

DELPHI – PROGRAMACIÓN ORIENTADA A OBJETOS. ENTORNO VISUAL



Miguel Rodríguez Penabad
2006

| | | |
|----------|--|-----------|
| 1 | INTRODUCCIÓN..... | 3 |
| 1.1 | CARACTERÍSTICAS PRINCIPALES DE DELPHI | 3 |
| 1.2 | EL ENTORNO DE DESARROLLO..... | 4 |
| 1.2.1 | La paleta de componentes..... | 5 |
| 1.2.2 | El inspector de objetos (<i>Object Inspector</i>) y <i>Object TreeView</i> | 5 |
| 1.2.3 | Gestor de Proyectos (<i>Project Manager</i>)..... | 6 |
| 1.2.4 | Formularios | 6 |
| 1.3 | PERSONALIZACIÓN DEL ENTORNO | 7 |
| 2 | PROYECTOS..... | 9 |
| 2.1 | FICHEROS DE UN PROYECTO DELPHI..... | 9 |
| 2.2 | EL PRIMER PROYECTO DELPHI..... | 9 |
| 2.2.1 | Crear un Nuevo proyecto..... | 10 |
| 2.2.2 | Añadir los componentes al formulario..... | 10 |
| 2.2.3 | Modificar las propiedades de los componentes | 11 |
| 2.2.4 | Añadir manejadores de eventos | 12 |
| 2.2.5 | Compilar y ejecutar la aplicación..... | 12 |
| 2.3 | PROPIEDADES DE UN PROYECTO | 13 |
| 3 | PROGRAMACIÓN CON COMPONENTES.. | 14 |
| 3.1 | DESCRIPCIÓN GENERAL DE LOS COMPONENTES | 14 |
| 3.2 | PROPIEDADES, MÉTODOS Y EVENTOS MÁS IMPORTANTES..... | 14 |
| 3.2.1 | Propiedades, métodos y eventos comunes a todos los componentes..... | 14 |
| 3.2.2 | Propiedades, métodos y eventos de los controles | 15 |
| 3.3 | EJEMPLOS | 16 |
| 3.3.1 | MiniCalculadora..... | 16 |
| 3.3.2 | Editor de texto simple | 18 |
| 4 | TRABAJANDO CON MÁS DE UN FORMULARIO..... | 19 |
| 4.1 | CUADROS DE DIÁLOGO PREDEFINIDOS..... | 19 |
| 4.1.1 | ShowMessage..... | 19 |
| 4.1.2 | MessageDlg..... | 19 |
| 4.1.3 | Application.MessageBox..... | 20 |
| 4.1.4 | InputDialog e InputQuery..... | 20 |
| 4.1.5 | OpenDialog y SaveDialog..... | 20 |
| 4.2 | VARIOS FORMULARIOS | 20 |
| 4.2.1 | Introducción..... | 20 |
| 4.2.2 | Creación y destrucción de Ventanas. La ventana principal de la aplicación. | 22 |
| 4.2.3 | Ventanas modales y no modales..... | 24 |
| 4.2.4 | Aplicaciones MDI y SDI..... | 25 |
| 5 | APLICACIONES DE BASES DE DATOS | 27 |
| 5.1 | COMPONENTES DE ACCESO A BASES DE DATOS | 27 |
| 5.1.1 | La capa de acceso a datos | 28 |
| 5.1.2 | TDataSource y Controles de Datos..... | 35 |
| 5.2 | CONSULTAS..... | 35 |
| 5.2.1 | Selección de datos..... | 36 |
| 5.2.2 | Modificación de datos..... | 36 |
| 5.2.3 | Campos y consultas con parámetros..... | 36 |
| 5.3 | EVENTOS | 37 |
| 5.3.1 | Eventos del DataSource | 37 |
| 5.3.2 | Eventos de DataSet | 38 |
| 5.4 | FORMULARIOS MAESTRO-DETALLE..... | 38 |

| | | |
|----------|--|-----------|
| 5.4.1 | <i>Maestro-Detalle con tablas</i> | 39 |
| 5.4.2 | <i>Maestro-Detalle con Consultas</i> | 40 |
| 6 | LISTADOS | 41 |
| 6.1 | EJEMPLO..... | 42 |
| 7 | MANEJO DE EXCEPCIONES | 44 |
| 8 | PROGRAMAS DE EJEMPLO..... | 46 |
| 8.1 | PROGRAMA ‘BIENVENIDA’ | 46 |
| 8.1.1 | <i>Bienvenida.DPR</i> | 46 |
| 8.1.2 | <i>Formulario Principal</i> | 46 |
| 8.2 | MINICALCULADORA (VERSIÓN 1, MINICALC1) | 48 |
| 8.2.1 | <i>MiniCalc1.DPR</i> | 48 |
| 8.2.2 | <i>Formulario Principal</i> | 48 |
| 8.3 | MINICALCULADORA (VERSIÓN 2, MINICALC2) | 51 |
| 8.3.1 | <i>Formulario principal</i> | 51 |
| 8.4 | EDITOR DE TEXTOS | 52 |
| 8.4.1 | <i>Editor1.DPR</i> | 52 |
| 8.4.2 | <i>Formulario principal</i> | 52 |
| 8.5 | AGENDA | 55 |
| 8.5.1 | <i>Agenda.DPR</i> | 55 |
| 8.5.2 | <i>Formulario Principal</i> | 55 |

1 Introducción

1.1 Características principales de Delphi

Delphi es un Entorno Integrado de Desarrollo (IDE: Integrated Development Environment) que permite:

- Programación Visual
- Programación Orientada a Objetos (utilizando Object Pascal)
- Acceso a bases de datos, tanto locales como remotas
- Generación de ejecutables (.EXE, sin necesidad de librerías *runtime*) o bibliotecas de enlace dinámico (DLLs)

La Figura 1 muestra el entorno Delphi 6 en ejecución.

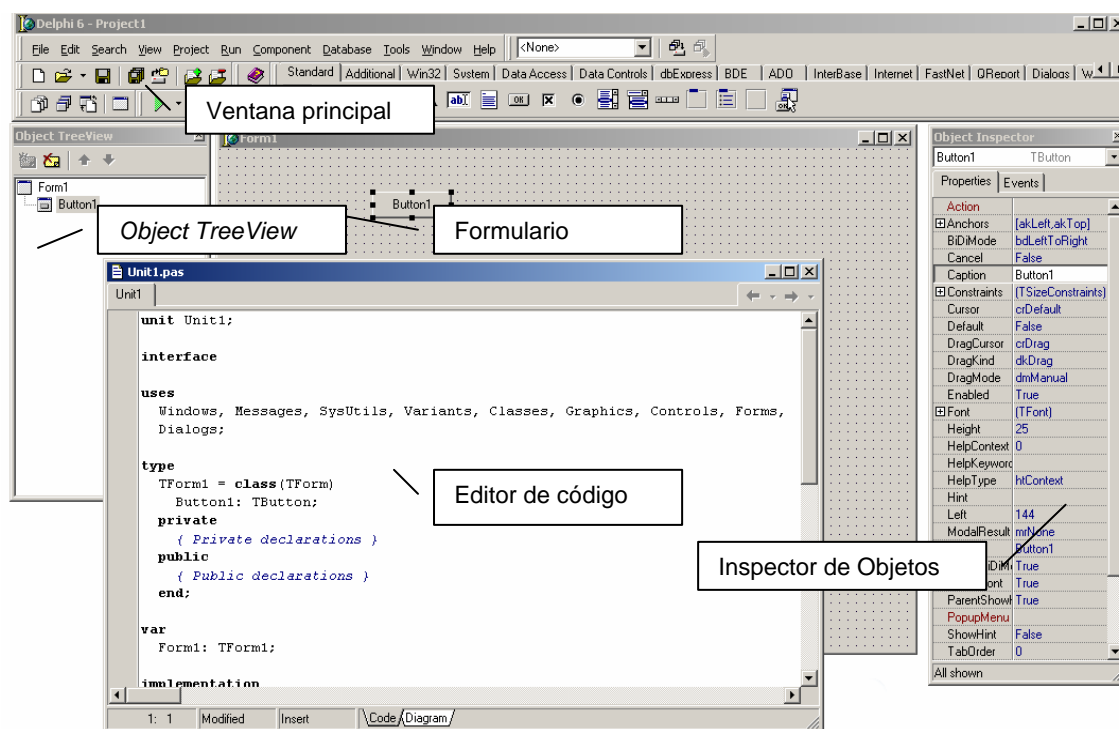


Figura 1. Delphi 6

Delphi es una herramienta demasiado compleja para verla completa en un curso de este tipo. Por ello, nos centraremos sólo en alguna de las posibilidades que nos ofrece. Así, por ejemplo:

- Delphi permite generar distintos tipos de aplicaciones, como aplicaciones “normales” de Windows, aplicaciones con interfaz MS-DOS, DLLs, módulos y aplicaciones para el Panel de Control de Windows, etc., como se ve en la Figura 2. Aquí nos centraremos sólo en la creación de aplicaciones Windows “normales”, lo cual quiere decir que será un aplicación con una ventana (o más) estándar en Windows.
- A partir de la versión 6, Delphi permite usar dos conjuntos diferentes (y mutuamente exclusivos) de componentes. Uno de estos conjuntos es la

- biblioteca VCL (Visual Component Library), que había sido usado en todas las versiones anteriores de Delphi. La otra biblioteca, que no veremos, se denomina CLX y es usada tanto por Delphi como por Kylix, un entorno de desarrollo orientado a Linux.
- Delphi permite utilizar otro tipo de componentes estándar en Windows, como son los componentes VBX y OCX. Dado que no son componentes nativos y que (y esto es una apreciación personal del autor) no dan un resultado óptimo y sí causan problemas, no los veremos.

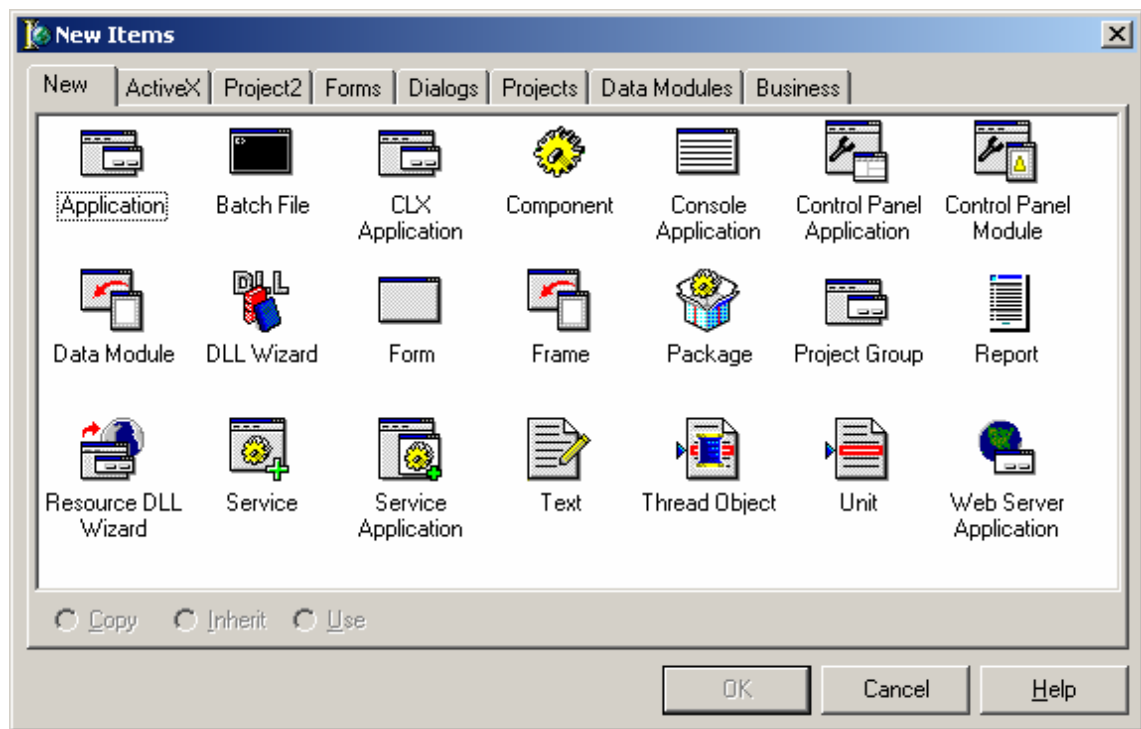


Figura 2. Tipos de proyectos que se pueden crear con Delphi

1.2 El Entorno de Desarrollo

Veamos las ventanas principales que encontramos en Delphi. La ventana en la parte superior de la Figura 1 es la ventana principal de Delphi, que se muestra también en la Figura 3. En esta ventana podemos ver el menú, los botones de acceso rápido (a la izquierda) y la **Paleta de Componentes**, que se verá a continuación.

Además, normalmente veremos el Inspector de Objetos, un Formulario y el Editor de Código.



Figura 3. Ventana principal de Delphi

1.2.1 La paleta de componentes

La paleta de componentes nos da acceso a los componentes, que pueden ser visuales (llamados controles) o no visuales, y que implementan las funciones básicas de interfaz de usuario, acceso a bases de datos, etc.

Ejemplos de controles son botones, campos de edición, listas desplegables o paneles, mientras que los componentes de acceso a bases de datos o los temporizadores son componentes no visuales.

Delphi proporciona una librería, denominada VCL (Visual Component Library) con una gran variedad de componentes predefinidos, que están organizados y agrupados funcionalmente en la paleta de componentes. Pulsando en la pestaña adecuada de la paleta accedemos a un grupo de componentes. Para incluir un componente en nuestra aplicación, simplemente seleccionamos el componente y al hacer clic en el formulario de nuestra aplicación se añade a él (en la posición en donde pulsamos).

La paleta de componentes es también configurable: podemos quitar, poner o recolocar componentes. Para ello, pulsamos con el botón derecho del ratón en la paleta, y seleccionamos la opción *Properties* del menú contextual que aparece.

1.2.2 El inspector de objetos (Object Inspector) y Object TreeView

El Inspector de Objetos (a la izquierda en la Figura 4) nos permite ver y modificar las propiedades de los componentes (incluidos los formularios). También nos permite asociar *event handlers* (manejadores de eventos) para estos componentes.

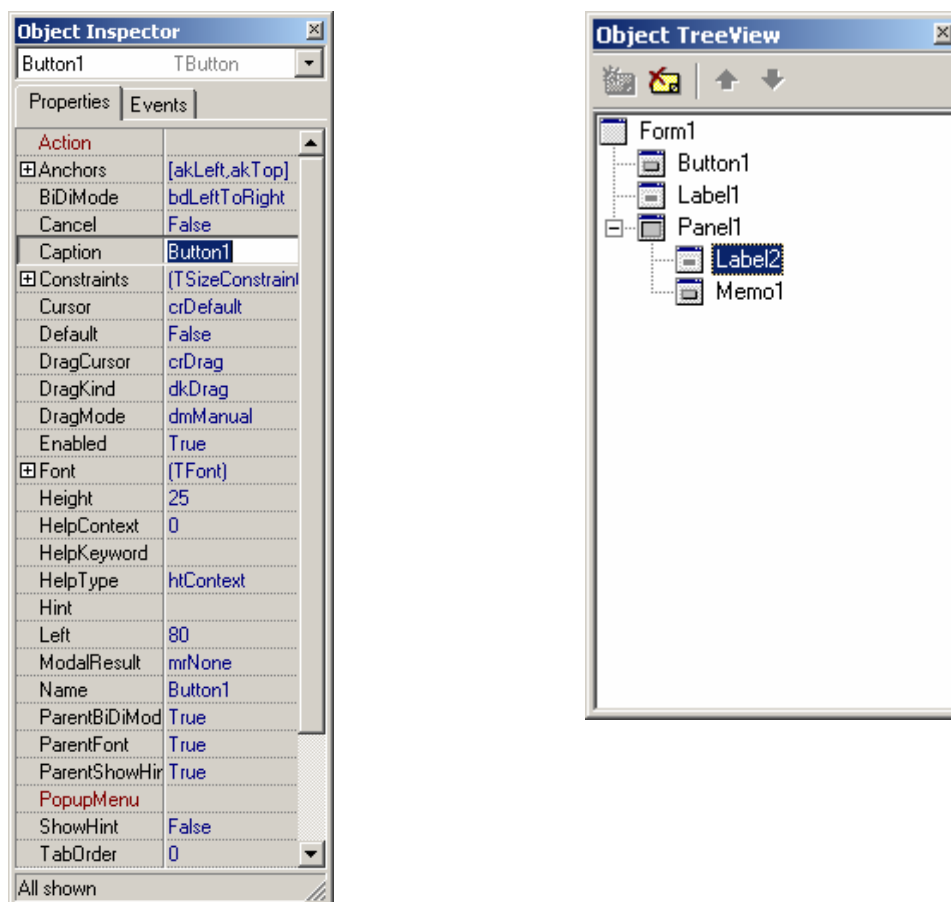


Figura 4. Object Inspector y Object TreeView

Podemos acceder al Inspector de Objetos pulsando F11 (o usando la opción de menú *View/Object Inspector*), o pulsando sobre él en caso de que esté visible. En la parte superior del mismo se ve nombre del componente seleccionado y su tipo.

El *Object TreeView*, que se muestra a la derecha en la misma figura, aparece como novedad en la versión 6 de Delphi. Muestra el formulario y sus componentes en forma de árbol (existen controles, como los paneles, que pueden a su vez contener otros componentes). Esta herramienta es muy útil para acceder de forma rápida a componentes que están ocultos (por ejemplo, si tenemos un panel encima de otro panel, los controles que estén en el primero no se verán en el formulario).

1.2.3 Gestor de Proyectos (Project Manager)

Como se ha dicho, cada aplicación Delphi es un proyecto. Cada proyecto estará formado por varios ficheros (units, formularios, ...). Para poder acceder a cada uno de estos ficheros es útil el uso del gestor de proyectos (Project Manager), al que se puede acceder mediante la opción de menú *View/Project Manager*, o pulsando las teclas Ctrl+Alt+F11.

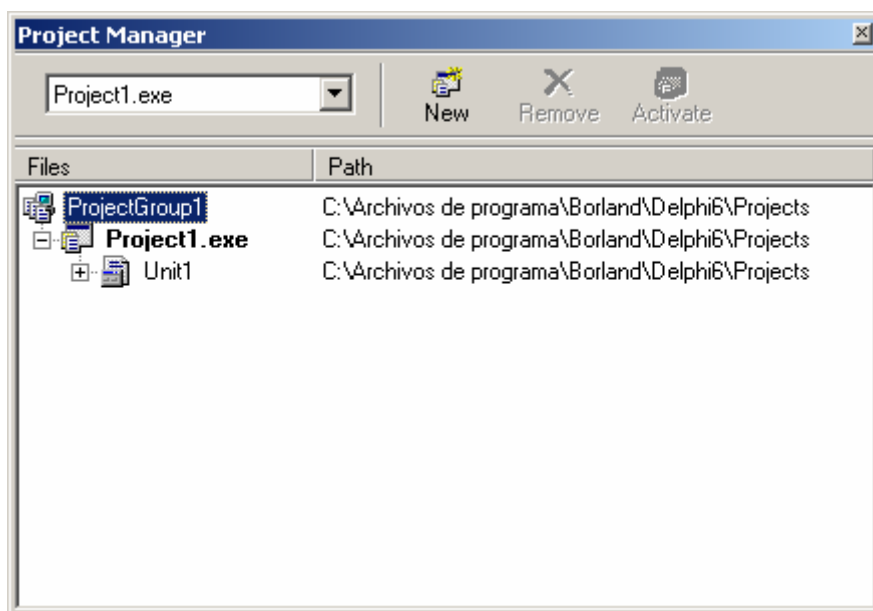


Figura 5. Gestor de proyectos

1.2.4 Formularios

El formulario o *Form* es el componente principal del desarrollo de aplicaciones Windows en Delphi. Durante la fase de diseño, se creará un formulario para cada ventana que se necesite. Los controles que configuran cada formulario se colocarán sobre el mismo.

Cada formulario se almacena en un fichero con extensión DFM, y tendrá asociada una unit con el mismo nombre y extensión .PAS. En esta unit se almacena tanto la definición de la clase del formulario como sus componentes, y el código correspondiente a los manejadores de eventos (y otras funciones y procedimientos que se creen) de los componentes del formulario.

1.3 Personalización del entorno

Accediendo a la opción de menú *Tools/Environment Options*, se accede a la ventana que se muestra en la Figura 6, donde se pueden personalizar diversos aspectos del entorno de Delphi. Puede consultar la ayuda de Delphi para ver el significado de cada opción, de las que podemos destacar las “*Autosave options*”: Si se activa la opción *Editor Files*, se grabarán los ficheros del proyecto antes de ejecutarlo, y si se activa *Project Desktop*, se grabará el proyecto actual, que se abrirá automáticamente al abrir de nuevo Delphi. La primera opción es especialmente interesante, ya que si ejecutamos el programa que estamos desarrollando y esto provoca que Delphi (o Windows) se “cuelgue”, los ficheros ya habrán sido grabados y no perderemos el trabajo realizado.

En la pestaña *Library* de esta ventana hay una serie de opciones que se verán más adelante, y que serán especialmente importantes si queremos incluir componentes o paquetes que no vienen por defecto con Delphi, como componentes desarrollados por terceras partes, o componentes desarrollados por nosotros mismos.

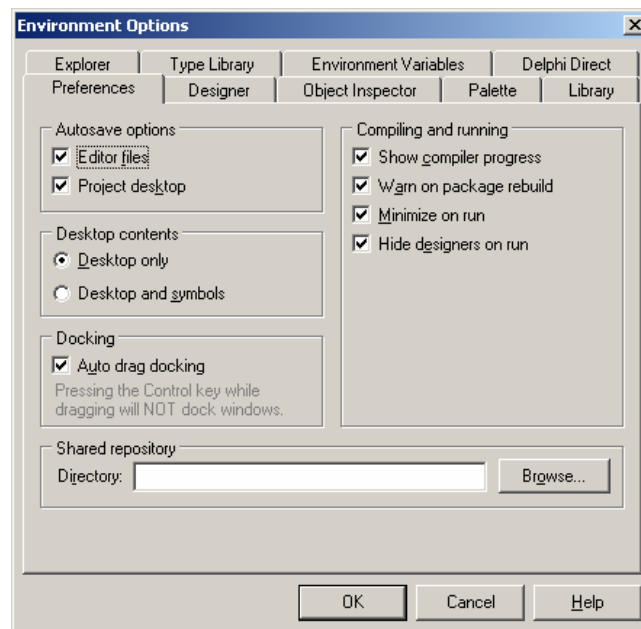


Figura 6. Personalización del entorno de Delphi

La Figura 7 nos muestra cómo podemos cambiar el comportamiento y aspecto visual del editor de código de Delphi. A esta ventana se accede mediante la opción del menú *Tools/Editor options*.

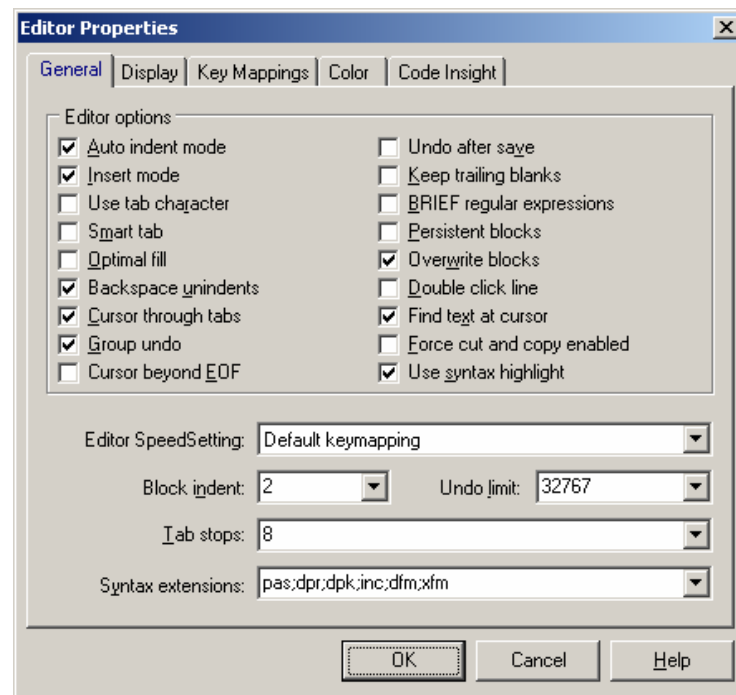


Figura 7. Propiedades del editor de código

2 Proyectos

2.1 Ficheros de un proyecto Delphi

El desarrollo de aplicaciones en Delphi se lleva a cabo mediante *Proyectos*, cada uno de los cuales da lugar a un programa (o a una DLL, a una aplicación del Panel de Control, etc.). Además, estos proyectos se pueden agrupar en grupos de proyectos (*Borland Project Groups*). En muchos casos trabajaremos con proyectos independientes, para los que Delphi crea un grupo con un único proyecto.

Lo más recomendable es crear cada proyecto en un directorio separado, de forma que no tengamos conflictos con ficheros con el mismo nombre.

Al trabajar con Delphi, creamos ficheros de distintos tipos, y además Delphi creará, al compilar los ficheros, otros ficheros adicionales. El tipo de fichero se distingue por su extensión, y los principales tipos de ficheros que aparecen (bien creados por nosotros o bien generados por Delphi al compilar el proyecto) son los siguientes:

| Ficheros creados por nosotros | |
|-------------------------------|---|
| Extensión | Significado |
| BPG | Borland Project Group. Fichero que nos permite agrupar varios proyectos |
| DPR | Delphi PProject: Proyecto de Delphi, que dará lugar a una aplicación o DLL con el mismo nombre |
| DFM | Delphi Form: Formulario o ventana, en la que podremos trabajar de forma visual añadiendo componentes |
| PAS | Fichero Pascal: En estos ficheros se define |
| Ficheros generados por Delphi | |
| DCU | Delphi Compiled Unit: Resultado de compilar una unit, escrita en un fichero .PAS y a la que opcionalmente está asociado un formulario (DFM) |
| EXE | Ejecutable que resulta de compilar un proyecto (application) |
| DLL | Biblioteca de Enlace Dinámico, que resulta de compilar un proyecto (Library) |

Existen otros tipos de ficheros que se crean cuando compilamos un proyecto Delphi, pero los citados en la tabla anterior son los más importantes.

2.2 El primer proyecto Delphi

Vamos a crear una aplicación Windows que nos sirva para ver la forma básica de desarrollar aplicaciones en Delphi. El programa constará de una ventana con un botón y una etiqueta de texto. Al pulsar el botón, la etiqueta mostrará el texto “Bienvenido/a a Delphi”.

Los pasos a seguir serán los siguientes:

1. Crear un nuevo proyecto (nueva aplicación).
2. En el formulario que aparece, colocar los componentes necesarios.
3. Modificar las propiedades de los componentes.
4. Añadir el código necesario para los manejadores de eventos necesarios.

No es necesario realizar los tres últimos pasos de forma secuencial, sino que por ejemplo podemos colocar un componente, modificar sus manejadores de eventos, luego sus propiedades, y pasar al siguiente componente.

2.2.1 Crear un Nuevo proyecto

Accediendo a la opción de menú *File/New/Application*, se crea un nuevo proyecto, que tendrá como nombre por defecto *Project1.DPR*. De forma automática se crea para el proyecto el formulario principal (*Unit1.DFM*), con la unit *Unit1.PAS*.

Es importante saber que el nombre del fichero DPR será el mismo que el del ejecutable generado. En este caso sería *Project1.EXE*.

Grabemos el proyecto. Es conveniente crear una nueva carpeta para cada proyecto, y darle un nombre significativo. Por ejemplo, grabaremos este proyecto en la carpeta *C:\DELPHI\BIENVENIDA* (ver Programa 8.1), renombrándolo como *Bienvenida.DPR*. Asimismo, cambiaremos el nombre de *Unit1.pas* a *Principal.pas*. Con ello, podemos ver los ficheros creados en el gestor de proyectos.

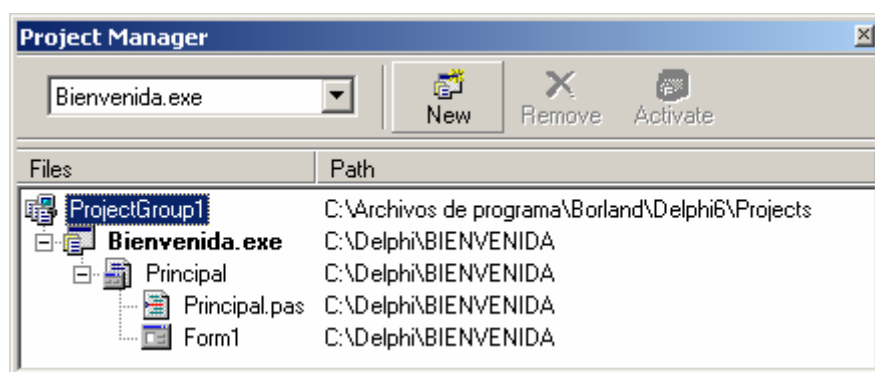


Figura 8. Ficheros del primer proyecto Delphi.

2.2.2 Añadir los componentes al formulario

Añadiremos al formulario del proyecto dos componentes: un botón y una etiqueta de texto. Es decir, un control *TButton* y un control *TLabel*. Ambos controles están en la pestaña “Standard” de la paleta de componentes. Si desplazamos el cursor del ratón sobre los componentes de la paleta, aparecerá un pequeño globo de ayuda (*hint* en la terminología de Delphi) con el nombre del componente. El resultado podría ser el que se muestra en la Figura 9.

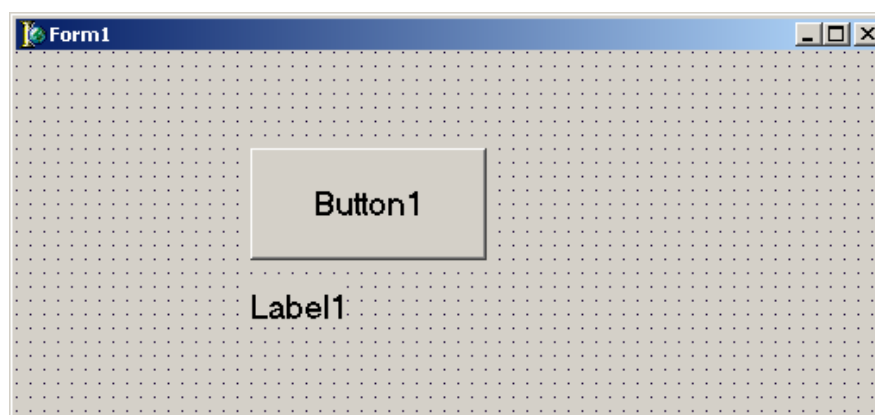


Figura 9. Un formulario

2.2.3 Modificar las propiedades de los componentes

Cuando se crea un nuevo elemento, ya sea este un proyecto, un formulario, o cualquier componente, este tendrá una serie de valores por defecto. Entre estos valores será el nombre.

Cada componente pertenece a una clase, que por convención empieza siempre por la letra T (TForm es la clase de los formularios, TButton la clase de los botones, etc.). Todos los componentes tienen una propiedad *Name*, que especifica el nombre del componente. El nombre por defecto del componente creado será el nombre de la clase sin la T seguido de un número correlativo. Es decir, si añadimos varios botones a un formulario, estos se llamarán Button1, Button2, etc. Es una buena idea dar nombres significativos a los elementos, para que luego al referirnos a ellos el código sea más comprensible.

En este caso, cambiaremos la propiedad *Name* del formulario y de sus controles, llamándoles FPrincipal, Boton, y Etiqueta. Esto lo haremos usando el inspector de objetos, como se ve en la Figura 10.

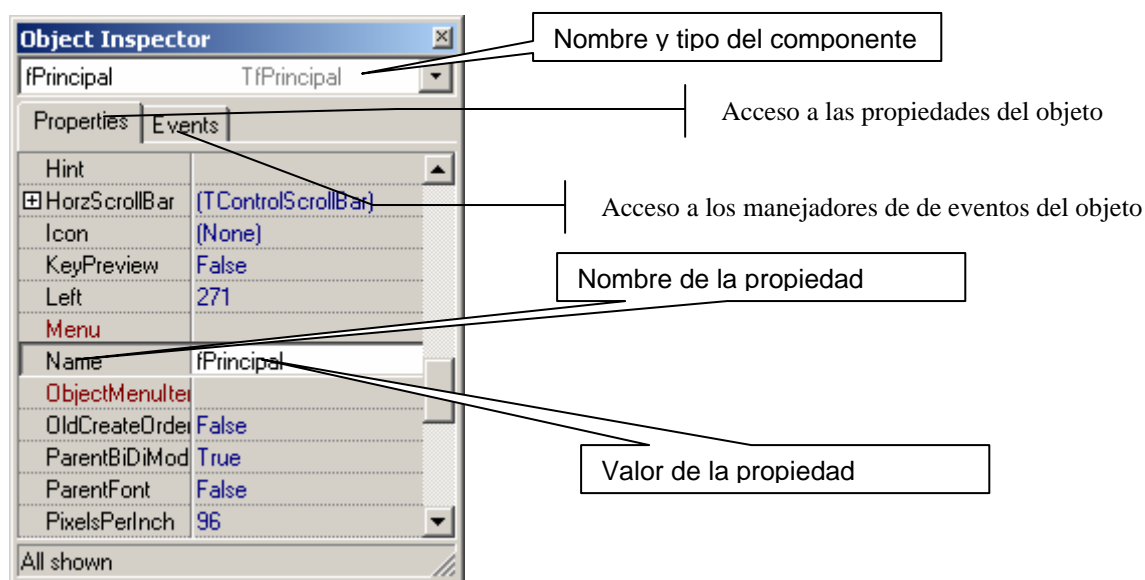


Figura 10. Modificación de propiedades de un componente

Existen 3 formas básicas para cambiar los valores de las propiedades de un componente (no siempre pueden ser aplicadas las 3):

- Usando el inspector de objetos
- De forma visual (si el componente es visual). Propiedades como la posición (propiedades Top y Left) y el tamaño (Altura: Height, y Anchura: Width) pueden ser cambiadas directamente en el formulario. Por ejemplo, si redimensionamos el formulario (pulsando en el borde de la ventana y arrastrando el ratón, como en cualquier programa Windows), los valores Height y Width del inspector de objetos reflejarán los nuevos valores.
- Por programa: Usando código ObjectPascal, podemos asignar un valor a una propiedad, cambiando dinámicamente el valor de la propiedad del objeto. Así, podremos tener en algún lugar del programa la asignación `fPrincipal.Caption:='Ventana Principal'`, que cambia el título del formulario.

2.2.4 Añadir manejadores de eventos

Usando el Inspector de Objetos, en la pestaña *Events*, se puede añadir el código (en ObjectPascal) que define cómo un objeto (componente) reaccionará cuando se produzca determinado evento. Como se muestra en la Figura 11, el evento considerado es *OnClick* del componente de tipo *TButton*, que se produce cuando se hace clic con el ratón en el botón. Al hacer doble clic en la casilla a la derecha del evento deseado en el inspector de objetos, se crea un nuevo procedimiento en la unit asociada al formulario (si no existía ya), y el cursor se sitúa en el cuerpo del procedimiento, para que podamos empezar a codificar. El nombre por defecto del procedimiento es el del componente seguido del evento sin el prefijo *On*, en este caso *BotonClick*.

Ahí podemos hacer que el texto (Caption) de la etiqueta cambie, mostrando el mensaje deseado.

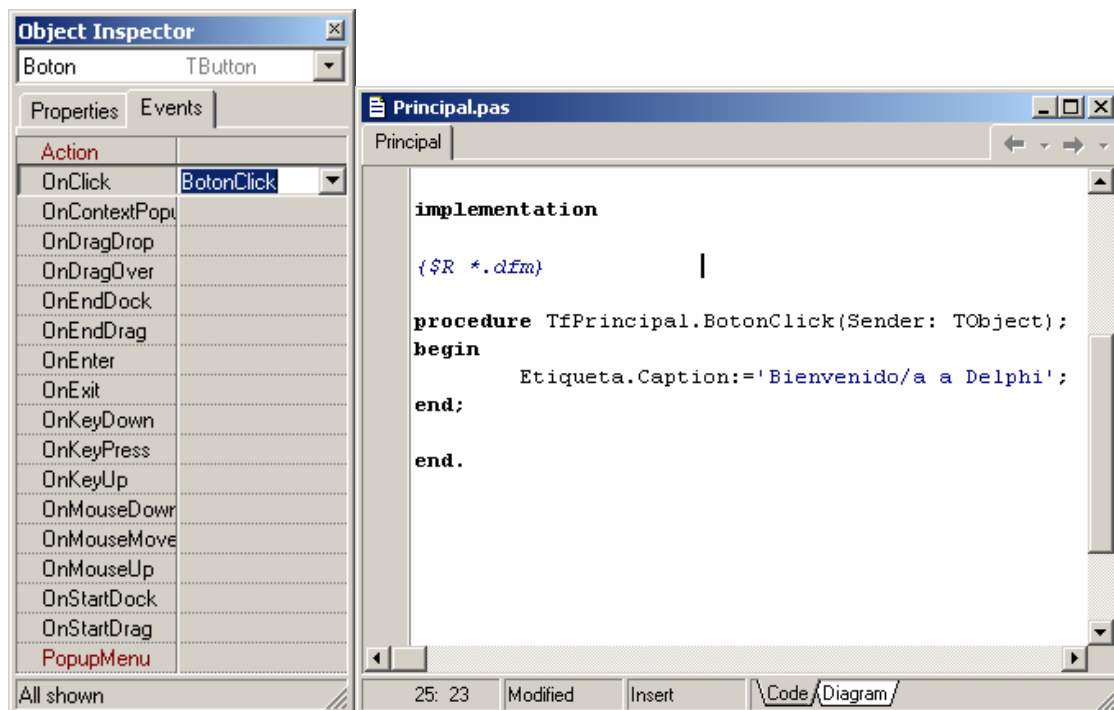


Figura 11. Añadir un manejador de eventos

2.2.5 Compilar y ejecutar la aplicación

Para **compilar** el proyecto, podemos pulsar **Ctrl+F9**, o bien ir a la opción del menú *Project/Compile <nombre del proyecto>*. Esto hace que se compilen las units que se hayan modificado, se enlacen las bibliotecas necesarias (proceso de enlazado o linkado) y se cree el ejecutable final.

La compilación de Delphi es incremental, esto es, sólo compilará aquello que haya cambiado. A veces Delphi no interpreta bien qué ha cambiado y puede que no compile correctamente el programa. Si esto sucede, deberemos compilarlo todo, accediendo a la opción de menú *Project/Build <nombre del proyecto>* o *Project/Build all projects*. Esto nos garantiza que todas las units serán compiladas de nuevo.

Para **ejecutar** el programa podemos hacerlo desde dentro del entorno IDE de Delphi o fuera de él, como una aplicación más de Windows. Si lo hacemos desde dentro, podremos depurar la aplicación haciendo uso del *debugger* o depurador integrado que ofrece Delphi. Se ejecuta el proyecto actual pulsando F9, usando la opción *Run/Run*, o pulsando el botón con la flecha verde en la barra de botones.

2.3 Propiedades de un proyecto

El proyecto, la aplicación desarrollada, también tiene una serie de características o propiedades que pueden ser modificadas, desde la opción de menú *Project/Options*. Entre ellas, podemos cambiar (en la pestaña *Application*) el título o nombre de la aplicación (si no se establece, se usará el nombre del ejecutable como título), el fichero de ayuda asociado, y el icono de la misma, como se ve en la Figura 12.

En la pestaña *Directories/Conditionals* podemos cambiar algunos directorios por defecto. Por ejemplo, si queremos que las units compiladas (ficheros .DCU) no se guarden con los fuentes, sino en un directorio aparte, lo especificaremos en la opción *Unit Output Directory*.

Existen opciones más avanzadas, que establecen opciones de compilación y enlazado, en las que no entraremos.

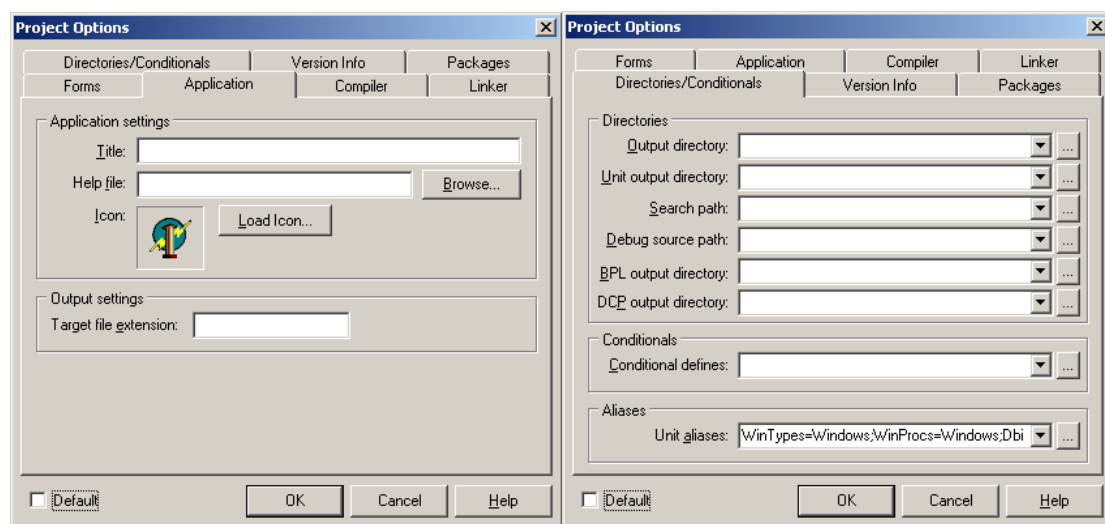


Figura 12. Propiedades del proyecto

3 Programación con componentes

3.1 Descripción general de los componentes

Casi todo el trabajo en Delphi se realiza usando componentes. Por ello, vamos a ver qué es realmente un componente, y a qué partes de él tenemos acceso.

Un **componente** es un objeto, que pertenece a una clase, en la que se definirán los atributos y métodos asociados a la misma. En un componente distinguimos básicamente cuatro partes, algunas de las cuales serán fácilmente reconocibles para el lector que tenga conocimientos básicos de Programación Orientada a Objetos:

- **Privada:** Son atributos (campos o variables del objeto) y métodos (procedimientos y funciones) privadas al componente. Es decir, no podemos acceder a ellos desde “fuera” del componente (sólo estarán disponibles en la implementación del mismo).
- **Protegida:** Es una parte que será accesible por la clase y las clases derivadas de ella, y no accesible para todas las demás clases. Es útil en la creación de componentes, para definir una clase genérica y luego derivar otras clases.
- **Pública:** Serán atributos y métodos accesibles desde la clase y desde fuera de ella. Podremos acceder “por programa” a los atributos y métodos públicos de todos los componentes en los programas que realicemos (es decir, cuando estemos codificando, no desde el inspector de objetos).
- **Publicada:** Está formada por *propiedades* (*properties*), que son un tipo especial de atributos de los componentes. Es una parte pública, y además todas las propiedades publicadas aparecerán en el Inspector de Objetos, en la pestaña de *Properties* o la de *Events*, dependiendo de su naturaleza. Además, también serán accesibles por programa.

Dado que inicialmente vamos a trabajar con componentes, y no crearlos nuevos, nos centraremos en aquellas partes que vamos a poder usar, es decir, las partes pública y publicada de los componentes. Como ya se ha dicho, la parte publicada aparece en el inspector de objetos, y podremos acceder a ella desde él (o desde programa). La parte pública es accesible únicamente desde el código de nuestra aplicación.

3.2 Propiedades, métodos y eventos más importantes

Cada componente tiene un conjunto de propiedades y eventos particulares, pero existen algunas propiedades comunes a muchos componentes, que se verán a continuación. Será una relación muy sucinta, ya que examinar cada una de las propiedades de cada control excedería en mucho el cometido de este documento. Consulte la ayuda en línea de Delphi para obtener información específica de un componente, o el inspector de objetos (sólo para ver las propiedades o eventos) seleccionando un objeto concreto.

3.2.1 Propiedades, métodos y eventos comunes a todos los componentes

- **Propiedades:**
Name: Es el nombre del componente

Tag: Es un entero, que no tiene un uso específico.

- Métodos:

Create: Crea el componente. Puede o no tener un parámetro, el objeto “padre”

Release/Free/Destroy: Destruye el componente. El método Destroy no es muy recomendado. Para los formularios se recomienda Release, y para muchos otros componentes, Free.

- Eventos:

OnCreate: Se produce cuando se crea el componente.

OnDestroy: Se produce cuando se destruye (de hecho, justo antes de destruir) el componente.

3.2.2 Propiedades, métodos y eventos de los controles

De los controles o componentes visuales destacamos lo siguiente:

- Propiedades:

Top, Left: Especifican la posición (distancia desde arriba y desde la izquierda) con respecto al contenedor del objeto (el escritorio para un formulario, el formulario o un panel para otros controles)

Height, Width: Especifican el tamaño (altura y anchura).

Caption: Si el control tiene un texto estático, como botones, etiquetas, formularios, etc., Caption especifica el texto.

Font: Especifica la fuente del control.

Enabled: Es un valor lógico o booleano que especifica si el componente está habilitado

Text: Si el texto del control se puede modificar, esta propiedad especifica este texto. Es el caso de los controles tipo *EditBox*, *ComboBox*, o *Memo*.

Visible: Es un valor lógico que especifica si el componente se ve o no.

TabStop, TabOrder: La tecla *Tab* permite en Windows desplazarnos por los controles de un formulario, de forma cíclica. Si queremos que se “pase” por un control determinado, pondremos TabStop a TRUE, y a FALSE en caso contrario. El orden en que se visitan los controles se especifica mediante la propiedad TabOrder. Para variar esta última más fácilmente, podemos pulsar el botón derecho del ratón en el formulario y acceder a la opción *Tab Order...* del menú emergente.

- Métodos:

SetFocus: Da el “foco” al componente, de forma que reaccione ante eventos del ratón o el teclado. La propiedad **ActiveControl** del formulario (TForm) también permite establecer (o comprobar) el control activo.

- Eventos:

OnClick: Se produce cuando el usuario hace clic con el ratón en el control. En algunos casos también se activa al pulsar ENTER o la barra espaciadora con el control seleccionado (por ejemplo, en los botones).

OnEnter, OnExit: Se producen cuando el control recibe (OnEnter) o pierde (OnExit) el foco.

3.3 Ejemplos

3.3.1 MiniCalculadora

Vamos a construir una minicalculadora: una aplicación con dos cuadros de edición, un botón de suma, y una etiqueta. El programa debe verificar que el valor introducido en el primer campo de edición es un número, hacer lo mismo en el segundo y, al pulsar el botón, mostrar la suma en la etiqueta.

Para la realización de este programa debes conocer lo siguiente:

- Para la conversión entre cadenas de caracteres y números se pueden usar las funciones *StrToInt* e *IntToStr*. Si la conversión de string a entero no es posible (porque el texto no representa un número), se genera una excepción, en concreto del tipo *EConvertError*.
- El método *SetFocus* de un control, o la propiedad *ActiveControl* de un formulario, establece el componente activo.

Primera Aproximación

El listado completo de la primera aproximación se encuentra en el apartado 8.2. Aquí destacaremos los principales pasos para implementarlo.

En primer lugar, debemos crear dos campos de edición (TEdit), que llamaremos ESumando1 y ESumando2; un botón (TButton) bSuma, y una etiqueta (TLabel), LTotal. Al crearlos en este orden el componente activo al ejecutar la aplicación pasa a ser el primer sumando, y el orden en que se visitan los componentes el deseado, por lo que no tenemos que modificar el TabOrder.

Una vez que tenemos los componentes, pasaremos a codificar los manejadores de eventos. El evento OnClick del botón no tiene mayor complicación: convertimos los textos de los sumandos a número, los sumamos, y ponemos como Caption de la etiqueta esa suma, de nuevo convertida en String.

El control de que los números introducidos en los sumandos son correctos se puede hacer tratando de convertir el texto (Text) a número. Si da un error, avisamos y volvemos a poner el control activo, y si no da error es que el número introducido es correcto. Para el primer sumando (haciendo doble clic en el Inspector de Objetos para añadir el manejador del evento OnExit) sería:

```
procedure TFormCalc.ESumando1Exit(Sender: TObject);
var num: integer;
begin
  try
    num:=StrToInt(ESumando1.Text)
  except
    showmessage('El número introducido no es válido');
    ESumando1.SetFocus;
  end;
end;
```

Para el segundo sumando sería igual, cambiando ESumando1 por ESumando2.

Podemos comprobar, en el listado del apartado 8.2, que el programa funciona correctamente. Sin embargo, tiene un problema: Hay una duplicidad de código (que como ya sabemos no es algo deseable), ya que el manejador del evento OnExit del primer y segundo sumando son casi iguales.

En la segunda aproximación veremos cómo podemos mejorar este ejemplo para que no exista esta duplicidad de código

Segunda aproximación: Compartir manejador de eventos

Para mejorar el ejemplo, debemos fijarnos en que el manejador de eventos tiene un parámetro, Sender, que es el objeto sobre el que se produce el evento. Si usamos Sender en vez de un nombre de control concreto, podemos compartir manejadores de eventos.

Para ello, en primer lugar borraremos los manejadores que hemos creado. **Borrar un manejador de eventos** es un proceso sencillo, pero debemos hacerlo con cuidado para que el automatismo de Delphi pueda seguir funcionando. No debemos borrar todo el código, sino sólo lo que hemos añadido. Borrando lo que está tachado en el siguiente fragmento de código,

```
procedure TFormCalc.ESumando1Exit(Sender: TObject);  
var num: integer;  
begin  
try  
num:=StrToInt(ESumando1.Text)  
except  
showmessage('El número introducido no es válido');  
ESumando1.SetFocus;  
end;  
end;
```

nos queda

```
procedure TFormCalc.ESumando1Exit(Sender: TObject);  
begin  
end;
```

que es exactamente lo que Delphi había creado automáticamente. Si ahora grabamos el fichero, ese bloque desaparece, y también lo hacen la cabecera que se crea en la definición de clase del formulario y la asignación del manejador al evento (que se almacena en el propio formulario, en el fichero DFM). Así pues, nunca borraremos un manejador de eventos de forma completa, sino que borraremos lo que añadimos nosotros y dejamos que Delphi elimine lo que él ha creado.

Ahora, añadiremos un único manejador de eventos para controlar el evento OnExit de ambos sumandos.

Esto se puede hacer de dos formas:

- Seleccionando los dos TEdits (esto hace que lo que modifiquemos afecte a ambos) y hacer doble clic en el manejador de ese evento (o mejor aún, escribir un nombre distinto del que se crearía automáticamente para el manejador).
- Crear el manejador para el primer sumando, y para el segundo en vez de crear uno nuevo, seleccionar el ya existente (en vez de hacer doble clic o escribir un nombre para el manejador, elegirlo de la lista desplegable)

El siguiente fragmento muestra cómo sería el manejador de eventos para ambos componentes. Debemos fijarnos en que Sender, el parámetro del manejador, es siempre de tipo TObject, el cual no dispone ni de la propiedad Text ni del método

SetFocus. Para acceder a ellos, simplemente usaremos un *cast*, es decir, forzamos el cambio de tipo de este parámetro, usando `TEdit(Sender)` (o de forma equivalente, `Sender as TEdit`).

```
procedure TForm1.SumandosExit(Sender: TObject);
    var num: integer;
begin
    try
        num:=StrToInt(TEdit(Sender).Text)
    except
        showmessage('El número introducido no es válido');
        TEdit(Sender).SetFocus;
    end;
end;
```

3.3.2 Editor de texto simple

Queremos crear un editor de textos muy simple (más aún que el bloc de notas de Windows...) similar al de la Figura 13, con los siguientes componentes:

- El componente principal será de tipo TMemo, que es como un TEdit pero admite más de una línea. De hecho, contiene una propiedad (Lines) de tipo TStrings, que posee métodos para cargar o grabar en disco un fichero de texto directamente. Consulta la ayuda de Delphi para el componente Memo antes de ver este ejemplo.
- Botones para abrir (cargar) un fichero, guardarlo en disco, o borrar el texto actual del editor.

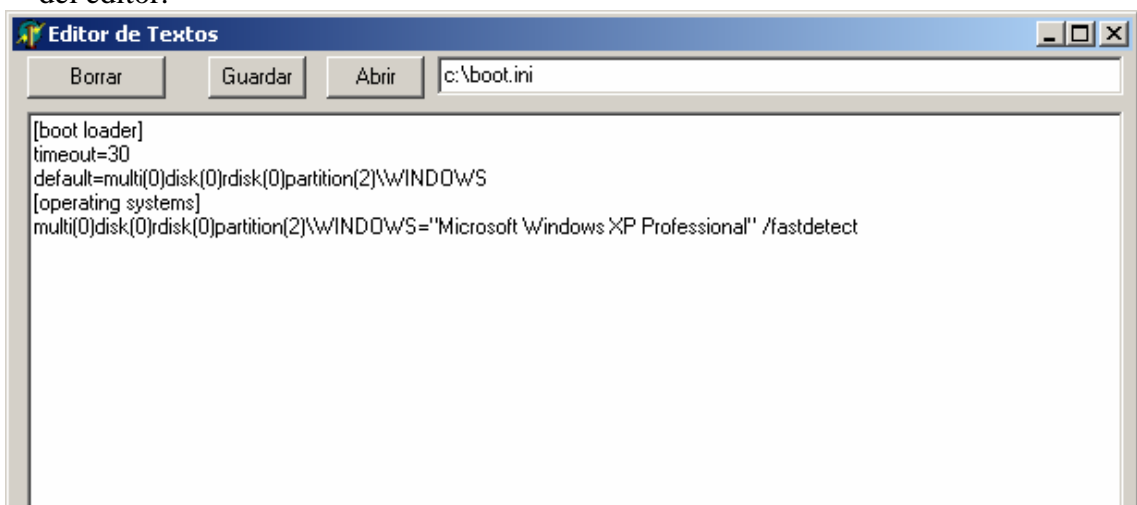


Figura 13. Editor de texto

La solución a este ejemplo se muestra en el punto 8.4 de este documento.

4 Trabajando con más de un formulario

4.1 Cuadros de diálogo predefinidos

La unit `Dialogs`, que normalmente se incluye por defecto al crear un nuevo par unit/formulario, nos proporciona un conjunto de llamadas que muestran cuadros de diálogo sencillos, y pueden aceptar una respuesta del usuario. Entre todos los existentes, destacamos los siguientes:

4.1.1 ShowMessage

Es la llamada más sencilla y proporciona un cuadro de diálogo muy sencillo, con un texto y un botón “Ok” para cerrar el cuadro.

```
ShowMessage('Texto del mensaje');
```

El título del cuadro de diálogo será el nombre del ejecutable, o el título del mismo si se ha cambiado (mediante la propiedad `Application.Title` o en las propiedades del proyecto).

4.1.2 MessageDlg

Aún siendo sencillo, permite una mayor personalización del cuadro de diálogo, mediante la sintaxis

```
MessageDlg(<mensaje>, <tipo_mensaje>, <botones>, <contexto_ayuda>);
```

donde

- `<texto>` es el texto del mensaje
- `<tipo_mensaje>` especifica el tipo de cuadro de diálogo, y puede ser `mtWarning`, `mtError`, `mtConfirmation`, `mtInformation`, `mtCustom`.
- `<botones>` indica los botones que aparecen en el cuadro. Es un conjunto con uno o más de los siguientes botones: `mbYes`, `mbNo`, `mbOk`, `mbCancel`, `mbAbort`, `mbRetry`, `mbIgnore`, `mbAll`, `mbHelp`. Existen constantes predefinidas en Delphi para conjuntos de botones muy usados, como `mbYesNoCancel`, que incluiría los botones Sí, No y Cancelar.
- `<contexto_ayuda>`: Especifica el contexto (numérico) dentro del fichero de ayuda asociado a la aplicación que se activará si pulsamos F1. Podemos especificar 0 si no usamos un fichero de ayuda.

Dado que hay varios botones, la respuesta del usuario puede variar. Esta respuesta, de tipo `TModalResult` (realmente, un valor entero), especifica el botón pulsado, y puede ser `mrYes`, `mrNo`, `mrCancel`, etc.

Ejemplo:

```
If MessageDlg('Fichero modificado. ¿Desea grabarlo?',  
             mtConfirmation, mbYesNo, 0) = mrYes  
then RutinaGrabarFichero;
```

4.1.3 Application.MessageBox

El objeto Application, que representa la aplicación en ejecución, tiene el método MessageBox que permite mostrar mensajes personalizados, incluyendo textos, botones e incluso iconos. Dado que esta función no usa para los mensajes cadenas tipo String sino PChar, y sus opciones son muy numerosas, no entraremos aquí en detalle sobre ella. Consulta la ayuda de Delphi para ver sus posibilidades.

4.1.4 InputBox e InputQuery

Estos métodos se usan para aceptar un texto que el usuario tecleará.

La sentencia

```
texto:=InputBox('Título ventana', 'Mensaje de texto', 'valor por defecto');
```

presenta un cuadro de diálogo con el título y el texto, un cuadro de edición con 'valor por defecto', y los botones de Aceptar y Cancelar. Si se pulsa Aceptar, se devuelve el texto actual del cuadro de diálogo, y si se pulsa cancelar se devuelve 'valor por defecto'.

La función InputQuery es similar, pero devuelve un valor booleano: TRUE si se pulsó Aceptar y FALSE si se pulsó cancelar. La sintaxis sería

```
if InputQuery('Titulo', 'Texto', 'valor por defecto', valor)
then {valor contiene el texto introducido}
else {se pulsó Cancelar}
```

4.1.5 OpenFileDialog y SaveDialog

Estos dos componentes se encuentran en la pestaña "Dialogs" de la paleta de componente. Se colocan en el formulario y, tras modificar sus propiedades (especialmente Filter que controlará los tipos de archivos que queremos abrir, e InitialDir que indica en qué carpeta se abre), se ejecuta mediante una llamada a la función Execute, que devolverá TRUE si se pulsa Aceptar y FALSE en caso contrario.

4.2 Varios formularios

4.2.1 Introducción

Si queremos que nuestra aplicación muestre varios tipos de ventanas o formularios (Forms), debemos diseñarlos, usando un par unit/form para cada ventana distinta que queramos crear. Existen diversas combinaciones y posibilidades en el manejo de varios formularios, como son:

- Crear aplicaciones MDI (Multiple Document Interface) o SDI (Single Document Interface).
- Dejar que Delphi Cree todos los formularios de forma automática al inicio de la ejecución del programa, o crearlos y destruirlos por programa a nuestro antojo.
- Decidir qué ventana será la principal.
- Mostrar ventanas de forma modal o no modal.
- Etc.

Todas estas opciones las veremos en este capítulo, pero antes veamos un ejemplo de algo que siempre es igual: la forma de acceder a un formulario desde otro formulario (que, evidentemente, está en otra unit). Para ello, empecemos con un nuevo proyecto, que contendrá un formulario “Form1” asociado a la unit Unit1. Si pulsamos en el botón rápido “New Form” (o bien en la opción de menú File|New, seleccionando “Form”), Delphi creará en vista de diseño un segundo formulario “Form2”, en la unit Unit2. Aunque es muy conveniente cambiar estos nombres, aquí los conservaremos para seguir más fácilmente las indicaciones.

Supongamos que en el Form1 añadimos un botón, y queremos que al pulsarlo se muestre el formulario Form2. Esto se hará usando el método Show (o ShowModal) asociado al formulario. Es decir:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show
end;
```

Pero, dado que la Unit1 no “conoce” la Unit2, ni el Form2 que esta incluye, es necesario que la primera haga uso (añadiéndola a la cláusula Uses de la unit) de Unit2. De hecho, si tratamos de compilar ese simple programa, Delphi nos avisa de ello, y lo hace automáticamente por nosotros.

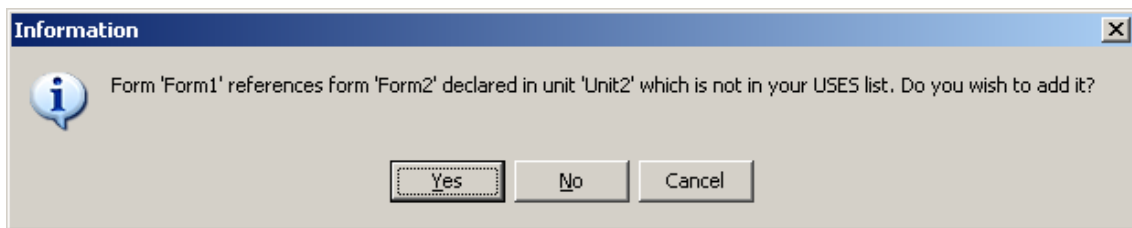


Figura 14. Añadir una unit a la lista USES

Si decimos que No, el programa no compila. Si respondemos afirmativamente, la Unit1 quedaría así:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;

implementation
uses Unit2;
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show
end;
end.
```

Por supuesto, podemos añadir manualmente Unit2 a la lista Uses de Unit1, o dejar que Delphi lo haga por nosotros. Si dejamos que Delphi lo haga, es importante ver que “Unit2” se añade a la lista USES de la parte IMPLEMENTATION de “Unit1” (y así lo debemos hacer nosotros si lo hacemos manualmente). Esto se hace así porque normalmente no es necesario conocer Unit2 en la parte INTERFACE de Unit1 (donde sólo hay declaraciones) y además (y es el motivo principal) para evitar referencias cíclicas entre units:

- Es correcto que Unit1 use Unit2 y Unit2 use Unit1 si ambas lo hacen en la *implementation*
- Es **incorrecto** que Unit1 use Unit2 y Unit2 use Unit1 si ambas lo hacen en la *interface* (de hecho, el proyecto no compilaría).

4.2.2 Creación y destrucción de Ventanas. La ventana principal de la aplicación.

Cualquier aplicación normal de Windows abrirá una ventana (la *ventana principal*), y cuando esa ventana se cierra, el programa termina. Cuando tenemos una sola ventana en nuestro proyecto, esta se crea automáticamente y por supuesto será la ventana principal, pero cuando tenemos varios formularios, debemos decidir cómo y en qué orden se crean, y cual será la ventana principal. Esto podemos verlo en dos sitios:

- En las opciones del proyecto (opción de menú *Project/Options*), pestaña de *Forms*.
- En el código del proyecto (para acceder a él usaremos la opción de menú *Project/View Source*).

La primera opción se muestra en la Figura 15, usando el ejemplo anterior. En ella se ve que ambos formularios, Form1 y Form2, se crean automáticamente, y que Form1 será la ventana principal del programa. El orden en que se crean los formularios se establece (de arriba abajo) en la lista de *Auto-create forms*, pudiendo variarse arrastrando y soltando los nombres de los formularios. La ventana principal se selecciona de la lista desplegable indicada en la figura como *Main Form*.

La misma información se ve en el código del proyecto, que se muestra a continuación. Aquí (entre el *begin* y el *end* del programa), mediante el método *Application.CreateForm*, se crean los formularios Form1 y Form2, en este orden. La ventana principal de la aplicación es el primer formulario que se crea.

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

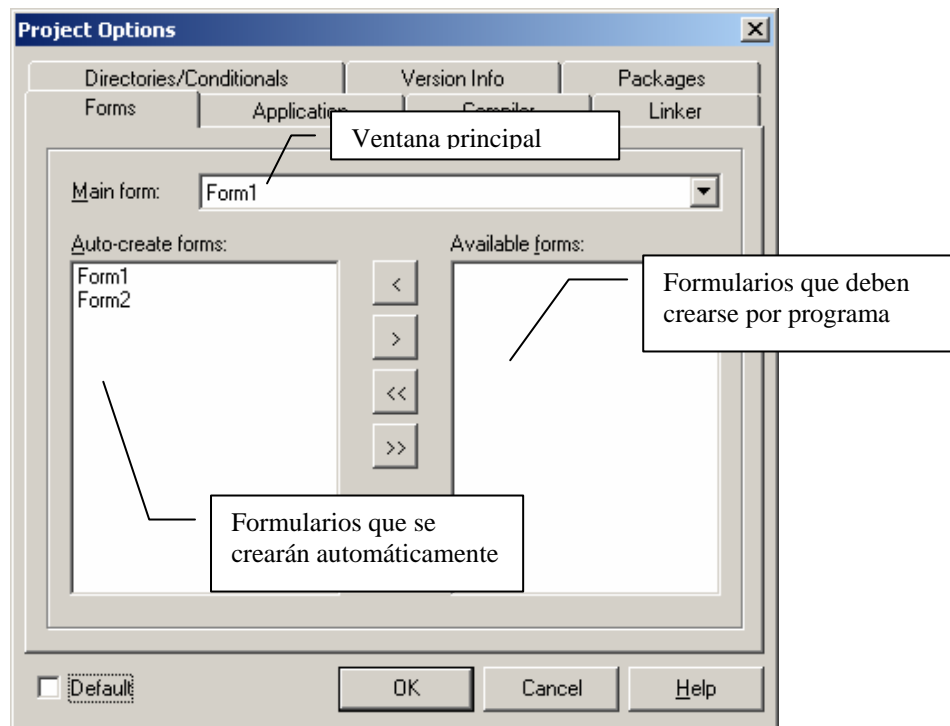



Figura 15. Opciones del proyecto — Formularios

El hecho de crear todos los formularios al inicio y no destruirlos hasta el fin de la aplicación tiene sus ventajas, como que siempre podremos acceder a sus componentes porque sabemos que estarán creados. Sin embargo, también tiene inconvenientes: un excesivo consumo de recursos al tener en memoria (visibles o no) formularios que no necesitamos, y una excesiva lentitud al arrancar el programa, mientras todos los formularios se crean.

Si no deseamos crear todos los formularios al inicio, simplemente vamos a la ventana de la Figura 15, y pasamos a la lista *Available forms* aquellos que no queremos que se creen automáticamente.

Cuando sea necesario crearlos, lo podemos hacer mediante dos formas:

- El método `CreateForm` del objeto `Application`, que toma como parámetros la clase (descendiente de `TForm`) y la variable que queremos instanciar:

```
Application.CreateForm(TForm2, Form2);
```

- Creando el formulario como un objeto más, usando el constructor `Create`, que en este caso toma un parámetro que será el componente “dueño” del formulario, y que podemos rellenar con, por ejemplo, `Application` (el objeto que representa el programa en ejecución):

```
Form2 := TForm2.Create(Application);
```

Una vez creado el formulario se puede mostrar normalmente, usando el método `Show`. Evidentemente, debemos comprobar que el formulario no está ya creado. Para ello podemos compararlo con `nil`, ya que si el formulario está creado la variable que lo almacena no será `nil`. Aunque normalmente Delphi asigna el valor `nil` a las variables que representan objetos antes de que estos se creen, si no “confiamos” en este automatismo, podemos usar el bloque `INITIALIZATION` de la `Unit2` para asegurarnos de ello, añadiendo lo siguiente:

```
Unit Unit2;
```

```

Interface
...
Implementation
...
Initialization
    Form2:=nil;
End;

```

Entonces, el código que desde Form1 crea (si es necesario) y muestra Form2 sería el siguiente.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    if Form2=nil
    then Form2:=TForm2.Create(Application);
        {o bien Application.CreateForm(TForm2, Form2)}
    Form2.Show
end;

```

Usando esta técnica evitamos la lentitud inicial al arrancar el programa (no se nota si son 2 formularios solamente, pero la cosa cambia bastante si son 40 los distintos formularios que puede usar una aplicación). También se evita parcialmente el desperdicio de recursos al mantener en memoria formularios no usados, ya que no se crean hasta que se necesiten. Sin embargo, no lo soluciona del todo ya que el formulario no se destruye cuando se cierra, sino que simplemente se oculta (haciendo una llamada implícita al método `Hide` del formulario). Por ello no sólo debemos saber cómo crear formularios, sino también como destruirlos, eliminando los recursos asociados.

Esto se puede hacer en dos lugares:

- En el punto (unit, formulario) desde el que se creó, llamando al método `Release` del formulario que queremos eliminar
- Desde el propio formulario que queremos eliminar, en el evento `OnClose`. El manejador de este tipo de eventos tiene un parámetro de tipo `TCloseAction`, que indica lo que se hace al “cerrar” el formulario, y puede tener los siguientes valores:
 - o *caNone*: No se permite cerrar el formulario, por lo que nada ocurre.
 - o *caHide*: El formulario se oculta, pero no se destruye. La aplicación aún puede acceder a él, como ocurría en el ejemplo anterior.
 - o *caFree*: El formulario se cierra (destruye) y se libera la memoria asociada.
 - o *caMinimize*: El formulario se minimiza. Es el comportamiento por defecto en los formularios tipo `MDIChild`, que se verán más adelante.

Si añadimos el siguiente manejador para el evento `OnClose` de Form2, el formulario se destruirá y no sólo se ocultará:

```

procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Action:=caFree;
end;

```

4.2.3 Ventanas modales y no modales

Cuando una ventana se muestra, puede hacerlo de dos formas:

- Modal: Sólo se puede acceder a la ventana que se muestra, y no al resto de las ventanas de la aplicación. Es el caso de los cuadros de diálogo estándar que se mostraron en el apartado 4.1.
- No modal: La nueva ventana se muestra, pero podemos acceder a las demás ventanas de la aplicación.

Para mostrar una ventana de forma *no modal* se usa el procedimiento `Show` del formulario; para mostrarla de forma *modal* se usa la función `ShowModal`. El procedimiento `Show` muestra la nueva ventana y continúa la ejecución en la siguiente sentencia, mientras que la función `ShowModal` “espera” a que la nueva ventana sea cerrada, pudiendo controlar el valor que devuelve al cerrarse, de tipo `TModalResult` (como `MessageDlg`, por ejemplo).

El siguiente fragmento sería válido: crea la ventana, la muestra, y cuando el usuario la cierre, la destruye (sin necesidad de controlar el evento `OnClose` de `Form2`).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Form2=nil then Form2:=TForm2.Create(Application);
    Form2.ShowModal;
    Form2.Release;
end;
```

Sin embargo, el mismo código usando `Show` no sería válido, ya que se crearía el formulario `Form2`, se mostraría e inmediatamente se destruiría (`Show` no “espera” sino que la ejecución continúa).

4.2.4 Aplicaciones MDI y SDI

Una aplicación MDI (Multiple Document Interface) puede abrir varias ventanas “hija” dentro de la ventana principal, mientras que una aplicación SDI (Simple Document Interface) tendremos normalmente un único documento. Las aplicaciones SDI, como todos los ejemplos vistos hasta ahora, pueden usar varias ventanas (Delphi es un ejemplo de aplicación SDI) pero no existen varias ventanas “hijas” dentro de la principal.

Para crear aplicaciones de ambos tipos utilizaremos la propiedad `FormStyle` de los formularios. Esta propiedad puede tener los siguientes valores:

- *fsNormal*: El formulario no es ni la ventana MDI “padre” ni “hija”. Es el valor por defecto de esta propiedad, y el que usaremos en todos los formularios de una aplicación SDI.
- *fsMDIChild*: Es una ventana MDI hija.
- *fsMDIForm*: Es la ventana MDI padre. Nótese que sólo debe haber un único formulario de este tipo en la aplicación, y que debe ser la ventana principal.
- *fsStayOnTop*: “Siempre encima”, este tipo de formularios permanece siempre visible, por encima de las demás ventanas (tanto de la aplicación como de las demás aplicaciones que se estén ejecutando en Windows).

La Figura 16 muestra una aplicación MDI, creada mediante un asistente de Delphi, al que se accede en el menú *File/New/Other...*, en la pestaña *Projects*, la opción *MDI Application*.

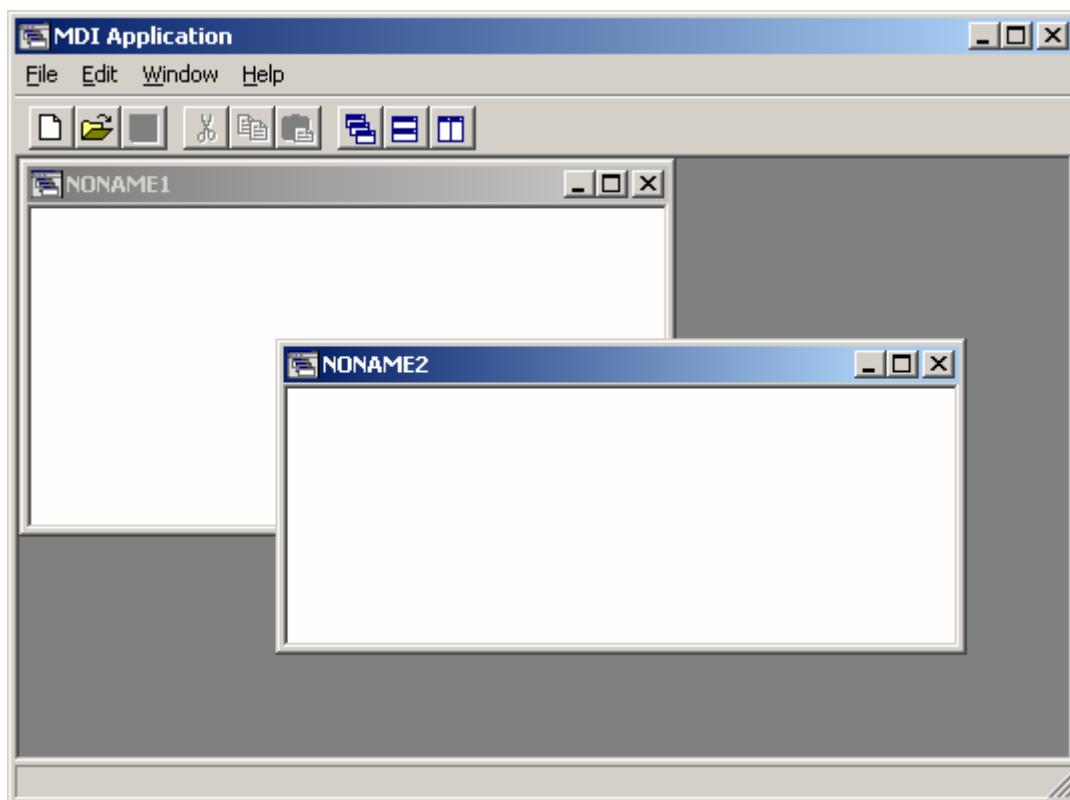


Figura 16. Aplicación MDI

5 Aplicaciones de Bases de Datos

5.1 Componentes de acceso a bases de datos

Delphi fue diseñado desde sus inicios para facilitar la creación de aplicaciones que acceden a bases de datos, tanto locales (por ejemplo, usando ficheros de dBase o Paradox) como servidores de bases de datos remotos (Internase, MySQL, Oracle, Informix, DB2, PostgreSQL,...o cualquier servidor que provea una conexión ODBC o ADO).

Desde su aparición, se ha desarrollado para Delphi un gran número de componentes de acceso a bases de datos, cada uno especializado para un tipo de base de datos en concreto (la mayoría de ellos de "terceras partes", no desarrollados por Borland). Esto hace que el acceso sea eficiente, pero hace la migración de una aplicación (por ejemplo, pasar de usar Paradox a usar un servidor Oracle) imposible o al menos muy difícil. Por ello no hablaremos de este tipo de componentes en este documento, sino que nos centraremos en la arquitectura BDE (Borland Database Engine), presente en todas las versiones de Delphi.

La arquitectura del BDE se divide en tres capas, como muestra la Figura 17. Esta arquitectura nos permite lograr un cierto nivel de independencia física de la aplicación con respecto a la base de datos que use

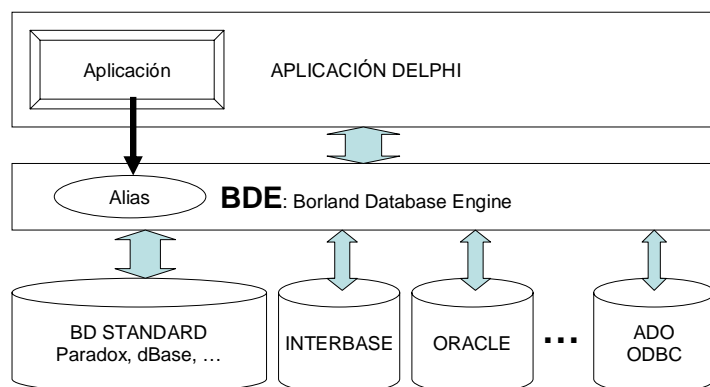


Figura 17. Arquitectura de bases de datos usando BDE

En una aplicación desarrollada en Delphi, también tenemos tres capas (en este caso de componentes):

- *Acceso a Datos*: Estos componentes se encuentran en la pestaña *BDE* de la paleta de componentes. Son componentes no visuales, que realizan el acceso físico real a los datos. De ellos veremos en detalle los componentes *TDatabase* y *TDataSet* (del que usaremos sus descendientes *TTable* y *TQuery*).
- *Componentes "puente"*: Están colocados en la pestaña *Data Access*, y se utilizan para vincular componentes de acceso a datos con los componentes visuales o Controles de Datos. De este tipo de componentes veremos uno solo: el *TDataSource*.
- *Controles de datos*: Estos componentes están en la pestaña *Data Controls*. Son componentes visuales similares a los componentes que permiten ver o editar textos en Windows (*TLabel*, *TEdit*, *TMemo*, ...) pero que en este caso nos

permiten visualizar y modificar los datos, moverlos por los registros o filas, etc. Así, tendremos los componentes TDBLabel, TDBEdit, TDBMemo, TDBNavigator, etc.

La Figura 18 muestra de forma esquemática cómo se usarían estas tres capas en una aplicación desarrollada en Delphi. Aquí se ve la utilidad de la capa “puente”, de los componentes TDataSource: Si tenemos un conjunto de datos (dataset), ya sea una tabla o una consulta, el DataSource se comunica con todos los controles ligados y hace que estos muestren la información de la fila actual. Si el dataset cambia los datos (por ejemplo, al moverse a la fila siguiente), es el DataSource el que se encarga de indicar a los controles que actualicen sus datos, sin que nosotros tengamos que codificar una sola línea.

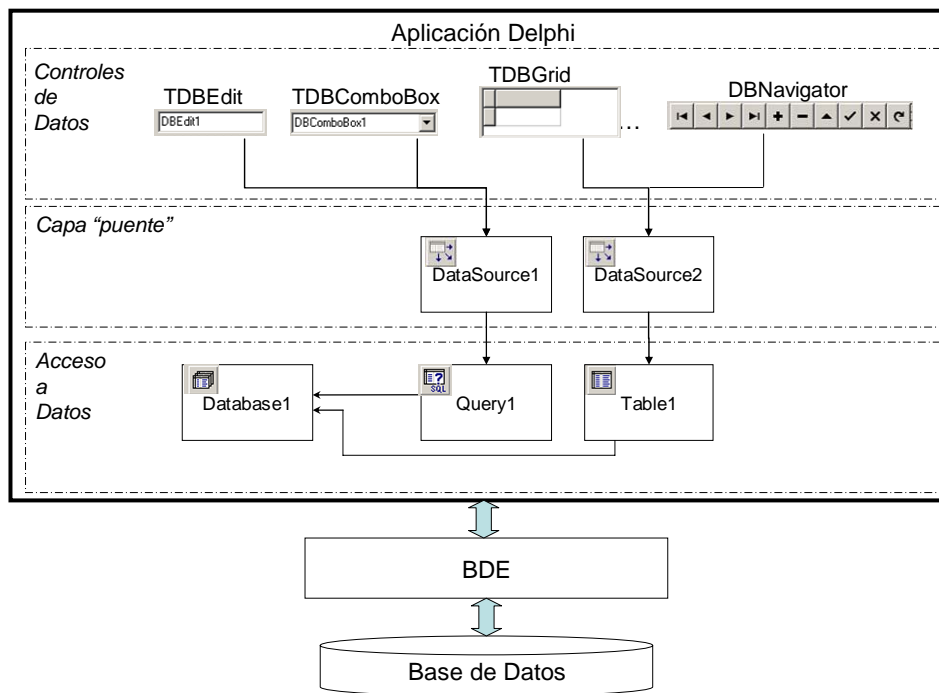


Figura 18. Una aplicación de Delphi que usa BDE

5.1.1 La capa de acceso a datos

Dentro de los componentes de acceso a datos podemos destacar principalmente dos tipos:

- *TDatabase*: Sirve para identificar y establecer la conexión con una base de datos a través de una serie de parámetros. En el caso de que estemos accediendo a un servidor remoto, puede ser el nombre de la máquina, un número de puerto, el nombre de la base de datos y un par usuario/clave para identificarnos. Si la “base de datos” es del tipo XBase o Paradox (donde no hay realmente una BD sino un directorio con ficheros que contienen las tablas y los índices), será simplemente el nombre del directorio. A partir de aquí, y aunque suponga un abuso del idioma, llamaremos “base de datos” a cualquiera de estos sistemas de almacenamiento de información.
- *TDataSet*: permite el acceso a los datos. Es el equivalente a lo que se define como cursor en lenguajes con SQL embebido, o al ResultSet del lenguaje Java.

La clase TDataSet no se suele usar directamente (de hecho no está en la paleta de componentes), sino a través de sus descendientes: TTable, que permite acceder a una tabla de la BD, y TQuery, que utiliza una sentencia SQL para acceder a los datos.

En una aplicación Delphi, todos los componentes TTable y TQuery tienen que estar asociados a una base de datos. El nombre que se da a una base de datos en BDE recibe el nombre de *Database Alias*. Existen 2 tipos de alias: permanentes y temporales (o de aplicación):

5.1.1.1 Alias. Alias permanentes y temporales

Los **Alias permanentes** se crean mediante una aplicación externa (normalmente incluida con Delphi) como Database Desktop o SQL Explorer. Usando Database Desktop, la opción de menú *Tools/Alias Manager* nos permite gestionar los alias permanentes. La Figura 19 muestra dos ejemplos. Como se ve, hay que indicar el nombre del alias, el tipo (para BDs tipo XBase o Paradox se usa el tipo STANDARD), y los parámetros necesarios, que serán dependientes del tipo de base de datos.

Un alias permanente es conocido por todas las aplicaciones que usen el BDE en la computadora. Es decir, que cualquier aplicación que acceda al alias *NuevoAliasParadox* estará accediendo a una base de datos Paradox cuyos datos se almacenan en el directorio C:\Datos\.

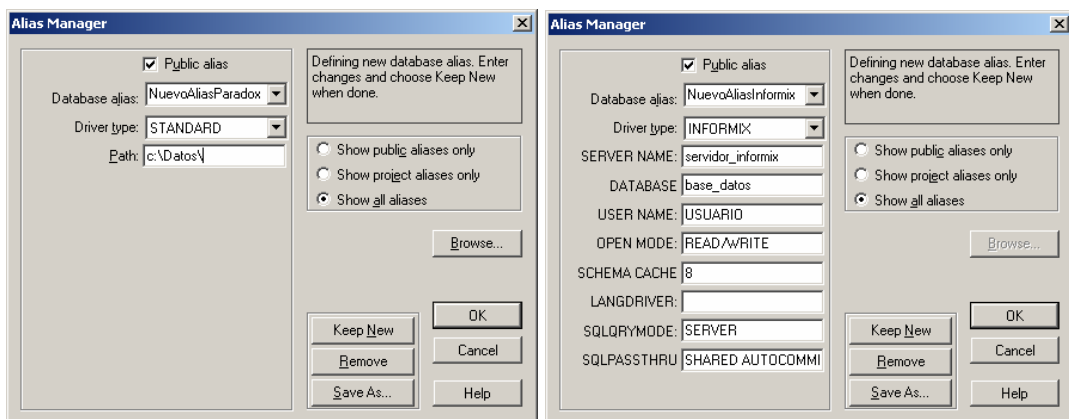
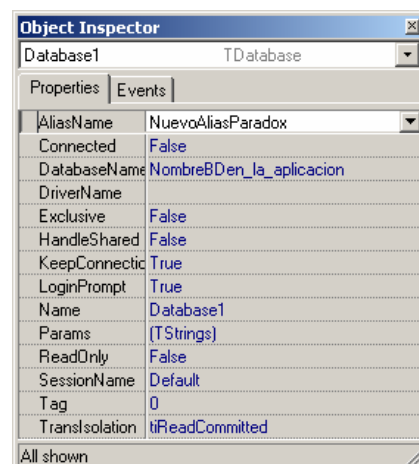


Figura 19. Creación de nuevos alias con Database Desktop

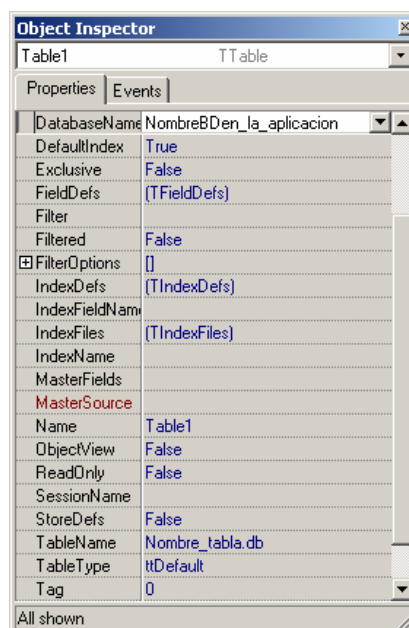
Para crear una aplicación que acceda a una base de datos representada por una alias permanente, se usarán los componentes de acceso a datos (TDatabase y TTable/TQuery).

Colocaremos un componente TDatabase y lo enlazaremos con el alias usando la propiedad AliasName, como muestra la figura.



A continuación, colocaremos componentes TDataset (TTable o TQuery, según los necesitemos), y los enlazaremos, usando la propiedad DatabaseName, con el componente TDatabase.

¡OJO! En la propiedad DatabaseName del TDataset pondremos el nombre que antes hemos dado a la propiedad DatabaseName del TDatabase, **no** el AliasName (que es el alias permanente). Si pusiésemos este último, no estaríamos usando el componente TDatabase.



De hecho, es posible hacer el acceso a datos de forma más directa, sin usar el componente TDatabase. Si no lo usamos, y en la propiedad DatabaseName ponemos (o elegimos de la lista) un nombre de Alias, será totalmente válido. Cuando el TDataset se abre, el propio programa creará de forma dinámica un componente TDatabase “anónimo”, y lo destruirá cuando no sea necesario. Sin embargo, esta opción **NO ES RECOMENDABLE**. Es mejor usar siempre un componente TDatabase¹.

Si quisiésemos realizar los pasos anteriores por programa, en vez de utilizar el Inspector de Objetos, también es posible. La secuencia de instrucciones sería la siguiente (podríamos colocarlas, por ejemplo, en el manejador de eventos correspondiente al evento OnCreate del formulario principal):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Database1.AliasName:='NuevoAliasParadox';
    Database1.DatabaseName:='NombreBD_en_la_aplicacion';
    Table1.DatabaseName:='NombreBD_en_la_aplicacion';
    {O bien Table1.DatabaseName:=Database1.DatabaseName}
    Database1.Connected:=true; {Open: abrimos la BD}
    Table1.Open; {Abrimos la tabla}
end;
```

El uso de alias permanentes tiene sus ventajas. Por ejemplo, si tenemos varias aplicaciones que acceden a una base de datos, todas usarán el mismo alias. Si en un momento dado queremos cambiar de base de datos (o la localización de los datos si es de tipo Standard) simplemente cambiaremos los parámetros del Alias usando Database Desktop, y no necesitamos cambiar nada en las aplicaciones.

¹ Esta es una apreciación personal del autor de este documento

Sin embargo, también tiene desventajas. Entre ellas, la principal es que no podremos tener dos instalaciones del mismo programa, en dos directorios distintos, que accedan cada uno de ellos a una base de datos distinta (el alias permanente es único, no depende de la aplicación que lo use).

Para evitar esta desventaja, podremos usar **Alias Temporales**, también denominados **Alias de Aplicación**. Este tipo de alias se crea usando el componente TDatabase y sólo tiene “vida” mientras la aplicación se esté ejecutando, y sólo será conocido en la aplicación que lo define. Es decir, dos ejecuciones simultáneas de la aplicación pueden usar el mismo nombre de alias de aplicación, pero serán distintos y pueden acceder a dos bases de datos distintas.

Para crear un alias temporal lo haremos colocando un componente TDatabase en el formulario, y en vez de asignarle valor a la propiedad AliasName, configuraremos manualmente los parámetros que lo caracterizan. Esto se puede hacer desde el Inspector de Objetos, desde código, o desde el *Database Editor*, al que se accede haciendo doble clic en el componente TDatabase, o seleccionando en el menú contextual (que aparece al pulsar el botón derecho del ratón sobre el componente) la opción *Database Editor*.

La Figura 20 muestra un ejemplo del Database Editor. Como se ve, en la parte superior hay 3 parámetros:

- *Name*: Se corresponde con la propiedad DatabaseName, y será el nombre del alias temporal.
- *Alias Name*: Se deja en blanco, ya que no usamos alias permanentes.
- *Driver Name*: Dado que no usamos un alias, y que en algún lugar hemos de decir qué tipo de base de datos vamos a usar, este es el lugar en donde lo indicamos. Puede ser STANDARD, Informix, INTRBASE, ..., dependiendo de la instalación del ordenador.

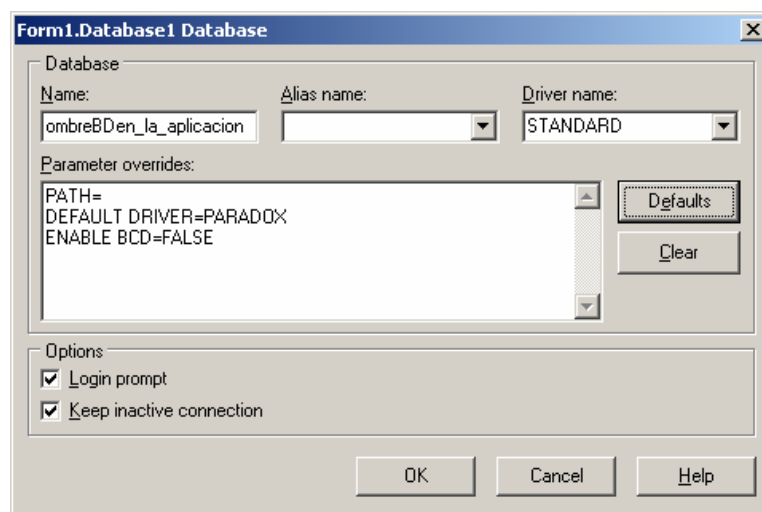


Figura 20. Database Editor

Como ya se ha dicho, cada tipo de base de datos tiene una serie de parámetros para caracterizarlo. En el campo *Parameter Overrides* podremos especificar estos valores. Para saber los valores por defecto de determinado Driver, pulsaremos el botón *Defaults*. (Ejercicio: Selecciona los distintos tipos de drivers, y pulsa el botón *Defaults* para ver qué parámetros caracterizan a cada driver).

Para el tipo STANDARD, vemos que existe, entre otros, el parámetro PATH, que será el que tendremos que modificar para indicar dónde tenemos los datos. Los otros dos parámetros que aparecen tienen su valor por defecto, por lo que podemos dejarlos como están o incluso borrarlos de ese campo de edición.

Una vez que el hemos configurado el componente TDatabase, y creado por tanto el alias temporal, podemos usar los componentes TTable y TQuery exactamente igual que con los alias permanentes, enlazándolos mediante la propiedad DatabaseName.

El siguiente código crea el alias temporal NombreBD_en_la_aplicacion. Puede verse que el parámetro PATH del alias se cambia por programa para que sea el mismo path del ejecutable de la aplicación². Si tuviésemos dos copias de la aplicación en dos directorios distintos (con sus correspondientes bases de datos), estas podrían ejecutarse sin interferir entre ellas.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Database1.Close;
    Database1.DatabaseName:='NombreBD_en_la_aplicacion';
    Database1.DriverName:='STANDARD';
    Database1.Params.Clear;
    Database1.Params.Add('PATH='+ExtractFilePath(ParamStr(0)));
    Table1.DatabaseName:='NombreBD_en_la_aplicacion';
    {O bien Table1.DatabaseName:=Database1.DatabaseName}
    Database1.Connected:=true; {Open: abrimos la BD}
    Table1.Open; {Abrimos la tabla}
end;
```

5.1.1.2 DataSets

A continuación veremos una serie de propiedades y métodos válidos tanto para TTable como para TQuery (si se usa con una consulta SELECT). De hecho, son propiedades y métodos de la clase TDataSet, redefinidos para ambas subclases:

Propiedades

- DatabaseName: Esta propiedad ya ha sido explicada en el punto anterior.
- Active: Es una propiedad booleana. Si su valor es falso, la tabla o consulta estará “cerrada”, por lo que no se está accediendo a los datos. Al asignarle el valor cierto la tabla se abre y se realiza el acceso a los datos.
- State: Determina el estado del dataset. Puede ser dsInactive (cerrado, Active=false), dsBrowse (abierto), dsInsert (estado de inserción, se ha añadido un registro en blanco que se rellenará con datos), dsEdit (se está modificando la fila actual).

Para indicar a qué datos se accede dentro de la base de datos, se hace usando la propiedad indicada, que será TableName para los componentes TTable, y SQL para los componentes TQuery.

- TableName: (Para los componentes TTable) Es un String que indica el nombre de la tabla, o el nombre del fichero si es una base de datos *Standard*. En este último caso, no es necesario indicar la extensión del fichero (es decir, si es

² Si no entiendes la sentencia en negrita, busca ayuda sobre las funciones ExtractFilePath y ParamStr.

una base de datos Paradox, para acceder a la tabla *Persona.db* basta indicar “Persona” en la propiedad `TableName`.

- `SQL`: (Para los componentes `TQuery`) Contiene la consulta, escrita en `SQL`. Por ejemplo, “`Select * From Persona`”. Más adelante veremos en más detalle los distintos tipos de consultas (tanto de consulta como de modificación de datos) que podemos usar, aunque en todos los casos la consulta se especifica en la propiedad `SQL`.

Métodos:

1. `Open`: Abre el dataset. Es equivalente a poner la propiedad `Active=TRUE`. Si la ejecución falla por algún motivo, se genera una excepción. En caso de éxito, el estado del dataset pasa a ser `dsBrowse`.

Los siguientes métodos requieren que la tabla esté abierta.

2. `Close`: Cierra el dataset. Equivale a la asignación `Active:=FALSE`.
3. `First`, `Last`, `Prior`, `Next`: Se usan para “navegar” por el dataset, moviéndonos respectivamente al primer, último, anterior o siguiente registro. Si el dataset está abierto no produce error, pero para que exista un “desplazamiento” real de registro actual, este tiene que existir. Esto se puede comprobar verificando la propiedad `EOF` (End Of File) y `BOF` (Bottom Of File). Si `EOF` es cierto estamos en el último registro, por lo que `Next` no avanza de registro. Si `BOF` es cierto, `Prior` tampoco retrocederá.

Los siguientes métodos permiten realizar modificaciones en los datos. Para los componentes `TQuery`, sólo funcionará si la propiedad `RequestLive` es cierta (el resultado está “vivo”).

- `Delete`: Borra el registro actual (si no hay registros, se produce una excepción).
- `Edit`: Pone el dataset en modo `dsEdit`. Esto permite modificar los valores de los atributos de la fila actual.
- `Insert`: Pone el dataset en modo `dsInsert`, añadiendo una nueva fila (lo hace temporalmente en memoria, no creando una fila en la base de datos) en la que todos los atributos tienen un valor nulo, y pueden ser modificados (asignarles valores).
- `Post`: “Acepta los cambios” y los graba en la base de datos. Si el dataset estaba en modo `dsEdit`, modifica en la base de datos la fila actual, y si estaba en modo `dsInsert`, añade la nueva fila. Al trasladar la modificación a la base de datos se realizarán una serie de comprobaciones (como integridad de entidad o referencial, comprobar que los valores de un atributo son válidos para su tipo, etc.) y si todo es correcto, el dataset pasa a estado `dsBrowse`. Si ocurre algún problema, se genera una excepción.
- `Cancel`: Cancela los cambios, ya sea “recargando” el registro actual con los valores de la base de datos, si se estaba modificando, o eliminando de la memoria el nuevo registro si se estaba insertando.
- `InsertRecord`: Toma como parámetros un array de valores. Se inserta una fila en la tabla con esos valores, en la posición actual (salvo que existan índices).
- `AppendRecord`: Similar, pero añade la fila al final (de nuevo, si no hay índices).

5.1.1.3 TFields: Los atributos de una tabla o consulta

TField es un tipo de datos que permite acceder a los valores de cada atributo (columna, campo) de un dataset, ya sea este una consulta o una tabla. Podemos crear TFields (realmente, descendientes de la clase TField) de forma permanente para un dataset, y si no lo hacemos estos se crearán de forma automática al abrirlo (de la misma forma que se creaba un componente TDatabase).

Es recomendable crear los TField, ya que esto tiene algunas ventajas. Una de ellas es que sabemos el orden en el que se accederá a los campos, lo que es necesario para algunos métodos del dataset (como InsertRecord). Además, si queremos acceder a los valores de los campos por programa, será más sencillo, ya que existen gran cantidad de métodos que permiten realizar conversión de datos.

Existen los siguientes tipos de descendientes de la clase TField:

| | | |
|-----------------|-----------------|--------------------|
| TADTField | TDateField | TReferenceField |
| TAggregateField | TDateTimeField | TSmallIntField |
| TArrayField | TFloatField | TSQLTimeStampField |
| TAutoIncField | TFMTBCDField | TStringField |
| TBCDField | TGraphicField | TTimeField |
| TBinaryField | TGuidField | TVarBytesField |
| TBlobField | TIDispatchField | TVariantField |
| TBooleanField | TIntegerField | TWideStringField |
| TBytesField | TInterfaceField | TWordField |
| TCurrencyField | TLargeintField | |
| TDataSetField | TMemoField | |

Entre las funciones de conversión aplicables a estos tipos, destacamos las siguientes (no todas las funciones serán aplicables a todos los tipos):

- Value: Sí es aplicable a todos los tipos, y devuelve el valor en el tipo “nativo”. Es decir, un <TStringField>.Value devolverá un valor de tipo string, y un <TIntegerField>.Value devolverá un entero.
- AsInteger: Devuelve el valor del campo convertido a entero.
- AsString: Devuelve el valor del campo convertido a String.
- AsDateTime: Convierte el valor a tipo TDateTime (fecha/hora).

Estas propiedades son de lectura y escritura. Es decir, podemos tanto acceder a sus valores como modificarlos. Por ejemplo, si tenemos una tabla TPersona con los siguientes campos:

```
...
TPersona: TTable;
TPersonaDni: TStringField;
TPersonaNombre: TStringField;
TPersonaTelefono: TStringField;
TPersonaFechaNacimiento: TDateField;
TPersonaNumeroHijos: TIntegerField;
```

...

Las siguientes sentencias serían válidas.

```
ShowMessage('La persona se llama ' + TPersonaNombre.Value);
ShowMessage('La persona se llama ' + TPersonaNombre.AsString);
TPersonaFechaNacimiento.AsDateTime := Now;
```

```
TPersonaNumeroHijos.Value:=0;
```

Para añadir los `TField` a un `datasource`, accederemos al Editor de Campos. Para ello haremos doble clic sobre el componente `TTable` o `TQuery` (o en la opción *Fields Editor* del menú contextual). Una vez tenemos los campos, podremos acceder a diversas propiedades, como la etiqueta que se mostrará para referirse al campo (`DisplayLabel`), que es el que se usa para nombrar la cabecera de cada campo en un control `TDBGrid`.

En el editor de campos podemos añadir los campos existentes (*Add Fields*, o *Add all fields*) o añadir campos nuevos, que no serán normalmente campos calculados (*New Field*).

5.1.2 TDataSource y Controles de Datos

Como muestra la Figura 18, las aplicaciones que usan el BDE necesitan controles (es decir, componentes visuales) de datos, y un componente `TDataSource` que los una al `datasource` que, a su vez, accede a los datos.

Del componente `TDataSource` destacamos las propiedades `DataSet` (que será el componente `TTable` o `TQuery` que contiene los datos) y `AutoEdit`. Esta última es una propiedad booleana que, si es cierta, hace que el dataset se ponga en modo de edición automáticamente si modificamos los datos en el control correspondiente.

De los controles visuales, sólo indicaremos de momento que las propiedades más importantes, que debemos establecer para que la aplicación funcione, son las siguientes:

- `DataSource`: El `TDataSource` que enlaza con la fuente de datos.
- `DataField`: El campo del `datasource` al que hace referencia, si es necesario. Para los controles `TDBEdit` o `TDBLabel` será necesario, mientras que para el `TDBNavigator` o `TDBGrid` no, ya que no referencia un campo solamente.

El apartado 8.4 muestra un ejemplo de aplicación completa, que funciona simplemente colocando componentes y ajustando propiedades, sin que tengamos que programar nada.

EJERCICIO 1: Elimina el componente `TDBNavigator` del programa del punto 8.4. Coloca una serie de botones “primero”, “anterior”, “siguiente”, etc. y usa las propiedades del componente `TDataSet` para realizar las acciones necesarias.

5.2 Consultas

El componente `TQuery` permite realizar cualquier tipo de consulta SQL que el servidor de bases de datos admita. Esto dependerá del servidor, ya que evidentemente las capacidades de SQL de una base de datos Paradox (realmente simuladas por el BDE) no pueden compararse con las ofrecidas por un servidor como Oracle.

De cualquier forma, si usamos el estándar SQL-92, normalmente no tendremos problemas con ningún servidor. La propiedad `SQL` del componente `TQuery` es de tipo

TStrings, y puede modificarse tanto por programa como desde el inspector de objetos (que abre un minieditor de textos).

5.2.1 Selección de datos

La selección de datos se hará con una sentencia SELECT. Al abrirla, se crea un *cursor* sobre los datos que nos permite recorrerlos.

La propiedad booleana `Unidirectional` especifica si se podrá recorrer el dataset en ambas direcciones, tanto hacia delante (`Next`, `Last`, etc.) como hacia atrás (`Prior`). Si se pone a falso, no se podrá recorrer el conjunto de datos hacia atrás, pero en cambio se consumen menos recursos.

La propiedad booleana `RequestLive` indica si el dataset está “vivo”, en el sentido de poder modificar los datos usando los métodos `Insert`, `Delete`, etc. Si es falso, no se podrá hacer. Dependiendo de la consulta, a veces no será posible poner esta propiedad a cierto (al igual que no todas las vistas creadas en una base de datos son actualizables), por ejemplo si la consulta selecciona datos de más de una tabla.

5.2.2 Modificación de datos

También se puede usar el componente `TQuery` para modificar datos: añadir filas, borrarlas, o modificarlas, usando respectiva sentencias `INSERT`, `DELETE` o `UPDATE`.

En este caso, la consulta no se “abre” como las consultas de selección de datos, sino que se “ejecuta”, mediante el procedimiento `ExecSQL`. Si la ejecución no es exitosa se genera una excepción de tipo `EDBEngineError`, a través de la cual podemos recoger el código de error “nativo” que se produce en el servidor.

5.2.3 Campos y consultas con parámetros

Podemos acceder a los valores obtenidos por una consulta de selección usando los `TField` si se han definido, al igual que haríamos con un `TTable`.

Sin embargo, es muy común crear consultas dinámicamente, por lo que los `TField` estáticos no estarán definidos. Por ello, si queremos acceder a los datos por programa, podemos usar la propiedad `Fields` (que es un array de 0 al número de expresiones seleccionadas menos 1), o la función `FieldByName`, que accede a un campo por su nombre.

A continuación se muestra un ejemplo donde creamos la consulta dinámicamente y mostramos todos los nombres y DNIs de nuestra agenda (NOTA: tanto `Fields` como `FieldByName` se puede aplicar también a las tablas):

```
procedure TForm2.Button1Click(Sender: TObject);
  var Q:TQuery;
begin
  try
    Q:=TQuery.Create(Application);
    Q.DatabaseName:='Agenda';
    Q.SQL.Add('SELECT NOMBRE, DNI FROM PERSONA');
    Q.Open;
    While not Q.EOF do begin
      ShowMessage(Q.Fields[0].AsString+' con DNI: '+

```

```

        Q.FieldName('DNI').AsString);
    Q.Next;
End;
finally
    Q.Destroy;
end;

```

Algo específico de las consultas es que la sentencia SQL puede ser “dinámica”, admitiendo parámetros, que se rellenarán antes de abrir o ejecutar la consulta. Los parámetros van precedidos por dos puntos “:” en la sentencia SQL, y se puede acceder a ellos a través de la propiedad Params (accesible también desde el Inspector de Objetos) o la función ParamByName, de forma similar a los TField.

Por ejemplo, el siguiente código borra de la agenda la persona cuyo DNI introduzcamos en el control TEdit1 (en este caso, suponemos que la consulta Q está en el formulario Form2).

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Q.SQL.Clear;
    Q.SQL.Add('DELETE FROM PERSONA WHERE DNI=:el_dni');
    Q.ParamByName('el_dni').AsString:=Edit1.Text;
    Q.ExecSql;
end;

```

EJERCICIO 2: Modifica el programa Agenda añadiendo dos campos de edición EHijosMin y EHijosMax, y haciendo que la agenda muestre sólo las personas con un número de hijos entre esos dos valores

5.3 Eventos

Existen eventos ante los que los componentes DataSource y DataSet (TTable y TQuery) responden. A continuación veremos los eventos principales y sus posibles utilidades.

5.3.1 Eventos del DataSource

- OnStateChange: Se produce cuando el estado del DataSet asociado (dsEdit, dsBrowse, etc.) cambia. Es útil, por ejemplo, para activar o desactivar botones que ejecutan determinados métodos. Así, si el estado es dsEdit, habilitaríamos los botones que ejecuten los métodos Post o Cancel del DataSet, pero deshabilitaríamos aquellos que ejecuten Prior, Next, etc.
- OnDataChange: Se produce cuando cambian los datos actuales, normalmente como consecuencia de cambiarnos de fila. Es útil para refrescar datos en controles que no son “data-aware” (si queremos simular un TDBEdit con un TEdit, por ejemplo), o de nuevo para habilitar o deshabilitar controles (por ejemplo, tras una llamada al método Next, se produce este evento y podemos comprobar si estamos en el último registro, y desactivar el botón que avanza una fila en el dataset).

- `OnUpdateData`: Se produce al hacer `Post` el `DataSource` asociado (realmente, *antes* de ejecutar el método `Post`). Es útil para hacer cambios o validaciones de datos justo antes de guardarlos en la base de datos.

5.3.2 Eventos de DataSet

Los siguientes eventos se producen para los dos tipos de `DataSet` que conocemos: las tablas y las consultas. Dado que algunos de ellos (de hecho, la mayoría) de estos eventos ocurren al modificar datos, sólo serán válidos para las consultas si tienen la propiedad `RequestLive` a cierto.

- `OnNewRecord`: Se da al pasar a modo de inserción (al usar los métodos `Insert` o `Append`), y es útil para asignar valores iniciales a la nueva fila. Por ejemplo, en una tabla que guarde facturas, al añadir una nueva fila la fecha será la actual.
- `BeforeScroll/AfterScroll`: Ocurren justo antes y justo después de cambiarnos de fila.
- Relacionados con los métodos que modifican datos, existen eventos que ocurren justo antes (`Before`) o después (`After`) de la ejecución del método. Así, tendremos eventos como `BeforeInsert` y `AfterInsert`, que ocurren justo antes/después de poner el dataset en modo de inserción, o `BeforePost/AfterPost`, que se producen antes/después de hacer la operación de `Post`.

EJERCICIO 3: Modifica el programa del EJERCICIO 1, de forma que los botones se activen o desactiven automáticamente

5.4 Formularios Maestro-Detalle

Hasta ahora hemos visto cómo crear aplicaciones en Delphi que usan una tabla. Para usar más de una, de forma independiente, se programaría de la misma forma. Sin embargo, existen casos en los que las tablas están enlazadas. Es el caso de las relaciones 1:N (del modelo Entidad-Relación), se suelen transformar en el modelo relacional en 2 tablas, de forma que una de ellas será la “principal”, y la otra contendrá una clave foránea hacia ella.

Por ejemplo, es normal diseñar las tablas de una base que almacenan facturas y sus líneas usando el siguiente esquema:

- Factura (Numero, Fecha)
- Linea (NumFac, NumLinea, Producto, Precio)

Si creamos una aplicación Delphi con dos componentes `TTable` “independientes”, obtendríamos algo como lo que se muestra en la Figura 21. Es decir, veremos todas las facturas en el componente `DBGrid` superior, y todas las líneas de todas las facturas en el inferior.

Por supuesto, sería mucho más cómodo que el `DBGrid` de las líneas nos mostrase sólo aquellas líneas de la factura actual. Esto se podría programar “manualmente” usando diversos eventos (como `OnScroll`) de la tabla de facturas, pero dada la

frecuencia con que son necesarias, Delphi nos permite hacerlo de forma mucho más sencilla.

The screenshot shows a Delphi application window titled 'Form1'. It contains two tables. The first table, 'FACTURAS', has columns 'Numero' and 'Fecha'. The second table, 'LÍNEAS', has columns 'NumFac', 'NumLinea', 'Producto', and 'Precio'.

| Numero | Fecha |
|--------|------------|
| 1 | 01/01/2003 |
| 2 | 21/04/2004 |

| NumFac | NumLinea | Producto | Precio |
|--------|----------|-----------|--------|
| 1 | 1 | Lechuga | 0,96 € |
| 1 | 2 | Tomate | 3,00 € |
| 2 | 1 | Folios | 3,25 € |
| 2 | 2 | Bolígrafo | 0,15 € |
| 2 | 3 | Clips | 1,25 € |

Figura 21. Gestión de Facturas y Líneas sin usar Maestro-Detalle

5.4.1 Maestro-Detalle con tablas

Siguiendo con el ejemplo anterior, modificaremos las propiedades `MasterSource` y `MasterFields` (en la tabla de detalle) de los componentes `TTable`:

- `MasterSource`: Indica el `DataSource` asociado a la tabla “maestra”, en este caso *DSFactura*.
- `MasterFields`: Especifica la relación de clave foránea, como se ve en la Figura 22. Para acceder a esta ventana, simplemente pulsaremos en los puntos suspensivos (...) de esta propiedad, en el Inspector de Objetos. Aquí indicamos que el campo `Numero` (de la tabla facturas) se enlaza con el campo `NumFac` de la tabla de líneas.

The screenshot shows the 'Field Link Designer' dialog box. It has a dropdown for 'Available Indexes' set to 'Primary'. There are two sections: 'Detail Fields' and 'Master Fields'. The 'Detail Fields' section contains 'NumLinea'. The 'Master Fields' section contains 'Fecha'. There is an 'Add' button between them. Below these sections is a 'Joined Fields' section containing 'NumFac -> Numero'. There are 'Delete' and 'Clear' buttons next to it. At the bottom are 'OK', 'Cancel', and 'Help' buttons.

Figura 22. Especificación de clave foránea en relaciones Maestro-Detalle

Haciendo solamente este cambio, la aplicación ahora se comportará como en la Figura 23. Tenemos seleccionada la factura número 2, por lo que sólo vemos las líneas de la factura número 2. De hecho (y sin tener que programar nada) si añadimos una fila a la tabla de líneas (pulsando la tecla de cursor abajo en el DBGrid inferior) el número de factura adecuado aparece automáticamente.

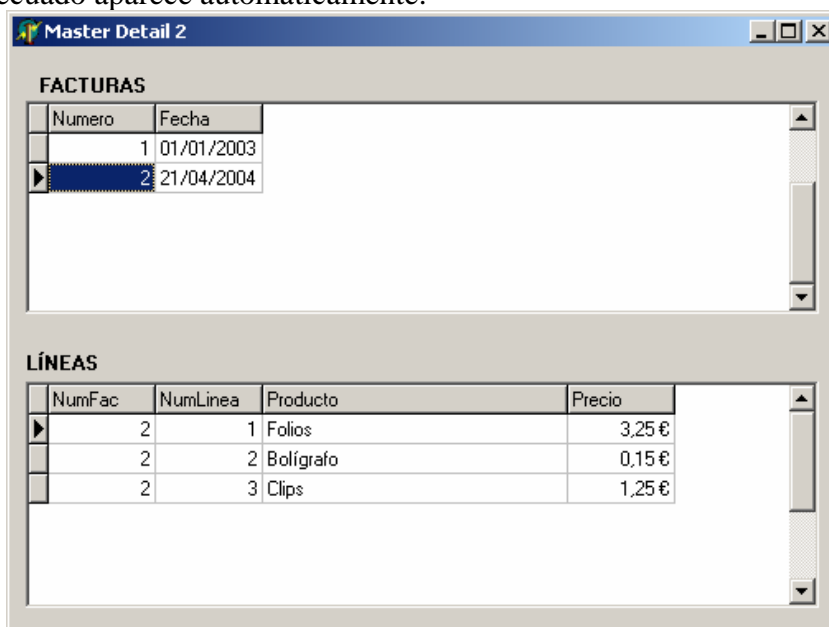


Figura 23. Gestión de Facturas y Líneas usando la relación Maestro-Detalle

5.4.2 Maestro-Detalle con Consultas

También es posible crear formularios Maestro-Detalle usando consultas (TQuery) en vez de tablas. En este caso, no existe la propiedad `MasterSource`, por lo que la relación tenemos que establecerla de otra forma. Veamos el mismo ejemplo de facturas y líneas:

- En la consulta maestra (QFactura) no hacemos nada especial. En su propiedad SQL indicamos que queremos obtener todos los datos de facturas: `SELECT * FROM FACTURA`.
- En la consulta de detalle haremos uso de los parámetros, para indicar que cuando se cambie de tupla en la consulta de facturas debemos reejecutar la consulta de detalle para obtener las líneas de la nueva factura. Esto se consigue haciendo lo siguiente:
 - o La propiedad SQL tendrá un parámetro, que será el “enlace” con la consulta maestra (NOTA: el parámetro debe llamarse igual que el campo de la consulta maestra: “Numero”):
`SELECT * FROM LINEA WHERE NumFac = :Numero.`
 - o A la propiedad `DataSource` de la consulta de detalle se asigna el `DataSource` de la consulta maestra: `DSFactura`.
 Esto hace que automáticamente se cree un parámetro (puede verse en la propiedad `TParams` de la consulta) llamado “Numero”, que servirá de enlace entre ambas consultas.

6 Listados

La mayoría de las aplicaciones Windows ofrecen la posibilidad de generar un informe o listado, por impresora, de lo que el programa gestiona. Delphi también ofrece esta posibilidad, a través de los denominados *generadores de informes*.

Existe gran variedad de estos generadores, tanto incluidos en Delphi (desde la versión 3, Delphi incluye *QuickReport*), como desarrollados por otras compañías (*FastReport* y su versión libre *FreeReport*, *Rave Reports*, *Crystal Reports*, etc).

La mayoría de ellos permite elaborar informes a partir de datos “estáticos”, incluir figuras, fragmentos en RTF, etc., pero su mayor utilidad es la de poder generar fácilmente listados a partir de datos almacenados en una base de datos.

En este capítulo veremos cómo crear informes usando QuickReport ya que, aunque no es un generador especialmente bueno³, viene incluido con Delphi.

QuickReport es un generador “orientado a bandas”. Es decir, en tiempo de diseño nosotros añadiremos bandas horizontales (ver Figura 24) y colocaremos en ellas los componentes que queremos que se impriman. El comportamiento de cada banda dependerá de su tipo, que básicamente será uno de los siguientes (todas son de la clase TQrBand, y se indica entre paréntesis el valor de su propiedad BandType):

- Título (rbTitle): Banda de título, que aparecerá en la parte superior de la primera página.
- Cabecera (rbColumnHeader): Aparecerá en todas las páginas en la parte superior (debajo del título en la primera) para identificar los datos. Es especialmente útil en informes “tabulares” del estilo de una agenda telefónica o una factura.
- Detalle (rbDetail): Esta banda se repetirá tantas veces como filas tenga el dataset asociado. Se usa para imprimir los datos que se obtienen de la base de datos.
- Subdetalle (rbSubdetail): Similar a los formularios maestro-detalle, esta banda sirve para imprimir valores de un dataset “hijo” del que gobierna el informe, por ejemplo las líneas de determinada factura, si queremos imprimir varias facturas completas.
- Resumen (rbSummary): Se imprime una vez se acaban de imprimir todas las bandas de detalle (y subdetalle si hubiese). Serviría, por ejemplo, para colocar el total de una factura una vez impresas todas sus líneas.
- Cabecera o pie de página (rbPageHeader, rbPageFooter)
- Hija (rbChild): Es un tipo de banda especial que se puede asociar a cualquier otro tipo de banda, y se imprimirá después de ella.

En cada banda colocaremos los componentes necesarios para imprimir los datos. De entre todos los posibles (que se pueden ver en la paleta de componentes, en la pestaña QReport) destacamos los siguientes:

- QRLabel: Es el equivalente al TLabel en un formulario, y sirve para imprimir texto estático (usando la propiedad Caption).
- QRDBText: Equivale al TDBText, y a través de sus propiedades DataSource y DataField accede al valor de un atributo de una fila de una

³ De nuevo, es una opinión personal del autor.

tabla o consulta. Será el tipo de componente que usaremos normalmente en las bandas de detalle.

- QRDBExpr: Permite manejar expresiones tales como totales, contadores de registros (equivalente al `select count...`). Suele ser usado en la banda de resumen.
- QRSysData: Imprime automáticamente valores del sistema (fecha, hora, ...) o del informe (número de página actual).

Los pasos a seguir para crear un informe son los siguientes:

1. Colocar un componente TQuickRep en el formulario. Aparecerá una especie de “hoja” cuadriculada. El tamaño y su posición no importan, ya que este es el “diseñador” del informe y no se verá en la aplicación cuando esta se ejecute.
2. Asignar a la propiedad DataSet la tabla o consulta de la que se obtendrán los datos a imprimir.
3. Añadir las bandas necesarias para el informe, colocando los componentes de impresión adecuados. Para añadir una banda podemos seleccionar un componente QRBand de la paleta de componentes y colocarla en el diseñador del informe, o acceder a la propiedad Bands de este, que nos permite añadir las bandas más comunes.

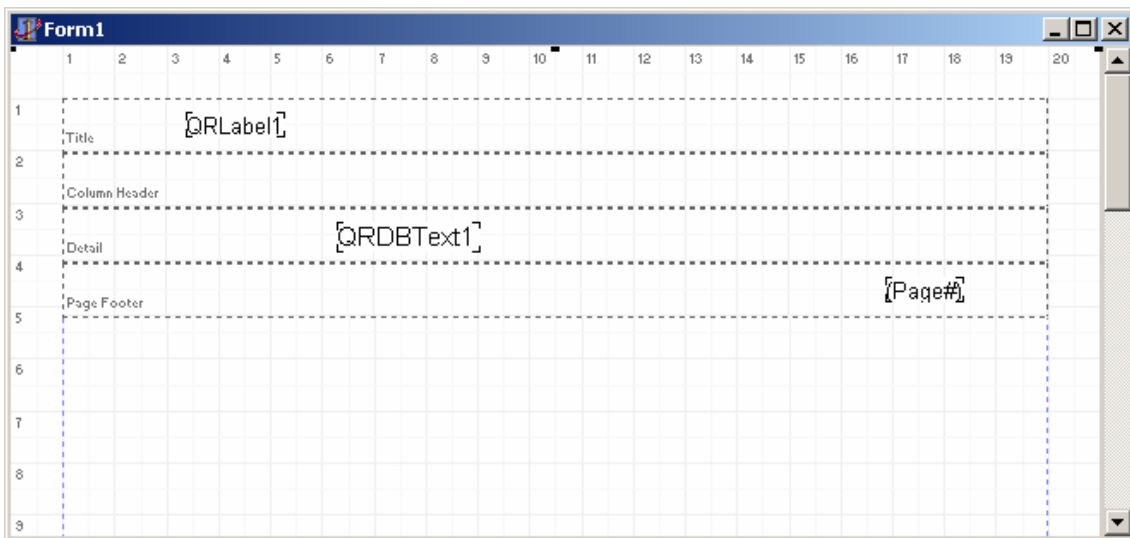


Figura 24. QuickReport diseñando un informe

Una vez que el informe está diseñado, podemos previsualizarlo en pantalla o imprimirlo directamente en la impresora. Para lo primero usaremos el método `Preview` del componente `QuickRep`, y para lo segundo, `Print`. En tiempo de diseño, seleccionando la opción *Preview* del menú emergente del componente `QuickRep`, podemos previsualizar el informe.

6.1 Ejemplo

Para ver un ejemplo real y detallado, expandiremos la agenda del programa 8.4 de forma que podamos imprimir un listado de las personas almacenadas en la agenda (sólo imprimiremos el nombre y el teléfono).

Por la forma en que se diseñan los informes con `QuickReport`, necesitaremos un formulario para colocar el componente `QuickRep`. Por supuesto, podríamos utilizar el

formulario principal de la aplicación, puesto que el diseñador del informe no se vería al ejecutar la aplicación, pero sería demasiado engorroso. Por ello, añadiremos un nuevo formulario al proyecto. Dejaremos que Delphi cree este formulario automáticamente.

Este nuevo formulario (`FListado`, en la `unit Listado`) nunca será mostrado, sólo lo usaremos como “contenedor” para poder diseñar cómodamente el informe, y realizaremos los pasos siguientes:

1. Colocamos un componente `QuickRep` en él (llamado `LST`). Como `DataSet` que gobierna el informe pondremos `FPrincipal.TPersona` (para ello debemos incluir en la lista `Uses` de la `unit Listado` la `unit Principal`).
2. Añadimos una banda de tipo título (por ejemplo, poniendo la propiedad `Bands.HasTitle` a cierto), y en ella un `QRLabel` para identificar el listado.
3. Añadimos una banda de tipo `rbColumnHeader` para identificar las columnas del listado `Nombre` y `Teléfono`, usando de nuevo `QRLabels`.
4. Añadimos una banda de tipo detalle (`rbDetail`) para imprimir los nombres y teléfonos. Para ello usaremos componentes `QRDBText`, estableciendo sus propiedades `DataSet` y `DataField` para obtener los datos.
5. En el formulario `FPrincipal` añadimos un botón que haga que al pulsarlo se previsualice el informe (`FListado.LST.Preview`). Para ello también es necesario “usar” la `unit Listado`, que es donde se encuentra el componente `QuickRep`.

EJERCICIO 4: Añade un listado (completo, de todos los datos de cada persona) al programa `Agenda`.

7 Manejo de Excepciones

Delphi, como muchos otros lenguajes de programación, permite controlar las excepciones que pueden ocurrir durante la ejecución de un programa.

Una excepción es un error de ejecución, que en Delphi es también un objeto (de clase `Exception` y descendientes). Esto nos permite realizar un buen control sobre fragmentos de código problemático.

El control de excepciones en Delphi se suele realizar en Delphi mediante un bloque `try...except`:

```
try
{Bloque protegido}
except
{Manejadores de excepciones}
end;
```

El bloque `try...except` se comporta de la siguiente forma: Las sentencias del bloque protegido se ejecutan. Si no se produce un error, la parte de “Manejadores de excepciones” no se ejecuta y el control se pasa a la línea siguiente al `end`. Si se produce una excepción en una instrucción, las restantes sentencias del bloque protegido no se ejecutan, sino que el control pasa a los manejadores de excepciones. Así, en el siguiente fragmento de código, la sentencia `c:=a/b` produce un error (en concreto, la excepción de tipo `EZeroDivide`), y en vez de continuar la ejecución en la sentencia `c:=c*2`, el control pasa a la sentencia `c:=0` (que es el manejador de excepciones en este caso).

```
try
  a:=5;
  b:=0;
  c:=a/b;
  //Las sentencias a partir de aquí no se ejecutarán
  //debido al error de división por cero
  c:=c*2;
except
  c:=0;
end;
```

Existe otro método de control de excepciones, que se usa normalmente para asegurarnos de que un recurso que es creado dinámicamente se libera cuando ya no es necesario: el bloque `try...finally`. Por ello, a este bloque se le suele denominar bloque de “protección de recursos”. Se diferencia de la anterior en que la parte `finally` se ejecuta siempre, ocurra o no un error.

```
try
{Creación y utilización de recursos}
finally
{Destrucción del recurso}
end;
```

En el siguiente fragmento de código se crea un objeto de tipo `TStrings`, se usa, y luego se destruye, liberando la memoria asociada. Es decir, si en el manejo de la lista se produce un error, inmediatamente se pasa el control al bloque de destrucción del recurso. Si no se produce error, al terminar el bloque de creación y utilización de recurso también se ejecuta el método de destrucción de la parte `finally` del bloque.

```
/*var lista: TStrings;*/
try
```

```

    lista := TStringList.Create;
    // operaciones con lista
finally
    lista.Destroy;
end;

```

Los ejemplos anteriores tenían un único manejador de excepciones. Si sabemos que se pueden dar varios tipos de excepciones (que serán de clases descendientes de `Exception`), se pueden controlar de forma separada usando la estructura

```

On <TipoDeExcepcion> do <Manejador específico>;

```

Podemos tener una o varias sentencias de este tipo, y puede haber una parte `else` para manejar el resto de los tipos de excepciones, como en el siguiente ejemplo.

```

function DivideValores(s1,s2:string):string;
    var n1,n2,division:integer;
begin
    try
        n1:=StrToInt(s1);
        n2:=StrToInt(s2);
        division:=n1 DIV n2;
        Result:= IntToStr(division)
    except
        On EConvertError do
            ShowMessage('Error: Algún valor no es un número');
        On EZeroDivide do
            ShowMessage('Error: División por cero');
        else
            ShowMessage('Excepción no inesperada');
    end;
end;

```

Además, podemos también instanciar una variable de tipo `Exception` (o derivada) para acceder, por ejemplo, al mensaje original:

```

try
    ...
except
    On E:Exception do ShowMessage(E.Message);
end;

```

Finalmente, también podemos generar excepciones de forma manual, mediante la sentencia

```

Raise Exception.Create('Mensaje de excepción personalizado');

```

8 Programas de ejemplo

A continuación se mostrará el código de los programas de ejemplo de este documento. Se mostrará para cada uno de ellos una imagen del formulario (o formularios), el fichero DFM de dicho formulario visto como texto, y el código ObjectPascal de la unit asociada.

8.1 Programa ‘Bienvenida’

8.1.1 Bienvenida.DPR

```
program Bienvenida;
uses
  Forms,
  Principal in 'Principal.pas' {fPrincipal};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TfPrincipal, fPrincipal);
  Application.Run;
end.
```

8.1.2 Formulario Principal

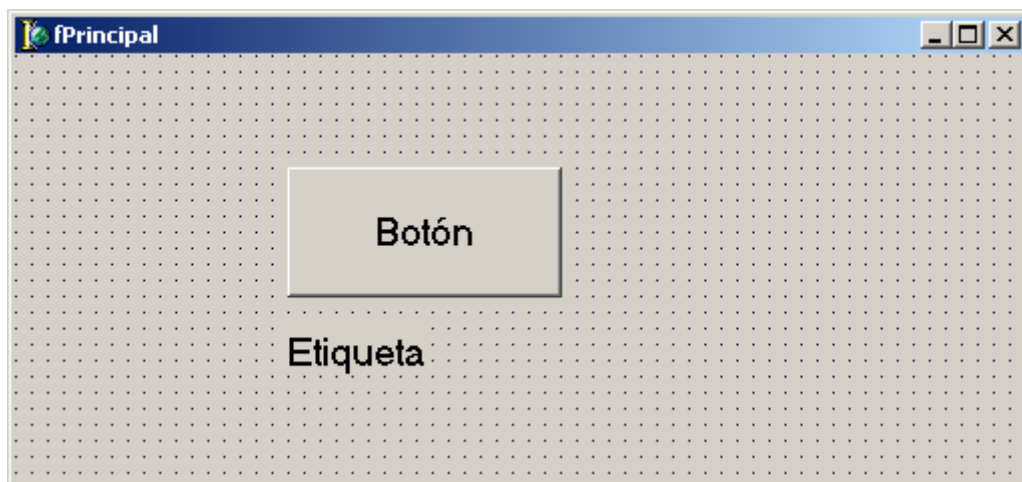


Figura 25. Formulario del programa Bienvenida

8.1.2.1 Principal.DFM

```
object fPrincipal: TfPrincipal
  Left = 239
  Top = 434
  Width = 512
  Height = 239
  Caption = 'fPrincipal'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -19
```



```

Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 24
object Etiqueta: TLabel
Left = 136
Top = 136
Width = 68
Height = 24
Caption = 'Etiqueta'
end
object Boton: TButton
Left = 136
Top = 56
Width = 137
Height = 65
Caption = 'Botón'
TabOrder = 0
OnClick = BotonClick
end
end

```

8.1.2.2 Principal.PAS

```

unit Principal;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;
type
TfPrincipal = class(TForm)
Boton: TButton;
Etiqueta: TLabel;
procedure BotonClick(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
var
fPrincipal: TfPrincipal;
implementation
{$R *.dfm}
procedure TfPrincipal.BotonClick(Sender: TObject);
begin
Etiqueta.Caption:='Bienvenido/a a Delphi';
end;
end.

```

8.2 Minicalculadora (versión 1, MiniCalc1)

8.2.1 MiniCalc1.DPR

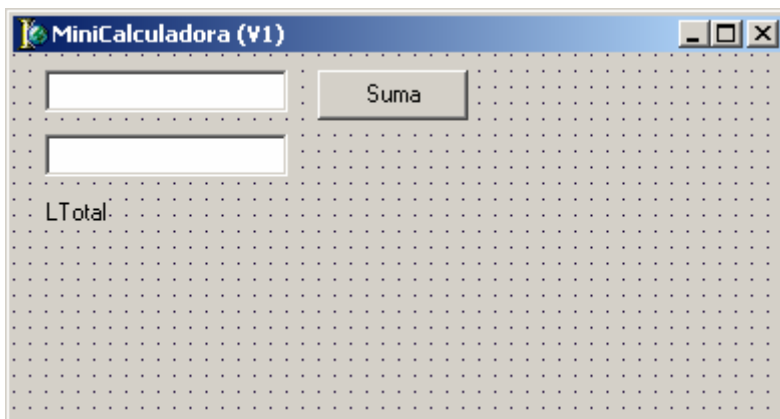
```
program MiniCalc1;

uses
  Forms,
  FCalc in 'FCalc.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TFormCalc, FormCalc);
  Application.Run;
end.
```

8.2.2 Formulario Principal



8.2.2.1 FCalc.DPR

```
object FormCalc: TFormCalc
  Left = 192
  Top = 107
  Width = 392
  Height = 208
  Caption = 'MiniCalculadora (V1)'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object LTtotal: TLabel
    Left = 16
    Top = 72
    Width = 30
    Height = 13
    Caption = 'LTtotal'
```

```

end
object ESumando1: TEdit
    Left = 16
    Top = 8
    Width = 121
    Height = 21
    TabOrder = 0
    OnExit = ESumando1Exit
end
object ESumando2: TEdit
    Left = 16
    Top = 40
    Width = 121
    Height = 21
    TabOrder = 1
    OnExit = ESumando2Exit
end
object bSuma: TButton
    Left = 152
    Top = 8
    Width = 75
    Height = 25
    Caption = 'Suma'
    TabOrder = 2
    OnClick = bSumaClick
end
end
end

```

8.2.2.2 FCalc.PAS

```

unit FCalc;

interface

uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;

type
    TFormCalc = class(TForm)
        ESumando1: TEdit;
        ESumando2: TEdit;
        LTotal: TLabel;
        bSuma: TButton;
        procedure ESumando1Exit(Sender: TObject);
        procedure ESumando2Exit(Sender: TObject);
        procedure bSumaClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    FormCalc: TFormCalc;

implementation

{$R *.dfm}

procedure TFormCalc.ESumando1Exit(Sender: TObject);
    var num: integer;

```

```

begin
    try
        num:=StrToInt(ESumando1.Text)
    except
        showmessage('El número introducido no es válido');
        ESumando1.SetFocus;
    end;
end;

procedure TFormCalc.ESumando2Exit(Sender: TObject);
    var num: integer;
begin
    try
        num:=StrToInt(ESumando2.Text)
    except
        showmessage('El número introducido no es válido');
        ESumando2.SetFocus;
    end;
end;

procedure TFormCalc.bSumaClick(Sender: TObject);
begin
    LTotal.Caption:=
        IntToStr(StrToInt(ESumando1.Text)+StrToInt(ESumando2.Text));
end;

end.

```

8.3 Minicalculadora (versión 2, MiniCalc2)

8.3.1 Formulario principal

Dado que sólo cambia la gestión de eventos, incluimos solamente el código correspondiente al FCalc.PAS (ambos componentes TEdit tienen como manejador del evento OnExit el procedimiento SumandosExit):

```
unit FCalc;

interface

uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        ESumando1: TEdit;
        ESumando2: TEdit;
        LTotal: TLabel;
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
        procedure SumandosExit(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
    LTotal.Caption:=
        IntToStr(StrToInt(ESumando1.Text)+StrToInt(ESumando2.Text));
end;

procedure TForm1.SumandosExit(Sender: TObject);
var num: integer;
begin
    try
        num:=StrToInt(TEdit(Sender).Text)
    except
        showmessage('El número introducido no es válido');
        TEdit(Sender).SetFocus;
    end;
end;

end;
end.
```

8.4 Editor de textos

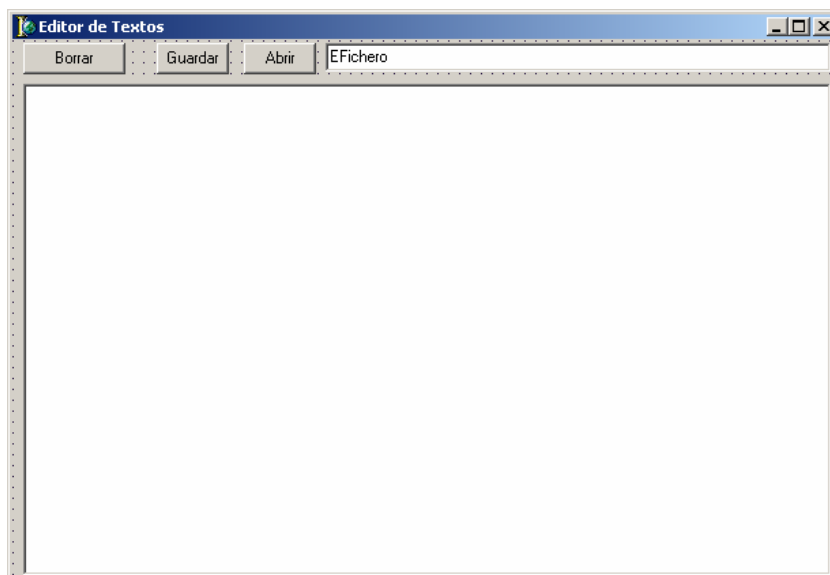
8.4.1 Editor1.DPR

```
program editor1;
uses
  Forms,
  Edit in 'Edit.pas' {Form1};

{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

8.4.2 Formulario principal

8.4.2.1 Edit.DFM



```
object Form1: TForm1
  Left = 260
  Top = 445
  Width = 614
  Height = 421
  Caption = 'Editor de Textos'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object EFichero: TEdit
    Left = 230
```

```

        Top = 2
        Width = 371
        Height = 21
        TabOrder = 0
        Text = 'EFichero'
    end
    object bAbrir: TButton
        Left = 170
        Top = 2
        Width = 53
        Height = 23
        Caption = 'Abrir'
        TabOrder = 1
        OnClick = bAbrirClick
    end
    object bBorrar: TButton
        Left = 8
        Top = 2
        Width = 75
        Height = 23
        Caption = 'Borrar'
        TabOrder = 2
        OnClick = bBorrarClick
    end
    object Texto: TMemo
        Left = 8
        Top = 32
        Width = 593
        Height = 361
        TabOrder = 3
    end
    object bGuardar: TButton
        Left = 106
        Top = 2
        Width = 53
        Height = 23
        Caption = 'Guardar'
        TabOrder = 4
        OnClick = bGuardarClick
    end
end
end

```

8.4.2.2 Edit.PAS

```

unit Edit;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs,
    StdCtrls;

type
    TForm1 = class(TForm)
        EFichero: TEdit;
        bAbrir: TButton;
        bBorrar: TButton;
        Texto: TMemo;
        bGuardar: TButton;
        procedure bAbrirClick(Sender: TObject);
    end;

```

```

        procedure bGuardarClick(Sender: TObject);
        procedure bBorrarClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.bAbrirClick(Sender: TObject);
begin
    try
        Texto.Lines.LoadFromFile(Efichero.Text);
    except
        Application.MessageBox(
            Pchar(Efichero.Text+ ' no existe o no se puede leer'),
            'Error al abrir fichero', MB_ICONERROR)
    end
end;

procedure TForm1.bGuardarClick(Sender: TObject);
begin
    try
        Texto.Lines.SaveToFile(Efichero.Text);
    except
        Application.MessageBox(
            Pchar('No se puede guardar el texto en '+Efichero.Text),
            'Error al guardar fichero', MB_ICONERROR)
    end
end;

procedure TForm1.bBorrarClick(Sender: TObject);
begin
    Texto.Clear;
end;
end.

```

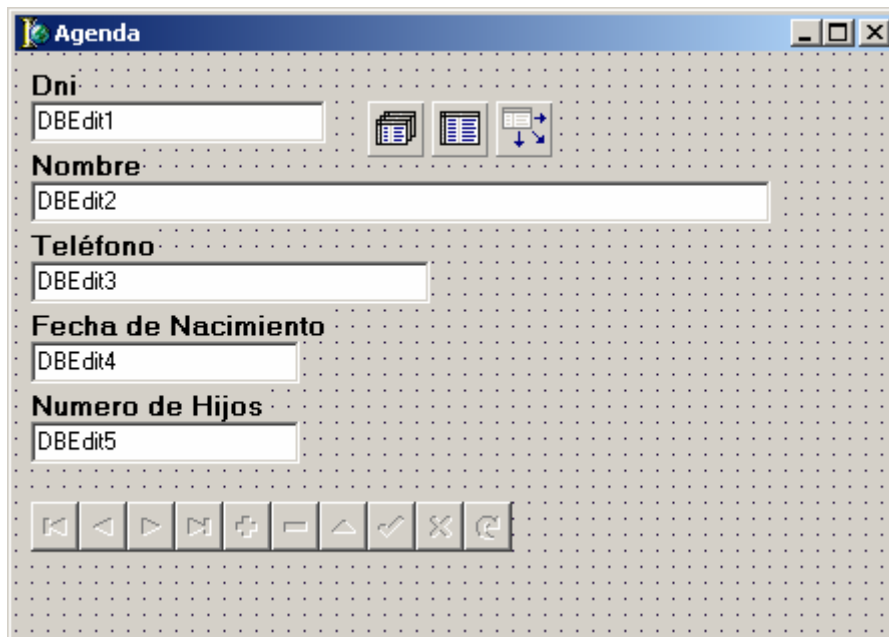

8.5 Agenda

8.5.1 Agenda.DPR

```
program agenda;  
  
uses  
    Forms,  
    Principal in 'Principal.pas' {Form1};  
  
{$R *.res}  
  
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

8.5.2 Formulario Principal

8.5.2.1 Principal.DFM



```
object Form1: TForm1  
    Left = 317  
    Top = 353  
    Width = 447  
    Height = 317  
    Caption = 'Agenda'  
    Color = clBtnFace  
    Font.Charset = DEFAULT_CHARSET  
    Font.Color = clWindowText  
    Font.Height = -11  
    Font.Name = 'MS Sans Serif'  
    Font.Style = []  
    OldCreateOrder = False
```

```

PixelsPerInch = 96
TextHeight = 13
object Label1: TLabel
    Left = 8
    Top = 8
    Width = 24
    Height = 16
    Caption = 'Dni'
    FocusControl = DBEdit1
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
end
object Label2: TLabel
    Left = 8
    Top = 48
    Width = 56
    Height = 16
    Caption = 'Nombre'
    FocusControl = DBEdit2
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
end
object Label3: TLabel
    Left = 8
    Top = 88
    Width = 63
    Height = 16
    Caption = 'Teléfono'
    FocusControl = DBEdit3
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
end
object Label4: TLabel
    Left = 8
    Top = 128
    Width = 148
    Height = 16
    Caption = 'Fecha de Nacimiento'
    FocusControl = DBEdit4
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
end
object Label5: TLabel
    Left = 8
    Top = 168

```

```

Width = 117
Height = 16
Caption = 'Numero de Hijos'
FocusControl = DBEdit5
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -13
Font.Name = 'MS Sans Serif'
Font.Style = [fsBold]
ParentFont = False
end
object DBEdit1: TDBEdit
  Left = 8
  Top = 24
  Width = 147
  Height = 21
  DataField = 'Dni'
  DataSource = DSPersona
  TabOrder = 0
end
object DBEdit2: TDBEdit
  Left = 8
  Top = 64
  Width = 369
  Height = 21
  DataField = 'Nombre'
  DataSource = DSPersona
  TabOrder = 1
end
object DBEdit3: TDBEdit
  Left = 8
  Top = 104
  Width = 199
  Height = 21
  DataField = 'Telefono'
  DataSource = DSPersona
  TabOrder = 2
end
object DBEdit4: TDBEdit
  Left = 8
  Top = 144
  Width = 134
  Height = 21
  DataField = 'FechaNacimiento'
  DataSource = DSPersona
  TabOrder = 3
end
object DBEdit5: TDBEdit
  Left = 8
  Top = 184
  Width = 134
  Height = 21
  DataField = 'NumeroHijos'
  DataSource = DSPersona
  TabOrder = 4
end
object DBNavigator1: TDBNavigator
  Left = 8
  Top = 224
  Width = 240
  Height = 25

```

```

        DataSource = DSPersona
        TabOrder = 5
    end
    object DSPersona: TDataSource
        DataSet = TPersona
        Left = 240
        Top = 24
    end
    object Databasel: TDatabase
        AliasName = 'personall'
        Connected = True
        DatabaseName = 'BDAgenda'
        SessionName = 'Default'
        Left = 176
        Top = 24
    end
    object TPersona: TTable
        Active = True
        DatabaseName = 'BDAgenda'
        TableName = 'persona'
        Left = 208
        Top = 24
        object TPersonaDni: TStringField
            FieldName = 'Dni'
            Size = 11
        end
        object TPersonaNombre: TStringField
            FieldName = 'Nombre'
            Size = 50
        end
        object TPersonaTelefono: TStringField
            DisplayLabel = 'Teléfono'
            FieldName = 'Telefono'
            Size = 15
        end
        object TPersonaFechaNacimiento: TDateField
            DisplayLabel = 'Fecha de Nacimiento'
            FieldName = 'FechaNacimiento'
        end
        object TPersonaNumeroHijos: TIntegerField
            DisplayLabel = 'Numero de Hijos'
            FieldName = 'NumeroHijos'
        end
    end
end
end

```

8.5.2.2 *Principal.PAS*

Como vemos, en este primer ejemplo de programa que accede a bases de datos, tenemos un programa de mantenimiento (altas, bajas, modificaciones) sin escribir una sola línea de código.

```

unit Principal;

interface

uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, DB, DBTables, ExtCtrls, DBCtrls,
    StdCtrls, Mask;

```

```

type
  TForm1 = class(TForm)
    DSPersona: TDataSource;
    Databasel: TDatabase;
    TPersona: TTable;
    TPersonaDni: TStringField;
    TPersonaNombre: TStringField;
    TPersonaTelefono: TStringField;
    TPersonaFechaNacimiento: TDateField;
    TPersonaNumeroHijos: TIntegerField;
    Label1: TLabel;
    DBEdit1: TDBEdit;
    Label2: TLabel;
    DBEdit2: TDBEdit;
    Label3: TLabel;
    DBEdit3: TDBEdit;
    Label4: TLabel;
    DBEdit4: TDBEdit;
    Label5: TLabel;
    DBEdit5: TDBEdit;
    DBNavigator1: TDBNavigator;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.

```