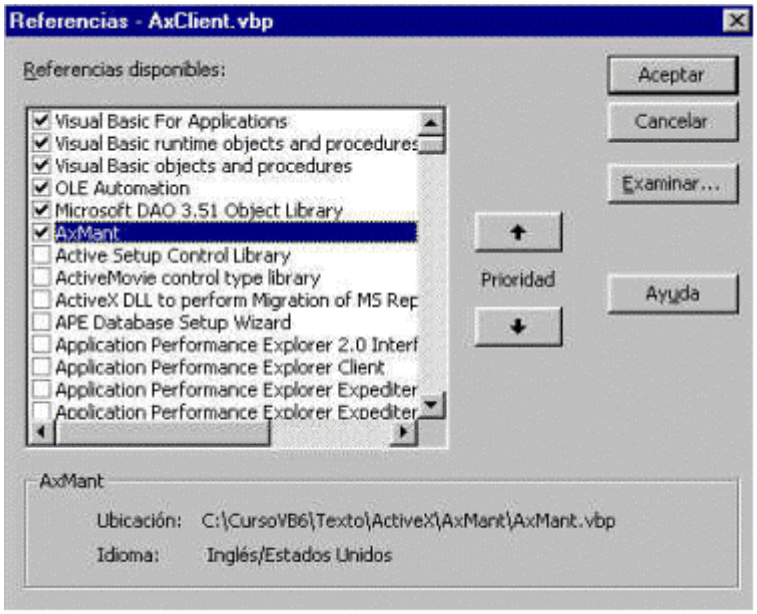


Figura 395. Estableciendo desde la aplicación cliente, la referencia hacia el componente ActiveX DLL.

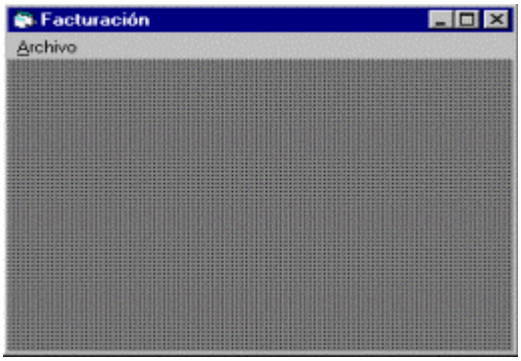


Figura 396. Formulario mdiPrincipal de la aplicación cliente.

En la figura 396 vamos a crear los formularios que servirán al usuario para comunicarse con la aplicación. Empezaremos por la ventana *mdiPrincipal*, definida como MDI, y que servirá de base al resto de ventanas del proyecto, que se definirán como secundarias.

El código de este formulario corresponde a las diferentes opciones de su menú, como vemos en el código fuente 363.

```
Private Sub mnuArClientes_Click()
' declarar una variable del formulario
' a instanciar
Dim lfrmClientes As frmClientes
' instanciar el formulario y cargarlo
Set lfrmClientes = New frmClientes
Load lfrmClientes
If lfrmClientes.InicioOK Then
   ' deshabilitar esta opción de menú para
   ' no permitir al usuario crear múltiples
   ' copias del mismo formulario
   Me.mnuArClientes.Enabled = False
```

```
   lfrmClientes.Show
Else
   ' si hay errores al cargar el formulario
   ' descargarlo para poder liberar las referencias
   ' establecidas a los objetos del componente ActiveX DLL
   Unload lfrmClientes
End If
End Sub

' --------------------------------

Private Sub mnuArFacturas_Click()
' declarar una variable del formulario
' a instanciar
Dim lfrmFacturas As frmFacturas
' instanciar el formulario y cargarlo
Set lfrmFacturas = New frmFacturas
Load lfrmFacturas
If lfrmFacturas.InicioOK Then
   ' deshabilitar esta opción de menú para
   ' no permitir al usuario crear múltiples
   ' copias del mismo formulario
   Me.mnuArFacturas.Enabled = False
   lfrmFacturas.Show
Else
   ' si hay errores al cargar el formulario
   ' descargarlo para poder liberar las referencias
   ' establecidas a los objetos del componente ActiveX DLL
   Unload lfrmFacturas
End If
End Sub

' --------------------------------

Private Sub mnuArSalir_Click()
Unload Me
End Sub
```

Código fuente 363

Para el mantenimiento de los clientes, crearemos el formulario *frmClientes*, que vemos en la figura 397.

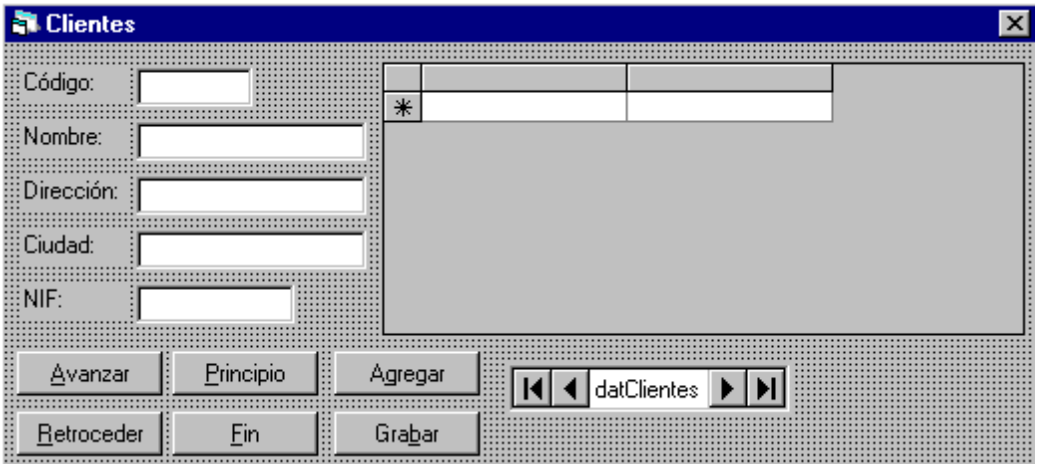


Figura 397. Formulario frmClientes de la aplicación cliente.

Aunque la base del mantenimiento de la tabla es la misma que en el ejemplo con ActiveX EXE, en este formulario se ha introducido un control DBGrid para poder ver los registros en conjunto. El código fuente 364 muestra el código desarrollado para este formulario con las explicaciones oportunas para cada acción.

```
' ------- sección declaraciones ----------------
' declarar una variable a nivel de módulo de la
' clase Cliente. Al usar WithEvents, esta clase
' será notificada cuando se produzcan los eventos
' en el objeto moCliente
Private WithEvents moCliente As Cliente
' esta variable indicará si se ha
' cargado el formulario sin errores
Private mbInicioOK As Boolean
' ----------------------------------------
' definición de procedimientos de propiedad
' para comprobar la carga del formulario
Public Property Get InicioOK() As Boolean
InicioOK = mbInicioOK
End Property
Public Property Let InicioOK(ByVal vbInicioOK As Boolean)
mbInicioOK = vbInicioOK
End Property
' ----------------------------------------
Private Sub CargarObj()
' traspasar el contenido de las propiedades
' del objeto moCliente a los controles
' del formulario
Me.txtCodCli.Text = moCliente.CodCli
Me.txtNombre.Text = moCliente.Nombre
Me.txtDireccion.Text = moCliente.Direccion
Me.txtCiudad.Text = moCliente.Ciudad
Me.txtNIF.Text = moCliente.NIF
End Sub
' ----------------------------------------
Private Sub VolcarObj()
' traspasar el contenido de los controles
' del formulario a las propiedades
' del objeto moCliente
moCliente.CodCli = Me.txtCodCli.Text
moCliente.Nombre = Me.txtNombre.Text
moCliente.Direccion = Me.txtDireccion.Text
moCliente.Ciudad = Me.txtCiudad.Text
moCliente.NIF = Me.txtNIF.Text
' grabar el objeto a la tabla
moCliente.Grabar
' situarse en el primer registro
Me.cmdPrincipio.Value = True
End Sub
' ----------------------------------------
Private Sub cmdAgregar_Click()
' preparar el formulario para
' editar un nuevo objeto cliente
moCliente.Nuevo
CargarObj
End Sub
' ----------------------------------------
Private Sub cmdAvanzar_Click()
' avanzar al siguiente registro de la tabla
moCliente.Avanzar
' cargar el registro en las propiedades
' del objeto
moCliente.Cargar
```

```
' cargar el objeto en los controles
' del formulario
CargarObj
End Sub
' ----------------------------------------
Private Sub cmdFin_Click()
' posicionarse en el último registro
' de la tabla
moCliente.IrUltimo
' cargar el registro en las propiedades
' del objeto
moCliente.Cargar
' cargar el objeto en los controles
' del formulario
CargarObj
End Sub
' ----------------------------------------
Private Sub cmdGrabar_Click()
VolcarObj
End Sub
' ----------------------------------------
Private Sub cmdPrincipio_Click()
' posicionarse en el primer registro
' de la tabla
moCliente.IrPrimero
' cargar el registro en las propiedades
' del objeto
moCliente.Cargar
' cargar el objeto en los controles
' del formulario
CargarObj
End Sub
' ----------------------------------------
Private Sub cmdRetroceder_Click()
' retroceder al anterior registro de la tabla
moCliente.Retroceder
' cargar el registro en las propiedades
' del objeto
moCliente.Cargar
' cargar el objeto en los controles
' del formulario
CargarObj
End Sub
' ----------------------------------------
Private Sub Form_Load()
' cargar el formulario
Me.InicioOK = False
' crear un objeto Cliente del componente ActiveX DLL
Set moCliente = CreateObject("AxMant.Cliente")
' si hay error al crear el objeto cliente, salir
If Not moCliente.AbrirDatos Then
   MsgBox "Error al abrir los datos del objeto Cliente"
   Exit Sub
End If
' establecer estado de la carga del formulario
Me.InicioOK = True
' situarse en el primer registro
Me.cmdPrincipio.Value = True
' el objeto cliente dispone de la propiedad Lista,
' que no es otra cosa que el Recordset que contiene
' la tabla de clientes. La pasaremos al control Data
' del formulario para que pueda verse en el DBGrid
Set Me.datClientes.Recordset = moCliente.Lista
End Sub
' ----------------------------------------
Private Sub Form_Unload(Cancel As Integer)
```

```
' eliminar la referencia al objeto
' del componente para no consumir
' recursos innecesarios
Set moCliente = Nothing
' habilitar la opción de menú en la ventana MDI
gmdiPrincipal.mnuArClientes.Enabled = True
End Sub
' ----------------------------------------
Private Sub moCliente_PrimerRegistro()
' este procedimiento corresponde al evento
' PrimerRegistro del objeto cliente
' y se ejecutará cada vez que el objeto lo
' genere
MsgBox "Primer cliente", , "Aviso"
End Sub
' ----------------------------------------
Private Sub moCliente_UltimoRegistro()
' este procedimiento corresponde al evento
' UltimoRegistro del objeto cliente
' y se ejecutará cada vez que el objeto lo
' genere
MsgBox "Último cliente", , "Aviso"
End Sub
```

Código fuente 364

Como habrá podido observar el lector, al descargar el formulario se elimina la referencia que contiene el objeto instanciado desde el componente ActiveX DLL. Este aspecto es muy importante, siempre que utilicemos objetos de componentes ActiveX, hemos de liberarlos cuando ya no sean necesarios, para evitar el gasto excesivo de recursos del sistema. Un componente ActiveX finaliza su ejecución cuando no existe en la aplicación cliente ninguna referencia hacia los objetos del componente.

En cuanto al mantenimiento de facturas, crearemos el formulario *frmFacturas*, veámoslo en la figura 398.

Como la mecánica de edición de registros es igual en este formulario que en el anterior de clientes, obviaremos repetir el código que tenga la misma funcionalidad en ambos formularios, limitándonos a lo que realmente aporta esta ventana de novedoso.

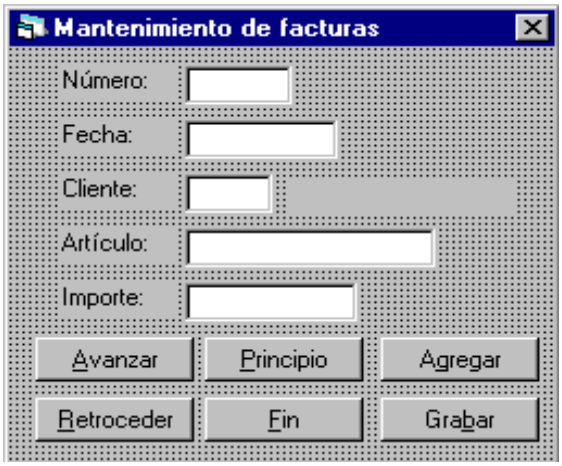


Figura 398. Formulario frmFacturas de la aplicación cliente.

```
Private Sub BuscarCliente()
' buscar un objeto cliente en su tabla
' en función del valor del control de
' esta ventana que contiene el código
' de cliente
moCliente.IrPrimero
If moCliente.Buscar(Me.txtCodCli.Text) Then
   ' si encontramos el cliente, lo cargamos
   ' a sus propiedades
   moCliente.Cargar

   ' traspasamos el nombre del cliente
   ' al control label que muestra dicho nombre
   Me.lblNombreCliente.Caption = moCliente.Nombre
End If

End Sub
' ---------------------------------------
Private Sub txtCodCli_LostFocus()
' cuando salgamos del control
' txtCodCli, buscar el cliente
' según el valor del control
BuscarCliente
End Sub
```

Código fuente 365

Los campos de fecha e importe de la factura no efectúan ninguna comprobación del valor, ya que este aspecto no es el objetivo a resolver en esta aplicación, por lo que el usuario deberá proporcionar los valores correctos para esos campos o se producirá un error en ejecución.

Finalmente, crearemos un procedimiento Main() de entrada a la aplicación, que guardaremos en el módulo estándar GENERAL.BAS, y que tendrá el código fuente 366.

```
Public Sub Main()
Set gmdiPrincipal = New mdiPrincipal
gmdiPrincipal.Show
End Sub
```

Código fuente 366

La variable gmdiPrincipal es de ámbito global y contiene el formulario MDI de esta aplicación.

Finalizado el desarrollo de código para la aplicación cliente, pasaremos a ejecutar el grupo de proyectos. El usuario podrá abrir las ventanas de mantenimiento de las tablas, consultar y editar registros. Todo este desarrollo se ha realizado bajo un enfoque totalmente orientado a objetos, a excepción del procedimiento de entrada a la aplicación, el resto del trabajo se ha realizado a base de formularios y clases. Esto demuestra como es posible afrontar un proyecto empleando exclusivamente un diseño orientado a objetos.

Compilar un componente de código ActiveX

Finalizado el desarrollo del componente, debemos compilarlo mediante la opción *Archivo+Generar NombreComponente* del menú de VB. Durante este proceso, VB creará, además del correspondiente

fichero DLL o EXE, una entrada en el Registro de Windows y una librería de tipos del componente. La librería incluye la descripción de los interfaces que proporciona el componente.

Si queremos seguir probando el componente desde una aplicación cliente, hemos de reasignar la referencia en la aplicación cliente, estableciéndola al fichero que contiene el componente compilado en vez del proyecto en donde se estaba desarrollando el componente.

Es importante establecer la compatibilidad de versiones en el componente en el caso de que necesitemos esta característica, puesto que cada vez que compilemos el componente, VB genera un nuevo conjunto de identificadores globales únicos para las clases del componente. Si hemos establecido la compatibilidad de versiones en el componente, estos identificadores se mantendrán con su valor original entre cada compilación, en caso contrario, se generarán unos nuevos, con lo que las aplicaciones cliente que estuvieran usando versiones anteriores del componente, provocarán un error al intentar usar el componente con unos identificadores distintos de los que tiene.

Distribuir un componente de código ActiveX

El componente una vez creado, ya está listo para ser distribuido tanto a usuarios que lo utilicen en sus aplicaciones, como a otros programadores que necesiten aprovechar las capacidades que proporcionan sus objetos.

Para distribuir el componente, es necesario registrarlo en el equipo que vaya a usarlo. Para ello lo más adecuado es usar el asistente de instalación de aplicaciones de Visual Basic, y generar un programa de instalación para el componente. En el caso de que no queramos usar esta utilidad, podemos usarla al menos para crear un fichero de dependencias (DEP), que contenga la información sobre los ficheros que necesita el componente para instalarse, además de la información para el Registro de Windows.

Automatización ActiveX

Este concepto se puede definir como un interfaz que permite a una aplicación cliente un elevado nivel de control sobre otra aplicación servidora. Según la definición de Microsoft, Automatización ActiveX es "un modo, independiente del lenguaje, de manipular los métodos de un control ActiveX desde una aplicación externa. Se utiliza preferentemente para crear componentes que hacen que sus métodos estén disponibles para las herramientas de programación y los lenguajes de macros (script languages), tales como los lenguajes de macros de Excel o Word, o los lenguajes VBScript o JScript".

Uno de los grandes beneficios de la Automatización ActiveX consiste en la programación de las diferentes aplicaciones incluidas en Office. La programación de una herramienta de Office consiste en emplear su lenguaje de macros para adaptar el funcionamiento de dicha herramienta a nuestras necesidades a través de un conjunto de macros. Con las versiones anteriores de Office esto ya era posible, sin embargo, cada aplicación tenía un lenguaje de macros distinto, lo que no permitía crear macros para realizar ciertas tareas comunes a todas las aplicaciones de la suite. Ante esta situación, se hacía necesario un lenguaje común, que sirviera para programar cualquier aspecto de Office.

La solución a este problema viene de la mano de Visual Basic para Aplicaciones (VBA), que consiste en un lenguaje y entorno de desarrollo común, no sólo para programar las herramientas de Office, sino para todas las aplicaciones que hasta la fecha lo han licenciado. Esto supone que nuestro potencial campo de acción se amplia a todo ese abanico de productos que integran VBA como lenguaje de macros. Si sabemos programar en Visual Basic, se puede decir que sabemos programar en Visual

Basic para Aplicaciones, ya que a excepción de pequeñas diferencias, y de la jerarquía de objetos de la herramienta a programar, se utiliza de la misma manera.

Todo lo anterior se refiere a mejoras para el usuario normal o avanzado de Office, pero si nuestro caso es el de programador de VB, disponemos además, de un aspecto mayormente interesante si cabe: el de utilizar Office como herramienta de apoyo para nuestras aplicaciones. De esta forma nuestra aplicación actuaría como cliente de Office, utilizando las características que necesitáramos, ahorrando tiempo de desarrollo, aparte de la ventaja de saber que empleamos elementos sobradamente fiables que ya están funcionando en otras aplicaciones.

Los ejemplos mostrados con las aplicaciones de Office a lo largo de este tema, pertenecen a la versión Office97, por lo que el lector deberá disponer de dicha versión del producto para poder ejecutarlos en su equipo.

Conexión con Office. Uso de Word desde Visual Basic

Imaginemos la siguiente situación: estamos creando un formulario para realizar una entrada de datos, uno de los campos está destinado a escribir un texto a modo de observaciones o comentarios para el registro de la tabla y necesitamos realizar una operación de corrección ortográfica una vez que el usuario haya escrito dicho texto. Tenemos dos posibles soluciones al problema: escribir nosotros una rutina de revisión del texto, complicada tarea, sobre todo si vamos justos en los plazos de entrega de la aplicación; o lo más interesante, utilizar el revisor ortográfico que incorpora Word, conocido por la mayoría de usuarios y de gran fiabilidad. Estaremos de acuerdo en que esta última solución es la más atractiva.

Veamos pues, el ejemplo propuesto en la aplicación *AxWord*. En primer lugar, hemos de establecer en la ventana Referencias del proyecto, una referencia hacia la librería de objetos de Word, *Microsoft Word 8.0 Object Library*, de manera que las clases y constantes empleadas por este procesador de textos estén disponibles tanto para el código de la aplicación como para el Examinador de objetos.

A continuación crearemos el formulario *frmNota*. Se trata de un sencillo formulario en el cual disponemos de dos controles TextBox multilínea. En el control *txtOriginal* introduciremos el texto a revisar, y al pulsar el CommandButton *cmdRevisar*, se ejecutará el código de su procedimiento Click(), iniciando Word en el caso de que no estuviera ya en ejecución, mediante un objeto de automatización. Una vez iniciado Word, al que siempre controlaremos desde nuestra aplicación, tengamos esto muy presente, creamos un nuevo documento al que se traspasa el contenido de txtOriginal, invocamos al corrector ortográfico de esta herramienta, de forma que el usuario pueda revisar los posibles errores, y una vez corregidos, seleccionamos el texto del documento pasándolo al control *txtRevisado* destinado a contener el texto corregido. El código fuente 367 muestra el código del procedimiento, con los comentarios correspondientes para cada acción.

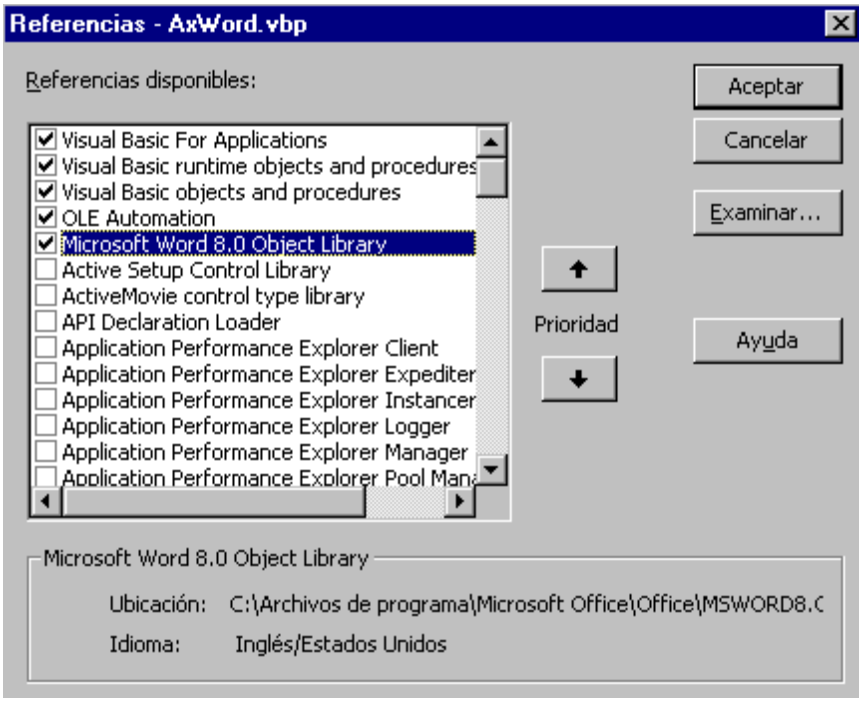


Figura 399. Establecimiento de la referencia a la libraría de objetos de Word.

```
Private Sub cmdRevisar_Click()
' declarar un objeto Word
Dim oWord As Word.Application
Dim lbCerrarWord As Boolean
On Error Resume Next
lbCerrarWord = False
' si ya existe una instancia de Word, usarla
Set oWord = GetObject(, "Word.Application")
' si Word no está funcionando...
If Err.Number > 0 Then
   Err.Clear

   ' instanciar un objeto Word
   Set oWord = CreateObject("Word.Application")
   lbCerrarWord = True
End If
' visualizar Word
oWord.Application.Visible = True
' añadirle un nuevo documento
oWord.Documents.Add
' asignar posición y tamaño
oWord.Application.Resize Width:=257, Height:=288
oWord.Application.Move Left:=214, Top:=64
' traspasar texto de la aplicación a Word
oWord.Selection.TypeText Text:=Me.txtOriginal.Text
' situar el cursor al principio del documento
oWord.Selection.HomeKey
' revisar ortografía en el documento
oWord.ActiveDocument.CheckSpelling
' seleccionar el texto del documento Word
oWord.Selection.Paragraphs(1).Range.Select
' eliminar de la selección el caracter de retorno de carro
oWord.Selection.MoveLeft Unit:=wdCharacter, Count:=1, Extend:=wdExtend
' traspasar texto del documento Word
' a la aplicación
Me.txtRevisado.Text = oWord.Selection
```

```
' cerrar el documento sin salvar los cambios
oWord.ActiveDocument.Close wdDoNotSaveChanges
' si al ejecutar este procedimiento
' Word no estaba abierto, cerrarlo
If lbCerrarWord Then
   ' cerrar Word
   oWord.Application.Quit
End If
' eliminar la referencia a la variable que
' contiene el objeto Word
Set oWord = Nothing
End Sub
```

Código fuente 367

Como podría darse la situación de que Word ya estuviera cargado, utilizamos la función *GetObject()*, que tiene la siguiente sintaxis:

`GetObject([cRutaObjeto], [cClase])`

- cRutaObjeto. Cadena con la ruta y el nombre del archivo que contiene la clase de la que se va a instanciar el objeto. Si se omite este parámetro, es obligatorio utilizar cClase.

- cClase. Cadena con el nombre de la clase de la cual vamos a recuperar el objeto. Dicho objeto puede estar contenido en el parámetro cRutaObjeto o bien estar ya instanciado.

Aunque se puede utilizar para crear nuevas instancias de una clase, en nuestro caso utilizamos GetObject() para comprobar si Word ya está siendo utilizado. En caso afirmativo, GetObject() devuelve una referencia al objeto Word. Si por el contrario, Word no se está utilizando, se generará un error que capturaremos acto seguido, instanciando el objeto esta vez con CreateObject().

La figura 400, muestra el corrector ortográfico durante su funcionamiento con nuestra aplicación:

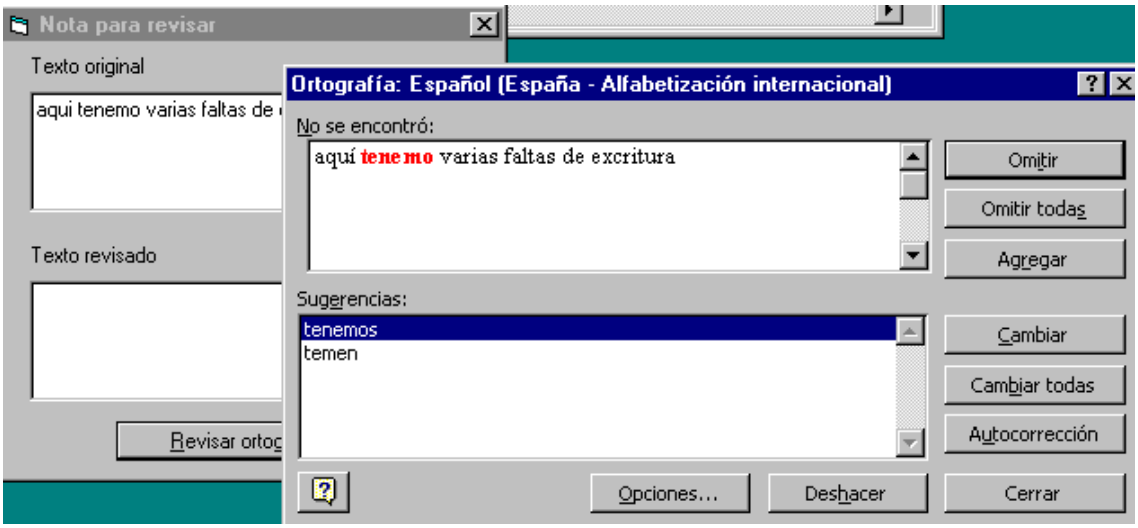


Figura 400. Aplicación VB usando el corrector ortográfico de Word.

Hemos podido comprobar que utilizando la jerarquía de objetos de Word, podemos manipular cualquier aspecto de esta aplicación. Si necesitamos consultar el contenido de esta jerarquía, podemos

hacerlo con el Examinador de objetos o desde el entorno de programación de Word, elemento este que veremos en el siguiente apartado.

Como aspectos adicionales a comentar del ejemplo anterior, tenemos el hecho de que una vez revisado el documento, después de seleccionar el texto, es conveniente eliminar de dicha selección, la última posición, correspondiente al carácter de retorno de carro, o de lo contrario se verá en la aplicación VB una vez traspasado el texto desde Word, lo que produce un mal efecto estético de cara al usuario.

Por otro lado, este ejemplo visualiza Word al utilizar el corrector, lo que no es habitual en este tipo de situaciones, ya que quien proporciona el interfaz visual es la aplicación cliente VB, mientras que Word se emplea como mera herramienta de apoyo. El motivo de mostrarlo es sólo a efectos didácticos, en condiciones normales debería permanecer oculto.

Office llama al código de Visual Basic

El título de este apartado quizá suene sensacionalista, pero es real, cualquiera de las aplicaciones de Office, puede ser programada para usar las clases de un componente ActiveX que hayamos desarrollado en Visual Basic, y utilizarlo mediante automatización.

Recordemos uno de los ejemplos anteriores, en el cual una aplicación cliente, instanciaba objetos de las clases contenidas en un componente ActiveX DLL. Pues bien, aquí estamos en el mismo caso, sólo que ahora será Excel la aplicación cliente que genere objetos a partir de esa misma DLL. Para ello, debemos disponer de la aplicación de ejemplo incluida en el apartado *Desarrollo de un componente ActiveX DLL* de este mismo tema; en concreto, necesitaremos los ficheros que forman el componente AxMant.

El trabajo que vamos a realizar, se basa en crear desde Excel un objeto de la clase Factura, contenida en el componente AxMant, y mediante ese objeto, recuperaremos ciertos valores de los registros de la tabla de facturas, situándolos en las celdas de la hoja de cálculo. Seguidamente, seleccionaremos las celdas en las que hemos situado la información y construiremos un gráfico estadístico. Todo ello utilizando código, sin intervención del usuario. El fichero *Fact.XLS* contiene la hoja de Excel con el código de este ejemplo.

Comenzaremos ejecutando Excel, y mediante la opción *Herramientas+Macro+Editor de Visual Basic* de su menú, entraremos en el entorno de programación de esta herramienta, que no es otro que Visual Basic para Aplicaciones.

Lo que vamos a hacer aquí es crear una macro para Excel, pero en lugar de crearla con la grabadora de macros, lo haremos escribiendo directamente el procedimiento de código que contiene la macro.

Antes de escribir código en Excel, pasaremos al entorno de Visual Basic y pondremos en ejecución el componente AxMant. Seguidamente volveremos al entorno VBA de Excel y estableceremos la referencia hacia el componente que acabamos de poner en ejecución, desde la ventana Referencias.

Si no siguiéramos estos pasos, al escribir el código en Excel, no tendríamos disponible en el Examinador de objetos las clases del componente, ni tampoco aparecerían las listas de propiedades y métodos en el editor de código, que son de suma ayuda en la fase de desarrollo.

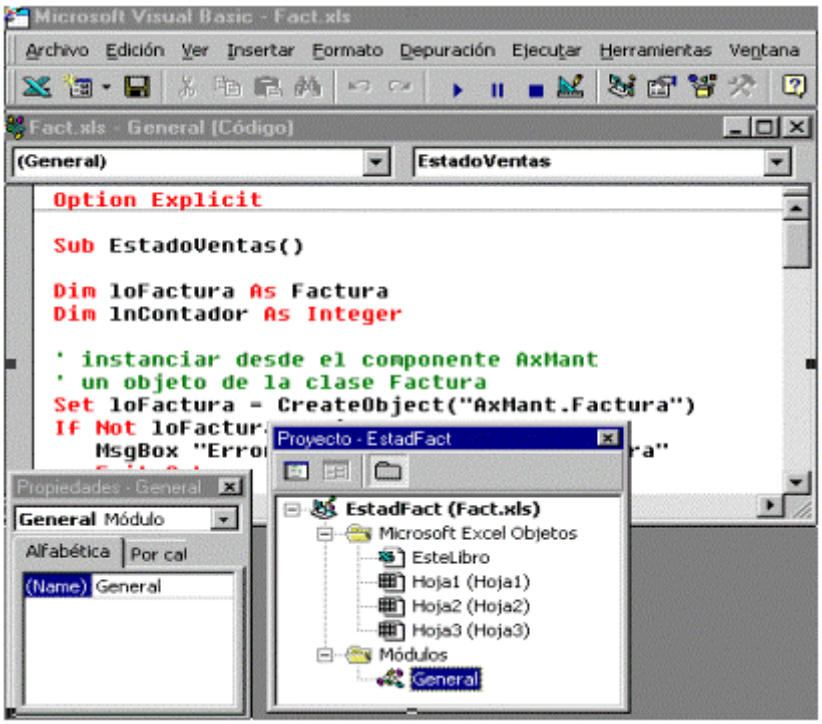


Figura 401. Entorno de programación VBA para las aplicaciones de Office.

Tras las anteriores acciones, procederemos a escribir el código para el procedimiento en Excel. Si durante este proceso, necesitamos consultar la jerarquía de objetos programables de la aplicación de Office sobre la que estamos trabajando (Excel, Word, etc.), sólo tenemos que invocar el Examinador de objetos para ver el conjunto de clases contenidas en la herramienta. En el caso de que necesitemos visualizar la estructura gráfica de una determinada jerarquía, pulsaremos el botón de ayuda en el Examinador de objetos, que nos introducirá en la ayuda de VB para esa herramienta, y en la que podremos ver la estructura de la jerarquía.

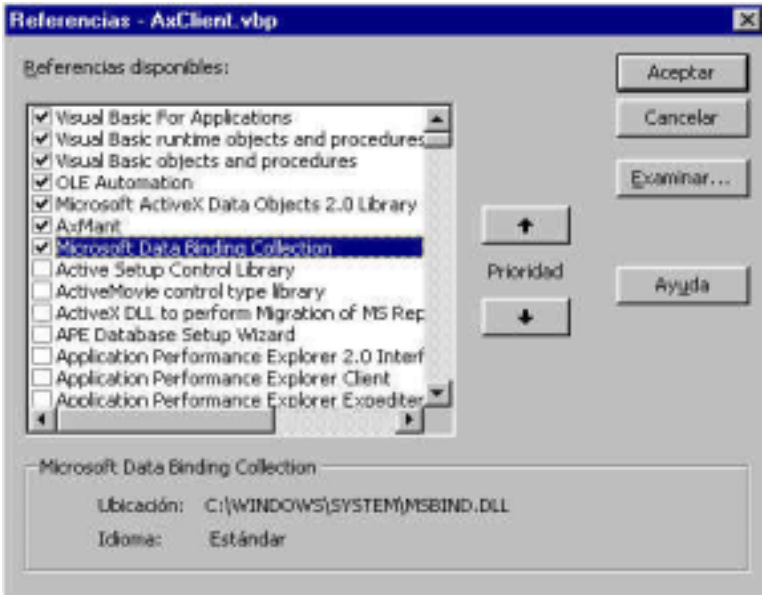


Figura 402. Establecimiento desde el entorno VBA de Excel de la referencia hacia el componente ActiveX AxMant.

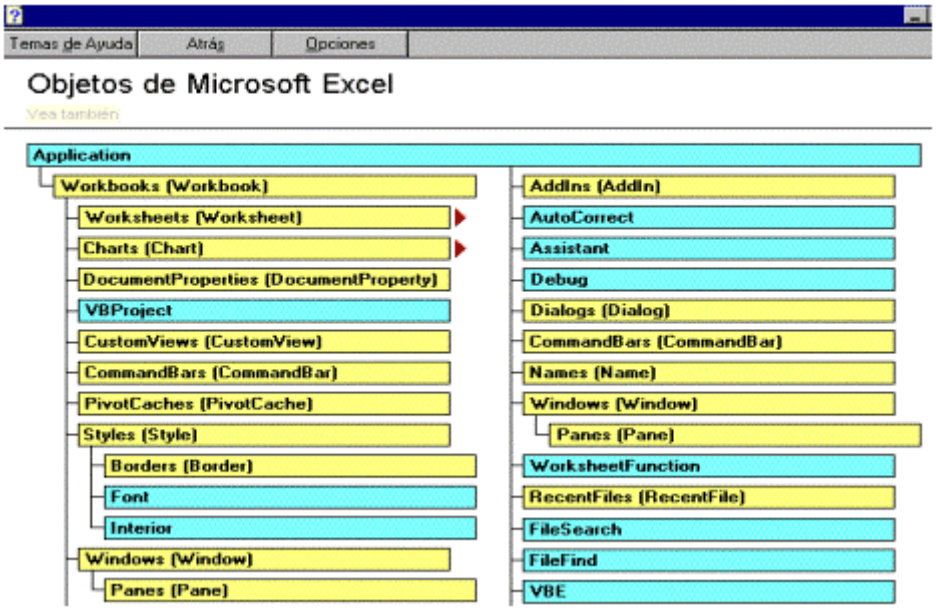


Figura 403. Jerarquía de objetos de Excel.

Ahora ya sólo queda escribir el código para el procedimiento de la macro, que podemos ver en el código fuente 368, con los comentarios correspondientes paso a paso.

```
Sub EstadoVentas()
' declarar una variable para el objeto
' a instanciar desde el componente ActiveX
Dim loFactura As Factura
Dim lnContador As Integer
' instanciar desde el componente AxMant
' un objeto de la clase Factura
Set loFactura = CreateObject("AxMant.Factura")
' si se producen errores al acceder a la tabla
' de facturas, salir del procedimiento
If Not loFactura.AbrirDatos Then
   MsgBox "Error al crear el objeto factura"
   Exit Sub
End If
' inicializar variable de contador
' para las filas de la hoja
lnContador = 1
' recorrer la tabla de clientes
' identificada por la propiedad Lista
' del objeto factura
Do While Not loFactura.Lista.EOF
   ' cargar en el objeto
   ' el registro de la tabla
   loFactura.Cargar

   ' seleccionar celda para poner
   ' el nombre del artículo
   ' la variable de contador nos
   ' guía para situarnos en la fila
   ' correcta de la hoja
   Range("A" & lnContador).Select
   ActiveCell.Formula = loFactura.Articulo
   ' seleccionar celda para poner
```

```
   ' el importe del artículo
   Range("B" & lnContador).Select
   ActiveCell.Formula = loFactura.Importe
   loFactura.Lista.MoveNext
   lnContador = lnContador + 1
Loop
' agregar un gráfico estadístico
' en la hoja
Charts.Add
' establecer el tipo de gráfico
ActiveChart.ChartType = xl3DColumnClustered
' seleccionar el rango de datos que
' usará el gráfico
ActiveChart.SetSourceData Source:=Sheets("Hoja1").Range("A1:B3"), _
   PlotBy:=xlRows
' situar el gráfico como un objeto en la hoja
ActiveChart.Location Where:=xlLocationAsObject, Name:="Hoja1"
' asignar un título al gráfico
ActiveChart.HasTitle = True
ActiveChart.ChartTitle.Characters.Text = "Estado de ventas"
End Sub
```

Código fuente 368

Existe un estupendo truco cuando nos encontramos en la situación de que no sabemos que objeto o método utilizar para realizar cierta acción. Consiste en poner en marcha la grabadora de macros de la herramienta en la que nos encontremos, y realizar las mismas operaciones que necesitemos codificar. Una vez terminada la grabación de la macro, procederemos a editarla, lo que nos llevará a un procedimiento en VBA que contiene el código de la macro. Aquí sólo tenemos que comprobar como está creado dicho código y adaptarlo a la situación que necesitemos.

El resultado de ejecutar esta macro desde VBA o desde Excel, podemos verlo en la figura 404.

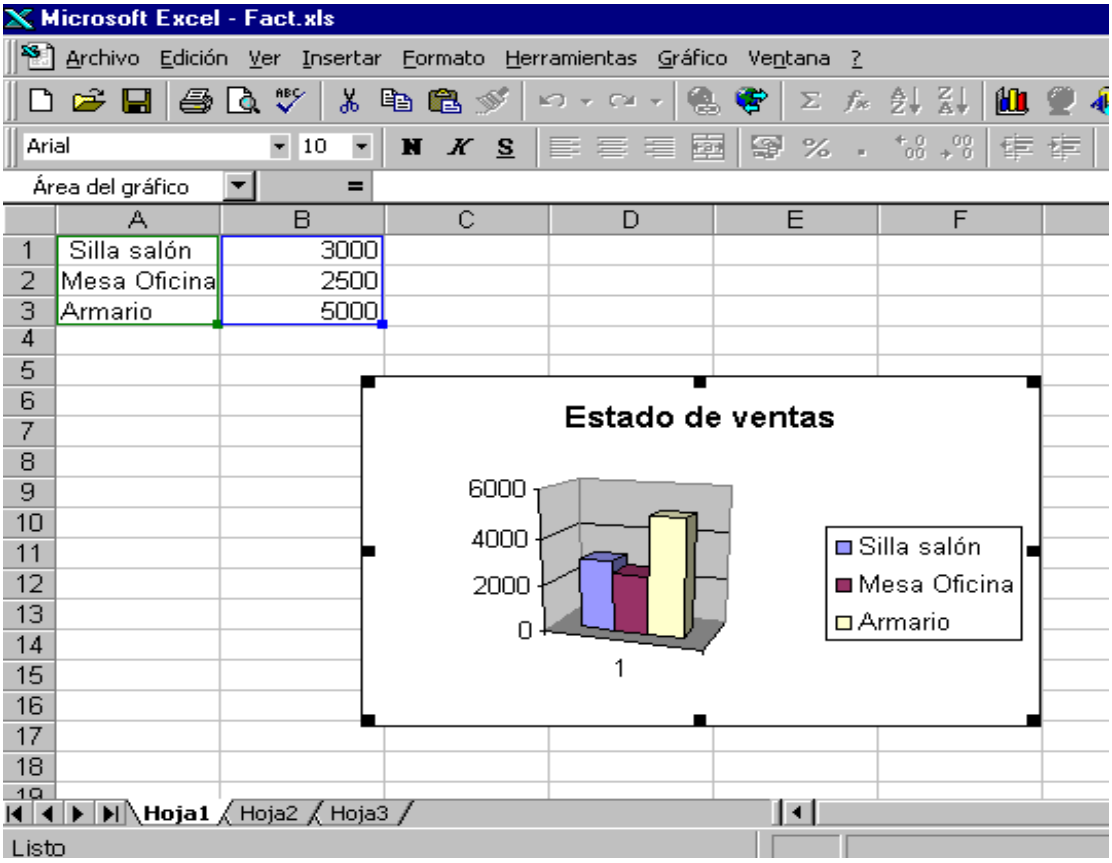


Figura 404. Resultado de la ejecución de la macro que hace llamadas a un componente ActiveX desarrollado en VB.

Como acabamos de comprobar, el uso combinado de Visual Basic y Office mediante Automatización ActiveX, proporciona a nuestros desarrollos una potencia inusitada, consiguiendo un alto nivel de integración mediante el uso de componentes, que nos permiten manipular prácticamente todos los aspectos de las aplicaciones basadas en esta tecnología.

Objetos insertables

Hasta el momento, hemos cubierto la parte complicada en la programación de componentes ActiveX. Pueden existir situaciones, sin embargo, en las cuales no necesitemos una programación exhaustiva del componente para conseguir los resultados necesitados, primando por el contrario, la rapidez en disponer de la aplicación lista para ser utilizada.

En este tipo de casos, existe otra forma de manipular los objetos de las aplicaciones basadas en ActiveX: mediante objetos insertables o a través del control contenedor OLE.

Un objeto insertable, es un elemento de una aplicación que podemos incluir en un formulario de un programa desarrollado en Visual Basic y manipular desde este programa, igual que si estuviéramos trabajando con el objeto desde su aplicación nativa.

Uno de los ejemplos más habituales para ilustrar este particular, es la inclusión de un documento de Word en un formulario y la edición del mismo desde dicho formulario. Para ello, una vez situados en un proyecto de VB, seleccionaremos la opción de su menú *Proyecto+Componentes*, que mostrará la

ventana *Componentes*. Pulsaremos sobre la pestaña *Objetos insertables* de esta misma ventana para obtener una lista de los objetos con los que podemos trabajar, como se puede ver en la figura 405.

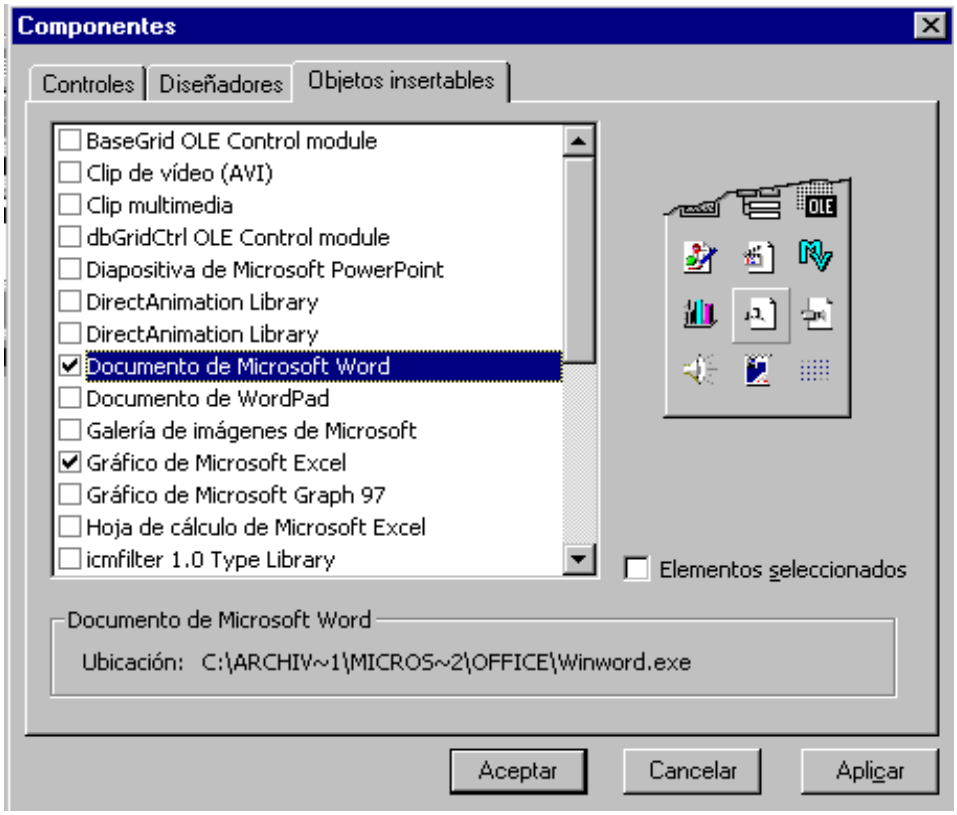


Figura 405. Objetos insertables disponibles desde Visual Basic.

Marcaremos el objeto *Documento de Microsoft Word* y pulsaremos el botón Aceptar de esta ventana, con lo cual se agregará en el Cuadro de herramientas de VB el objeto seleccionado, obsérvese en la figura 406.

Figura 406. Cuadro de herramientas de VB incluyendo Documento de Word.

El siguiente paso consiste en seleccionar este elemento del Cuadro de herramientas y dibujarlo en un formulario de la misma forma que hacemos con cualquiera de los controles habituales de VB. Lo que sucederá a continuación es que Word se pondrá en marcha permitiéndonos editar el documento insertado en el formulario igual que si estuviéramos trabajando desde el mismo Word. El propio menú de Visual Basic será sustituido por el menú de Word durante el periodo de tiempo en el que estemos editando el documento, permitiéndonos disponer de las funcionalidades del procesador de textos. Una vez terminada la edición del documento, pulsaremos *Escape* o haremos clic fuera del área del documento para volver al entorno habitual de VB. La figura 407 muestra el momento de la edición de un documento insertado en un formulario.

Figura 407. Edición de un documento de Word insertado en un formulario.

Si desde VB queremos volver al documento, seleccionaremos el mismo en el formulario y abriremos su menú contextual el cual nos mostrará las posibilidades de edición o verbos OLE, disponibles para el mismo (figura 408).

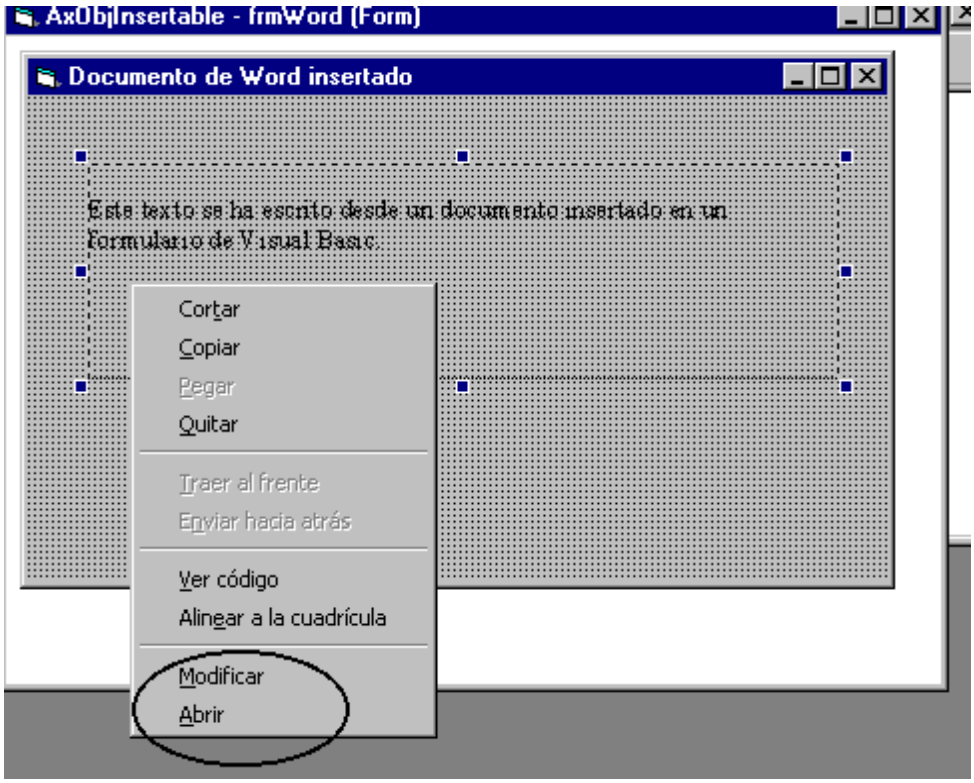


Figura 408. Modos de edición del documento desde el entorno de Visual Basic.

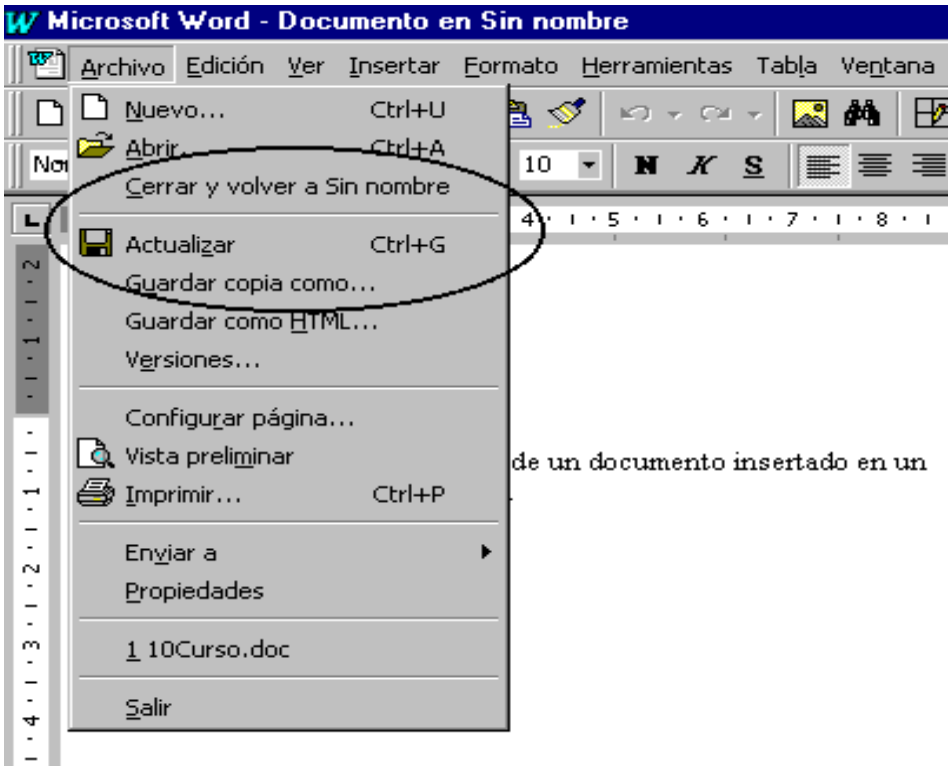


Figura 409. Menú Archivo de Word durante la edición de un documento insertado en un formulario de VB.

Si seleccionamos *Modificar*, se editará el documento en la forma vista anteriormente, desde el propio formulario.

Por el contrario, si elegimos *Abrir*, se pondrá en marcha Word y se realizará la edición desde el propio Word en un documento especial denominado *Documento en Sin Nombre*. Este es un detalle muy importante, ya que cuando editamos un documento de esta manera, Word lo sitúa en una zona especial denominada *Sin Nombre*, cambiando algunas de las opciones de su menú *Archivo*.

Cuando ejecutemos la aplicación, si queremos editar el documento, sólo debemos hacer doble clic sobre él para entrar en el modo de edición desde el formulario.

El lector puede comprobar los resultados de los pasos llevados a cabo en este apartado mediante el proyecto *AxObjInsertable*, que se acompaña como ejemplo, y que contiene el formulario con el documento insertado, empleado para estas pruebas.

El control Contenedor OLE

La técnica de objetos insertables, si bien presenta la ventaja de una gran facilidad y rapidez de uso, tiene el inconveniente de la poca flexibilidad que ofrece su programación, sólo tenemos disponibles dos verbos OLE en tiempo de diseño, limitándonos a una edición simple en tiempo de ejecución.

Para suplir las carencias de los objetos insertables, se recomienda el uso del control *Contenedor OLE*. Este control puede contener cualquier tipo de documento OLE. Además, de estar provisto de un conjunto de propiedades y métodos con los que obtenemos una mayor flexibilidad a la hora de programarlo. La figura 410 muestra el Cuadro de herramientas incluyendo este control:

Figura 410. Cuadro de herramientas de VB incluyendo control Contenedor OLE.

La aplicación *AxCtlOLE*, que se adjunta como ejemplo, utiliza el formulario MDI *mdiInicio*, desde el cual se visualizan varios formularios secundarios que contienen un control OLE cada uno, con las diferentes formas de tratamiento de dicho control. Veamos a continuación, cada uno de estos formularios

frmIncrusta

Una vez incluido este formulario en el proyecto, procederemos a insertar en su interior, un control Contenedor OLE. Dibujado este control, aparecerá automáticamente un cuadro de diálogo en el que deberemos seleccionar el tipo de objeto que contendrá el control.

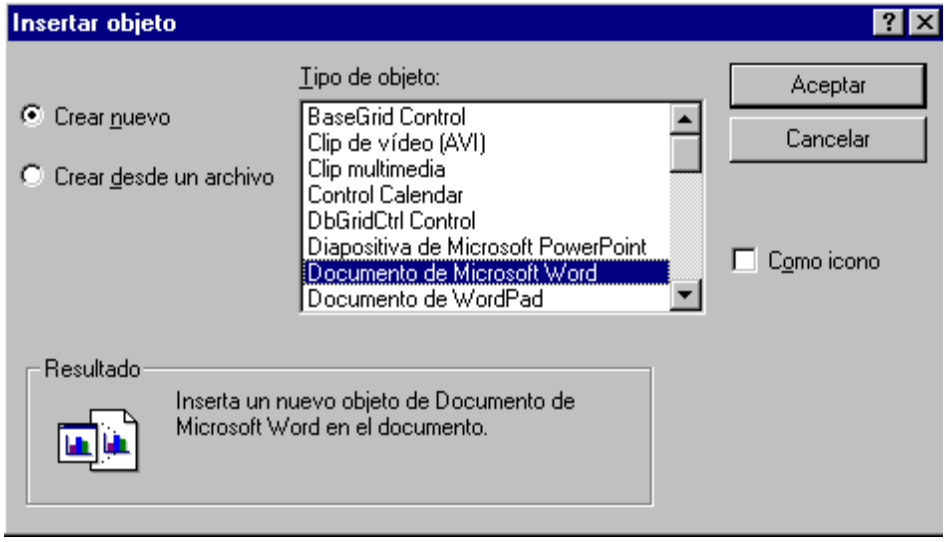


Figura 411. Cuadro de diálogo de selección de objeto a insertar en un Contenedor OLE.

Seleccionaremos el objeto *Documento de Microsoft Word*, dejando pulsado el OptionButton *Crear nuevo*. Pulsaremos el botón *Aceptar*, con lo que habremos incrustado un documento en el formulario a través de este control, que quedará listo para ser usado.

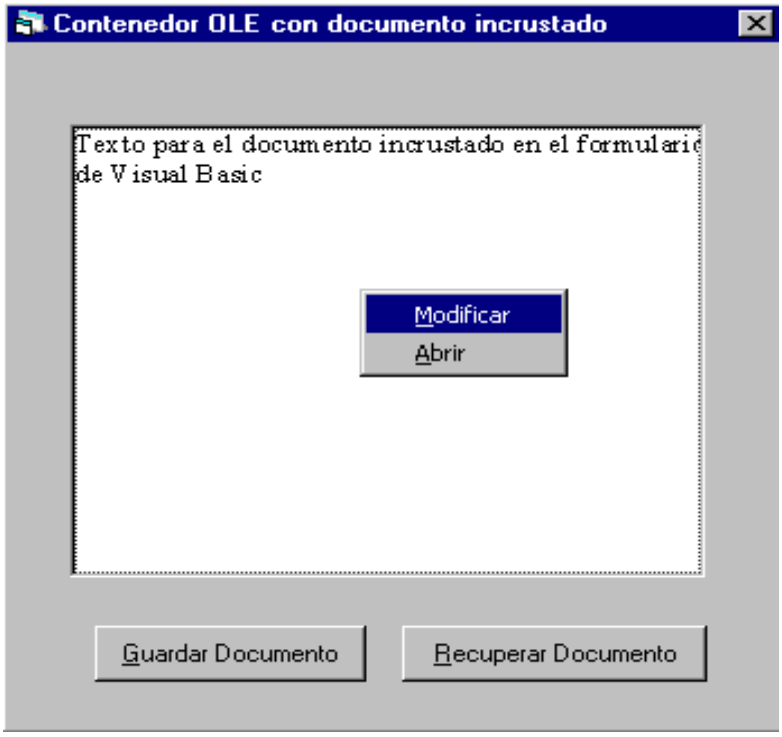


Figura 412. Verbos OLE disponibles en un control Contenedor en tiempo de ejecución.

Hasta el momento, el modo de trabajo es muy similar al que vimos con los objetos insertables. Sin embargo, ahora tenemos la ventaja de que al ejecutar la aplicación y hacer clic con el botón secundario del ratón sobre el control OLE, aparece la lista de verbos OLE disponibles para el documento (con los objetos insertables, sólo disponíamos de los verbos en modo de edición del control). Ahora podemos editar el documento desde el formulario o abrir su aplicación original para editarlo desde allí. La figura 412 nos muestra el formulario en ejecución con los verbos OLE disponibles.

A continuación vamos a utilizar alguno de los métodos de este control, para proporcionar al documento la funcionalidad de que carecía en el anterior ejemplo. Como acabamos de ver en la anterior figura, hemos insertado en el formulario, dos CommandButton cuya misión será guardar y recuperar el contenido del documento en un fichero respectivamente, para mostrarlo en el control OLE del formulario. La manipulación del fichero se realizará a bajo nivel, empleando los métodos *SaveToFile()* y *ReadFromFile()*, de esta forma nos aseguramos que siempre vamos a trabajar con los datos más recientes del documento.

El código fuente 369, muestra el código para estos dos botones:

```
Private Sub cmdGuardar_Click()
On Error GoTo HayError
Dim lnManejador As Integer
' tomar un manejador de fichero libre
lnManejador = FreeFile
' abrir el fichero donde vamos a guardar
' el contenido del documento
Open "c:\mis documentos\texto.doc" For Binary As #lnManejador
' guardar el documento en el fichero
oleDocumento.SaveToFile lnManejador
' -----------------------
' las siguientes líneas cierran el
' fichero y finalizan el procedimiento
FinProc:
Close #lnManejador
Exit Sub
' -----------------------
' si se produce un error, avisar al usuario
' y finalizar el procedimiento
HayError:
MsgBox "Error al guardar el documento"
Resume FinProc
End Sub
' ********************************************
Private Sub cmdRecuperar_Click()
On Error GoTo HayError
Dim lnManejador As Integer
' tomar un manejador de fichero libre
lnManejador = FreeFile
' abrir el fichero del que vamos a recuperar
' el contenido
Open "c:\mis documentos\texto.doc" For Binary As #lnManejador
' traspasar el contenido del fichero al control
oleDocumento.ReadFromFile lnManejador
' -----------------------
' las siguientes líneas cierran el
' fichero y finalizan el procedimiento
FinProc:
Close #lnManejador
Exit Sub
' -----------------------
' si se produce un error, avisar al usuario
```

```
' y finalizar el procedimiento
HayError:
MsgBox "Error al recuperar el documento"
Resume FinProc
End Sub
```

Código fuente 369

Es importante tener en cuenta, que la primera vez que usemos este formulario, después de añadir texto al documento, lo guardemos pulsando el botón *cmdGuardar*. De esta forma se incluirá información válida en el fichero que contiene el documento incrustado, creándose dicho fichero si no existiera. Si intentáramos recuperar sin la existencia del fichero, obtendríamos un error.

Aunque podemos editar el documento abriendo su aplicación original, al disponer de un documento incrustado, lo más sencillo es editarlo desde el propio formulario, dejando la edición desde la aplicación nativa para los documentos vinculados, que veremos a continuación.

frmVincula

La inserción en este formulario del control OLE, es igual que para el anterior hasta llegar a la caja de diálogo *Insertar objeto* del control. Aquí pulsaremos el OptionButton *Crear desde un archivo*, con lo que el aspecto de este diálogo cambiará al mostrado en la figura 413.

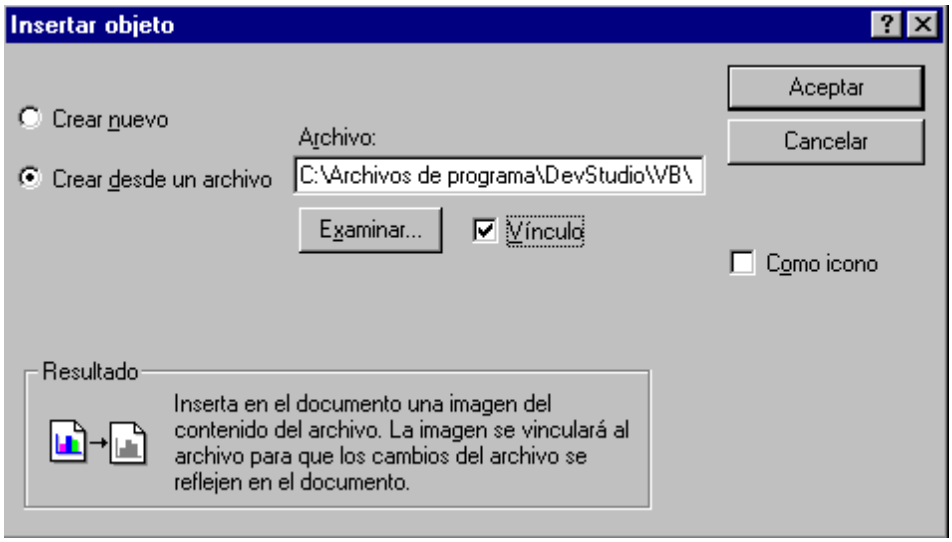


Figura 413. Insertar objeto en el Contenedor OLE desde un archivo vinculado.

Una vez seleccionado un archivo, marcaremos el CheckBox *Vínculo*, con lo cual el archivo que contiene el documento, quedará vinculado al control OLE de este formulario. Ahora, cada vez que ejecutemos alguno de sus verbos OLE para editarlo, se ejecutará la aplicación nativa del documento y lo editaremos desde ella.

El CommandButton *Actualizar*, incluido en este formulario, refresca el contenido del control OLE con el contenido del documento de Word. La ventaja de este sistema de trabajo, es que evitamos la manipulación del documento a bajo nivel, simplificando la actualización del control.

```
Private Sub cmdActualizar_Click()
Me.oleDocumento.Update
End Sub
```

Código fuente 370

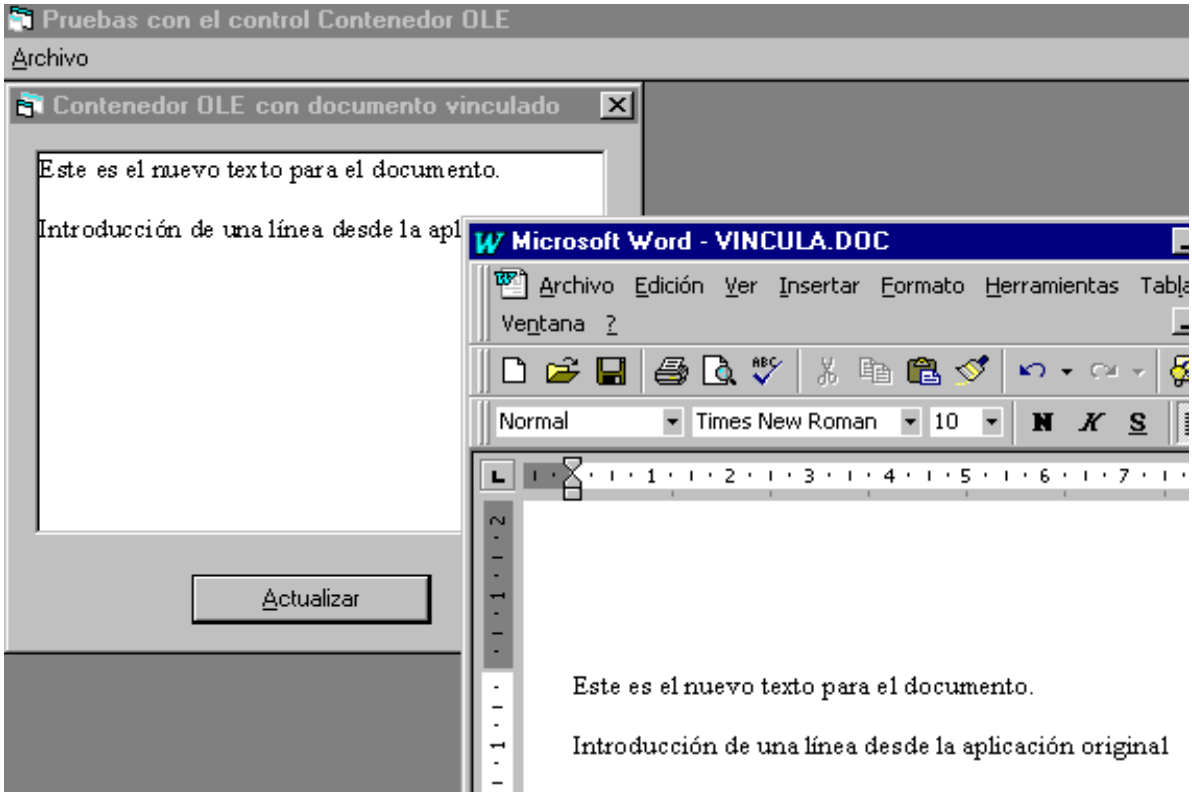


Figura 414. Edición del documento vinculado, lanzando desde el programa VB, la aplicación original del documento.

Fíjese el lector, que la propiedad *SourceDoc* en este ejemplo, contiene la ruta del documento vinculado, por lo que si utiliza una ruta diferente para probar esta aplicación, deberá actualizarla en dicha propiedad.

FrmEjecInc - frmEjecVinc

La particularidad de estos dos formularios, reside en que incrustan y vinculan respectivamente en tiempo de ejecución, un documento en su control Contenedor OLE. Para ello, una vez incluido el control OLE en el formulario, se cancela el cuadro de diálogo *Insertar objeto*, estableciendo su propiedad *OLETypeAllowed* a *Embedded* para el control incrustado y *Linked* para el vinculado. Por último, crearemos la conexión OLE correspondiente al visualizar cada formulario; esto lo haremos desde los procedimientos de menú del formulario *mdiInicio*. El código fuente 371, contiene estos procedimientos de apertura de los formularios y conexión al documento.

```
Private Sub mnuArIncrustEjec_Click()
Dim lfrmEjecInc As frmEjecInc
```

```
' cargar formulario que contiene el control OLE
Set lfrmEjecInc = New frmEjecInc
Load lfrmEjecInc
lfrmEjecInc.Show
' incrustar un documento en el control
' mediante el método CreateEmbed de dicho control
lfrmEjecInc.oleDocumento.CreateEmbed "c:\Mis documentos\datos.xls"
' ejecutar el verbo OLE por defecto
' del control
lfrmEjecInc.oleDocumento.DoVerb
End Sub
' ------------------------------------------
Private Sub mnuArVinculEjec_Click()
Dim lfrmEjecVinc As frmEjecVinc
' cargar formulario que contiene el control OLE
Set lfrmEjecVinc = New frmEjecVinc
Load lfrmEjecVinc
lfrmEjecVinc.Show
' vincular un documento en el control
' mediante el método CreateLink de dicho control
lfrmEjecVinc.oleDocumento.CreateLink "c:\Mis documentos\stock.xls"
' ejecutar el verbo OLE por defecto
' del control
lfrmEjecVinc.oleDocumento.DoVerb
End Sub
```

Código fuente 371

Emplearemos el método *CreateEmbed()* del control, para incrustar un archivo que pasaremos como parámetro. Para vincular un archivo, usaremos el método *CreateLink()*, perteneciente también a este control. En ambos casos ejecutaremos por último el método *DoVerb()*, que llama al verbo o acción OLE por defecto del documento (el primero de la lista de verbos disponibles), lo que provoca la ejecución de la aplicación asociada al documento (Excel en este caso), para permitir la edición al usuario.

Como hemos podido comprobar, el empleo de controles OLE, si bien no ofrece la potencia de la programación de componentes de código ActiveX, presenta la ventaja de una mayor facilidad de creación y configuración, resultando la solución idónea para aplicaciones que requieran de poca complejidad y/o debamos desarrollar lo más rápidamente posible.

Controles ActiveX

Un control ActiveX es un elemento comúnmente visual, que puede utilizarse dentro de un formulario de Visual Basic, documento de Internet, y en definitiva, en cualquier aplicación que soporte controles de este tipo (OCX).

En lo que respecta a la programación en VB, los controles ActiveX están accesibles a través del Cuadro de herramientas, y su manipulación en tiempo de diseño, es igual a la de los controles estándar, salvo la configuración específica que requiera el control, debido a sus características particulares.

La posibilidad de crear controles ActiveX con VB, era una parcela de desarrollo reservada exclusivamente al programador de C++ hasta hace poco tiempo. Ahora se abre para los programadores de VB una nueva vía de amplias posibilidades, hasta el momento limitada a lo que el desarrollador de C++ nos quisiera ofrecer. Ahora podemos crear nuestros propios controles para resolver los problemas que se nos planteen durante la creación de un programa.

Crear un control ActiveX no es tarea simple, pero gracias a los asistentes proporcionados por VB para esta labor, contaremos con una gran ayuda en nuestro trabajo.

En este apartado, vamos a abordar la creación de un control viendo los aspectos relevantes de cada fase del desarrollo del mismo. El lector puede hacer las pruebas necesarias mediante el grupo de proyectos Pruebas.VBG incluido como ejemplo, y que contiene por una parte, el proyecto AxFechas, al que pertenece el control ActiveX, y por otro lado, el proyecto PruFecha, una aplicación estándar que hace uso del control creado. Un grupo de proyectos es necesario en este tipo de desarrollo, ya que mientras en el proyecto que contiene el control ActiveX, creamos el mismo, en el proyecto que contiene la aplicación estándar, podemos insertar un formulario para ir probando los resultados del control.

Diseño del control

Al crear un control, hemos de situarnos primeramente en el papel del programador que lo usará al desarrollar una aplicación, proporcionando al control el comportamiento en cuanto a manipulación y código que dicho programador, al igual que con todos los controles, espera también del nuestro. Por ejemplo, al cambiarle de tamaño, que redimensione adecuadamente todo su contenido, o que disponga de los procedimientos necesarios para los diferentes eventos que ocurran en el control.

Después hemos de tener en cuenta al usuario final que utilizará el control en un programa, que también espera un comportamiento estándar de los elementos que forman el control. Por ejemplo, si hemos incluido un CommandButton en nuestro control, el usuario espera que ocurra algo al pulsarlo.

Otro aspecto muy importante en el diseño del control, lo hallamos en la creación de su interfaz de usuario. Dicho interfaz podemos enfocarlo partiendo de tres diferentes perspectivas:

- Combinar varios controles. Podemos aglutinar un grupo de controles, para conseguir una funcionalidad determinada. Por ejemplo: si usamos un TextBox y dos CommandButton, podemos conseguir un control autoincremental, que aumente o disminuya un número al pulsar en los botones.

- Mejorar un control existente. Si disponemos de un control que no contempla cierta característica, crearemos un nuevo control a partir del existente, añadiéndole esa nueva funcionalidad. Por ejemplo: un TextBox que sólo admita letras y las convierta a mayúsculas según se van introduciendo.

- Crear un control a partir de cero. Cuando necesitemos una característica no disponible en ninguno de los controles existentes, crearemos un nuevo control. Por ejemplo: un ListBox que disponga de un TextBox y dos CommandButton, para añadir a la lista, el contenido del TextBox y para eliminar el elemento seleccionado actualmente.

Efectuadas estas consideraciones, fijaremos el objetivo a desempeñar por nuestro control. En este ejemplo vamos a crear un control para introducir fechas, que podrán ser convertidas a letra, y a las que se podrá añadir o quitar días.

Crear el interfaz del control

En primer lugar crearemos un nuevo proyecto de tipo Control ActiveX (figura 415), asignándole el nombre AxFechas.

Un proyecto de este tipo, incorpora por defecto un UserControl o control de usuario, que es una plantilla que servirá como base para la creación del control. Aquí iremos depositando los controles que formarán parte de nuestro control. Si el proyecto no dispusiera de ningún control de usuario, podremos añadirlo mediante la opción del menú de VB Agregar+Agregar control de usuario, o mediante el menú contextual del Explorador de proyectos, en su opción Agregar.

Figura 415. Creación de un nuevo proyecto Control ActiveX.

Ya que el control va a consistir en una fecha personalizada, asignaremos a la propiedad Name del UserControl el valor FechaPers. Los controles ActiveX se guardarán en ficheros con extensión CTL y CTX, este tipo de ficheros contendrán el código e información binaria de los controles, no deben confundirse con los ficheros OCX, que contienen los controles ya compilados.

Insertaremos en el UserControl un TextBox llamado txtValor para poder teclear las fechas, y dos CommandButton: cmdSumar y cmdRestar, que serán los encargados de añadir y quitar respectivamente días a la fecha que aparezca en el control de texto. Estos controles que forman parte de nuestro control se denominan controles constituyentes. El control en modo de diseño, quedará como muestra la figura 416.

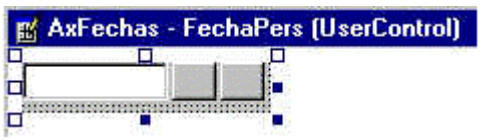


Figura 416. Control FechaPers en modo de diseño.

El código del control

A efectos de código fuente, un control se organiza igual que una clase, mediante propiedades, métodos, eventos, etc. En resumidas cuentas, un control es una clase que además proporciona interfaz visual. Por ello, cada vez que insertamos un control en un formulario, estamos creando una instancia de una clase. El control que aparece en el formulario es un objeto.

El código fuente 372, muestra la definición de las propiedades para el control, así como un conjunto de constantes con los valores por defecto que se usarán para inicializar las propiedades y la declaración de eventos del control.

```
Option Explicit
' valores por defecto para propiedades
Const mcDEFFormatoDiaSem As String = "ddd"
Const mcDEFFormatoDia As String = "d"
Const mcDEFFormatoMes As String = "m"
Const mcDEFFormatoAño As String = "yy"
Const mcDEFSeparador As String = "/"
Const mcDEFCaracterSumar As String = "+"
Const mcDEFCaracterRestar As String = "-"
' variables de propiedad
' ----------------------
' cadena con el formato del día de la semana
' para la fecha del control
Private mcFormatoDiaSem As String
' cadena con el formato del día del mes
' para la fecha del control
Private mcFormatoDia As String
' cadena con el formato del mes
' para la fecha del control
Private mcFormatoMes As String
' cadena con el formato del año
' para la fecha del control
Private mcFormatoAño As String
' cadena con el tipo de separador
' empleado para mostrar la fecha
' en el control
Private mcSeparador As String
' cadena con el caracter que aparecerá
' en el botón que realiza la operación
' de suma de fechas
Private mcCaracterSumar As String
' cadena con el caracter que aparecerá
' en el botón que realiza la operación
' de resta de fechas
Private mcCaracterRestar As String
' objeto con el fuente utilizado
' por los controles constituyentes
' del control
Private moFuente As Font
' declaración de eventos
Event DblClick()
Event KeyPress(KeyAscii As Integer)
```

Código fuente 372

También se crearán los respectivos procedimientos Property para las propiedades anteriores, de forma que sean accesibles desde el exterior del código del control. Estos procedimientos y el resto del código no incluido en este apartado, para no recargarlo en exceso, puede consultarlo el lector abriendo el proyecto desde VB.

Seguidamente, y como ejemplo de método perteneciente al control, se muestra el código fuente 373 con el código que realiza la conversión a cadena de caracteres de la fecha que aparece en el TextBox del control.

```
Public Function Convertir() As String
' pasar la fecha actual del control a
' una fecha compuesta por letras
If Not IsDate(txtValor.Text) Then
   Convertir = ""
   Exit Function
End If
Convertir = DiaSem & " " & Dia & " de " & Mes & " de " & Año
End Function
```

Código fuente 373

Redimensionar el control

Cuando insertamos un control cualquiera en un formulario y lo redimensionamos, el control cambia de tamaño adaptándose a las nuevas medidas que le asigna el programador. Si intentamos hacer lo mismo con nuestro control, veremos que sólo se redimensiona el control base, pero no los controles constituyentes, ¿por qué?. La causa es que no proporcionamos el código que lo haga. Para ello, hemos de codificar el evento Resize() del UserControl, que se produce cada vez que el programador modifica el tamaño del control y al ejecutar la aplicación.

En este evento se deben establecer las unidades de medida para el alto y ancho del control a través de las propiedades ScaleWidth y ScaleHeight, y en función de esta escala, se asignará a cada control constituyente la zona que puede ocupar dentro del control. Veamos la anterior explicación de un modo práctico: supongamos que asignamos al control un valor de 100 para la altura y 105 para la anchura, los controles podrían distribuirse en ese caso según el esquema mostrado en la figura 417.

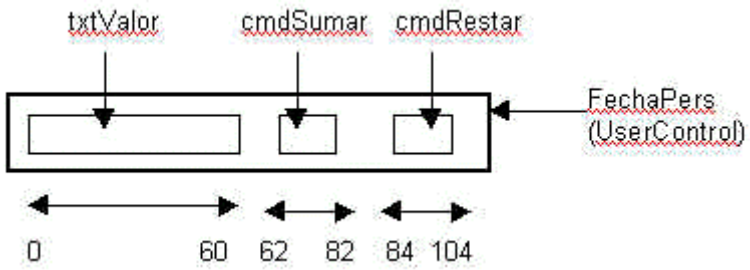


Figura 417. Distribución de controles constituyentes dentro de un control ActiveX, según la escala de tamaños del control.

Según estos valores, cada vez que redimensionemos el control, los controles constituyentes tomarán un nuevo tamaño en función de la escala establecida, adaptándose siempre al tamaño del UserControl. Veamos, en el código fuente 374, como se aplica lo anterior al código de Resize().

```
Private Sub UserControl_Resize()
' establecer la escala de medida para el control
ScaleWidth = 105
ScaleHeight = 100
' reposicionar los controles,
' adaptándolos al nuevo tamaño
txtValor.Move 0, 0, 60, 100
cmdSumar.Move 62, 0, 20, 100
cmdRestar.Move 84, 0, 20, 100
```

```
End Sub
```

Código fuente 374

Las figuras 418 y 419, muestran el control incluido dentro de un formulario después de haber sido redimensionado.

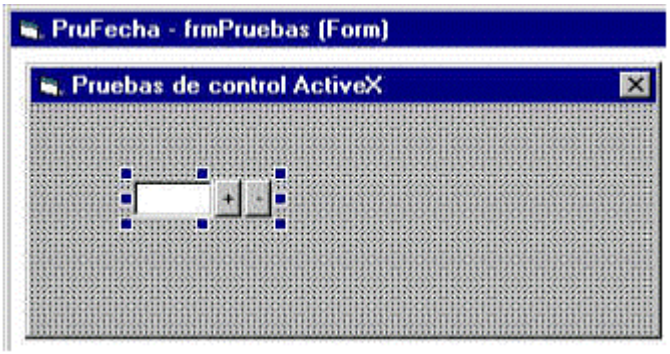


Figura 418. Control FechaPers con tamaño reducido.

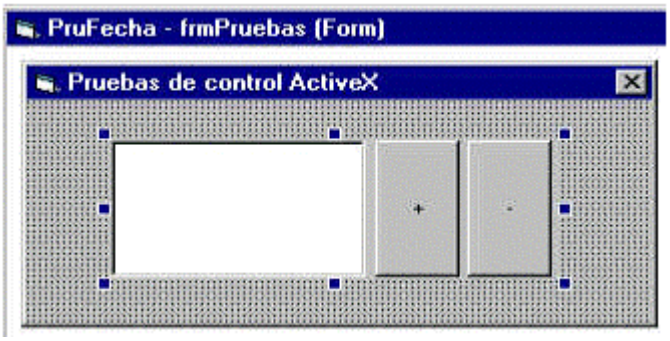


Figura 419. Control FechaPers con tamaño ampliado.

Eventos notificados por el control

Cuando insertamos un control ActiveX en un formulario, este proporciona por defecto un conjunto de eventos: DragDrop(), DragOver(), GotFocus() y LostFocus(); que podemos codificar para cuando se produzcan estos sucesos en el control, pero ¿y si necesitamos controlar otro tipo de evento?. Supongamos que queremos ser notificados cuando el usuario hace doble clic en el TextBox de nuestro control. Los pasos a seguir son los siguientes:

- En la zona de declaraciones del módulo de código del control, declararemos el evento que se notificará.

```
' declaración de eventos
Event DblClick()
```

Código fuente 375

- Nos situaremos en el código del evento del mismo nombre del TextBox, situando la instrucción que lanza el evento.

```
Private Sub txtValor_DblClick()
RaiseEvent DblClick
End Sub
```

Código fuente 376

- A partir de ahora, al insertar el control en un formulario, cuando accedamos al código del control, entre los eventos a codificar, dispondremos del que acabamos de definir. La figura 420 nos muestra la lista de procedimientos disponibles para este control, una vez incluido en un formulario. Entre estos estará el que acabamos de definir desde el código del control. En este caso particular, la acción a desencadenar cuando se haga doble clic en el TextBox del control, será llamar al método Convertir() del control y mostrar la fecha como cadena de caracteres.

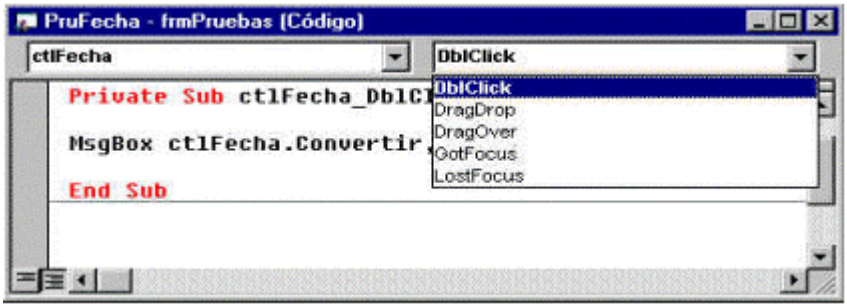


Figura 420. Lista de procedimientos de evento para el control ActiveX.

Un aspecto muy importante, que debe tener en cuenta el lector sobre lo que acabamos de ver, reside en el hecho de que el evento se lanzará sólo desde el elemento del control que se indique. En el caso anterior, el TextBox constituyente del control ActiveX. Si necesitáramos, pongamos por caso, que dicho evento se lanzara también al hacer doble clic en una zona del control no ocupada por un control constituyente (en el espacio que hay entre los dos CommandButton, por ejemplo), deberíamos hacer la misma operación en el evento DblClick() de UserControl que la efectuada en el TextBox.

```
Private Sub UserControl_DblClick()
RaiseEvent DblClick
End Sub
```

Código fuente 377

Debemos proporcionar todos los eventos que consideremos oportunos en nuestros controles, de ello depende que el control sea más o menos flexible a la hora de ser utilizado por el programador. En este ejemplo sólo hemos puesto a disposición del programador el evento DblClick(), pero en un caso real, hemos de procurar, si el diseño del control o la finalidad a que va a ser destinado lo permiten, proporcionar un conjunto más amplio de eventos.

Creación de propiedades

Las propiedades de un control ActiveX, si bien se crean de igual forma que para una clase normal, tienen un tratamiento claramente distinto con respecto a estas últimas. Durante el desarrollo de una aplicación, las propiedades de una clase normal se manipulan mediante el código de la clase, pero cuando se trata de un control ActiveX, debemos tener en cuenta que también se manipulan mediante la ventana de propiedades del control y a través de las páginas de propiedades o PropertyPage.

Veamos a continuación las fases que componen el desarrollo de una propiedad de un control, tomando como muestra la propiedad CaracterSumar. Esta propiedad es la encargada de mostrar un carácter en el Caption del botón que se ocupa de sumar días a la fecha del control.

- Declarar variable y constante de propiedad. Comenzaremos situando en la zona de declaraciones del módulo, la variable de propiedad para el control y una constante que será tomada al crear el control por primera vez.

```
' valores por defecto para propiedades
Const mcDEFCaracterSumar As String = "+"
' variables de propiedad
' ----------------------
' cadena con el caracter que aparecerá
' en el botón que realiza la operación
' de suma de fechas
Private mcCaracterSumar As String
```

Código fuente 378

- Crear los procedimientos Property para la propiedad. Como ya sabe el lector, mediante estos procedimientos, tomaremos y asignaremos el valor de la propiedad.

```
Public Property Get CaracterSumar() As String
CaracterSumar = mcCaracterSumar
End Property

' ------------------------------------------------
Public Property Let CaracterSumar(ByVal vcCaracterSumar As String)
mcCaracterSumar = vcCaracterSumar
PropertyChanged "CaracterSumar"
TituloBotones
End Property
```

Código fuente 379

Observe el lector, que en el procedimiento de asignación se utiliza el método PropertyChanged(), al que se le pasa como parámetro la propiedad que acaba de ser cambiada. Este método sirve para notificar al control contenedor (UserControl) de que se acaba de modificar una de sus propiedades, lo que desencadenará el evento WriteProperties() del control, en el que deberemos actualizar los valores de las propiedades del control, lo veremos más adelante.

Eventos del control

Una vez definida la propiedad, debemos hacer que el control note su presencia, es decir, debemos incluir el código necesario en los eventos del UserControl que asignan o recuperan el valor de las propiedades.

- InitProperties. Este evento es lanzado cada vez que se incluye el control en un formulario. Por lo que aquí situaremos la asignación de las propiedades, a sus valores por defecto.

```
Private Sub UserControl_InitProperties()
' este evento es llamado cada vez que
' se crea un control de este tipo
' en un formulario
' establecemos los valores por defecto
' para las propiedades del control
Set Fuente = Ambient.Font
mcFormatoDiaSem = mcDEFFormatoDiaSem
mcFormatoDia = mcDEFFormatoDia
mcFormatoMes = mcDEFFormatoMes
mcFormatoAño = mcDEFFormatoAño
mcSeparador = mcDEFSeparador
mcCaracterSumar = mcDEFCaracterSumar
mcCaracterRestar = mcDEFCaracterRestar
' si ejecutamos la aplicación, no mostrar
' nada en el control
If Ambient.UserMode Then
   txtValor.Text = ""
Else
   ' si estamos en modo diseño, mostrar una
   ' cadena en el control
   txtValor.Text = "##/##/####"
End If
TituloBotones
End Sub
```

Código fuente 380

Como particularidad destacable en este procedimiento, tenemos el uso de la propiedad Ambient, que contiene un objeto AmbientProperties. Este objeto, es el encargado de guardar las llamadas propiedades de ambiente, o conjunto de propiedades por defecto del control, como pueden ser el tipo de fuente, color de fondo, de primer plano, etc. En este caso, tomamos el fuente de ambiente y lo asignamos a la propiedad Fuente del control.

Otra propiedad muy importante de este control es UserMode, la cual utilizamos en este procedimiento para saber si quien está utilizando el control es el usuario de la aplicación o el programador, de forma que visualicemos una cadena en el texto del control.

Finalmente llamaremos al método TituloBotones() de este control, para asignar el Caption a los botones del mismo, como se muestra en el código fuente 381.

```
Public Sub TituloBotones()
' asignar a los botones del control
' el contenido de las propiedades
' usadas para describir el signo de
' suma y resta de fecha
cmdSumar.Caption = CaracterSumar
```

```
cmdRestar.Caption = CaracterRestar
End Sub
```

Código fuente 381

- ReadProperties. Este evento sucede cada vez que, en tiempo de diseño, se abre el formulario que contiene el control, al ejecutar la aplicación y al finalizarla. Su misión es leer los valores de la propiedades del control contenidas en el objeto PropertyBag que recibe como parámetro.

```
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
' este evento es llamado al ejecutar la aplicación
' tomar los valores actuales de las propiedades
' del control
Set moFuente = PropBag.ReadProperty("Fuente", Ambient.Font)
mcFormatoDiaSem = PropBag.ReadProperty("FormatoDiaSem", mcDEFFormatoDiaSem)
mcFormatoDia = PropBag.ReadProperty("FormatoDia", mcDEFFormatoDia)
mcFormatoMes = PropBag.ReadProperty("FormatoMes", mcDEFFormatoMes)
mcFormatoAño = PropBag.ReadProperty("FormatoAño", mcDEFFormatoAño)
mcSeparador = PropBag.ReadProperty("Separador", mcDEFSeparador)
mcCaracterSumar = PropBag.ReadProperty("CaracterSumar", mcDEFCaracterSumar)
mcCaracterRestar = PropBag.ReadProperty("CaracterRestar", mcDEFCaracterRestar)
' si ejecutamos la aplicación, no mostrar
' nada en el control
If Ambient.UserMode Then
   txtValor.Text = ""
Else
   ' si estamos en modo diseño, mostrar una
   ' cadena en el control
   txtValor.Text = "##/##/####"
End If
TituloBotones
FuenteControles
End Sub
```

Código fuente 382

Leemos las propiedades del control mediante el método ReadProperty() del objeto PropertyBag, que tiene la siguiente sintaxis:

```
ReadProperty(cPropiedad, cValorPredeterminado)
```

- cPropiedad. Cadena con el nombre de la propiedad a recuperar.

- cValorPredeterminado. Parámetro opcional, con el valor a recuperar en el caso de no exista un valor válido en cPropiedad.

El método FuenteControles() es empleado aquí, ya que en caso contrario, los controles no mostrarían la fuente adecuada. También se utiliza en el método de asignación de un nuevo valor a la propiedad Fuente, de forma que cada vez que cambiemos el tipo de fuente para el control, los controles constituyentes reflejen adecuadamente el cambio.

```
Public Sub FuenteControles()
' asignar el tipo de fuente del control
' a los controles constituyentes
Set txtValor.Font = moFuente
```

```
Set cmdSumar.Font = moFuente
Set cmdRestar.Font = moFuente
End Sub
```

Código fuente 383

- WriteProperties. Este evento sucede cada vez que se cierra un formulario en tiempo de diseño, al ejecutar la aplicación y al finalizarla. Su misión es escribir los valores de la propiedades del control en el objeto PropertyBag que recibe como parámetro.

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)

' este evento es llamado al ejecutar la aplicación
' escribir los nuevos valores en las propieades
' del control

PropBag.WriteProperty "Fuente", moFuente, Ambient.Font
PropBag.WriteProperty "FormatoDiaSem", mcFormatoDiaSem, mcDEFFormatoDiaSem
PropBag.WriteProperty "FormatoDia", mcFormatoDia, mcDEFFormatoDia
PropBag.WriteProperty "FormatoMes", mcFormatoMes, mcDEFFormatoMes
PropBag.WriteProperty "FormatoAño", mcFormatoAño, mcDEFFormatoAño
PropBag.WriteProperty "Separador", mcSeparador, mcDEFSeparador
PropBag.WriteProperty "CaracterSumar", mcCaracterSumar, mcDEFCaracterSumar
PropBag.WriteProperty "CaracterRestar", mcCaracterRestar, mcDEFCaracterRestar
End Sub
```

Código fuente 384

Escribimos las propiedades del control mediante el método WriteProperty() del objeto PropertyBag, que tiene la siguiente sintaxis:

```
WriteProperty(cPropiedad, xValor, cValorPredeterminado)
```

- cPropiedad. Cadena con el nombre de la propiedad a grabar.
- xValor. Valor a grabar en cPropiedad.
- cValorPredeterminado. Parámetro opcional, con el valor a grabar en el caso de que no exista un valor válido en cPropiedad.

En los procedimientos que acabamos de ver, es siempre conveniente proporcionar un valor predeterminado para asignar/recuperar las propiedades, ya que esto optimiza el tamaño del fichero que contiene el control.

Terminada la definición de los eventos del control, ya estamos en disposición de manipular las propiedades del mismo cuando lo incluyamos en un formulario, utilizando su ventana de propiedades, como vemos en la figura 421.

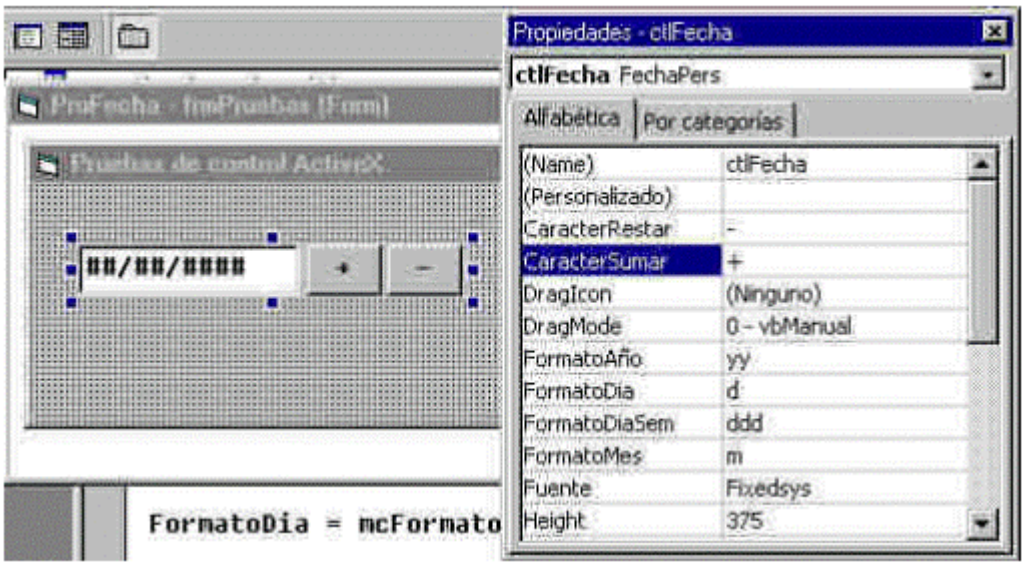


Figura 421. Ventana de propiedades del control ActiveX FechaPers, incluido en un formulario (modo diseño).

Páginas de propiedades

Una página de propiedades o PropertyPage, es una variante en la presentación de las propiedades de un control con respecto a la clásica ventana de VB, permitiendo la configuración del control en aspectos que mediante la típica ventana de propiedades serían muy difíciles o imposibles de conseguir.

Las propiedades dentro de una página de propiedades se suelen agrupar, aunque esto no es obligatorio, de forma que tengan una característica común, por ejemplo: todas las propiedades que hacen referencia a los colores de un control se agruparán bajo la misma página de propiedades. El conjunto de páginas creado será mostrado en una ventana que contendrá cada página en forma de ficha tabulada. Podremos cambiar de página simplemente pulsando la pestaña de cada página y modificar las propiedades contenidas en cada una de ellas. Para guardar los cambios, pulsaremos el botón Aceptar o Aplicar de esta ventana, o la tecla Enter. Si cambiamos de pestaña, las modificaciones efectuadas en la página activa antes del cambio serán guardadas automáticamente. Pulsaremos el botón Cancelar o la tecla Escape para abandonar las páginas de propiedades sin guardar los cambios.

La ventana que muestra las páginas de propiedades es creada automáticamente por VB, previa configuración por parte del programador en lo que respecta a las páginas que deberá incluir. La figura 422 muestra las páginas de propiedades para un control DBGrid.

Ilustraremos la creación de páginas de propiedades tomando como base el control FechaPers que estamos utilizando como ejemplo. La tabla 92 muestra las páginas de propiedades creadas y las propiedades que incluye cada una.

Nombre de página	**Propiedades**
Símbolos	CaracterSumar, CaracterRestar
Formato	FormatoDiaSem, FormatoDia, FormatoMes, FormatoAño, Separador

Tabla 92. Páginas de propiedades para el control FechaPers.

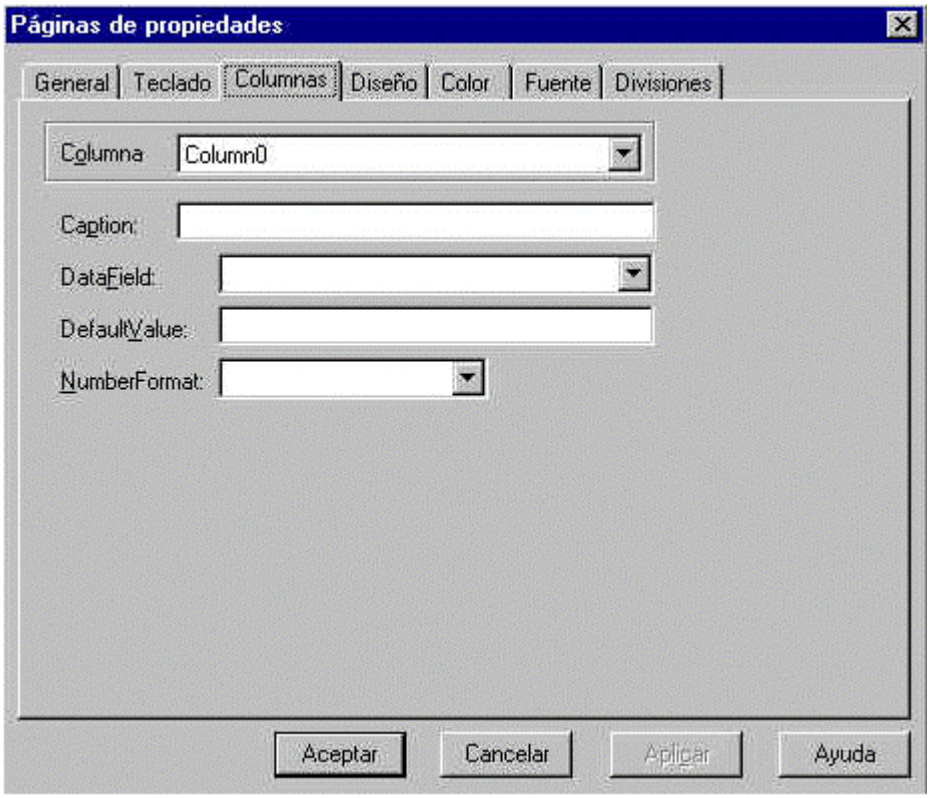


Figura 422. Páginas de propiedades para un control DBGrid.

- Interfaz de usuario de la página. Para crear una nueva página de propiedades, debemos estar situados en primer lugar, en el proyecto que contiene el control ActiveX al que vamos a añadir la página. A continuación, usaremos la opción del menú de VB Proyecto+Agregar página de propiedades, o el menú contextual del Explorador de proyectos, en su opción Agregar, para añadir una nueva página de propiedades. Esta acción creará una nueva página vacía o PropertyPage, que se asemeja a los formularios o controles base que ya hemos visto. Este tipo de elementos del programa se guardan en ficheros con extensión .PAG y .PGX. Aquí podremos situar los controles necesarios, para que el programador que use nuestro control, use las propiedades del control ActiveX. Este proceso es igual al seguido para incluir los controles constituyentes de un control o de un formulario. Veamos los pasos seguidos para la página Simbolos. Asignaremos a la propiedad Name del PropertyPage el nombre Simbolos, e incluiremos dos Label con los nombres de las propiedades a utilizar y dos TextBox en los que se editará el valor de estas propiedades. El resultado podemos verlo en la figura 423.

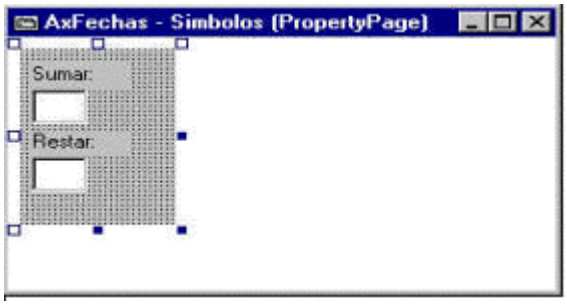


Figura 423. Página de propiedades Simbolos.

- Escribir el código de la página. Un objeto PropertyPage dispone de la propiedad Changed, a la que asignaremos el valor True cuando se modifique el valor de alguna propiedad. Mediante esta acción, la ventana que contiene la página, habilita el botón Aplicar para poder grabar las modificaciones de las propiedades en el control ActiveX.

  En la página Simbolos, cuando se cambie el valor de alguno de sus TextBox, mediante su evento Change(), notificaremos del suceso a la página, veámoslo en el código fuente 385.

```
Private Sub txtRestar_Change()
Changed = True
End Sub

' ------------------------

Private Sub txtSumar_Change()
Changed = True
End Sub
```

Código fuente 385

En esta página en concreto hemos utilizado controles TextBox para almacenar los valores de propiedad, pero si empleamos otro tipo de control, la forma de notificar el cambio a la página variará según el funcionamiento del control usado.

Para recuperar o grabar los valores de los controles de la página a las propiedades del control ActiveX, disponemos de los siguientes eventos que debemos codificar:

- SelectionChanged. Este evento se produce cuando la página es activada. Aquí deberemos traspasar el valor de las propiedades del control ActiveX a la página, igual que vemos en el código fuente 386.

```
Private Sub PropertyPage_SelectionChanged()
txtSumar.Text = SelectedControls(0).CaracterSumar
txtRestar.Text = SelectedControls(0).CaracterRestar
End Sub
```

Código fuente 386

- ApplyChanges. Cada vez que aceptamos los cambios en la página de propiedades de la forma antes explicada, se lanzará este evento, en el cual debemos incluir el código que haga el traspaso del contenido de los controles de la página, a las propiedades del control ActiveX, como se hace en el código fuente 387.

```
Private Sub PropertyPage_ApplyChanges()
SelectedControls(0).CaracterSumar = txtSumar.Text
SelectedControls(0).CaracterRestar = txtRestar.Text
End Sub
```

Código fuente 387

Un factor muy importante a tener en cuenta, de cara a este evento, es el hecho de que en VB podemos seleccionar varios controles de un formulario y modificar sus propiedades comunes. Para conseguir esto mismo en nuestros controles ActiveX, debemos realizar unos sencillos cambios en ApplyChanges(), manipulando la propiedad SelectedControls en la forma que se muestra en el código fuente 388.

```
Private Sub PropertyPage_ApplyChanges()
Dim loFechaP As FechaPers
For Each loFechaP In SelectedControls
  loFechaP.CaracterSumar = txtSumar.Text
  loFechaP.CaracterRestar = txtRestar.Text
Next
End Sub
```

Código fuente 388

La propiedad SelectedControls de una página de propiedades, contiene una colección con todos los controles seleccionados en el formulario, de forma que podamos realizar cambios globales sobre un grupo de controles al mismo tiempo.

- Enlazar la página al control. Terminada la creación de la página, hemos de conectarla al control, de forma que al desplegar el menú contextual de este, aparezcan las páginas al seleccionar la opción Propiedades de dicho menú.

  Para realizar esta tarea, abriremos la ventana de propiedades del UserControl y en la propiedad PropertyPages, pulsaremos el botón que contiene. Véase en la figura 424

  Al pulsar dicho botón, aparecerá el cuadro de diálogo Conectar páginas de propiedades, que nos permitirá como su nombre indica, conectar las páginas de propiedades existentes en el proyecto a nuestro control ActiveX. En este caso vamos a conectar sólo la página Símbolos (figura 425).

  Finalizada esta operación, ya podemos insertar el control en un formulario y manipular sus propiedades a través de la ventana de páginas, según muestra la figura 426.

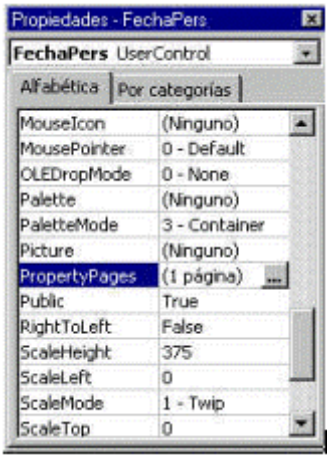


Figura 424. Ventana de propiedades de un control de usuario, con la propiedad PropertyPages seleccionada.

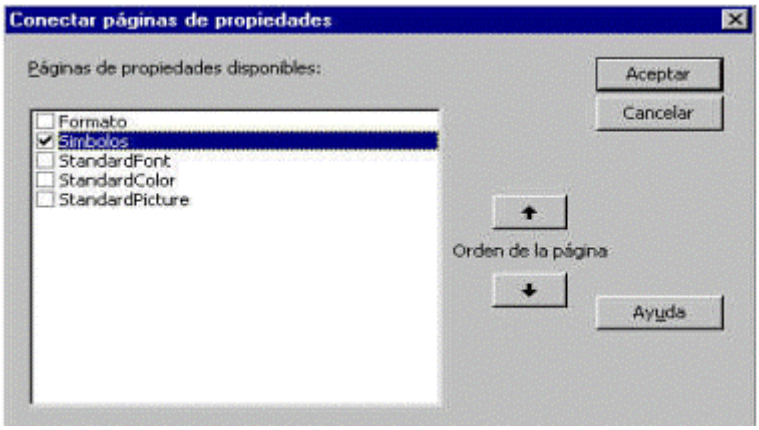


Figura 425. Cuadro de diálogo para conexión de páginas de propiedades a un control.

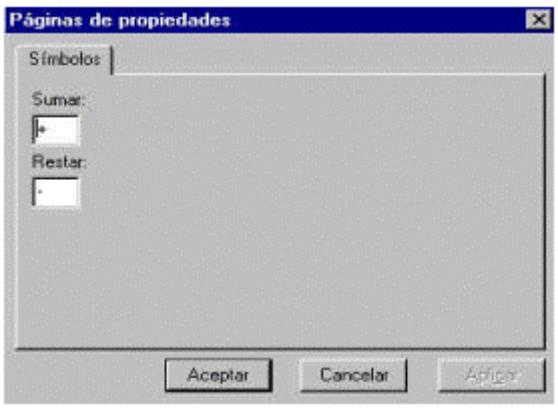


Figura 426. Página de propiedades para el control FechaPers.

En cuanto a las páginas que emplean otro tipo de controles para manejar las propiedades, en este ejemplo tenemos la página Formato, que usa varios ComboBox, para las propiedades que han de ocuparse del formato utilizado para mostrar la fecha.

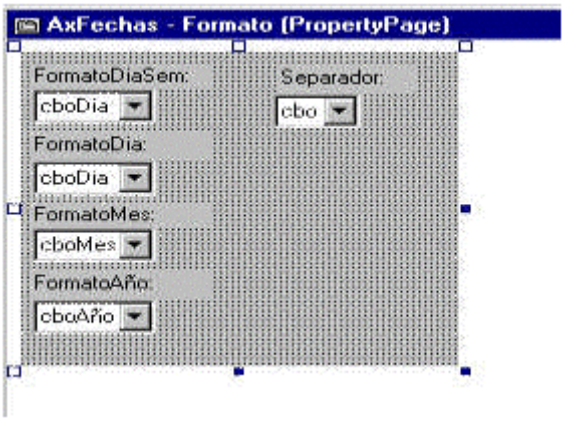


Figura 427. Página de propiedades Formato.

El hecho de que en esta página se usen todos los controles del mismo tipo es meramente casual, es posible hacer todas las combinaciones de controles dentro de la página que el creador del control ActiveX estime oportunas. Para notificar a la página un cambio en el control que contiene el valor de la propiedad, usaremos el evento Click() del ComboBox, tal y como muestra el código fuente 389 para la propiedad Separador.

```
Private Sub cboSeparador_Click()
Changed = True
End Sub
```

Código fuente 389

La forma de conectar la página al control es la misma que vimos anteriormente.

Configurar una página de propiedades estándar

Si nuestro control dispone de propiedades que se encarguen de manejar el tipo de fuente, color o imágenes, no es necesario construir una página para estas propiedades partiendo de cero. Tenemos una serie de páginas predefinidas en las que podemos situar tales propiedades de nuestro control, ahorrándonos una considerable cantidad de trabajo.

En este ejemplo, la propiedad Fuente del control FechaPers se adapta al perfil de propiedad que podemos incluir en una de estas páginas, ya que contiene un objeto de tipo Font, empleado para el tipo de fuente que han de mostrar los controles constituyentes del control ActiveX.

Una vez creado el código para esta propiedad en el módulo de código del control, seleccionaremos la propiedad PropertyPages, abriendo la ventana de páginas disponibles. Seleccionaremos la página StandardFont, de la forma vista anteriormente, y a continuación, mediante la opción Herramientas+Atributos del procedimiento, del menú de VB, abriremos la ventana de este mismo nombre, pulsando en su botón Avanzados para verla al completo.

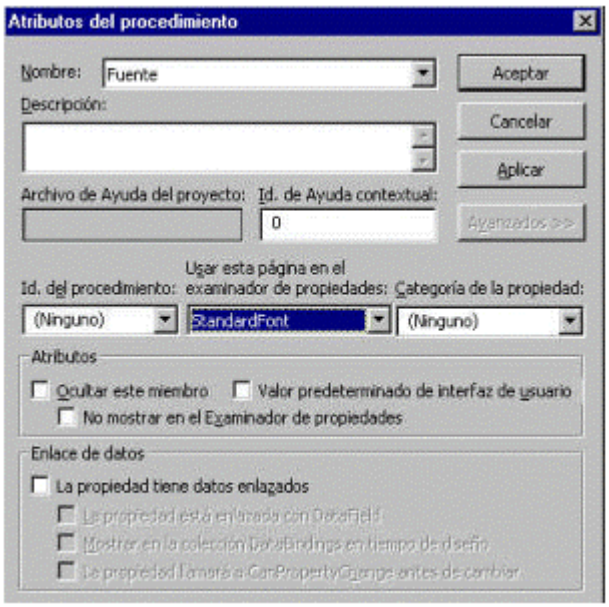


Figura 428. Atributos para la propiedad Fuente del control FechaPers.

Después de seleccionar en el campo Nombre la propiedad Fuente, iremos al campo Usar esta página en el examinador de propiedades, y seleccionaremos la página StandardFont. A partir de ahora, esta propiedad aparecerá en dicha página al consultar las propiedades de un control de este tipo.

Como punto final en la creación del control, podemos asignar a su propiedad ToolBoxBitmap un fichero de imagen, que sea usado por el Cuadro de herramientas para identificar el control. Para este control hemos usado el fichero Icon.GIF, incluido con este ejemplo.

Uso del control en un formulario

Ultimado el desarrollo del control, agregaremos un nuevo proyecto estándar al que llamaremos PruFecha, creando de esta manera un grupo de proyectos. El proyecto recién creado será usado para probar el control.

Configuraremos este proyecto para que se ejecute como inicial, y le agregaremos un formulario al que llamaremos frmPruebas. Ahora sólo resta insertar en el formulario el control ActiveX que acabamos de crear, tomándolo del Cuadro de herramientas. Le daremos el nombre ctlFecha y configuraremos las propiedades necesarias.

En lo que respecta al código a escribir para este control, ya que uno de los eventos disponibles para el mismo es DblClick(), le añadiremos el código fuente 390, para que muestre al usuario la fecha del campo de texto en letra.

```
Private Sub ctlFecha_DblClick()
MsgBox ctlFecha.Convertir, , "Fecha convertida a letra"
End Sub
```

Código fuente 390

Por último, sólo queda ejecutar la aplicación y manipular el control, para comprobar su funcionamiento.

Figura 429. Control ActiveX en ejecución.

Uso del control en una página web

Un control ActiveX se puede incluir en una página web, haciéndolo de esta manera, accesible a través de Internet. Junto a estos ejemplos, se incluye el fichero PAGINA.HTM, que contiene una copia del control para poder ejecutarlo con un navegador.

Veamos las operaciones necesarias para incluir este control en una página HTML. Si disponemos de una herramienta de creación de páginas web, la inserción de controles se simplifica en gran medida. Nosotros hemos empleado aquí FrontPage, pero cualquier otra utilidad de diseño para Internet será perfectamente válida.

En primer lugar ejecutamos FrontPage, iniciando una nueva página. Escribimos un sencillo título, y acto seguido, seleccionamos la opción de menú Insertar+Avanzadas+Control ActiveX, que visualiza un cuadro de diálogo para seleccionar uno de los controles ActiveX disponibles, registrados en el sistema, como podemos ver en la figura 430.

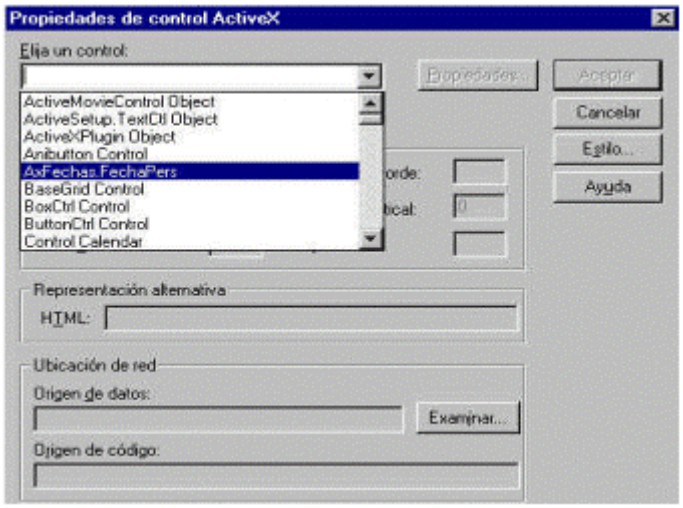


Figura 430. Diálogo para insertar un control ActiveX en una página de Internet.

Desplegamos el ComboBox Elija un control, y seleccionamos el que acabamos de crear: AxFechas.FechaPers. Dejaremos el resto de las propiedades con los valores por defecto y aceptaremos esta ventana. Tras grabar nuestro trabajo, podemos lanzar nuestro navegador y cargar en él la página que aparece en la figura 431.

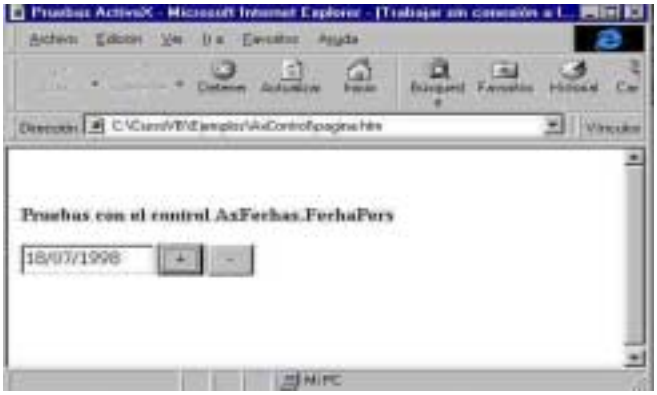


Figura 431. Control FechaPers ejecutándose en el navegador Internet Explorer.

Como podemos comprobar, incorporar un control propio, con una funcionalidad básica, a una página de Internet, es una tarea sencilla.

Asistente para la creación de controles

Entre los numerosos asistentes que incluye la actual versión de Visual Basic para facilitar nuestro trabajo, se encuentra este que nos permite crear la funcionalidad básica para un control ActiveX.

Antes de usarlo, hemos de crear la parte correspondiente al interfaz de usuario del control ActiveX, es decir, incluir todos los controles constituyentes que va a utilizar.

Para demostrar el funcionamiento de este asistente, vamos a emplearlo con el control FechaPers, con la particularidad de que hemos eliminado todo el código del control para comprobar los resultados obtenidos por el asistente.

Pondremos en marcha el asistente mediante la opción del menú de VB, Complementos+Asistente para interfax de control ActiveX, que nos mostrará la ventana de la siguiente figura. Cada vez que completemos una fase del asistente, pulsaremos el botón Siguiente para avanzar al próximo paso. Usaremos el botón Atrás, si necesitamos retroceder pasos durante la ejecución de esta utilidad, para modificar alguna de sus fases.

Figura 432. Introducción al Asistente para interfaz de control ActiveX.

Como primer paso, disponemos de dos listas para seleccionar los miembros del control: métodos, propiedades, eventos, etc. Una de estas listas, muestra los miembros disponibles, y la otra los seleccionados que se van a crear. En esta última se presentan un grupo de miembros por defecto (los más comunes), teniendo la posibilidad de agregar o quitar miembros de la lista que se creará.

Figura 433. Selección de miembros para el control.

Vamos a dejar los valores por defecto, pasando a la siguiente ventana del asistente. Ahora debemos indicar cuales son los miembros propios aparte de los que incorpora el control, es decir, los métodos y propiedades que vamos a crear para que el control se comporte conforme a nuestras necesidades.

Figura 434. Creación de miembros personalizados para el control.

Crearemos el método Convertir() y la propiedad CaracterSumar. No incluimos el resto, ya que está disponible en el ejemplo anterior y en este punto sólo queremos ilustrar el funcionamiento del asistente.

Después de pulsar Siguiente, veremos la fase correspondiente, en la que podemos seleccionar las rutinas de código pertenecientes al control y asignarlas a alguno de los controles constituyentes del control ActiveX, de forma que al usar dicho control, se ejecute el código de la rutina seleccionada.

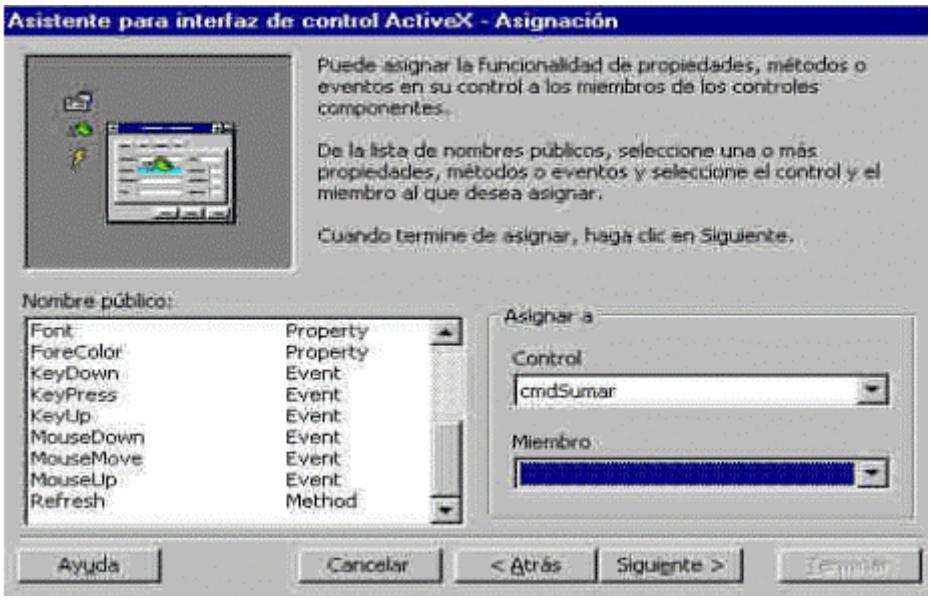


Figura 435. Asignación del código del control a los diferentes controles constituyentes.

El siguiente paso consiste en establecer los valores a los nuevos miembros del control o modificar los existentes. Seleccionaremos en la lista de miembros Nombre público, el procedimiento a modificar, para establecer sus valores en los diferentes controles agrupados en el marco Información de atributos. Adicionalmente, es posible incluir en el campo Descripción, un breve comentario sobre el trabajo del control, vemos un ejemplo en la figura 436.

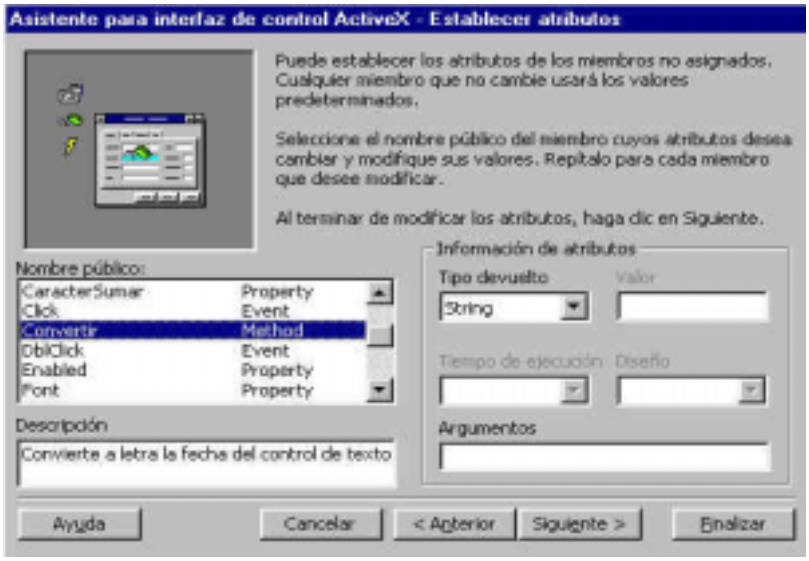


Figura 436. Asignación de atributos a los procedimientos del control.

Con el anterior paso, hemos completado las tareas a realizar para el control. Aparecerá la ventana final de este asistente para terminar la creación del control.

Figura 437. Ventana final del asistente de creación de controles.

Pulsando Terminar, se creará el código para el control según las instrucciones dadas a lo largo de las fases de esta utilidad. Como muestra, tenemos el código fuente 438, que contiene algunas de las rutinas creadas por el asistente.

```
'Zona de declaraciones del módulo de código
'del control
'Default Property Values:
Const m_def_BackColor = 0
Const m_def_ForeColor = 0
Const m_def_Enabled = 0
Const m_def_BackStyle = 0
Const m_def_BorderStyle = 0
Const m_def_CaracterSumar = "0"

'Property Variables:
Dim m_BackColor As Long
Dim m_ForeColor As Long
Dim m_Enabled As Boolean
Dim m_Font As Font
Dim m_BackStyle As Integer
Dim m_BorderStyle As Integer
Dim m_CaracterSumar As String
'Event Declarations:
Event Click()
Event DblClick()
Event KeyDown(KeyCode As Integer, Shift As Integer)
Event KeyPress(KeyAscii As Integer)
Event KeyUp(KeyCode As Integer, Shift As Integer)
Event MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
Event MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Event MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
'Código del control
'---------------------------------
Public Function Convertir() As String
End Function
'---------------------------------
Public Property Get CaracterSumar() As String
   CaracterSumar = m_CaracterSumar
End Property
Public Property Let CaracterSumar(ByVal New_CaracterSumar As String)
   m_CaracterSumar = New_CaracterSumar
   PropertyChanged "CaracterSumar"
End Property
'---------------------------------
' Inicializar propiedades para control de usuario
Private Sub UserControl_InitProperties()
   m_BackColor = m_def_BackColor
   m_ForeColor = m_def_ForeColor
   m_Enabled = m_def_Enabled
   Set m_Font = Ambient.Font
   m_BackStyle = m_def_BackStyle
   m_BorderStyle = m_def_BorderStyle
   m_CaracterSumar = m_def_CaracterSumar
End Sub
'---------------------------------
' Cargar valores de propiedades desde el almacenamiento
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
   m_BackColor = PropBag.ReadProperty("BackColor", m_def_BackColor)
   m_ForeColor = PropBag.ReadProperty("ForeColor", m_def_ForeColor)
   m_Enabled = PropBag.ReadProperty("Enabled", m_def_Enabled)
   Set m_Font = PropBag.ReadProperty("Font", Ambient.Font)
   m_BackStyle = PropBag.ReadProperty("BackStyle", m_def_BackStyle)
   m_BorderStyle = PropBag.ReadProperty("BorderStyle", m_def_BorderStyle)
```

```
   m_CaracterSumar = PropBag.ReadProperty("CaracterSumar", m_def_CaracterSumar)
End Sub
'-----------------------------------
' Escribir valores de propiedades en el almacenamiento
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
   Call PropBag.WriteProperty("BackColor", m_BackColor, m_def_BackColor)
   Call PropBag.WriteProperty("ForeColor", m_ForeColor, m_def_ForeColor)
   Call PropBag.WriteProperty("Enabled", m_Enabled, m_def_Enabled)
   Call PropBag.WriteProperty("Font", m_Font, Ambient.Font)
   Call PropBag.WriteProperty("BackStyle", m_BackStyle, m_def_BackStyle)
   Call PropBag.WriteProperty("BorderStyle", m_BorderStyle, m_def_BorderStyle)
   Call PropBag.WriteProperty("CaracterSumar", m_CaracterSumar,
m_def_CaracterSumar)
End Sub
```

Código fuente 391

Lógicamente, de nuestros procedimientos sólo se proporciona la declaración de los mismos vacía. Nosotros deberemos de encargarnos de escribir el código necesario para que realice su trabajo.

Asistente para páginas de propiedades

De igual forma que para crear el código base de un control, disponemos de un asistente que nos ayudará a crear las páginas de propiedades para ese control. Mediante la opción Complementos + Asistente para páginas de propiedades, del menú de VB, iniciaremos esta utilidad que nos ayudará mediante una serie de sencillos pasos. Para mostrar al lector su funcionamiento, crearemos con el asistente la página de propiedades Símbolos, que previamente hemos eliminado del proyecto. Esta página no se creará al completo, sólo se añadirá una propiedad, ya que simplemente lo haremos a efectos demostrativos, para que el lector compruebe el funcionamiento del asistente. La figura 438 muestra su pantalla introductoria.

Figura 438. Introducción al asistente para páginas de propiedades.

El siguiente paso, consiste en crear las páginas de propiedades para el control, o añadir nuevas páginas. Teniendo la posibilidad de cambiarles el nombre y alterar su orden de presentación.

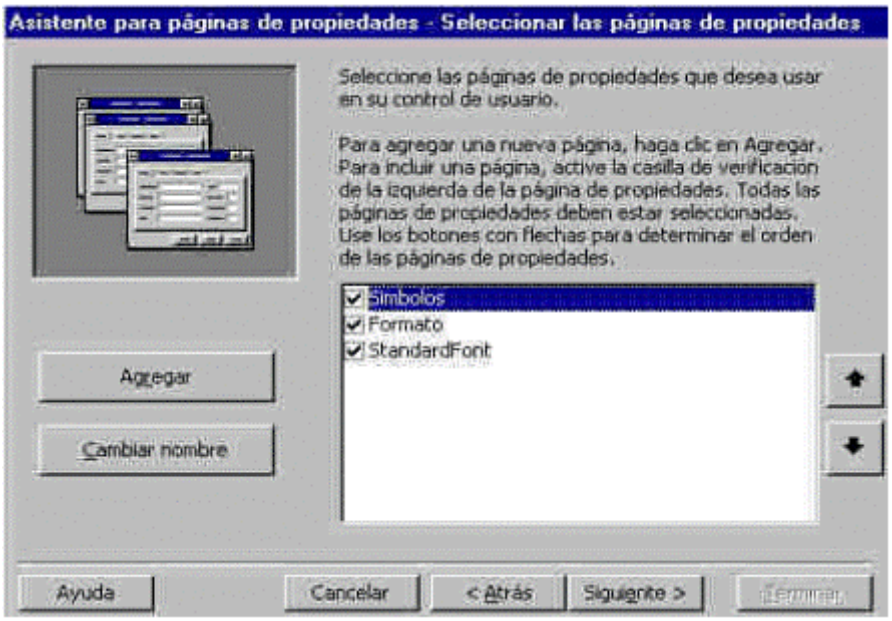


Figura 439. Creación de páginas y asignación de nombre a las mismas.

Como hemos comentado antes, crearemos la página Símbolos, situándola la primera del conjunto de páginas disponibles.

La siguiente acción a realizar, consiste en incorporar las propiedades a las diferentes páginas que hayamos definido para el control (figura 440).

Figura 440. Inserción de las propiedades del control en las diferentes páginas.

Finalmente, llegaremos a la pantalla de la siguiente figura, en la que habremos terminado este proceso. Pulsando Terminar, se creará el interfaz de usuario y el código de las páginas de propiedades.

Figura 441. Fin del asistente y creación del interfaz de usuario y código de las páginas.

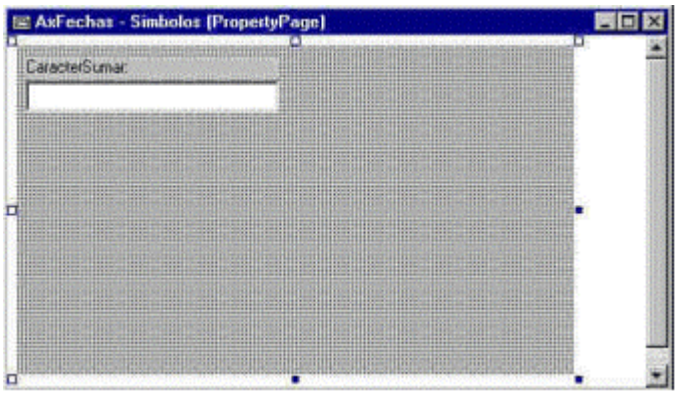


Figura 442. Página de propiedades creada con el asistente.

Este asistente crea las páginas, utilizando por defecto controles TextBox y Label para mostrar los valores de las propiedades. La figura 442 muestra el resultado de la página Símbolos, que acabamos de crear.

El código generado por el asistente, podemos verlo en el código fuente 392.

```
Private Sub txtCaracterSumar_Change()
    Changed = True
End Sub
'---------------------------------
Private Sub PropertyPage_ApplyChanges()
    SelectedControls(0).CaracterSumar = txtCaracterSumar.Text
End Sub
'---------------------------------
Private Sub PropertyPage_SelectionChanged()
    txtCaracterSumar.Text = SelectedControls(0).CaracterSumar
End Sub
```

Código fuente 392

Compilación y distribución

Una vez que hemos comprobado que el control funciona correctamente, es momento de compilarlo. Antes de dar este paso, hemos de considerar si queremos que el control incluya una clave de licencia o no. Para incluir dicha licencia, hemos de ir a las propiedades del proyecto y marcar la propiedad Solicitar clave de la licencia. Un control que incluya licencia podrá ser utilizado por otro programador en tiempo de diseño, en cualquier otro caso, el control sólo estará accesible por medio de las aplicaciones que hagan uso de él, sin posibilidad de usarlo en la fase de desarrollo de otra aplicación.

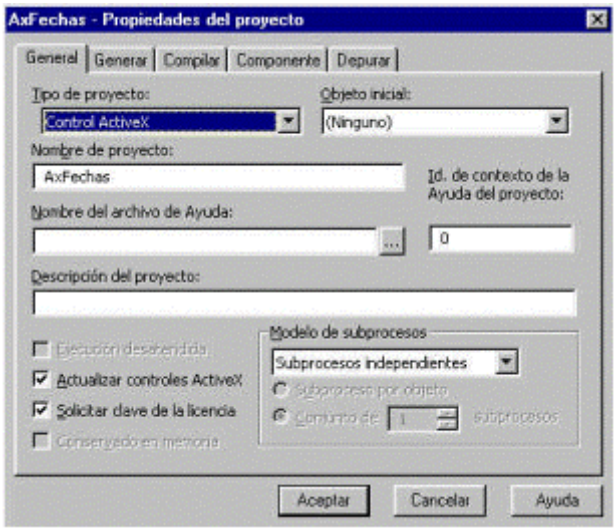


Figura 443. Propiedades del control ActiveX con la solicitud de licencia activada.

Determinado el aspecto referente a la licencia, es momento de compilar el control y crear un programa de instalación mediante el asistente de instalación de aplicaciones, para poder distribuirlo incluido como parte de una aplicación, para su uso por otros desarrolladores o para ser usado desde Internet.

Documentos ActiveX

Puesto que ActiveX es una tecnología orientada a Internet, y Visual Basic proporciona los instrumentos necesarios para manejarla, es lógico pensar que podemos utilizar VB en la creación de contenidos para Internet. De esta forma, es posible, empleando nuestro lenguaje habitual de programación, crear documentos ActiveX en nuestras aplicaciones. En este apartado, mostraremos la manera de hacer accesible desde Internet, un mantenimiento de datos creado en VB.

Documento ActiveX

Un Documento ActiveX es un elemento que contiene información de muy diverso tipo: texto, gráfico, hoja de cálculo, etc., que puede ser visualizada en un Contenedor de Documentos ActiveX, como por ejemplo Internet Explorer.

Servidor de Documentos ActiveX

Un Servidor de Documentos ActiveX no es otra cosa que un componente ActiveX DLL o EXE, que contiene uno o más documentos ActiveX, los cuales pone a disposición de una aplicación contenedora de documentos (navegador). En los ejemplos de este apartado, usaremos Internet Explorer como aplicación contenedora de documentos ActiveX.

En apartados anteriores, comentábamos como Excel es un componente ActiveX que actúa como servidor de objetos. Además de las cualidades antes mencionadas, es posible tomar un fichero que contenga una hoja de cálculo creada con Excel y abrirlo desde un contenedor de documentos.

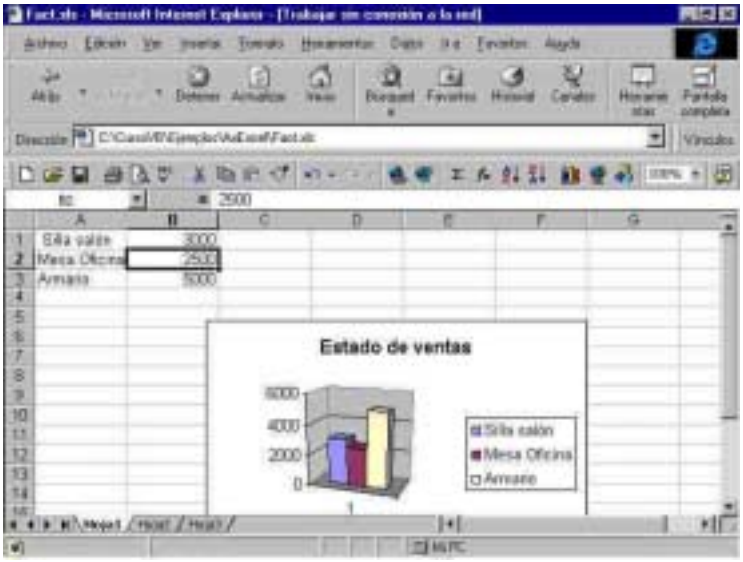


Figura 444. Edición de un documento Excel desde Internet Explorer.

Si iniciamos Internet Explorer y usamos la opción Archivo+Abrir de su menú, podemos abrir dentro de esta aplicación, el fichero FACT.XLS usado en uno de los anteriores ejemplos. La hoja de cálculo del fichero se abrirá dentro del navegador, iniciando al mismo tiempo Excel y realizándose una negociación de menús entre la aplicación servidora de documentos (Excel) y la contenedora (Internet Explorer). Recordemos que la negociación de menús consiste en que el menú de la aplicación servidora sustituye al menú de la aplicación cliente durante el periodo de tiempo que se produce la edición del objeto o documento proporcionado por el servidor. La figura 444, muestra la situación que acabamos de describir.

Fíjese el lector que en este caso, en la barra de tareas de Windows aparece Excel como tarea en ejecución, pero si la abrimos, veremos que está vacío, todas sus funciones se han transferido al navegador de Internet, siendo en este desde donde deberemos trabajar con la hoja de cálculo.

Creación de un Servidor de Documentos ActiveX

Como se ha explicado anteriormente, un servidor de Documentos ActiveX es un componente ActiveX DLL o EXE, que contiene documentos ActiveX. Vamos a mostrar las fases de creación de un componente de este tipo utilizando el proyecto *AxDocum* incluido con los ejemplos de este curso. Este proyecto se basa en parte del ejemplo utilizado anteriormente en el proyecto AxMant, en el apartado de creación de componentes de código ActiveX. En concreto, se trata de la parte del mantenimiento de

la tabla de Clientes, pero con algunas modificaciones para adaptarlo a las características propias de este ejemplo.

Se ha cambiado el nombre de la base de datos por AXDOCUM.MDB, que ahora sólo contiene la tabla Clientes. En cuanto a la clase Cliente, el método AbrirDatos(), ahora no solicita la ruta de datos al usuario, emplea una cadena con la ruta completa de la base de datos. El motivo es que si empleamos esta aplicación desde un servidor web, lo lógico es que dicho servidor ya conozca la ubicación de la base de datos, encargándose de proporcionar los datos al usuario, junto con el documento para manipularlos. El código fuente 393, muestra el método que acabamos de comentar. En cuanto a la ruta de la base de datos, el lector deberá emplear una igual o modificar el fuente para adaptarla al directorio que contenga este ejemplo.

```
Private Function IEditar_AbrirDatos() As Boolean

On Error GoTo ControlErrores

IEditar_AbrirDatos = False

' crear conexión
Set macnAxDocum = New ADODB.Connection
macnAxDocum.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=C:\CursoVB6\Texto\ActiveX\AxDocum\AxDocum.mdb"
macnAxDocum.Open

' crear recordset
Set marsClientes = New ADODB.Recordset
marsClientes.CursorLocation = adUseClient
marsClientes.Open "Clientes", macnAxDocum, adOpenDynamic, _
   adLockOptimistic, adCmdTable

' establecer orden
marsClientes("CodCli").Properties("Optimize") = True
marsClientes.Sort = "CodCli ASC"

IEditar_AbrirDatos = True

Exit Function
' -------------------------
ControlErrores:

MsgBox "Nº error: " & Err.Number & vbCrLf & _
   "Descripción: " & Err.Description, , "Error"

End Function
```

Código fuente 393

- Crear el proyecto para el servidor de documentos. Si partimos desde cero en la creación del proyecto. Crearemos un nuevo proyecto en Visual Basic de tipo DLL Documento ActiveX o bien EXE de Documento ActiveX.

Figura 445. Ventana para la creación de un proyecto servidor de documentos ActiveX.

Lo que haremos con esta operación, será crear un nuevo proyecto EXE ActiveX o DLL ActiveX, que son los que se usan para crear componentes ActiveX, y son los únicos que admiten documentos ActiveX. La diferencia entre crear el proyecto de esta manera o crear un proyecto para componente ActiveX DLL o EXE directamente, reside en que haciéndolo de esta forma, el proyecto ya incluye un documento ActiveX por defecto. El proyecto AxDocum de nuestro ejemplo se trata de un proyecto ActiveX DLL.

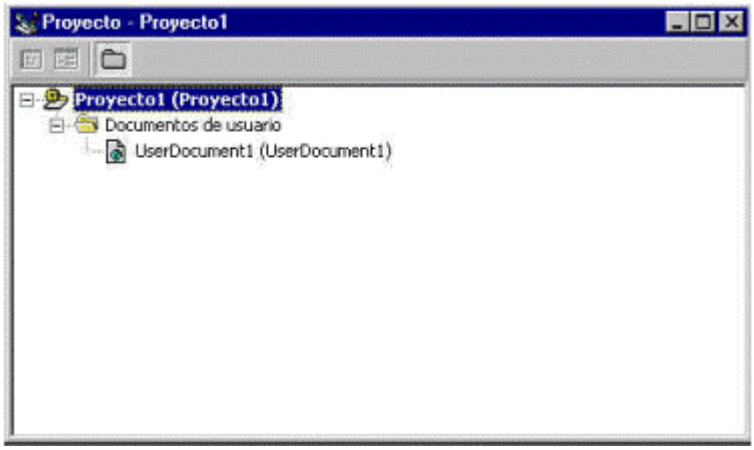


Figura 446. Proyecto servidor de documentos ActiveX con un documento por defecto.

También es posible usar un proyecto ya existente de tipo EXE ActiveX o DLL ActiveX, o en último caso, si el proyecto al que debemos incluirle un documento ActiveX es de tipo estándar, convertirlo previamente a ActiveX DLL o EXE, e incorporarle el documento después. La figura 446 muestra el Explorador de proyectos, después de crear el proyecto para manejo de documentos.

Si hemos de establecer alguna propiedad al proyecto, procederemos de igual forma que la explicada en el manejo de componentes ActiveX.

- Diseño del documento ActiveX. El documento ActiveX o UserDocument del proyecto consiste en un elemento o plantilla, que sirve de base para la introducción de controles, de igual manera que en un formulario situamos controles o en un UserControl de un control ActiveX.

Una de las grandes diferencias entre un UserDocument y un Form, la encontramos en que el documento no utiliza los eventos Load() y Unload() para iniciarse y terminar, en su lugar se emplean Initialize() y Terminate(), en los cuales hemos de situar el código para las tareas de inicio y fin del documento.

Si necesitamos añadir documentos al proyecto, emplearemos la opción Proyecto+Agregar documento de usuario, o bien la opción Agregar del menú contextual del Explorador de proyectos. Los documentos se guardan en ficheros con la extensión .DOB y .DOX.

Estos documentos son los que realizarán la labor de interfaz de usuario, una vez cargados en el contenedor de documentos.

Para este ejemplo, crearemos un UserDocument, al que llamaremos docClientes, y que será el encargado de realizar el mantenimiento de datos de la tabla Clientes. La figura 447 muestra este documento una vez incluidos todos los controles.

Figura 447. Documento ActiveX docClientes.

Como puede comprobar el lector, es prácticamente igual que el formulario empleado en el ejemplo que realizaba el mismo mantenimiento de la tabla Clientes, con la excepción de que aquí incluimos un control Label, que se mostrará al usuario cuando exista un problema al cargar el documento en el navegador, mientras tanto, permanecerá oculto.

En el código fuente 394 incluimos el código que se ejecuta al iniciar y terminar el documento ActiveX. Para cualquier aspecto que necesitemos controlar en el documento, deberemos codificar los eventos del objeto UserDocument, incluidos en el módulo de código de dicho objeto. El resto de procedimientos para la edición, navegación, etc., por la tabla de datos, es el mismo que en el ejemplo anterior, por lo que no se volverán a repetir aquí.

```
Private Sub UserDocument_Initialize()
' carga del documento
' instanciar un objeto Cliente
Set moCliente = New Cliente
If moCliente.AbrirDatos Then
```

```
   cmdPrincipio.Value = True
   Set datClientes.Recordset = moCliente.Lista
Else
   ' si hay problemas al acceder a la tabla
   ' deshabilitar controles del documento
   ' y mostrar aviso de error de carga
   txtCodCli.Enabled = False
   txtNombre.Enabled = False
   txtDireccion.Enabled = False
   txtCiudad.Enabled = False
   txtNIF.Enabled = False
   cmdAvanzar.Enabled = False
   cmdRetroceder.Enabled = False
   cmdPrincipio.Enabled = False
   cmdFin.Enabled = False
   cmdAgregar.Enabled = False
   cmdGrabar.Enabled = False
   grdClientes.Enabled = False
   lblError.Visible = True
End If
End Sub
' ------------------------------------
Private Sub UserDocument_Terminate()
' finalizar la ejecución del documento,
' descargar el objeto utilizado por el
' documento
Set moCliente = Nothing
End Sub
```

Código fuente 394

Terminada la creación del UserDocument, podemos dar por finalizada la etapa de creación del proyecto. Ahora hemos de comprobar su funcionamiento desde un navegador.

Visualización de un documento ActiveX

Finalizado el desarrollo del servidor de documentos, necesitamos comprobar si el documento que acabamos de crear se ejecuta correctamente. Para ello, debemos abrir la ventana de propiedades del proyecto y en la pestaña Depurar, marcar el OptionButton Iniciar el componente, y seleccionar el documento que acabamos de crear del ListBox adjunto. También deberemos marcar el CheckBox Usar explorador existente, para que el documento al ser ejecutado, pueda visualizarse en el contenedor correspondiente, en este caso Internet Explorer.

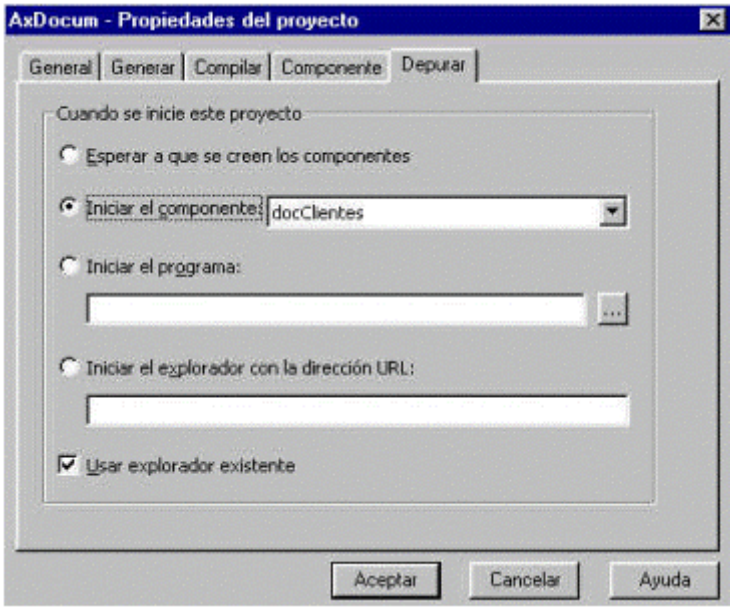


Figura 448. Opciones de depuración para un proyecto con documentos ActiveX.

Al ejecutar la aplicación, se iniciará Internet Explorer cargando el UserDocument docClientes, como podemos comprobar en la figura 449.

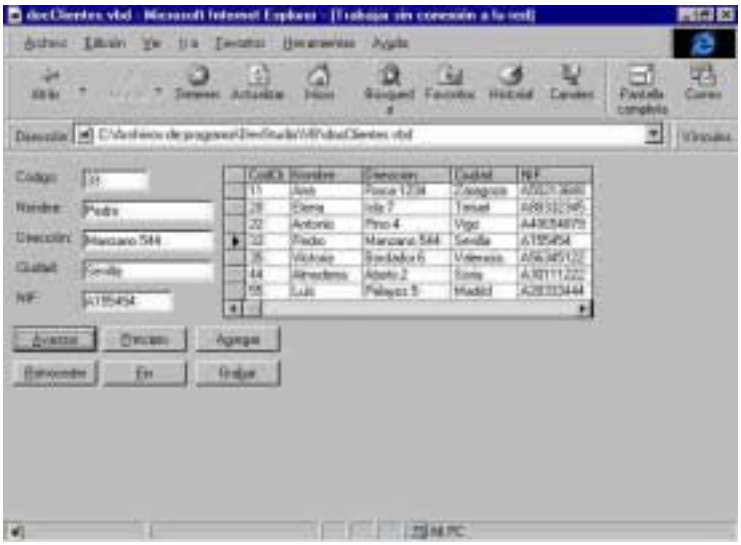


Figura 449. Internet Explorer mostrando el documento ActiveX creado con Visual Basic.

Una vez cargado el documento en el navegador, podremos usarlo como si fuera un formulario normal, para realizar el mantenimiento de datos de la tabla Clientes.

Compilar el proyecto

Finalizadas todas las pruebas y ajustes en el código, procederemos a compilarlo. Este proceso nos creará la DLL y demás ficheros necesarios de este proyecto. Adicionalmente se creará un fichero

.VBD por cada UserDocument existente en el proyecto; estos ficheros contienen toda la información necesaria para que la aplicación contenedora de documentos los visualice.

Distribución de la aplicación servidora y documentos

Tras la compilación, procederemos a utilizar el asistente para empaquetado y distribución de aplicaciones, con el que crearemos un paquete para su distribución en Internet.

En el primer paso de este asistente, seleccionaremos el fichero que contiene el proyecto que acabamos de terminar, y pulsaremos el botón Empaquetar.

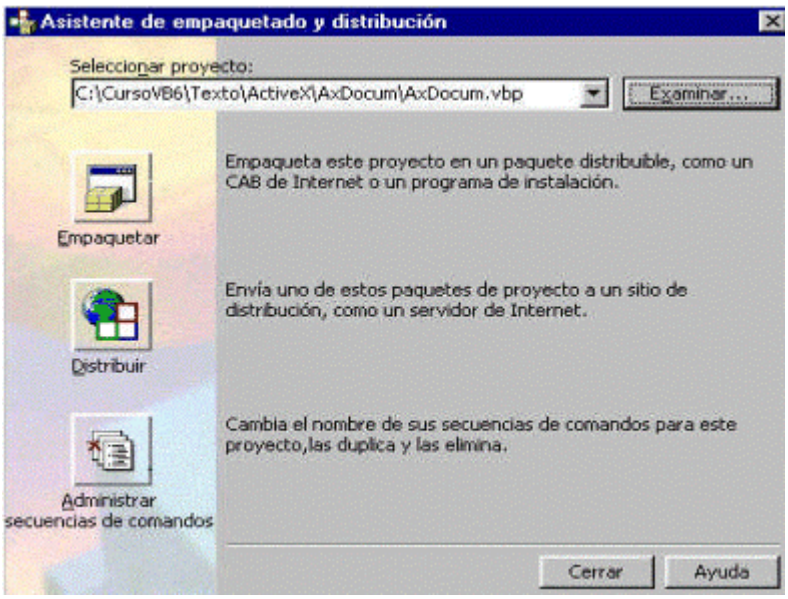


Figura 450. Inicio del asistente de empaquetado.

Continuaremos seleccionando el tipo de empaquetado: Paquete de Internet.

Figura 451. Selección del tipo de empaquetado.

Seguidamente debemos elegir la carpeta en la que se depositarán los ficheros generados por este asistente; en el caso de que no exista, será creada de forma automática.

Figura 452. Selección de la carpeta contenedora del paquete.

Debido a que este proyecto utiliza DAO, se muestra una lista con los controladores de datos disponibles, para elegir los que necesitemos.

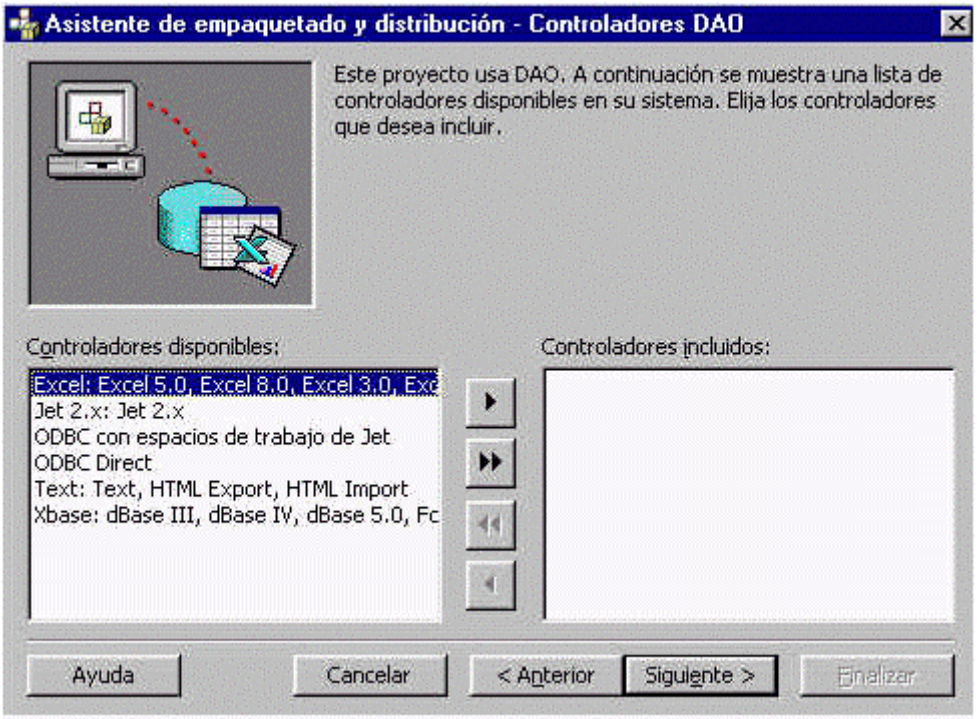


Figura 453. Selección de controladores de datos.

En el siguiente paso, se muestran los ficheros que el asistente ha determinado necesarios para empaquetar. Si existe algún otro fichero adicional, que necesitemos incluir, debemos hacerlo en este punto, pulsando el botón Agregar e indicando el fichero o ficheros a añadir.

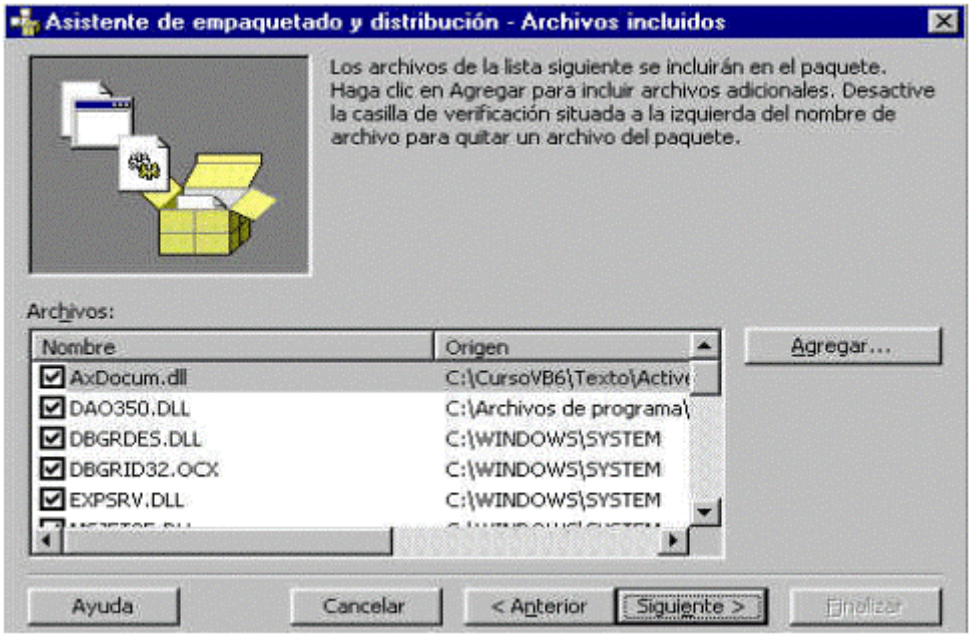


Figura 454. Agregar ficheros al paquete.

Es posible establecer ficheros situados en direcciones de Internet para que sean descargados en el momento de la instalación del paquete. En este paso, debemos indicar tales ficheros y sitios de Internet en los que residen.

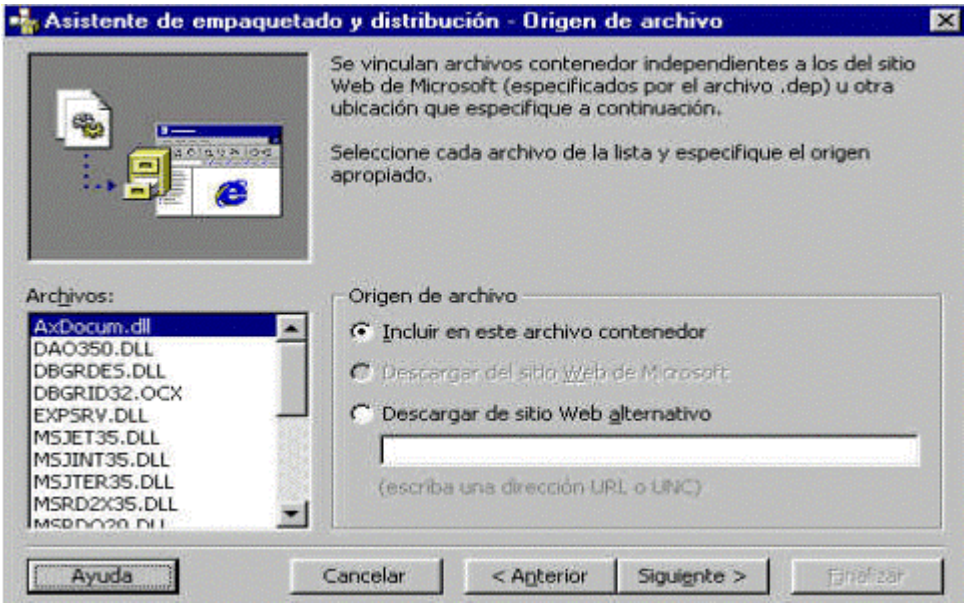


Figura 455. Indicar origen de ficheros a instalar.

En el siguiente paso, debemos establecer el nivel de seguridad que tienen los componentes de la aplicación.

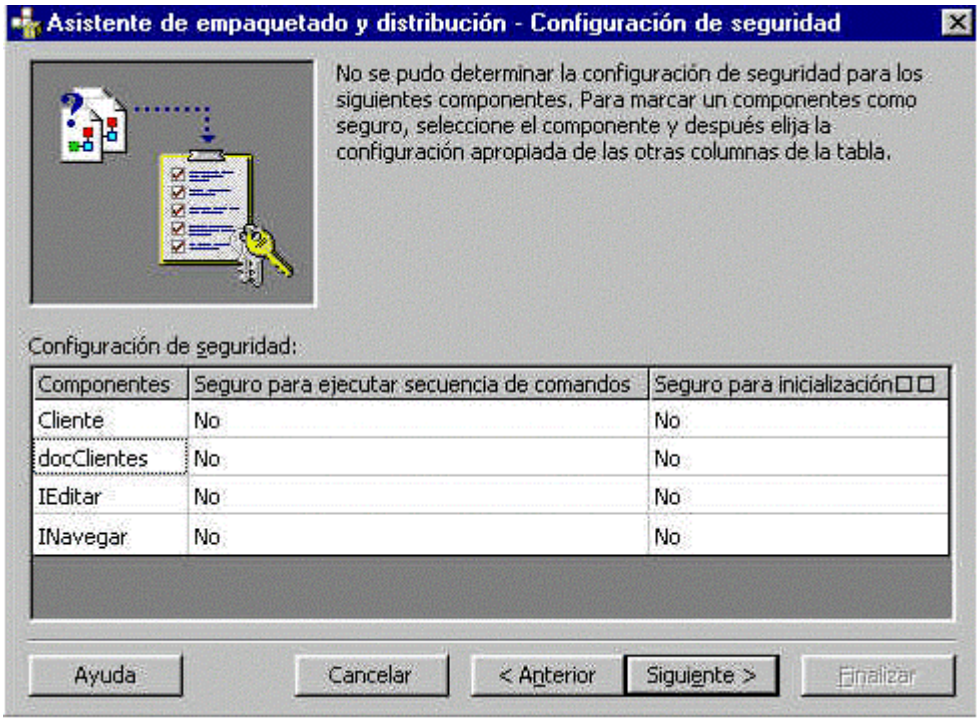


Figura 456. Establecer nivel de seguridad para los componentes.

Llegados a este punto, se ha terminado de recopilar la información necesaria para el paquete. Tan sólo queda dar un nombre a esta sesión del asistente y pulsar Finalizar (figura 457), creándose los ficheros .CAB y otros necesarios para la posterior distribución.

Figura 457. Final del proceso de empaquetado.

Un fichero .CAB o cabinet, contiene comprimidos, un conjunto de componentes necesarios para una página web. La estructura de este tipo de ficheros, está diseñada especialmente para su descarga en Internet, ya que al estar su contenido comprimido, se agiliza dicha descarga.

Los componentes de una aplicación pueden comprimirse en varios ficheros .CAB, de forma que existan ficheros que sea necesario descomprimir y otros no, en función de si necesitan actualizar o no ciertos componentes.

Otro de los usos que se le dan a este tipo de ficheros, consiste en la instalación de aplicaciones. Existiendo un extenso grupo de aplicaciones que usa este sistema de instalación.

Terminado este proceso, volveremos a la pantalla principal del asistente, en la que deberemos pulsar el botón Distribuir, e indicar los ficheros y servidor web desde el que vamos a realizar la distribución de la aplicación en Internet.

Entre los ficheros creados, encontraremos un .HTM con un hipervínculo que realizará la carga del documento ActiveX.

12

Tratamiento avanzado de datos con ADO

Clases orientadas a datos

Una de las nuevas características incorporadas en Visual Basic 6 en cuanto al manejo de datos, consiste en la posibilidad de hacer que una clase se convierta en un origen o receptor de datos, o como se denominan en ADO, proveedor o consumidor de datos, se dice en este caso, que la clase está orientada a datos (Data-Aware Class).

Proveedor de datos

Una clase que actúa como proveedor de datos, se encarga de realizar una conexión con una fuente de datos y proporcionar los mismos a otros objetos en una aplicación. Dicha clase puede disponer de interfaz de usuario, pero no es obligatorio. Un ejemplo lo constituye el control Data.

Otra de las ventajas de este tipo de clases consiste en que podemos hacer que los datos proporcionados no residan en un formato preestablecido de base de datos, es posible, por ejemplo, tomar la información de un fichero de texto.

Mediante la nueva propiedad DataSourceBehavior, incorporada a los módulos de clase, podemos establecer nuestra clase como origen de datos, asignándole el valor vbDataSource, y por supuesto, escribiendo el código necesario para proporcionar los datos.

Consumidor de datos

Una clase que actúa como consumidor de datos, se encarga de exponer los datos que le proporciona un proveedor, y al igual que este, puede tener o no interfaz de usuario.

El control TextBox es el ejemplo más conocido de consumidor de datos, visualizando un campo de una fila que le proporcione un control Data u otra clase proveedora.

La propiedad DataBindingBehavior, también nueva en los módulos de clase, nos sirve para indicar el tipo de enlace que se va a establecer en una clase consumidora con respecto a una proveedora. Si el valor es vbSimpleBound, el enlace se establecerá con un único campo del objeto proveedor de datos; en el caso de que esta propiedad valga vbComplexBound, el enlace se hará hacia una fila o registro completo del proveedor.

BindingCollection

Pero de nada nos sirve un objeto proveedor de datos y otro consumidor, sin algo que los conecte. Y aquí es donde aparece el objeto BindingCollection, que dicho de forma simple, es un conjunto de enlaces entre un objeto proveedor de datos y uno o varios objetos consumidores de datos. Los consumidores de datos, pueden ser tanto objetos de código, como controles que hayamos incluido en un formulario.

En los siguientes apartados, veremos algunos ejemplos de creación y manejo de clases orientadas a datos.

Crear una clase proveedora de datos

Seguidamente veremos los pasos necesarios para desarrollar la aplicación de ejemplo ClsDatDB, en la cual crearemos una clase que proporcionará información de la base de datos MUSICA.MDB, ya vista en anteriores ejemplos.

Agregaremos al proyecto en primer lugar, un módulo de clase, al que daremos el nombre Grabaciones, y estableceremos en su propiedad DataSourceBehavior el valor vbDataSource. Cono se muestra en la figura 458.

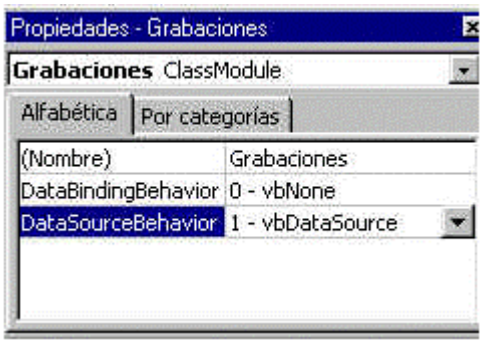


Figura 458. Establecer una clase como proveedor de datos.

Abriendo su ventana de código, declararemos las variables que aparecen en el código fuente 395 a nivel de módulo, que contendrán la conexión, el conjunto de registros y un valor para averiguar si hemos llegado al final del Recordset.

```
Option Explicit

Private macnConexion As ADODB.Connection
Private marsGrabaciones As ADODB.Recordset
Private mbEOF As Boolean
```

Código fuente 395

En el evento Initialize() de la clase, es donde deberemos escribir el código que realiza la apertura del Recordset que mostrará los datos, como muestra el código fuente 396.

```
Private Sub Class_Initialize()
' agregar la fuente de datos a la
' lista de miembros de datos
DataMembers.Add "Grabaciones"
' instanciar conexion
Set macnConexion = New ADODB.Connection
' establecer cadena de conexión
macnConexion.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=C:\CursoVB6\Texto\DatosADO\ClsDatDB\musica.mdb"
' abrir conexión
macnConexion.Open
' crear el recordset y asignarle la conexión
Set marsGrabaciones = New ADODB.Recordset
Set marsGrabaciones.ActiveConnection = macnConexion
' asignar al recordset la sentencia SQL
' para recuperar datos
marsGrabaciones.Source = "SELECT * FROM Grabaciones " & _
   "WHERE Titulo >= 'T' AND Titulo <= 'U' " & _
   "ORDER BY Titulo"
' abrir recordset
marsGrabaciones.Open , , adOpenForwardOnly, adLockOptimistic, adCmdText
Me.EOF = False
End Sub
```

Código fuente 396

La operación de añadir el nombre del origen de datos a la colección DataMembers, será comentada en un apartado posterior. La ruta de la base de datos, a diferencia de otros ejemplos, no se toma del Registro de Windows; en su lugar se emplea una ruta fija, por lo que el lector deberá crearla también y depositar en ella este proyecto, o modificarla en el código.

En el código fuente 397, crearemos dos procedimientos Property para la propiedad EOF, de manera que sepamos desde el exterior de la clase, cuando hemos llegado al final de los registros; también escribiremos un procedimiento para avanzar al siguiente registro y otro para mostrar el valor del campo del conjunto de registros obtenidos.

```
Public Property Get EOF() As Boolean
EOF = mbEOF
End Property
'--------------------------------------------
```

```
Public Property Let EOF(ByVal vbEOF As Boolean)
mbEOF = vbEOF
End Property
'-------------------------------------------
Public Sub MoveNext()
marsGrabaciones.MoveNext
If marsGrabaciones.EOF Then
   Me.EOF = True
End If
End Sub
'-------------------------------------------
Public Sub Mostrar()
MsgBox marsGrabaciones("Titulo"), , "Título de la grabación"
End Sub
```

Código fuente 397

Tan sólo queda ya, escribir el código del procedimiento Main(), código fuente 398, para este programa, que será el encargado de instanciar un objeto de la clase Grabaciones y visualizar la información que como proveedor de datos, nos proporciona.

```
Public Sub Main()
Dim loGrabaciones As Grabaciones
' instanciar un objeto de la clase
' orientada a datos
Set loGrabaciones = New Grabaciones
' recorrer los registros del recordset
' que contiene la clase, mostrando
' el valor de un campo
Do While Not loGrabaciones.EOF
   loGrabaciones.Mostrar
   loGrabaciones.MoveNext
Loop
Set loGrabaciones = Nothing
End Sub
```

Código fuente 398

Crear una clase consumidora de datos

Y como es lógico, vista la creación de un proveedor de datos, no podemos pasar por alto el modo de crear un consumidor de los mismos, para lo cual se facilita el proyecto ClsConsumer. La aplicación desarrollada aquí, contiene como clase proveedora Grabaciones, la misma del anterior ejemplo, agregaremos una nueva clase que llamaremos Consumidor, y que se encargará de mostrar la información que le suministre Grabaciones.

Nada más agregar la clase Consumidor, debemos asignar a su propiedad DataBindingBehavior el valor vbSimpleBound, como se muestra en la figura 459, ya que el enlace con el proveedor se establecerá campo a campo.

El código que escribiremos para esta clase consistirá en procedimientos de propiedad; cada conjunto de procedimientos Get/Let corresponderán a un campo de los devueltos por el objeto proveedor al que se enlazarán los objetos instanciados de esta clase. En el código fuente 399 se muestra el fuente de la clase Consumidor.

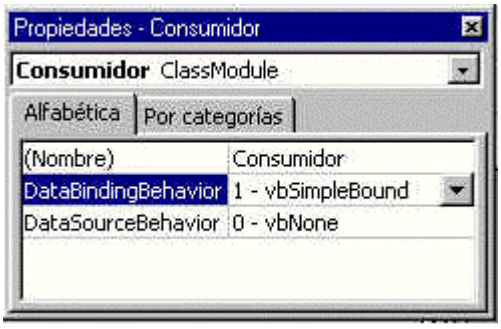


Figura 459. Establecer una clase como consumidor de datos.

```
Option Explicit
' en esta clase creamos los procedimientos de
' propiedad, que van a ser enlazados con los
' campos de un objeto proveedor de datos
Private mcClaveDisco As String
Private mcClaveInterprete As String
Private mcNombreDisco As String
Private mcFormatoDisco As String
Public Property Get ClaveDisco() As String
ClaveDisco = mcClaveDisco
End Property
Public Property Let ClaveDisco(ByVal vcClaveDisco As String)
mcClaveDisco = vcClaveDisco
End Property
'--------------------------------------
Public Property Get ClaveInterprete() As String
ClaveInterprete = mcClaveInterprete
End Property
Public Property Let ClaveInterprete(ByVal vcClaveInterprete As String)
mcClaveInterprete = vcClaveInterprete
End Property
'--------------------------------------
Public Property Get NombreDisco() As String
NombreDisco = mcNombreDisco
End Property
Public Property Let NombreDisco(ByVal vcNombreDisco As String)
mcNombreDisco = vcNombreDisco
End Property
'--------------------------------------
Public Property Get FormatoDisco() As String
FormatoDisco = mcFormatoDisco
End Property
Public Property Let FormatoDisco(ByVal vcFormatoDisco As String)
mcFormatoDisco = vcFormatoDisco
End Property
```

Código fuente 399

Los nombres de estas propiedades son deliberadamente diferentes a los nombres de los campos del proveedor; de forma que podamos demostrar que no es necesario que se correspondan para poder establecer el enlace.

Y ya por último, sólo resta escribir el código que realiza la conexión entre el proveedor y el consumidor de datos. El lugar para ello será el procedimiento Main() del programa, cuyas líneas de código vemos en el código fuente 400.

```
Public Sub Main()
Dim loGrabaciones As Grabaciones
Dim loConsumidor As Consumidor
Dim lbndEnlace As BindingCollection
' instanciar los objetos para realizar
' el enlace de datos
Set loGrabaciones = New Grabaciones  ' proveedor de datos
Set loConsumidor = New Consumidor  ' consumidor de datos
Set lbndEnlace = New BindingCollection  ' enlace de datos
' indicar al objeto de enlace, cual va a ser
' el objeto que va a proporcionar los datos
Set lbndEnlace.DataSource = loGrabaciones
' establecer el enlace entre las propiedades
' del objeto consumidor y el proveedor,
' por cada campo del consumidor habrá que añadir
' un enlace al objeto BindingCollection,
' el formato del enlace es:
' - objeto consumidor de datos
' - propiedad del consumidor a enlazar
' - nombre del campo del proveedor que enviará
'   el valor del campo del registro actual a la
'   propiedad del consumidor
lbndEnlace.Add loConsumidor, "ClaveDisco", "CodGrabacion"
lbndEnlace.Add loConsumidor, "ClaveInterprete", "CodAutor"
lbndEnlace.Add loConsumidor, "NombreDisco", "Titulo"
lbndEnlace.Add loConsumidor, "FormatoDisco", "Soporte"
' recorrer el proveedor, mostrando sus registros
' mediante el objeto consumidor
Do While Not loGrabaciones.EOF
   MsgBox "ClaveDisco: " & loConsumidor.ClaveDisco & vbCrLf & _
      "ClaveInterprete: " & loConsumidor.ClaveInterprete & vbCrLf & _
      "NombreDisco: " & loConsumidor.NombreDisco & vbCrLf & _
      "FormatoDisco: " & loConsumidor.FormatoDisco _
      , , "Información del proveedor mostrada por el consumidor"

   loGrabaciones.MoveNext
Loop
' liberar recursos ocupados por los objetos
Set loGrabaciones = Nothing
Set loConsumidor = Nothing
Set lbndEnlace = Nothing
End Sub
```

Código fuente 400

Ejecutando la aplicación, el lector podrá comprobar como se muestran los campos de los registros en el objeto proveedor, a través de las propiedades del objeto consumidor.

Proveedor de datos con la información en un formato distinto

La aplicación anterior puede servir perfectamente para mostrar la forma básica de crear un proveedor de datos utilizando una base de datos Access, pero se ha comentado anteriormente, que ADO también es capaz de manejar información residente en otros formatos. Por este motivo, supongamos que necesitamos mostrar el contenido de un fichero de texto; esto es posible, siempre y cuando sepamos como está organizado el interior de dicho fichero.

Tomemos para nuestros propósitos, el fichero cdplayer.INI, que está en el directorio de Windows. Este fichero, contiene la información (si el usuario la ha introducido), acerca de los CD's de audio que se reproducen con esta aplicación incorporada en el sistema: intérprete, título, número de canciones, nombre de las canciones, etc. En el ejemplo de este apartado, ClsDatTxt, se adjunta este fichero, por si el que tuviera el lector se encontrara vacío, debiendo copiarlo al directorio de Windows. Dicho fichero contiene la información habitual, a excepción de los nombres de las canciones, a pesar de ello cumplirá a la perfección nuestros propósitos. La figura 460, muestra un fragmento del mismo.

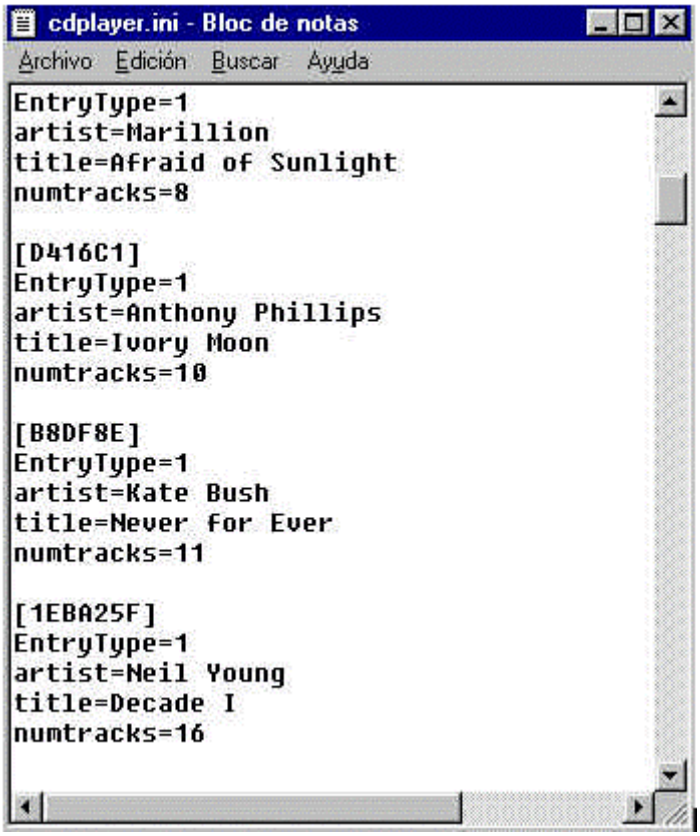


Figura 460. Contenido de cdplayer.INI.

Podemos observar, que para cada CD, existen cinco líneas de información y una en blanco de separación entre CD's, por lo que podemos considerar que un registro en este fichero, lo componen seis líneas de texto.

Conociendo como se estructura la información en cdplayer.INI, procederemos a crear la clase proveedora de datos a partir de dicho fichero.

Crearemos en primer lugar un nuevo proyecto, al que llamaremos ClsDatTxt, añadiendo un módulo de clase con el nombre CDPlayer; las propiedades referentes a datos serán iguales que las establecidas en el ejemplo del apartado anterior, ya que esta clase también actuará como origen de datos.

Como ya sabemos, el evento Initialize() de la clase es el encargado de crear y llenar el Recordset con los datos a exponer. En este caso, deberemos trabajar un poco más que en el anterior programa, ya que aquí, tenemos que ocuparnos de leer el fichero de texto, y volcar los valores que necesitamos al Recordset. Igualmente, tenemos que crear en el Recordset, campos para almacenar dichos valores. Todo ello se puede observar en el código fuente 401.

```
Private Sub Class_Initialize()
' declaración de variables
' para manejo del contenido
' de ficheros
Dim lfsoCDPlayer As FileSystemObject
Dim lfilCdPlayer As File
Dim ltxsTextoCD As TextStream
' resto de variables
Dim lnInd As Integer
Dim lcLinea As String
Dim lcArtista As String
Dim lcTitulo As String
Dim lnCanciones As Integer
' agregar la fuente de datos a la
' lista de miembros de datos
DataMembers.Add "ListaCD"
' crear el recordset, configurar y abrir
Set marsListaCD = New ADODB.Recordset
' añadir campos al recordset, que corresponderán
' a las cadenas recuperadas del fichero de texto
marsListaCD.Fields.Append "Artista", adBSTR
marsListaCD.Fields.Append "Titulo", adBSTR
marsListaCD.Fields.Append "Canciones", adInteger
' asignar propiedades al recordset
marsListaCD.CursorType = adOpenDynamic
marsListaCD.LockType = adLockOptimistic
' abrir recordset
marsListaCD.Open
' crear objeto para manipular el sistema de ficheros
Set lfsoCDPlayer = New FileSystemObject
' abrir el fichero de texto y pasarlo
' a un objeto File
Set lfilCdPlayer = lfsoCDPlayer.GetFile("c:\windows\cdplayer.ini")
' leer el contenido del objeto fichero y
' volcarlo en un objeto TextStream
Set ltxsTextoCD = lfilCdPlayer.OpenAsTextStream(ForReading)
' recorrer el texto obtenido del fichero
' y extraer las líneas correspondientes
' al artista, nombre del cd y nº canciones
Do While Not ltxsTextoCD.AtEndOfStream
   ' cada 6 líneas forman una entrada
   ' en el fichero cdplayer.ini,
   ' equivalente a un registro
   For lnInd = 1 To 6
      lcLinea = ltxsTextoCD.ReadLine
      ' si llegamos al final del texto, salir
      If ltxsTextoCD.AtEndOfStream Then
         Exit Do
      End If
      ' según la línea tomada, se interpretará
      ' como la correspondiente al artista,
      ' título o número de canciones del cd
      Select Case lnInd
      Case 3
         lcArtista = Mid(lcLinea, 8)
      Case 4
         lcTitulo = Mid(lcLinea, 7)

      Case 5
         lnCanciones = CInt(Mid(lcLinea, 11))
      End Select
      ' si la línea es la correspondiente
      ' al num.canciones
      ' agregar registro al recordset
      If lnInd = 5 Then
         marsListaCD.AddNew
```

```
         marsListaCD("Artista") = lcArtista
         marsListaCD("Titulo") = lcTitulo
         marsListaCD("Canciones") = lnCanciones
         marsListaCD.Update
      End If
   Next
Loop
marsListaCD.MoveFirst
Me.BOF = False
Me.EOF = False
End Sub
```

Código fuente 401

Sobre los objetos encargados en este procedimiento de la manipulación del fichero de texto, sugerimos al lector que consulte el tema Manipulación de ficheros, donde se describe este conjunto de objetos.

La variable que contiene el Recordset en esta clase, se ha declarado WithEvents para poder manipular cualquiera de sus eventos, en nuestro caso codificaremos el evento WillMove(), que se produce en el momento en el que se va a cambiar el registro actual en el objeto. La declaración de la variable y el evento se pueden observar a en el código fuente 402.

```
Option Explicit
' declarar un Recordset con eventos, para poder
' manipular su evento WillMove()
Private WithEvents marsListaCD As ADODB.Recordset
'-----------------------------------------------

Private Sub marsListaCD_WillMove(ByVal adReason As ADODB.EventReasonEnum, adStatus
As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
' este evento se produce cuando se va a cambiar
' el registro del Recordset contenido en la clase
Select Case adReason
Case adRsnMoveNext
   Me.Operacion = "Pasamos al siguiente registro"
Case adRsnMovePrevious
   Me.Operacion = "Pasamos al anterior registro"
Case adRsnMoveFirst
   Me.Operacion = "Pasamos al primer registro"
Case adRsnMoveLast
   Me.Operacion = "Pasamos al último registro"
End Select
End Sub
```

Código fuente 402

Dependiendo de la acción llevada a cabo en el Recordset, se evaluará la constante contenida en el parámetro adReason, asignando una cadena descriptiva a la propiedad Operacion, creada por nosotros para esta clase.

En cuanto a los métodos para el desplazamiento de registros, podemos observar, como muestra, el que realiza el movimiento al registro anterior, el resto puede consultarlos el lector al cargar este proyecto en VB.

```
Public Sub MovePrevious()
' retroceder registro y establecer
' la propiedad BOF de esta clase
```

```
' según la posición en el Recordset
marsListaCD.MovePrevious
If marsListaCD.BOF Then
   Me.BOF = True
Else
   Me.BOF = False
End If
End Sub
```

Código fuente 403

Para visualizar el contenido de un registro, disponemos del método Mostrar(). Observe el lector, que una vez que en Initialize() hemos volcado el contenido del fichero en el Recordset, la manipulación de los datos, se realiza mediante dicho Recordset igual que si estuviéramos manejando filas de una tabla de Access, por ejemplo.

```
Public Sub Mostrar()
' visualizar el contenido de un registro
' del Recordset de esta clase
MsgBox "Artista: " & marsListaCD("Artista") & vbCrLf & _
   "Título del CD: " & marsListaCD("Titulo") & vbCrLf & _
   "Número de canciones: " & marsListaCD("Canciones")
End Sub
```

Código fuente 404

Finalizada la creación de la clase proveedora, vamos a crear el código para mostrar su contenido al ejecutar esta aplicación. Esta tarea la realizaremos de dos maneras: sin interfaz de usuario, mostrando directamente con una caja de mensajes al usuario, el contenido de los registros del objeto proveedor; y mediante un formulario al que enlazaremos varios controles, de forma que ilustremos las características de ADO en cuanto a enlace de objetos proveedores a consumidores.

Para el primer caso, añadiremos, al procedimiento Main() de esta aplicación, el código fuente 405, que instanciarán un objeto CDPlayer y visualizarán su contenido hasta el final del Recordset que contiene el objeto.

```
Public Sub Main()
Dim loCDPlayer As CDPlayer

' instanciar un objeto de la clase
' proveedora de datos
Set loCDPlayer = New CDPlayer
' mostrar los datos del proveedor
' sin utilizar interfaz visual
Do While Not loCDPlayer.EOF
   loCDPlayer.Mostrar
   loCDPlayer.MoveNext
Loop
' liberar recursos del objeto
Set loCDPlayer = Nothing
End Sub
```

Código fuente 405

En el segundo caso, deberemos trabajar un poco más, ya que será necesario crear un formulario y escribir el código de creación del objeto proveedor y enlace a controles. Insertaremos pues un formulario en el proyecto, al que llamaremos frmCDPlayer, que contendrá un TextBox por cada campo a visualizar; un DataGrid, que mostrará los datos del objeto CDPlayer en forma de tabla, un control Label que mostrará el resultado del evento WillMove() producido en el objeto CDPlayer, y finalmente un CommandButton para las acciones de movimiento de registros.

En cuanto al código de frmCDPlayer, en su zona de declaraciones, crearemos una variable para el objeto CDPlayer y otra para un objeto BindingCollection, como se muestra en el código fuente

```
Option Explicit

Private moCDPlayer As CDPlayer
Private mbndEnlaces As BindingCollection
```

Código fuente 406

En el evento de carga del formulario, que vemos en el código fuente 407, es donde deberemos incluir todo el trabajo de creación de objetos y enlace a los controles.

```
Private Sub Form_Load()
' instanciar objeto de la clase
' proveedora de datos
Set moCDPlayer = New CDPlayer
' instanciar objeto para crear enlaces
Set mbndEnlaces = New BindingCollection
' establecer la fuente de datos en
' el objeto de enlaces
Set mbndEnlaces.DataSource = moCDPlayer
' establecer en el objeto de enlaces:
' - que controles del formulario se enlazan
' - la propiedad del control enlazado que utilizará el proveedor
'   de datos en dicho control para visualizar la información
' - nombre del campo que se mostrará en la propiedad del control
mbndEnlaces.Add Me.txtArtista, "Text", "Artista"
mbndEnlaces.Add Me.txtTitulo, "Text", "Titulo"
mbndEnlaces.Add Me.txtCanciones, "Text", "Canciones"
' conectar también el DataGrid con el
' proveedor de datos
Set Me.grdCdPlayer.DataSource = moCDPlayer
End Sub
```

Código fuente 407

Como podemos comprobar, una vez asignado al objeto BindingColletion el proveedor de datos, usaremos su método Add(), para conectar un control del formulario con un campo de los contenidos en el proveedor.

Por último, incluiremos el código para los botones de navegación en el proveedor de datos. Como ejemplo, en el código fuente 408 tenemos el que se ocupa de pasar al siguiente registro.

```
Private Sub cmdAvanzar_Click()
moCDPlayer.MoveNext
If moCDPlayer.EOF Then
```

```
   moCDPlayer.MoveLast
End If
Me.lblOperacion.Caption = moCDPlayer.Operacion
End Sub
```

Código fuente 408

Una vez terminada la labor de creación de este formulario, añadiremos en el procedimiento Main(), (seguido al código fuente 408), las líneas encargadas de mostrar esta ventana al usuario dichas líneas se encuentran en el código fuente 409.

```
Public Sub Main()
Dim loCDPlayer As CDPlayer
Dim lfrmCDPlayer As frmCDPlayer
..............
..............
..............
' utilizar un formulario para
' interactuar con un objeto
' CDPlayer que se creará en
' su interior
Set lfrmCDPlayer = New frmCDPlayer
Load lfrmCDPlayer
lfrmCDPlayer.Show
End Sub
```

Código fuente 409

El resultado de la ejecución de este formulario, podemos verlo en la figura 461.

Figura 461. Formulario con controles enlazados a una clase proveedora de datos.

Trabajar con varios miembros de datos

Un miembro de datos se puede definir como un conjunto de información (consulta SQL, tabla, etc.), contenido en un proveedor u origen de datos. De un modo más familiar y simple para el lector, podemos decir que cada Recordset creado dentro de una clase proveedora de datos, es un miembro de datos, ya que una de las grandes cualidades de los proveedores de datos en Visual Basic 6, es la posibilidad de contener más de un miembro de datos, y de que cada uno proporcione la información, no sólo de una misma base de datos, sino de diferentes bases de datos y con distinto tipo; como es lógico, la potencia que esta característica brinda a nuestros desarrollos es enorme.

Los ejemplos usados hasta el momento, sólo han aprovechado esta capacidad mínimamente, limitándose a crear un único miembro de datos para la clase proveedora; sin embargo, el programa DataMember, desarrollado para este apartado, mostrará la manera de crear y utilizar más de un miembro de datos residente en un proveedor.

El primer paso a dar, consiste en la creación de la clase proveedora. Dicha clase tendrá el nombre Informa, y va a proporcionar registros de los ya conocidos ficheros Cine.MDB y Musica.MDB. Como ya sabemos de anteriores apartados, la creación de los orígenes de datos se realizará en el evento Initialize() de la clase, que vemos en el código fuente 410.

```
Private Sub Class_Initialize()

Dim lcRuta As String
Dim lnInd As Integer
lcRuta = "C:\CursoVB6\Texto\DatosADO\DataMember\"

' base de datos cine.mdb
' agregar el nombre del miembro de datos
' a la colección
DataMembers.Add "Cine"

' instanciar conexion
Set macnCine = New ADODB.Connection

' establecer cadena de conexión
macnCine.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
"Data Source=" & lcRuta & "cine.mdb"

' abrir conexión
macnCine.Open

' crear el recordset y asignarle la conexión
Set marsCine = New ADODB.Recordset
Set marsCine.ActiveConnection = macnCine

' asignar al recordset la sentencia SQL
' para recuperar datos
marsCine.Source = "SELECT * FROM Directores ORDER BY Director"

' abrir recordset
marsCine.Open , , adOpenForwardOnly, adLockOptimistic, adCmdText
marsCine.MoveFirst

'************************************
' base de datos musica.mdb
' agregar el nombre del miembro de datos
' a la colección
DataMembers.Add "Musica"

' instanciar conexion
```

```
Set macnMusica = New ADODB.Connection

' establecer cadena de conexión
macnMusica.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
"Data Source=" & lcRuta & "musica.mdb"

' abrir conexión
macnMusica.Open

' crear el recordset y asignarle la conexión
Set marsMusica = New ADODB.Recordset
Set marsMusica.ActiveConnection = macnMusica

' establecer propiedades en el recordset
' para recuperar datos
marsMusica.Source = "Autores"

' abrir recordset
marsMusica.Open , , adOpenForwardOnly, adLockOptimistic, adCmdTable
marsMusica.MoveFirst

End Sub
```

Código fuente 410

Para no complicar en exceso el código de este ejemplo, se utiliza una ruta fija de datos en la variable lcRuta, por lo que el lector deberá utilizar esta misma, o modificar el valor de dicha variable, asignando una nueva ruta para los datos.

En el resto del código de este evento se pueden distinguir dos bloques bien diferenciados: por un lado las líneas que crean el Recordset de la tabla Directores; y por otro, las que crean el Recordset de la tabla Autores. Por cada conjunto de resultados o Recordset, deberemos añadir a la colección DataMembers de la clase, una cadena con el nombre para dicho miembro, que nos facilite el acceso posterior a los datos cuando, por ejemplo, vayamos a enlazar el proveedor con los controles de un formulario, como se muestra en el código fuente 411.

```
DataMembers.Add "Cine"
.................
.................
DataMembers.Add "Musica"
.................
.................
```

Código fuente 411

Al establecer esta clase como proveedora de datos, se añade automáticamente, el evento vacío GetDataMember(), que será el encargado de devolver al consumidor, el miembro de datos adecuado en el parámetro Data, en función del valor que tenga el parámetro DataMember.

```
Private Sub Class_GetDataMember(DataMember As String, Data As Object)
' asignar el recordset al objeto Data
' en función del miembro de datos solicitado
Select Case DataMember
Case "Cine"
   Set Data = marsCine
Case "Musica"
```

```
   Set Data = marsMusica
End Select
End Sub
```

Código fuente 412

Para que el consumidor sepa cuando ha llegado al principio o final de los registros en cada miembro de datos, escribiremos los procedimientos de propiedad que aparecen en el código fuente 413.

```
Public Property Get EOFCine() As Boolean
EOFCine = mbEOFCine
End Property
Public Property Let EOFCine(ByVal vbEOFCine As Boolean)
mbEOFCine = vbEOFCine
End Property
'-----------------------------------
Public Property Get EOFMusica() As Boolean
EOFMusica = mbEOFMusica
End Property
Public Property Let EOFMusica(ByVal vbEOFMusica As Boolean)
mbEOFMusica = vbEOFMusica
End Property
'-----------------------------------
Public Property Get BOFCine() As Boolean
BOFCine = mbBOFCine
End Property
Public Property Let BOFCine(ByVal vbBOFCine As Boolean)
mbBOFCine = vbBOFCine
End Property
'-----------------------------------
Public Property Get BOFMusica() As Boolean
BOFMusica = mbBOFMusica
End Property
Public Property Let BOFMusica(ByVal vbBOFMusica As Boolean)
mbBOFMusica = vbBOFMusica
End Property
```

Código fuente 413

Para finalizar con esta clase, debemos escribir los procedimientos de navegación por los registros de los miembros de datos, y ya que disponemos de más de uno, hemos de proporcionar código extra para discernir sobre qué miembro debemos efectuar el movimiento de fila. Como estos procedimientos siguen una técnica similar, sólo se muestra en el código fuente 414, el encargado de realizar el avance de fila.

```
' procedimiento de movimiento para los
' recordsets de la clase, mediante la
' cadena pasada como parámetro, sabremos
' sobre cual se necesita efectuar el
' movimiento de registro
Public Sub MoveNext(ByVal vcDataMember As String)
' avanzar al siguiente registro
' del miembro de datos solicitado
Select Case vcDataMember
Case "Cine"
   marsCine.MoveNext
   If marsCine.EOF Then
```

```
      Me.EOFCine = True
   Else
      Me.EOFCine = False
   End If
Case "Musica"
   marsMusica.MoveNext
   If marsMusica.EOF Then
      Me.EOFMusica = True
   Else
      Me.EOFMusica = False
   End If
End Select
End Sub
```

Código fuente 414

En el siguiente paso, crearemos el formulario frmDataMember con dos TextBox, los cuales serán enlazados a un objeto de la clase Informa; y varios CommandButton, que efectuarán el desplazamiento por los registros del objeto Informa. Cada conjunto de controles formado por un TextBox y los CommandButton de navegación, operarán sobre uno de los miembros de datos de la clase proveedora. La figura 462, muestra este formulario en ejecución.

Figura 462. Formulario con controles enlazados a una clase proveedora con varios miembros de datos.

El evento Load() del formulario es el lugar donde crearemos un objeto de la clase Informa, y lo enlazaremos con los TextBox, que actuarán como consumidores de datos.

```
Private Sub Form_Load()
' instanciar objeto proveedor de datos
Set moInforma = New Informa
' instanciar el objeto de enlace
' para los datos de la base de
' datos Cine
Set mbndDirectores = New BindingCollection
' indicar al objeto de enlace,
' cual de los miembros de datos
' existentes en el proveedor
' es con el que vamos a trabajar
mbndDirectores.DataMember = "Cine"
' asignar el proveedor al objeto
' de enlace
Set mbndDirectores.DataSource = moInforma
' establecer el enlace hacia el control
' del formulario
mbndDirectores.Add Me.txtDirector, "Text", "Director"
' realizar las mismas operaciones que
' en las líneas anteriores, pero esta vez
' creando un nuevo objeto BindingCollection
' para enlazar con el miembro de datos
```

```
' Musica
Set mbndAutores = New BindingCollection
mbndAutores.DataMember = "Musica"
Set mbndAutores.DataSource = moInforma
mbndAutores.Add Me.txtAutor, "Text", "Autor"
End Sub
```

Código fuente 415

Observe el lector, que los objetos encargados de la manipulación de datos, se han declarado a nivel de módulo. Esto se debe hacer así, de forma que una vez creado el objeto, se mantenga la referencia hacia él, durante el tiempo en que el formulario esté en ejecución. Si se hubieran declarado locales a Load(), se perdería esa referencia al terminar este procedimiento, y no tendríamos acceso a los datos de los miembros.

También debemos destacar, que para cada miembro de datos que necesitemos enlazar con consumidores, crearemos un objeto BindingCollection. El enlace en este objeto, se produce asignando una cadena con el nombre del miembro a enlazar, a la propiedad DataMember de este objeto; a continuación, asignaremos a la propiedad DataSource, el objeto proveedor, esto hará que se desencadene el evento GetDataMember() (del que hemos visto su código anteriormente) del proveedor de datos. Como este evento recibe en su parámetro DataMember, el valor de la propiedad DataMember del objeto BindingCollection, se devolverá el Recordset correspondiente, quedando sólo la tarea en este caso, de enlazar el control consumido, con el campo del miembro de datos. Recomendamos el establecimiento de puntos de interrupción tanto en el procedimiento GetDataMember() como en Load(), de forma que el lector pueda comprobar usando el depurador como se realizan las llamadas y asignación de propiedades en esta parte del código del ejemplo.

El resto de código en el formulario, pertenece a los botones encargados de la navegación por los registros del proveedor; como es similar al de otros ejemplos, evitaremos repetirlo aquí, pudiendo ser consultado al cargar este proyecto en el IDE de Visual Basic.

Por último, escribiremos el código fuente 416 en el procedimiento Main() del programa, para que visualice el formulario.

```
Public Sub Main()
Dim lfrmDataMember As frmDataMember
Set lfrmDataMember = New frmDataMember
Load lfrmDataMember
lfrmDataMember.Show
End Sub
```

Código fuente 416

Creación de un control personalizado orientado a datos

La posibilidad de crear controles orientados a datos en VB, ya estaba disponible desde la versión 5, pero presentaba importantes inconvenientes, debido al hecho de que para desarrollar un control ActiveX de este tipo, se debía conocer de antemano el campo de la tabla o consulta con el que se iba a enlazar. Esto, obviamente, resultaba poco útil en el caso de controles que necesitaran interaccionar con campos diferentes, de también diferentes orígenes de datos.

Como es habitual en estos casos, podíamos crear un control que trabajara con cualquier campo, pero era necesario tener conocimientos de C++ y una serie de complejas especificaciones de Microsoft para conseguir este propósito. Todo ello, unido al elevado ritmo de desarrollo que imponen generalmente las aplicaciones, hacía que la mayoría de programadores renunciasen a la creación de controles de este tipo, optando por los habitualmente ofrecidos por VB, que si bien en algunos casos, no disponían de todas las cualidades esperadas por el programador, compensaban el tiempo que había que emplear en crear uno nuevo.

En Visual Basic 6, este problema cambia radicalmente, ya que es posible crear controles ActiveX orientados a datos de una forma sencilla. Para ello, se incluye el proyecto CtlDatos, que consiste en un control ActiveX personalizado, con una funcionalidad similar a la del control ADO Data, aunque más recortada, ya que de lo que se trata, es de mostrar al lector las principales bases de código en un control de este tipo, y de como realizar el enlace entre este control y los controles enlazados en un formulario. Este ejemplo contiene también el proyecto EXE estándar Pruebas, de manera que ambos forman el grupo de proyectos GrupoPruebas, para poder realizar las pruebas con el control desde un formulario.

Crearemos pues, en primer lugar, un proyecto de tipo Control ActiveX. No entraremos en los detalles de creación de un UserControl, ya que este aspecto es tratado en el tema ActiveX, apartado Controles ActiveX. En este control, además del código propio del control, introduciremos las características que hemos aprendido sobre una clase proveedora de datos, de manera que obtendremos un control de datos al estilo del proporcionado por VB. No debemos olvidar establecer en el proyecto la referencia hacia la librería Microsoft ActiveX Data Objects 2.0 Library, para poder trabajar con este modelo de objetos.

Interfaz del control

En la plantilla o UserControl de este proyecto, insertaremos varios CommandButton, que nos servirán para realizar la navegación por los registros de la fuente de datos a la que nos conectemos, y para editar sus registros. También se incluirá un Label que nos informará cuando nos situemos al principio o final de los ficheros. La figura 463, muestra el resultado del control una vez diseñado su interfaz de usuario.

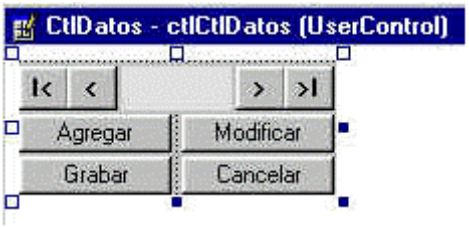


Figura 463. Interfaz de usuario del control ActiveX, CtlDatos.

También es muy importante que en la propiedad DataSourceBehavior del UserControl, asignemos el valor vbDataSource, para que este control se comporte como proveedor de datos.

Variables y procedimientos para propiedades

En la zona de declaraciones del módulo de código perteneciente al control, declararemos un conjunto de variables para guardar los valores de propiedad del control, el objeto Connection y Recordset para datos, así como varias constantes con los valores por defecto para las propiedades del control.

```
Option Explicit
' constantes con los valores por
' defecto para las propiedades
Const mcDEFConnectionString = ""
Const mcDEFRecordSource = ""
Const mkDEFCommandType = adCmdTable
Const mkDEFCursorType = adOpenDynamic
Const mkDEFLockType = adLockOptimistic
' variables de propiedad
Private mcConnectionString As String
Private mcRecordSource As String
Private mkCommandType As CommandTypeEnum
Private mkCursorType As CursorTypeEnum
Private mkLockType As LockTypeEnum
' variables normales
Private macnConexion As ADODB.Connection
Private marsRecordset As ADODB.Recordset
```

Código fuente 417

En cuanto a los procedimientos de propiedad, debemos destacar los que reciben el valor para la cadena de conexión y la fuente de datos; en ellos se deben eliminar los objetos Connection y Recordset, de forma que puedan generarse de nuevo cuando este control se vuelva a conectar a un control enlazado. Si no hiciéramos esta operación, al intentar enlazar un control a nuestro control de usuario, se intentarían recuperar unos campos que no corresponderían con los que realmente contiene el Recordset.

```
Public Property Let ConnectionString(ByVal vcConnectionString As String)
mcConnectionString = vcConnectionString
PropertyChanged "ConnectionString"
' al establecer una nueva cadena de conexión
' es necesario eliminar el contenido de
' los objetos conexión y recordset existentes
' para que al asignar nuevos valores a los
' controles enlazados, se genere la nueva
' conexión y recordset
Set macnConexion = Nothing
Set marsRecordset = Nothing
End Property
'-------------------------------------------------
Public Property Let RecordSource(ByVal vcRecordSource As String)
mcRecordSource = vcRecordSource
PropertyChanged "RecordSource"
' al establecer una nueva fuente de datos
' es necesario eliminar el contenido de
' los objetos conexión y recordset existentes
' para que al asignar nuevos valores a los
' controles enlazados, se genere la nueva
' conexión y recordset
Set macnConexion = Nothing
Set marsRecordset = Nothing
End Property
```

Código fuente 418

Se han definido una serie de propiedades que contendrán los valores de algunos tipos enumerados ya definidos en VB, para ello es necesario tipificar tanto la variable de propiedad como sus procedimientos Property asociados con ese tipo enumerado.

```
Public Property Get CursorType() As CursorTypeEnum
CursorType = mkCursorType
End Property
Public Property Let CursorType(ByVal vkCursorType As CursorTypeEnum)
mkCursorType = vkCursorType
End Property
```

Código fuente 419

Esta acción tiene como ventaja, que al insertar nuestro control en un formulario, en estas propiedades aparezca la lista de valores del tipo enumerado, evitando ese trabajo al programador, como vemos en la figura 464.

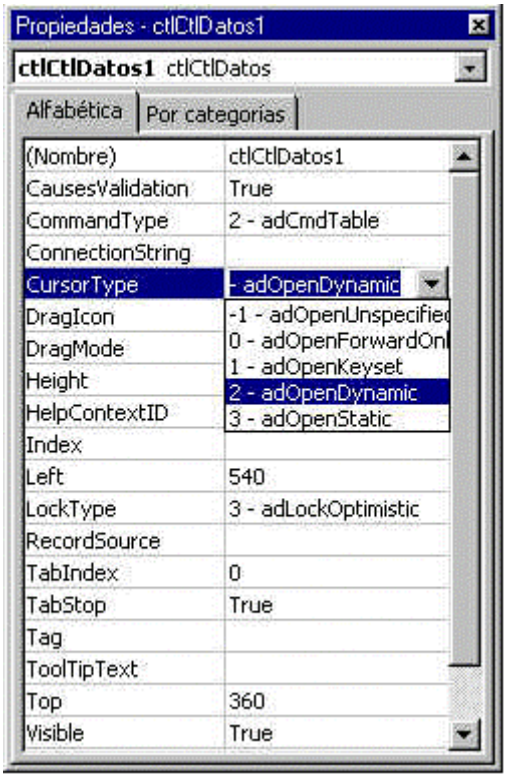


Figura 464. Propiedad creada en un control de usuario, mostrando valores de un tipo enumerado.

Código del UserControl

En este apartado sólo comentaremos el código perteneciente al evento GetDataMember() del control, en el cual se crean los objetos Connection y Recordset internos del control, en base a los valores de sus propiedades, establecidas por el usuario, y que proporcionarán los datos a los controles enlazados en un formulario. El código fuente 420, contiene comentarios explicativos de las acciones realizadas.

```
Private Sub UserControl_GetDataMember(DataMember As String, Data As
Object)
' cuando el usuario modifique los valores en la
' propiedad ConnectionString o RecordSource, o en
' ambas a la vez; al establecer en alguno de los
```

```
' controles enlazados a este control, un nuevo
' valor: por ejemplo en la propiedad DataField
' de un TextBox enlazado, se producirá este evento
' que actualizará la conexión y el recordset para
' dichos controles enlazados
If (macnConexion Is Nothing) Or (marsRecordset Is Nothing) Then
   ' si no se ha establecido una cadena de
   ' conexión o fuente de datos, avisar al
   ' usuario
   If Len(Me.ConnectionString) = 0 Then
      MsgBox "Es necesario establecer una cadena de conexión", _
         vbCritical, "Error"
      Exit Sub
   End If
   If Len(Me.RecordSource) = 0 Then
      MsgBox "Es necesario establecer la fuente de datos", _
         vbCritical, "Error"
      Exit Sub
   End If
   ' crear el objeto conexión,
   ' la cadena de conexión debe
   ' contener la suficiente información
   ' para establecerla
   Set macnConexion = New ADODB.Connection
   macnConexion.Open Me.ConnectionString
   ' crear el objeto recordset,
   ' el valor de la propiedad RecordSource
   ' puede ser el nombre de una tabla
   ' o una consulta SQL
   Set marsRecordset = New ADODB.Recordset

   ' aunque las propiedades para el tipo de cursor
   ' bloqueo y comando tienen un valor por
   ' defecto, es necesario ser cuidadosos
   ' para que no causen conflictos entre sí
   marsRecordset.CursorType = Me.CursorType
   marsRecordset.LockType = Me.LockType

   ' abrir el recordset
   marsRecordset.Open Me.RecordSource, macnConexion, , , Me.CommandType
   Set Data = marsRecordset
End If
End Sub
```

Código fuente 420

Eventos como Resize(), ReadProperties() y otros propios del control, al haber sido tratados en el tema sobre ActiveX, no se comentarán aquí, ya que su codificación es la misma, aunque cambien las propiedades.

Botones de navegación

En lo que respecta a los botones que realizan el desplazamiento de registros, en el código fuente 421 podemos ver el fuente del que efectúa el avance al siguiente.

Debido a que no sabemos si el programador que utilice nuestro control, habilitará o no los controles enlazados al nuestro, comprobamos si se ha modificado alguno de los campos del registro actual, deshaciendo dichos cambios con el método CancelUpdate() del Recordset; de esta manera, sólo permitimos la edición cuando el usuario pulse sobre el botón Agregar o Modificar de nuestro control.

```
Private Sub cmdAvanzar_Click()
' avanzar al siguiente registro del recordset
' si el usuario ha modificado alguno de los
' controles enlazados que muestran los campos
' del registro, cancelar dicha edición, ya que
' no se ha producido por medio del UserControl
If marsRecordset.EditMode = adEditInProgress Then
   marsRecordset.CancelUpdate
End If
marsRecordset.MoveNext
If marsRecordset.EOF Then
   marsRecordset.MoveLast
   lblEstadoEdic.Caption = "Último registro"
Else
   lblEstadoEdic.Caption = ""
End If
End Sub
```

Código fuente 421

El resto de botones de movimiento puede consultarlos el lector desde el entorno de trabajo de VB.

Botones de edición

En el código fuente 422 se muestra el código correspondiente a los botones del UserControl, encargados de las labores de edición del registro actual del Recordset. Para esta tarea, disponemos como apoyo de dos procedimientos: BotonesEditar() y BotonesNoEditar(), cuya labor consiste en habilitar y deshabilitar los controles constituyentes de nuestro UserControl, en función del modo de edición en el que se encuentre actualmente su Recordset.

```
Private Sub cmdAgregar_Click()
' añadir un nuevo registro al recordset
' y preparar los controles constituyentes
' para su edición
marsRecordset.AddNew
BotonesEditar
End Sub
'------------------------------
Private Sub cmdModificar_Click()
BotonesEditar
End Sub
'------------------------------
Private Sub cmdGrabar_Click()
marsRecordset.Update
BotonesNoEditar
End Sub
'------------------------------
Private Sub cmdCancelar_Click()
' cancelar la edición del registro
marsRecordset.CancelUpdate
' refrescar el contenido del registro
' con los valores originales, moviendo
' cero registro en el recordset
marsRecordset.Move 0
BotonesNoEditar
End Sub
```

```
'-------------------------------
Private Sub BotonesEditar()
' al agregar o modificar un registro
' habilitar y deshabilitar los botones
' correspondientes
cmdAvanzar.Enabled = False
cmdRetroceder.Enabled = False
cmdPrincipio.Enabled = False
cmdFinal.Enabled = False
cmdAgregar.Enabled = False
cmdModificar.Enabled = False
cmdGrabar.Enabled = True
cmdCancelar.Enabled = True
End Sub
'-------------------------------
Private Sub BotonesNoEditar()
' después de grabar un registro
' o cancelar su edición,
' habilitar y deshabilitar los
' botones correspondientes
cmdAvanzar.Enabled = True
cmdRetroceder.Enabled = True
cmdPrincipio.Enabled = True
cmdFinal.Enabled = True
cmdAgregar.Enabled = True
cmdModificar.Enabled = True
cmdGrabar.Enabled = False
cmdCancelar.Enabled = False
End Sub
```

Código fuente 422

El proyecto para pruebas

Finalizada la creación del control personalizado, agregaremos un proyecto EXE Estándar, al que llamaremos Pruebas, en el que comprobaremos el comportamiento del control que acabamos de crear. En el Explorador de proyectos, deberemos establecer este que acabamos de crear, como inicial. Estos proyectos los guardaremos bajo el nombre GrupoPruebas.

El formulario de prueba

En el proyecto EXE Estándar, tomaremos el formulario que aparece por defecto, o crearemos uno nuevo, dándole el nombre frmPruebas. Seguidamente, tomaremos del Cuadro de herramientas (figura 465) nuestro control y lo dibujaremos en el formulario.

Figura 465. Cuadro de herramientas de VB, mostrando el UserControl ctlCtlDatos.

A continuación, abriremos la Ventana de Propiedades del control ctlCtlDatos que acabamos de crear y le daremos el nombre datCine. Igualmente, daremos valores a las propiedades ConnectionString y RecordSource. Si utilizamos los valores que vienen por defecto para el control en este proyecto, deberemos crear la ruta para los datos y depositar en ella la base de datos de este ejemplo; aunque esto no es necesario, ya que el lector puede utilizar la ruta que prefiera, teniendo siempre la precaución de establecerla también en la propiedad ConnectionString. Las siguientes líneas, muestran los valores por defecto que se han asignado a estas propiedades.

- ConnectionString -> Provider=Microsoft.Jet.OLEDB.3.51;Data Source=C:\CursoVB6\Texto\DatosADO\CtlDatos\cine.mdb

- RecordSource -> Peliculas

Para poder ver el valor de los registros contenidos en el control datCine, insertaremos dos TextBox en el formulario. En la propiedad DataSource de dichos controles, asignaremos el nombre de nuestro control de usuario: datCine; mientras que al pulsar sobre la propiedad DataField, se disparará el evento GetDataMember() del control de datos, creando la conexión y Recordset especificados, y se desplegará una lista con los campos disponibles.

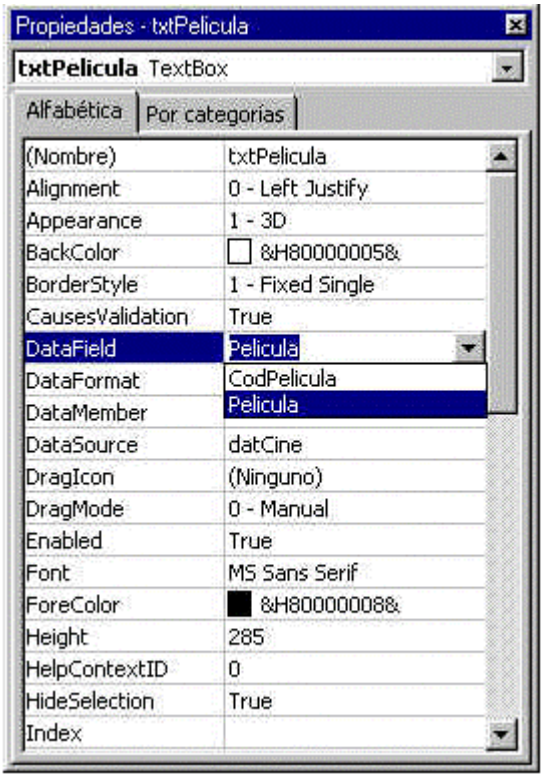


Figura 466. Ventana de propiedades de un control enlazado, con los campos disponibles para conectar.

Con la operación anterior, podemos dar por finalizada la creación del formulario de prueba para el control ctlCtlDatos, ahora sólo queda ejecutarlo, desplazarse por los registros y editarlos. La figura 467 muestra esta ventana en ejecución.

Figura 467. Formulario realizando pruebas con un control ctlCtlDatos.

Como apunte final en este ejemplo, sólo recordar que este programa se inicia desde el sencillo procedimiento Main() que vemos en el código fuente 423.

```
Public Sub Main()
Dim lfrmPruebas As frmPruebas
Set lfrmPruebas = New frmPruebas
Load lfrmPruebas
lfrmPruebas.Show
End Sub
```

Código fuente 423

El control DataRepeater

La misión de este control consiste en albergar un control ActiveX desarrollado por el programador y mostrarlo tantas veces como registros existan en una fuente de datos a la que se conectará el DataRepeater. La figura 468 muestra un esquema del funcionamiento de este control.

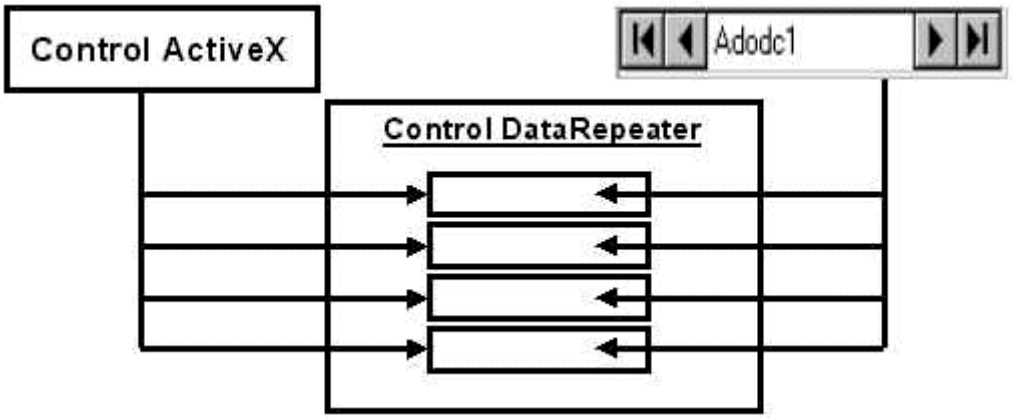


Figura 468. Esquema de trabajo del control DataRepeater.

El control DataRepeater actúa como central de proceso y punto de conexión entre un control ActiveX y una fuente de datos, en este caso un control ADO Data. El DataRepeater realizará una conexión con

el control de datos y por cada registro que obtenga, creará una nueva copia del control ActiveX, añadiendo en este control los valores de los campos del registro y mostrando el ActiveX en el interior del DataRepeater.

A continuación veremos un ejemplo práctico de creación de un control ActiveX y uso de un DataRepeater que actuará como contenedor del primero. En InfoCampos disponemos del proyecto que se encarga de crear el control ActiveX, mientras que en RepiteCtls tenemos una aplicación que hace uso del control ActiveX desde un DataRepeater. A continuación se explican los pasos necesarios para la creación de ambos proyectos.

En primer lugar crearemos un proyecto de tipo Control ActiveX con el nombre InfoCampos. En el UserControl al que daremos el nombre ctlInfoCampos, situaremos varios controles Label y TextBox tal y como muestra la figura 469.

Figura 469. Interfaz de usuario del control ActiveX.

En lo que respecta al código del control, escribiremos en el código fuente 424 los procedimientos Property, correspondientes a las propiedades del control.

```
Public Property Get Nombre() As String
Nombre = txtNombre.Text
End Property
'-------------------------------------------------
Public Property Let Nombre(ByVal vcNombre As String)
txtNombre.Text = vcNombre
End Property
'-------------------------------------------------
Public Property Get Ciudad() As String
Ciudad = txtCiudad.Text
End Property
'-------------------------------------------------
Public Property Let Ciudad(ByVal vcCiudad As String)
txtCiudad.Text = vcCiudad
End Property
'-------------------------------------------------
Public Property Get FechaAlta() As String
FechaAlta = txtFechaAlta.Text
End Property
'-------------------------------------------------
Public Property Let FechaAlta(ByVal vcFechaAlta As String)
txtFechaAlta.Text = vcFechaAlta
End Property
```

Código fuente 424

En el evento Change() de cada TextBox llamaremos al método PropertyChanged(), para notificar al control de que el usuario ha modificado el contenido de uno de estos controles constituyentes y actualice el campo de la fuente de datos.

```
Private Sub txtCiudad_Change()
PropertyChanged "Ciudad"
End Sub
'------------------------------------------------
Private Sub txtFechaAlta_Change()
PropertyChanged "FechaAlta"
End Sub
'------------------------------------------------
Private Sub txtNombre_Change()
PropertyChanged "Nombre"
End Sub
```

Código fuente 425

Para finalizar la configuración del control ActiveX, seleccionaremos la opción del menú de VB Herramientas+Atributos del procedimiento, que abrirá una ventana como la que muestra la figura 470, en la que estableceremos las propiedades orientadas a datos para las propiedades del control.

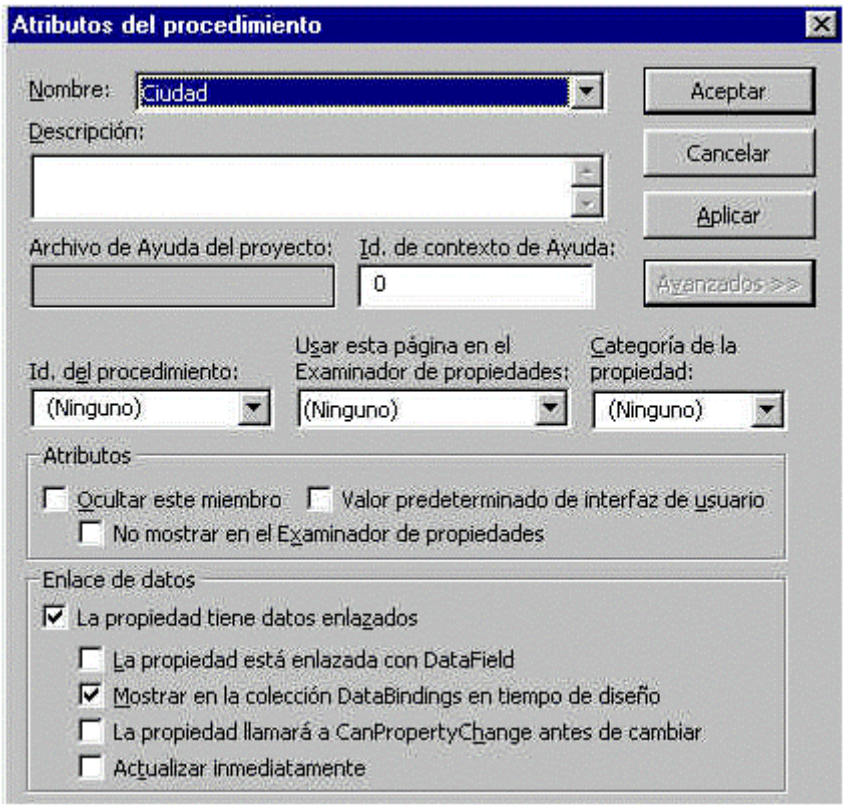


Figura 470. Ventana Atributos del procedimiento.

El ComboBox Nombre de esta ventana, contiene las propiedades que hemos definido a nivel de código. Seleccionaremos la primera que aparece: Ciudad, y pulsaremos el botón Avanzados>>, que nos dará acceso a todas las características disponibles para la propiedad. En el Frame Enlace de datos, marcaremos los CheckBox La propiedad tiene datos enlazados y Mostrar en la colección DataBindings en tiempo de diseño, de esta forma pondremos la propiedad a disposición del control DataRepeater en la aplicación que utilice este ActiveX. Repetiremos esta operación con las demás propiedades contenidas en Nombre, terminando el diseño del control.

Por último compilaremos el control, esto creará el correspondiente fichero .OCX y lo registrará en el sistema.

Seguidamente crearemos un nuevo proyecto de tipo EXE Estándar con el nombre RepiteCtls. Abriremos desde el Cuadro de herramientas de VB la ventana Componentes, agregando el control ADO Data y el DataRepeater, como se muestra en la figura 471.

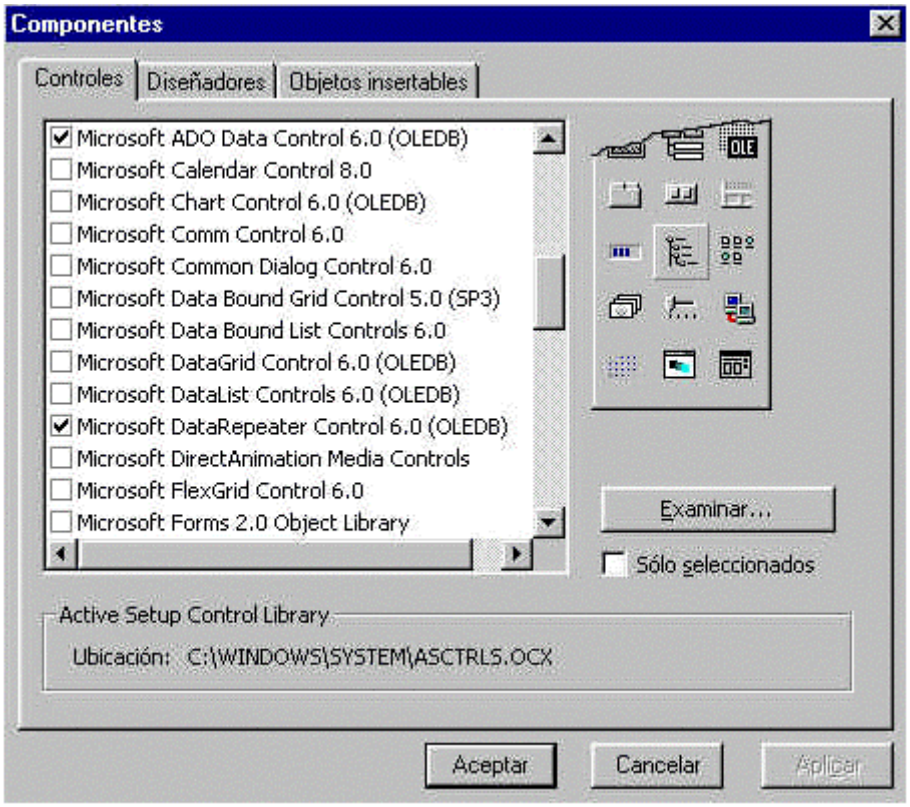


Figura 471. Añadir el control DataRepeater al proyecto.

Continuaremos agregando el formulario frmRepetir al proyecto, en el que insertaremos un control ADO Data con el nombre adcClientes, el cuál configuraremos para que conecte con la base de datos Empresa.MDB y abra la tabla Clientes, proceso este que ya conocemos de anteriores apartados.

En lo que respecta al control DataRepeater, después de insertarlo en el formulario y asignarle el nombre drpClientes, abriremos su ventana de propiedades. En la propiedad DataSource estableceremos el nombre del control ADO Data del que se obtendrán los registros: adcClientes; y en la propiedad RepeatedControlName seleccionaremos de la lista de controles ActiveX registrados, el que hemos desarrollado anteriormente: ctlInfoCampos. Esto hará que el control DataRepeater muestre en su interior un grupo de estos controles tal y como podemos ver en la figura 472.

El único paso que nos queda para conseguir que en los controles ActiveX contenidos en el DataRepeater aparezca la información de los registros de la tabla, consiste en abrir mediante el menú contextual del DataRepeater, la ventana de página de propiedades. Seleccionaremos la pestaña RepeaterBindings en la que encontramos los ComboBox PropertyName y DataField, que contendrán respectivamente los nombres de las propiedades del control ActiveX repetido y los campos de la tabla o consulta de la fuente de datos a la que se ha conectado el DataRepeater. Lo que debemos hacer es establecer la correspondencia entre cada propiedad y campo de la fuente de datos y añadirla al control pulsando el botón Agregar. La figura 473, muestra como ya se han conectado las propiedades y campos Ciudad y FechaAlta, restando sólo Nombre.

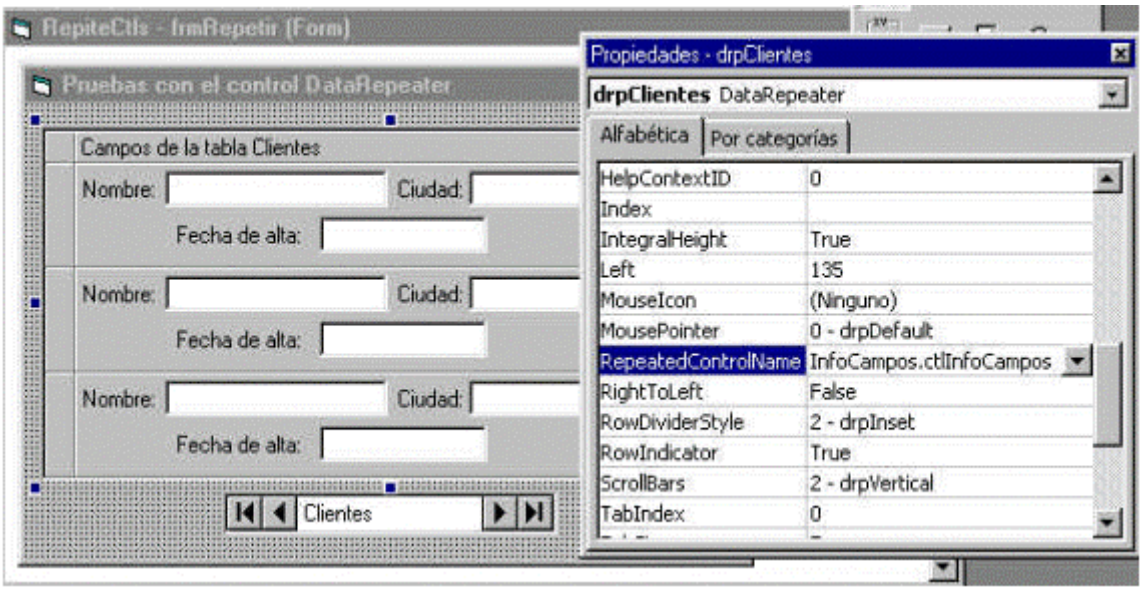


Figura 472. Selección del control a repetir en el DataRepeater.

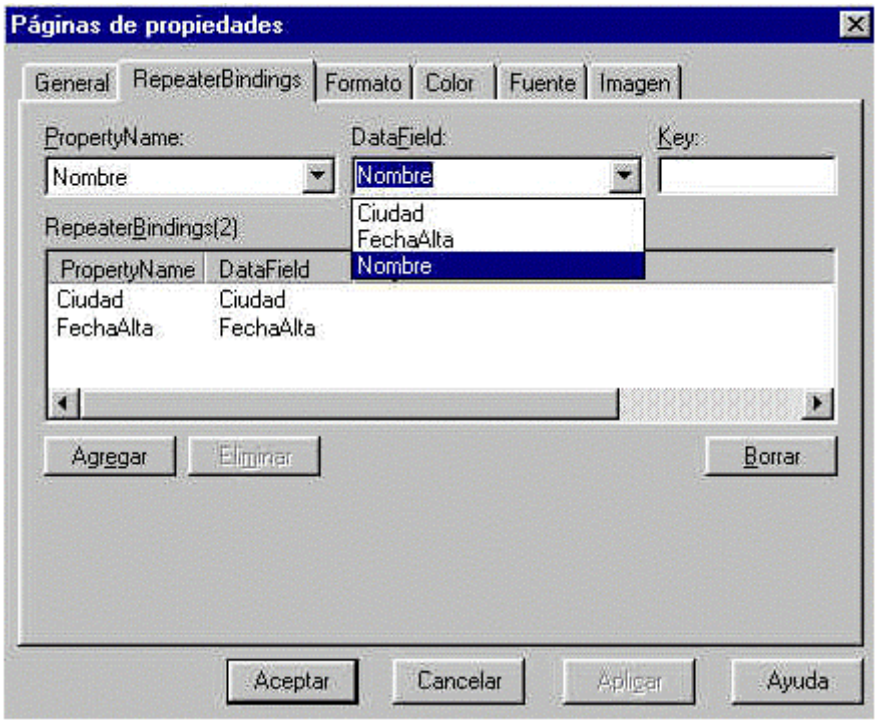


Figura 473. Creación de correspondencias entre las propiedades del control ActiveX contenido y los campos de la fuente de datos

En el caso de que necesitemos aplicar un formato de salida especial, en la pestaña Formato disponemos de los elementos necesarios para esta tarea. La figura 474 muestra como aplicamos un formato de fecha a la propiedad FechaAlta.

Con este paso podemos dar por concluida la creación y configuración del control. Ejecutaremos la aplicación, que cargará el formulario desde un procedimiento Main(), obteniendo un resultado similar al que aparece en la figura 475.

Figura 474. Establecimiento de formato para una propiedad.

Figura 475. Control DataRepeater en ejecución.

Cada uno de los controles ActiveX contenidos en el DataRepeater representa una fila de la fuente de datos, por la que podremos desplazarnos utilizando el control ADO Data.

Modelado de datos (Data Shaping)

El modelado de datos es una de las más interesantes características de ADO, consiste en la posibilidad de definir en un Recordset una relación maestro/detalle con varios niveles de profundidad. Bien es cierto que ya podíamos acceder a los datos de esta manera a través de una consulta SQL que incluyera un JOIN, pero la desventaja de este último es que repite información innecesariamente de las tablas.

Gracias al modelado de datos o creación de recordsets jerárquicos como también se le denomina, obtenemos única y exclusivamente la información necesaria sin recargar el Recordset con datos repetitivos, mejorando el rendimiento del acceso a datos.

El modo de conseguir de una base de datos la información organizada jerárquicamente, consiste en utilizar el proveedor de OLE DB: MSDataShape, y el conjunto de comandos SHAPE, que se combinan con las instrucciones SQL formando una extensión de estas.

Sintaxis

```
SHAPE {ComandoPrincipal} [AS] [TablaPrincipal]
    APPEND ({ComandoSecundario} [AS] [TablaSecundaria]
    RELATE ColumnaPrincipal TO ColumnaSecundaria) [AS] ColumnaEnlace
```

- ComandoPrincipal, ComandoSecundario. Instrucciones de recuperación de datos que contienen la selección de registros para la tabla de nivel superior o maestra, y para la tabla de nivel inferior o detalle.
- TablaPrincipal, TablaSecundaria. Nombre que se asigna a los comandos de recuperación de registros de la instrucción Shape, para poder referenciarlos posteriormente en el código de la aplicación.
- ColumnaPrincipal, ColumnaSecundaria. Nombre de la columna del comando principal y secundario, utilizadas para relacionar la tabla maestra con la tabla de detalles.
- ColumnaEnlace. Nueva columna creada en TablaPrincipal. Por cada registro de TablaPrincipal, esta columna devolverá una referencia a TablaSecundaria, que es la que contiene el Recordset con los registros de nivel secundario o de detalle, relacionados con un registro de la tabla maestra.

Para poder realizar pruebas empleando este sistema de manipulación de datos, el lector dispone del proyecto de ejemplo Modelado, que utiliza la base de datos Comercios.MDB. Esta base de datos contiene la información de una supuesta cadena de tiendas de papelería, en la que hallamos una tabla con los datos de las tiendas, otra para los artículos vendidos en cada tienda, para los empleados y las ventas realizadas.

A través de las opciones de menú del formulario principal del proyecto: mdiInicio, se conducirá al lector por las principales formas de manejo de datos empleando esta técnica de modelado.

Modelado simple

Comencemos por una relación sencilla, en la que obtendremos por cada registro de la tabla Papelerias, todos los registros relacionados en la tabla Empleados. La opción de menú a ejecutar es Modelado de datos+Relación simple, que contiene el código fuente 426.

```
Private Sub mnuMoRelac_Click()
Dim lacnComercios As ADODB.Connection
Dim larsPapelerias As ADODB.Recordset
Dim larsEmpleados As ADODB.Recordset
Dim lcComando As String
' crear objeto Connection, establecer propiedades
' y abrir
Set lacnComercios = New ADODB.Connection
lacnComercios.Provider = "MSDataShape"
lacnComercios.ConnectionString = "Data Provider=Microsoft.Jet.OLEDB.3.51;" & _
```

```
   "Data Source=" & gcRutaDatos
lacnComercios.Open
' crear objeto Recordset
Set larsPapelerias = New ADODB.Recordset
' definir comando SHAPE para
' Recordset jerárquico
lcComando = "SHAPE {SELECT * FROM Papelerias} AS TblPapelerias " & _
   "APPEND ( {SELECT * FROM Empleados} AS TblEmpleados " & _
   "RELATE IDPapeleria TO IDPapeleria ) AS TblEmpleados"

' abrir Recordset
larsPapelerias.Open lcComando, lacnComercios
' mostrar contenido del recordset
' tanto los registros del primer nivel
' como los del segundo
Do While Not larsPapelerias.EOF
   ' en el bucle exterior recorremos los registros
   ' de la tabla Papelerias
   MsgBox "Papelería: " & larsPapelerias("Nombre"), , "Tabla Papelerías"

   ' por cada registro de la tabla Papelerias
   ' obtenemos un recordset secundario con los
   ' registros relacionados en la tabla Empleados
   ' que recorremos en este otro bucle anidado
   Set larsEmpleados = larsPapelerias("TblEmpleados").Value
   Do While Not larsEmpleados.EOF
      MsgBox "Empleado: " & larsEmpleados("Nombre"), , _
         "Papelería: " & larsPapelerias("Nombre")
      larsEmpleados.MoveNext
   Loop

   larsPapelerias.MoveNext
Loop
End Sub
```

Código fuente 426

Hemos comentado anteriormente que el proveedor de datos a utilizar cuando creemos recordsets jerárquicos es MSDataShape, pero este proveedor sólo no es suficiente. Como habrá comprobado el lector en el fuente, a la propiedad Provider del objeto Connection se le asigna este proveedor, pero en la propiedad ConnectionString se utiliza sin embargo el proveedor que corresponde a la base de datos de la que obtendremos los registros (Jet). La razón de configurar el objeto Connection de esta manera, es que mientras que el proveedor Jet es el que realmente se comunica con la base de datos y recupera los registros, el proveedor MSDataShape se encarga de darles forma o modelar la presentación de los datos.

El control Hierarchical FlexGrid

El ejemplo anterior, si bien nos ha proporcionado la suficiente información sobre cómo recuperar datos desde un recordset jerárquico, no resulta lo bastante práctico de cara a una aplicación real en la que debamos mostrar al usuario una vista general de la información. Debemos utilizar para ello el control Hierarchical FlexGrid o HFlexGrid, especialmente diseñado para dicha tarea e incluido en esta versión de VB.

El control HFlexGrid muestra los datos en forma de rejilla al igual que el conocido DataGrid, salvo que el primero emplea características avanzadas de agrupación jerárquica de la información.

Para poder emplear este control en nuestro programa, debemos abrir el menú contextual del Cuadro de herramientas y seleccionar la opción Componentes, para abrir la ventana del mismo nombre. Una vez abierta esta ventana, elegiremos el control Microsoft Hierarchical FlexGrid Control 6.0 (OLEDB) y pulsaremos Aceptar, con lo que el nuevo control se añadirá al cuadro.

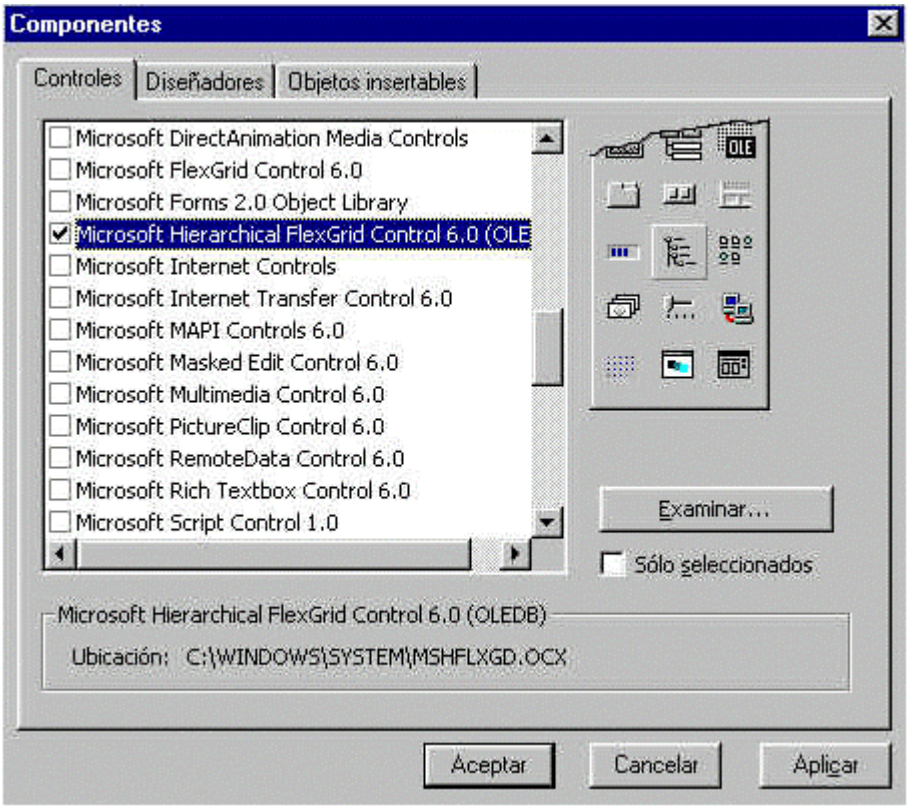


Figura 476. Selección del control Hierarchical FlexGrid en la ventana Componentes.

Figura 477. Control Hierarchical FlexGrid agregado al Cuadro de herramientas.

A continuación agregaremos un formulario al proyecto, al que le daremos el nombre frmFlexGrid insertándole un control de este tipo, y asignándole el nombre grdPapelerias. El resto del trabajo se realiza desde el código de la aplicación. La opción Modelado de datos+Control Hierarchical FlexGrid, abrirá el formulario frmFlexGrid, en cuyo evento Load() estará el código que se encargará de la creación del recordset jerárquico y la asignación de este a la propiedad DataSource del control FlexGrid. Esta propiedad es la que conecta los datos del Recordset con el control para poder ser visualizados. En el código fuente 427 podemos ver el código del evento Load() de la ventana frmFlexGrid.

```
Private Sub Form_Load()
Dim lacnComercios As ADODB.Connection
Dim larsPapelerias As ADODB.Recordset
Dim lcComando As String
' crear objeto Connection, establecer propiedades
```

```
' y abrir
Set lacnComercios = New ADODB.Connection
lacnComercios.Provider = "MSDataShape"
lacnComercios.ConnectionString = "Data Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnComercios.Open
' crear objeto Recordset
Set larsPapelerias = New ADODB.Recordset
' definir comando SHAPE para
' crear el recordset jerárquico
lcComando = "SHAPE {SELECT * FROM Papelerias} AS TblPapelerias " & _
   "APPEND ( {SELECT * FROM Empleados} AS TblEmpleados " & _
   "RELATE IDPapeleria TO IDPapeleria ) AS TblEmpleados"

' abrir Recordset
larsPapelerias.Open lcComando, lacnComercios
' asignar el Recordset al FlexGrid
Set Me.grdPapelerias.DataSource = larsPapelerias
End Sub
```

Código fuente 427

El formulario en ejecución tendrá un aspecto parecido al de la figura 478. Si pulsamos en el bitmap situado en la primera columna, podremos expandir/contraer los registros mostrados por el control.

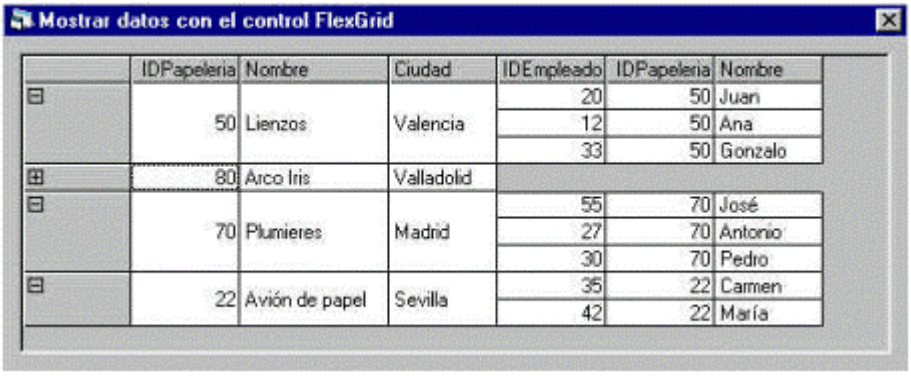


Figura 478. Control Hierarchical FlexGrid en ejecución.

Obtener varios detalles desde una instrucción principal

El modelado de datos no está limitado a una simple relación maestro/detalle, sino que es posible especificar por ejemplo, que a partir de una instrucción principal obtengamos más de un detalle en el mismo nivel, simplemente añadiendo nuevas cláusulas Relate a la instrucción Shape.

Partiendo del ejemplo anterior, si además de obtener los empleados por papelería, queremos también visualizar los artículos que se venden en cada una, escribiremos una sentencia Shape como la que se muestra en el código fuente 428, que pertenece al formulario frmRelacionMult encargado de mostrar este tipo de recordset jerárquico, y al que accederemos mediante la opción Modelado de datos+Relación múltiple del formulario principal.

```
Private Sub Form_Load()
Dim lacnComercios As ADODB.Connection
Dim larsPapelerias As ADODB.Recordset
Dim lcComando As String
```

```
' crear objeto Connection, establecer propiedades
' y abrir
Set lacnComercios = New ADODB.Connection
lacnComercios.Provider = "MSDataShape"
lacnComercios.ConnectionString = "Data Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnComercios.Open
' crear objeto Recordset
Set larsPapelerias = New ADODB.Recordset
' definir comando SHAPE para el recordset jerárquico,
' con dos recordsets dependientes en un
' mismo nivel
lcComando = "SHAPE {SELECT * FROM Papelerias} AS TblPapelerias " & _
   "APPEND ( {SELECT * FROM Empleados} AS TblEmpleados " & _
   "RELATE IDPapeleria TO IDPapeleria ) AS TblEmpleados, " & _
   "( {SELECT * FROM Articulos} AS TblArticulos " & _
   "RELATE IDPapeleria TO IDPapeleria ) AS TblArticulos "
' abrir Recordset
larsPapelerias.Open lcComando, lacnComercios
' asignar el Recordset al FlexGrid
Set Me.grdPapelerias.DataSource = larsPapelerias
End Sub
```

Código fuente 428

El curioso resultado de este modelado de datos podemos verlo en la figura 479.

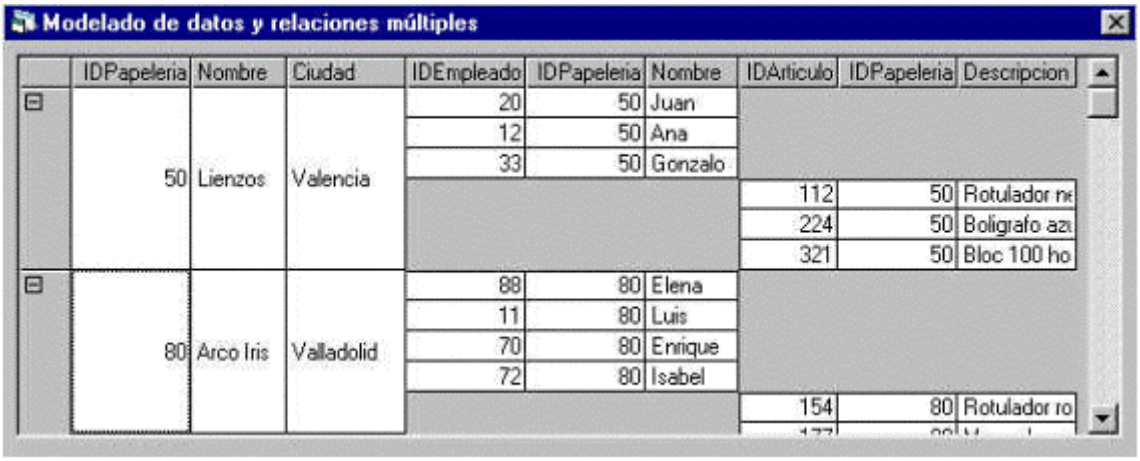


Figura 479. Recordset jerárquico con relación maestro/dos tipos de detalle.

Recuperar datos desde varios niveles de profundidad

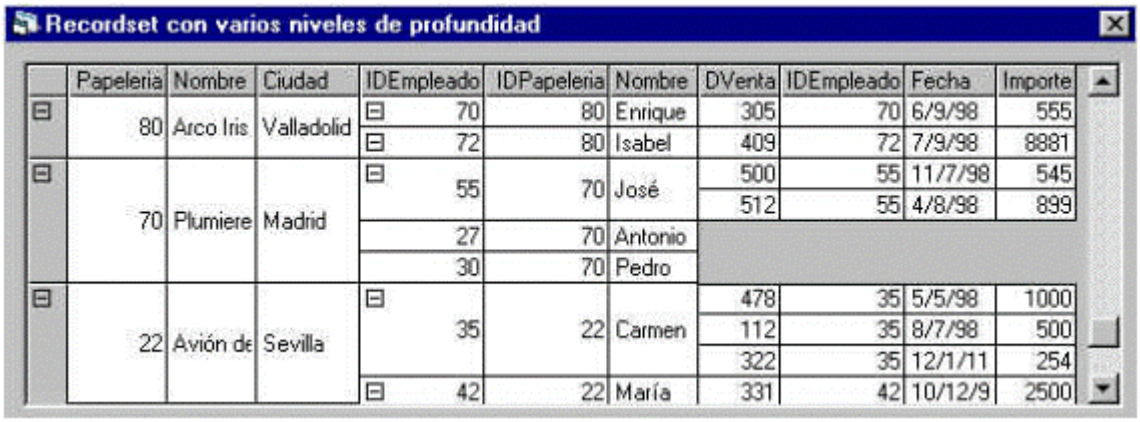
Es posible crear un recordset jerárquico que contenga datos relacionados a más de un nivel de profundidad. Para ello debemos incluir en la instrucción Shape, nuevas instrucciones Shape anidadas en función del nivel de profundidad que necesitemos conseguir.

Este es el caso del ejemplo mostrado en la opción de menú Modelado de datos+Varios niveles de profundidad de mdiInicio, que carga el formulario frmVariosNiveles, en el que se crea un recordset jerárquico que muestra la información de la tabla Papelerias en el nivel superior. Como detalle se visualizan los datos de la tabla Empleados relacionados con la tabla Papelerias por el campo IDPapeleria, finalizando con un detalle a un nivel más de profundidad, en el que se mostrarán los datos de la tabla Ventas relacionados con la tabla Empleados por el campo IDEmpleado. Veamos en el código fuente 429 el evento de carga de este formulario.

```
Private Sub Form_Load()
Dim lacnComercios As ADODB.Connection
Dim larsPapelerias As ADODB.Recordset
Dim lcComando As String
' crear objeto Connection, establecer propiedades
' y abrir
Set lacnComercios = New ADODB.Connection
lacnComercios.Provider = "MSDataShape"
lacnComercios.ConnectionString = "Data Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnComercios.Open
' crear objeto Recordset
Set larsPapelerias = New ADODB.Recordset
' definir comando SHAPE para
' Recordset jerárquico con varios
' niveles de profundidad
lcComando = "SHAPE {SELECT * FROM Papelerias} AS TblPapelerias " & _
   "APPEND (( SHAPE {SELECT * FROM Empleados} AS TblEmpleados " & _
      "APPEND ( {SELECT * FROM Ventas} AS TblVentas " & _
      "RELATE IDEmpleado TO IDEmpleado ) AS TblVentas ) AS TblEmpleados " & _
   "RELATE IDPapeleria TO IDPapeleria ) AS TblEmpleados"
' abrir Recordset
larsPapelerias.Open lcComando, lacnComercios
' asignar a control HFlexGrid
Set Me.grdPapelerias.DataSource = larsPapelerias
End Sub
```

Código fuente 429

El resultado lo podemos observar en la figura 480.



Papeleria	Nombre	Ciudad	IDEmpleado	IDPapeleria	Nombre	DVenta	IDEmpleado	Fecha	Importe
80	Arco Iris	Valladolid	70	80	Enrique	305	70	6/9/98	555
			72	80	Isabel	409	72	7/9/98	8881
70	Plumiere	Madrid	55	70	José	500	55	11/7/98	545
						512	55	4/8/98	899
			27	70	Antonio				
			30	70	Pedro				
22	Avión de	Sevilla	35	22	Carmen	478	35	5/5/98	1000
						112	35	8/7/98	500
						322	35	12/1/11	254
			42	22	María	331	42	10/12/9	2500

Figura 480. Recordset jerárquico con varios niveles de profundidad.

El lector podrá apreciar que para los empleados que no hayan realizado ninguna venta, no existirán datos en las columnas correspondientes a las ventas efectuadas. También es importante destacar, que al existir un anidamiento de instrucciones Shape, en la columna correspondiente al código del empleado también aparecerá el bitmap que permite contraer/expander los datos de sus columnas dependientes.

Realizar cálculos mediante modelado de datos

Empleando la cláusula Compute de la instrucción Shape, es posible efectuar una serie de cálculos sobre los registros de una consulta, devolviendo el resultado en forma de columna.

Sintaxis:

```
SHAPE {ComandoPrincipal} [AS] [TablaPrincipal]
    COMPUTE Campos-Func
    BY CamposAgrupac
```

- ComandoPrincipal. Selección de registros sobre los que se va a realizar un cálculo.
- TablaPrincipal. Nombre que utilizaremos en el código para referirnos al Recordset que devuelve la ejecución de este comando.
- Campos-Func. Lista de campos sobre los que se va a realizar el cálculo junto con la función a aplicar. La tabla 93 relaciona las funciones disponibles.

Función	**Descripción**
SUM(Tabla.Campo_Calcular)	Calcula la suma de todos los valores en el campo especificado.
AVG(Tabla.Campo_Calcular)	Calcula la media de todos los valores en el campo especificado.
MAX(Tabla.Campo_Calcular)	Calcula el valor máximo en el campo especificado.
MIN(Tabla.Campo_Calcular)	Calcula el valor mínimo en el campo especificado.
COUNT(Tabla[.Campo_Calcular])	Cuenta el número de filas en el campo especificado.
STDEV(Tabla.Campo_Calcular_NivelI Inferior)	Calcula la desviación en el campo especificado.
ANY(Tabla.Campo_Calcular)	El valor de una columna (donde el valor de la columna es el mismo para todas las filas).
CALC(Expresión)	Calcula una expresión arbitraria, pero sólo en la fila actual.
NEW (TipoCampo [(Ancho | Escala [,Precisión])]	Añade una columna vacía del tipo especificado al Recordset.

Tabla 93. Funciones disponibles para la cláusula Compute de la instrucción Shape.

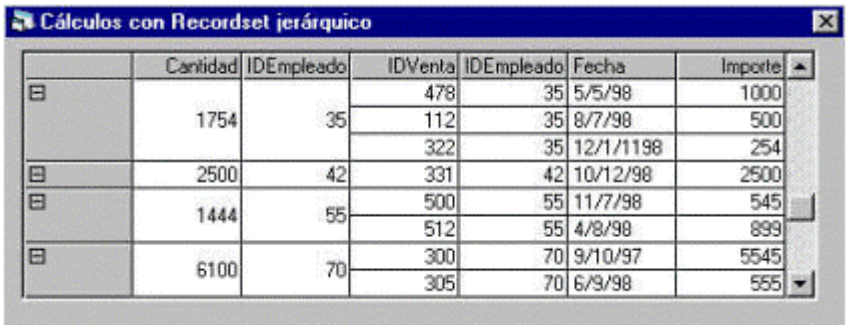
- CamposAgrupac. Uno o más nombres de columna que indica el tipo de agrupación de campos para aplicar la función.

El ejemplo contenido en la opción Modelado de datos+Realizar cálculos, del menú correspondiente al formulario principal, suma los registros de la tabla Ventas, separando los resultados por empleado. En el código fuente 430 podemos ver el código para crear este recordset.

```
Private Sub Form_Load()
Dim lacnComercios As ADODB.Connection
Dim larsVentas As ADODB.Recordset
Dim lcComando As String
' crear objeto Connection, establecer propiedades
' y abrir
Set lacnComercios = New ADODB.Connection
lacnComercios.Provider = "MSDataShape"
lacnComercios.ConnectionString = "Data Provider=Microsoft.Jet.OLEDB.3.51;" & _
   "Data Source=" & gcRutaDatos
lacnComercios.Open
' crear objeto Recordset
Set larsVentas = New ADODB.Recordset
' definir comando SHAPE para
' Recordset jerárquico con varios
' niveles de profundidad
lcComando = "SHAPE {SELECT * FROM Ventas} AS TblVentas " & _
   "COMPUTE TblVentas, SUM(TblVentas.Importe) AS Cantidad " & _
   "BY IDEmpleado"
' abrir Recordset
larsVentas.Open lcComando, lacnComercios
' asignar a control HFlexGrid
Set Me.grdPapelerias.DataSource = larsVentas
End Sub
```

Código fuente 430

El resultado lo podemos apreciar en la figura 481.



	Cantidad	IDEmpleado	IDVenta	IDEmpleado	Fecha	Importe
⊟	1754	35	478	35	5/5/98	1000
			112	35	8/7/98	500
			322	35	12/1/1198	254
⊟	2500	42	331	42	10/12/98	2500
⊟	1444	55	500	55	11/7/98	545
			512	55	4/8/98	899
⊟	6100	70	300	70	9/10/97	5545
			305	70	6/9/98	555

Figura 481. Recordset jerárquico que suma los valores de una columna en base a una condición.

Hasta aquí hemos comprobado alguna de las capacidades que pone a nuestra disposición el modelado de datos en ADO. Debido sin embargo a que la creación de este tipo de recordsets jerárquicos puede ser relativamente compleja si intentamos realizarla creando nosotros mismos el código, el apartado dedicado al Entorno de Datos o DataEnvironment nos muestra como podemos desde un interfaz visual más sencillo, crear este tipo de jerarquías de datos sin necesidad de luchar con las instrucciones para su creación.

13

Manipulación de ficheros

¿Por qué usar ficheros?

Durante el desarrollo de una aplicación, puede darse el caso de que nos encontremos en la necesidad de guardar una determinada cantidad de información para ser usada en una ejecución posterior de esa misma aplicación o de otra que necesite consultar dichos datos.

Debido al diseño del programa, puede que no sea conveniente o que no estemos interesados en emplear una base de datos para guardar la información. Algunos de los motivos de esta decisión pueden ser: el que la cantidad de datos a manipular sea pequeña y no merezca la pena el uso de un gestor de datos, o bien, que vayamos a realizar un seguimiento de la ejecución del programa, grabando ciertas acciones del usuario en un fichero de log, por lo cual será más conveniente el uso de un fichero de texto para efectuar dicho seguimiento.

Ante las anteriores situaciones o cualquier otra que se pudiera plantear, Visual Basic nos proporciona un amplio conjunto de instrucciones y una jerarquía de objetos para la manipulación de ficheros; con los que podemos efectuar operaciones de lectura/escritura tanto de texto como de información binaria sobre ficheros.

En los siguientes apartados veremos las instrucciones básicas para abrir y cerrar ficheros. Posteriormente y en función del tipo de acceso, se mostrará el modo de leer y escribir información en tales ficheros.

Apertura de un fichero

- Open. Mediante esta instrucción, podemos abrir un fichero y dejarlo preparado para efectuar sobre él operaciones de lectura/escritura.

  Sintaxis.

  ```
  Open cNombreFichero [For ModoApertura] [Access TipoAcceso]
  [TipoBloqueo] As [#]nNumeroFichero [Len = nLongitudRegistro]
  ```

  - cNombreFichero. Cadena con el nombre del fichero a abrir. Puede incluir la ruta donde se encuentra el fichero. En el caso de que no se especifique la ruta, se abrirá el fichero en el directorio actual de la aplicación.

  - ModoApertura. Una de las siguientes palabras clave, que indican como se va a abrir el fichero en función de las operaciones que vayamos a efectuar sobre él: Append, Binary, Input, Output, Random.

  - TipoAcceso. Palabra clave que establece las operaciones a realizar en el fichero: Read, Write o Read Write.

  - TipoBloqueo. Palabra clave que especifica el tipo de bloqueo a realizar sobre el fichero: Shared, Lock Read, Lock Write, Lock Read Write.

  - nNumeroFichero. Valor numérico comprendido en el rango de 1 a 511, utilizado por Open para identificar cada uno de los ficheros abiertos por la aplicación.

  - nLongitudRegistro. Número igual o inferior a 32767. Si el fichero se abre para acceso aleatorio, este número indica la longitud del registro. Si el fichero se abre para acceso secuencial, indica el número de caracteres situados en la zona de memoria al realizar las operaciones de lectura y escritura. La cláusula Len se ignora cuando el modo de apertura es Binary.

  Si el archivo indicado en cNombreFichero no existe al intentar abrirlo, se creará un nuevo archivo vacío con ese mismo nombre cuando el modo de apertura sea uno de los siguientes: Append, Binary, Output, Random.

  Si al intentar abrir un fichero, ya estuviera abierto y bloqueado para no permitir otros accesos, o el modo de apertura fuera Input, se produciría un error.

  Un fichero abierto se puede abrir de nuevo con un número distinto, siempre y cuando el modo de apertura sea Binary, Input y Random. Sin embargo, para abrir un fichero en los modos Append y Output, dicho fichero debe estar cerrado y no ser utilizado por ningún otro proceso.

  En lo que respecta al número identificativo del fichero abierto, para asegurarnos que no intentamos utilizar un número ya en uso por otro proceso, podemos emplear la función FreeFile(), que devuelve un número de fichero libre para poder ser utilizado con Open.

  Sintaxis.

  ```
  FreeFile [(nNumeroIntervalo)]
  ```

  - nNumeroIntervalo. Especifica el rango del que se va a tomar el número de fichero:

- 0. Valor por defecto, devuelve un número de 1 a 255.
- 1. Devuelve un número de 256 a 511.

Cierre de un fichero

Para cerrar uno o más ficheros abiertos con la instrucción Open, se utilizan las siguientes instrucciones:

- Close. Cierra uno o varios ficheros abiertos, pasando como parámetro los números de fichero.

  Sintaxis

  ```
  Close [ListaNumFichero]
  ```

  - ListaNumFichero. Indica el número o números de fichero que se van a cerrar de la siguiente forma: Close #NumFicheroA, #NumFicheroB, ... Si no se indica el número de fichero, se cierran todos los ficheros abiertos.

- Reset. Cierra todos los ficheros abiertos, volcando previamente a disco, el contenido de todas las zonas de memoria reservadas a los ficheros.

Modos de acceso a ficheros

En el apartado dedicado a la instrucción Open, hemos visto que hay varias formas de abrir un fichero, en función de como necesitemos escribir o acceder a la información que contiene.

En los siguientes apartados, veremos el resto de instrucciones y funciones relacionadas con la manipulación de ficheros, dependiendo del modo de acceso a efectuar sobre los mismos. Utilizaremos como ejemplo la aplicación ManipFic, y a través de las opciones de menú de su ventana principal mdiInicio, veremos las diferentes operaciones a efectuar según el modo de acceso.

Esta aplicación comienza mediante un procedimiento Main(), en el cual usamos el objeto del sistema App, para establecer como directorio de trabajo el mismo directorio en el que está situada la aplicación, según muestra el código fuente 431.

```
Public Sub Main()
Dim lmdiInicio As mdiInicio
' establecer como directorio actual
' el directorio donde está situada la
' aplicación
ChDir App.Path
' cargar la ventana principal
Set lmdiInicio = New mdiInicio
Load lmdiInicio
lmdiInicio.Show
End Sub
```

Código fuente 431

El formulario mdiInicio, dispone del control CommonDialog dlgFicheros, que emplearemos para abrir ficheros a lo largo de los diferentes ejemplos de esta aplicación.

Acceso Secuencial

Este es el modo de acceso más sencillo a un fichero. Se utiliza habitualmente para leer o escribir texto en el fichero.

Apertura

Cuando abramos un fichero para este tipo de acceso, utilizaremos alguna de las siguientes palabras clave en la cláusula For de la instrucción Open:

- Input. Abre el fichero para leer la información que contiene desde el principio del mismo.
- Output. Abre el fichero para escribir información. Si dicho fichero ya existe, su contenido se elimina, en el caso de que no exista, se crea un nuevo fichero.
- Append. Abre el fichero para escribir. La nueva información se añade a partir de los últimos datos existentes.

Lectura

Para leer información de un fichero abierto en modo de acceso secuencial, emplearemos las siguientes instrucciones:

- Input #. Lee información de un fichero secuencial asignándola a variables.

  Sintaxis.

  ```
  Input #nNumFichero, [ListaVariables]
  ```

  - nNumFichero. Número de fichero del que se va a leer.
  - ListaVariables. Lista de variables separadas por comas o punto y coma, en las que se irá introduciendo la información leída.

  Para comprobar como funciona esta instrucción, seleccionaremos la opción Secuencial + lectura Input del menú de mdiInicio, que ejecutará el código fuente 432.

```
Private Sub mnuSeLecInput_Click()
Dim lcFichero As String
Dim lcNombreFich As String
Dim lnManip As Integer
Dim lcUna As String
Dim lcDos As String
Dim lcTres As String
lcNombreFich = "SecuenLee.txt"
' establecer título del diálogo
Me.dlgFicheros.DialogTitle = "Acceso Secuencial - lectura con Input"
' establecer fichero a abrir
```

```
Me.dlgFicheros.filename = lcNombreFich
' abrir diálogo
Me.dlgFicheros.ShowOpen
lcFichero = Me.dlgFicheros.FileTitle
' si no se selecciona el fichero correcto,
' salir del procedimiento
If lcNombreFich <> Me.dlgFicheros.FileTitle Then
   MsgBox "El fichero que se debe seleccionar es: " & _
      Me.dlgFicheros.filename, , "Error"

   Exit Sub
End If
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso secuencial y lectura
Open Me.dlgFicheros.filename For Input As lnManip
' tomar varias líneas del fichero y traspasarlas
' a variables
Input #lnManip, lcUna, lcDos, lcTres
' visualizar las variables
MsgBox "lcUna: " & lcUna, , "Contenido de la variable"
MsgBox "lcDos: " & lcDos, , "Contenido de la variable"
MsgBox "lcTres: " & lcTres, , "Contenido de la variable"
' cerrar el fichero
Close #lnManip
End Sub
```

Código Fuente 432

- Line Input #. Lee una línea de un fichero secuencial y la asigna a una variable. Cualquier caracter usado como delimitador, es incluido en la variable.

  Sintaxis.

  ```
  Line Input #nNumFichero, cVariable
  ```

  - nNumFichero. Número de fichero del que se va a leer.

  - cVariable. Variable de tipo cadena, en la que se va a introducir el valor de la línea del fichero.

  Ejecutando la opción del formulario principal Secuencial+Lectura Line Input, leeremos el mismo fichero que en el ejemplo anterior, pero tomando la información por líneas del fichero en vez de por los delimitadores. El código fuente 433 muestra como se realiza esta tarea.

```
Private Sub mnuSeLecLineInput_Click()
Dim lcFichero As String
Dim lcNombreFich As String
Dim lnManip As Integer
Dim lcUna As String
Dim lcDos As String
Dim lcTres As String
lcNombreFich = "SecuenLee.txt"
' establecer título del diálogo
Me.dlgFicheros.DialogTitle = "Acceso Secuencial - lectura con Line Input"
' establecer fichero a abrir
Me.dlgFicheros.filename = lcNombreFich
' abrir diálogo
Me.dlgFicheros.ShowOpen
```

```
lcFichero = Me.dlgFicheros.FileTitle
' si no se selecciona el fichero correcto,
' salir del procedimiento
If lcNombreFich <> Me.dlgFicheros.FileTitle Then
   MsgBox "El fichero que se debe seleccionar es: " & _
      Me.dlgFicheros.filename, , "Error"

   Exit Sub
End If
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso secuencial y lectura
Open Me.dlgFicheros.filename For Input As lnManip
' tomar varias líneas del fichero y traspasarlas
' a variables
Line Input #lnManip, lcUna
Line Input #lnManip, lcDos
Line Input #lnManip, lcTres
' visualizar las variables
MsgBox "lcUna: " & lcUna, , "Contenido de la variable"
MsgBox "lcDos: " & lcDos, , "Contenido de la variable"
MsgBox "lcTres: " & lcTres, , "Contenido de la variable"
' cerrar el fichero
Close #lnManip
End Sub
```

Código fuente 433

- Input( ). Función que toma un número determinado de caracteres de un fichero secuencial y los asigna a una variable. Entre los caracteres leídos se pueden incluir delimitadores, finales de línea y retorno de carro, etc.

  Sintaxis.

  ```
  Input(cNumCaracteres, #nNumFichero)
  ```

  - nNumCaracteres. Número de caracteres a leer del fichero.

  - nNumFichero. Número de fichero del que se va a leer.

  La opción Secuencial+Lectura función Input de la aplicación de ejemplo, toma la información del fichero de texto empleado anteriormente. Como podrá observar el lector, el texto es leído según el número de caracteres que se necesiten extraer, sin tener en cuenta delimitadores, finales de línea o cualquier otro elemento, como vemos en el código fuente 434.

```
Private Sub mnuSeLecFuncInput_Click()
Dim lcFichero As String
Dim lcNombreFich As String
Dim lnManip As Integer
Dim lcUna As String
Dim lcDos As String
Dim lcTres As String
Dim lcCuatro As String
lcNombreFich = "SecuenLee.txt"
' establecer título del diálogo
Me.dlgFicheros.DialogTitle = "Acceso Secuencial - lectura con función
Input()"
' establecer fichero a abrir
```

```
Me.dlgFicheros.filename = lcNombreFich
' abrir diálogo
Me.dlgFicheros.ShowOpen
lcFichero = Me.dlgFicheros.FileTitle
' si no se selecciona el fichero correcto,
' salir del procedimiento
If lcNombreFich <> Me.dlgFicheros.FileTitle Then
   MsgBox "El fichero que se debe seleccionar es: " & _
      Me.dlgFicheros.filename, , "Error"
   Exit Sub
End If
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso secuencial y lectura
Open Me.dlgFicheros.filename For Input As lnManip
' tomar varias cadenas del fichero y traspasarlas
' a variables
lcUna = Input(14, #lnManip)
lcDos = Input(9, #lnManip)
lcTres = Input(17, #lnManip)
lcCuatro = Input(12, #lnManip)
' visualizar las variables
MsgBox "lcUna: " & lcUna, , "Contenido de la variable"
MsgBox "lcDos: " & lcDos, , "Contenido de la variable"
MsgBox "lcTres: " & lcTres, , "Contenido de la variable"
MsgBox "lcCuatro: " & lcCuatro, , "Contenido de la variable"
' cerrar el fichero
Close #lnManip
End Sub
```

Código fuente 434

Escritura

Para escribir los datos en un fichero secuencial, emplearemos las siguientes instrucciones:

- Print #. Escribe información en un fichero secuencial, la salida de los datos a escribir se realiza de la misma forma que si empleáramos la instrucción Print para mostrar datos en pantalla.

  Sintaxis.

  ```
  Print #nNumFichero, [ListaExpresiones]
  ```

  - nNumFichero. Número de fichero en el que se va a escribir.

  - ListaExpresiones. Expresiones a escribir en el fichero. Cada una de las expresiones puede tener el siguiente formato: [Spc(nCaracteres) | Tab[(nColumna)]] [Expresión] [cDelimitExpr]

  - Spc(nCaracteres). Inserta nCaracteres en blanco al escribir la expresión en el fichero.

  - Tab(nColumna). Sitúa el punto de escritura de la expresión en la columna indicada por nColumna.

  - Expresión. Valor a escribir en el fichero.

- cDelimitExpr. Caracter, por ejemplo punto y coma ";", para indicar que finaliza la escritura de una expresión y comienza la siguiente.

Veamos un ejemplo de este tipo de escritura en ficheros, seleccionando la opción Secuencial+Escritura Print de la aplicación de ejemplo, que contiene el código fuente 435.

```
Private Sub mnuSeEscPrint_Click()
Dim lnManip As Integer
Dim lbValor As Boolean
Dim ldtFecha As Date
Dim lxSinValor
Dim lxError
' tomar un manipulador de fichero libre
lnManip = FreeFile
' asignar valores a variables cuyo contenido
' se va a escribir en el fichero
lbValor = True
ldtFecha = CDate("05/09/1998")
lxSinValor = Null
lxError = CVErr(250)
' abrir el fichero para acceso secuencial y escritura
Open "SecuenEsc1.txt" For Output As lnManip
' escribir datos en el fichero
Print #lnManip, "prueba"; "de escritura"; "con PRINT"
Print #lnManip, "prueba "; "de escritura "; "con PRINT"
Print #lnManip,
Print #lnManip, "prueba"; Spc(2); "de escritura"; Spc(4); "con PRINT"
Print #lnManip, Tab(2); "prueba "; Tab(16); "de escritura"; Tab(30); " con
PRINT"
Print #lnManip,
Print #lnManip, lbValor
Print #lnManip, ldtFecha
Print #lnManip, lxSinValor
Print #lnManip, lxError
' cerrar el fichero
Close #lnManip
MsgBox "Finalizada la escritura con Print", , "Acceso Secuencial"
End Sub
```

Código fuente 435

Si abrimos el fichero resultante para visualizar su contenido mediante el Bloc de notas de Windows, podremos observar que la escritura de la información especial tal como valores lógicos, fecha, error, etc., son escritos adaptándose a la configuración local del sistema operativo. En lo que respecta al texto normal, ha de ser el programador el encargado de incluir los espacios o tabuladores necesarios para que el texto a escribir sea legible.

- Write #. Escribe una lista de expresiones a un fichero secuencial.

  Sintaxis.

  ```
  Write #nNumFichero, [ListaExpresiones]
  ```

  La lista de expresiones a escribir, irá separada por espacios, coma o punto y coma. La ventaja de esta instrucción es que incluye automáticamente los delimitadores en el fichero e inserta un final de línea al escribir la última expresión de la lista, lo que facilita la posterior lectura del fichero.

La opción Secuencial+Escritura Write, escribe una serie de cadenas de texto en un fichero. Si abrimos el fichero después de esta operación, comprobaremos que en dichas cadenas se ha incluido automáticamente un delimitador, para facilitar su posterior lectura. En el código fuente 436 se puede ver el código de esta opción.

```
Private Sub mnuSeEscWrite_Click()
Dim lnManip As Integer
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso secuencial y escritura
Open "SecuenEsc2.txt" For Output As lnManip
' escribir datos en el fichero
Write #lnManip, "prueba"; "de escritura"; "con WRITE"
Write #lnManip, "prueba "; "de escritura "; "con WRITE"
Write #lnManip, "escritura del número:"; 512
' cerrar el fichero
Close #lnManip
MsgBox "Finalizada la escritura con Write", , "Acceso Secuencial"
End Sub
```

Código fuente 436

En cuanto a la información especial: fechas, valores lógicos, errores, etc. El tratamiento que Write hace de los mismos, difiere del realizado por Print. Write escribe este tipo de datos según unas normas de formato estándar que no tienen en cuenta la configuración local del sistema. La opción Secuencial+Escritura Añadir con Write, agrega al fichero utilizado en el anterior ejemplo con Write una serie de líneas con este tipo de información, de manera que podamos ver la diferencia entre la escritura con Print y Write, además de comprobar como se agregan datos a un fichero ya existente, manteniendo la información que hubiera en el mismo. El código fuente 437 pertenece a esta opción.

```
Private Sub mnuSeEscAdd_Click()
Dim lnManip As Integer
Dim lbValor As Boolean
Dim ldtFecha As Date
Dim lxSinValor
Dim lxError

' tomar un manipulador de fichero libre
lnManip = FreeFile

' asignar valores a variables cuyo contenido
' se va a escribir en el fichero
lbValor = True
ldtFecha = CDate("05/09/1998")
lxSinValor = Null
lxError = CVErr(250)

' abrir el fichero para acceso secuencial y escritura,
' la información a escribir se añadirá a la ya existente
Open "SecuenEsc2.txt" For Append As lnManip

' escribir datos en el fichero
Write #lnManip,
Write #lnManip, lbValor
Write #lnManip, ldtFecha
Write #lnManip,
Write #lnManip, lxSinValor
Write #lnManip, lxError
```

```
' cerrar el fichero
Close #lnManip
MsgBox "Finalizada la escritura con Write (añadir)", , "Acceso Secuencial"
End Sub
```

Código fuente 437

Acceso Aleatorio

En un fichero diseñado para acceso aleatorio, la información está organizada en registros y estos a su vez en campos.

Apertura

Para abrir un fichero con acceso aleatorio, emplearemos la palabra clave Random en la cláusula For de la instrucción Open, y especificaremos la longitud del registro mediante la cláusula Len de la misma instrucción.

Escritura

- Put. Esta instrucción escribe un registro en un fichero abierto para acceso aleatorio. Si se intentan escribir más datos para un registro, que los especificados en la cláusula Len de la instrucción Open, se producirá un error.

  Sintaxis.

  ```
  Put #nNumFichero, [nNumRegistro], cVariable
  ```

  - nNumFichero. Número de fichero en el que se va a escribir.
  - nNumRegistro. Número de registro a grabar. Si se omite, se escribe el siguiente registro que corresponda después de haber ejecutado la última instrucción Get o Put.
  - cVariable. Nombre de la variable que contiene los datos a escribir en el fichero.

Para manejar registros, la mejor manera es emplear tipos definidos por el usuario. En los ejemplos de acceso aleatorio, se ha definido a nivel de módulo el tipo Ficha, cuyo fuente vemos en el código fuente 438.

```
Private Type Ficha
   Codigo As String * 2
   Nombre As String * 10
   Ciudad As String * 15
End Type
```

Código fuente 438

La opción de menú Aleatorio+Escritura de los ejemplos, nos muestra como escribir registros en un fichero utilizando esta instrucción y una variable del tipo Ficha.

```
Private Sub mnuAlEscritura_Click()
Dim lnManip As Integer
Dim lFicha As Ficha
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso aleatorio
Open "Aleatorio.txt" For Random As lnManip Len = Len(lFicha)
' escribir registros en el fichero
lFicha.Codigo = "10"
lFicha.Nombre = "Pedro"
lFicha.Ciudad = "Burgos"
Put #lnManip, 1, lFicha
lFicha.Codigo = "20"
lFicha.Nombre = "Antonio"
lFicha.Ciudad = "Salamanca"
Put #lnManip, 2, lFicha
lFicha.Codigo = "40"
lFicha.Nombre = "Raquel"
lFicha.Ciudad = "Ávila"
Put #lnManip, 3, lFicha
lFicha.Codigo = "45"
lFicha.Nombre = "Ana"
lFicha.Ciudad = "Barcelona"
Put #lnManip, 4, lFicha
lFicha.Codigo = "80"
lFicha.Nombre = "Verónica"
lFicha.Ciudad = "Madrid"
Put #lnManip, 5, lFicha
lFicha.Codigo = "82"
lFicha.Nombre = "Luis"
lFicha.Ciudad = "Valencia"
Put #lnManip, 6, lFicha
' cerrar el fichero
Close #lnManip
MsgBox "Finalizada la escritura con Put", , "Acceso Aleatorio"
End Sub
```

Código fuente 439

Lectura

- Get. Lee un registro de un archivo abierto en modo aleatorio, almacenando el registro en una variable.

  Sintaxis.

  ```
  Get #nNumFichero, nNumRegistro, cVariable
  ```

  - nNumFichero. Número de fichero del que se va a leer.
  - nNumRegistro. Número de registro a leer. Si se omite, se lee el siguiente registro que corresponda después de haber ejecutado la última instrucción Get o Put.
  - cVariable. Nombre de la variable en la que se va a situar el registro leído.

La opción Aleatorio+Lectura de la aplicación de ejemplo, ilustra la forma de recorrer los registros del fichero grabado en el ejemplo anterior y mostrarlos al usuario.

```
Private Sub mnuAlLectura_Click()
Dim lcFichero As String
Dim lcNombreFich As String
Dim lnManip As Integer
Dim lFicha As Ficha
Dim lnRegistro As Integer
lcNombreFich = "Aleatorio.txt"
' establecer título del diálogo
Me.dlgFicheros.DialogTitle = "Acceso Aleatorio - Lectura"
' establecer fichero a abrir
Me.dlgFicheros.filename = lcNombreFich
' abrir diálogo
Me.dlgFicheros.ShowOpen
lcFichero = Me.dlgFicheros.FileTitle
' si no se selecciona el fichero correcto,
' salir del procedimiento
If lcNombreFich <> Me.dlgFicheros.FileTitle Then
   MsgBox "El fichero que se debe seleccionar es: " & _
      Me.dlgFicheros.filename, , "Error"
   Exit Sub
End If
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso aleatorio
Open Me.dlgFicheros.filename For Random As lnManip Len = Len(lFicha)
' inicializar contador de registros
lnRegistro = 1
' recorrer el fichero hasta el final
Do While Not EOF(lnManip)
   ' tomar un registro del fichero
   Get #lnManip, lnRegistro, lFicha
   ' visualizar el registro
   MsgBox "Código: " & lFicha.Codigo & vbCrLf & _
      "Nombre: " & lFicha.Nombre & vbCrLf & _
      "Ciudad: " & lFicha.Ciudad, , "Contenido del registro"
   lnRegistro = lnRegistro + 1
Loop
' cerrar el fichero
Close #lnManip
End Sub
```

Código fuente 440

La función EOF(nNumFichero) utilizada aquí, sirve para saber cuando hemos llegado al final del fichero, cuyo número es pasado como parámetro, evitando que se produzca un error al intentar leer datos pasada la última posición del fichero. Esta función, devolverá False mientras se pueda leer información del fichero, y True una vez que se sobrepase la última posición del mismo.

Acceso Binario

El acceso binario a un fichero, nos permite manipular la información del mismo byte a byte, o dicho de otro modo caracter a caracter, sin tener en cuenta si la información del fichero está grabada bajo una determinada estructura de registros o no.

Apertura

Para abrir un fichero que disponga de acceso binario, emplearemos la palabra clave Binary en la cláusula For de la instrucción Open.

Escritura

Escribiremos información en un fichero abierto en modo binario mediante la instrucción Put, vista anteriormente al tratar el acceso aleatorio. En el caso de acceso binario, el parámetro nNumRegistro hará referencia a la posición o byte del fichero en la que se comienza a escribir.

La opción Binario+Escritura de la aplicación de ejemplo, nos muestra como escribir en un fichero, usando este tipo de acceso. En el código fuente 441 vemos el fuente de este punto.

```
Private Sub mnuBiEscritura_Click()
Dim lnManip As Integer
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso binario
Open "Binario.txt" For Binary As lnManip
' escribir información en el fichero
Put #lnManip, , "En ese parque hay diez árboles"
' cerrar el fichero
Close #lnManip
MsgBox "Finalizada la escritura con Put", , "Acceso Binario"
End Sub
```

Código fuente 441

Lectura

Leeremos los datos de un fichero abierto en modo binario mediante la instrucción Get, vista en el modo de acceso aleatorio. En acceso binario, el parámetro nNumRegistro hará referencia al número de bytes a leer del fichero.

En el código fuente 442 podemos ver el código perteneciente a la opción Binario+Lectura de la aplicación de ejemplo. En primer lugar, leemos una parte de la cadena escrita en el anterior ejemplo, para mostrarla al usuario y posteriormente sobrescribimos en el fichero dicha cadena por otra distinta, volviendo a mostrar el resultado.

```
Private Sub mnuBiLectura_Click()
Dim lnManip As Integer
Dim lcCadena As String * 4
' tomar un manipulador de fichero libre
lnManip = FreeFile
' abrir el fichero para acceso binario
Open "Binario.txt" For Binary As lnManip
' leer varios caracteres y pasarlos
' a una variable
Get #lnManip, 19, lcCadena
' mostrar el resultado
```

```
MsgBox "Valor leido del fichero: " & lcCadena, , "Acceso Binario"
' grabar una cadena distinta
' en la misma posición de la que se ha leido
Put #lnManip, 19, " 10 "
' volver a efectuar la operación de lectura
Get #lnManip, 19, lcCadena
MsgBox "Nuevo valor leido de la misma posición fichero: " _
   & lcCadena, , "Acceso Binario"
' cerrar el fichero
Close #lnManip
End Sub
```

Código fuente 442

Otras instrucciones para manejo de ficheros

A continuación se relaciona un grupo adicional de instrucciones y funciones que también se emplean en las operaciones con ficheros.

- Seek(). Función que devuelve la posición actual del fichero utilizado como parámetro. Si es un fichero abierto para acceso aleatorio, esta función devuelve el número de registro en el que está posicionado el fichero, para otros modos de acceso, devuelve el byte actual.

  Sintaxis.

  ```
  Seek(nNumFichero)
  ```

  - nNumFichero. Número de fichero del que se averigua la posición.

- Seek. Esta instrucción establece la posición del fichero sobre la que se va a realizar la siguiente operación de lectura o escritura.

  Sintaxis.

  ```
  Seek #nNumFichero, nPosicion
  ```

  - nNumFichero. Número de fichero en el que se establece la posición.
  - nPosicion. Posición del fichero a establecer.

  Si después de usar esta instrucción, utilizamos Get o Put especificando una posición en el fichero, esta última prevalecerá sobre la indicada en Seek.

- Loc(). Función que informa sobre la posición actual del fichero

  Sintaxis.

  ```
  Loc(nNumFichero)
  ```

  - nNumFichero. Número de fichero del que se obtiene la posición.

  Si el fichero está en modo binario, se obtiene la posición del último byte utilizado; si está en modo aleatorio, el último registro; finalmente si el acceso es secuencial, devuelve la posición del último byte dividida por 128.

- LOF(). Devuelve el tamaño de un fichero abierto.

  Sintaxis.

  ```
  LOF(nNumFichero)
  ```

  - nNumFichero. Número de fichero del que obtendremos su tamaño.

- FileLen(). Función que devuelve el tamaño de un fichero.

  Sintaxis.

  ```
  FileLen(cNombreFich)
  ```

  - cNombreFich. Nombre del fichero y ruta opcional, del que vamos a obtener su tamaño.

  La diferencia entre esta función y LOF() reside en que utilizando FileLen() no es necesario que el fichero esté abierto para conocer su tamaño.

- Kill. Borra ficheros del disco.

  Sintaxis.

  ```
  Kill cNombreFich
  ```

  - cNombreFich. Nombre del fichero/s y ruta opcional, que vamos a eliminar, es posible emplear los caracteres comodín "*" y "?" para borrar grupos de ficheros.

  Hemos de asegurarnos que el fichero/s a eliminar están cerrados o de lo contrario ocurrirá un error.

- Name. Esta instrucción permite cambiar de nombre un fichero o directorio.

  Sintaxis.

  ```
  Name cNombreFichActual As cNombreFichNuevo
  ```

  - cNombreFichActual. Nombre del fichero existente al que vamos a cambiar de nombre, opcionalmente puede incluir la ruta de acceso.

  - cNombreFichNuevo. Nombre del nuevo fichero y ruta opcional. No se puede utilizar el nombre de un fichero ya existente.

  Si el nombre de fichero empleado tanto en cNombreFichActual como en cNombreFichNuevo es igual, y las rutas de acceso son distintas, el fichero se cambiará de posición.

  Las siguientes instrucciones, si bien no operan directamente con ficheros, son necesarias en cuanto a la organización de los mismos.

- FileCopy. Copia un fichero

  Sintaxis.

```
FileCopy cNombreFichOrigen, cNombreFichDestino
```

- cNombreFichOrigen. Cadena con el nombre y ruta opcional del fichero que se va a copiar.

- cNombreFichDestino. Cadena con el nombre y ruta opcional del fichero en el que se va a efectuar la copia.

Si se intenta utilizar esta instrucción sobre un fichero abierto, se provocará un error.

- FileDateTime. Esta función retorna un valor de tipo fecha, que informa de la fecha y hora de creación de un fichero.

  Sintaxis.

  ```
  FileDateTime(cNombreFich)
  ```

  - cNombreFich. Cadena con el nombre y ruta opcional del fichero.

- FileAttr. Esta función devuelve información sobre un fichero abierto con Open.

  Sintaxis.

  ```
  FileAttr(nNumFichero, nTipoInfo)
  ```

  - nNumFichero. Número de fichero abierto.

  - nTipoInfo. Número que especifica el tipo de información, si es 1, se devolverá el modo del fichero, si es 2, se devolverá información sobre el selector de archivos del sistema operativo.

    Cuando se utiliza 1 en este parámetro, los valores que podemos obtener son los siguientes:

    - 1. Entrada.
    - 2. Salida.
    - 4. Aleatorio.
    - 8. Añadir.
    - 32. Binario.

- GetAttr. Devuelve un valor que identifica los atributos de un fichero, directorio o volumen.

  Sintaxis.

  ```
  GetAttr(cNombreFich)
  ```

  - cNombreFich. Cadena con el nombre y ruta opcional del fichero.

  El valor devuelto por esta función puede ser uno de los mostrados en la tabla 94.

Valor	Constante	Descripción
0	VbNormal	Normal.
1	vbReadOnly	Sólo lectura.
2	VbHidden	Oculto.
4	VbSystem	Archivo de sistema.
16	vbDirectory	Directorio o carpeta.
32	VbArchive	El archivo ha sido modificado después de efectuar la última copia de seguridad.

Tabla 94. Valores devueltos por la función GetAttr().

- SetAttr. Establece los atributos de un fichero.

  Sintaxis.

  ```
  SetAttr(cNombreFich, nAtributos)
  ```

  - cNombreFich. Cadena con el nombre y ruta opcional del fichero.
  - nAtributos. Uno o más valores numéricos o constantes que indica los atributos a establecer. Dichos valores son los mismos que devuelve la función GetAttr() excepto vbDirectory.

  El atributo se debe establecer con el fichero cerrado, en caso contrario, se producirá un error.

- MkDir. Crea un nuevo directorio.

  Sintaxis.

  ```
  MkDir cNombreDir
  ```

  - cNombreDir. Cadena con el nombre del directorio a crear. Si no se indica la unidad de disco, el directorio se crea en la unidad actual.

- RmDir. Borra un directorio existente.

  Sintaxis.

  ```
  RmDir cNombreDir
  ```

  - cNombreDir. Cadena con el nombre del directorio a borrar. Si no se indica la unidad de disco, se borra el directorio de la unidad actual.

  Si el directorio que se desea borrar contiene ficheros, se producirá un error. Se deben eliminar previamente los ficheros del directorio antes de proceder a borrarlo.

- CurDir. Devuelve una cadena con la ruta de acceso actual.

Sintaxis.

```
CurDir [(cUnidad)]
```

- cUnidad. Cadena con la unidad de disco de la que se desea saber la ruta de acceso, si no se indica, se devuelve la ruta de la unidad actual.

- ChDir. Cambia el directorio actual.

  Sintaxis.

  ```
  ChDir cNombreDir
  ```

  - cNombreDir. Cadena con el nuevo directorio, puede incluir también la unidad de disco, sin embargo, la unidad predeterminada no cambia aunque se cambie el directorio en una unidad diferente.

- ChDrive. Cambia la unidad de disco actual.

  Sintaxis.

  ```
  ChDrive cUnidad
  ```

  - cUnidad. Cadena con la nueva unidad.

Objetos para el manejo de ficheros

Visual Basic 6 incorpora una interesante y seguramente esperada novedad para la manipulación de ficheros, consistente en un modelo de objetos denominado File System Objects (Objetos del sistema de ficheros), FSO a partir de ahora. Decimos esperada novedad, puesto que ya que esta es una herramienta que progresivamente ha incorporado la tecnología de objetos para todas las tareas a realizar, en este aspecto del manejo de archivos sólo disponíamos del tradicional conjunto de instrucciones para la apertura, lectura y escritura de información en ficheros, echándose en falta cada vez más un grupo de objetos para realizar este trabajo.

Gracias a este conjunto de objetos, podremos realizar las operaciones habituales de lectura y escritura en modo secuencial sobre un fichero (para acceso aleatorio y binario debemos seguir empleando la instrucción Open), pero no sólo disponemos de objetos para la manipulación de ficheros, esta jerarquía nos permite acceder también a las carpetas y unidades instaladas en nuestro equipo.

La aplicación de ejemplo para este apartado, ManipFSO, muestra al igual que el anterior ejemplo, a través del formulario principal mdiInicio, diversas operaciones efectuadas con los objetos de este modelo, de los que seguidamente veremos las propiedades y métodos principales.

Drive

Facilita información sobre una unidad instalada en el equipo: disco duro, CD ROM, disco RAM, etc.

Propiedades

- AvailableSpace. Informa sobre la cantidad de espacio disponible en la unidad.
- DriveLetter. Devuelve la letra identificativa de la unidad.
- DriveType. Constante que indica el tipo de unidad instalada según la tabla 95.

Constante	Descripción
CDRom	CD-ROM
Fixed	Disco Duro
RamDisk	Disco RAM o Disco en memoria
Remote	Disco de Red
Removable	Removible
Unknown	Desconocido

Tabla 95. Constantes para la propiedad DriveType.

- FileSystem. Sistema de archivos empleado por la unidad: FAT, NTFS, etc.
- SerialNumber. Número de serie de la unidad.
- TotalSize. Tamaño total que tiene la unidad.
- VolumeName. Establece o devuelve el nombre de volumen para la unidad.

La opción de menú Drive+Mostrar información, en el programa, ejecuta líneas de código del código fuente 443, que toman la unidad C: instalada en el equipo del usuario y visualizan algunas de sus propiedades.

```
Private Sub mnuDrMostrar_Click()
Dim lfsoFichero As FileSystemObject
Dim ldrvUnidad As Drive
Dim lcInfo As String
Dim lcTipoUnidad As String

' crear objeto FSO
Set lfsoFichero = New FileSystemObject

' obtener objeto Drive
Set ldrvUnidad = lfsoFichero.GetDrive("c:")

' mostrar propiedades del objeto
' comprobar el tipo de unidad
Select Case ldrvUnidad.DriveType
Case CDRom
   lcTipoUnidad = "CDRom"
```

```
Case Fixed
   lcTipoUnidad = "Disco Duro"
Case RamDisk
   lcTipoUnidad = "Disco en memoria"
Case Removable
   lcTipoUnidad = "Removible"
Case Remote
   lcTipoUnidad = "Red"
Case Unknown
   lcTipoUnidad = "Desconocida"
End Select
lcInfo = "Unidad: " & ldrvUnidad.DriveLetter & vbCrLf
lcInfo = lcInfo & "Tipo: " & lcTipoUnidad & vbCrLf
lcInfo = lcInfo & "Volumen: " & ldrvUnidad.VolumeName & vbCrLf
lcInfo = lcInfo & "Tamaño: " & _
   FormatNumber(ldrvUnidad.TotalSize, 0) & vbCrLf
lcInfo = lcInfo & "Espacio disponible: " & _
   FormatNumber(ldrvUnidad.AvailableSpace, 0) & vbCrLf
lcInfo = lcInfo & "Sistema de ficheros: " & ldrvUnidad.FileSystem & vbCrLf
MsgBox lcInfo, , "Información del objeto Drive"

End Sub
```

Código fuente 443

Folder

Proporciona información sobre una carpeta que reside en una unidad.

Propiedades

- Attributes. Devuelve los atributos establecidos en una carpeta o fichero. La tabla 96 muestra las constantes disponibles para los atributos.

Constante	Valor	Descripción
Normal	0	Archivo normal. No se establecen atributos.
ReadOnly	1	Archivo de sólo lectura. El atributo es de lectura o escritura.
Hidden	2	Archivo oculto. El atributo es de lectura o escritura.
System	4	Archivo del sistema. El atributo es de lectura o escritura.
Volume	8	Etiqueta del volumen de la unidad de disco. El atributo es de sólo lectura.
Directory	16	Carpeta o directorio. El atributo es de sólo lectura.
Archive	32	El archivo cambió desde la última copia de seguridad. El atributo es de lectura o escritura.

Alias	64	Vínculo o método abreviado. El atributo es de sólo lectura.
Compressed	128	Archivo comprimido. El atributo es de sólo lectura.

Tabla 96. Constantes para la propiedad Attributes en carpetas y ficheros.

- DateCreated. Devuelve la fecha y hora en que se creó la carpeta o fichero.
- DateLastAccesed. Devuelve la fecha y hora en que se efectuó el último acceso a la carpeta o fichero.
- Drive. Devuelve la letra de la unidad a la que pertenece la carpeta o fichero.
- Files. Colección de objetos File que pertenecen a la carpeta actual.
- Name. Devuelve o establece el nombre de la carpeta o fichero.
- Size. Devuelve el tamaño de la carpeta (suma de los ficheros y subcarpetas contenidos en la carpeta utilizada) o fichero.
- SubFolders. Colección de objetos Folder contenidos en la carpeta actual.
- ShortName. Nombre de la carpeta o fichero en el formato 8+3 caracteres.
- Type. Cadena con una descripción de la carpeta o fichero.

Métodos

- Copy(). Copia una carpeta o fichero a una ubicación diferente dentro de la misma o de otra unidad.

  Sintaxis.

  ```
  Copy(cDestino [, bSobrescribir])
  ```

  - cDestino. Cadena con la nueva ubicación.
  - bSobrescribir.Opcional. Valor lógico que será True (predeterminado), para sobrescribir la carpeta en el caso de que exista en cDestino, y False cuando no se sobrescriba.

- Delete(). Borra una carpeta o fichero.

  Sintaxis.

  ```
  Delete([bForzar])
  ```

  - bForzar. Opcional. Valor lógico que al ser True borrará las carpetas o ficheros con el atributo de sólo lectura. Si es False (predeterminado) no se borrará este tipo de carpetas o ficheros.

- Move(). Cambia la ubicación de una carpeta o fichero.

  Sintaxis.

  ```
  Move(cDestino)
  ```

  - cDestino. Cadena con la nueva ubicación para la carpeta o fichero.

  Debemos tener la precaución de poner el caracter indicador de carpeta en el destino para evitar errores, tal como muestra la siguientes línea, en las que un objeto Folder se mueve de carpeta.

  ```
  oFolder.Move "c:\correo\"
  ```

Las diferentes opciones del menú Folder, en la aplicación de ejemplo, muestran información sobre el nombre de una carpeta introducida por el usuario, efectuando también diversas acciones. El código de tales opciones se muestra en el código fuente 444.

```
Private Sub mnuFoMostrar_Click()
Dim lcNombreCarpeta As String
Dim lcInfo As String
Dim lfsoFichero As FileSystemObject
Dim lfolCarpeta As Folder
Dim lfolSubCarpeta As Folder
Dim lfilFichero As File
' solicitar datos al usuario
lcNombreCarpeta = InputBox("Nombre de la carpeta", _
   "Mostrar propiedades de una carpeta")
' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto carpeta
Set lfolCarpeta = lfsoFichero.GetFolder(lcNombreCarpeta)
' mostrar propiedades de la carpeta
lcInfo = "Nombre de carpeta: " & lfolCarpeta.Name & vbCrLf
lcInfo = lcInfo & "Nombre en formato 8+3: " & lfolCarpeta.ShortName & vbCrLf
lcInfo = lcInfo & "Tipo: " & lfolCarpeta.Type & vbCrLf
lcInfo = lcInfo & "Tamaño: " & FormatNumber(lfolCarpeta.Size, 0) & vbCrLf
lcInfo = lcInfo & "Creada el " & _
   FormatDateTime(lfolCarpeta.DateCreated, vbShortDate) & vbCrLf
lcInfo = lcInfo & "Usada por última vez el " & _
   FormatDateTime(lfolCarpeta.DateLastAccessed, vbShortDate) & vbCrLf
MsgBox lcInfo, , "Propiedades de la carpeta"
' recorrer la colección Files de la
' carpeta mostrando información sobre
' cada objeto fichero
If lfolCarpeta.Files.Count > 0 Then
   For Each lfilFichero In lfolCarpeta.Files
      lcInfo = "Nombre: " & lfilFichero.Name & vbCrLf
      lcInfo = lcInfo & "Tamaño: " & FormatNumber(lfilFichero.Size, 0) & vbCrLf
      lcInfo = lcInfo & "Ruta: " & lfilFichero.Path & vbCrLf
      lcInfo = lcInfo & "Nombre corto: " & lfilFichero.ShortName & vbCrLf

      MsgBox lcInfo, , "Fichero de la carpeta " & lfolCarpeta.Name
   Next
End If
' recorrer la colección SubFolders de la carpeta
' mostrando el nombre de cada SubCarpeta
If lfolCarpeta.SubFolders.Count > 0 Then
   For Each lfolSubCarpeta In lfolCarpeta.SubFolders
      MsgBox "SubCarpeta: " & lfolSubCarpeta.Name
   Next
```

```
End If
End Sub
'*********************************************
Private Sub mnuFoCopiar_Click()
Dim lcNombreCarpeta As String
Dim lcCopiarEnCarpeta As String
Dim lfsoFichero As FileSystemObject
Dim lfolCarpeta As Folder
' obtener datos del usuario
lcNombreCarpeta = InputBox("Nombre de la carpeta origen", _
    "Copiar carpeta")
lcCopiarEnCarpeta = InputBox("Nombre de la carpeta destino", _
   "Copiar carpeta")


' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto carpeta
Set lfolCarpeta = lfsoFichero.GetFolder(lcNombreCarpeta)


' copiar el contenido de la carpeta origen
' en la carpeta destino
lfolCarpeta.Copy lcCopiarEnCarpeta
End Sub
'*********************************************
Private Sub mnuFoBorrar_Click()
Dim lcNombreCarpeta As String
Dim lfsoFichero As FileSystemObject
Dim lfolCarpeta As Folder
' obtener datos del usuario
lcNombreCarpeta = InputBox("Nombre de la carpeta a borrar", _
   "Borrar carpeta")
' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto carpeta
Set lfolCarpeta = lfsoFichero.GetFolder(lcNombreCarpeta)


' borrar la carpeta
lfolCarpeta.Delete True
End Sub
'*********************************************
Private Sub mnuFoMover_Click()
Dim lcNombreCarpeta As String
Dim lcMoverA As String
Dim lfsoFichero As FileSystemObject
Dim lfolCarpeta As Folder
' obtener datos del usuario
lcNombreCarpeta = InputBox("Nombre de la carpeta", _
   "Mover carpeta")
lcMoverA = InputBox("Nueva ubicación para la carpeta", "Mover carpeta")


' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto carpeta
Set lfolCarpeta = lfsoFichero.GetFolder(lcNombreCarpeta)


' mover contenido de carpeta
lfolCarpeta.Move lcMoverA
End Sub
```

Código Fuente 444

File

Realiza las operaciones con archivos relacionadas con la administración de los mismos, proporcionando también información acerca de sus características. No crea ni modifica el contenido de ficheros, ya que para eso disponemos del objeto TextStream que veremos más adelante. Consulte el lector las propiedades y métodos del anterior objeto Folder, ya que son también aplicables a los objetos File.

En el código fuente 445 se muestran las líneas de código de las opciones contenidas en el menú File del ejemplo, que realizan diferentes operaciones con este tipo de objetos.

```
Private Sub mnuFiMostrar_Click()
Dim lfsoFichero As FileSystemObject
Dim lfilFichero As File
Dim lcNombreFichero As String
Dim lcInfo As String
' obtener datos del usuario
lcNombreFichero = InputBox("Nombre del fichero", _
   "Introducir el nombre del fichero")
' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto fichero
Set lfilFichero = lfsoFichero.GetFile(lcNombreFichero)
' mostrar propiedades del objeto fichero
lcInfo = "Nombre: " & lfilFichero.Name & vbCrLf
lcInfo = lcInfo & "Tipo: " & lfilFichero.Type & vbCrLf
lcInfo = lcInfo & "Creado: " & _
   FormatDateTime(lfilFichero.DateCreated, vbShortDate) & vbCrLf
lcInfo = lcInfo & "Tamaño: " & FormatNumber(lfilFichero.Size, 0) & vbCrLf
lcInfo = lcInfo & "Ubicación: " & lfilFichero.ParentFolder & vbCrLf
lcInfo = lcInfo & "Ruta: " & lfilFichero.Path
MsgBox lcInfo, , "Información del objeto File"
End Sub
'*******************************************
Private Sub mnuFiCopiar_Click()
Dim lcFicheroOrigen As String
Dim lcFicheroDestino As String
Dim lfsoFichero As FileSystemObject
Dim lfilFichero As File
' obtener datos del usuario
lcFicheroOrigen = InputBox("Fichero origen", _
   "Copiar fichero")
lcFicheroDestino = InputBox("Fichero destino", _
   "Copiar fichero")

' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto File
Set lfilFichero = lfsoFichero.GetFile(lcFicheroOrigen)

' copiar fichero en carpeta destino
lfilFichero.Copy lcFicheroDestino
End Sub
'*******************************************
Private Sub mnuFiBorrar_Click()
Dim lcNombreFichero As String
Dim lfsoFichero As FileSystemObject
Dim lfilFichero As File
' obtener datos del usuario
lcNombreFichero = InputBox("Nombre del fichero", "Borrar fichero")
' crear objeto FSO
```

```
Set lfsoFichero = New FileSystemObject
' obtener objeto File
Set lfilFichero = lfsoFichero.GetFile(lcNombreFichero)
' borrarlo
lfilFichero.Delete
End Sub
'*******************************************
Private Sub mnuFiMover_Click()
Dim lcFicheroOrigen As String
Dim lcFicheroDestino As String
Dim lfsoFichero As FileSystemObject
Dim lfilFichero As File
' obtener datos del usuario
lcFicheroOrigen = InputBox("Nombre del fichero a mover", _
   "Mover fichero")

lcFicheroDestino = InputBox("Destino del fichero a mover", _
   "Mover fichero")

' crear objeto FSO
Set lfsoFichero = New FileSystemObject
' obtener objeto fichero
Set lfilFichero = lfsoFichero.GetFile(lcFicheroOrigen)

' mover el fichero especificado
lfilFichero.Move lcFicheroDestino
End Sub
```

Código fuente 445

FileSystemObject

Objeto principal del modelo. Facilita la creación y manipulación de ficheros y carpetas; igualmente proporciona objetos File y Folder para obtener información de sus propiedades.

Propiedades

- Drives. Colección de objetos Drive conectados al equipo.

Métodos

- BuildPath(). Crea una ruta para archivos.

  Sintaxis.

  ```
  BuildPath(cRuta, cNombre)
  ```

  - cRuta. Ruta de archivos existente.
  - cNombre. Nombre que combinado con cRuta, creará la nueva ruta.

- GetTempName(). Devuelve una cadena con un nombre generado aleatoriamente, que puede utilizarse en conjunción con el método CreateTextFile() para realizar operaciones con ficheros temporales.

  ```
  lcNombre = loFSO.GetTempName    ' radC8BOF.tmp
  ```

- GetAbsolutePathName(). Devuelve una cadena con una ruta absoluta, creada basándose en una ruta pasada como parámetro

  Sintaxis.

  ```
  GetAbsolutePathName(cRuta)
  ```

  - cRuta. Cadena con una ruta de ficheros.

- DriveExists(). Devuelve un valor lógico que indica si la unidad pasada como parámetro existe (True) o no (False).

  Sintaxis.

  ```
  DriveExists(cUnidad)
  ```

  - cUnidad. Cadena de caracteres con la especificación de la unidad.

- GetDrive(). Retorna un objeto Drive si coincide con la especificación de unidad pasada como parámetro.

  Sintaxis.

  ```
  GetDrive(cUnidad)
  ```

  - cUnidad. Cadena de caracteres con la especificación de la unidad.

- GetDriveName(). Devuelve una cadena con el nombre de la unidad en función de la ruta pasada como parámetro.

  Sintaxis.

  ```
  GetDriveName(cRuta)
  ```

  - cRuta. Cadena con la ruta de la que se va a devolver el nombre de la unidad.

- CopyFolder(). Copia una carpeta en una nueva ruta.

  Sintaxis.

  ```
  CopyFolder(cOrigen, cDestino [, bSobrescribir])
  ```

  - cOrigen. Cadena con la carpeta que se va a copiar, puede incluir caracteres comodín para copiar varias carpetas que estén contenidas en el origen.

  - cDestino. Cadena con la carpeta en la que se va a realizar la copia.

  - bSobrescribir. Opcional. Valor lógico que indica si las carpetas que ya existan en el destino se van a sobrescribir (True) o no (False).

- CreateFolder(). Crea una nueva carpeta

  Sintaxis.

```
CreateFolder(cCarpeta)
```

- cCarpeta. Cadena con el nombre de la carpeta a crear.

- DeleteFolder(). Borra una carpeta.

  Sintaxis.

```
DeleteFolder(cCarpeta [, bForzar])
```

  - cCarpeta. Cadena con el nombre de la carpeta a borrar.

  - bForzar. Opcional. Valor lógico para indicar que se borrarán las carpetas que tengan el atributo de lectura (True) o no (False).

- FolderExists(). Devuelve un valor lógico True en el caso de que exista la carpeta pasada como parámetro, y False si no existe.

  Sintaxis.

```
FolderExists(cCarpeta)
```

  - cCarpeta. Cadena con el nombre de la carpeta que vamos a comprobar

- GetFolder(). Devuelve un objeto Folder correspondiente al nombre de la carpeta pasada como parámetro. Para mayor información sobre objetos Folder, consulte el lector el punto dedicado a ellos en este apartado.

  Sintaxis.

```
GetFolder(cCarpeta)
```

  - cCarpeta. Cadena con el nombre de la carpeta a recuperar.

- GetParentFolderName(). Devuelve el nombre de la carpeta de nivel superior de la ruta pasada como parámetro

  Sintaxis.

```
GetParentFolderName(cRuta)
```

  - cRuta. Cadena con la ruta de la que vamos a averiguar la carpeta de nivel superior.

- GetSpecialFolder(). Devuelve una carpeta especial del sistema, según una de las constantes pasada como parámetro, que podemos ver en la tabla 97.

Constante	**Valor**	**Descripción**
WindowsFolder	0	La carpeta Windows contiene archivos instalados por el sistema operativo Windows.

SystemFolder	1	La carpeta Sistema contiene bibliotecas, fuentes y unidades de dispositivo.
TemporaryFolder	2	La carpeta Temp se utiliza para almacenar archivos temporales. Su ruta se encuentra en la variable del entorno TMP.

Tabla 97. Constantes para obtener una carpeta especial.

Sintaxis.

```
GetSpecialFolder(kCarpeta)
```

- kCarpeta. Constante con el tipo de carpeta a recuperar.

- MoveFolder(). Mueve una o varias carpetas a una nueva ubicación.

  Sintaxis.

  ```
  MoveFolder(cOrigen, cDestino)
  ```

  - cOrigen. Cadena con la carpeta que se va a mover, puede incluir caracteres comodín para mover varias carpetas que estén contenidas en el origen.

  - cDestino. Cadena con la carpeta en la que se va a depositar la carpeta origen.

- CopyFile(). Copia uno o más ficheros en una nueva ubicación.

  Sintaxis.

  ```
  CopyFile(cOrigen, cDestino [, bSobrescribir])
  ```

  - cOrigen. Cadena con el fichero que se va a copiar, es posible emplear caracteres comodín.

  - cDestino. Cadena con la ruta en la que se va a realizar la copia.

  - bSobrescribir. Opcional. Valor lógico que indica si los ficheros que ya existan en el destino se van a sobrescribir (True) o no (False).

- CreateTextFile(). Crea un nuevo fichero.

  Sintaxis.

  ```
  CreateTextFile(cFichero [, bSobrescribir [, bUnicode ]])
  ```

  - cFichero. Cadena con el nombre del fichero a crear.

  - bSobrescribir. Opcional. Valor lógico que indica si los ficheros existentes se pueden sobrescribir (True) o no (False).

  - bUnicode. Opcional. Valor lógico que indica si el fichero se creará de tipo Unicode (True) o ASCII (False).

- DeleteFile(). Borra un fichero.

  Sintaxis.

  ```
  DeleteFile(cFichero [, bForzar])
  ```

  - cFichero. Nombre del fichero a borrar, pudiendo emplear caracteres comodín.
  - bForzar. Opcional. Valor lógico para indicar que se borrarán los ficheros que tengan el atributo de lectura (True) o no (False).

- FileExists(). Devuelve un valor lógico True en el caso de que exista el archivo pasado como parámetro, y False si no existe.

  Sintaxis.

  ```
  File(cArchivo)
  ```

  - cArchivo. Cadena con el nombre del archivo a comprobar.

- GetFile(). Devuelve un objeto File correspondiente a la ruta pasada como parámetro.

  Para mayor información sobre objetos File, consulte el lector el punto dedicado a ellos en este apartado.

  Sintaxis.

  ```
  GetFile(cRuta)
  ```

  - cRuta. Cadena con la ruta y nombre del fichero a recuperar.

- MoveFile(). Mueve uno o varios archivos a una nueva ubicación.

  Sintaxis.

  ```
  oveFile(cOrigen, cDestino)
  ```

  - cOrigen. Cadena con la ruta de los archivos a mover, puede incluir caracteres comodín para mover varios archivos.
  - cDestino. Cadena con la ruta en la que se van a depositar los archivos.

- OpenTextFile(). Abre el fichero pasado como parámetro, devolviendo un objeto TextStream para realizar operaciones de lectura y escritura en dicho fichero.

  Sintaxis.

  ```
  OpenTextFile(cFichero [, kModoAcceso [, bCrear [,
  kFormato]]])
  ```

  - cFichero. Cadena con el nombre del fichero a abrir.
  - kModoAcceso. Opcional. Una de las constantes que aparecen en la tabla 98 para el modo de acceso al fichero.

Constante	Valor	Descripción
ForReading	1	Abrir un archivo sólo para lectura. No puede escribir en este archivo.
ForAppending	8	Abrir un archivo y escribir al final del archivo.
ForWriting	2	Abrir un archivo para escritura.

Tabla 98. Constantes para el modo de acceso al fichero.

- bCrear. Opcional. Valor lógico que indica si es posible crear el fichero en el caso de que no exista (True) o no es posible crear el fichero (False).

- kFormato. Opcional. Constante que indica el formato para el fichero (tabla 99).

Constante	Valor	Descripción
TristateUseDefault	–2	Abrir el archivo utilizando el valor predeterminado del sistema.
TristateTrue	–1	Abrir el archivo como Unicode.
TristateFalse	0	Abrir el archivo como ASCII.

Tabla 99. Constantes para el formato del fichero.

Las opciones del menú FSO en el formulario mdiInicio, muestran algunos ejemplos del uso de este tipo de objeto en la manipulación de carpetas y archivos, tal y como podemos ver en el código fuente 446, que contiene el código de dichas opciones.

```
Private Sub mnuFSCarpetas_Click()
Dim loFSO As FileSystemObject
Dim lfolEspecial As Folder
Dim lfilFichero As File
Set loFSO = New FileSystemObject
' crear nuevas carpetas
If Not loFSO.FolderExists("c:\pruebas") Then
   loFSO.CreateFolder "c:\pruebas"
   loFSO.CreateFolder "c:\pruebas\datos"
End If
' copiar el contenido de una carpeta en otra
loFSO.CopyFolder "c:\mis documentos", "c:\pruebas\"
' nombre de la carpeta de nivel superior
MsgBox "Carpeta superior de la ruta 'c:\pruebas\datos' --> " & _
   loFSO.GetParentFolderName("c:\pruebas\datos")
' eliminar carpeta
loFSO.DeleteFolder "c:\pruebas\datos"
' cambiar de posición una carpeta
loFSO.MoveFolder "c:\pruebas", "c:\mis documentos\"
' recuperar una carpeta del sistema y mostrar
' el nombre de los ficheros que contiene
```

```
Set lfolEspecial = loFSO.GetSpecialFolder(WindowsFolder)
For Each lfilFichero In lfolEspecial.Files
   MsgBox lfilFichero.Name, , "Ficheros de la carpeta Windows"
Next
End Sub

*****************************************
Private Sub mnuFSFicheros_Click()
Dim loFSO As FileSystemObject
Dim lfolEspecial As Folder
Dim lfilFichero As File
Dim lcNombreFichero As String
Dim lcInfo As String
Set loFSO = New FileSystemObject
' obtener datos del usuario
lcNombreFichero = InputBox("Introducir el nombre de un fichero", _
   "Pruebas con ficheros")
' comprobar si existe el fichero
If Not loFSO.FileExists(lcNombreFichero) Then
   MsgBox "El fichero " & lcNombreFichero & " no existe"
   Exit Sub
End If
' tomar un objeto fichero del nombre
' introducido por el usuario y mostrar
' algunas propiedades
Set lfilFichero = loFSO.GetFile(lcNombreFichero)
lcInfo = "Nombre: " & lfilFichero.Name & vbCrLf
lcInfo = lcInfo & "Tamaño: " & FormatNumber(lfilFichero.Size, 0) & vbCrLf
lcInfo = lcInfo & "Tipo: " & lfilFichero.Type & vbCrLf
MsgBox lcInfo, , "Información del fichero " & lcNombreFichero
' copiar el fichero en una carpeta
loFSO.CopyFile lcNombreFichero, "c:\mis documentos\", True
' mover el fichero a una carpeta
loFSO.MoveFile lcNombreFichero, "c:\windows\temp\"
' borrar el fichero de su ubicación original
loFSO.DeleteFile lcNombreFichero
' crear fichero vacío
loFSO.CreateTextFile "registra.txt"
End Sub
```

Código Fuente 446

TextStream

Este objeto proporciona un conjunto de propiedades y métodos para realizar operaciones de lectura y escritura sobre un archivo secuencial.

Propiedades

- AtEndOfLine. Valor lógico que al ser verdadero indicará que el puntero del archivo está situado al final de una línea.
- AtEndOfStream. Valor lógico que al ser verdadero indicará que el puntero del archivo está situado al final del archivo.
- Column. Devuelve el número de columna en la que está situado el puntero del archivo.
- Line. Devuelve el número de línea en la que está situado el puntero del archivo.

Métodos

- Close(). Cierra el fichero.
- Read(). Lee un número de caracteres del fichero.

  Sintaxis

  ```
  Read(nCaracteres)
  ```

  - nCaracteres. Número de caracteres a leer.
- ReadAll(). Lee el contenido total del fichero y lo devuelve en una cadena. Este método no es muy recomendable ya que emplea muchos recursos, siendo más conveniente la lectura por líneas o por grupos de caracteres.
- ReadLine(). Lee una línea del fichero y la devuelve en una cadena.
- Skip(). Avanza un número de caracteres en la lectura de un fichero, sin devolverlos.

  Sintaxis

  ```
  Skip(nCaracteres)
  ```

  - nCaracteres. Número de caracteres a avanzar.
- SkipLine(). Avanza una línea en la lectura de un fichero, sin devolver su contenido.
- Write(). Escribe una cadena en un fichero.

  Sintaxis.

  ```
  Write(cCadena)
  ```

  - cCadena. Cadena de caracteres a escribir.
- WriteBlankLines(). Escribe una o más líneas en blanco en un fichero.

  Sintaxis.

  ```
  WriteBlankLines(nLineas)
  ```

  - nLineas. Número de líneas en blanco a insertar.
- WriteLine(). Escribe una línea en un fichero que puede incluir una cadena.

  Sintaxis.

  ```
  WriteLine([cCadena])
  ```

  - cCadena. Opcional. Cadena a escribir en la línea.

En el código fuente 447 se muestra el fuente que corresponde a las opciones del menú TextStream, encargadas de efectuar diversas tareas con ficheros mediante objetos de esta clase.

```
Private Sub mnuTeEscribir_Click()
Dim loFSO As FileSystemObject
Dim ltxsTexto As TextStream
Dim lcNombreFichero As String
Set loFSO = New FileSystemObject
' obtener datos del usuario
lcNombreFichero = InputBox("Introducir el nombre de un fichero", _
   "Crear y escribir en un fichero")
' comprobar si existe el fichero y
' crearlo si no existe
If Not loFSO.FileExists(lcNombreFichero) Then
   loFSO.CreateTextFile lcNombreFichero, True
End If
' abrir el fichero para escritura,
' añadiendo al final
Set ltxsTexto = loFSO.OpenTextFile(lcNombreFichero, ForAppending)
' escribir nuevas cadenas en el fichero
ltxsTexto.Write "cadena de caracteres para el fichero"
ltxsTexto.WriteBlankLines 2
ltxsTexto.WriteLine "nuevos datos"
ltxsTexto.Write "terminar aquí"
' cerrar el fichero
ltxsTexto.Close
End Sub
*****************************************
Private Sub mnuTeLeer_Click()
Dim loFSO As FileSystemObject
Dim ltxsTexto As TextStream
Dim lcNombreFichero As String
Dim lcCadena As String
Set loFSO = New FileSystemObject
' obtener datos del usuario
lcNombreFichero = InputBox("Introducir el nombre de un fichero", _
   "Abrir y leer de un fichero")
' comprobar si existe el fichero
If Not loFSO.FileExists(lcNombreFichero) Then
   MsgBox "El fichero " & lcNombreFichero & " no existe"
   Exit Sub
End If
' abrir el fichero para lectura
Set ltxsTexto = loFSO.OpenTextFile(lcNombreFichero, ForReading)
' leer varias cadenas y líneas
' agregando a una variable e
' incluyendo saltos en la lectura
lcCadena = ltxsTexto.Read(5)
ltxsTexto.Skip 30
lcCadena = lcCadena & ltxsTexto.Read(20)
ltxsTexto.Skip 10
lcCadena = lcCadena & ltxsTexto.ReadLine
ltxsTexto.SkipLine
lcCadena = lcCadena & ltxsTexto.ReadLine
MsgBox lcCadena, , "Resultado de la lectura de cadenas del fichero"
' cerrar el fichero
ltxsTexto.Close
End Sub

*****************************************
Private Sub mnuTeCompleto_Click()
Dim loFSO As FileSystemObject
Dim ltxsTexto As TextStream
Dim lcNombreFichero As String
Dim lcCadena As String
Set loFSO = New FileSystemObject
' obtener datos del usuario
lcNombreFichero = InputBox("Introducir el nombre de un fichero", _
```

```
   "Abrir y leer un fichero por completo")
' comprobar que existe el fichero
If Not loFSO.FileExists(lcNombreFichero) Then
   MsgBox "El fichero " & lcNombreFichero & " no existe"
   Exit Sub
End If
' abrir el fichero para lectura
Set ltxsTexto = loFSO.OpenTextFile(lcNombreFichero, ForReading)
' leer el fichero por líneas y mostrarlas
' al usuario
Do While Not ltxsTexto.AtEndOfStream
   MsgBox "Línea del fichero: " & ltxsTexto.Line & vbCrLf & _
      "Contenido: " & ltxsTexto.ReadLine
Loop
' cerrar el fichero
ltxsTexto.Close
End Sub
```

Código fuente 447

14

Acceso al API de Windows desde Visual Basic

Qué hacer cuando con Visual Basic no se puede

Visual Basic es una potente herramienta que provee al programador de un amplio número de recursos durante el desarrollo de su trabajo habitual. Para la mayoría de las aplicaciones que necesitemos desarrollar, existirá la instrucción, procedimiento o función adecuada que solvente el problema al que nos enfrentemos; pero de igual forma, se puede presentar un programa que necesite realizar una tarea especial, para la que Visual Basic no nos proporcione la solución.

En tales situaciones no debemos darnos por vencidos, es el momento de recurrir al API de Windows. En primer lugar, debemos examinar una serie de conceptos básicos que nos ayuden a conocer qué es el API, para después pasar a utilizarlo.

¿Qué es un API?

Un API o Interfaz de Programación de Aplicaciones, es un conjunto de funciones pertenecientes a una aplicación o sistema operativo (en nuestro caso es esto último), que pueden ser empleadas por otra aplicación mediante un protocolo de llamada y paso de parámetros. Aunque la aplicación esté desarrollada en un lenguaje distinto al que se desarrolló el API, el protocolo antes mencionado, permitirá a la aplicación hacer uso y aprovechar el trabajo desempeñado por tales funciones.

El API de Windows está formado por un enorme número de funciones, pero esto no debe preocupar al lector, no es necesario conocerlas todas, lo verdaderamente importante es disponer de la

documentación de referencia adecuada sobre las funciones, la labor que desempeñan y saber cuál aplicar en cada momento. Debemos pensar que el API es una parte integrante de Windows y al mismo tiempo, una extensión de nuestro lenguaje de programación; si algo no encontramos en VB, con toda seguridad habrá una función en el API que haga el trabajo que necesitamos. Se puede decir que el propio Windows se convierte en parte de nuestro lenguaje de programación

Para Windows existen diferentes API, en función de la versión del sistema operativo. De esta forma tenemos el API Win16, que es el primero, destinado a las versiones de 16 bits de Windows; Win32, que es el API original de Windows NT; Windows95/Windows32c, subconjunto de funciones de Win32 accesibles desde Windows95, en este API no se incluyen aquellas funciones específicas para resolver tareas de Windows NT.

DLL, el soporte físico del API

El conjunto de funciones que componen el API, está contenido en varios ficheros de tipo DLL, Librería de Enlace Dinámico. El lector puede encontrar una breve descripción de estas librerías en el tema introducción de este curso.

Este tipo de librería contiene compilado el código de funciones. Durante el proceso de creación de una DLL, el programador debe indicar qué funciones de las contenidas en esta librería podrán ser accesibles desde aplicaciones externas; a estas funciones se las denomina funciones exportadas, y son las que componen el API.

Recordemos que una función incluida en una DLL, puede ser utilizada por más de un programa que tenga establecida una referencia adecuada a la DLL y la función. La ventaja de esta técnica estriba en que el código de dicha función reside en un único lugar, que es la DLL, en vez de necesitar una copia de ese código para cada programa que lo vaya a usar, como ocurría con el formato tradicional de librería (LIB).

En Windows, el API está compuesto por una serie de DLL's, las principales son:

- KERNEL32. Contiene las funciones principales o de núcleo, tales como las de gestión de memoria, recursos, etc.
- USER32. Aquí disponemos de las funciones para la administración del sistema, manejo de menús, cursores, mensajes, etc.
- GDI32. Funciones para el interfaz de dispositivo gráfico. Aquí podemos encontrar todo lo relacionado con el dibujo, fuentes.

Como complemento a las anteriores, tenemos estas otras librerías de apoyo, que aportan características adicionales no incluidas en las básicas.

- COMDLG32. Acceso a cuadros de diálogo estándar.
- LZ32. Compresión de ficheros.
- VERSION. Control de versiones.

El uso de DLL's proporciona una gran flexibilidad y una magnífica capacidad de expansión al sistema operativo. Tomemos como ejemplo, el hecho de que en las primeras versiones de Windows, los elementos multimedia no disponían de la popularidad actual. Cuando todos estos elementos: CD-ROM, tarjetas de sonido, etc., se hicieron necesarios en el sistema, no fue necesario rescribir las

librerías existentes para adaptar estas nuevas capacidades. Se incorporaron las funciones multimedia en una nueva DLL que pasó a formar parte del API. Entre estas librerías que proporcionan características extendidas tenemos las siguientes:

- ODBC32. Las funciones de esta librería nos permiten trabajar con bases de datos mediante el estándar ODBC.
- COMCTL32. Conjunto de controles complementarios a los controles estándar de Windows.
- NETAPI32. Librería con funciones para redes.
- WINMM. Funciones multimedia.
- MAPI32. Funciones para el manejo de correo electrónico.

La instrucción Declare

Esta instrucción se utiliza en el código de un programa, para establecer una referencia hacia una función o procedimiento que reside en una DLL, de forma que al hacer la llamada a dicha función, no se produzca un error. A esta operación también se la conoce como importar una función.

Sintaxis para importar procedimientos.

```
[Public|Private] Declare Sub NombrePF Lib "NombreLib" [Alias
"NombreAlias"] [([Parámetros])]
```

Sintaxis para importar funciones.

```
[Public|Private] Declare Function NombrePF Lib "NombreLib" [Alias
"NombreAlias"] [([Parámetros])]
```

- Public. La función se declara con ámbito global, siendo visible por todos los módulos de la aplicación.
- Private. El ámbito de la función se limita sólo al módulo donde ha sido declarada.

  Si se omite en la declaración la palabra clave que especifica el ámbito, la función tendrá un ámbito global.
- Sub. Se utilizará al importar un procedimiento, sin devolución de valor al ser ejecutado.
- Function. Se utilizará al importar una función, que devuelve un valor al ser ejecutada.
- NombrePF. Nombre empleado en el programa para el procedimiento declarado. Es importante tener en cuenta que en este caso, se realiza una distinción entre mayúsculas y minúsculas.
- Lib. Indica que el procedimiento declarado se encuentra en una DLL o recurso.
- mbreLib. Nombre de la librería que contiene el procedimiento a importar. Se debe poner entre comillas y es necesario especificar la extensión, excepto en las siguientes DLL's: Kernel32, User32, GDI32.

- Alias. Sirve para especificar el nombre real que tiene el procedimiento dentro de la DLL, si por alguna causa no es posible especificar dicho nombre real en NombrePF. Las situaciones en las cuales podemos necesitar el uso de esta cláusula son varias:

    - Cuando el nombre del procedimiento en la DLL sea igual que una palabra clave de VB.

    - Cuando el nombre del procedimiento en la DLL sea igual que el nombre de un método, función, etc., del propio lenguaje Visual Basic. Si por ejemplo, importamos de la DLL User32, la función SetFocus(), nos encontraremos ante un problema, ya que este nombre es utilizado como método por muchos objetos de VB. Utilizando Alias, podemos resolver el problema de una forma parecida a la del código fuente 448.

```
Declare Function PonerFoco Lib "user32" Alias "SetFocus" (ByVal
hwnd As Long) As Long
```

Código fuente 448

Esta situación también se produce, si en el nombre del procedimiento de la DLL hay algún carácter no permitido por VB como nombre de procedimiento, por ejemplo el guión bajo "_".

- Otro uso de Alias tan importante como los anteriores, consiste en crear declaraciones de tipo estricto, para evitar los posibles problemas que acarrea el uso de As Any; este punto será tratado más adelante.

- NombreAlias. Contiene, entre comillas, el nombre real que tiene el procedimiento en la librería que estamos referenciando. Debido a que los procedimientos dentro de una DLL se clasifican mediante un número, es posible indicar dicho número de procedimiento precedido del símbolo "#", en lugar del nombre del procedimiento. Esta técnica proporciona una ligera mejora en el rendimiento de la aplicación, aunque no es muy recomendable debido a que dichos números pueden variar entre distintos sistemas operativos.

Vamos a efectuar una primera aproximación al manejo del API de Windows, mediante el proyecto Encuentra, incluido en los ejemplos de este tema. Consiste en una pequeña aplicación que busca si la calculadora de Windows está en ejecución y la trae a primer plano, por encima del resto de ventanas que haya abiertas. Para ello empleará dos funciones del API contenidas en el módulo Declaapi del proyecto, que vemos en el código fuente 449 con los comentarios explicativos para las funciones declaradas.

```
' declaraciones de funciones del API de Windows
' FindWindow()
' Buscar una ventana.
' Parámetros:
'  - lpClassName. Cadena con el nombre de la clase
'    de la ventana o cero para tomar cualquier clase.
'
'  - lpWindowName. Cadena con el título de la ventana a
'    a buscar o cero para tomar cualquier título.
'
' Valor de retorno. Número Long que identifica al manipulador
'  de la ventana o cero en el caso de no encontrarla
Declare Function FindWindow Lib "user32" _
```

```
   Alias "FindWindowA" (ByVal lpClassName As String, _
   ByVal lpWindowName As String) As Long
' BringWindowToTop()
' Muestra una ventana en primer plano.
' Parámetros:
'  - hwnd. Número Long que representa al manipulador
'    de la ventana a mostrar.
'
' Valor de retorno. Número Long cuyo valor es distinto de
'  cero en el caso de que tenga éxito, o cero
'  si se produce un error.
Declare Function BringWindowToTop Lib "user32" _
   (ByVal hwnd As Long) As Long
```

Código fuente 449

Esta aplicación dispone de un formulario llamado frmPruAPI, mostrado en la figura 482

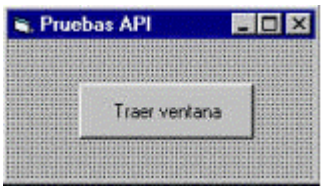


Figura 482. rmPruAPI del proyecto Encuentra.

Una vez cargado el formulario desde el procedimiento Main(), al pulsar su botón Traer ventana, se buscará la aplicación Calculadora, que de estar en ejecución, será mostrada en primer plano. Es importante tener en cuenta que si la calculadora está minimizada, no se visualizará; en sucesivos ejemplos, veremos como solventar ese pequeño inconveniente.

El código del botón que realiza este trabajo se muestra en el código fuente 450.

```
Private Sub cmdTraer_Click()
Dim llHWND As Long
Dim llResultado As Long
llHWND = FindWindow(vbNullString, "Calculadora")
llResultado = BringWindowToTop(llHWND)
End Sub
```

Código fuente 450

El Visor de API

Revisando el ejemplo del anterior apartado, el lector seguramente se estará preguntando como hemos conseguido averiguar los nombres de las funciones que debíamos utilizar, los parámetros de dichas funciones y su valor de retorno.

El API de Windows es muy extenso y compuesto por miles de funciones; algunas herramientas de programación, como Visual C++, u otro tipo de utilidades, suministran documentación diversa, generalmente en ficheros de ayuda, con información sobre las funciones del API, mensajes, valores de retorno, etc., que suponen una inestimable ayuda a la hora de realizar desarrollos orientados a emplear el API.

Estos ficheros de ayuda, suelen estar organizados de forma que puedan ser consultados en función de un tema determinado: ventanas, ficheros, gráficos, etc., facilitando al programador la tarea de buscar esa función que necesita para resolver un determinado problema. Una vez localizada la función, podremos obtener una descripción de la tarea que desempeña, los parámetros que necesita y el valor que devuelve. De esta forma, podemos crear la declaración en nuestro programa.

A este respecto, Visual Basic nos provee del llamado Visor de API, que facilita nuestra labor a la hora de seleccionar una función del API e incorporarla en el código de la aplicación. Es una utilidad localizada en el grupo de programas correspondiente a las herramientas de Visual Studio o en la opción Complementos+Visor de API del menú de VB, consistente en una aplicación que carga un fichero de texto en el que se encuentran las definiciones de funciones, constantes y estructuras del API Win32. Mediante las opciones de este visor, podemos buscar por ejemplo una función, y una vez localizada, copiarla al Portapapeles, pegándola después en el código de nuestra aplicación. El texto pegado, contendrá la declaración completa de la función, incluyendo nombre, parámetros, alias si es necesario y valor de retorno.

Al iniciar este visor, se mostrará la ventana de la figura 483.

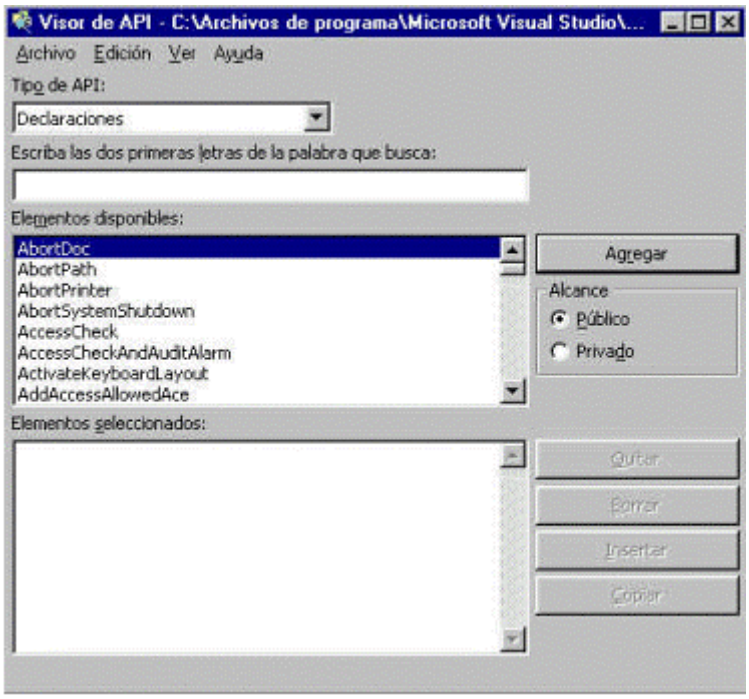


Figura 483. to API en ejecución.

A continuación seleccionaremos del menú, la opción Archivo+Cargar archivo de texto, para cargar el fichero de texto, que contiene la información sobre el API, para que pueda ser mostrada por esta utilidad. Mediante un cuadro de diálogo, seleccionaremos el fichero Win32api.TXT, cuyo contenido será mostrado en la lista Elementos disponibles de esta aplicación.

Para agilizar el tiempo de carga, es posible convertir la información del fichero de texto a una base de datos de Access, para en las siguientes sesiones de esta utilidad, mostrar dicha base de datos en vez del fichero de texto. La opción Archivo+Convertir texto a una base de datos, realiza tal conversión.

Una vez cargado en el visor el fichero de información correspondiente, seleccionaremos la información a buscar en el ComboBox Tipo de API, con lo que aparecerán todos los elementos disponibles en la lista del mismo nombre.

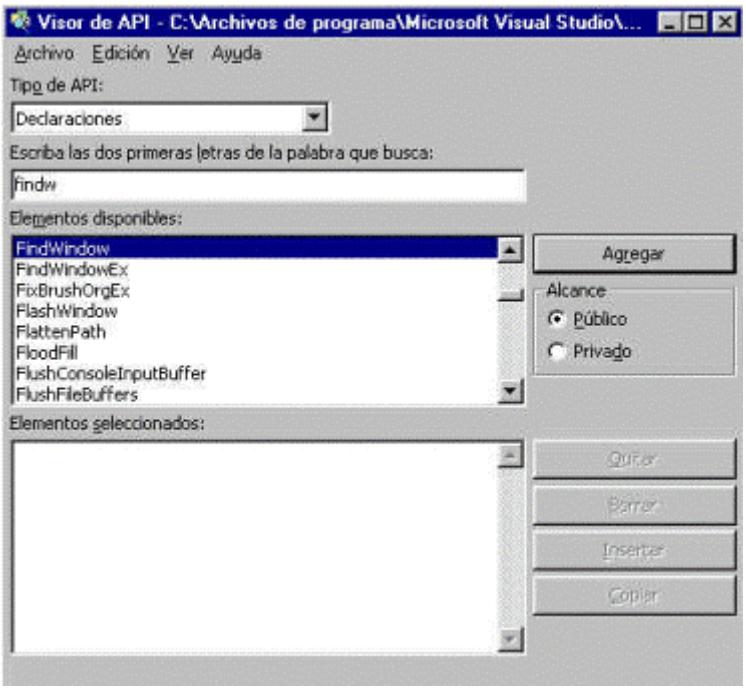


Figura 484. una función en el visor API.

Situémonos en el ejemplo anterior buscando la función FindWindow(). En primer lugar seleccionaremos en Tipo de API el valor Declaraciones, para mostrar las funciones existentes; después teclearemos las primeras letras de la función en el TextBox Escriba las primeras letras..., de esta forma, se efectuará una búsqueda dinámica en la lista Elementos disponibles. Una vez localizada la función, estableceremos su ámbito en el Frame Alcance y pulsando Agregar, se copiará la declaración en la lista Elementos seleccionados (figura 485).

Pulsando a continuación el botón Insertar, se situará la declaración seleccionada en el módulo de código que haya abierto en el proyecto actual de VB. Si pulsamos por el contrario Copiar, la declaración seleccionada pasará al Portapapeles, y desde allí podremos pegarla una vez estemos situados en la ventana de código de VB.

El botón Quitar del visor de texto, sirve para eliminar una declaración de las existentes en el cuadro Elementos seleccionados.

A pesar de que el Visor de API supone un gran apoyo para el programador, es vital disponer de un medio de información, ya sean ficheros de ayuda, manuales u otro sistema de consulta, para poder emplear como documentación de referencia, ya que sin tener la descripción del trabajo que realiza una determinada función, de poco nos servirá disponer de este visor API.

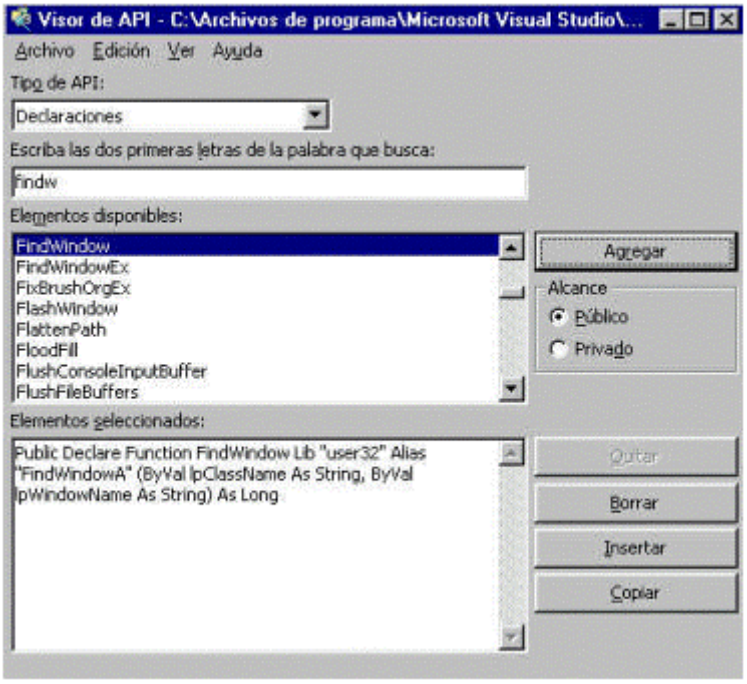


Figura 485. una declaración localizada.

Declaración de funciones que manejan cadenas

Si el lector observa la declaración de la función FindWindow(), utilizada en los anteriores apartados, se percatará de que el nombre real, utilizado con Alias, que tiene la función dentro de la DLL es FindWindowA(). Esta es una situación que se repetirá con todas las funciones que manipulen cadenas de caracteres, la causa reside en los juegos de caracteres que pueden manejar los diferentes sistemas operativos.

Mientras Windows 95 trabaja con juegos de caracteres de byte simple y doble, Windows NT utiliza un juego de caracteres ampliado, llamado Unicode; debido a que las API's para Win32 soportan los tres juegos, era necesario un medio de poder realizar llamadas a estas funciones desde cada sistema operativo sin que surgieran conflictos con el juego de caracteres a emplear.

La solución al problema, ha sido crear dos funciones, que realicen la misma tarea según el juego de caracteres que se deba emplear. De esta manera, tomando como ejemplo la función FindWindow(), cuando trabajemos para Windows 95, deberemos declarar esta función bajo el alias FindWindowA, donde "A" hará referencia al juego de caracteres ANSI. Si el sistema operativo es Windows NT, el alias a emplear será FindWindowW, donde "W" hará referencia al juego de caracteres ampliado (wide) Unicode.

Declaraciones de tipo estricto. Evitar el uso de "As Any"

Cuando el lector haya practicado con el visor API proporcionado por VB, es muy probable que en alguna declaración, el tipo de alguno de los parámetros esté especificado como As Any. Esto quiere decir que al realizar la llamada a la función del API, desde VB no se comprobará el tipo de ese parámetro, aumentando el riesgo de provocar un error del sistema en el caso de que el valor del parámetro no sea admitido por la función del API.

La causa de que el visor API genere la declaración de esta manera, reside en que existen parámetros en algunas funciones del API que admiten más de un tipo de datos. Tomemos como ejemplo la función AppendMenu(), que añade una opción de menú a una ventana, y cuya declaración es proporcionada por el visor API de la siguiente manera:

```
Declare Function AppendMenu Lib "user32" Alias "AppendMenuA"
(ByVal hMenu As Long, ByVal wFlags As Long, ByVal wIDNewItem As
Long, ByVal lpNewItem As Any) As Long
```

Los parámetros para esta función son los siguientes:

Número Long, corresponde al manipulador de menú al que se le añade una opción.

- wFlags. Número Long, constante o combinación de constantes, que corresponde al estado de la opción de menú.
- wIDNewItem. Número Long, sirve como identificador único para la opción de menú.
- lpNewItem. El valor a asignar a este parámetro depende de la/s constante/s empleada/s en el parámetro wFlags.
    - MF_STRING. El parámetro será una cadena de caracteres.
    - MF_BITMAP. El parámetro será un manipulador de un bitmap.
    - MF_OWNERDRAW. Un valor para las estructuras DRAWITEMSTRUCT y MEASUREITEMSTRUCT del menú.

Para resolver el problema planteado por el parámetro lpNewItem, podemos emplear declaraciones de tipo estricto, que se basan en la siguiente técnica: crear declaraciones distintas para la función, con el parámetro que puede tomar valores diferentes, adaptado a un único tipo de valor para cada declaración. De esta forma, se comprueba desde VB el tipo de parámetro que se va a enviar a la función del API, y en el caso de que no sea correcto, el error se provocará desde VB, con lo cual, obtendremos mayor información sobre lo que ha pasado. Empleando As Any, el error sería más difícil de seguir, ya que el comportamiento incorrecto se produciría al ejecutar la función del API.

En el código fuente 451 se presentan dos declaraciones para la función AppendMenu(), una es para añadir una cadena en la opción y la otra para añadir un bitmap.

```
Declare Function AppendMenuString Lib "user32" Alias "AppendMenuA" (ByVal hMenu As
Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, ByVal lpNewItem As String) As
Long

    Declare Function AppendMenuBitmap Lib "user32" Alias "AppendMenuA" (ByVal hMenu
As Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, ByVal lpNewItem As Long)
As Long
```

Código fuente 451

En el proyecto de ejemplo MenuAPI, se comprueba si la calculadora de Windows está en ejecución, y en caso afirmativo, se utiliza la primera de las declaraciones de tipo seguro anteriores, para agregar una nueva opción en el primero de sus menús desplegables.

Las declaraciones para el API en este proyecto, se encuentran en el módulo Declaapi. En el código fuente 452, podemos ver tales declaraciones; sólo se incluyen las nuevas declaraciones, ya que también se emplean las del ejemplo del apartado anterior.

```
' GetMenu()
' Obtiene el manipulador de menú de una ventana
' Parámetros:
'  - hwnd. Número Long que representa al manipulador
'    de la ventana de la que obtenemos el manipulador
'    de menú.
'
' Valor de retorno. Número Long cuyo valor representa al
'  manipulador de la ventana o cero si se produce
'  un error.
Declare Function GetMenu Lib "user32" (ByVal hwnd As Long) As Long
' GetSubMenu()
' Obtiene el manipulador de un menú desplegable
' de una ventana
' Parámetros:
'  - hMenu. Número Long que representa al manipulador
'    del menú de nivel superior de la ventana
'
'  - nPos. Número Long que representa al número de menú
'    desplegable del que queremos obtener su manipulador
'
' Valor de retorno. Número Long cuyo valor representa al
'  manipulador del menú desplegable o cero si se produce
'  produce un error.
Declare Function GetSubMenu Lib "user32" _
   (ByVal hMenu As Long, ByVal nPos As Long) As Long
' AppendMenu()
' Añade una opción a un menú.
' Se ha modificado el nombre por AppendMenuString()
' para obtener una declaración de función con parámetros
' de tipo estricto.
' Parámetros:
'  - hMenu. Número Long que representa al manipulador
'    del menú de nivel superior de la ventana.
'
'  - wFlags. Número Long, corresponde a una constante
'    que establece el estado de la opción.
'
'  - wIDNewItem. Número Long que actúa como identificador
'    de la opción.
'  - lpNewItem. Cadena con la opción de menú.
'
' Valor de retorno. Número Long con un valor distinto de
' cero si la función tiene éxito, o cero si se produce
' produce un error.
Declare Function AppendMenuString Lib "user32" _
   Alias "AppendMenuA" (ByVal hMenu As Long, _
   ByVal wFlags As Long, ByVal wIDNewItem As Long, _
   ByVal lpNewItem As String) As Long
' declaraciones de constantes del API de Windows
Public Const MF_STRING = &H0&
```

Código fuente 452

El formulario frmPruMenu de este programa, se encarga de realizar las acciones de manipulación del menú.

Figura 486. Formulario frmPruMenu de la aplicación MenuAPI.

Pulsando el botón Lanzar calculadora, de la figura 486, se ejecuta dicho programa mediante la función Shell() de VB. El código del evento Click() de este botón es el que muestra el código fuente 453.

```
Private Sub cmdLanza_Click()
' ejecutar la calculadora de Windows
Shell "calc.exe", vbNormalFocus
End Sub
```

Código fuente 453

El botón Modificar menú, tomará el manipulador de la calculadora y agregará una opción a uno de sus menús, como podemos ver en el código fuente 454.

```
Private Sub cmdModifMenu_Click()
Dim llHWND As Long
Dim llHMENU As Long
Dim llHSUBMENU As Long
Dim llResultado As Long
' buscar calculadora y mostrarla
llHWND = FindWindow(vbNullString, "Calculadora")
BringWindowToTop llHWND
' tomar manipuladores de menú de la calculadora
llHMENU = GetMenu(llHWND)
llHSUBMENU = GetSubMenu(llHMENU, 0)
' añadir nueva opción al primer menú de la calculadora
llResultado = AppendMenuString(llHSUBMENU, MF_STRING, 3000, _
"OPCIÓN CREADA CON EL API DE WINDOWS")
End Sub
```

Código fuente 454

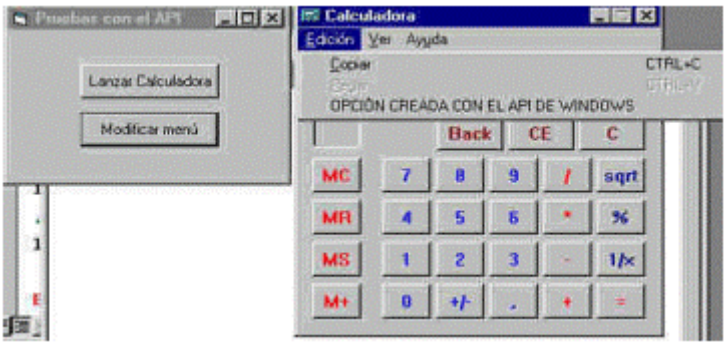


Figura 487. Opción de menú agregada a la aplicación Calculadora mediante el API de Windows.

Una vez ejecutado este código y si todo ha tenido éxito, al desplegar el primer menú de la calculadora, se mostrará una imagen como la figura 487

En conclusión, siempre que sea posible, hemos de evitar el uso de As Any, y emplear declaraciones de tipo estricto, para eludir los posibles problemas de los parámetros en las funciones del API, que pueden tomar diferentes valores.

Crear declaraciones seguras mediante una función "envolvente"

Continuando con el aspecto de la seguridad en las llamadas a las funciones del API, para evitar la posibilidad de errores graves del sistema, este apartado presenta una técnica, que empleada junto a las declaraciones de tipo estricto, supone un paso adelante, en el objetivo de conseguir declaraciones seguras para las funciones del API.

El empleo de una función envolvente, consiste en crear una función que sea la que realiza la llamada a la función del API que necesitamos. Para ilustrar esta técnica, haremos uso de nuevo, de la función del API AppendMenu(), empleada en el apartado anterior.

Partiendo del estado de declaración de tipo estricto: AppendMenuString(), en el cual dejamos a esta función, aún podemos encontrarnos con desagradables sorpresas si por ejemplo realizamos una llamada como la que aparece en el código fuente 455.

```
AppendMenuString(llHSUBMENU, MF_BITMAP, 3000, "Nueva opción de menú")
```

Código fuente 455

Al intentar crear una nueva opción de menú, pero utilizar la constante MF_BITMAP en el segundo parámetro, no obtendremos los resultados deseados. Las consecuencias ante este tipo de problemas pueden ser variadas: desde que la función simplemente no realice su cometido hasta provocar un error de protección en el sistema. Si se da el primer caso, el problema es menor, ya que lo normal es recoger el valor devuelto por la función, que nos indica si todo se ha desarrollado con normalidad; en el caso de los errores de protección, el asunto es más grave.

Para evitar este tipo de situaciones, y siempre que nos encontremos en el caso de añadir nuevas opciones de tipo cadena a un menú, crearemos una función estándar de VB (la función envolvente), desde la que realizaremos la llamada a la función del API. En esta nueva función, sólo definiremos los parámetros utilizados en la llamada al API que tengan posibilidad de cambio cada vez que se use la función. Veamos en el código fuente 456, como quedaría esta función envolvente.

```
Public Function AgregarOpcionMenu(ByVal vlhMenu As Long, ByVal vlIDOpcion As Long,
ByVal vcNombre As String) As Long
AgregarOpcionMenu = AppendMenuString(vlhMenu, MF_STRING, vlIDOpcion, vcNombre)
End Function
```

Código fuente 456

La función AgregarOpcionMenu(), será la que a partir de ahora utilicemos en nuestro código, y la que realmente haga la llamada a la declaración del API AppendMenuString(). La ventaja de todo ello, reside en que ahorramos la posibilidad de confusión en la constante utilizada en el segundo parámetro de AppendMenuString(); como siempre vamos a utilizar esa misma constante al crear nuevas opciones de tipo cadena, dejamos a la función envolvente tal trabajo.

La técnica de funciones envolventes, si bien no resuelve totalmente el problema de los errores en el manejo de las funciones del API, nos puede evitar parte de los problemas, simplificando las llamadas a tales funciones

Manipulación de mensajes

Los mensajes en Windows, componen uno de los pilares de la arquitectura del sistema. Cuando una aplicación realiza una operación, lo notifica al sistema mediante un mensaje. De igual manera, si el sistema necesita actuar sobre una aplicación, le envía mensajes. Este punto, quizá, sea el que más sorprende a los programadores provenientes de sistemas como MS-DOS, que comienzan a desarrollar en Windows.

En un entorno DOS, el programa hace llamadas al sistema operativo a través de interrupciones del sistema. Sin embargo, en Windows esta comunicación se establece en ambas direcciones: el programa hace llamadas al sistema, y el sistema a su vez, llama al programa cuando necesita comunicarle determinada situación. El mecanismo empleado para comunicar al sistema operativo con las aplicaciones, se basa en un sistema de mensajes.

Cuando el sistema necesita informar de un determinado suceso a una aplicación, le envía un mensaje. La aplicación a su vez, debe disponer de una función especial denominada procedimiento de ventana, que se encarga de recoger el mensaje enviado por el sistema. Sobre este procedimiento trataremos en el apartado siguiente. Nuestro objetivo actual será la comunicación en el sentido inverso: el programa mandando mensajes al sistema.

Cómo son los mensajes

Un mensaje es en realidad una constante que contiene un número para poder identificarlo de forma unívoca. Windows dispone de un juego de mensajes para el trabajo con las funciones del API. Adicionalmente, una aplicación puede crear su propio grupo de mensajes (obviamente, con números distintos de los empleados por los mensajes estándar de Windows), para notificar mensajes particulares.

El código fuente 457, contiene la declaración en un programa VB de varias constantes empleadas como mensajes para las llamadas al API.

```
' declaraciones de constantes del API de Windows
Public Const MF_STRING = &H0&
Public Const WM_CUT = &H300
Public Const WM_PASTE = &H302
Public Const EM_SETSEL = &HB1
Public Const WM_SYSCOMMAND = &H112
Public Const SC_CLOSE = &HF060
Public Const SC_SCREENSAVE = &HF140
```

Código fuente 457

Enviar mensajes

Las principales funciones para el envío de mensajes son las siguientes:

- SendMessage(). Envía un mensaje a una ventana, y no devuelve el control a la aplicación hasta que dicha ventana no ha terminado de procesar el mensaje. Esta función devuelve un valor que depende del mensaje enviado.

  Aunque decimos que esta función envía un mensaje a una ventana, hemos de matizar que quien primeramente procesa tal mensaje es el sistema, el cual se encargará de direccionarlo a la ventana correspondiente. Por ese motivo, al comienzo del apartado nos referimos a que el envío de mensajes se realiza al sistema y no a la ventana de destino.

  En el proyecto MsgAPI incluido como ejemplo para este apartado, se incluye una declaración para esta función, sobre la que se ha efectuado una modificación para adaptar su último parámetro a tipo estricto, ya que el visor API proporciona esta declaración sin comprobación de tipos para dicho parámetro. El código fuente 458 muestra la declaración junto a una breve descripción de la función.

```
' SendMessage()
' Envía un mensaje al sistema.
' Se ha modificado el nombre por SendMessageNum()
' para obtener una declaración de función con parámetros
' de tipo estricto.
' Parámetros:
'  - hwnd. Número Long que representa al manipulador
'    de la ventana sobre la que vamos a efectuar
'    una operación.
'  - wMsg. Número Long, corresponde a una constante
'    con el mensaje a enviar a la ventana.
'  - wParam. Número Long, su valor depende del mensaje
'    enviado.
'  - lParam. Número Long, su valor depende del mensaje
'    enviado.
'
' Valor de retorno. Número Long, su valor depende
' del mensaje enviado.
Declare Function SendMessageNum Lib "user32" _
   Alias "SendMessageA" (ByVal hwnd As Long, _
   ByVal wMsg As Long, ByVal wParam As Long, _
   ByVal lParam As Long) As Long
```

Código fuente 458

- PostMessage(). Sitúa un mensaje en la cola de mensajes de una ventana, devolviendo el control inmediatamente a la aplicación, sin esperar a que la ventana de destino termine de procesar el mensaje. Esta función devuelve un valor que depende del mensaje enviado. Los parámetros y valor de retorno son iguales que en SendMessage().

Vamos a ilustrar el envío de mensajes, mediante el uso de la función SendMessage(). Emplearemos para ello la aplicación de ejemplo MsgAPI. Este proyecto dispone del formulario frmMensajes que aparece en la figura 488.

Observe el lector en primer lugar, los dos TextBox y el botón Traspasar texto. Cuando el usuario introduzca una cadena en el cuadro de texto txtOrigen, y pulse el botón antes mencionado, se seleccionará el contenido de txtOrigen, y se traspasará al Portapapeles mediante la operación Cortar.

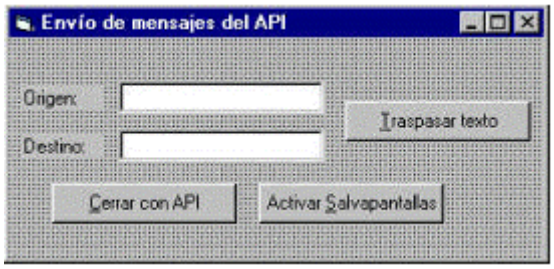


Figura 488. Formulario frmMensajes de la aplicación MsgAPI.

Finalmente se pegará el texto del Portapapeles en el TextBox txtDestino, todo ello enviando mensajes a la ventana mediante el API. El código fuente 459 pertenece al evento de pulsación del botón Traspasar texto.

```
Private Sub cmdTraspasar_Click()
Dim llResultado As Long
' seleccionar todo el texto del TextBox txtOrigen
llResultado = SendMessageNum(Me.txtOrigen.hwnd, EM_SETSEL, 0, -1)
' cortar el texto seleccionado en el TextBox txtOrigen
' pasándolo al Portapapeles
llResultado = SendMessageNum(Me.txtOrigen.hwnd, WM_CUT, 0, 0)
' pegar en el TextBox txtDestino el contenido
' del Portapapeles
llResultado = SendMessageNum(Me.txtDestino.hwnd, WM_PASTE, 0, 0)
End Sub
```

Código fuente 459

Pasemos ahora al botón Cerrar con API, el trabajo de este control, consiste en cerrar la ventana invocando a la orden de cierre que existe en el menú de control de cualquier ventana en Windows, mediante un mensaje a dicha ventana. En el código fuente 460 se muestra el código de este control.

```
Private Sub cmdCerrar_Click()
Dim llResultado As Long
' finalizar la aplicación mediante la opción
' Cerrar del menú de control del formulario
llResultado = SendMessageNum(Me.hwnd, WM_SYSCOMMAND, SC_CLOSE, 0)
End Sub
```

Código fuente 460

Finalmente, enviaremos un mensaje a este formulario, también de sistema como el anterior, aunque este no resulte tan intuitivo. Consiste en activar el salvapantallas del equipo, y sucederá al pulsar el botón Activar Salvapantallas, cuyas líneas de código vemos en el código fuente 461.

```
Private Sub cmdSalvaPant_Click()
Dim llResultado As Long
' activar el SalvaPantallas
llResultado = SendMessageNum(Me.hwnd, WM_SYSCOMMAND, SC_SCREENSAVE, 0)
End Sub
```

Código fuente 461

Subclasificación. Procesar los mensajes que envía el sistema

Como apuntábamos brevemente en el apartado anterior, toda ventana dispone de una función especial, encargada de recoger los mensajes que el sistema operativo le envía, para tratarlos si procede. En la terminología Windows, esta función se denomina procedimiento de ventana, y se basa en una estructura de tipo Select Case...End Select, en la que se toma cada mensaje que Windows manda a la ventana. Si para un mensaje se debe realizar alguna operación, habrá definido un Case con el código a ejecutar. Si no existe ninguna operación específica a realizar con el mensaje, la estructura Select Case, deberá tener definido un Case Else para todos aquellos mensajes que no se vayan a tratar en este procedimiento; en este Case Else se hará una llamada al denominado procedimiento de ventana por defecto, que se encargará de procesar todos los mensajes de los que no se ocupe el procedimiento de ventana normal.

Visual Basic es una herramienta poderosa, que proporciona al programador, todo lo necesario para realizar su trabajo. Existen sin embargo, aspectos de bajo nivel, a los cuales VB no da soporte. La causa de ello, es que si así lo hiciera, se incrementaría notablemente la complejidad de desarrollo, y precisamente, uno de los puntos fuertes de VB es la facilidad de programación. Para complicarnos profundizando a bajo nivel, ya tenemos Visual C++.

Si a pesar de todo, nos encontramos ante una aplicación que precisa un tratamiento especial, en la que se necesiten procesar los mensajes que envía Windows a un formulario o control, VB en la actual versión, nos proporciona una técnica para capturar tales mensajes, y poder personalizar el comportamiento de un determinado formulario. Dicha técnica se denomina subclasificación.

La subclasificación, consiste en la creación de un procedimiento de ventana personalizado, para un formulario o control, compuesto básicamente, y como mencionamos antes, de una estructura Select Case, en la que trataremos los mensajes que nos interese procesar de entre todos los que Windows envía a nuestro formulario. Aquellos mensajes que no necesitemos manipular, los enviaremos al procedimiento de ventana por defecto, del que previamente habremos obtenido su manipulador.

Subclasificación para una ventana

Veamos a continuación, una sencilla muestra de subclasificación, en la aplicación de ejemplo SubclasWin. Esta aplicación, dispone de un formulario llamado frmSubclasificar, para el que vamos a crear un procedimiento de ventana personalizado, de manera que al detectar la pulsación del botón derecho del ratón sobre el formulario, mostremos las coordenadas de dicha pulsación en el título de la ventana.

El trabajo a realizar en primer lugar, es declarar las funciones del API que vayamos a utilizar. Este código será incluido en el módulo Declaapi, que vemos en el código fuente 462, y que dispone de una breve descripción para cada declaración.

```
' declaración de funciones del API de Windows
' SetWindowLong()
' Sitúa un valor de tipo Long, en un desplazamiento
' determinado de la zona de memoria extra de una
' ventana. En función del byte de desplazamiento en
' donde se sitúe el valor, el empleo de esta función
' será diverso.
' En este caso reemplazaremos el procedimiento de
' ventana por defecto, por otro personalizado.
```

```
' Posteriormente, utilizaremos esta misma función
' para restaurar el procedimiento de ventana original
' Parámetros:
'  - hwnd. Número Long que representa al manipulador
'    de la ventana a la que vamos a asignar un nuevo
'    procedimiento de ventana.
'  - nIndex. Número Long para establecer la acción a
'    realizar. En este caso usaremos la constante GWL_WNDPROC
'  - dwNewLong. Número Long, que consiste en un puntero
'    a la función que actuará como nuevo procedimiento
'    de ventana.
'
' Valor de retorno. Número Long, manejador del
' procedimiento por defecto que tenía la ventana
' antes de ser reemplazado con esta función.
Declare Function SetWindowLong Lib "user32" _
   Alias "SetWindowLongA" (ByVal hwnd As Long, _
   ByVal nIndex As Long, ByVal dwNewLong As Long) As Long
' CallWindowProc()
' Realiza una llamada a un procedimiento de ventana
' Parámetros:
'  - lpPrevWndFunc. Número Long que contiene el manipulador
'    original del procedimiento de ventana.
'  - hwnd. Número Long que contiene el manipulador
'    de la ventana a la que se llama.
'  - Msg. Número Long con el mensaje a enviar.
'  - wParam. Número Long, primer parámetro del mensaje.
'  - lParam. Número Long, segundo parámetro del mensaje.
' Valor de retorno. Número Long, contiene el resultado
' de la llamada al procedimiento de ventana y depende
' del mensaje enviado a dicha ventana.
Declare Function CallWindowProc Lib "user32" _
   Alias "CallWindowProcA" (ByVal lpPrevWndFunc As Long, _
   ByVal hwnd As Long, ByVal Msg As Long, _
   ByVal wParam As Long, ByVal lParam As Long) As Long
' GetCursorPos()
' Obtener las coordenadas del ratón.
' Parámetros:
'  - lpPoint. Estructura de tipo POINTAPI, en la cual
'    se van a depositar las coordenadas del ratón.
'
' Valor de retorno. Número Long, con el resultado
' de la llamada a la función.
Declare Function GetCursorPos Lib "user32" _
   (lpPoint As POINTAPI) As Long

' SetWindowText()
' Establecer el título de una ventana.
' Parámetros:
'  - hwnd. Número Long que contiene el manipulador
'    de la ventana a la que se cambia el título.
'  - lpString. Cadena con el título para la ventana.
'
' Valor de retorno. Número Long, con el resultado
' de la llamada a la función.
Declare Function SetWindowText Lib "user32" _
   Alias "SetWindowTextA" (ByVal hwnd As Long, _
   ByVal lpString As String) As Long
'////////////////////////////////////////////
' declaración de constantes del API de Windows
Public Const GWL_WNDPROC = (-4)
Public Const WM_RBUTTONDOWN = &H204
'////////////////////////////////////////////
' declaración de estructuras del API de Windows
' estructura o tipo definido por el usuario para
' utilizar con las funciones que manejan coordenadas
' del cursor
```

```
Type POINTAPI
   x As Long
   y As Long
End Type
```

Código fuente 462

Seguidamente, escribiremos el procedimiento de ventana personalizado ProcVentPropio(), para el formulario frmSubclasificar, con el que reemplazaremos el procedimiento de ventana original que tuviera este formulario. En el código fuente 463 vemos el fuente del módulo código ProcVentana, que contiene dicho procedimiento.

```
' variable global que contendrá un manejador
' que apunta al procedimiento de ventana original
' del formulario que se subclasifica
Public glhWndProcVentOriginal As Long
' ----------------------------------------
Public Function ProcVentPropio(ByVal hwnd As Long, _
   ByVal idMsg As Long, ByVal wParam As Long, _
   ByVal lParam As Long) As Long

' esta función contiene un nuevo procedimiento de ventana para
' el formulario de esta aplicación
Dim llResultado As Long
Dim Cursor As POINTAPI
' estructura de proceso de mensajes
Select Case idMsg
Case WM_RBUTTONDOWN
   ' si se pulsa sobre la ventana el botón
   ' derecho del ratón...

   ' obtener las coordenadas del ratón
   llResultado = GetCursorPos(Cursor)

   ' mostrarlas en el título de la ventana
   llResultado = SetWindowText(hwnd, _
      "Posición cursor X: " & Cursor.x & " Y:" & Cursor.y)
   ' valor de retorno de la función
   ProcVentPropio = 1
Case Else
   ' para el resto de mensajes,
   ' llamar al procedimiento de ventana original
   ' con los mismos parámetros que para esta
   ' función, excepto el primero, que será
   ' el manejador del procedimiento de ventana original
   ProcVentPropio = CallWindowProc(glhWndProcVentOriginal, _
      hwnd, idMsg, wParam, lParam)
End Select
End Function
```

Código fuente 463

Este procedimiento, y todos los de su tipo, que se emplean para subclasificar una ventana, deben seguir el protocolo de paso de parámetros y devolución de valores establecido en Windows, que se muestra aquí. Se pueden utilizar los nombres que el programador estime oportuno, para el procedimiento y parámetros, siempre y cuando, el tipo, orden de los parámetros y valor de retorno sea el mostrado en el código fuente 463.

Como el lector habrá podido comprobar, disponemos de una estructura Select Case, que procesa los mensajes enviados por Windows. En este caso, sólo actuaremos cuando el mensaje indique que se ha

pulsado el botón derecho del ratón sobre el formulario (WM_RBUTTONDOWN), en cuyo caso, llamaremos a la función del API GetCursorPos(), a la que se pasará una variable de tipo estructura POINTAPI, que es un tipo definido por el usuario con dos números Long, sobre la que esta función situará los valores de las coordenadas.

Acto seguido, llamaremos a otra función del API: SetWindowText(), que sitúa un texto en el título de una ventana, y que usaremos para mostrar las coordenadas del ratón.

Para el resto de mensajes, se ejecutará el código contenido en Case Else, que se ocupa de llamar a la función CallWindowProc(). Esta función realiza una llamada al procedimiento de ventana que se pasa en forma de puntero en su primer parámetro, el resto de los parámetros son los mismos que para el procedimiento de ventana actual. El procedimiento llamado aquí, será el que tenía el formulario antes de ser subclasificado, y del cual hemos obtenido su dirección de entrada en forma de puntero, al activar la subclasificación.

Finalmente, pasemos al formulario frmSubclasificar, que podemos ver en la figura 489.

Figura 489. Formulario frmSubclasificar el proyecto SubclasWin.

El código de este formulario, con los comentarios explicativos oportunos, se puede ver en el código fuente 464.

```
Dim mcTitulo As String

' ------------------------------------

Private Sub cmdActivar_Click()
' guardar el título original del formulario
mcTitulo = Me.Caption
' Establecer el nuevo procedimiento de ventana para
' este formulario.
' El valor devuelto por SetWindowLong() es el manejador
' que apunta al procedimiento original de este formulario
' Para conseguir un puntero a la función que actuará como
' el nuevo procedimiento de ventana, emplearemos el
' operador AddressOf
glhWndProcVentOriginal = SetWindowLong(Me.hwnd, GWL_WNDPROC, _
   AddressOf ProcVentPropio)
Me.cmdActivar.Enabled = False
Me.cmdDesactivar.Enabled = True
End Sub

' ------------------------------------

Private Sub cmdDesactivar_Click()
' restablecer el procedimiento de ventana original
' para este formulario. De esta forma ya no se
```

```
' llamará al procedimiento de ventana personalizado
' ProcVentPropio()
SetWindowLong Me.hwnd, GWL_WNDPROC, glhWndProcVentOriginal
Me.cmdActivar.Enabled = True
Me.cmdDesactivar.Enabled = False
' recuperar el título original del formulario
Me.Caption = mcTitulo
End Sub
```

Código fuente 464

Mención especial en la rutina que activa la subclasificación, merece el uso del operador AddressOf; este operador, permite pasar a una función del API, la dirección de un procedimiento en forma de puntero.

En condiciones normales, cuando uno de los parámetros pasados a una función en VB es el nombre de otra función, la función que actúa como parámetro se ejecuta, devolviendo un valor, que es el que en realidad se envía como parámetro a la función llamada.

El uso de AddressOf, pasa como parámetro la dirección del procedimiento en lugar de su resultado. De esta forma, la función del API, utiliza la dirección para hacer una llamada al procedimiento; esta técnica se denomina devolución de llamada, y es la empleada por las funciones del API, para hacer llamadas a los procedimientos pasados como parámetro.

Ya sólo queda que el lector ejecute esta aplicación, y compruebe como se muestran las coordenadas del cursor después de pulsar el botón Activar Subclasificación.

Subclasificación para un control

Teniendo en cuenta que en Windows, todos los elementos de una ventana (controles), son tratados a su vez como ventanas, la subclasificación, como comentábamos al principio de este apartado, puede aplicarse también a un control, para el caso de que sólo sea necesario capturar los mensajes que Windows envía a determinado control.

Basándonos en parte del ejemplo anterior, utilizaremos la aplicación SubclasCtl, en la que se realizará una subclasificación para el TextBox txtPrueba, incluido en su formulario, de modo que alteraremos parte de su comportamiento original. Dicha alteración, se producirá una vez activada la subclasificación para el TextBox, y consistirá en que al pulsar el botón derecho del ratón sobre el control, en lugar de aparecer el menú contextual con las conocidas opciones de edición, mostraremos un menú diferente creado en tiempo de ejecución mediante llamadas al API.

También capturaremos el mensaje que se produce cuando se pulsa la tecla Enter y los caracteres tecleados en el control, avisando igualmente al usuario. En esta ocasión, en vez de suprimir el comportamiento normal del control, después de mostrar al usuario el mensaje de pulsación de la tecla, permitiremos que dicho comportamiento se lleve a cabo, llamando al procedimiento de ventana por defecto, que tiene este control; de esta manera, los caracteres tecleados se verán en el cuadro de texto.

Cuando decimos procedimiento de ventana, nos estamos refiriendo al procedimiento personalizado que se llama al subclasificar el control, y que permite la captura de mensajes enviados por Windows. El denominarlo “de ventana”, se debe al hecho de que Windows trata al control como una ventana más.

El código fuente 465, muestra los puntos más importantes a tener en cuenta para subclasificar el control. No se muestra el código al completo, sólo los aspectos principales, ya que es muy similar al del ejemplo anterior. Comencemos por el modulo Declaapi.

```
' Declaraciones para el API
' ------------------------
' CreatePopupMenu()
' Crea un menú desplegable vacío.
' Valor de retorno. Número Long, manipulador del menú
' creado o cero si existe error.
Declare Function CreatePopupMenu Lib "user32" () As Long
' AppendMenu()
' Añadir una opción a un menú.
' Se utilizará para esta función el nombre
' AppendMenuStr(), ya que se ha modificado su
' último parámetro para conseguir una declaración
' de tipo estricto.
' Establecer el título de una ventana.
' Parámetros:
'  - hMenu. Número Long que contiene el manipulador
'    de menú al que vamos a añadir una opción.
'  - wFlags. Número Long que puede ser una constante, y
'    especifica el tipo de opción
'  - wIDNewItem. Número Long que representa un identificador
'    para la opción.
'  - lpNewItem. Cadena con la descripción de la opción.
' Valor de retorno. Número Long, que será distinto de cero
' si la función tuvo éxito.
Declare Function AppendMenuStr Lib "user32" _
   Alias "AppendMenuA" _
   (ByVal hMenu As Long, _
   ByVal wFlags As Long, _
   ByVal wIDNewItem As Long, _
   ByVal lpNewItem As String) As Long
' TrackPopupMenu()
' Visualiza un menú desplegable en cualquier
' lugar del formulario.
' Se utilizará para esta función el nombre
' TrackPopupMenuNum(), ya que se ha modificado su
' último parámetro para conseguir una declaración
' de tipo estricto.
' Establecer el título de una ventana.
' Parámetros:
'  - hMenu. Número Long que contiene el manipulador
'    de menú a visualizar
'  - wFlags. Número Long que puede ser una constante, y
'    especifica el tipo de alineación del menú al ser
'    mostrado.
'  - x / y . Números Long que indican las coordenadas
'    en donde será mostrado el menú.
'  - nReserved. Número Long sin uso; poner cero.
'  - hwnd. Número Long con el manipulador de la ventana
'    o control que mostrará el menú.
'  - lprc. Número Long, que deberá ser cero.
' Valor de retorno. Número Long, que será distinto de cero
' si la función tuvo éxito.
Declare Function TrackPopupMenuNum Lib "user32" _
   Alias "TrackPopupMenu" _
   (ByVal hMenu As Long, _
   ByVal wFlags As Long, _
   ByVal x As Long, _
   ByVal y As Long, _
   ByVal nReserved As Long, _
   ByVal hwnd As Long, _
```

```
   ByVal lprc As Long) As Long
```

Código fuente 465

A continuación, pasaremos al módulo ProcVentana, que contiene el código fuente 466 del procedimiento de ventana empleado para el TextBox del formulario de la aplicación.

```
' ////////////////////////////////////////
' variable global que contendrá un manejador
' que apunta al procedimiento de ventana original
' del TextBox que se subclasifica
Public glhWndProcVentOriginal As Long
' variable global que contendrá el manipulador
' del menú desplegable que sustituirá al menú
' de contexto habitual del control TextBox
' del formulario
Public glhMenuContexto As Long
' identificadores para las opciones
' del nuevo menú de contexto del TextBox
Public Const IDM_BUSCAR = 3000
Public Const IDM_ORTOGRAFIA = 3001
Public Const IDM_SINONIMOS = 3002
Public Function ProcVentPropio(ByVal hwnd As Long, _
   ByVal idMsg As Long, ByVal wParam As Long, _
   ByVal lParam As Long) As Long

' esta función contiene un nuevo procedimiento de ventana para
' el TextBox del formulario de esta aplicación
Dim llResultado As Long
Dim curCursor As POINTAPI
' estructura de proceso de mensajes
Select Case idMsg
Case WM_COMMAND
   ' se ha producido una selección en una
   ' opción de menú
   Select Case wParam
   Case IDM_BUSCAR
      MsgBox "Buscar cadena"

   Case IDM_ORTOGRAFIA
      MsgBox "Revisión ortográfica"

   Case IDM_SINONIMOS
      MsgBox "Lista de sinónimos"

   End Select

   ProcVentPropio = 1
Case WM_CONTEXTMENU
   ' se ha efectuado la acción de abrir
   ' el menú contextual del TextBox
   GetCursorPos curCursor
   ' abrir el menú de contexto
   llResultado = TrackPopupMenuNum(glhMenuContexto, _
      TPM_LEFTALIGN, curCursor.x, curCursor.y, 0, hwnd, 0)
   ProcVentPropio = 1
Case WM_CHAR
   ' se ha pulsado una tecla
   If wParam = VK_RETURN Then
      MsgBox "Se pulsó Enter"
   Else
      MsgBox "Se pulsó otro caracter"
   End If
```

```
   ' llamando a CallWindowProc() después de procesar este
   ' mensaje, se introducirá en el TextBox, el valor de la
   ' tecla pulsada; puesto que la llamada a esta función, ejecuta
   ' el comportamiento por defecto programado en Windows para
   ' este control
   ProcVentPropio = CallWindowProc(glhWndProcVentOriginal, _
      hwnd, idMsg, wParam, lParam)
Case Else
   ' para el resto de mensajes,
   ' llamar al procedimiento de ventana original
   ' con los mismos parámetros que para esta
   ' función, excepto el primero, que será
   ' el manejador del procedimiento de ventana original
   ProcVentPropio = CallWindowProc(glhWndProcVentOriginal, _
      hwnd, idMsg, wParam, lParam)
End Select
End Function
```

Código fuente 466

Pruebe el lector a modificar en este fuente, en el mensaje WM_CHAR, que detecta los caracteres pulsados, la llamada a CallWindowProc(), por el valor 1, que se asigna como valor de retorno de la función. Comprobará que dichos caracteres no llegan a su destino: el TextBox.

Por último pasamos al formulario frmSubclasificar, lo vemos en la figura 490.

Figura 490. Formulario frmSubclasificar del proyecto SubclasCtl.

En lo que respecta al código, las rutinas que activan y desactivan la subclasificación no han variado, excepto en que el manejador de ventana que deben usar es el del control TextBox. Como novedad, nos encontramos la creación del menú desplegable en el evento de carga del formulario, y que vemos en el código fuente 467.

```
Private Sub Form_Load()
Dim llResultado As Integer
' durante la carga del formulario,
' se utilizarán llamadas al API, para
' crear un menú desplegable
glhMenuContexto = CreatePopupMenu()
llResultado = AppendMenuStr(glhMenuContexto, MF_STRING, IDM_BUSCAR, "Buscar")
llResultado = AppendMenuStr(glhMenuContexto, MF_STRING, IDM_ORTOGRAFIA,
"Ortografía")
llResultado = AppendMenuStr(glhMenuContexto, MF_STRING, IDM_SINONIMOS, "Sinónimos")
End Sub
```

Código fuente 467

Cuando el formulario esté bajo subclasificación y pulsemos el botón derecho del ratón sobre el TextBox, veremos algo parecido a la figura 491.

Cuando el usuario seleccione una opción de menú, se enviará un mensaje WM_COMMAND al procedimiento de ventana del formulario, en el que el parámetro wParam, contendrá el identificador de menú asignado al crear este menú en la carga del formulario; esta es la manera que tendremos de reconocer que opción ha sido seleccionada y actuar en consecuencia.

Figura 491. Aplicación SubclasCtl en ejecución mostrando un menú de contexto personalizado.

Al ser esta aplicación una prueba, y no realizarse un control exhaustivo, antes de finalizar la ejecución del programa, hemos de tener la precaución de desactivar la subclasificación, pulsando el botón existente a este efecto, ya que de lo contrario, se podría producir un error de protección.

Algunos ejemplos adicionales

En la aplicación de ejemplo APIVarios, el lector dispone de algunas muestras suplementarias de operaciones que pueden ser realizadas empleando el API de Windows.

Desde el formulario frmAPIVarios, comenzaremos ejecutando su opción de menú Archivo+Cascada, para mostrar como disponer en dicho orden las ventanas que tengamos abiertas en el escritorio. El fuente de esta opción es el código fuente 468.

```
Private Sub mnuArCascada_Click()
' situar en cascada las ventanas
' que se encuentran en el escritorio
Dim lnResultado As Integer
lnResultado = CascadeWindowsNum(GetDesktopWindow(), _
   MDITILE_SKIPDISABLED, 0, 0, 0)
End Sub
```

Código fuente 468

Podemos organizar de este modo, tanto las ventanas del escritorio como las de una aplicación MDI que tenga ventanas secundarias abiertas. En este caso al organizar el escritorio, utilizaremos la función del API GetDesktopWindow() para obtener su manipulador. Las funciones del API para esta opción están en el código fuente 469.

```
' GetDesktopWindow()
' Obtener el manipulador del escritorio de Windows.
'
' Valor de retorno. Número Long que identifica al manipulador
' del escritorio.
Declare Function GetDesktopWindow Lib "user32" () As Long
' CascadeWindows()
' Sitúa las ventanas en cascada.
' Se utilizará para esta función el nombre
```

```
' CascadeWindowsNum(), ya que se han modificado
' algunos parámetros para conseguir una declaración
' de tipo estricto.
' Parámetros:
'  - hwndParent. Número Long, manejador correspondiente a la
'    ventana padre cuyas hijas se dispondrán en cascada.
'  - wHow. Número Long, constante que indica el modo para las
'    ventanas MDI.
'  - lpRect. Número Long, cero para usar todo el área de pantalla
'    disponible.
'  - cKids. Número Long, cantidad de ventanas hijas indicadas
'    en lpkids.
'  - lpkids. Número Long, cero para incluir todas las ventanas
'    hijas.
'
' Valor de retorno. Número Integer, que contendrá la cantidad
' de ventanas que se han dispuesto en cascada.
Declare Function CascadeWindowsNum Lib "user32" _
   Alias "CascadeWindows" _
   (ByVal hwndParent As Long, _
   ByVal wHow As Long, _
   ByVal lpRect As Long, _
   ByVal cKids As Long, _
   ByVal lpkids As Long) As Integer
```

Código fuente 469

Para el siguiente ejemplo, introduciremos el título de una ventana o parte del mismo, en el TextBox txtBuscarWin, y pulsaremos el botón Buscar por manipulador, cuya labor será recorrer todas las ventanas abiertas en el sistema, obteniendo su manipulador; mediante este manipulador, recuperaremos la cadena que forma el título de la ventana; si al comparar con el valor introducido por el usuario existieran coincidencias, se mostrará la ventana encontrada. El código de este botón lo vemos en el código fuente 470.

```
Private Sub cmdBuscarManip_Click()
' buscar una ventana por el título
' utilizando los manipuladores de
' ventana
Dim llResultado As Long
Dim llhWnd As Long
Dim lcTitulo As String
' tomar el manipulador correspondiente
' al escritorio
llhWnd = GetDesktopWindow()
' tomar el manipulador correspondiente
' a la primera ventana hija del escritorio
llhWnd = GetWindow(llhWnd, GW_CHILD)
' en el siguiente bucle, tomar los manipuladores
' del resto de ventanas hija del escritorio
' hasta encontrar la que debemos mostrar
Do While llhWnd <> 0
   ' tomar el título de la ventana de la cuál
   ' acabamos de recuperar el manipulador
   lcTitulo = Space(70)
   llResultado = GetWindowText(llhWnd, lcTitulo, 70)

   ' comprobar si parte del título de la ventana
   ' coincide con la cadena introducida por el
   ' usuario
   If InStr(UCase(lcTitulo), UCase(Me.txtBuscarWin.Text)) > 0 Then
      ' si la ventana está iconizada, restaurarla antes
      If IsIconic(llhWnd) <> 0 Then
```

```
         llResultado = ShowWindow(llhWnd, SW_RESTORE)
      End If
      ' guardar el manipulador para el caso
      ' de que se le aplique parpadeo
      mlhWndParpadeo = llhWnd

      ' traer la ventana a primer plano
      BringWindowToTop llhWnd
      Exit Sub
   End If

   ' recuperar la siguiente ventana
   llhWnd = GetWindow(llhWnd, GW_HWNDNEXT)

Loop
End Sub
```

Código fuente 470

En el código fuente 470, se emplean varias funciones del API, cuyas declaraciones se muestran en el código fuente 471.

```
' GetWindow()
' Obtener el manipulador de una ventana respecto
' a una ventana padre.
' Parámetros:
'  - hwnd. Número Long con el manipulador de la ventana padre.
'
'  - wCmd. Número Long, constante que indica la relación entre
'    la ventana a obtener con respecto al manipulador hwnd.
'
' Valor de retorno. Número Long que identifica al manipulador
' de la ventana obtenida o cero en el caso de no encontrarla
Declare Function GetWindow Lib "user32" _
   (ByVal hwnd As Long, _
   ByVal wCmd As Long) As Long
' GetWindowText()
' Obtener el título de una ventana.
' Parámetros:
'  - hwnd. Número Long que contiene el manipulador
'    de la ventana de la que se obtiene el título.
'  - lpString. Cadena que actúa a modo de buffer
'    para almacenar el título de la ventana.
'  - cch. Número Long con la cantidad máxima de caracteres a copiar
'    en lpString.
'
' Valor de retorno. Número Long, con que indica el número
' de caracteres copiados o cero si la ventana no tiene
' título o se produjo algún problema
Declare Function GetWindowText Lib "user32" _
   Alias "GetWindowTextA" _
   (ByVal hwnd As Long, _
   ByVal lpString As String, _
   ByVal cch As Long) As Long
' ShowWindow()
' Visualiza una ventana en un determinado estado.
' Parámetros:
'  - hwnd. Número Long con el manipulador de la ventana a mostrar.
'
'  - wCmdShow. Número Long, constante con el modo de
'    visualización de la ventana.
'
' Valor de retorno. Número Long distinto de cero si la ventana ya
```

```
' era visible o cero en caso contrario.
Declare Function ShowWindow Lib "user32" _
   (ByVal hwnd As Long, _
   ByVal nCmdShow As Long) As Long
' IsIconic()
' Informa si una ventana está iconizada.
' Parámetros:
'  - hwnd. Número Long con el manipulador de la ventana.
'
' Valor de retorno. Número Long distinto de cero si la ventana
' está iconizada o cero en el caso de no estarlo
Declare Function IsIconic Lib "user32" _
   (ByVal hwnd As Long) As Long
```

Código fuente 471

El botón Buscar por título, también efectúa una búsqueda por el valor introducido en el cuadro de texto del formulario, pero en este caso, el contenido del cuadro de texto ha de ser exactamente igual al título de la ventana buscada.

```
Private Sub cmdBuscarTit_Click()
' buscar una ventana sólo por el título,
' se debe introducir el título completo
' y exacto de la ventana, de lo contrario,
' la búsqueda no tendrá éxito
Dim llResultado As Long
Dim llhWnd As Long
llhWnd = FindWindow(vbNullString, Me.txtBuscarWin.Text)
If llhWnd <> 0 Then
   If IsIconic(llhWnd) <> 0 Then
      llResultado = ShowWindow(llhWnd, SW_RESTORE)
   End If
   ' guardar el manipulador para el caso
   ' de que se le aplique parpadeo
   mlhWndParpadeo = llhWnd
   BringWindowToTop llhWnd
Else
   MsgBox "No se ha encontrado la ventana"

End If
End Sub
```

Código fuente 472

En cuanto al botón Cerrar, al igual que los anteriores, busca una ventana por el valor del TextBox del formulario, si la encuentra, le envía un mensaje de cierre, tal como vemos en el código fuente 473.

```
Private Sub cmdCerrarWin_Click()
' buscar una ventana y cerrarla
Dim llResultado As Long
Dim llhWnd As Long
Dim lcTitulo As String
' tomar el manipulador correspondiente
' al escritorio
llhWnd = GetDesktopWindow()
' tomar el manipulador correspondiente
' a la primera ventana hija del escritorio
llhWnd = GetWindow(llhWnd, GW_CHILD)
```

```
' en el siguiente bucle, tomar los manipuladores
' del resto de ventanas hija del escritorio
' hasta encontrar la que debemos cerrar
Do While llhWnd <> 0
   ' tomar el título de la ventana de la cuál
   ' acabamos de recuperar el manipulador
   lcTitulo = Space(70)
   llResultado = GetWindowText(llhWnd, lcTitulo, 70)

   ' comprobar si parte del título de la ventana
   ' coincide con la cadena introducida por el
   ' usuario
   If InStr(UCase(lcTitulo), UCase(Me.txtBuscarWin.Text)) > 0 Then
      ' si hemos localizado la ventana, cerrarla
      llResultado = SendMessageNum(llhWnd, WM_CLOSE, 0, 0)
      Exit Sub
   End If

   ' recuperar la siguiente ventana
   llhWnd = GetWindow(llhWnd, GW_HWNDNEXT)

Loop
End Sub
```

Código fuente 473

Para finalizar, veamos un efecto muy curioso, que podemos conseguir para atraer la atención del usuario sobre una ventana en particular. Se trata de hacer que su título parpadee, cambiando el color de activo a inactivo; al igual que sucede cuando durante la impresión de un documento, si se termina el papel, aparece un cuadro de aviso, que comienza a parpadear pasado un determinado tiempo, para que el usuario coloque más papel en la impresora.

La función del API que se utiliza para cambiar el color de una ventana es FlashWindow(), de la que podemos ver su declaración en el código fuente 474.

```
' FlashWindow()
' Aplica un efecto de parpadeo a una ventana, cambiando el
' color del título.
' Parámetros:
'  - hwnd. Número Long con el manipulador de la ventana.
'  - bInvert. Número Long, distinto de cero para cambiar el tono
'    del título. Cero para restaurar el título a su tono
'    original.
'
' Valor de retorno. Número Long distinto de cero si la ventana
' estaba activa al llamar a esta función o cero si estaba inactiva.
Declare Function FlashWindow Lib "user32" _
   (ByVal hwnd As Long, _
   ByVal bInvert As Long) As Long
```

Código fuente 474

Cada vez que se llama a esta función, se cambia el color de la ventana pasada como parámetro, de activo a inactivo y viceversa. Para conseguir un efecto continuado de parpadeo, utilizaremos el control Timer que el lector puede ver en el formulario durante la fase de diseño, que será llamado cada medio segundo.

La ventana a la que se aplicará este efecto será la última buscada mediante los primeros botones del formulario, guardando su manipulador en la variable mlhWndParpadeo, visible en todo el módulo del formulario.

Al seleccionar la opción de menú Archivo+Parpadeo <modo>, en el formulario, se comprobará si esta opción de menú está o no, marcada, activando o desactivando el mencionado efecto. En el código fuente 475 se muestra al lector el código de esta opción y el que se ejecuta cada vez que se produce el evento Timer() del control tmrTemporizador.

```
Private Sub mnuArParpadeo_Click()
Dim llResultado As Long
' activar/desactivar el parpadeo
' en una ventana
If Me.mnuArParpadeo.Checked Then
   ' la ventana parpadea, desactivar parpadeo
   Me.mnuArParpadeo.Caption = "Parpadeo inactivo"
   Me.mnuArParpadeo.Checked = False
   Me.tmrTemporizador.Enabled = False
   ' habilitar botones
   Me.cmdBuscarManip.Enabled = True
   Me.cmdBuscarTit.Enabled = True
   Me.cmdCerrarWin.Enabled = True
Else
   ' la ventana no parpadea, activar parpadeo
   Me.mnuArParpadeo.Caption = "Parpadeo activo"
   Me.mnuArParpadeo.Checked = True
   Me.tmrTemporizador.Enabled = True

   ' deshabilitar botones para evitar errores
   Me.cmdBuscarManip.Enabled = False
   Me.cmdBuscarTit.Enabled = False
   Me.cmdCerrarWin.Enabled = False

End If
' al desactivar el temporizador,
' forzar a que la ventana que parpadeaba
' quede con el título inactivo
If Not Me.tmrTemporizador.Enabled Then
   llResultado = FlashWindow(mlhWndParpadeo, 0)
End If
End Sub
' ----------------------------------------
Private Sub tmrTemporizador_Timer()
Dim llResultado As Long
' invertir el título de la ventana
' para dar el aspecto de parpadeo
llResultado = FlashWindow(mlhWndParpadeo, 1)
End Sub
```

Código fuente 475

Apéndices A

Glosario de términos

Normativa de codificación

Bases para la escritura de código fuente y denominación de controles y objetos en los desarrollos.

- **Controles.** Comenzarán por el tipo de control, según la tabla 100.

Prefijo	**Control**
adc	Data (ADO)
Ani	Animación
bed	Pen Bedit
cbo	Combobox y dropdown Listbox
chk	CheckBox
clp	Picture clip
cmd	CommandButton
cll	Collection

col	Column
com	Comunicaciones
ctr	Control Genérico (parámetro)
dat	Data (DAO)
db	Base de datos ODBC
dir	Dir ListBox
dlg	CommonDialog
drv	Drive Listbox
ds	Dynaset ODBC
flb	File Listbox
frm	Form
fra	Frame
gau	Barra gauge
gpb	Group Pushbutton
grd	Grid
hed	Pen Hedit
hsb	Horizontal Scrollbar
img	imagen
iml	ImageList
ink	Pen Lnk
key	Keyboard key status
lbl	Label
lin	Line
lst	Listbox
mpm	MAPI message
mps	MAPI session
mci	MCI

mnu	Menu
opt	OptionButton
ole	OLE Client
pic	Picture
pnl	Panel 3D
shp	Shape
sbr	StatusBar
spn	Spin
tbr	ToolBar
txt	TextBox
tmr	Timer
vsb	Vertical Scrollbar

Tabla 100. Controles

Ejemplos:

```
CommandButton: cmdAceptar
TextBox: txtNombre
CheckBox: chkRevisado
```

- **Menús.** Se usará el prefijo *mnu* seguido de un literal descriptivo de la opción, dependiendo del nivel en que se encuentre la opción.

  Tomemos una estructura de menú como la siguiente:

  Archivo

  Fichero

  Consultas -> Clientes

  Proveedores

  Salir

  Los nombres a asignar a cada elemento del menú quedarían de la siguiente forma:

  mnuArchivo

  mnuArFichero

  mnuArConsultas -> mnuArCoClientes

mnuArCoProveedores

mnuArSalir

Según lo visto, las opciones de menú situadas en el menú principal o primer nivel utilizarán el nombre completo de la opción. Las opciones en los niveles inferiores utilizarán las dos primeras letras del nivel superior seguidas del nombre de su opción.

- **Variables y funciones.** La estructura para denominar variables y funciones será la siguiente:

  ```
  <ámbito><tipo><cuerpo>
  ```

  Valores para ámbito:

g	Global
l	Local
s	Static
m	Módulo
v	Variable pasada por valor
r	Variable pasada por referencia

Tabla 101. Valores para ámbito

Valores para tipo:

b	Boolean
by	Byte
m	Currency 64 bit
d	Double 64 bit
db	Database
dt	Date+Time
f	Float/Single 32 bit
h	Handle
l	Long
n	Integer
c	String
u	Unsigned

ul	Unsigned Long
vnt	Variant
w	Word
a	Array
o	Objeto

Tabla 102. Valores para tipo

Valores para tipos de datos:

RDO

EN	Environment
CN	Connection
QY	Query
RSF	Resulset forward only
RSS	Resulset static
RSD	Resulset dynamic
RSK	Resulset keyset
RSB	Resulset client batch
RSN	Resulset no cursor

Tabla 103. Valores para tipo RDO

DAO

WS	Workspace
RCF	Recordset forward only
RCS	Recordset snapshot
RCD	Recordset dynaset
RCT	Recordset table

Tabla 104. Valores para tipo DAO

ADO

acn	Connection
acm	Command
ars	Recordset
apr	Parameter
afl	Field

Tabla 105 Valores para tipo ADO

Para el cuerpo de la variable se utilizará WordMixing.

Ejemplos:

- Variable local, integer, para guardar un número de referencia.

  ```
  LnNumReferencia
  ```

- Variable global, cadena de caracteres, que contiene el código de usuario de la aplicación.

  ```
  GcUsuarioApp
  ```

- Variable a nivel de módulo, cadena de caracteres, que contiene el nombre de la aplicación.

  ```
  McAppNombre
  ```

- **Constantes.** Seguirán las mismas reglas que las variables para el ámbito y tipo. El cuerpo de la constante deberá ir en mayúsculas, y en vez de utilizar WordMixing, se utilizará el guión bajo "_" para separar los distintos componentes del cuerpo.

  Ejemplo:

  ```
  gnNUMERO_LINEA

  Constante NUMERO_LINEA, global, de tipo integer.
  ```

Apéndices

B

Tratamiento de datos con Visual Basic 5

Introducción a la programación con bases de datos

Una base de datos es una estructura diseñada para el manejo de información. Dicha información se clasifica dentro de la base de datos en tablas. Una tabla es un grupo de datos que hacen referencia a un tema determinado como pueda ser la información de clientes de una empresa, las facturas emitidas por la empresa, etc. Dentro de la tabla, los datos se organizan en filas o registros; si la tabla contiene la información de clientes, una fila hará referencia a la información de uno de los clientes. A su vez las filas están compuestas por columnas o campos, que hacen referencia a un aspecto concreto de la fila; siguiendo con el ejemplo de los clientes, un campo de una fila de clientes puede ser el nombre del cliente, la dirección, el teléfono, etc.

La figura 492 nos muestra la representación gráfica de una tabla, en la que podemos ver sus filas y columnas.

Las bases de datos están contenidas en un *Sistema Gestor de Bases de Datos Relacionales (SGBDR),* que es el que proporciona los mecanismos necesarios para realizar consultas de datos, mantener los niveles de integridad de la información, seguridad en el acceso, etc. A este conjunto de mecanismos se le denomina también *motor de datos.*

Según el motor de datos utilizado, el SGBDR proporcionará un conjunto de capacidades que en función de su número u optimización darán una mayor o menor potencia a la base de datos utilizada con respecto a otras

Figura 492. Tabla de Clientes.

Diseño de la base de datos

Esta es la primera tarea que hemos de acometer en el desarrollo de una aplicación que manipule una base de datos, los pasos a dar en esta etapa del proyecto son los siguientes:

Establecer el modelo de la aplicación

En este paso, debemos analizar lo que la aplicación debe hacer, recabando información del futuro usuario/s de la aplicación. En función de sus necesidades, comenzaremos a tomar idea de las tareas que ha de afrontar el programa. Una vez que tenemos una visión global del trabajo a realizar, debemos separarlo en varios procesos para facilitar su programación.

Analizar que información necesita la aplicación

En este punto, y una vez averiguadas las tareas a realizar por el programa, hemos de determinar que información tenemos que incluir en la base de datos y cual no, de manera que no sobrecarguemos las tablas con datos innecesarios que podrían retardar la ejecución del programa.

Un ejemplo lo tendríamos en la típica tabla de clientes para un comercio; si se desea enviar a cada cliente una felicitación por su cumpleaños, es necesario incluir un campo con su fecha de nacimiento, en caso contrario, no se incluirá este campo evitando sobrecargar la tabla.

Establecer la información en tablas

Para organizar los datos en una aplicación, comenzaremos agrupándolos por temas. De esta manera los datos que tengan que ver con clientes formarán la tabla Clientes; los que tengan que ver con proveedores la tabla Proveedores, y así sucesivamente, por cada tema crearemos una tabla.

Hemos de tener cuidado en esta fase de diseño, puesto que por ejemplo, la información de las tablas Clientes y Proveedores (Código, Nombre, Dirección, etc.) es muy similar y podríamos pensar en ponerla en una única tabla, pero en el caso de los clientes, tendríamos que incluir un campo con la dirección del almacén de entrega de la mercancía, mientras que como a los proveedores no vamos a realizarles entregas, dicho campo no sería necesario, lo que ocasionaría muchas filas en la tabla con este campo inutilizado cuando se tratara de proveedores.

Normalización de la base de datos

Una vez creadas las tablas, el proceso de normalización consiste en eliminar la información redundante existente entre las tablas. Pongamos como ejemplo una tabla que contiene facturas, y cada fila de la tabla incluye los datos del cliente, como aparece en la figura 493.


CabFacturas : Tabla

Nombre	Direccion	Poblacion	NIF	NumFac	Fecha	Articulo	Importe
J.Torre	Alce 2	Madrid	1224466	35	10/05/98	Mesa	5000
I.Benitez	Cruces 5	Salamanca	5336678	36	5/06/98	Silla	2000
M.Gutierrez	Arcipreste 12	Soria	3188775	37	20/06/98	Banqueta	1200
E.Piquer	Valle 4	Zaragoza	5432311	38	22/06/98	Repisa	3000
							0

Registro: 2 de 4

Figura 493. Tabla de facturas con información sin normalizar.

Si en lugar de incluir todos los datos del cliente, creamos una tabla para los clientes con su información, agregando un campo de código, y en la tabla de cabecera de facturas, sustituimos los campos de clientes por el código de cliente, ahorraremos un importante espacio en la tabla de facturas y seguiremos manteniendo la referencia al cliente que hemos facturado, ya que el código de cliente nos llevará a sus datos dentro de la tabla Clientes, observémoslo en la figura 494.

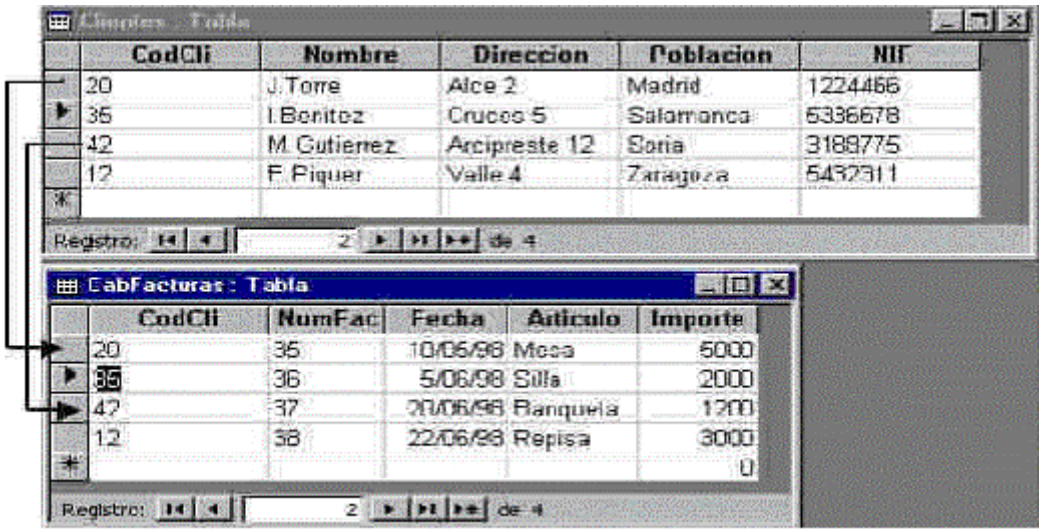


Figura 494. Información normalizada entre las tablas Clientes y CabFacturas.

Un factor muy importante a efectos del rendimiento de la aplicación, es realizar un estudio del tamaño necesario para los campos de las tablas, establecer por ejemplo, un tamaño de 50 caracteres para un campo de NIF es estar desperdiciando claramente espacio en el disco duro por un lado, y por otro, si tenemos varios campos con un desfase parecido, cuando la tabla comience a crecer, todo ese conjunto de pequeños detalles de diseño se volverá contra nosotros en forma de lentitud a la hora de hacer operaciones con la tabla.

Un elemento muy útil a la hora de aprovechar espacio lo constituyen las tablas de datos generales. Estas tablas contienen información que puede ser utilizada por todos los procesos de la aplicación, y sirven de apoyo a todas las tablas de la base de datos. Como ejemplo tenemos una tabla que guarde códigos postales, que puede ser utilizada tanto por una tabla de clientes como de proveedores, bancos, etc.

Relacionar las tablas

Eliminada toda información repetida o innecesaria de las tablas, necesitamos establecer relaciones entre las mismas. Una relación es un elemento por el cual dos tablas se vinculan mediante una clave, que pueden ser del siguiente tipo:

- Clave Primaria. Este tipo de clave consiste en que cada registro de la tabla es identificado de forma única mediante el valor de un campo o combinación de campos. Si intentamos dar de alta un registro con una clave ya existente, el motor de datos cancelará la operación provocando un error. Una clave primaria puede ser el código de un cliente (figura 495) o un código postal.

Figura 495. Tabla Clientes con clave primaria establecida en el campo CodCli.

- Clave Ajena. Esta clave se forma mediante el valor de un campo o combinación de campos de una tabla, y tiene como propósito establecer una relación con la clave primaria de otra tabla, pudiendo existir varios registros de la clave ajena que se relacionen con un único registro en la clave primaria de la tabla relacionada. En la figura 496, el campo CodCli de la tabla CabFacturas es clave ajena del campo del mismo nombre en la tabla Clientes, existiendo varias filas de CabFacturas con el mismo valor, que se corresponden con una única fila de Clientes.

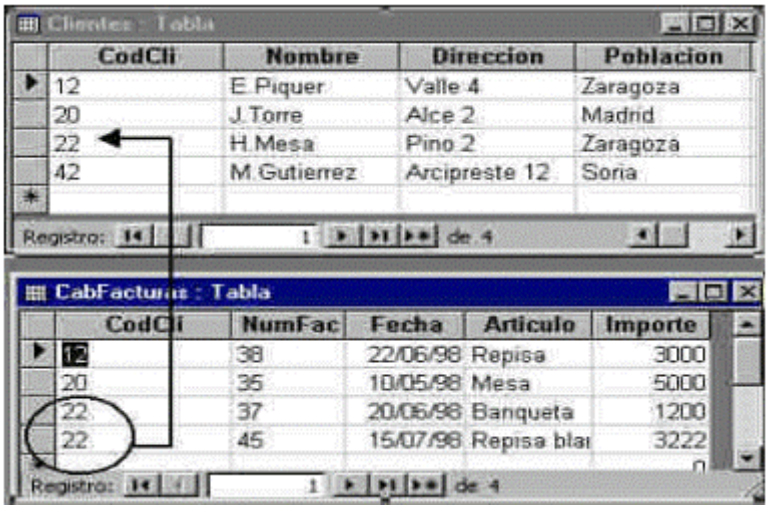


Figura 496. La tabla CabFacturas tiene una clave ajena en el campo CodCli.

En las relaciones por otra parte, tenemos los siguientes tipos:

- Relación de uno a varios. Este es el tipo de relación más común; en él, una fila de una tabla se relaciona con una o más filas de una segunda tabla, pero cada fila de la segunda tabla se relaciona sólo con una de la primera. Un ejemplo claro de esta situación lo tenemos en la figura anterior, cada fila de la tabla Clientes puede relacionarse con una o varias filas de la tabla CabFacturas por el campo CodCli; pero cada fila de la tabla CabFacturas se relaciona sólo con una fila de la tabla Clientes.

- Relación de varios a varios. Esta relación se produce, cuando cada fila de una tabla se relaciona con cada fila de una segunda tabla y viceversa. En esta situación debemos definir una tabla intermedia que tenga relaciones de uno a varios con las otras dos tablas. Pongamos como ejemplo un almacén de productos cosméticos que distribuye a diferentes perfumerías; cada perfumería puede adquirir varios tipos de cosméticos, e igualmente, cada cosmético puede ser adquirido por varias perfumerías. Al trasladar esta información a una base de datos podríamos hacerlo como muestra la figura 497.

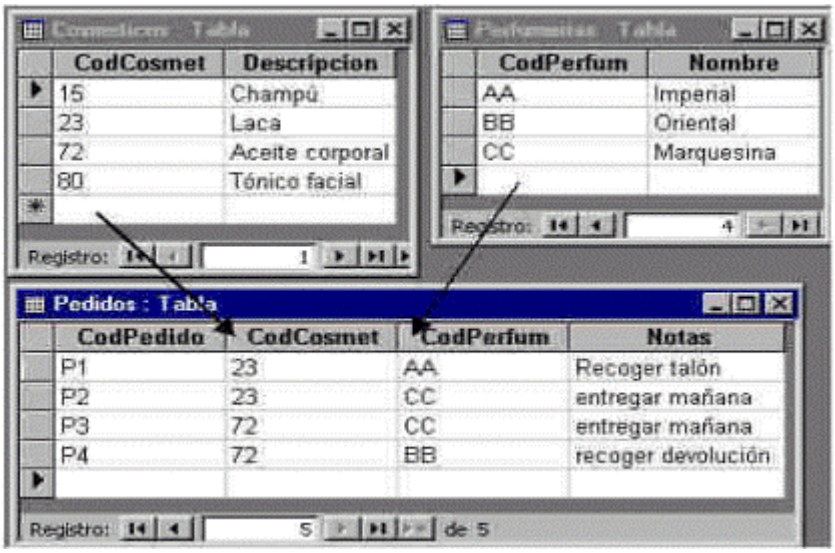


Figura 497. Relación de varios a varios resuelto con una tabla intermedia.

Establecer la integridad referencial

Una vez que se establece una relación entre dos tablas de la base de datos, el administrador de la base de datos debe decidir, en el caso de que el motor de datos lo permita, si se debe gestionar la integridad referencial.

La integridad referencial es la operación por la cual se debe mantener la consistencia entre la información de las tablas de la base de datos. Pensemos en el ejemplo anterior de las tablas Clientes y CabFacturas; si hemos establecido una relación entre las dos tablas a través del campo CodCli de ambas, y eliminamos en la tabla Clientes la fila que contiene el código de cliente 42, nos encontraremos con un problema de inconsistencia de información, ya que cuando desde la tabla CabFacturas nos posicionemos en la fila que contiene el código de cliente 42 y vayamos a recuperar los datos de ese código de cliente a la tabla Clientes, nos encontraremos con que no existe ninguna fila con el código 42, en este caso, al registro no relacionado de la tabla CabFacturas se le denomina *registro huérfano*.

El tema de la integridad referencial es una decisión de diseño, puede exigirse o no; en el caso de que sea necesaria, será responsabilidad del programador mantenerla o del motor de datos en el caso de que este último la soporte. De cualquiera de las maneras, al exigir integridad referencial nos aseguramos que no quedarán registros huérfanos entre las tablas relacionadas, puesto que si intentamos modificar los valores que forman parte de la relación y no existe el nuevo valor introducido como parte de la clave, se producirá un error que no permitirá completar el proceso.

Adicionalmente se pueden establecer actualizaciones y eliminaciones en cascada, de forma que si se modifica o borra una fila en la primera tabla de la relación, también se modificarán o borrarán los registros relacionados en la segunda tabla.

Definir índices para las tablas

En este punto hemos de analizar que tablas y campos de la base de datos son susceptibles de ser consultados un mayor número de veces, y definir un índice para ellos, ya que al ser información que necesitaremos consultar con más frecuencia que el resto, también será necesaria recuperarla más rápidamente.

Un índice ordena las filas de una tabla basándose en un criterio determinado. La clave del índice puede estar formada por uno o varios campos, de esta manera podemos encontrar rápidamente filas en la tabla, o mostrarla ordenada sobre la base de la clave del índice.

Definir validaciones en la inserción de datos

La validación consiste en comprobar que los datos que vamos a incluir en la fila de una tabla son correctos. Podemos definir validaciones a nivel de motor, de forma que sea el propio motor de datos el que compruebe la información y la rechace en caso de no ser válida. Otra manera de introducir validaciones es mediante el código del programa, lo cual veremos cuando manejemos datos usando código.

El motor Jet

A lo largo de este tema y salvo indicación expresa, haremos uso del motor Jet. Este motor se incluye con VB y proporciona acceso nativo a ficheros de DATOS.MDB, los mismos utilizados por Access. Al hablar del formato nativo de datos, nos referimos a que cuando usemos controles y objetos de datos en nuestras aplicaciones, si no indicamos el formato de datos al que queremos acceder, por defecto se usará el FORMATO.MDB; por este motivo, los ejemplos de diseño de las bases de datos (creación de tablas, índices, relaciones, etc.) se realizarán usando Access, aunque como veremos hacia el final de este tema, VB incluye entre sus complementos un administrador visual de datos llamado *VisData*, que también nos permite realizar estas labores.

El motor Jet permite acceder también a bases de datos en formato FoxPro, dBASE, Paradox, etc., y utilizar controladores ODBC para realizar conexiones a bases de datos que soporten esta tecnología.

Sin tener la potencia de grandes gestores de datos como SQL Server, Oracle, etc., Jet puede servir perfectamente para acometer proyectos de pequeño y mediano tamaño, proporcionando una gran flexibilidad en la manipulación de los datos.

La tabla 106 muestra los tipos de datos que puede manejar el motor Jet.

Tipo de datos	**Tamaño**
Texto, valores alfanuméricos	El valor definido en el campo hasta 255 caracteres
Memo	Hasta 65.535 caracteres en el caso de que se vaya a manejar sólo texto y no información binaria
Numérico	1, 2, 4 u 8 bytes (16 bytes si el valor de la propiedad Tamaño del campo es Id. de réplica). Valores numéricos con los cuales se vaya a operar matemáticamente
Fecha / Hora	8 bytes. El intervalo de años comprende desde el 100 hasta el 9999
Moneda	8 bytes, con precisión de 15 dígitos y 4 dígitos decimales
Autoincremental	4 bytes (16 bytes si el valor de la propiedad Tamaño del campo es Id. de réplica). Número que es incrementado por el motor de datos cada vez que se agrega un registro a una tabla
Lógico	1 bit. Valores Verdadero / Falso
Objeto	Hasta 1 gigabyte, si el disco duro lo permite. Puede contener un objeto OLE tal como un documento Word
Hipervínculo	Enlace a una dirección de Internet. Cada una de las tres partes de un hipervínculo puede contener hasta 2048 caracteres. El hipervínculo puede tener un texto que es el que se presenta al usuario; una ruta de acceso a un archivo o página de Internet; y una subdirección, que indica la posición dentro del archivo o página

Asistente para búsquedas	Crea un tipo de campo especial que contiene una lista de valores que puede estar vinculada a los valores de un campo de otra tabla de la base de datos. Su tamaño es igual al del campo clave principal utilizado para realizar la búsqueda (habitualmente 4 bytes).

Tabla 106. Tipos de datos en Access.

Establecidas unas mínimas bases teóricas para afrontar el manejo de datos, pasaremos al desarrollo de aplicaciones que realicen mantenimientos de los mismos. Comenzaremos empleando los medios más sencillos, de forma que podamos crear un mantenimiento simple con el mínimo trabajo, gradualmente iremos complicando el desarrollo para agregar una mayor potencia a los mantenimientos.

El control Data

Visual Basic proporciona mediante el control Data y los controles enlazados a datos, el medio más fácil y rápido de acceder a la información contenida en una base de datos. La aplicación de ejemplo *DataCtl1*, muestra como sin escribir ni una línea de código, podemos crear un mantenimiento para una tabla de una base de datos; un mantenimiento limitado, eso sí, pero piense el lector por otra parte que todo se ha elaborado sin utilizar el editor de código, sólo empleando el editor de formulario, el cuadro de herramientas y la ventana de propiedades. Debemos tener en cuenta que la mayor potencia del control Data reside en su sencillez de manejo.

Esta aplicación se acompaña de la base de datos DataCtl1.MDB, que contiene la tabla Clientes, utilizada para las pruebas con los controles.

Cuando el control Data accede a los datos, internamente utiliza un objeto DAO de tipo Recordset; estudiaremos con detalle este objeto en el apartado dedicado a DAO, por ahora baste saber que un objeto de este tipo contiene el grupo de registros que va a consultar el usuario.

Vamos a dar a continuación, los pasos necesarios para crear el programa. En este ejemplo nos limitaremos a configurar el control Data el mínimo necesario para que funcione, algunas de las propiedades a las que habitualmente se da valor como Name van a quedar con su valor por defecto, todo ello tiene como objetivo que el lector aprecie la configuración básica que necesita el control para poder funcionar.

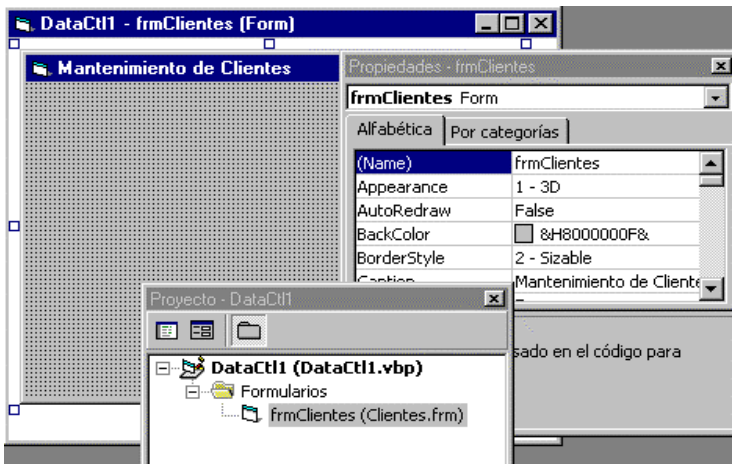


Figura 498. Formulario frmClientes de la aplicación DataCtl1.

En primer lugar, creamos un nuevo proyecto EXE estándar, agregando un nuevo formulario o empleando el que incorpora por defecto el proyecto; a este formulario le asignamos *frmClientes* en su propiedad Name y lo grabamos al disco como CLIENTES.FRM. Figura 498.

A continuación seleccionamos del cuadro de herramientas el control Data, figura 499, y lo dibujamos en el formulario.

Figura 499. Icono del control Data en el cuadro de herramientas.

Los botones de navegación de este control nos permitirán desplazarnos por los registros seleccionados.

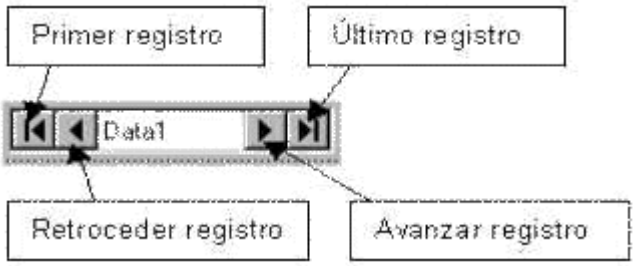


Figura 500. Botones de navegación del control Data.

A continuación especificaremos las siguientes propiedades para el control:

- DatabaseName. Especifica la ruta y el nombre de la base de datos que vamos a utilizar, bien tecleando su nombre o pulsando en el botón con puntos suspensivos para seleccionar el fichero mediante una caja de diálogo, para este ejemplo (figura 501), deberemos crear el directorio C:\CursoVB6\Texto\DatosDAO\DataCtl1, y depositar allí los ficheros de esta aplicación.

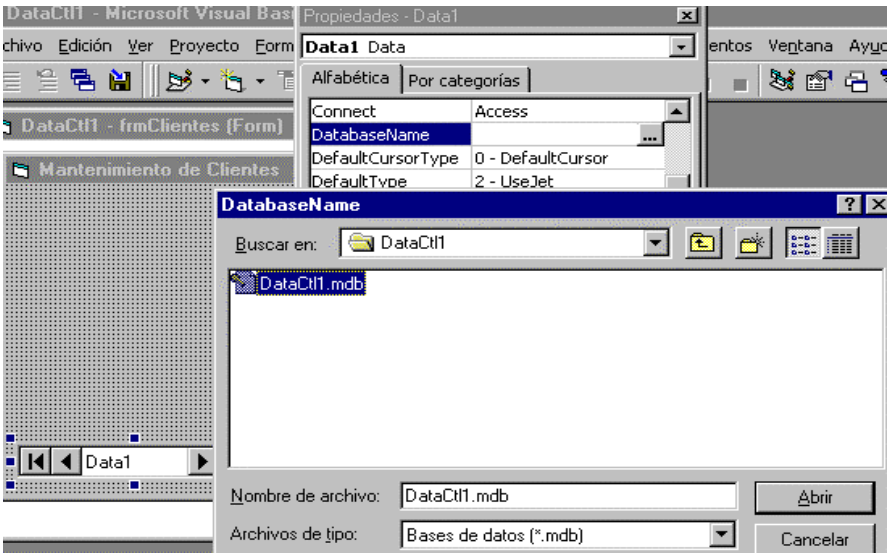


Figura 501. Establecimiento de la propiedad DatabaseName en un control Data.

- RecordSource. Una vez que el control sabe con que base de datos va a trabajar, usamos esta propiedad para seleccionar la tabla que vamos a manejar (figura 502). Podemos indicar el nombre de una tabla, QueryDef o una sentencia SQL.

Figura 502. Propiedad RecordSource del control Data.

En nuestro caso, seleccionamos la tabla Clientes de una lista que incorpora esta propiedad.

- BOFAction. Establece el comportamiento del control al llegar al principio del Recordset. La tabla 107 muestra los posibles casos.

Acción	Constante	Valor	Descripción
Move First	vbMoveFirst	0	El puntero del Recordset permanece en el primer registro.
BOF	vbBOF	1	El puntero del Recordset se sitúa en la zona previa al primer registro, el botón retroceder del control se deshabilita.

Tabla 107

- EOFAction. Establece el comportamiento del control al llegar al final del Recordset. La tabla 108 muestra los posibles casos.

Acción	Constante	Valor	Descripción
Move Last	vbMoveLast	0	El puntero del Recordset permanece en el último registro.
EOF	vbEOF	1	El puntero del Recordset se sitúa en la zona posterior al último registro, el botón avanzar del control se deshabilita.

Add New	vbAddNew	2	Agrega un nuevo registro y se sitúa en él. Si se añade información a sus campos, el nuevo registro queda incorporado a la tabla, en caso contrario se pierde.

Tabla 108

Otras propiedades que pueden ser utilizadas, aunque no sean necesarias en este ejemplo concreto son:

- Exclusive. Si contiene False, el resto de usuarios podrán acceder a la base de datos; si contiene True, sólo podrá usar la base de datos el usuario de nuestra aplicación.

- ReadOnly. Si contiene False, se podrá modificar y agregar información a la base de datos; si contiene True sólo será posible consultar datos sin editar.

Llegados a este punto, el control Data está listo para recuperar información de la tabla, pero este control por sí mismo, no puede mostrar datos al usuario, por este motivo se hace necesario incluir controles que puedan enlazarse al Data y mostrar la información que este contiene.

Lo que haremos ahora, será incluir desde el cuadro de herramientas, un control TextBox y un Label por cada campo de la tabla Clientes. En la propiedad Caption de los Label pondremos el nombre de cada campo y en los TextBox modificaremos las siguientes propiedades:

- DataSource. Esta propiedad abre una lista con todos los controles Data que contiene el formulario; en nuestro caso sólo disponemos de uno, que es el que seleccionamos.

- DataField. Esta propiedad abre una lista con los campos que contiene el control Data incluido en la propiedad DataSource del control enlazado. Lo que haremos aquí será seleccionar un campo diferente para cada TextBox.

- MaxLength. Aquí definimos el máximo número de caracteres que admite el TextBox, si no usáramos esta propiedad e intentáramos por ejemplo poner más de dos dígitos en el campo del código de cliente, VB nos avisaría que esto no es posible y no permitiría agregar el nuevo registro hasta que los datos fueran correctos.

Una vez hecho esto, ya hemos finalizado el trabajo, iniciamos la aplicación que tendrá un aspecto como el de la figura 503.

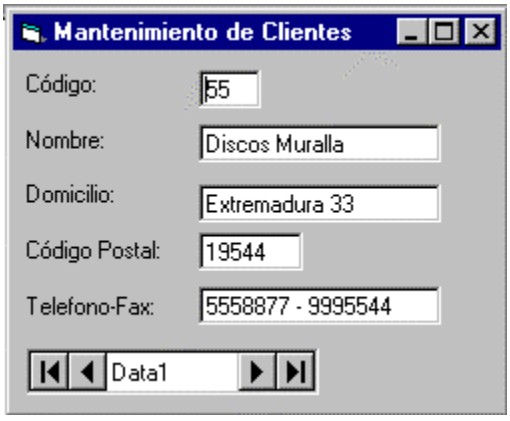


Figura 503. Mantenimiento usando control Data.

Ahora podemos utilizar el control Data1 para navegar por los registros de la tabla, modificar el registro que aparece en el formulario o añadir nuevos registros desde el final de la tabla; los controles TextBox serán rellenados automáticamente con la información del registro actual, y todo esto se ha realizado utilizando básicamente el ratón. Por supuesto que una aplicación real, siempre requerirá un tipo de validación a la hora de editar datos, que nos obligue a utilizar algo más que el simple control, pero puede darse el caso de que necesitemos desarrollar un programa en el cual sea necesaria la introducción inmediata de datos; mientras se desarrollan los procesos definitivos de entrada de datos, podemos utilizar formularios de este estilo para que los usuarios puedan comenzar a introducir información, siempre que el diseño del programa y base de datos lo permitan.

Validaciones a nivel de motor

Pongámonos en el caso de que en el anterior programa no se debe dar un valor inferior a "10" para el campo código de cliente, si queremos conseguirlo sin tener que escribir una rutina de validación, podemos emplear una regla de validación a nivel de motor, que consiste en que el propio motor de datos se encarga de comprobar los valores a introducir en un registro, si no se cumple el criterio de validación establecido, no se grabará el registro.

Puesto que comparte el motor Jet, vamos a utilizar Access para establecer la regla, ya que nos permite una mayor sencillez en este tipo de operaciones.

Abrimos en Access la base de datos DataCtl1.MDB y la tabla Clientes en modo diseño, nos situamos en el campo CodCli, e introducimos la validación tal y como se muestra en la figura 504.

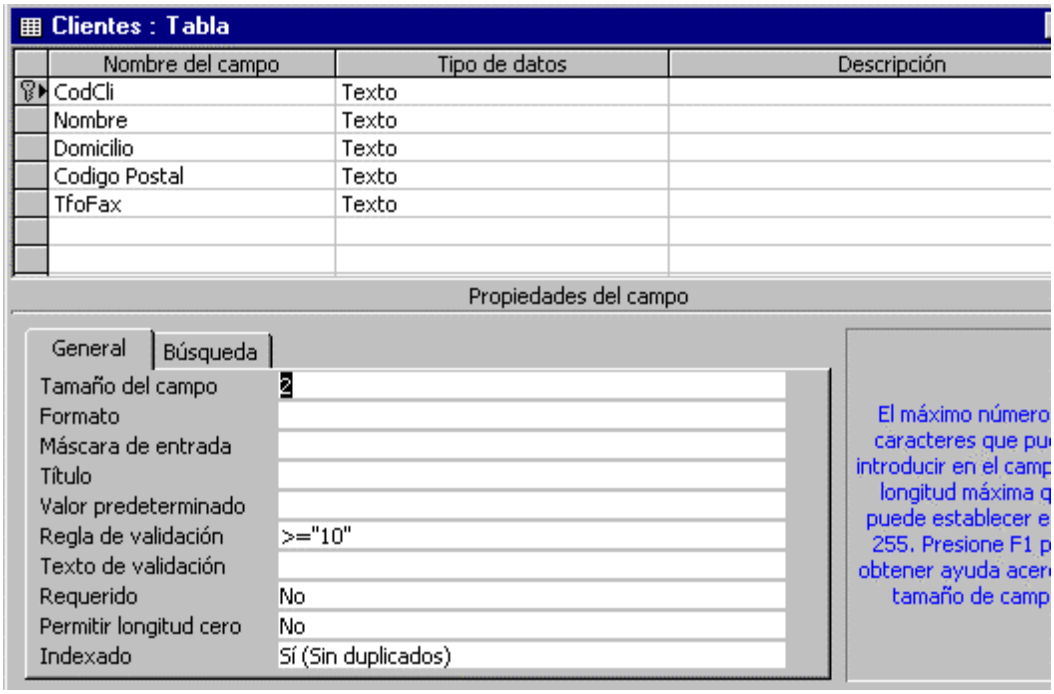


Figura 504. Regla de validación establecida en Access para un campo de una tabla.

Grabamos los cambios realizados en la tabla, cerramos Access y volvemos a ejecutar el programa DataCtl1 desde VB. Ahora si intentamos crear un nuevo registro, dando al campo de código el valor "06" por ejemplo, nos encontraremos con el aviso que muestra la figura 505.

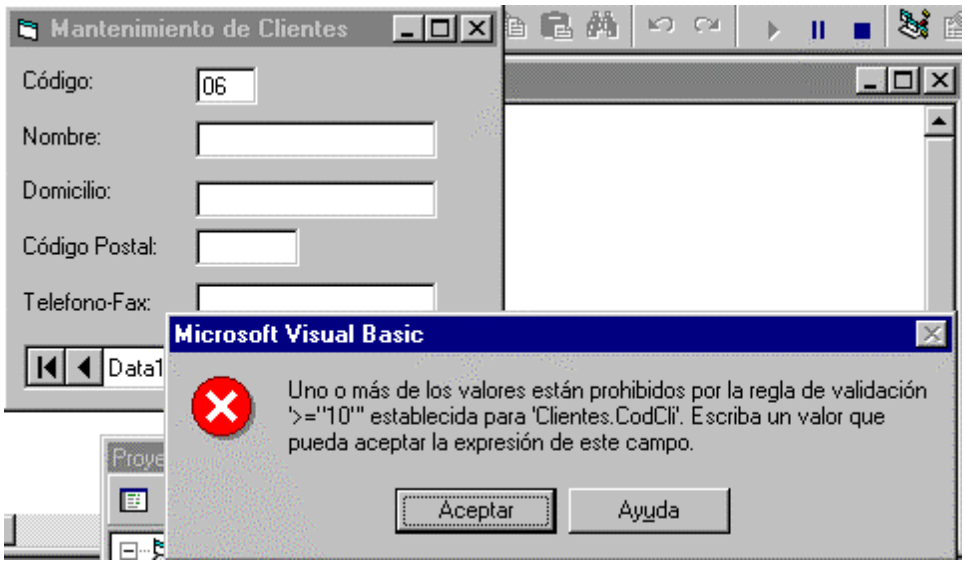


Figura 505. Aviso del motor de datos, por haber transgredido una regla de validación.

No podremos grabar el registro hasta que no cumplamos la regla de validación.

Es posible incluir reglas de validación para todos los campos de la tabla, y aunque la validación mediante una rutina de código siempre será más potente y flexible, este medio de controlar la grabación de los datos siempre puede sernos útil en validaciones sencillas.

El control DBGrid

El mantenimiento del ejemplo anterior tiene un defecto muy importante, y es que el usuario puede necesitar una visión global de los registros de la tabla y no tener que verlos uno a uno. Para solventar ese problema disponemos del control DBGrid o cuadrícula, que muestra los datos en forma de tabla, con lo cual podemos tener de un solo vistazo, una mayor cantidad de información y navegar de forma más rápida por las filas.

En el caso de que este control no se encuentre en el cuadro de herramientas, deberemos incorporarlo mediante la ventana de Componentes (ver tema dedicado al entorno de desarrollo), seleccionando el control denominado *Microsoft Data Bound Grid Control*. Una vez seleccionado el control, se agregará al cuadro de herramientas el icono mostrado en la figura 506.

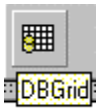


Figura 506. Icono para el control DBGrid en el cuadro de herramientas.

La aplicación de ejemplo DataCtl2 proporciona un mantenimiento similar al visto en el apartado del control Data, pero utilizando un DBGrid en lugar de los TextBox para editar la información de los campos de un registro.

Para este ejemplo, deberemos crear el directorio C:\CursoVB6\Texto\DatosDAO\DataCtl2, y depositar allí los ficheros de esta aplicación.

Los pasos para crear el formulario del programa y el control Data, son iguales que en el anterior ejemplo, por lo que no los repetiremos aquí. Una vez llegados a este punto, insertaremos un control DBGrid de la misma forma que lo hacemos con otro control.

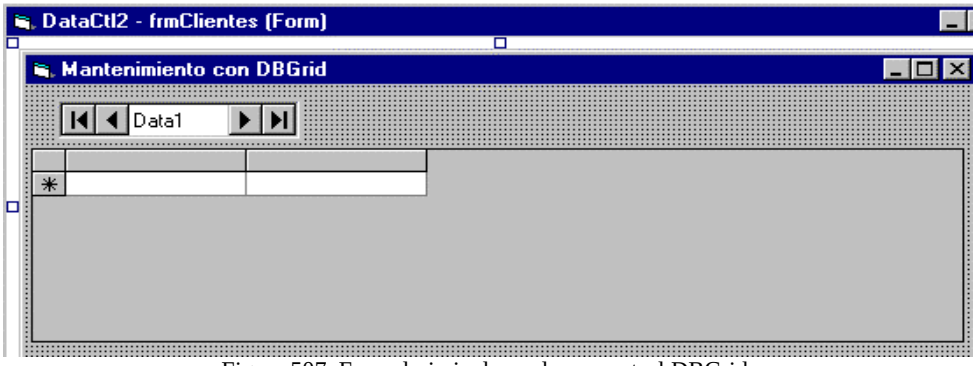


Figura 507. Formulario incluyendo un control DBGrid.

Seleccionando el DBGrid y abriendo su ventana de propiedades, sólo tenemos que completar la propiedad DataSource con el nombre del control Data del formulario, y echar a andar la aplicación, los registros aparecerán en el DBGrid en forma de tabla, como se puede apreciar en la figura 508.

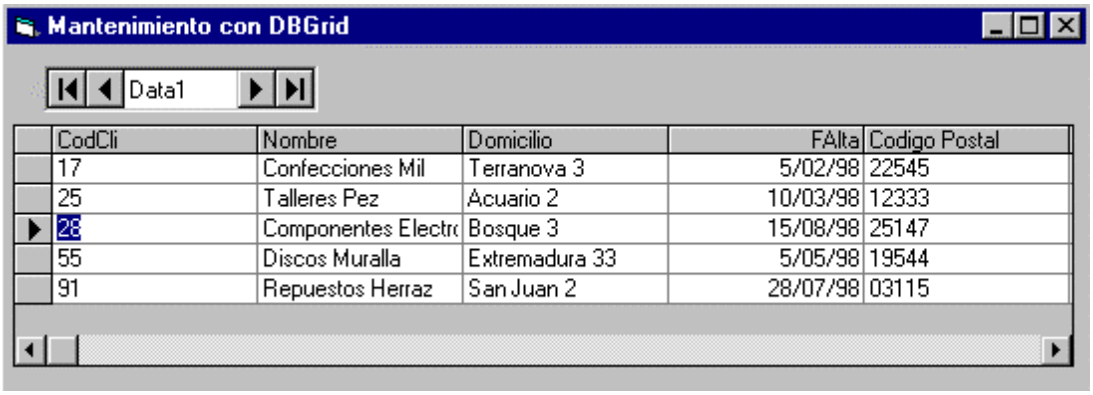


Figura 508. Aplicación de mantenimiento de datos con DBGrid.

Algunas de las mejoras que necesitaría esta aplicación en el estado actual serían: un título en el control DBGrid; poder dar un nombre personalizado en los títulos de las columnas, diferente del nombre de campo de la tabla; ajustar el ancho de las columnas al mostrar la ventana; dar formato a la información; poder editar y agregar registros, etc. Veamos que propiedades tenemos que modificar en el DBGrid para conseguir dotarle de estas cualidades.

En primer lugar, seleccionaremos el control DBGrid del formulario y abriremos su menú contextual, eligiendo la opción *Recuperar campos*; de esta forma, la información de los campos de la tabla que contiene el control Data pasará a la cuadrícula y podrán ser modificados.

Nuevamente abriremos el menú contextual del DBGrid, seleccionando la opción *Propiedades*, que mostrará una ventana del tipo página de propiedades, como la que se muestra en la figura 509, con varias fichas para configurar este control.

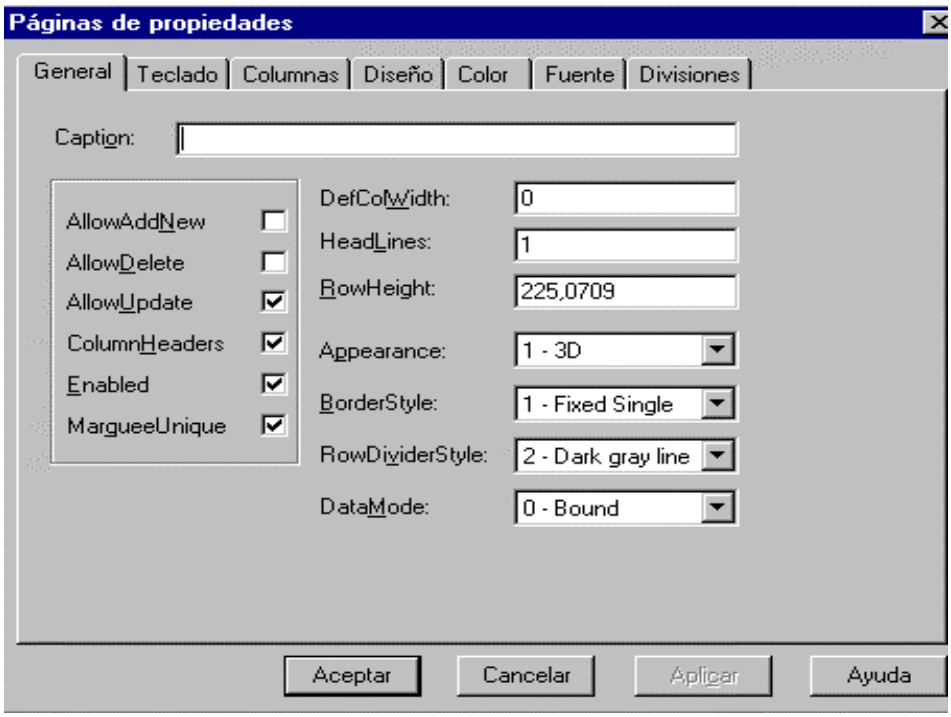


Figura 509. Ventana de páginas de propiedades para un control DBGrid.

A continuación comentaremos algunas de estas propiedades, las que necesitaremos modificar para este ejemplo, invitando al lector a que experimente con cualquier otra que no sea comentada aquí para comprobar sus efectos sobre el control.

- Caption. Texto que servirá de título para la cuadrícula.
- ColumnHeaders. Si está seleccionada esta propiedad, se mostrarán los títulos para las columnas del control.
- AllowAddNew. Si está seleccionada, se creará un nuevo registro al sobrepasar el último registro mostrado en el control.
- AllowDelete. Seleccionando esta propiedad, podremos marcar un registro del DBGrid, haciendo clic en la columna izquierda que indica con una flecha el registro actual, y borrarlo pulsando la tecla *Supr*.
- AllowUpdate. Seleccionando esta propiedad, podremos modificar los registros directamente en el DBGrid.
- AllowArrows. Si está seleccionada, podremos utilizar las teclas de movimiento del cursor para desplazarnos por la cuadrícula.

Como seguramente nos interese poner un nombre más descriptivo en los títulos de las columnas, en vez del nombre de campo, en la ficha *Columnas* de esta ventana podemos poner el texto que queramos. Igualmente podemos poner una cadena para dar formato a los datos que aparezcan en la columna.

En la ficha *Diseño* es posible indicar si una columna se puede dimensionar, establecer el ancho, que sea visible, etc.

Después de haber configurado el DBGrid según nuestras necesidades, volvemos a ejecutar el programa con unos resultados bastante diferentes de los obtenidos al comienzo.

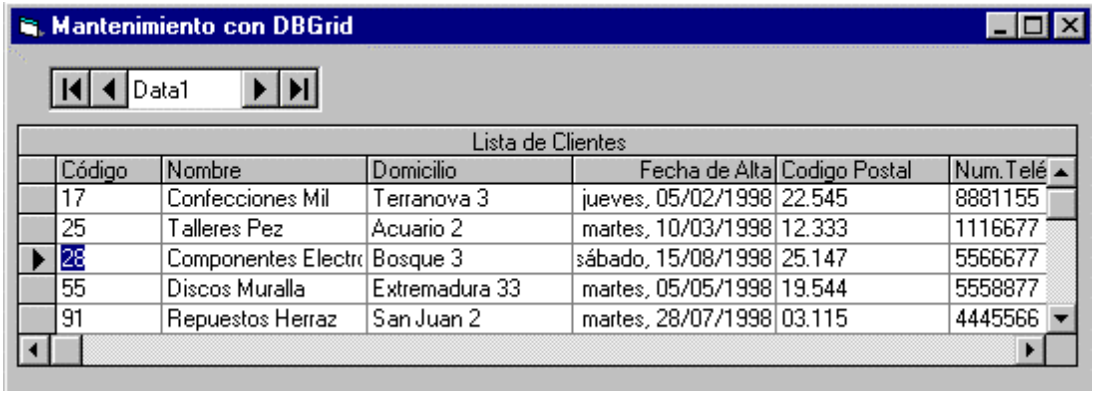


Figura 510. Programa con DBGrid personalizado.

Reflexiones sobre los controles enlazados a datos

Como acabamos de ver, el control Data y el resto de controles de datos, proporcionan un medio rápido y fácil de acceder y editar las filas de una tabla, pero toda esa sencillez se convierte en un problema cuando necesitamos que el programa haga ese *algo más* que siempre se le pide.

Entre el conjunto de operaciones que no son posibles de realizar por el control Data, y para las que tendremos que utilizar código en el programa están el borrado de registros, cambiar el índice activo, realizar una búsqueda de registros, introducir nuevos registros desde cualquier punto de la tabla sin tener que ir hasta el último, y de ahí pasar a un registro vacío, etc.

En lo que respecta al DBGrid, puede que durante el transcurso del programa, necesitemos ocultar y mostrar una determinada columna, cambiar varias veces su título, habilitar y deshabilitar el control para permitir al usuario navegar por las filas, etc.

Otro problema añadido lo constituye el hecho de que en muchas aplicaciones no conocemos durante el desarrollo de las mismas la ubicación de la base de datos, ya que es durante la instalación del programa cuando el usuario proporciona dicha ubicación, grabándose con mucha probabilidad en una entrada del registro de Windows. En este tipo de casos, no podemos indicarle al control donde están los datos, teniendo que utilizar una rutina de código para buscar en el registro la ruta de la base de datos.

Cuando nos encontramos en esta situación, si utilizamos un DBGrid en un formulario, la mejor manera de configurar sus columnas es mediante código, de forma que una vez que sepamos la ruta de los datos y hayamos asignado un conjunto de registros al control Data conectado al DBGrid, escribamos un procedimiento de configuración para el mismo con los títulos de columna, ancho, formato de salida de los datos, etc.

En definitiva, son muchas las situaciones en que el simple uso de los controles de datos no nos ayudará a resolver el problema, por lo cual podemos afirmar, que si realmente queremos potencia y flexibilidad en nuestros mantenimientos de datos, tendremos ineludiblemente que combinar controles

de datos y la creación de un conjunto más o menos numeroso de procedimientos que hagan uso de los objetos de acceso a datos.

Controles enlazados y código, un punto intermedio

Antes de abordar el desarrollo de aplicaciones utilizando fundamentalmente código, vamos a usar una técnica situada a medio camino entre el empleo intensivo de controles o código. Se trata de combinar tanto controles de datos como código para manejar esos datos, de manera que sólo escribamos código allá donde sea absolutamente necesario, delegando en los controles el resto del trabajo. Aprovecharemos al mismo tiempo para presentar un nuevo control de datos: el control DBCombo.

La aplicación de ejemplo

En primer lugar, debemos crear el directorio C:\CursoVB6\Texto\DatosDAO\ComboDat, y depositar en él los ficheros que componen la aplicación *Entradas*, que nos servirá de ejemplo.

Esta aplicación hace uso de la base de datos MERCANCI.*MDB*, que contiene una tabla de productos y otra de envíos de esos productos a diferentes destinos. El objetivo de la aplicación es crear un formulario para introducir la información de cada envío, es decir, cada grupo de datos introducido correspondería al envío de un producto, lo que equivaldría a un registro en la tabla Envios.

Se utilizará un control Data para la tabla Envios, enlazado a los controles TextBox que reflejan el registro actual. Para poder seleccionar los registros de la tabla Productos se usará otro control Data, pero este se enlazará a un control DBCombo, cuyas interesantes características veremos más adelante.

El formulario de datos

Una vez incluidos todos los controles en el formulario frmEntradas, este tendrá el aspecto que muestra la figura 511.

Figura 511. Ventana frmEntradas.

Para navegar por los registros de la tabla Envios, utilizaremos el control Data datEnvios, visualizándose el contenido del registro en los TextBox del formulario. Si queremos insertar un nuevo registro pulsaremos el botón cmdAgregar y escribiremos los valores en los campos correspondientes. Es posible también, modificar directamente el registro actual del formulario, con la ventaja de que si

decidimos suprimir la modificación sólo tenemos que pulsar el botón cmdCancelar. Para seleccionar los productos utilizaremos la lista de productos incluida en el DBCombo.

El control DBCombo

Un control DBCombo es una combinación de cuadro de texto y lista de valores. En función de su estilo es posible seleccionar un valor de la lista desplegable o escribirlo en la zona de texto. Se diferencia del control ComboBox normal, en que es posible enlazarlo a un control Data, de manera que la lista de valores se rellene automáticamente con los valores de uno de los campos del Data. Pero cuando este control muestra todo su potencial es al enlazarlo a un segundo control Data, de forma que se pueda traspasar el valor de un campo del primer Data al segundo.

Situémonos en el ejemplo que ahora nos ocupa. Debemos crear registros para la tabla Envios y uno de los campos a completar es el código del producto que se va a enviar. ¿Pero qué es más fácil, recordar el código de cada uno de los productos o el nombre?, lógicamente el nombre nos será siempre más familiar, y puestos a pedir facilidades, si disponemos de una lista con los nombres de los productos a elegir, nos evitará tener que conocer todas las descripciones de los mismos.

Una vez configurado el DBCombo cboProductos la mecánica de este control es la siguiente: cada vez que seleccionemos el nombre de un producto, el código del mismo será pasado al TextBox del formulario destinado a ese valor. El usuario trabajará mejor empleando los nombres de los productos, mientras que el DBCombo se encargará internamente de usar los códigos.

Para conseguir que el control cboProductos se comporte de esta manera, las propiedades a configurar son las siguientes. Se indica entre paréntesis el valor utilizado para cada propiedad en este ejemplo.

- RowSource. Nombre del control Data que se utiliza para llenar la lista (datProductos).
- ListField. Nombre de un campo del recordset contenido en RowSource con cuya información se completará la lista del control (Descrip).
- DataSource. Nombre del control Data al que se traspasa un valor cada vez que se selecciona un elemento de la lista (datEnvios).
- DataField. Nombre de un campo del recordset contenido en DataSource al que se traspasa el valor seleccionado en la lista (CodProd).
- BoundColumn. Nombre de un campo del recordset incluido en RowSource cuyo valor se traspasa a DataField cuando se hace una selección en la lista (CodProd).

Abra el lector la ventana de propiedades del control cboProductos y comprobará que la configuración de las propiedades antes descritas para este control usan el campo Descrip para mostrar los nombres de los productos en la lista, pero el valor a traspasar entre los controles Data es el del campo CodProd.

El código a incluir

Como hemos dicho, en esta aplicación se va a incluir sólo el código imprescindible allá donde los controles de datos no puedan llegar. Esto es una verdad sólo a medias, ya que en el caso del botón cmdAgregar se puede configurar el control Data para que inserte un nuevo registro al llegar al final del

fichero, pero mediante el código fuente 476 en este CommandButton agregaremos el nuevo registro en cualquier momento.

```
Private Sub cmdAgregar_Click()
Me.datEnvios.Recordset.AddNew
End Sub
```

Código fuente 476

El botón cmdCancelar también necesitará código para reestablecer el valor del registro en el caso de que el usuario decida no terminar de modificarlo, dicho código se muestra en el código fuente 477.

```
Private Sub cmdCancelar_Click()
Me.datEnvios.UpdateControls
End Sub
```

Código fuente 477

Por su parte, el botón cmdEliminar ejecutará el código fuente 478 para borrar un registro y reposicionar el recordset al principio.

```
Private Sub cmdEliminar_Click()
Me.datEnvios.Recordset.Delete
Me.datEnvios.Recordset.MoveFirst
End Sub
```

Código fuente 478

Y como punto final tenemos el DBCombo cboProductos. En principio no es necesario incluir código, tal y como están configuradas sus propiedades, el traspaso de valores entre los recordsets de los controles Data funciona perfectamente, pero existe un pequeño inconveniente: cada vez que se selecciona un producto de la lista, el TextBox que contiene su valor no refleja el código de producto seleccionado. Para esta situación disponemos del evento Click() en el DBCombo, que nos permite saber cuando un usuario ha pulsado con el ratón en este control y en que área del mismo lo ha hecho. De esta forma, cuando el usuario haga clic en un elemento de la lista, traspasaremos el código del producto al TextBox utilizando la propiedad BoundText del DBCombo, que contiene el valor del campo a traspasar.

```
Private Sub cboProductos_Click(Area As Integer)
Select Case Area
Case dbcAreaButton
   ' no traspasa valores al campo
Case dbcAreaEdit
   ' no traspasa valores al campo
Case dbcAreaList
   Me.txtCodProd.Text = Me.cboProductos.BoundText
End Select
End Sub
```

Código fuente 479

La aplicación en funcionamiento

Terminado el trabajo de creación del programa, podemos ponerlo en funcionamiento para comprobar los resultados.

El lector observará que según navega por los registros del control datEnvios, el nombre del producto se va actualizando automáticamente en la parte de texto del DBCombo. De igual forma al crear o editar un registro, como hemos dicho anteriormente, cada vez que es seleccionado un nuevo valor en la lista del DBCombo, el código asociado al producto se sitúa en el TextBox establecido para dicho valor.

Como conclusión, podemos apuntar que el empleo de controles de datos y código, constituye una forma mejorada de manejar los datos que utilizando exclusivamente controles. Combinados con los objetos de acceso a datos que veremos a continuación, proporcionarán una gran potencia y versatilidad a nuestros desarrollos.

En cuando al control DBCombo, lo explicado en este apartado para dicho control es aplicable igualmente para el control DBList, con las particularidades propias de cada control, naturalmente.

Objetos de Acceso a Datos (DAO)

Visual Basic proporciona mediante DAO, una completa jerarquía de objetos que pueden ser utilizados para todos los procesos involucrados en el manejo de datos: consulta, edición de registros, borrado, creación de tablas e índices, etc.

En los próximos apartados, veremos cada uno de los objetos que forman parte de esta jerarquía, así como su forma de uso. En primer lugar, el lector ha de tener en cuenta que para poder utilizar las características de DAO, ha de abrir la ventana *Referencias* de VB (ver tema dedicado al IDE), y si no lo ha hecho ya, incluir la librería *Microsoft DAO 3.51 Object Library*, que contiene la jerarquía DAO, de lo contrario, cualquier intento de utilizar estos objetos dará error.

Guardar la ruta de datos en el registro de Windows

Una cualidad muy apreciada por los usuarios y que hará ganar muchos puntos a la aplicación, es permitir que en su proceso de instalación, el usuario pueda elegir la ruta de los archivos que componen la aplicación. Esto supone para el programador, el inconveniente de tener que desarrollar el programa, desconociendo a priori la ubicación de los ficheros de la aplicación.

En los sucesivos ejemplos se hace necesario el uso de ficheros de base de datos. Para no obligar al lector a utilizar una ruta concreta de acceso a los mismos, emplearemos el registro de configuraciones de Windows para guardar y recuperar la ruta de los ficheros que componen cada ejemplo, de manera que el lector pueda situar dichos ficheros en la ruta del disco que prefiera.

Recomendamos al lector un repaso al tema *El Registro de Windows*, donde se detallan las técnicas para obtener y grabar valores en el registro de configuraciones del sistema.

Para realizar el trabajo con el registro, emplearemos la función RutaDatosRegistro(), que busca un valor en el registro de configuraciones, correspondiente a la ruta del fichero de datos de la aplicación que se está ejecutando en ese momento, si no encuentra nada, abre la ventana frmRutaApp (figura

512), para que el usuario introduzca dicha ruta; si el valor de la ruta proporcionada es correcto, es decir, existe una base de datos, se graba en el registro.

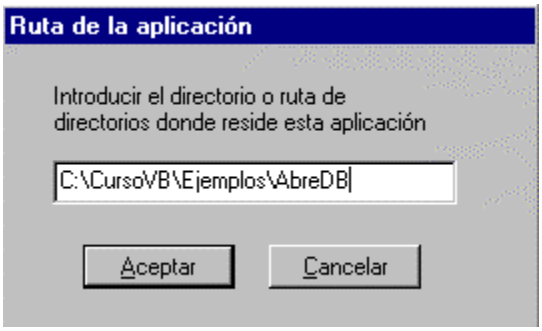


Figura 512. Ventana frmRutaApp, para incluir la ruta de los ficheros de la aplicación.

La función RutaDatosRegistro(), pertenece al módulo del formulario frmRutaApp; dicho módulo se incluye con todos los programas de ejemplo que necesiten buscar la ruta de la base de datos, y su código podemos verlo en el código fuente 480.

```
Public Function RutaDatosRegistro(rcNombreApp As String, _
   rcSeccion As String, rcClave As String, _
   rcNombreFichDatos As String, rfrmPrincip As Object) As String
Dim lcRutaDatos As String
Dim lcFichero As String
Dim lfrmRutaApp As frmRutaApp
Do While True
   ' inicializar valor de retorno
   RutaDatosRegistro = ""
   ' obtener del registro la ruta de la base de datos
   lcRutaDatos = GetSetting(rcNombreApp, rcSeccion, rcClave)
   ' si había algún valor en el registro
   ' comprobar si corresponde con una
   ' ruta válida de datos
   If Len(lcRutaDatos) > 0 Then
      lcFichero = Dir(lcRutaDatos & rcNombreFichDatos)
      If Len(lcFichero) = 0 Then
         ' no es válida la ruta del fichero
         lcRutaDatos = ""
      Else
         ' si es válida la ruta del fichero,
         ' asignarla al valor de retorno
         ' de la función y finalizar función
         RutaDatosRegistro = lcRutaDatos
         Exit Function
      End If
   End If
   ' si no hay en el registro una entrada
   ' para esta aplicación, crearla
   If Len(lcRutaDatos) = 0 Then
      ' abrir ventana para introducir ruta aplicación
      Set lfrmRutaApp = New frmRutaApp
      Load lfrmRutaApp
      lfrmRutaApp.Show vbModal, rfrmPrincip
      lcRutaDatos = lfrmRutaApp.mcRutaDatos
      Unload lfrmRutaApp
      ' si lcRutaDatos no tiene valor
      ' significa que el usuario no ha
      ' proporcionado la ruta de datos
```

```
      ' salir de la función
      If Len(lcRutaDatos) = 0 Then
         RutaDatosRegistro = ""
         Exit Function
      Else
         ' si lcRutaDatos tiene valor,
         ' crear entrada en el registro
         SaveSetting rcNombreApp, rcSeccion, rcClave, lcRutaDatos
         lcRutaDatos = ""
         ' ahora volvemos al inicio del bucle
         ' para volver a recuperar el valor
         ' del registro y comprobar si la
         ' ruta de datos es correcta
      End If
   End If
Loop
End Function
```

Código fuente 480

Apertura de una base de datos

La aplicación de ejemplo AbreDB, nos mostrará como abrir una base de datos empleando código. El punto de entrada a la aplicación será el procedimiento *Main()*; recomendamos al lector un repaso al tema *El Lenguaje*, apartado *Sub Main(). Un procedimiento especial*, sobre la forma de establecer Main() como inicio de la aplicación.

Incluir un procedimiento Main() para el programa es una buena costumbre, ya que en él podemos realizar cualquier trabajo previo a la carga del primer formulario, o seleccionar el primer formulario a mostrar, dependiendo de ciertas condiciones del programa. El lector puede consultar el fuente al completo de este procedimiento en el módulo *General*. En este apartado iremos insertando extractos de líneas de código para ver lo que se hace en cada momento durante la apertura de la base de datos.

En primer lugar cargaremos la ventana de la aplicación y comprobaremos la ruta de la base de datos, tomando del registro el valor de la clave que guarda la cadena con la ruta; para ello utilizaremos la función RutaDatosRegistro(), que si devuelve una cadena vacía, quiere decir que no existe la ruta, por lo que se cierra el formulario del programa y se cancela su ejecución.

```
Public Sub Main()
Dim lRCTClientes As Recordset
Dim lfrmAbreDB As frmAbreDB
' instanciar el formulario y mostrarlo
Set lfrmAbreDB = New frmAbreDB
Load lfrmAbreDB
lfrmAbreDB.Show
' tomar la ruta del fichero de datos del registro
gcRutaDatos = frmRutaApp.RutaDatosRegistro("AbreDB", "Valores", "RutaDatos", _
   "AbreDB.MDB", lfrmAbreDB)
If Len(gcRutaDatos) = 0 Then
   Unload lfrmAbreDB
   Exit Sub
End If
......
......
End Sub
```

Código fuente 481

Si la ruta de la base de datos es correcta, el primer objeto que hemos de utilizar es *DBEngine*, que ocupa el nivel superior en la jerarquía DAO, y representa al motor Jet; se instancia al comienzo de la aplicación sin intervención del programador y se destruye al finalizar la aplicación, contiene al resto de objetos de esta jerarquía.

Mediante DBEngine, y utilizando su método *CreateWorkspace()*, crearemos un objeto *Workspace* o *espacio de trabajo*, que consiste en una zona o sesión de trabajo, en la que el usuario puede abrir una o mas bases de datos, sus tablas, realizar transacciones y controlar el nivel de seguridad para los usuarios que accedan.

```
Public Sub Main()
......
......
' crear espacio de trabajo
Set gwsAbreDB = DBEngine.CreateWorkspace("AbreDB", "admin", "", dbUseJet)
......
......
End Sub
```

Código fuente 482

`CreateWorkspace( cNombre, cUsuario, cContraseña, xTipo )`

- cNombre. Cadena con el nombre del objeto.
- cUsuario. Cadena con el nombre de usuario.
- cContraseña. Cadena de hasta 14 caracteres con la clave de acceso al objeto.
- xTipo. Una de las siguientes constantes que definen el tipo de Workspace creado:
  - dbUseJet. Crea un espacio de trabajo, utilizando el motor Jet para tener acceso a los datos. Como se ha comentado anteriormente, se pueden usar ficheros .MDB, ficheros accesibles a través de controladores ISAM, y fuentes de datos ODBC.
  - dbUseODBC. Crea un espacio de trabajo *ODBCDirect*, que conecta directamente con una fuente de datos ODBC a través de RDO, mediante características DAO, evitando el uso del motor Jet, con lo que se obtiene un mejor acceso a fuentes ODBC. Mediante esta técnica, si no existe ningún espacio de trabajo que use Jet, este no se cargará en memoria.

Si no se especifica el parámetro *xTipo*, se tomará el valor de la propiedad *DefaultType* del objeto *DBEngine*.

En este caso vamos a utilizar el motor Jet junto a una base de datos Access, en sucesivos ejemplos usaremos *ODBCDirect* y *controladores ISAM*.. Existe una forma más rápida de acceder a un espacio de trabajo para el programa, a través de la colección *Workspaces* de *DBEngine*, tomando el objeto predeterminado de la colección como se muestra en el código fuente 483.

```
Set gwsAbreDB = DBEngine.Workspaces(0)
```

Código fuente 483

Que devolverá un espacio de trabajo listo para usar.

Las variables *gcRutaDatos*, *gwsAbreDB* y *gdbAbreDB*, que contienen la ruta de la base de datos, el espacio de trabajo y el nombre de la base de datos de la aplicación respectivamente; se han declarado a nivel de módulo como públicas, de forma que sean visibles desde cualquier parte del programa.

```
Public gcRutaDatos As String
Public gwsAbreDB As Workspace
Public gdbAbreDB As Database
```

Código fuente 484

Si se hubieran declarado locales al procedimiento Main(), una vez terminado dicho procedimiento, se perdería la referencia a los valores que guardan, por lo que la ventana que muestra los datos tendría un comportamiento extraño y se producirían errores.

Una vez creado el objeto Workspace, utilizamos su método *OpenDatabase()* para abrir la base de datos a consultar, lo que nos devuelve un objeto *Database* o *base de datos*. Veamos en el código fuente 485 la parte del fuente que realiza esta operación.

```
Public Sub Main()
......
......
' abrir base de datos
Set gdbAbreDB = gwsAbreDB.OpenDatabase(gcRutaDatos & "AbreDB.MDB")
......
......
End Sub
```

Código fuente 485

`OpenDatabase( cNombreBD, xOpciones, lSoloLectura, cConexion )`

- cNombreBD. Cadena con la ruta y nombre de un archivo de datos Microsoft Jet, o el nombre de una fuente de datos ODBC. Si este parámetro contiene una cadena vacía y cConexion contiene "ODBC", aparecerá un cuadro de diálogo para elegir una fuente de datos ODBC.

- xOpciones. Si es un espacio de trabajo Microsoft Jet, este parámetro admite los siguientes valores:

  - True. Abre la base de datos en modo exclusivo.

  - False. Abre la base de datos en modo compartido, es el valor por defecto.

  Si el espacio de trabajo es *ODBCDirect*, los valores para este parámetro son las siguientes constantes:

  - dbDriverNoPrompt. El controlador ODBC usa la cadena de conexión de los parámetros cNombreBD y cConexion.

  - dbDriverPrompt. Aparece un cuadro de diálogo con los orígenes de datos ODBC disponibles para que el usuario seleccione uno.

  - dbDriverComplete. Si los parámetros cNombreBD y cConexion contienen información válida, se usa la cadena de cConexion. En caso contrario, actuará como si se hubiera pasado el parámetro dbDriverPrompt. Este es el parámetro por defecto en caso de que no se pase ninguno en Opciones.

  - dbDriverCompleteRequired. Es igual que dbDriverComplete, con la diferencia de que el controlador ODBC inhabilita el envío de información innecesaria para la conexión.

- lSoloLectura. Valor lógico, si es verdadero, la base de datos se abrirá como sólo lectura, si es falso lo hará en lectura/escritura. Por defecto se abre en este último modo.

- cConexion. Cadena que incluye los argumentos para establecer una conexión mediante ODBC. Si el espacio de trabajo es ODBCDirect y cConexion tiene valor, en cNombreBD se puede poner una cadena que identifique la base de datos posteriormente en el programa.

  El formato de este parámetro es: *"TipoBD; RestoInformaciónConexión"*.

A partir de aquí, una vez abierta la base de datos, ya podemos comenzar a manipular la información de las tablas.

El objeto Recordset

Abierta la base de datos y creado el objeto Database, usamos su método *Open*Recordset*()* para crear un objeto *Recordset* o *conjunto de registros*, que representa una tabla de la base de datos o una consulta SQL contra una o varias tablas. Continuando con el procedimiento Main() del apartado anterior, creamos el objeto Recordset para acceder a la tabla Clientes.

```
Public Sub Main()
......
......
' abrir Recordset
Set lRCTClientes = gdbAbreDB.OpenRecordset("Clientes", dbOpenTable)
......
......
End Sub
```

Código fuente 486

`OpenRecordset( cOrigen, xTipo, xOpciones, xTipoBloqueo )`

- cOrigen. Cadena que indica el origen de los registros. Puede ser el nombre de una tabla o una sentencia SQL.

- xTipo. Constante que define el Recordset, existiendo los siguientes tipos:

  - dbOpenTable. Es la forma más directa de acceso a los registros, ya que es la representación física de una tabla de la base de datos. Permite utilizar índices para acceder

y ordenar los registros y cualquier modificación que otro usuario efectúe, se refleja de forma inmediata sin tener que crear de nuevo el objeto.

La búsqueda de registros individuales mediante el comando *Seek* es más rápida con este tipo de Recordset. Por contra no se pueden especificar filtros para crear un rango de registros, y si queremos buscar varios registros con la misma clave hay que hacerlo mediante código.

Sólo está disponible para espacios de trabajo Microsoft Jet.

- dbOpenDynaset. Contiene un conjunto de registros de una o más tablas, pueden estar filtrados y ordenados según una determinada condición. Se pueden modificar los registros, pero si otro usuario de la aplicación elimina o añade algún registro, no será visible a menos que generemos de nuevo el Recordset o llamemos a su método *Requery*, que vuelve a ejecutar la consulta en la que se basa el objeto.

  La parte principal en la creación de este tipo de objetos, consiste en una cadena que contiene una instrucción SQL, que es la encargada de recuperar los registros de una o varias tablas, filtrarlos y ordenarlos. En el caso de usar la sentencia SQL, el segundo parámetro del método no sería obligatorio, creándose el objeto automáticamente de tipo *Dynaset.*

- dbOpenSnapshot. Es muy similar a una Dynaset; se crean y manipulan de la misma forma que estas últimas, residiendo su mayor diferencia en que los registros del objeto no se pueden actualizar. Esta particularidad es al mismo tiempo su mayor ventaja, ya que al ser de sólo lectura, se crea más rápidamente y es idóneo para procesos de consulta o listados.

- dbOpenForwardOnly. Es igual a una Snapshot pero más restringida aún, ya que sólo permite avance de registros.

- dbOpenDynamic. Conjunto de registros de una o más tablas al que se pueden añadir, modificar o borrar. Los cambios hechos por otros usuarios serán visibles en este objeto.

  Sólo está disponible para espacios de trabajo *ODBCDirect*.

- xOpciones. Combinación de una o varias constantes que definen las características del Recordset.

  - dbDenyWrite. Impide al resto de usuarios añadir o modificar registros.

  - dbDenyRead. Impide al resto de usuarios ver los registros.

  - dbReadOnly. Crea el Recordset como sólo lectura. Se proporciona para compatibilidad con las versiones anteriores, debiendo usar la constante del mismo nombre en el parámetro *TipoBloqueo*.

  - dbAppendOnly. Sólo permite añadir registros.

  - dbInconsistent. Si tomamos dos tablas relacionadas por un campo, el uso de este parámetro permite actualizar los registros de una tabla en los que el campo relacionado no coincida con los registros de la tabla relacionada.

- dbConsistent. Este caso es justo al contrario del anterior, sólo se permite actualizar los registros de una tabla relacionada con otra, cuando dichos registros cumplan las reglas de integridad establecidas con la tabla relacionada.

- dbForwardOnly. El objeto Recordset sólo permite desplazamiento hacia adelante. Se proporciona para compatibilidad con versiones anteriores, debiendo usar la constante *dbOpenForwardOnly* en el parámetro *Tipo*.

- dbRunAsync. Ejecuta una consulta asíncrona contra la base de datos. Este tipo de consulta devuelve inmediatamente el control una vez lanzada, terminando de procesarse en segundo plano, de forma que la aplicación puede seguir con su ejecución.

- dbExecDirect. Envía la consulta al servidor llamando al método *SQLExecDirect*, evitando el método *SQLPrepare*. Esta opción no se puede utilizar si el Recordset se crea con una consulta de parámetros. Sólo está disponible en espacios de trabajo *ODBCDirect*.

- dbSQLPassTrough. La consulta se realiza contra un servidor de ODBC.

- dbSeeChanges. Si otro usuario modifica los datos que estamos editando se genera un error.

- xTipoBloqueo. Constante que indica el bloqueo para el Recordset.

  - dbReadOnly. El Recordset no admite modificaciones en los registros. Valor por defecto en espacios de trabajo *ODBCDirect*.

  - dbPessimistic. Utiliza el bloqueo pesimista en la edición de registros del objeto. En este bloqueo, la página que contiene el registro a editar estará bloqueada mientras se usa el método *Edit*, y se liberará al llamar a *Update*. Valor por defecto en espacios de trabajo Microsoft Jet.

  - dbOptimistic. Utiliza el bloqueo optimista en la edición de registros. En este bloqueo, la página que contiene el registro a editar, estará bloqueada durante la ejecución del método *Update*.

  - dbOptimisticValue. Usa la concurrencia optimista basándose en valores de fila. Disponible sólo en espacios de trabajo *ODBCDirect*.

  - dbOptimisticBatch. Activa la actualización optimista por lotes. En este tipo de objeto, las actualizaciones se almacenan en un buffer, y son enviadas todas juntas al servidor. Disponible sólo en espacios de trabajo *ODBCDirect*.

Finalizamos el procedimiento Main(), asignando el Recordset recién creado a la propiedad del mismo nombre del control Data del formulario, para poder ver su contenido en el DBGrid enlazado.

```
Public Sub Main()
......
' asignar recordset al control data del formulario
Set lfrmAbreDB.datClientes.Recordset = lRCTClientes
......
End Sub
```

Código fuente 487

La ventana al mostrarse, tendrá el aspecto de la figura 513.

Figura 513. Resultado de la apertura de una base de datos usando código.

Conectando mediante ODBCDirect

Como hemos comentado en el apartado anterior, mediante *ODBCDirect* podemos acceder a fuentes de datos ODBC mediante RDO, utilizando características DAO que evitan cargar el motor Jet. En la aplicación AbreODBC, conectaremos con una base de datos mediante ODBCDirect, y veremos los registros de la tabla que contiene.

El modo de iniciar el programa es el mismo del apartado anterior, usando un procedimiento Main(), y para mostrar el resultado de la conexión, también empleamos un formulario con un Data y un DBGrid. Las diferencias las encontramos al crear los objetos Workspace, Database y Recordset, utilizando valores diferentes en los métodos de creación a los empleados en el ejemplo anterior.

En el código fuente 488 observamos el contenido de Main().

```
Public Sub Main()
Dim lRCDClientes As Recordset
Dim lfrmAbreODBC As frmAbreODBC
Dim lcSQL As String
' instanciar el formulario y mostrarlo
Set lfrmAbreODBC = New frmAbreODBC
Load lfrmAbreODBC
lfrmAbreODBC.Show
' tomar la ruta del fichero de datos del registro
gcRutaDatos = frmRutaApp.RutaDatosRegistro("AbreODBC", "Valores", _
   "RutaDatos", "AbreODBC.MDB", lfrmAbreODBC)
' si no hay una ruta válida de datos
' cancelar la ejecución
If Len(gcRutaDatos) = 0 Then
   Unload lfrmAbreODBC
   Exit Sub
End If
' crear espacio de trabajo
Set gwsAbreODBC = DBEngine.CreateWorkspace("AbreODBC", "admin", "", dbUseODBC)
' abrir la base de datos empleando una cadena
' con información para realizar una conexión ODBC
Set gdbAbreODBC = gwsAbreODBC.OpenDatabase("", _
```

```
   dbDriverNoPrompt, , _
   "ODBC;DBQ=" & gcRutaDatos & "ABREODBC.MDB;DefaultDir=" & _
   gcRutaDatos & ";" & _
   "Driver={Microsoft Access Driver (*.mdb)};DriverId=25;" & _
   "FIL=MS Access;ImplicitCommitSync=Yes;MaxBufferSize=512;" & _
   "MaxScanRows=8;PageTimeout=5;SafeTransactions=0;Threads=3;" & _
   "UID=admin;UserCommitSync=Yes;")
' abrir un Recordset de tipo Dynamic,
' utilizamos una variable para contener la sentencia SQL,
' sólo a efectos de mayor claridad y organización del código
lcSQL = "SELECT * FROM Clientes"
Set lRCDClientes = gdbAbreODBC.OpenRecordset(lcSQL, dbOpenDynamic)
' asignar Recordset al control data del formulario
Set lfrmAbreODBC.datClientes.Recordset = lRCDClientes
End Sub
```

Código fuente 488

El resultado obtenido será similar a la figura 514.

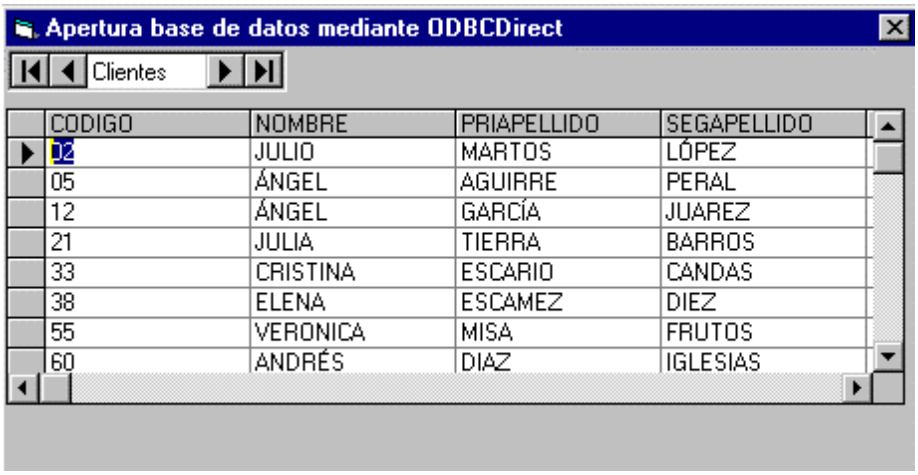


Figura 514. Conexión a base de datos utilizando ODBCDirect.

Ya que la fuente de datos a conectar es de tipo Access, este modo de hacer las cosas es un tanto artificioso, pero suficiente para demostrar una conexión efectuada mediante ODBCDirect.

Uso de controladores ISAM

Un controlador *ISAM (Método de Acceso Secuencial Indexado)*, permite acceder a formatos de bases de datos no nativos del motor Jet, tales como dBASE, Paradox, etc. Cuando se les hace referencia en el código de una aplicación, el motor Jet carga el controlador adecuado, pudiendo manejar los datos mediante objetos DAO, de la forma a la que estamos acostumbrados para los ficheros MDB. La situación de estos controladores se mantiene en el registro de Windows.

En la aplicación AbreISAM, abriremos el fichero Elecdom.DBF y lo mostraremos en un formulario, al igual que en los anteriores ejemplos.

La diferencia con los otros ejemplos reside en la línea de código del procedimiento Main() que abre la base de datos, y que podemos ver en el código fuente 489.

```
' abrir base de datos,
' el último parámetro contiene el
' nombre del controlador ISAM
Set gdbAbreISAM = gwsAbreISAM.OpenDatabase( _
Left(gcRutaDatos, (Len(gcRutaDatos) - 1)), False, False, "dBASE III;")
```

Código fuente 489

En esta línea, se elimina mediante las funciones Left() y Len() el separador de directorio "\" añadido por la función RutaDatosRegistro() a la variable gcRutaDatos. Esto es debido a que en este caso, el primer parámetro de OpenDatabase() tiene un tratamiento especial, ya que se incluye sólo la ruta que contiene los ficheros DBF, de forma que dicho directorio actúa como si fuera la base de datos, conteniendo los ficheros que representan a las tablas.

Al crear el Recordset, usaremos el nombre del fichero DBF como si fuera una de las tablas de la base de datos, pudiendo abrir el fichero en modo tabla, o utilizando una sentencia SQL como se muestra en el código fuente 490.

```
' abrir Recordset
lcSQL = "SELECT * FROM Elecdom WHERE CODIGO >= '25' ORDER BY CODIGO"
Set lRCDElecdom = gdbAbreDB.OpenRecordset(lcSQL)
```

Código fuente 490

El resultado lo podemos ver en la figura 515.

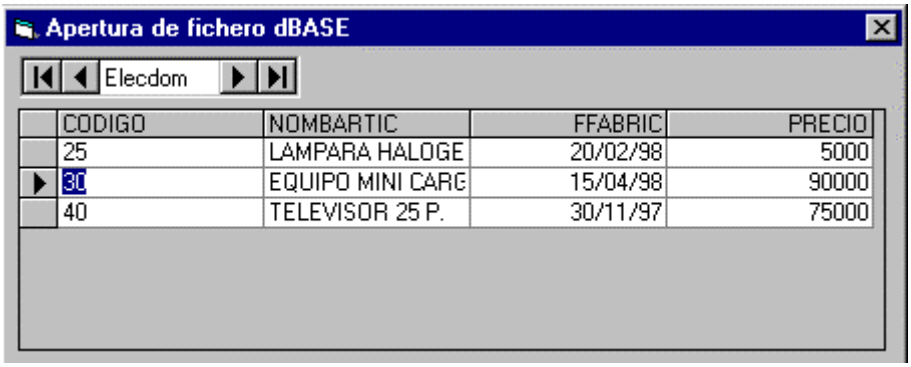


Figura 515. Apertura de un fichero DBF utilizando SQL.

Otras operaciones con DAO

En los últimos apartados, hemos visto los objetos DAO más importantes y fundamentales, los que nos permiten abrir una base de datos y acceder a su contenido. Pero el trabajo con datos no se limita sólo a consultar y agregar información a las tablas; existe un variado conjunto de acciones y elementos referentes a las operaciones con bases de datos, que veremos en los sucesivos apartados.

Todos los ejemplos concernientes a estos elementos se van a integrar en un único proyecto llamado VarioDAO. El procedimiento Main() de esta aplicación se encargará de abrir la base de datos VarioDAO.MDB a escala global, de igual manera que en los anteriores apartados, de forma que no sea necesario abrirla para cada uno de los ejemplos, los cuales serán accesibles a través de las opciones del menú incluido en la ventana MDI de esta aplicación.

```
Public Sub Main()
Dim lfrmPrincipal As frmPrincipal
' instanciar el formulario
Set lfrmPrincipal = New frmPrincipal
Load lfrmPrincipal
lfrmPrincipal.Show
' tomar la ruta del fichero de datos del registro
gcRutaDatos = frmRutaApp.RutaDatosRegistro("VarioDAO", "Valores",  _ "RutaDatos",
"VarioDAO.MDB", lfrmPrincipal)
' si no hay entradas en el registro
' con la ruta del fichero, salir
If Len(gcRutaDatos) = 0 Then
   Unload lfrmPrincipal
   Exit Sub
End If
' crear espacio de trabajo
Set gwsVarioDAO = DBEngine.CreateWorkspace("VarioDAO", "admin", "", _
   dbUseJet)
' abrir base de datos
Set gdbVarioDAO = gwsVarioDAO.OpenDatabase(gcRutaDatos & "VarioDAO.MDB")

End Sub
```

Código fuente 491

Índices

Como vimos en el diseño de bases de datos, un índice ordena una tabla basándose en un criterio determinado. Para poder utilizar índices, debemos crear un Recordset tipo tabla, ya que es el único que soporta índices, e indicar en su propiedad Index el nombre del índice a utilizar

En la aplicación VarioDAO, elegiremos la opción *Índices* del menú, que cargará la ventana *frmIndices*. En el método Load() de esta ventana, se creará un Recordset tipo tabla y activará uno de sus índices, para después asignar el Recordset al control Data del formulario. El código fuente 492 nos muestra el código.

```
Private Sub Form_Load()
Dim lRSTClientes As Recordset
' posicionar el formulario dentro
' de la ventana principal
Me.Top = 15
Me.Left = 45
Set lRSTClientes = gdbVarioDAO.OpenRecordset("Clientes", dbOpenTable)
lRSTClientes.Index = "PrimaryKey"
Set Me.datClientes.Recordset = lRSTClientes
End Sub
```

Código fuente 492

La ventana a visualizar será similar a la figura 516.

Figura 516. Ventana de pruebas con índices.

El lector se preguntará el motivo de usar las propiedades *Top* y *Left* del formulario para situarlo dentro de la ventana MDI *frmPrincipal*. Si no las usamos, la ventana MDI situará cada nueva ventana secundaria de forma automática un poco más abajo y a la derecha de la anteriormente visualizada, en efecto de cascada. Esta organización es útil cuando la aplicación muestra varias ventanas secundarias simultáneamente, pero en nuestro caso, sólo queremos mostrar una ventana secundaria a la vez, y si dejamos que la ventana MDI las coloque, dará un efecto poco estético al programa, al ir apareciendo cada vez un poco más en el extremo inferior derecho.

Volviendo a los índices, que es el tema que nos ocupa, para cambiar uno de ellos, sólo tenemos que pulsar en alguno de los botones de opción de la ventana, que contienen el código fuente 493.

```
Private Sub optPrimary_Click()
Me.datClientes.Recordset.Index = "PrimaryKey"
End Sub
' ----------------------------------------

Private Sub optNombre_Click()
Me.datClientes.Recordset.Index = "Nombre"
End Sub
' ----------------------------------------

Private Sub optPoblacion_Click()
Me.datClientes.Recordset.Index = "Poblacion"
End Sub
```

Código fuente 493

En el momento en que asignemos un nuevo índice al Recordset, el criterio de ordenación del índice se reflejará inmediatamente en las filas del formulario.

El uso de índices para las tablas nos permite buscar rápidamente los registros en función del índice activo; igualmente, los nuevos registros serán situados en su posición correspondiente dentro del orden del índice, de forma que siempre mantendremos ordenada la tabla. Como inconveniente podemos

apuntar el hecho de que definir muchos índices para una tabla puede perjudicar el rendimiento de la misma, ya que cada nueva fila que agreguemos o modifiquemos ha de ser actualizada en cada índice, lo que puede suponer un retraso si disponemos de demasiados índices. Es conveniente buscar un equilibrio, y analizar que información necesitaremos consultar más a menudo, para definir sólo los índices realmente necesarios, ya que sería absurdo tener por ejemplo un índice por el que realizáramos consultas una vez al mes; en este caso sería más recomendable realizar una consulta usando SQL e indicando los valores a buscar, como veremos en uno de los siguientes apartados.

Órdenes

Para ordenar los registros, debemos utilizar un Recordset de tipo Dynaset o Snapshot, disponiendo de dos modos de realizar la ordenación.

Por un lado, si creamos el Recordset utilizando una sentencia SQL, podemos especificar el criterio de ordenación en dicha sentencia mediante la cláusula *ORDER BY*. En la aplicación de ejemplo, seleccionamos la opción de menú *Órdenes + Sentencia SQL*, que abre el formulario *frmOrdenes* y llama a su procedimiento *OrdenSQL()*, en el cual se crea un Recordset utilizando una consulta SQL.

```
Public Sub OrdenSQL()
Dim lRCDClientes As Recordset
Dim lcSQL As String
' abrir un Recordset
Me.Caption = "Registros ordenados mediante sentencia SQL"
lcSQL = "SELECT * FROM Clientes ORDER BY Nombre"
Set lRCDClientes = gdbVarioDAO.OpenRecordset(lcSQL, dbOpenDynaset)
Set Me.datClientes.Recordset = lRCDClientes
End Sub
```

Código fuente 494

Al mostrarse el formulario, los registros ya aparecerán ordenados.

La otra forma de ordenar se basa en la propiedad *Sort* del objeto Recordset. No es una manera tan directa, pero puede sernos útil en ciertos casos. Seleccionamos la opción *Órdenes + Propiedad Sort* del menú principal, que carga la ventana frmOrdenes y llama a su procedimiento *OrdenSort()*.

```
Public Sub OrdenSort()
Dim lRCDClientes As Recordset
Dim lcSQL As String
' abrir un Recordset
Me.Caption = "Registros ordenados mediante propiedad Sort"
lcSQL = "SELECT * FROM Clientes"
Set lRCDClientes = gdbVarioDAO.OpenRecordset(lcSQL, dbOpenDynaset)
Set Me.datClientes.Recordset = lRCDClientes
Me.cmdOrdenSort.Visible = True
End Sub
```

Código fuente 495

Este procedimiento crea un Recordset con todos los registros de la tabla Clientes y visualiza el botón *cmdOrdenSort*, que permanecía oculto. Es al pulsar este botón, en su método *Click()*, cuando establecemos el nuevo orden en la propiedad Sort del Recordset. Pero con esto no obtenemos los registros ordenados; para conseguirlo, hemos de abrir un segundo Recordset a partir del existente, usando el método OpenRecordset(). Es en este último, donde ya obtenemos los registros ordenados.

```
Private Sub cmdOrdenSort_Click()
Dim lRCDClientes As Recordset
Me.datClientes.Recordset.Sort = "NIF"
Set lRCDClientes = Me.datClientes.Recordset.OpenRecordset
Set Me.datClientes.Recordset = lRCDClientes
End Sub
```

Código fuente 496

La propiedad Sort se define mediante una cadena que contiene una cláusula *ORDER BY* de una consulta SQL, sin la palabra clave *ORDER BY*.

Con Sort podemos ordenar por cualquier campo de la tabla, sin necesidad de que forme parte de ningún índice.

Figura 517. Columna NIF ordenada mediante la propiedad Sort.

Ambas formas de trabajo son válidas, siendo la primera mucho más directa, ya que no necesita crear un segundo Recordset, aunque dependiendo de la aplicación, puede haber situaciones en que la última forma sea preferible.

Es necesario indicar que la apertura de un nuevo Recordset a partir de uno existente, no se puede realizar si el primero es de tipo *ForwardOnly*, o el espacio de trabajo es *ODBCDirect*.

Como inconveniente con el uso de órdenes, tenemos el hecho de que una vez ordenados los registros, si añadimos algún registro nuevo a la tabla, se situará al final, sin tener en cuenta el orden establecido anteriormente, por lo que deberemos ordenar nuevamente los registros. Si queremos que los nuevos registros se vean inmediatamente ordenados, es más recomendable utilizar índices.

Filtros

Podemos establecer un filtro usando un Recordset de tipo Dynaset o Snapshot, mediante dos sencillas maneras.

Una de ellas es definir el filtro al generar el Recordset con una consulta SQL, mediante la cláusula *WHERE*.

Seleccionando en la aplicación, la opción de menú *Filtros + Sentencia SQL*, se carga la ventana *frmFiltros*, y se llama a su procedimiento *FiltroSQL()*.

```
Public Sub FiltroSQL()
Dim lRCDClientes As Recordset
Dim lcSQL As String
' abrir un Recordset

Me.Caption = "Registros filtrados mediante sentencia SQL"
lcSQL = "SELECT * FROM Clientes WHERE CodCli > '20' AND CodCli < '50'"
Set lRCDClientes = gdbVarioDAO.OpenRecordset(lcSQL, dbOpenDynaset)
Set Me.datClientes.Recordset = lRCDClientes
End Sub
```

Código fuente 497

Los registros aparecerán ya filtrados al mostrar la ventana.

La otra forma de filtrar registros, consiste en fijar la condición a través de la propiedad *Filter* del Recordset. En este caso elegimos la opción *Filtros + Propiedad Filter* del menú principal, que carga la ventana frmFiltros y llama a su procedimiento *FiltroFilter()*.

```
Public Sub FiltroFilter()
Dim lRCDClientes As Recordset
Dim lcSQL As String
' abrir un Recordset

Me.Caption = "Registros filtrados mediante propiedad Filter"
lcSQL = "SELECT * FROM Clientes"
Set lRCDClientes = gdbVarioDAO.OpenRecordset(lcSQL, dbOpenDynaset)
Set Me.datClientes.Recordset = lRCDClientes
Me.cmdFiltroFilter.Visible = True
End Sub
```

Código fuente 498

En este procedimiento creamos un Recordset con todos los registros de la tabla Clientes, y visualizamos el botón *cmdFiltroFilter*, que permanecía oculto; al pulsar este botón será cuando se llame a su método Click() que establecerá el filtro.

```
Private Sub cmdFiltroFilter_Click()
Dim lRCDClientes As Recordset
Me.datClientes.Recordset.Filter = "CodCli > '20' AND CodCli < '50'"
```

```
Set lRCDClientes = Me.datClientes.Recordset.OpenRecordset
Set Me.datClientes.Recordset = lRCDClientes
End Sub
```

Código fuente 499

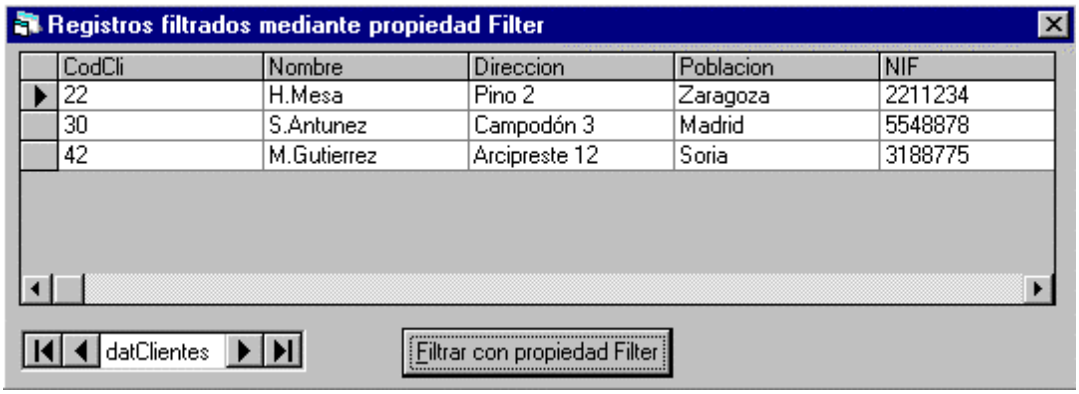


Figura 518. Registros filtrados mediante la propiedad Filter del Recordset.

En este procedimiento insertamos una condición de filtrado en la propiedad Filter del Recordset, pero como sucedía con la propiedad Sort, todavía no es efectivo. Hemos de abrir un segundo Recordset a partir del existente para que sólo aparezcan los registros deseados.

La propiedad Filter se establece mediante una cadena que contiene una cláusula *WHERE* de una consulta SQL, sin la palabra clave *WHERE*.

Aquí es igualmente recomendable por cuestiones de rapidez, crear el Recordset con una consulta SQL, pero como decíamos anteriormente, todo depende de la aplicación, y pudiera haber circunstancias en que el uso de Filter fuera preferible.

Navegación por los registros

Para desplazarnos entre los registros de un Recordset, disponemos de sus métodos *Move*, que se detallan a continuación.

- MoveFirst. Se desplaza hasta el primer registro del objeto Recordset. Hay que tener en cuenta que si el Recordset es de tipo Dynaset o Snapshot, no quiere decir que el movimiento sea al primer registro físico de una tabla, ya que el Recordset puede haberse creado con una consulta SQL filtrada hacia unos pocos registros de la tabla.

- MoveLast. Se sitúa en el último registro del Recordset.

- MoveNext. Se sitúa en el siguiente registro del Recordset.

- MovePrevious. Se sitúa en el anterior registro del Recordset.

- Move lRegistros, vntMarcadorInicio. Avanza o retrocede el número de registros pasado como parámetro en función de que *lRegistros* sea un número positivo o negativo. Si hay definido un *Bookmark o marcador* para el Recordset y se pasa como segundo parámetro, el desplazamiento de registros comenzará a partir del registro indicado por el Bookmark.

Las propiedades *BOF* y *EOF* del Recordset, devuelven un valor lógico que indican cuando se ha llegado al principio o final de los registros, respectivamente.

Mediante la opción de menú Navegación, incluida en la aplicación de ejemplo, abriremos la ventana *frmNavegacion*, que nos mostrará de forma práctica todo este movimiento de registros en el interior del Recordset.

```
Private Sub mnuNavegacion_Click()
Dim lfrmNavegacion As frmNavegacion
Dim lRCTClientes As Recordset
Set lRCTClientes = gdbVarioDAO.OpenRecordset("Clientes", dbOpenTable)
lRCTClientes.Index = "PrimaryKey"
Set lfrmNavegacion = New frmNavegacion
Load lfrmNavegacion
Set lfrmNavegacion.datClientes.Recordset = lRCTClientes
lRCTClientes.MoveLast
lRCTClientes.MoveFirst
lfrmNavegacion.lblNroRegistros.Caption = lRCTClientes.RecordCount
lfrmNavegacion.Show
End Sub
```

Código fuente 500

Figura 519. Ventana de desplazamiento entre los registros de un Recordset.

La mayoría de las operaciones realizadas por los botones son auto explicativas. Podríamos destacar el control Label *lblNroRegistros*, que indica el número de registros de la tabla, usando la propiedad *RecordCount* del Recordset en el procedimiento anterior.

La propiedad *RecordCount* devuelve el número de registros que contiene el Recordset. Para asegurarnos que es un valor fiable, nada más crear el Recordset, es conveniente llamar a *MoveLast*, con lo cual se fuerza un recuento de registros y actualización de *RecordCount*. Después llamamos a *MoveFirst* con lo que volvemos de nuevo al registro inicial.

También es importante resaltar que al avanzar o retroceder filas en el Recordset, si llegamos al principio o final del mismo (propiedades BOF y EOF a True respectivamente), estaremos en una zona del objeto en la que no existe registro, por lo tanto hemos de reposicionar el puntero del Recordset en el primer o último registro, en función de donde nos encontremos.

```
Private Sub cmdAnterior_Click()
' retrocede un registro
Me.datClientes.Recordset.MovePrevious
' si nos encontramos en el principio del
' recordset, reposicionar al primer registro
If Me.datClientes.Recordset.BOF Then
   Me.datClientes.Recordset.MoveNext
   MsgBox "Primer registro", , "Movimiento de registros"
End If
End Sub
' ---------------------
Private Sub cmdSiguiente_Click()
' avanza al siguiente registro de la tabla
Me.datClientes.Recordset.MoveNext
' si nos encontramos en el final del
' recordset, reposicionar al último registro
If Me.datClientes.Recordset.EOF Then
   Me.datClientes.Recordset.MovePrevious
   MsgBox "Último registro", , "Movimiento de registros"
End If
End Sub
Private Sub cmdMarcar_Click()
' captura el valor del marcador de registros
' para poder volver a este registro posteriormente
mvntMarcador = Me.datClientes.Recordset.Bookmark
End Sub
' ------------------------------------------------
Private Sub cmdVolver_Click()
' vuelve al registro marcado cuyo valor
' está en la variable mvntMarcador
Me.datClientes.Recordset.Bookmark = mvntMarcador
End Sub
```

Código fuente 501

Si en algún momento necesitamos guardar la posición de un registro del Recordset para después regresar a él, disponemos de la propiedad *Bookmark*, que devuelve un valor que identifica de forma única a un registro. Los botones *Marcar Registro* y *Volver a la marca* muestran como hacerlo.

```
Private Sub cmdMarcar_Click()
' captura el valor del marcador de registros
' para poder volver a este registro posteriormente
mvntMarcador = Me.datClientes.Recordset.Bookmark
End Sub
' ------------------------------------------------
Private Sub cmdVolver_Click()
' vuelve al registro marcado cuyo valor
' está en la variable mvntMarcador
Me.datClientes.Recordset.Bookmark = mvntMarcador
End Sub
```

Código fuente 502

El *Bookmark* se guarda en la variable *mvntMarcador*, definida a nivel de módulo para que su valor sea visible desde todos los procedimientos de este formulario.

En el caso de que la aplicación use una base de datos distinta de Access, hay que tener la precaución de comprobar la propiedad *Bookmarkable* del Recordset creado, para ver si soporta esta característica.

Como curiosidad sobre el método *Move*, al introducir un valor en el TextBox *txtMoverReg* que hay dentro del *Frame Mover Registros* y pulsar Enter, el puntero se desplazará el número de registros indicado. Pero si además marcamos el CheckBox *Usando marcador*, el desplazamiento de registros se realizará a partir del registro indicado por el marcador que se haya definido al pulsar *Marcar Registro.*

Búsqueda de registros

Las búsquedas de registros vendrán determinadas por el tipo de Recordset que utilicemos. Si es de tipo *Table*, usaremos su método *Seek*, que tiene la siguiente sintaxis.

```
Seek cComparación, xClave1, xClave2...xClave13
```

- cComparación. Una cadena que contiene un operador de comparación: <, <=, =, >=, >.
- xClave1, xClave2...xClave13. Uno o más valores correspondientes a los campos que componen la clave del índice. Se pueden incluir hasta 13 claves.

Antes de realizar una búsqueda con Seek, hemos de establecer el índice por el que vamos a realizar la búsqueda mediante la propiedad *Index* del Recordset.

Cuando los operadores de comparación sean *igual*, *mayor o igual* o *mayor que*, la búsqueda comenzará desde el principio del índice hacia adelante. Si los operadores son *menor que* o *menor o igual que*, la búsqueda comenzará desde el final del índice hacia atrás.

Si tenemos un índice cuya clave está compuesta por más de un campo, al realizar una búsqueda de igualdad "=", hemos de especificar en Seek todos los valores de la clave, de lo contrario no se localizará el registro deseado. Si no especificamos todos los valores de la clave, es más conveniente realizar una búsqueda por mayor que o igual ">=".

Para ver ejemplos de búsqueda en una tabla, seleccionaremos la opción de menú *Búsquedas + Tabla*, en la aplicación VarioDAO, que abrirá la ventana *frmBusquedas* y crea un Recordset de tipo Table llamando a su procedimiento *BuscarTabla()*.

```
Private Sub mnuBuTabla_Click()
Dim lfrmBusquedas As frmBusquedas
Set lfrmBusquedas = New frmBusquedas
Load lfrmBusquedas
lfrmBusquedas.BuscarTabla
lfrmBusquedas.Show
End Sub
' ------------------------------------
Public Sub BuscarTabla()
Dim lRCTClientes As Recordset
mcTipoRecordset = "TABLE"
' buscar registros en un recordset tipo tabla
Me.Caption = "Búsquedas en Recordset tipo Tabla"
Set lRCTClientes = gdbVarioDAO.OpenRecordset("Clientes", dbOpenTable)
lRCTClientes.Index = "PrimaryKey"
Set Me.datClientes.Recordset = lRCTClientes
fraSeleccionar.Visible = True
End Sub
```

Código fuente 503

Figura 520. Ventana de búsquedas utilizando un Recordset Table.

En esta ventana, podemos seleccionar uno de los índices de la tabla pulsando un *OptionButton* del Frame *Seleccionar índice*. Establecer el tipo de comparación para la búsqueda (> mayor que, = igual que, <= menor igual que, etc.) en el Combo *cboComparacion* e introducir un valor a buscar en el TextBox *txtValorBuscar*. Al terminar de editar este control, pulsando Enter se efectuará la búsqueda en su método *KeyPress*.

```
Private Sub txtValorBuscar_KeyPress(KeyAscii As Integer)
Dim lcCadenaBuscar As String
If KeyAscii = vbKeyReturn Then
   Select Case mcTipoRecordset
   Case "DYNASET"
..............
..............
   Case "TABLE"
      ' buscar valor en un recordset table
      Me.datClientes.Recordset.Seek Me.cboComparacion.Text, Me.txtValorBuscar.Text

   End Select

   If Me.datClientes.Recordset.NoMatch Then
      MsgBox "Registro no encontrado", , "Resultado de la búsqueda"
   Else
      MsgBox "Registro encontrado", , "Resultado de la búsqueda"
   End If

End If
End Sub
```

Código fuente 504

El método *Seek* busca el primer registro que cumple la condición. Para saber si ha tenido éxito, debemos preguntar el valor de la propiedad *NoMatch*, que devuelve falso cuando se ha encontrado, y verdadero cuando no.

Para realizar búsquedas mediante un Recordset de tipo *Dynaset*, debemos emplear sus métodos *Find*. A estos métodos se les pasa como parámetro una cadena que contiene una cláusula *WHERE* de una consulta SQL, pero sin la palabra *WHERE*. Los métodos Find disponibles son los siguientes.

- FindFirst. Busca desde el principio del Recordset, el primer registro que coincida con la clave de búsqueda.
- FindLast. Busca desde el final del Recordset, el primer registro que coincida con la clave de búsqueda.
- FindNext. Una vez localizado un registro, este método busca el siguiente que coincida con la clave.
- FindPrevious. Una vez localizado un registro, este método busca el anterior que coincida con la clave.

Una vez empleado un método Find, hemos de usar la propiedad *NoMatch*, para averiguar si la búsqueda se realizó con éxito.

Para ver los ejemplos correspondientes a estos métodos, seleccionaremos la opción de menú *Búsquedas + Dynaset*, que vuelve a lanzar la ventana anterior, pero ocultando algunos controles y mostrando otros que en el anterior ejemplo no se habían visualizado.

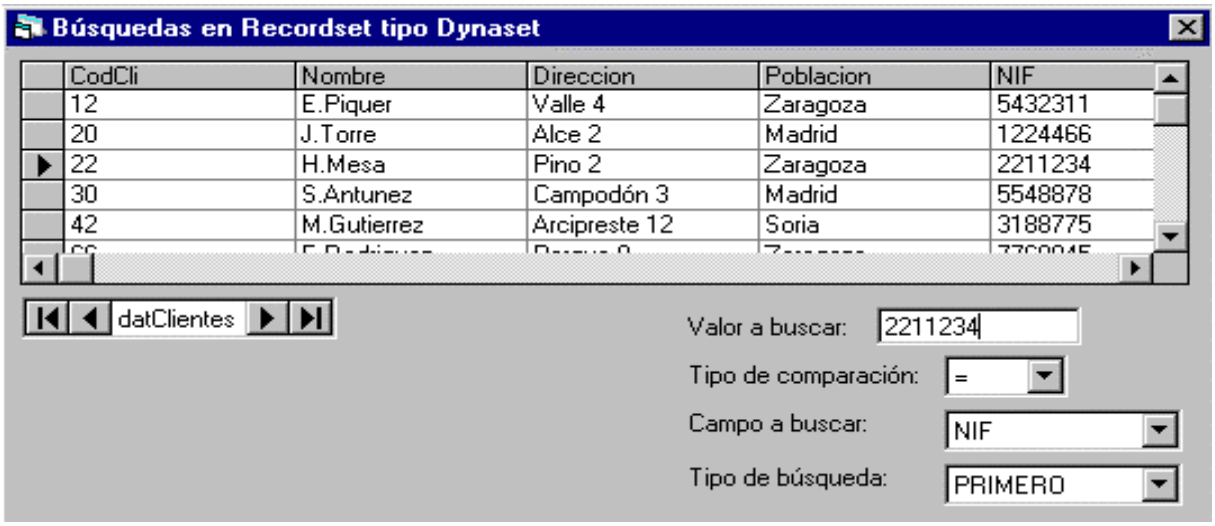


Figura 521. Búsquedas en un Recordset de tipo Dynaset.

Como un *Dynaset* no admite índices, no aparece el *Frame* con los índices disponibles. Tenemos sin embargo, dos Combos nuevos, *cboCampo*, que contiene los campos en los que se pueden hacer búsquedas, y *cboTipoBusqueda*, para buscar el primer registro, último, siguiente y anterior con los métodos *Find*. Volvemos al evento *KeyPress()*, pero esta vez en la parte referente a búsquedas con *Dynaset*.

```
Private Sub txtValorBuscar_KeyPress(KeyAscii As Integer)
Dim lcCadenaBuscar As String
If KeyAscii = vbKeyReturn Then
   Select Case mcTipoRecordset
```

```
   Case "DYNASET"
      ' buscar valor en un recordset dynaset
      lcCadenaBuscar = Me.cboCampo.Text & Me.cboComparacion.Text & "'" & _
         Me.txtValorBuscar.Text & "'"

      Select Case Me.cboTipoBusqueda.Text
      Case "PRIMERO"
         Me.datClientes.Recordset.FindFirst lcCadenaBuscar
      Case "ULTIMO"
         Me.datClientes.Recordset.FindLast lcCadenaBuscar

      Case "SIGUIENTE"
         Me.datClientes.Recordset.FindNext lcCadenaBuscar

      Case "ANTERIOR"
         Me.datClientes.Recordset.FindPrevious lcCadenaBuscar
      End Select
   Case "TABLE"
.................
.................
   End Select

   If Me.datClientes.Recordset.NoMatch Then
      MsgBox "Registro no encontrado", , "Resultado de la búsqueda"
   Else
      MsgBox "Registro encontrado", , "Resultado de la búsqueda"
   End If

End If
End Sub
```

Código fuente 505

Lo que hacemos en esta parte del procedimiento, es construir una cadena de búsqueda con los valores de los diferentes ComboBox, y después aplicar el método Find, en función del tipo de búsqueda que queramos.

Objetos para la definición de datos

Hasta el momento, todas las operaciones con objetos que hemos visto, se han basado en la manipulación de datos en las tablas. Hemos consultado registros, añadido, modificado, ordenado, etc.

Pero DAO también permite crear los diferentes elementos dentro de los cuales está organizada la información, como son una base de datos, una tabla, un índice, etc., lo que nos puede ser muy útil para crear bases de datos, tablas temporales, índices, consultas, etc., en tiempo de ejecución.

A continuación veremos varios ejemplos de este tipo de situaciones, todos ellos contenidos en las opciones del menú *Definición de datos*, de la aplicación VarioDAO.

Bases de datos

Crear una nueva base de datos es tan sencillo como llamar al método *CreateDatabase()* de un objeto espacio de trabajo.

```
CreateDatabase( cNombre, cEscenario, xOpciones )
```

- cNombre. Cadena con el nombre de la base de datos, se puede incluir la ruta completa del fichero a crear.

- cEscenario. Combinación de cadena y constante, que establecen una contraseña y el tipo de ordenación dependiendo del lenguaje utilizado. Algunas de las constantes utilizadas son: dbLangGeneral, dbLangArabic, dbLangSpanish, dbLangJapanese, etc.

  Ejemplo:

  ```
  dbLangSpanish & "pwd=truco"
  ```

- xOpciones. Constante o combinación de constantes, que indican si el objeto estará codificado, y la versión a usar. Algunas de las constantes disponibles son: *dbEncrypt*, *dbVersion10*, *dbVersion30*, etc.

La opción *Base de datos* del menú, ejecuta el código fuente 506.

```
Private Sub mnuDeBaseDatos_Click()
' crear una nueva base de datos
Dim ldbAlmacen As Database
Set ldbAlmacen = gwsVarioDAO.CreateDatabase(gcRutaDatos & "Almacen", _
   dbLangGeneral)
MsgBox "Se ha creado la base de datos Almacén", , "Aviso"
End Sub
```

Código fuente 506

Mediante Access u otra herramienta para manejar bases de datos podremos ver los resultados.

Figura 522. Nueva base de datos creada en tiempo de ejecución.

Tablas

El siguiente paso consiste en crear una tabla y añadirla a la nueva base de datos.

En el código fuente 507 vemos como hacer estas operaciones en el código que corresponde a la opción de menú *Tabla*.

```
Private Sub mnuDeTabla_Click()
' crear una tabla para la nueva base
' de datos y añadirle campos
Dim ldbAlmacen As Database
Dim ltblArticulos As TableDef
Dim lfldCampoArticulos As Field
Dim lRCTArticulos As Recordset
' abrir la base de datos recién creada
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
' crear una nueva tabla para la base de datos
' con el método CreateTableDef() del objeto
' base de datos
Set ltblArticulos = ldbAlmacen.CreateTableDef("Articulos")
MsgBox "Se ha creado la tabla Artículos", , "Aviso"
' crear los campos para la tabla usando el método
' CreateField() del objeto tabla
' crear campo referencia
Set lfldCampoArticulos = ltblArticulos.CreateField("Referencia", _
   dbText, 5)
' agregar los campos a la tabla, esto lo conseguimos
' utilizando el método Append() de la colección Fields
' perteneciente al objeto tabla
ltblArticulos.Fields.Append lfldCampoArticulos
' crear campo descripción
Set lfldCampoArticulos = ltblArticulos.CreateField("Descripcion", _
   dbText, 30)
ltblArticulos.Fields.Append lfldCampoArticulos
' crear campo tipo
Set lfldCampoArticulos = ltblArticulos.CreateField("Tipo", _
   dbText, 2)
ltblArticulos.Fields.Append lfldCampoArticulos
' crear campo precio
Set lfldCampoArticulos = ltblArticulos.CreateField("Precio", _
   dbLong)
ltblArticulos.Fields.Append lfldCampoArticulos
MsgBox "Se han creado los campos para la tabla Artículos", , "Aviso"
' añadir la tabla a la base de datos usando el
' método Append() de la colección TableDefs
' perteneciente al objeto base de datos
ldbAlmacen.TableDefs.Append ltblArticulos
MsgBox "Se ha añadido la tabla Artículos a la base de datos Almacén", _
   , "Aviso"
' abrir recordset de la tabla artículos
Set lRCTArticulos = ldbAlmacen.OpenRecordset("Articulos", dbOpenTable)
' añadir registros al recordset
lRCTArticulos.AddNew
lRCTArticulos("Referencia") = "88"
lRCTArticulos("Descripcion") = "Impresora matricial"
lRCTArticulos("Tipo") = "TT"
lRCTArticulos("Precio") = 25000
lRCTArticulos.Update
lRCTArticulos.AddNew
lRCTArticulos("Referencia") = "20"
lRCTArticulos("Descripcion") = "Procesador 166"
lRCTArticulos("Tipo") = "BB"
lRCTArticulos("Precio") = 200
lRCTArticulos.Update
lRCTArticulos.AddNew
lRCTArticulos("Referencia") = "75"
lRCTArticulos("Descripcion") = "Mesa ordenador"
```

```
lRCTArticulos("Tipo") = "MM"
lRCTArticulos("Precio") = 5000
lRCTArticulos.Update
lRCTArticulos.AddNew
lRCTArticulos("Referencia") = "66"
lRCTArticulos("Descripcion") = "Silla regulable"
lRCTArticulos("Tipo") = "AA"
lRCTArticulos("Precio") = 100
lRCTArticulos.Update
MsgBox "Se han añadido registros la tabla Artículos", , "Aviso"
End Sub
```

Código fuente 507

Después de abrir la base de datos, creamos una nueva tabla con el método *CreateTableDef()* del objeto *Database*.

```
CreateTableDef( cNombre, xAtributos, cOrigen, cConectar )
```

- cNombre. Cadena con el nombre para la tabla.
- xAtributos. Constante o combinación de constantes, que definen varias características para la tabla.
  - dbAttachExclusive. En bases de datos que usen el motor Jet, esta constante señala que la tabla es una tabla adjunta abierta para uso exclusivo. Esta constante se puede usar para un objeto TableDef añadido para una tabla local, pero no se puede usar con una remota.
  - dbAttachSavePWD. En bases de datos que usen el motor Jet, indica que la identificación de usuario y la contraseña para la tabla conectada de forma remota se guardan junto con la información de la conexión. Es posible establecer esta constante en un objeto TableDef añadido para una tabla remota, pero no para una tabla local.
  - dbSystemObject. Indica que la tabla es una tabla del sistema proporcionada por el motor Jet. Es posible establecer esta constante en un objeto TableDef añadido.
  - dbHiddenObject. La tabla es una tabla oculta del motor Jet. Esta constante, se puede establecer en un objeto TableDef añadido.
  - dbAttachedTable. Indica que la tabla es una tabla adjunta de una base de datos que no es de ODBC como Microsoft Jet (de sólo lectura).
  - dbAttachedODBC. Indica que la tabla es una tabla adjunta de una base de datos ODBC, como el Microsoft SQL Server (de sólo lectura).
- cOrigen. Parámetro opcional que contiene una cadena con el nombre de una tabla externa, que es la que proporciona los datos.
- cConectar. Cadena de conexión a una base de datos.

Para crear cada campo de la tabla, usamos el método *CreateField()* del objeto tabla.

```
CreateField( cNombre, xTipo, nTamaño )
```

- cNombre. Cadena con el nombre del campo.

- xTipo. Constante con el tipo de dato del campo, como dbDate, dbLong, dbText, dbInteger, etc.

- nTamaño. Número que indica el tamaño del campo. Este parámetro no se utiliza en los campos numéricos o de ancho fijo.

Finalizadas estas operaciones, podemos pasar a Access y ver el resultado.

Figura 523. Nueva tabla creada en tiempo de ejecución.

Índices

Si una tabla va a tener una gran cantidad de registros, es muy recomendable crear uno o varios índices para ella. La opción *Índices* del menú, nos muestra como hacer este trabajo.

```
Private Sub mnuDeIndices_Click()
' crear un índice para la tabla
Dim ldbAlmacen As Database
Dim ltblArticulos As TableDef
Dim lfldCampoIndice As Field
Dim lidxIndiceTabla As Index
' abrir base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
' tomar una tabla
Set ltblArticulos = ldbAlmacen.TableDefs("Articulos")
' crear un índice para la tabla
Set lidxIndiceTabla = ltblArticulos.CreateIndex("Referencia")
MsgBox "Se ha creado el índice Referencia para la tabla Artículos", , "Aviso"
' crear un campo para el índice
Set lfldCampoIndice = lidxIndiceTabla.CreateField("Referencia")
' añadir el campo al índice
lidxIndiceTabla.Fields.Append lfldCampoIndice
' añadir el índice a la tabla
ltblArticulos.Indexes.Append lidxIndiceTabla
MsgBox "Se ha añadido el índice a la tabla", , "Aviso"
End Sub
```

Código fuente 508

Después de abrir la base de datos y tomar un objeto tabla de su colección *TableDefs*, creamos un índice con el método *CreateIndex()* de la tabla, pasando como parámetro el nombre del índice a crear.

Este índice está todavía incompleto, puesto que hay que crear tantos campos como vaya a tener su clave con el método *CreateField()* (en nuestro caso sólo hay un campo). A continuación, añadimos el campo al índice, y el índice a la tabla, con lo que tenemos completado el proceso. Desde Access, ya es posible ver el índice recién creado.

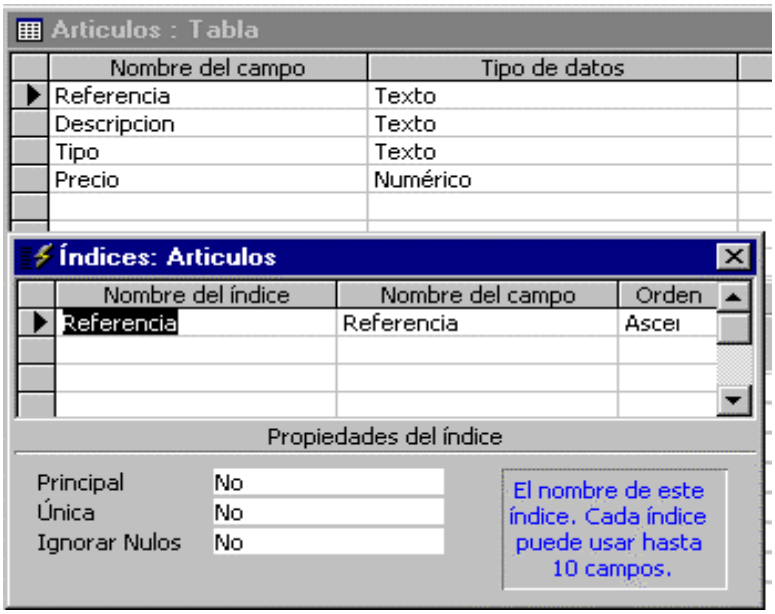


Figura 524. Índice creado mediante código.

Si necesitásemos cambiar alguna de las características del índice, lo haríamos a través de sus propiedades, alguna de las cuales se describen a continuación.

- DistinctCount. Si tenemos un índice que permite valores clave duplicados. Esta propiedad cuenta las ocurrencias de las claves una sola vez, y devuelve su número
- Foreign. El índice es clave ajena de otra tabla.
- IgnoreNulls. Si la clave del índice permite valores nulos y establecemos esta propiedad a verdadero, no se incluirán en el índice los valores nulos.
- Primary. Contiene un valor lógico que indica si el índice es la clave principal de la tabla.
- Required. Valor lógico que determina si se fuerza a dar valor al campo o campos que forman la clave del índice.
- Unique. Si es verdadero, el índice no podrá contener en su clave, valores duplicados. Si es falso, podrá contenerlos.

Consultas almacenadas

Un objeto *QueryDef* contiene una consulta contra una tabla de la base de datos, pudiéndose almacenar dentro de la misma, con lo cual se ejecuta más rápido que la consulta habitual mediante el método *OpenRecordset()* de un objeto *Database*, y una cadena con la consulta SQL. Esto es debido a que el *QueryDef* contiene la instrucción ya compilada, por lo que se ejecuta directamente, sin tener que

compilarla previamente, al contrario de lo que ocurre con el método *OpenRecordset*() del objeto *Database*.

En la aplicación VarioDAO, creamos una consulta y la almacenamos en la base de datos, al seleccionar la opción *Consulta* del menú.

```
Private Sub mnuDeConsulta_Click()
' crear una consulta para la tabla
Dim ldbAlmacen As Database
Dim lqryConsulta As QueryDef
' abrir nueva base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
' crear consulta para la base de datos
Set lqryConsulta = ldbAlmacen.CreateQueryDef("PorReferencia", _
   "SELECT * FROM Articulos WHERE Referencia > '70'")
MsgBox "Se ha creado una consulta y se ha añadido a la base de datos", , "Aviso"
End Sub
```

Código fuente 509

Como podemos ver, es una tarea muy simple. Basta con llamar al método *CreateQueryDef()* de la base de datos, y pasarle como parámetro el nombre de la consulta y la cadena con la sentencia SQL a ejecutar. Desde Access podemos ver el resultado.

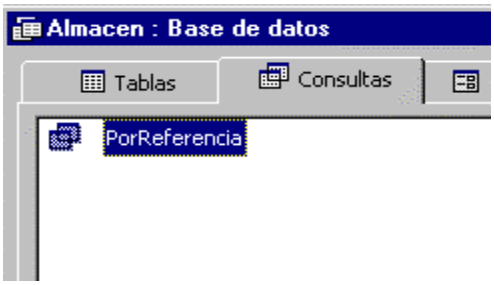


Figura 525. Ficha Consultas de la base de datos con la consulta creada en tiempo de ejecución.

Pero una consulta así no sirve de mucho, necesitamos ver su resultado. Así que para ello volvemos a la aplicación y ejecutamos la opción *Lanza Consulta* del menú, que carga la ventana *frmLanzaConsulta*, con todos los registros de la tabla.

```
Private Sub mnuDeLanConsulta_Click()
' ejecutar una consulta almacenada
' en la base de datos
Dim ldbAlmacen As Database
Dim lRCDAlmacen As Recordset
Dim lcSQL As String
' abrir base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
lcSQL = "SELECT * FROM Articulos"
Set lRCDAlmacen = ldbAlmacen.OpenRecordset(lcSQL)
Load frmLanzaConsulta
frmLanzaConsulta.mcTipoConsulta = "NORMAL"
frmLanzaConsulta.Caption = "Consulta almacenada"
Set frmLanzaConsulta.datDatos.Recordset = lRCDAlmacen
```

```
frmLanzaConsulta.Show
End Sub
```

Código fuente 510

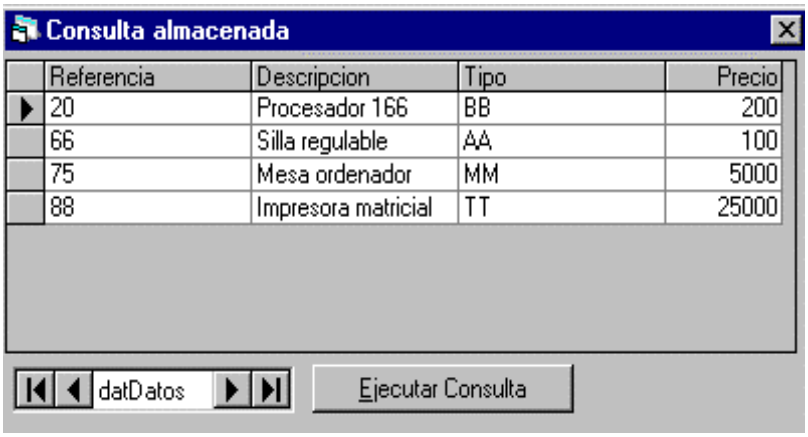


Figura 526. Ventana con todos los registros, antes de ejecutar la consulta.

Una vez aquí, pulsando el botón *Ejecutar Consulta*, lanzaremos la consulta, que filtrará los registros según indique la consulta.

```
Private Sub cmdEjecutar_Click()
' lanzar la consulta almacenada
Dim ldbAlmacen As Database
Dim lqryConsulta As QueryDef
Dim lRCDAlmacen As Recordset
' abrir base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
Select Case mcTipoConsulta
Case "NORMAL"
   ' abrir consulta almacenada
   Set lqryConsulta = ldbAlmacen.QueryDefs("PorReferencia")
   Set lRCDAlmacen = lqryConsulta.OpenRecordset
   Set datDatos.Recordset = lRCDAlmacen
.......
.......
End Select
End Sub
```

Código fuente 511

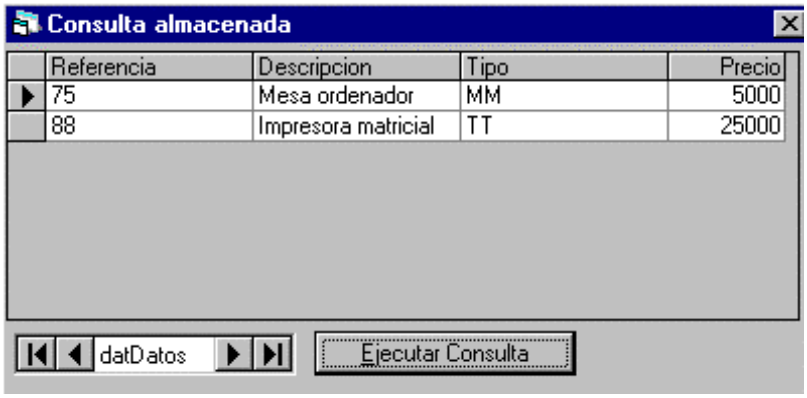


Figura 527. Después de ejecutar la consulta.

Consultas con parámetros

Tras el apartado anterior, podemos pensar que la consulta almacenada es muy potente, pero algo rígida, ¿y si queremos dar valores a la consulta en tiempo de ejecución?. Pues bien, esto también es posible en los objetos *QueryDef*, ya que se pueden pasar parámetros para obtener diferentes resultados, como podemos ver al seleccionar la opción *Parámetros*, en la aplicación de ejemplo.

```
Private Sub mnuDeParametros_Click()
' crear una consulta con parámetros para la tabla
Dim lcSQL As String
Dim ldbAlmacen As Database
Dim lqryConsulta As QueryDef
lcSQL = "PARAMETERS XTipo Text; "
lcSQL = lcSQL & "SELECT * FROM Articulos "
lcSQL = lcSQL & "WHERE Tipo <= XTipo "
lcSQL = lcSQL & "ORDER BY Precio "
' abrir nueva base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
' crear consulta con parámetros
Set lqryConsulta = ldbAlmacen.CreateQueryDef("ParamTipo", lcSQL)
' añadir la consulta a la base de datos
MsgBox "Se ha creado una consulta con parámetros " & _
   "y se ha añadido a la base de datos", , "Aviso"
End Sub
```

Código fuente 512

Aquí creamos otra consulta para la tabla, variando con respecto a la anteriormente definida en que incluimos los parámetros en la cadena que contiene la sentencia SQL.

Una vez creada la consulta en la base de datos, para ver su resultado, seleccionamos la opción *Lanza Parámetros* del menú, que visualiza la ventana del anterior ejemplo, pero mostrando esta vez un TextBox para incorporar el valor del parámetro que se pasará al objeto QueryDef que ejecuta la consulta.

```
Private Sub mnuDeLanParametros_Click()
' ejecutar una consulta con parámetros
Dim lfrmLanzaConsulta As frmLanzaConsulta
Dim ldbAlmacen As Database
```

```
Dim lRCDAlmacen As Recordset
Dim lcSQL As String
' abrir base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
lcSQL = "SELECT * FROM Articulos"
Set lRCDAlmacen = ldbAlmacen.OpenRecordset(lcSQL)
Set lfrmLanzaConsulta = New frmLanzaConsulta
Load lfrmLanzaConsulta
lfrmLanzaConsulta.Label1.Visible = True
lfrmLanzaConsulta.txtTipo.Visible = True
lfrmLanzaConsulta.mcTipoConsulta = "PARAMETROS"
lfrmLanzaConsulta.Caption = "Consulta con parámetros"
Set lfrmLanzaConsulta.datDatos.Recordset = lRCDAlmacen
lfrmLanzaConsulta.Show
End Sub
```

Código fuente 513

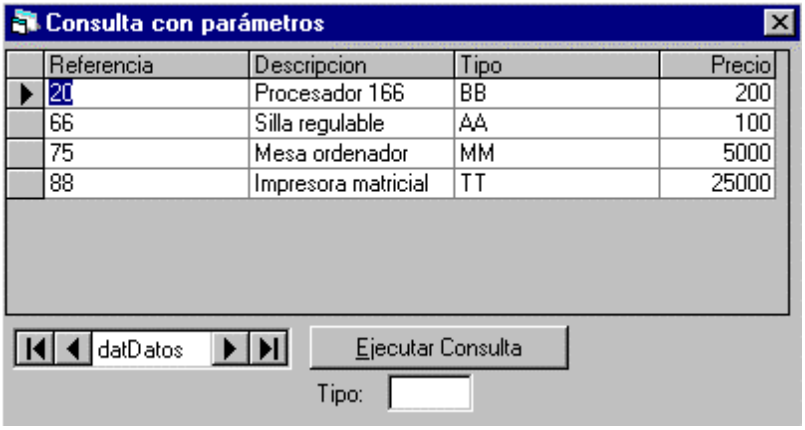


Figura 528. Antes de ejecutar la consulta con parámetros.

Ahora pulsaremos el botón *Ejecutar Consulta*, que llamará al mismo procedimiento que el anterior ejemplo, pero con un comportamiento diferente, ya que en este caso tomamos el valor del campo del formulario, utilizándolo como parámetro.

```
Private Sub cmdEjecutar_Click()
' lanzar la consulta almacenada
Dim ldbAlmacen As Database
Dim lqryConsulta As QueryDef
Dim lRCDAlmacen As Recordset
' abrir base de datos
Set ldbAlmacen = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Almacen.mdb")
Select Case mcTipoConsulta
........
........
........
Case "PARAMETROS"
   ' abrir consulta con parámetros
   ' y dar valores al parámetro
   Set lqryConsulta = ldbAlmacen.QueryDefs("ParamTipo")
   lqryConsulta("XTipo") = Me.txtTipo.Text
   Set lRCDAlmacen = lqryConsulta.OpenRecordset
   Set Me.datDatos.Recordset = lRCDAlmacen
End Select
End Sub
```

Código fuente 514

Figura 529. Después de ejecutar la consulta con parámetros.

Relaciones

Durante el diseño de una base de datos, utilizando Access, podemos establecer relaciones entre sus diferentes tablas. Una vez establecida una relación, se puede exigir en ella integridad referencial, para que al modificar o borrar registros en unas tablas, se produzca una eliminación o actualización en cascada en las tablas relacionadas.

Desde Visual Basic, los objetos DAO nos van a permitir hacer esto sin necesidad de tocar Access para nada. Desde la aplicación de ejemplo, las opciones del menú *Relaciones*, nos mostrarán los pasos a seguir.

Utilizaremos una base de datos llamada *FACTU.MDB*, que contiene una tabla de clientes y otra de facturas.

En primer lugar, pulsamos la opción *Establecer Relación* del menú, que ejecutará el código fuente 515.

```
Private Sub mnuReEstablecer_Click()
' crear relación entre dos tablas
Dim ldbFactu As Database
Dim ltblCliFac As TableDef
Dim ltblFacturas As TableDef
Dim lrelClienFactu As Relation
' abrir base de datos
Set ldbFactu = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Factu.mdb")
' abrir tablas a relacionar
Set ltblCliFac = ldbFactu.TableDefs("CliFac")
Set ltblFacturas = ldbFactu.TableDefs("Facturas")
' establecer relación entre tablas CliFac - Facturas
Set lrelClienFactu = ldbFactu.CreateRelation("ClienFactu", "CliFac", _
   "Facturas", dbRelationUpdateCascade + dbRelationDeleteCascade)
' crear campos de la relación
lrelClienFactu.Fields.Append lrelClienFactu.CreateField("CodigoCli")
lrelClienFactu.Fields("CodigoCli").ForeignName = "CodigoCli"
ldbFactu.Relations.Append lrelClienFactu
MsgBox "Relación entre tablas, creada", , "Atención"
End Sub
```

Código fuente 515

Lo que hacemos aquí es abrir la base de datos, y crear una relación entre las dos tablas con el método *CreateRelation()* del objeto *Database*.

```
CreateRelation(cNombre, cTablaPrincipal, cTablaExterna, kAtributos)
```

- cNombre. Cadena con el nombre del objeto Relation.
- cTablaPrincipal. Nombre de la tabla principal de la relación.
- cTablaExterna. Nombre de la tabla relacionada.
- kAtributos. Constante que define características para la relación, disponiendo de los siguientes:
  - dbRelationUnique. Define una relación de uno a uno.
  - dbRelationDontEnforce. Relación no impuesta, sin integridad referencial.
  - dbRelationInherited. Se trata de una relación existente en una base de datos no activa.
  - dbRelationUpdateCascade. Realiza las actualizaciones en cascada.
  - dbRelationDeleteCascade. Realiza las eliminaciones en cascada.

Una vez creado el objeto *Relation*, se crean los campos que formarán la relación y se añaden a la colección *Relations* de la base de datos. Si ahora pasamos a Access y vemos las relaciones de la base de datos *Factu*, nos aparecerá lo que podemos ver en la figura 530.

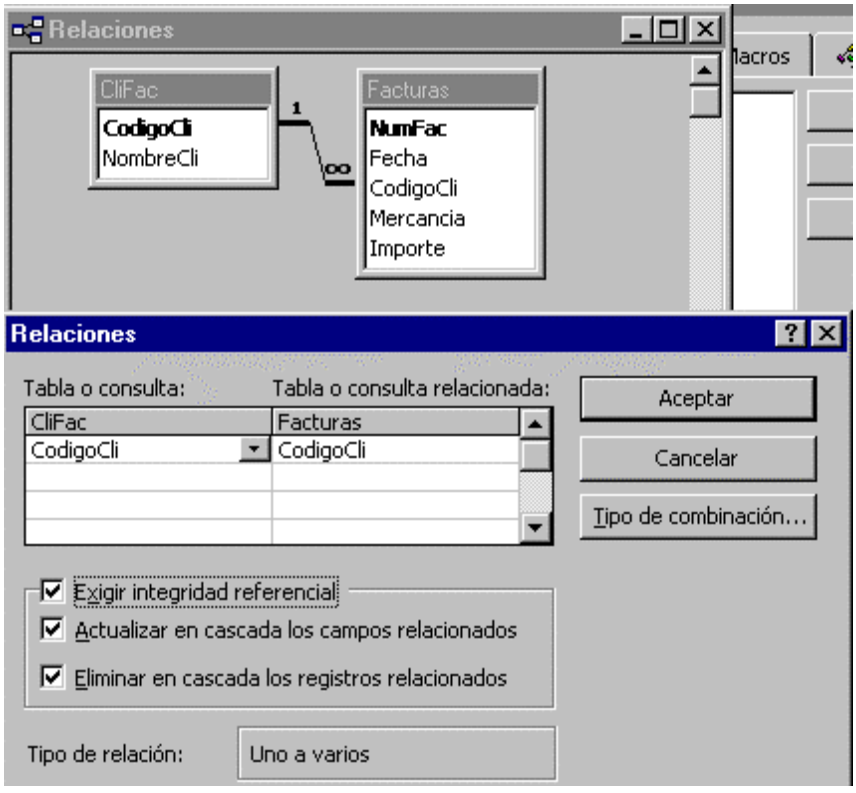


Figura 530. Relación establecida usando código.

En el código fuente 516, comprobaremos el funcionamiento de la integridad referencial que hemos exigido en la relación. Para ello seleccionamos la opción *Modificar Datos* del menú.

```
Private Sub mnuReModificar_Click()
' realizar modificación en una tabla,
' modificando en cascada la tabla
' relacionada
Dim ldbFactu As Database
Dim lRCDCliFac As Recordset
Dim lcSQL As String
Set ldbFactu = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Factu.mdb")
lcSQL = "SELECT * FROM CliFac WHERE CodigoCli = '20' "
Set lRCDCliFac = ldbFactu.OpenRecordset(lcSQL)
If lRCDCliFac.RecordCount > 0 Then
   lRCDCliFac.Edit
   lRCDCliFac("CodigoCli") = "77"
   lRCDCliFac.Update
End If
MsgBox "Modificación en tablas relacionadas, completada", , "Atención"
End Sub
```

Código fuente 516

En este procedimiento, tomamos un registro de la tabla principal CliFac y modificamos el valor del campo CodigoCli que tiene "20" por "77", con lo cual, todos lo registros en la tabla relacionada Facturas con el valor "20" en el campo CodigoCli, se actualizarán en cascada automáticamente a "77", quedando las tablas como se muestran en la figura 531.

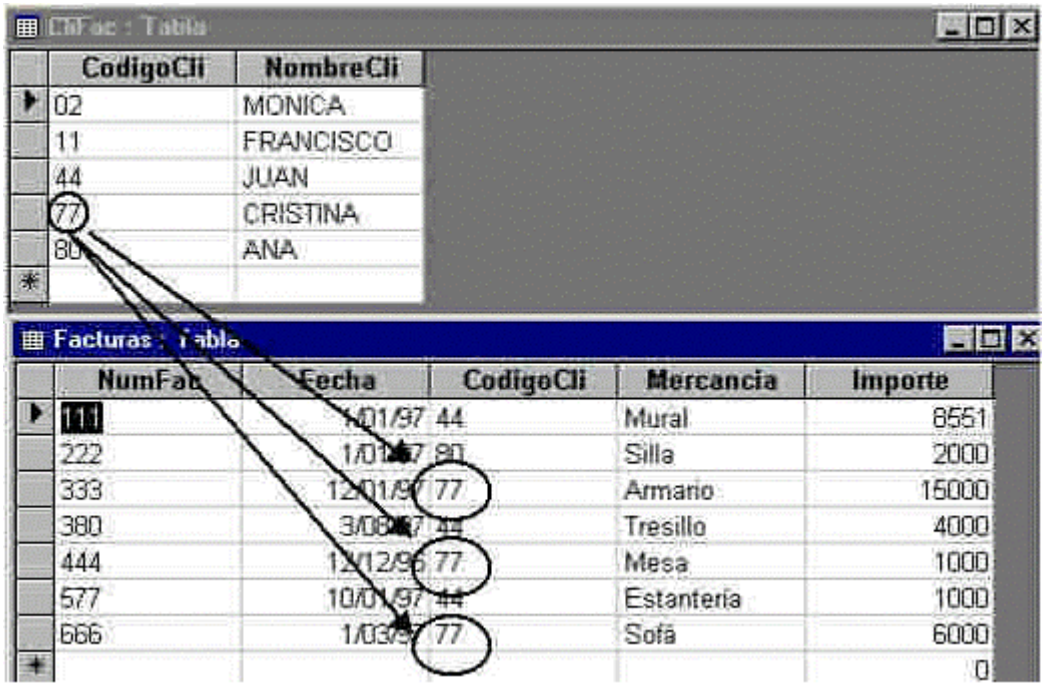


Figura 531. Actualizaciones en cascada entre dos tablas relacionadas.

En la siguiente prueba, se borrará el registro de la tabla CliFac con el valor "44" en el campo CodigoCli, de forma que se borrarán en cascada, todos los registros de la tabla Facturas que tengan el valor "44" en su campo CodigoCli. Para esta operación pulsamos la opción *Borrar Datos*, del menú.

```
Private Sub mnuReBorrar_Click()
' realizar borrado en una tabla,
' y borrado en cascada en la tabla
' relacionada
Dim ldbFactu As Database
Dim lRCDCliFac As Recordset
Dim lcSQL As String
Set ldbFactu = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Factu.mdb")
lcSQL = "SELECT * FROM CliFac WHERE CodigoCli = '44' "
Set lRCDCliFac = ldbFactu.OpenRecordset(lcSQL)
If lRCDCliFac.RecordCount > 0 Then
   lRCDCliFac.Delete
End If
MsgBox "Borrado en tablas relacionadas, completado", , "Atención"
End Sub
```

Código fuente 517

Como hemos podido comprobar, las capacidades de tratamiento de registros en cascada son muy potentes con el motor Jet. Ahora bien, también es cierto que hemos de tener sumo cuidado al efectuar este tipo de operaciones, ya que un error, podría hacernos perder accidentalmente un gran número de registros.

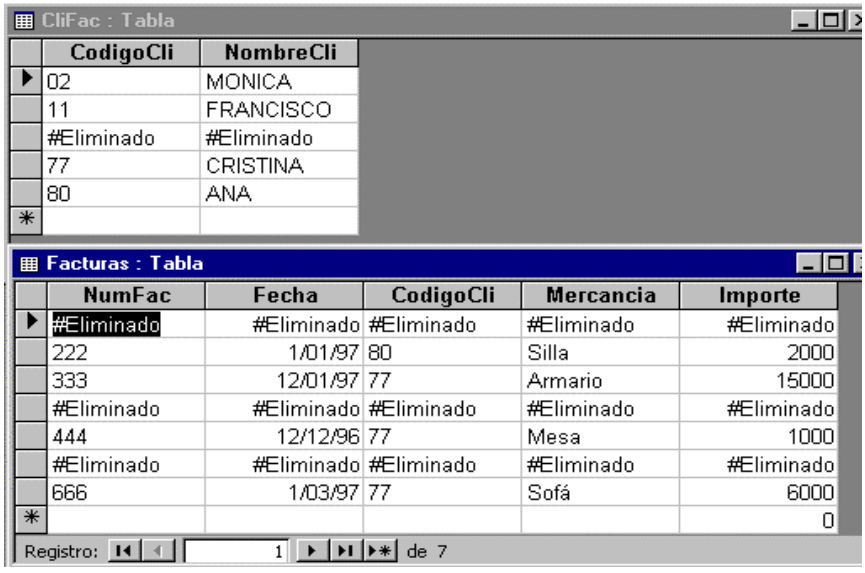


Figura 532. Eliminación en cascada de registros entre tablas relacionadas.

Transacciones

Una transacción permite tratar un conjunto de acciones realizadas en la base de datos como un único proceso, de forma que podamos volver al estado en que se encontraba la base de datos antes de comenzar dichas acciones.

En el ejemplo dedicado a estas operaciones, vamos a abrir una tabla, poner en marcha una transacción y realizar un borrado o modificación de registros. Pero antes de guardar definitivamente los cambios, pediremos confirmación de la operación, con lo que podremos dejar la base de datos tal y como estaba antes de empezar la transacción.

Seleccionamos la opción del menú *Transacciones* del programa VarioDAO, que visualiza la ventana *frmTransaccion.*

```
Private Sub mnuTransacciones_Click()
Dim lfrmTransaccion As frmTransaccion
Dim ldbTransac As Database
Dim lRCDProvincias As Recordset
Dim lcSQL As String
Set ldbTransac = gwsVarioDAO.OpenDatabase(gcRutaDatos & "Transac.mdb")
lcSQL = "SELECT * FROM Provincias"
Set lRCDProvincias = ldbTransac.OpenRecordset(lcSQL)
Set lfrmTransaccion = New frmTransaccion
Load lfrmTransaccion
Set lfrmTransaccion.datTransac.Recordset = lRCDProvincias
lfrmTransaccion.Show
End Sub
```

Código fuente 518

Figura 533. Ventana para realizar transacciones.

Al pulsar el botón *Borrar registros*, se ejecuta su método Click(), que contiene el código fuente 519.

```
Private Sub cmdBorrar_Click()
' borrar registros dentro de una transacción
Dim lnInd As Integer
Dim lnRespuesta As Integer
gwsVarioDAO.BeginTrans
Me.datTransac.Recordset.MoveFirst
For lnInd = 1 To 10
   Me.datTransac.Recordset.Delete
   Me.datTransac.Recordset.MoveNext
Next
lnRespuesta = MsgBox("¿Completar el borrado de registros?", _
```

```
   vbDefaultButton2 + vbYesNo, "Atención")
If lnRespuesta = vbYes Then
   gwsVarioDAO.CommitTrans
Else
   gwsVarioDAO.Rollback
End If
Me.datTransac.Recordset.Requery
End Sub
```

Código fuente 519

Se inicia una transacción por medio del método *BeginTrans*, del espacio de trabajo definido como global para todo el programa, seguida del borrado de una serie de registros. En la figura 534 se pide confirmación al usuario, y si es aceptada, se finaliza la transacción con *CommitTrans;* en el caso de que sea rechazada, se deja todo como estaba con *Rollback*.

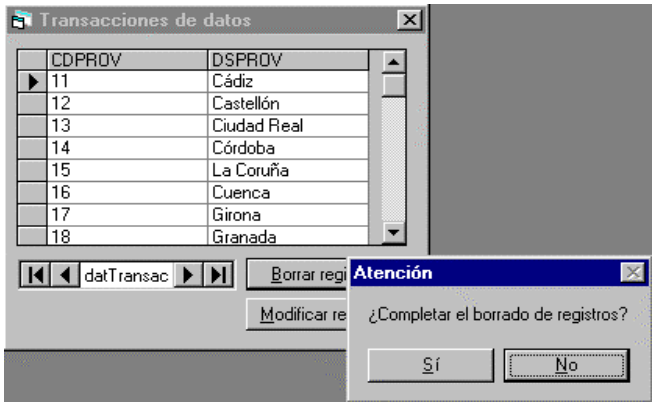


Figura 534. Transacción para el borrado de registros.

El trabajo del botón *Modificar registros* es similar al anterior. En este caso, modificamos el valor de varios registros y pedimos confirmación para grabar la modificación o para cancelarla.

```
Private Sub cmdModificar_Click()
' modificar registros dentro de una transacción
Dim lnInd As Integer
Dim lnRespuesta As Integer
gwsVarioDAO.BeginTrans
Me.datTransac.Recordset.MoveFirst
For lnInd = 1 To 10
   Me.datTransac.Recordset.Edit
   Me.datTransac.Recordset("DsProv") = "modificado"
   Me.datTransac.Recordset.Update
   Me.datTransac.Recordset.MoveNext
Next
lnRespuesta = MsgBox("¿Completar la modificación de registros?", _
   vbDefaultButton2 + vbYesNo, "Atención")
If lnRespuesta = vbYes Then
   gwsVarioDAO.CommitTrans
Else
   gwsVarioDAO.Rollback
End If
Me.datTransac.Recordset.Requery
End Sub
```

Código fuente 520

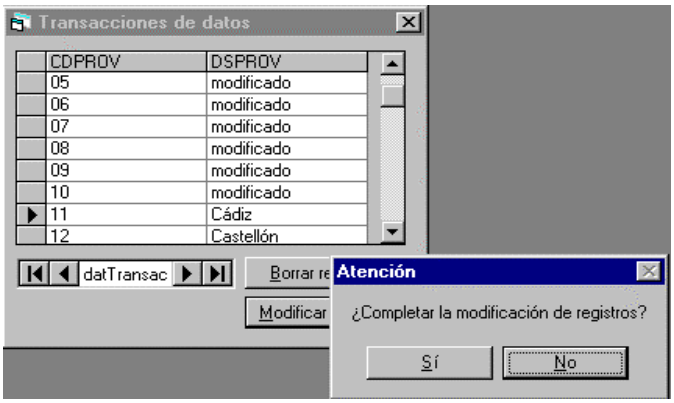


Figura 535. Transacción para la modificación de registros.

Operaciones con SQL

Las instrucciones SQL empleadas hasta ahora, se engloban dentro del llamado *Lenguaje de Manipulación de Datos (DML)*, ya que se encargan de realizar consultas de recuperación de registros sobre las tablas.

Por otro lado tenemos el Lenguaje de Definición de Datos (DDL), que es el encargado de crear los elementos de los que está formada una base de datos: tablas, campos, índices, etc.

Ambos tipos de sentencias, definición y manipulación, están disponibles utilizando el método *Execute* del objeto *Database*, pasándole como parámetro una cadena con la instrucción a ejecutar. De esta forma, disponemos de toda la potencia del SQL para actuar con las bases de datos.

En el menú *Operaciones con SQL* de la aplicación, se encuentra un conjunto de ejemplos de acciones utilizando sentencias SQL. Comenzaremos con la opción *Abrir Base de Datos*, que abre la base de datos utilizada en este ejemplo y habilita el resto de opciones de este menú, que permanecían deshabilitadas para que no se produjese ningún error por estar la base de datos cerrada. La variable que contiene la base de datos se ha definido global a la aplicación, para poder utilizarla en el resto de opciones de este menú.

```
Private Sub mnuOpAbrir_Click()
' abrir base de datos para ejemplos
' con sentencias SQL
Set gdbEjemSQL = gwsVarioDAO.OpenDatabase(gcRutaDatos & "EjemSQL.MDB")
MsgBox "Base de datos para operaciones SQL abierta", , "Atención"
' una vez abierta la base de datos
' habilitar el resto de opciones
' de este menú
Me.mnuOpCrearTabla.Enabled = True
Me.mnuOpCrearIndice.Enabled = True
Me.mnuOpAgregar.Enabled = True
Me.mnuOpModificarReg.Enabled = True
Me.mnuOpModificarTab.Enabled = True
Me.mnuOpBorrarReg.Enabled = True
Me.mnuOpEliminar.Enabled = True
End Sub
```

Código fuente 521

Continuaremos creando una nueva tabla para la base de datos, seleccionando la opción *Crear Tabla* del menú. Desde Access podremos ver el resultado.

```
Private Sub mnuOpCrearTabla_Click()
Dim lcSQL As String
lcSQL = "CREATE TABLE Inventario "
lcSQL = lcSQL & "( Codigo TEXT (2), "
lcSQL = lcSQL & "Nombre TEXT (30), "
lcSQL = lcSQL & "Cantidad LONG )"
gdbEjemSQL.Execute lcSQL
MsgBox "Tabla Inventario, creada", , "Atención"
End Sub
```

Código fuente 522

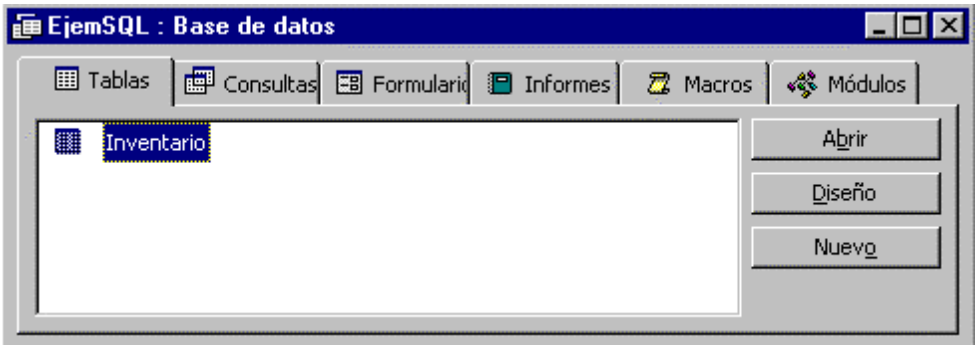


Figura 536. Tabla creada mediante instrucción SQL.

El siguiente paso, será crear un índice para la tabla con la opción *Crear Índice.*

```
Private Sub mnuOpCrearIndice_Click()
Dim lcSQL As String
lcSQL = "CREATE INDEX IdxCodigo ON Inventario ( Codigo ) "
gdbEjemSQL.Execute lcSQL
MsgBox "Se ha creado el índice IdxCodigo para la tabla Inventario", , _
   "Atención"
End Sub
```

Código fuente 523

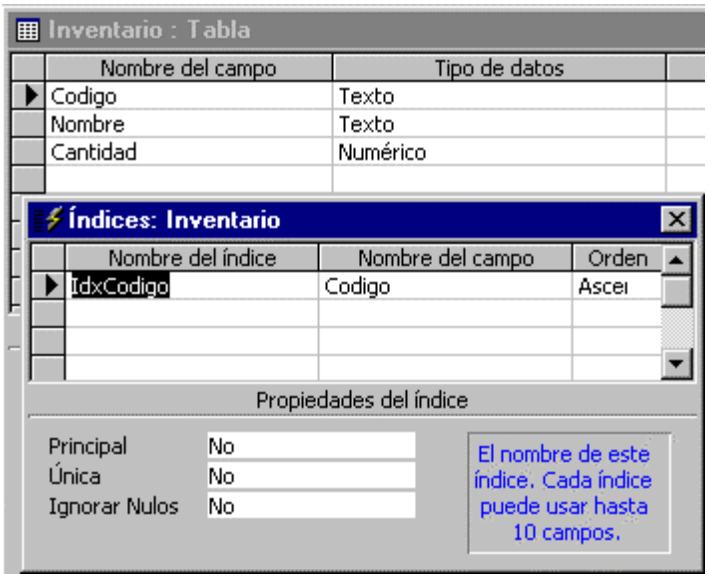


Figura 537. Índice creado mediante sentencia SQL.

Mediante la opción *Agregar registros* incorporaremos algunas filas a la tabla.

```
Private Sub mnuOpAgregar_Click()
Dim lcSQL As String
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '60', 'Silla de ruedas', 20 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '40', 'Mesa despacho', 10 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '88', 'Teclados', 55 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '20', 'Monitores', 40 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '10', 'Mesa reuniones', 2 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '90', 'Impresoras', 15 )"
gdbEjemSQL.Execute lcSQL
lcSQL = "INSERT INTO Inventario "
lcSQL = lcSQL & "( Codigo, Nombre, Cantidad ) "
lcSQL = lcSQL & "VALUES ( '50', 'C.P.U.', 45 )"
gdbEjemSQL.Execute lcSQL
MsgBox "Creadas las nuevas filas para la tabla Inventario", , "Atención"
End Sub
```

Código fuente 524

Figura 538. Filas agregadas a la tabla mediante sentencias SQL.

Observemos que, a pesar de que los registros no se han incorporado en orden, nos aparecen ordenados gracias al índice creado anteriormente para la tabla.

Otra de las acciones habituales en el trabajo con registros, es su modificación. Con la opción *Modificar registros*, vamos a cambiar el valor del campo *Nombre* en algunos registros, sobre la base de una condición.

```
Private Sub mnuOpModificarReg_Click()
Dim lcSQL As String
lcSQL = "UPDATE Inventario SET Nombre = 'MATERIAL DE OFICINA' "
lcSQL = lcSQL & "WHERE Codigo > '20' AND Codigo < '80' "
gdbEjemSQL.Execute lcSQL
MsgBox "Registros modificados", , "Atención"
End Sub
```

Código fuente 525

Figura 539. Registros modificados usando una instrucción SQL.

Para cambiar la estructura de una tabla, disponemos de la instrucción *ALTER TABLE*, que usamos en la opción *Modificar Tabla*, para añadir un nuevo campo a la tabla Inventario.

```
Private Sub mnuOpModificarTab_Click()
Dim lcSQL As String
lcSQL = "ALTER TABLE Inventario ADD COLUMN Actualizado DATETIME "
gdbEjemSQL.Execute lcSQL
MsgBox "Se ha insertado un nuevo campo a la tabla", , "Atención"
End Sub
```

Código fuente 526

Figura 540. Nueva columna creada mediante una sentencia SQL.

Tras haber jugueteado un buen rato con los registros, añadiendo, modificando, etc., vamos a dejar de molestar a algunos, borrándolos de la tabla. En esta ocasión seleccionamos la opción del menú *Borrar registros.*

```
Private Sub mnuOpBorrarReg_Click()
Dim lcSQL As String
lcSQL = "DELETE * FROM Inventario "
lcSQL = lcSQL & "WHERE Codigo > '25' "
gdbEjemSQL.Execute lcSQL
MsgBox "Registros eliminados", , "Atención"
End Sub
```

Código fuente 527

Figura 541. Resultado del borrado de registros mediante instrucción SQL.

Y ya por fin, después de darle tantas vueltas a la tabla *Inventario*, la mandamos a descansar, haciéndola desaparecer de la base de datos con la opción *Eliminar Tabla*.

```
Private Sub mnuOpEliminar_Click()
Dim lcSQL As String
lcSQL = "DROP TABLE Inventario "
gdbEjemSQL.Execute lcSQL
MsgBox "Tabla Inventario eliminada", , "Atención"
End Sub
```

Código fuente 528

Este modo de trabajo no hace un uso intensivo de los objetos DAO, ya que sólo se utiliza un objeto base de datos que llama a su método Execute, pasándole como parámetro una cadena con la instrucción SQL, pero puede ser interesante para el lector en situaciones que requieran tratar con un elevado número de registros.

Edición de registros

Hasta aquí hemos visto las capacidades que pone el motor Jet a nuestra disposición, de manera que ha llegado la hora de emplear en una aplicación real, algunas de las técnicas vistas a lo largo de los ejemplos. La aplicación a desarrollar, llamada DaoNif, se basa en un mantenimiento de códigos de identificación fiscal contenidos en una tabla, de esta forma comprobaremos que tal se desenvuelven los objetos DAO en la edición de registros.

Al iniciar un nuevo proyecto que deba utilizar objetos DAO, como condición indispensable hemos de abrir la ventana Referencias de VB (ver tema dedicado al IDE) e incluir la librería de objetos DAO: *Microsoft DAO 3.51 Object Library*, o de lo contrario cualquier intento de utilizar objetos DAO dará error.

En primer lugar hemos de analizar lo que queremos que el mantenimiento haga. Lógicamente el usuario debe poder crear, modificar y eliminar registros, pero si todas las características se redujeran a estas, la aplicación quedaría excesivamente limitada en lo referente a funcionalidades de cara al usuario.

Pensando más detenidamente en las necesidades del usuario, que las meramente dedicadas a la edición, podríamos plantear el incluir lo siguiente:

- Lista de registros. Será un control DBGrid incluido en el formulario, para poder consultar el contenido de la tabla.

- Cambiar el orden de los registros. Ya que la tabla dispone de una clave primaria por el campo Nif, y un índice por el campo Nombre, podemos utilizar OptionButton para cambiar el índice activo, y presentar los registros ordenados por uno u otro campo.

- Búsqueda de registros. Usaremos un TextBox, en el que al ir introduciendo un valor, se buscará en la tabla en función del índice activo.

Teniendo en cuenta las anteriores premisas, el interfaz de usuario podría resultar como se muestra en la figura 542.

Figura 542. Formulario para editar los registros de la tabla Nif.

El modo de iniciar la aplicación será mediante el procedimiento Main() y de la forma vista en ejemplos anteriores, tomando la ruta de los ficheros de una entrada en el registro de Windows, por lo que recomendamos al lector que vea el código del programa o de los ejemplos anteriores en lo que respecta a este punto.

Los controles básicos de este formulario son txtNif y txtNombre, que sirven para editar el valor de los campos de un registro.

Al cargar el formulario, las columnas del grid grdNif aparecen con un tamaño y valores predeterminados, que posiblemente no se adapten a las necesidades del usuario. Durante la etapa de diseño del formulario, podemos añadir un botón de forma temporal, con el código fuente 529.

```
Private Sub cmdAncho_Click()
MsgBox "col 0 " & Me.grdNif.Columns(0).Width
MsgBox "col 1 " & Me.grdNif.Columns(1).Width
End Sub
```

Código fuente 529

De esta forma al ejecutar la aplicación, podemos cambiar el ancho de las columnas hasta que sea el adecuado, al pulsar este botón nos devolverá el valor de ese ancho, que utilizaremos en un procedimiento propio denominado ConfigGrid(), este procedimiento será llamado dentro del método Load() del formulario, y configurará el control DBGrid del formulario en lo que respecta al título de las columnas y su ancho.

```
Public Sub ConfigGrid()
Me.grdNif.Columns(0).Width = 1171
Me.grdNif.Columns(0).Caption = "Código Nif"
Me.grdNif.Columns(1).Width = 2820
Me.grdNif.Columns(1).Caption = "Descripción del Código"
End Sub
```

Código fuente 530

Este método también será ejecutado cuando cambie el índice activo al pulsar uno de los OptionButton, ya que al establecer un nuevo índice para el recordset, se restauran los valores por defecto para el grid.

```
Private Sub optNIF_Click()
Me.datNif.Recordset.Index = "PrimaryKey"
Me.ConfigGrid
End Sub
```

Código fuente 531

Una vez terminada la fase de desarrollo, eliminaremos el CommandButton temporal que nos informa del ancho de las columnas del grid.

A propósito de los índices, como el lector podrá observar en el código de los botones de opción, para cambiar el índice actual, simplemente hemos de asignar a la propiedad Index del Recordset contenido en el control Data, un nuevo nombre válido de índice, y automáticamente, las filas reflejarán el nuevo orden contenido en el índice.

Es muy posible que al crear el formulario, este no se encuentre situado en el centro de la pantalla. Si deseamos que el formulario se presente en esta posición, podemos variar las propiedades correspondientes o hacer uso de la ventana de posición del formulario, situándolo directamente en el centro, como se muestra en la figura 543.

Figura 543. Situando el formulario de la aplicación en el centro de la pantalla.

Para buscar un registro por el índice activo, sólo hemos de ir tecleando el valor a localizar en el control txtBuscar, el método Change() de este control irá buscando dinámicamente el valor según vaya cambiando su contenido. Ya que el Recordset del control datNif es de tipo tabla, se usará el método Seek para las búsquedas.

```
Private Sub txtBuscar_Change()
Me.datNif.Recordset.Seek ">=", txtBuscar.Text
End Sub
```

Código fuente 532

Durante el funcionamiento de la ventana, según la operación que estemos realizando, los controles se habilitan o deshabilitan dependiendo de su cometido. El motivo de este comportamiento, es debido a que si por ejemplo, estamos modificando una fila de la tabla y dejamos activo el botón de borrado, podríamos borrar la fila en proceso de modificación, y encontrarnos con un error al intentar grabar el registro que estamos modificando, puesto que ya no existiría. El mismo problema nos encontraríamos, si pudiéramos cambiar de registro mediante el control Data o el grid, mientras estamos editándolo.

Por este motivo usamos la variable mcTipoEdit, declarada a nivel de módulo, en conjunción con el procedimiento PonTipoEdit(), de forma que según el tipo de edición que estemos realizando: nuevo registro, modificación o ninguna edición; la variable contendrá un valor que será evaluado en el procedimiento, habilitando y deshabilitando los controles pertinentes.

```
Private Sub PonTipoEdit()
' prepara los controles de ventana para editar
Select Case mcTipoEdit
Case "NUEVO", "MODIFICAR"
   ' preparar los controles que
   ' guardan los valores de los campos
   Me.txtNombre.Enabled = True

   If mcTipoEdit = "NUEVO" Then
      ' si se va a crear un nuevo registro
      Me.optNIF.Value = True
      Me.txtNIF.Enabled = True
      Me.txtNIF.SetFocus
   Else
      ' si vamos a editar un registro existente
      ' pasar el valor de los campos a los TextBox
      ' pero sin habilitar el campo que contiene el
      ' NIF, ya que este no se debe modificar
      Me.txtNIF.Text = Me.datNif.Recordset("NIF")
      Me.txtNombre.Text = Me.datNif.Recordset("Nombre")
      Me.txtNombre.SetFocus
   End If
   ' preparar el resto de controles del formulario
   Me.fraOrdenFichero.Enabled = False
   Me.txtBuscar.Enabled = False
   Me.grdNif.Enabled = False

   Me.datNif.Visible = False

   Me.cmdGrabar.Enabled = True
   Me.cmdCancelarEdic.Enabled = True

   Me.cmdNuevo.Enabled = False
   Me.cmdModificar.Enabled = False
   Me.cmdBorrar.Enabled = False
Case "FINEDIT"
   ' al terminar la edición de un registro
   ' volver a preparar los controles para
   ' este modo de trabajo
   Me.txtNIF.Enabled = False
   Me.txtNombre.Enabled = False
   Me.txtBuscar.Enabled = True
   Me.datNif.Visible = True
   Me.grdNif.Enabled = True
   Me.cmdGrabar.Enabled = False
   Me.cmdCancelarEdic.Enabled = False

   Me.fraOrdenFichero.Enabled = True
```

```
   Me.cmdNuevo.Enabled = True
   Me.cmdModificar.Enabled = True
   Me.cmdBorrar.Enabled = True
   Me.grdNif.SetFocus
End Select
End Sub
```

Código fuente 533

Una vez que hemos terminado de editar un registro (nuevo o modificado), pulsaremos el botón cmdGrabar, que en su método Click(), comprobará que existe valor para los campos en la ventana y grabará el registro en la tabla, notificándonos los posibles errores que se puedan producir al tener activada la comprobación de errores. Al terminar la grabación del registro, se volverán a establecer los controles en su estado inicial.

```
Private Sub cmdGrabar_Click()
' grabar a la tabla los campos del formulario
Dim lnRespuesta As Integer
Dim lcNIF As String
' activar control de errores
' para este procedimiento
On Error Resume Next
' comprobar que los campos tienen valor
If Len(Me.txtNIF.Text) = 0 Then
   MsgBox "El campo NIF ha de tener valor", , "Error"
   Exit Sub
End If
If Len(Me.txtNombre.Text) = 0 Then
   MsgBox "El campo Nombre ha de tener valor", , "Error"
   Exit Sub
End If
' en función de si es un nuevo registro
' o uno existente se agrega un nuevo
' registro o se pone en modo edición el
' que estamos modificando
Select Case mcTipoEdit
Case "NUEVO"
   ' añadir registro
   Me.datNif.Recordset.AddNew

Case "MODIFICAR"
   ' editar registro
   Me.datNif.Recordset.Edit
End Select
' pasar el contenido de los TextBox
' a los campos del registro
Me.datNif.Recordset("NIF") = Me.txtNIF.Text
Me.datNif.Recordset("Nombre") = Me.txtNombre.Text
Me.datNif.Recordset.Update
If Err.Number = 3022 Then
   lnRespuesta = MsgBox("Ya existe ese NIF", vbOKOnly, "Atención")
   Err.Clear
   Me.datNif.Recordset.MoveFirst
   Exit Sub
End If
lcNIF = Me.txtNIF.Text
LimpiaControles
Select Case mcTipoEdit
Case "NUEVO"
   Me.datNif.Recordset.MoveFirst
```

```
   mcTipoEdit = "FINEDIT"
   PonTipoEdit

   ' posicionarse en el registro
   ' que se acaba de editar
   Me.datNif.Recordset.Seek "=", lcNIF

Case "MODIFICAR"
   mcTipoEdit = "FINEDIT"
   PonTipoEdit

End Select
End Sub
```

Código fuente 534

Ya sólo queda que el lector experimente con este programa, y modifique alguna de sus funcionalidades o incluya otras nuevas, para comprobar el comportamientos de los diferentes objetos y controles.

Aplicando la teoría

Terminados los aspectos teóricos de la programación con bases de datos, que hemos salpicado con algunos breves ejemplos, es hora de abordar el desarrollo de una aplicación más completa y compleja, aunque no demasiado, no es nuestra intención asustar al lector.

Lo que haremos en este apartado será aplicar algunos de los puntos aprendidos desde el comienzo de este tema hasta el momento actual. Todo el conocimiento adquirido lo enfocaremos en desarrollar una aplicación que confeccione el libro de IVA de las facturas recibidas en una empresa, o IVA Soportado, de manera que tendremos que crear una base de datos, tablas, diseñar un interfaz de usuario, y unir todas las piezas para que funcionen conjuntamente.

No vamos a partir completamente desde cero, ya que el mantenimiento de la tabla de Nif del apartado anterior podrá aprovecharse en este programa.

La aplicación IVASop, contiene todo el código de los ejemplos mostrados a continuación, para que el lector pueda efectuar un mejor seguimiento de los puntos explicados en este apartado.

Diseño de la base de datos

En esta fase del desarrollo, crearemos la base de datos IvaSop, en la que incluiremos la ya mencionada tabla Nif, para los códigos fiscales. Otra de las tablas que debemos incluir será para contener los diferentes tipos de IVA, a la que llamaremos TipoIVA.

En este punto hemos de pensar como afrontar la captura de apuntes del libro de IVA. Ya que este problema se puede abordar desde diferentes puntos de vista, nos hemos decidido por una solución que puede ser tan buena como las demás. Al comienzo de cada ejercicio fiscal, crearemos en tiempo de ejecución una tabla que será la que contenga los apuntes del año en curso. El código que genere esta tabla, deberá crear las relaciones con las otras tablas de la base de datos y la clave primaria para la tabla de apuntes.

Como ayuda en la generación de listados, crearemos las tablas de apoyo Lineas y Resumen, en las cuales volcaremos los apuntes del trimestre a imprimir y el resumen de las cantidades por tipo de IVA.

La figura 544, muestra las relaciones establecidas entre las tablas de la base de datos. Para más información sobre los campos, índices, etc., rogamos al lector el uso de alguna herramienta tipo Access para manejo de bases de datos.

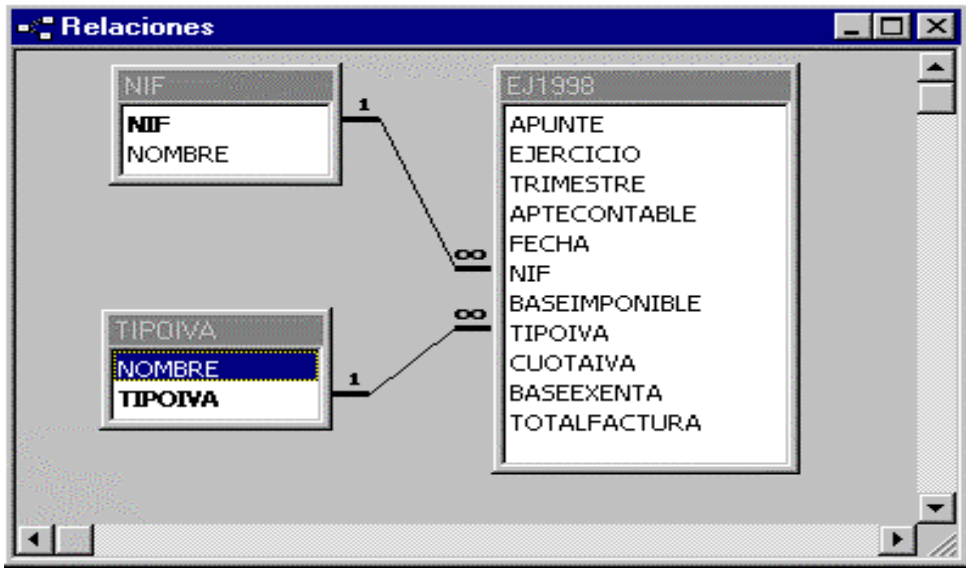


Figura 544. Relaciones entre las tablas de la base de datos IvaSop.

Las relaciones establecidas, exigirán integridad referencial, pero no se efectuarán borrados ni actualizaciones en cascada, ya que piense el lector en el grave problema que se produciría si borramos uno de los códigos de Nif, eliminándose en cascada todos los registros de la tabla de apuntes que tengan ese mismo código, en el caso de que estableciéramos eliminaciones en cascada.

Interfaz de usuario

Comenzaremos escribiendo el procedimiento Main(), para efectuar la apertura del espacio de trabajo y base de datos para la aplicación.

A continuación, crearemos el formulario frmInicio (figura 545), de tipo MDI, que servirá como ventana principal de la aplicación, contenedora de las demás. Estableceremos las propiedades necesarias y crearemos un menú con las opciones de esta aplicación.

Figura 545. Formulario frmInicio de la aplicación IvaSop.

Como ya tenemos creado el mantenimiento para la tabla de NIF, lo incorporaremos al proyecto agregando el código para abrir esta ventana en la opción *Archivo+Fichero de NIF* del formulario frmInicio.

```
Private Sub mnuArNif_Click()
Dim lfrmNif As frmNif
Set lfrmNif = New frmNif
Load lfrmNif
lfrmNif.Show
lfrmNif.grdNif.SetFocus
Me.mnuArNif.Enabled = False
End Sub
```

Código fuente 535

El lector habrá notado que al final de este procedimiento, se deshabilita esta opción de menú. Lógicamente esto se hace para no permitir al usuario abrir al mismo tiempo, más de una ventana de mantenimiento de NIF durante la ejecución del programa. Para volver a habilitar esta opción, haremos uso del evento Unload() del formulario de mantenimiento de la tabla NIF, que se ejecuta al cerrar la ventana.

```
Private Sub Form_Unload(Cancel As Integer)
gfrmInicio.mnuArNif.Enabled = True
End Sub
```

Código fuente 536

La variable gfrmInicio, contiene el objeto formulario MDI del programa, se ha definido a nivel global a la aplicación, debido a que al igual que en el fuente anterior, necesitaremos acceder a ella desde varios puntos del programa, en este caso ha sido para habilitar una opción de menú.

Creación de la tabla de apuntes del ejercicio fiscal en tiempo de ejecución

Al seleccionar el menú *Archivo+Libro de Registro* en la ventana principal, se abrirá la ventana *frmEjercTrim*, que nos pedirá el ejercicio y trimestre con el cual deseamos trabajar.

```
Private Sub mnuArLibro_Click()
Dim lfrmEjercTrim As frmEjercTrim
Set lfrmEjercTrim = New frmEjercTrim
Load lfrmEjercTrim
lfrmEjercTrim.Show vbModal, gfrmInicio
End Sub
```

Código fuente 537

Figura 546. Ventana para seleccionar el ejercicio y trimestre.

Este formulario se muestra con estilo *modal*, puesto que necesitamos que el usuario nos proporcione los datos de ejercicio y trimestre antes de poder introducir apuntes al libro de IVA. Una vez completados los campos y pulsando *Aceptar*, se comprobará que exista en la base de datos una tabla para los apuntes del ejercicio, de no ser así, y si el usuario da su conformidad, se creará automáticamente la tabla y las relaciones correspondientes con las demás tablas de la base de datos. El código fuente 538, corresponde a la llamada al procedimiento de creación de la tabla desde el método *Click()* del botón *Aceptar*, y a *CrearEjercicio*, que es el procedimiento de creación de la tabla.

```
Private Sub cmdAceptar_Click()
.......
.......
' intentar abrir tabla del ejercicio correspondiente
lcEjercicio = "EJ" & Trim(Me.txtEjercicio.Text)
Do While True
   Set lRCTEjercicio = gdbIvaSop.OpenRecordset(lcEjercicio, dbOpenTable)
   ' si existe la tabla y no ha habido error,
   ' salir del bucle
   If Err.Number = 0 Then
      Exit Do
   End If
   ' si no existe la tabla del ejercicio, error...
   If Err.Number > 0 Then
      Err.Clear

      lnRespuesta = MsgBox("No existen datos del ejercicio " & _
         Trim(Me.txtEjercicio.Text) & ". ¿Crearlo?", vbYesNo, _
         "Nuevo Ejercicio")

      If lnRespuesta = vbYes Then
         CrearEjercicio lcEjercicio
      Else
         Exit Sub
      End If

   End If
Loop
.......
.......
End Sub
' ------------------------------------------------------
Private Sub CrearEjercicio(rcEjercicio As String)
' crear tabla para los apuntes de
' un nuevo ejercicio fiscal
Dim lcCrear As String
Dim ltblNIF As TableDef
Dim ltblTipoIVA As TableDef
Dim ltblEjercicio As TableDef
Dim lrelRelacion As Relation
```

```
Dim lfldCampoRel As Field
' componer una cadena con la sentencia SQL
' de creación de la tabla
lcCrear = "CREATE TABLE " & rcEjercicio & " "
lcCrear = lcCrear & "(APUNTE LONG, "
lcCrear = lcCrear & "EJERCICIO TEXT(4), "
lcCrear = lcCrear & "TRIMESTRE TEXT(1), "
lcCrear = lcCrear & "APTECONTABLE TEXT(4), "
lcCrear = lcCrear & "FECHA DATETIME, "
lcCrear = lcCrear & "NIF TEXT(9), "
lcCrear = lcCrear & "BASEIMPONIBLE LONG, "
lcCrear = lcCrear & "TIPOIVA BYTE, "
lcCrear = lcCrear & "CUOTAIVA LONG, "
lcCrear = lcCrear & "BASEEXENTA LONG, "
lcCrear = lcCrear & "TOTALFACTURA LONG) "
gdbIvaSop.Execute lcCrear
' crear índice-clave primaria para la tabla
lcCrear = "CREATE INDEX PrimaryKey ON " & rcEjercicio & " ( APUNTE ) "
lcCrear = lcCrear & "WITH PRIMARY"
gdbIvaSop.Execute lcCrear
' ---------------------------------------
' establecer relaciones entre las tablas
' NIF y TIPOIVA con la tabla para el nuevo
' ejercicio recién creada
' tomar definiciones de tablas
Set ltblNIF = gdbIvaSop.TableDefs("NIF")
Set ltblTipoIVA = gdbIvaSop.TableDefs("TIPOIVA")
Set ltblEjercicio = gdbIvaSop.TableDefs(rcEjercicio)
' establecer relación entre la tabla NIF y la
' tabla del ejercicio
Set lrelRelacion = gdbIvaSop.CreateRelation("RelNif" & rcEjercicio, _
   "NIF", rcEjercicio)
' crear campo que forma la relación
Set lfldCampoRel = lrelRelacion.CreateField("NIF")
' añadir campo a la relación
lrelRelacion.Fields.Append lfldCampoRel
' indicar cual es el campo relacionado
' de la tabla ajena
lrelRelacion.Fields("NIF").ForeignName = "NIF"
' añadir la relación a la base de datos
gdbIvaSop.Relations.Append lrelRelacion
' establecer relación entre la tabla TIPOIVA y la
' tabla del ejercicio
Set lrelRelacion = gdbIvaSop.CreateRelation("RelTipo" & rcEjercicio, _
   "TIPOIVA", rcEjercicio)
' crear campo que forma la relación
Set lfldCampoRel = lrelRelacion.CreateField("TIPOIVA")
' añadir campo a la relación
lrelRelacion.Fields.Append lfldCampoRel
' indicar cual es el campo relacionado
' de la tabla ajena
lrelRelacion.Fields("TIPOIVA").ForeignName = "TIPOIVA"
' añadir la relación a la base de datos
gdbIvaSop.Relations.Append lrelRelacion
End Sub
```

Código fuente 538

Una vez abierto el Recordset de la tabla de apuntes, hemos de abrir otro Recordset un poco más especial. Dicho Recordset se basará en una consulta SQL que contendrá los campos de la tabla de apuntes del ejercicio y el campo Nombre de la tabla NIF, y será utilizado para visualizar los registros que vamos incorporando a la tabla, ordenados por el campo Apunte en un DBGrid del formulario *frmFichero*, utilizado para editar los apuntes.

```
Private Sub cmdAceptar_Click()
.......
.......
' construir la cadena para la consulta SQL
' que crea un recordset, que será el que visualice
' los apuntes de IVA en el grid de la ventana
lcSQL = "SELECT " & lcEjercicio & ".Apunte, "
lcSQL = lcSQL & lcEjercicio & ".Ejercicio, "
lcSQL = lcSQL & lcEjercicio & ".Trimestre, "
lcSQL = lcSQL & lcEjercicio & ".ApteContable, "
lcSQL = lcSQL & lcEjercicio & ".Fecha, "
lcSQL = lcSQL & lcEjercicio & ".NIF, "
lcSQL = lcSQL & "NIF.Nombre, "
lcSQL = lcSQL & lcEjercicio & ".BaseImponible, "
lcSQL = lcSQL & lcEjercicio & ".TipoIVA, "
lcSQL = lcSQL & lcEjercicio & ".CuotaIVA, "
lcSQL = lcSQL & lcEjercicio & ".BaseExenta, "
lcSQL = lcSQL & lcEjercicio & ".TotalFactura "
lcSQL = lcSQL & "FROM NIF INNER JOIN " & lcEjercicio & " "
lcSQL = lcSQL & "ON NIF.NIF = " & lcEjercicio & ".NIF "
lcSQL = lcSQL & "ORDER BY " & lcEjercicio & ".Apunte"
Set lRCDEjercGrid = gdbIvaSop.OpenRecordset(lcSQL, dbOpenDynaset)
' ocultar esta ventana
Me.Hide
' instanciar la ventana de apuntes de IVA
Set lfrmFichero = New frmFichero
Load lfrmFichero
lfrmFichero.mcEjercicio = Me.txtEjercicio.Text
lfrmFichero.mcTrimestre = Me.cboTrimestre.Text
lfrmFichero.Caption = lfrmFichero.Caption & "- Ejercicio: " & _
   Me.txtEjercicio.Text & " Trimestre: " & Me.cboTrimestre.Text
' asignar los recordsets con la tabla de apuntes
' del ejercicio
Set lfrmFichero.mRCDEjercicio = lRCTEjercicio
Set lfrmFichero.datIvaSop.Recordset = lRCDEjercGrid
lfrmFichero.Show
lfrmFichero.Configurar
gfrmInicio.mnuArLibro.Enabled = False
Unload Me
.......
.......
End Sub
```

Código fuente 539

El lector se preguntará la causa de solicitar los campos en la sentencia SQL de forma tan extraña, cuando se podía haber acortado con algo más simple como el código fuente 540.

```
lcSQL = "SELECT " & lcEjercicio & ".*, "NIF.Nombre"...
```

Código fuente 540

El motivo reside en que el orden de los campos solicitados en la sentencia SQL para crear el Recordset, será el mismo orden en que visualizaremos las columnas en el DBGrid enlazado al control Data que contiene el Recordset, por lo que si usáramos la forma fácil de la consulta obtendríamos la

columna con el nombre del NIF en un poco estético último lugar. Por esta causa, la forma empleada en el código es más trabajosa pero se obtienen resultados más vistosos.

Otra cuestión que podría plantear dudas, es la que atañe a la creación de un Recordset que abre la tabla de apuntes. El lector puede pensar que si ya hemos creado un Recordset con estos campos, que utilizamos para el DBGrid, ¿por qué no aprovecharlo para la edición de registros?. El problema reside en el campo Nombre de la tabla NIF incluido en la consulta del Recordset, ya que al intentar agregar o modificar un registro directamente a este Recordset, como dicho campo no pertenece a la tabla contra la que queremos actuar, se podrían provocar errores en la edición de registros.

Por esta causa, es mejor crear un Recordset exclusivamente para efectuar las operaciones de edición de registros de la tabla de apuntes de IVA y tener otro para visualizar el contenido de los campos mediante un DBGrid. De esta forma, tenemos la ventaja de que podemos visualizar la información en el formulario basándonos en una consulta SQL que englobe a varias tablas de la base de datos.

Una vez creados los recordsets, ocultamos la ventana frmEjercTrim y abrimos frmFichero, a la que asignamos varias propiedades antes de descargar definitivamente frmEjercTrim.

El formulario de edición de apuntes de IVA

El final del punto anterior, sirve de entrada al actual, ya que al cargar el formulario de edición de apuntes, se ejecuta su procedimiento *Configurar*, que prepara los controles contenidos en el formulario para editar los registros de la tabla: habilitando, deshabilitando, asignando valores iniciales, etc.

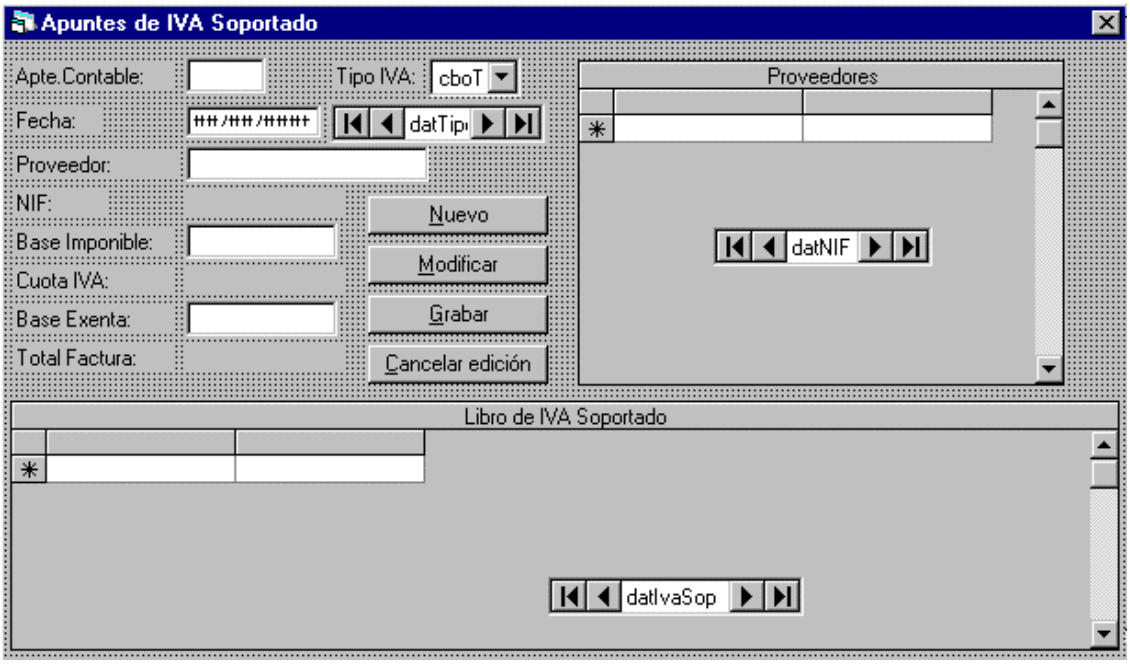


Figura 547. Formulario de edición de apuntes de IVA.

Uno de estos controles es el *DBCombo cboTipoIva*, que se trata de un *ComboBox* al que se le puede asignar un control Data a su propiedad *RowSource*, de manera que se rellene su contenido automáticamente con un campo de los contenidos en el Recordset del Data, como podemos ver en el código fuente 541.

```
Public Sub Configurar()
' preparar los controles de la ventana
Dim lRCTTipoIva As Recordset
Dim lRCTNif As Recordset
' abrir tabla TipoIVA y pasarla al control Data enlazado
Set lRCTTipoIva = gdbIvaSop.OpenRecordset("TIPOIVA", dbOpenTable)
lRCTTipoIva.Index = "NOMBRE"
Set Me.datTipoIva.Recordset = lRCTTipoIva
' configurar el combo que contiene los tipos de IVA
Me.cboTipoIva.ListField = "TipoIva"
' situarse en el IVA ordinario
Me.datTipoIva.Recordset.Seek "=", "ORDINARIO"
' poner el valor del iva en el combo
Me.cboTipoIva.Text = Me.datTipoIva.Recordset("TipoIva")
..............
..............
End Sub
```

Código fuente 541

En el anterior fragmento hemos visto como se utiliza la propiedad *ListField* del DBCombo, pasándole una cadena con el nombre del campo que se ha de utilizar para rellenar la parte de la lista de este control. La forma de configurar este control y el control DBList es la misma, en el caso de que necesitemos de las particularidades de un control de lista. Nosotros en este caso, hemos optado por un DBCombo, ya que cuando no es necesario utilizarlo, ocupa menos espacio en el formulario que un DBList.

Por pura inercia, para diseñar la selección de un NIF al confeccionar un apunte de IVA, rápidamente pensaríamos en utilizar una técnica como la anterior, mediante un DBCombo. Bien, esto es posible, pero no recomendable, ya que podemos crear un DBCombo enlazado a un control Data que contuviera la tabla NIF, pero el sistema de búsqueda de valores dentro de un DBCombo no se adapta como debiera a las necesidades de nuestro caso en este formulario, por lo que vamos a recurrir a una solución algo menos habitual, en lo que a utilizar los controles que por lógica deberíamos emplear para el diseño del formulario.

Vamos a usar un Recordset con la tabla NIF ordenada por el campo Nombre, pero enlazaremos el control Data que contiene el Recordset a un DBGrid en vez de a un DBCombo, ya que aunque debamos escribir nosotros el código para realizar la búsqueda de los registros en la tabla NIF, la potencia de búsqueda obtenida compensará el esfuerzo, teniendo en cuenta que será muy poco código el que tengamos que añadir. El código fuente 542 muestra como preparar este control para realizar búsquedas en él.

```
Public Sub Configurar()
............
............
............
............
' abrir tabla NIF y pasarla al control Data enlazado
Set lRCTNif = gdbIvaSop.OpenRecordset("NIF", dbOpenTable)
lRCTNif.Index = "NOMBRE"
Set Me.datNIF.Recordset = lRCTNif
' configurar el grid que contiene los NIF
Me.grdNIF.Columns(0).Visible = False
Me.grdNIF.Columns(1).Width = 3400
Me.grdNIF.Columns(1).Caption = "Nombre"
```

```
.............
.............
.............
End Sub
```

Código fuente 542

Cuando estemos editando un apunte de IVA, al tomar el foco el control *txtNombre* ("Proveedor" en el formulario), cada vez que tecleemos un carácter, se desencadenará el evento Change() de este control. Si en el método correspondiente a este evento incluimos la línea de código del código fuente 543, se realizará una búsqueda dinámicamente del contenido de este control en la tabla NIF del control Data datNIF, que se reflejará en el *DBGrid grdNif* ("Proveedores" en el título del grid).

```
Private Sub txtNombre_Change()
Me.datNIF.Recordset.Seek ">=", Me.txtNombre.Text
End Sub
```

Código fuente 543

Continuando con la configuración del formulario de apuntes de IVA, le toca el turno al DBGrid *grdIvaSop*, que será el que refleje las líneas que se vayan introduciendo en el libro de IVA Soportado. Para ello disponemos del procedimiento *ConfigGrid*, que se encarga sólo de preparar este control.

```
Public Sub Configurar()
.............
.............
' configurar grid del formulario
ConfigGrid
.............
.............
End Sub
' ------------------------------
Private Sub ConfigGrid()
' configurar grid que contiene los apuntes de IVA
' ocultar columnas que no son necesarias
Me.grdIvaSop.Columns(1).Visible = False
Me.grdIvaSop.Columns(5).Visible = False
' cambiar los títulos de las columnas
Me.grdIvaSop.Columns(0).Caption = "Apte."
Me.grdIvaSop.Columns(2).Caption = "T"
Me.grdIvaSop.Columns(3).Caption = "Cont."
Me.grdIvaSop.Columns(4).Caption = "Fecha"
Me.grdIvaSop.Columns(6).Caption = "Nombre"
Me.grdIvaSop.Columns(7).Caption = "Base Imp."
Me.grdIvaSop.Columns(8).Caption = "Tipo"
Me.grdIvaSop.Columns(9).Caption = "Cuota"
Me.grdIvaSop.Columns(10).Caption = "Base Ex."
Me.grdIvaSop.Columns(11).Caption = "Total Fac."
' ajustar el ancho de las columnas para que
' entren todas dentro del área visible asignada
' al control grid
Me.grdIvaSop.Columns(0).Width = 494
Me.grdIvaSop.Columns(2).Width = 195
Me.grdIvaSop.Columns(3).Width = 494
Me.grdIvaSop.Columns(4).Width = 794
Me.grdIvaSop.Columns(6).Width = 2294
```

```
Me.grdIvaSop.Columns(7).Width = 900
Me.grdIvaSop.Columns(8).Width = 420
Me.grdIvaSop.Columns(9).Width = 945
Me.grdIvaSop.Columns(10).Width = 945
Me.grdIvaSop.Columns(11).Width = 1019
' dar formato a las columnas numéricas
Me.grdIvaSop.Columns(7).NumberFormat = "###,###,###"
Me.grdIvaSop.Columns(9).NumberFormat = "###,###,###"
Me.grdIvaSop.Columns(10).NumberFormat = "###,###,###"
Me.grdIvaSop.Columns(11).NumberFormat = "###,###,###"
End Sub
```

Código fuente 544

Pensamos que los comentarios introducidos dentro del propio procedimiento son lo suficientemente explicativos. Como podrá ver el lector, el objeto DBGrid dispone de una colección llamada Columns, que contiene todos los objetos columna del control, por lo que para modificar alguna de las propiedades de una columna, sólo hemos de hacer referencia al número de orden de la columna dentro de la colección, de la misma forma que nos referimos a un elemento dentro de un array, y seguidamente utilizar la propiedad correspondiente.

En cuanto a la manera de aplicar un formato a una columna mediante la propiedad *NumberFormat* del objeto columna, hemos de indicar que la cadena de formato es del mismo tipo que la utilizada en la función Format(), por lo que recomendamos al lector consultar esta función en el tema *El Lenguaje.*

Formato para controles con texto

Como el lector habrá podido comprobar utilizando esta aplicación, los controles para introducir texto no contienen un formato predeterminado, se comprueba si su contenido es correcto al validar la ventana o un conjunto de controles que formen un registro, ya que el control TextBox no está diseñado para admitir formateo del texto que contiene.

El código fuente 545, comprueba los valores del control *txtBaseImponible* del formulario frmFichero cuando se pulsa el botón *Grabar*.

```
Private Sub cmdGrabar_Click()
..........
..........
' comprobar si hay algún contenido en el control
If Len(Me.txtBaseImponible.Text) = 0 Then
   MsgBox "El campo Base Imponible ha de tener valor", , "Error"
   Exit Sub
End If
' comprobar si el contenido es un valor convertible a número
If Not IsNumeric(Me.txtBaseImponible.Text) Then
   MsgBox "El campo Base Imponible no es válido", , "Error"
   Exit Sub
End If
..........
..........
End Sub
```

Código fuente 545

La excepción a esta norma lo constituye el control MaskEdBox, que es un control de aspecto parecido al TextBox, pero que si permite formatear la entrada de caracteres.

Para disponer en el Cuadro de herramientas de este control, hemos de abrir la ventana de Componentes y seleccionar el *Microsoft Masked Edit Control 6.0.* Una vez disponible, insertaremos un control MaskEdBox en la ventana *frmFichero*, al que llamaremos *txtFecha*, para formatear la fecha de los apuntes de IVA.

Figura 548. Icono del control Mask Edit.

Algunas de las propiedades características de este control son las siguientes:

- PromptChar. Especifica el carácter que se va a utilizar para indicar al usuario la posición de edición en la que se encuentra.

- PromptInclude. Valor lógico que indica si la propiedad Text contiene los caracteres utilizados para indicar la posición de edición (PromptChar). Mientras estamos editando el control, normalmente valdrá True, pero cuando necesitemos recuperar el contenido del texto, será necesario ponerlo a False. Más adelante veremos como se realizan estas acciones.

- Mask. Cadena con la máscara de entrada de datos. Consulte la función Format() para ver los diferentes caracteres disponibles.

- Format. Cadena con la expresión de formato: cadena, número, etc.

Con la configuración de este control para que admita fechas, será muy fácil para el usuario introducir las mismas mediante la máscara de entrada establecida en el control, ya que sólo tendrá que teclear la parte numérica sin preocuparse de los signos de separación, puesto que vienen incluidos en el propio control.

Una vez tecleada la fecha, al cambiar el foco de este control a otro, se producirá el evento LostFocus(). Es en el método de este evento donde tendremos que comprobar que la fecha introducida por el usuario es correcta, porque aunque hemos proporcionado una guía para facilitar la introducción del valor de fecha, todavía no podemos tener la absoluta seguridad de que lo introducido por el usuario es válido; podría haber tecleado 44/88/1998, que claramente no es un valor correcto de fecha.

```
Private Sub txtFecha_LostFocus()
Me.txtFecha.PromptInclude = False
If Len(Me.txtFecha.Text) = 0 Then
   Me.txtFecha.Text = mcFecha
End If
Me.txtFecha.PromptInclude = True
ChkFecha Me.txtFecha
End Sub
```

Código fuente 546

En este método deshabilitamos en primer lugar la inclusión del indicador de edición, para comprobar si hay algún valor en su propiedad Text. Si el control está vacío, se le asigna el valor del anterior apunte, y por último se ejecuta un procedimiento que hemos creado nosotros llamado *ChkFecha*, que recibe como parámetro el propio control y comprueba que su valor sea el correcto. Este procedimiento admite un segundo parámetro lógico, que si es True, se encarga de dar el valor de la fecha del sistema al control, en el caso de que el valor tecleado por el usuario no sea correcto; si es False y el valor no es correcto, vacía el contenido del control.

```
Public Sub ChkFecha(rctrFecha As MaskEdBox, _
   Optional rvntLlenarCampo As Variant)
' comprobar que el contenido del control
' pasado como parámetro se puede convertir
' a un valor de fecha

Dim lcPuente As String
Dim lcCaracter As String
Dim lcDia As String
Dim lcMes As String
Dim lcYear As String
Dim lcFecha As String
Dim lbLlenarCampo As Boolean
If VarType(rvntLlenarCampo) <> vbBoolean Then
   lbLlenarCampo = True
Else
   lbLlenarCampo = rvntLlenarCampo
End If
rctrFecha.PromptInclude = False
' si la fecha está vacía,
' poner fecha del sistema
If rctrFecha.Text = "" Then
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   End If

   rctrFecha.PromptInclude = True
   Exit Sub
End If
' si la longitud de la cadena es menor
' de 6 caracteres, no es un valor convertible
' a fecha
If Len(rctrFecha.Text) < 6 Then
   MsgBox "La fecha no es válida", , "Aviso"

   ' rellenar el control con la fecha del sistema
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   Else
      rctrFecha.Text = ""
   End If
   rctrFecha.PromptInclude = True

   Exit Sub
End If
' coger el día y comprobar si es correcto
lcDia = Left(rctrFecha.Text, 2)
If CInt(lcDia) > 31 Then
   MsgBox "El día de la fecha no es correcto", , "Aviso"

   ' rellenar campo fecha con la del sistema
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   Else
```

```
      rctrFecha.Text = ""
   End If
   rctrFecha.PromptInclude = True

   Exit Sub

End If
' coger el mes y comprobar si es correcto
lcMes = Mid(rctrFecha.Text, 3, 2)
If CInt(lcMes) > 12 Then
   MsgBox "El mes de la fecha no es correcto", , "Aviso"

   ' rellenar campo fecha con la del sistema
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   Else
      rctrFecha.Text = ""
   End If
   rctrFecha.PromptInclude = True

   Exit Sub

End If
' coger el año y comprobar si es correcto
lcYear = Mid(rctrFecha.Text, 5)
If Len(lcYear) = 3 Then
   MsgBox "El año de la fecha no es correcto", , "Aviso"

   ' rellenar campo fecha con la del sistema
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   Else
      rctrFecha.Text = ""
   End If
   rctrFecha.PromptInclude = True

   Exit Sub
End If
' si hemos llegado hasta aquí, la fecha es correcta
' pasar cada partícula de la fecha a una variable
' con formato de fecha
lcFecha = lcDia & "/" & lcMes & "/" & lcYear
' si la cadena formada es convertible
' a fecha, asignarla al control
If IsDate(lcFecha) Then
   rctrFecha.Text = Format(lcFecha, rctrFecha.Format)
Else
   ' si la cadena formada no es convertible
   ' a fecha, usar la fecha del sistema
   If lbLlenarCampo Then
      rctrFecha.Text = Format(Date, rctrFecha.Format)
   Else
      rctrFecha.Text = ""
   End If
End If
rctrFecha.PromptInclude = True
End Sub
```

Código Fuente 547

Agilizar la edición de apuntes

El trabajo que ha de resolver este formulario, hace que la rapidez en la introducción de los valores en los controles que forman los registros para la tabla de apuntes de IVA sea uno de los principales problemas a encarar. Si queremos proporcionar al usuario un medio ágil de introducción de datos, no nos sirve el simple cambio de foco entre controles por medio de la tecla Tab. Nosotros proponemos aquí un sistema en el cual, cuando un usuario edita un apunte, ya sea nuevo o para modificar, el foco cambia sólo entre los controles que forman el valor del apunte utilizando para mayor comodidad las teclas Flecha Arriba, Flecha Abajo y Enter. En el caso de que el usuario necesite interactuar con otro control diferente, siempre podrá utilizar el sistema normal, con el ratón por ejemplo.

Para conseguir esta funcionalidad, debemos codificar el método KeyDown(), asociado al evento del mismo nombre, y que se desencadena cuando el usuario pulsa una tecla. El código fuente 548 corresponde al control *txtNombre*.

```
Private Sub txtNombre_KeyDown(KeyCode As Integer, Shift As Integer)
If KeyCode = vbKeyReturn Or KeyCode = vbKeyDown Then
   Me.txtBaseImponible.SetFocus
End If
If KeyCode = vbKeyUp Then
   Me.txtFecha.SetFocus
End If
End Sub
```

Código fuente 548

Este método recibe el parámetro *KeyCode*, que contiene un valor numérico correspondiente a la tecla pulsada. Si el usuario pulsa la tecla [Flecha Abajo] o [Enter], se pasa el foco al siguiente control, y si pulsa [Flecha Arriba] se retrocede al anterior. Los controles que forman el grupo de edición de apuntes de IVA son: *cmdNuevo, txtApteContable, txtFecha, txtNombre, txtBaseImponible, txtBaseExenta, cmdGrabar*.

Aprovechar valores repetitivos

Durante la introducción de apuntes, puede darse el caso de que el usuario tenga que introducir varios registros seguidos con una serie de valores iguales, como la misma fecha, el mismo proveedor, etc. Para este caso, se han definido unas variables a nivel de módulo, que guardan el valor de ciertos campos del registro anteriormente grabado, de forma que cuando el control pierda el foco para pasar al siguiente, si se detecta que el campo está vacío, se utilizará para ese campo, el mismo valor del anterior apunte.

El evento que ocurre cuando un control pierde el foco es LostFocus(), y este es el método que tendremos que codificar para manejar esta situación. En el código fuente 549 vemos las variables definidas para repetir los valores y el método para el control *txtApteContable*.

```
Public mRCDEjercicio As Recordset
Public mcEjercicio As String
Public mcTrimestre As String
Public mcTipoEdit As String
Public mcApteContable As String
```

```
Public mcFecha As String
Public mcNombre As String
Public mcNIF As String
' ------------------------------------
Private Sub txtApteContable_LostFocus()
If Len(Me.txtApteContable.Text) = 0 Then
   Me.txtApteContable.Text = mcApteContable
End If
End Sub
```

Código fuente 549

Grabar un registro

Una vez que el usuario ha completado todos los campos que componen un apunte de IVA, pulsando el botón *Grabar* se salvará el contenido de los controles a la tabla del ejercicio con el que estemos trabajando.

En primer lugar se comprueba que el contenido de los controles concuerde con los valores de los campos a los que se van a pasar.

```
Private Sub cmdGrabar_Click()
..........
..........
Dim llApunte As Long
' comprobar que no haya campos vacios
' y que el tipo de dato coincida
If Len(Me.txtApteContable.Text) = 0 Then
   MsgBox "El campo Apte.Contable ha de tener valor", , "Error"
   Exit Sub
End If
If Not IsDate(Me.txtFecha.Text) Then
   MsgBox "El campo Fecha no es válido", , "Error"
   Exit Sub
End If
If Len(Me.txtNombre.Text) = 0 Then
   MsgBox "El campo Nombre Proveedor ha de tener valor", , "Error"
   Exit Sub
End If
If Len(Me.txtBaseImponible.Text) = 0 Then
   MsgBox "El campo Base Imponible ha de tener valor", , "Error"
   Exit Sub
End If
If Not IsNumeric(Me.txtBaseImponible.Text) Then
   MsgBox "El campo Base Imponible no es válido", , "Error"
   Exit Sub
End If
If Len(Me.txtBaseExenta.Text) = 0 Then
   MsgBox "El campo Base Exenta ha de tener valor", , "Error"
   Exit Sub
End If
If Not IsNumeric(Me.txtBaseExenta.Text) Then
   MsgBox "El campo Base Exenta no es válido", , "Error"
   Exit Sub
End If
If CLng(Me.lblTotalFactura.Caption) = 0 Then
   MsgBox "El total ha de ser mayor de cero", , "Error"
   Exit Sub
End If
```

```
.........
.........
End Sub
```

Código fuente 550

Después se guarda el contenido de los campos que pueden ser repetidos en el próximo apunte.

```
Private Sub cmdGrabar_Click()
.........
.........
' guardar el contenido de algunos campos
' para que en el siguiente apunte, si
' el campo se deja vacío se toma el valor
' del apunte anterior, de forma que se
' creen los apuntes más rápidamente
mcApteContable = Me.txtApteContable.Text
mcFecha = Me.txtFecha.Text
mcNombre = Me.txtNombre.Text
mcNIF = Me.lblNIF.Caption
.........
.........
End Sub
```

Código fuente 551

A continuación, y en función del tipo de edición que estemos realizando al registro, se calculará el número de apunte, y se agregará un nuevo registro al Recordset o se pondrá en modo edición al registro actual.

```
Private Sub cmdGrabar_Click()
.........
.........
Select Case mcTipoEdit
Case "NUEVO"
   ' calcular siguiente nro.apunte
   If Me.mRCDEjercicio.BOF And Me.mRCDEjercicio.EOF Then
      ' si no hay registros en la tabla...
      llApunte = 1
   Else
      ' si hay registros en la tabla...
      Me.mRCDEjercicio.MoveLast
      llApunte = Me.mRCDEjercicio("Apunte")
      llApunte = llApunte + 1
   End If

   ' añadir registro
   Me.mRCDEjercicio.AddNew

Case "MODIFICAR"
   ' el apunte es el mismo
   llApunte = Me.mRCDEjercicio("Apunte")

   ' poner en modo de edición el registro actual
   Me.mRCDEjercicio.Edit
End Select
.........
.........
```

```
End Sub
```

Código fuente 552

Por último volcamos el contenido de los controles al registro del Recordset y lo grabamos en la tabla. Para que los cambios se reflejen en el DBGrid que muestra los apuntes del libro de IVA, hemos de ejecutar el método *Requery* sobre el Recordset del control datIvaSop enlazado al DBGrid, y volvemos a ejecutar el procedimiento *ConfigGrid()*, debido a que el método *Requery* afecta a las columnas del DBGrid haciendo que vuelvan a mostrarse con sus propiedades iniciales.

```
Private Sub cmdGrabar_Click()
..........
..........
' volcar el contenido de variables
' y controles a los campos del registro
Me.mRCDEjercicio("Apunte") = llApunte
Me.mRCDEjercicio("Ejercicio") = mcEjercicio
Me.mRCDEjercicio("Trimestre") = mcTrimestre
Me.mRCDEjercicio("ApteContable") = Me.txtApteContable.Text
Me.mRCDEjercicio("Fecha") = CDate(Me.txtFecha.Text)
Me.mRCDEjercicio("NIF") = Me.lblNIF.Caption
Me.mRCDEjercicio("BaseImponible") = CLng(Me.txtBaseImponible.Text)
Me.mRCDEjercicio("TipoIVA") = CByte(Me.cboTipoIva.Text)
Me.mRCDEjercicio("CuotaIVA") = CLng(Me.lblCuotaIva.Caption)
Me.mRCDEjercicio("BaseExenta") = CLng(Me.txtBaseExenta.Text)
Me.mRCDEjercicio("TotalFactura") = CLng(Me.lblTotalFactura.Caption)
' grabar registro
Me.mRCDEjercicio.Update
' actualizar el data control
' para reflejar los cambios
' en el grid que visualiza los
' apuntes
Me.datIvaSop.Recordset.Requery
LimpiaControles
PreparaControlEdit "FINEDIT"
Me.grdIvaSop.Refresh
Me.datIvaSop.Recordset.MoveLast
ConfigGrid
Me.cmdNuevo.SetFocus
End Sub
```

Código fuente 553

Estilo de la ventana principal

Para esta aplicación hemos creado una ventana MDI *frmInicio*, que hace las veces de contenedora de las otras ventanas de edición de esta aplicación.

Como el número de ventanas no es muy elevado en este programa, podríamos haber suprimido el interfaz MDI y optar por uno SDI, pero con este último estilo corremos el riesgo de dar al usuario una sensación de dispersión entre las ventanas de la aplicación, al no existir un lugar de origen de las mismas, que en nuestro caso es la ventana MDI.

Otra de las ventajas de usar un formulario MDI, es que hemos definido un menú *Ventana* al que en su propiedad *WindowList* hemos dado valor verdadero, de forma que al abrir más de una ventana, si una

de ellas oculta a otra que hayamos abierto, podemos acceder fácilmente a la que está por debajo de la ventana mayor, abriendo este menú y seleccionándola.

VisData, Administrador visual de datos

VisData es una aplicación que ha sido desarrollada con VB, y se incluye como complemento del mismo. El código fuente de esta aplicación se incluye en el directorio de ejemplos y es una estupenda manera de comprobar la potencia de VB en el tratamiento de datos, además de servirnos de ayuda en el manejo de bases de datos no sólo de Access, sino también ISAM y ODBC.

Una vez abierta una base de datos, disponemos de la *Ventana de base de datos* que nos muestra a través de una estructura de colecciones y objetos, el contenido de la base de datos actualmente abierta.

Mediante esta estructura, podemos ir expandiendo cada uno de los objetos que forman la base de datos para visualizar y editar los valores de sus propiedades.

Para manipular los datos de las tablas, disponemos de la ventana *Instrucción SQL*, en la que escribiremos la sentencia SQL a ejecutar, mostrándose a continuación en una ventana de resultados.

Esta ventana de resultados no tiene un comportamiento estático, muy al contrario, los botones que dispone en su parte superior, nos permiten realizar varias acciones sobre los registros mostrados, como indican los propios nombres de estos botones.

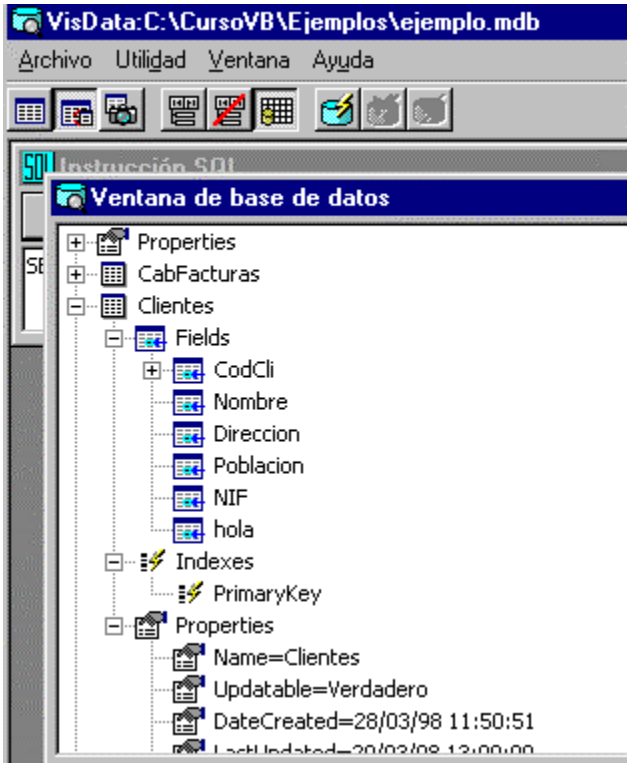


Figura 549. Ventana de base de datos de VisData.

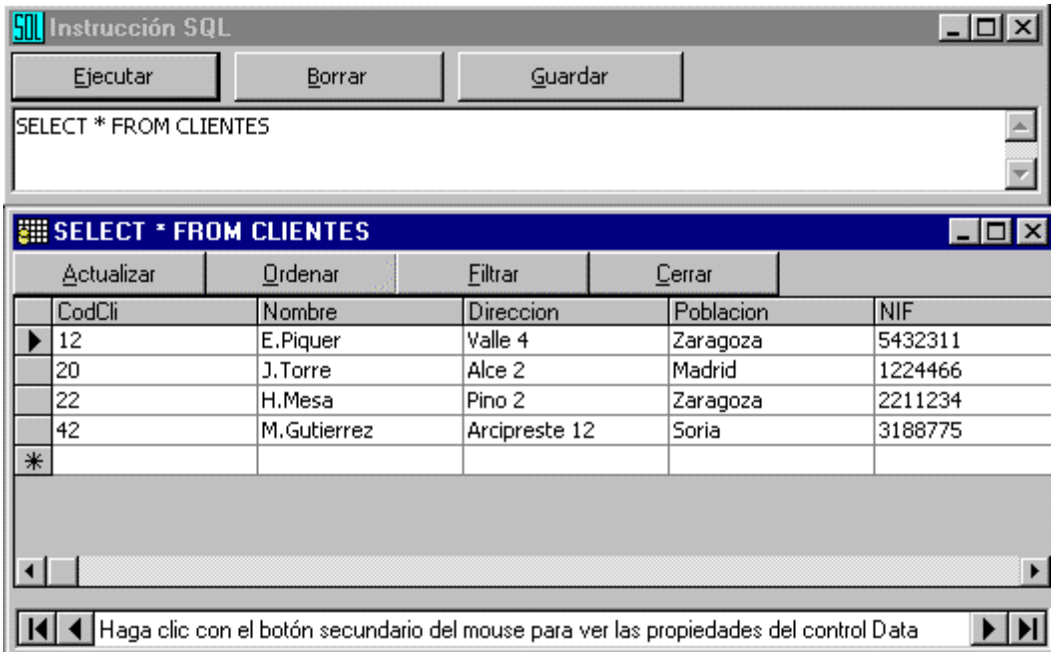


Figura 550. Ventanas de Instrucción SQL y resultados de la consulta.

La forma más cómoda de realizar una consulta y ver su resultado es la que acabamos de mostrar. Pero si el usuario lo requiere, también es posible mostrar el resultado en forma de ventana de edición, registro a registro, sólo tiene que seleccionar en la barra de herramientas de VisData, el modo de presentación del resultado de la consulta.

 . La ventana de resultados usará un control Data.

 . La ventana de resultados no usará un control Data.

 . La ventana de resultados usará un control DBGrid para presentar los datos.

Si pulsamos con el botón derecho en el control Data de la ventana de resultados, obtendremos una ventana con las propiedades del mismo, como vemos en la figura 551.

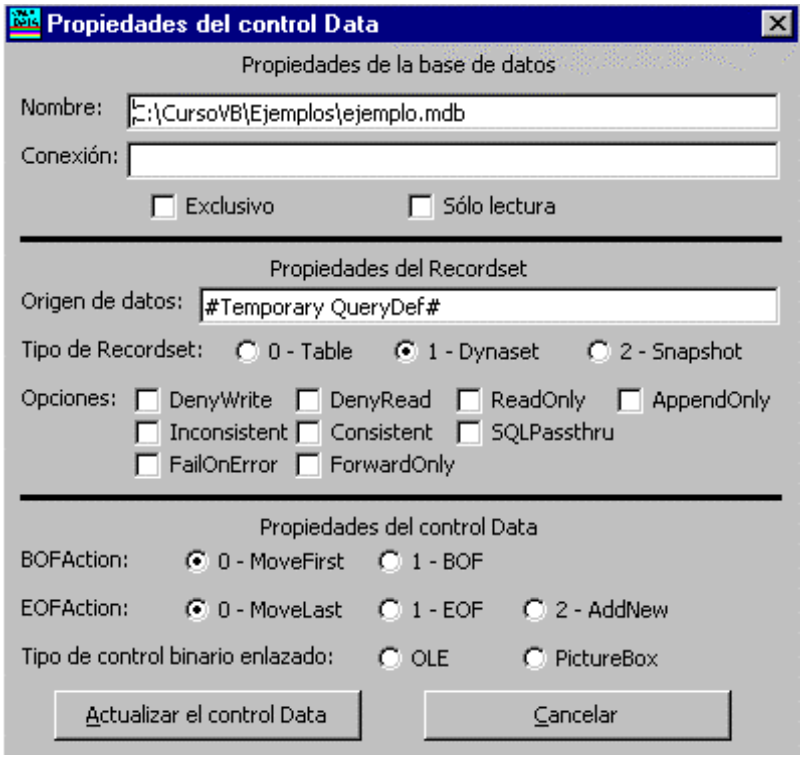


Figura 551. Propiedades del control Data de la ventana de resultados de VisData.

Otra de las útiles herramientas proporcionadas por VisData es un *Generador de Consultas*, que permite crear complejas consultas muy fácilmente, con la posibilidad incluso de guardarlas dentro de la base de datos en forma de QueryDef.

Para acceder a esta utilidad hemos de seleccionar el menú *Utilidad+Generador de Consultas* (figura 552).

Para terminar este breve repaso a VisData, comentaremos otra de las utilidades que proporciona. Se trata de un eficaz Diseñador de formularios de datos (figura 553), que nos permite seleccionar una tabla de la base de datos, los campos sobre los que deseamos realizar el mantenimiento de datos, y después de proporcionar un nombre para el formulario, este es generado e incluido en nuestro proyecto.

El formulario creado, se basa en un mantenimiento de datos típico, que incluye el código necesario para las operaciones más habituales de edición de registros de una tabla, y aunque siempre tendremos que retocarlo para adaptarlo a nuestras necesidades, hay que reconocer que supone una gran ayuda en mantenimientos básicos. Este tipo de características son las que proporcionan a VB la categoría de herramienta RAD tan comentada en esta versión.

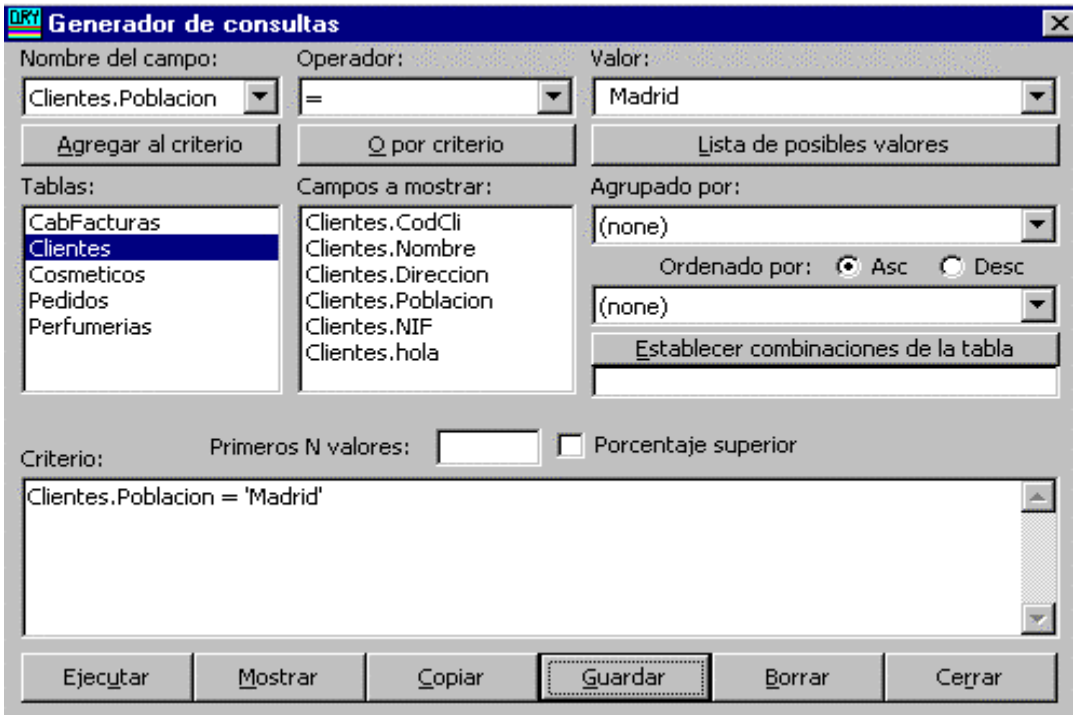


Figura 552. Generador de Consultas de VisData.

Figura 553. Diseñador de formularios de datos de VisData.

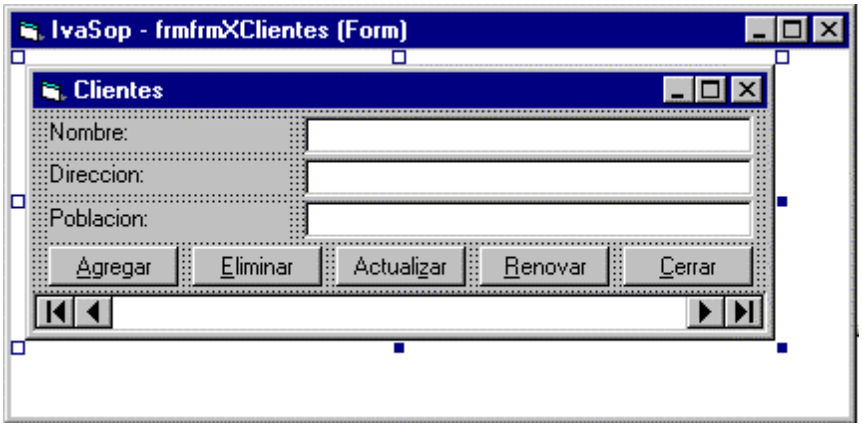


Figura 554. Formulario creado con el generador de formularios de VisData