2.3.4 Creación de componentes en tiempo de ejecución

Los mejores entornos de programación visual incluyen una serie de herramientas que tienen como fin facilitar al desarrollador el proceso de creación de todo tipo de objetos gráficos y la gestión de los eventos asociados a ellos. Este es el caso del Borland C++ Builder.

El entorno de desarrollo Borland C++ Builder incluye una serie de herramientas de diseño RAD (desarrollo rápido de aplicaciones) incluidas plantillas de aplicación y de ficha y asistentes de programación. Además permite programación orientada a objetos real: la biblioteca de clases que incluye contiene objetos que encapsula la API de Windows y otras técnicas de programación de gran utilidad.

El entorno de desarrollo de C++ Builder incluye asimismo un diseñador visual de fichas, un inspector de objetos y una paleta de componentes, aparte de las herramientas más típicas, como gestor de proyectos, editor de código fuente, depurador y herramienta de instalación.

Todo el trabajo de diseño visual en C++ Builder se realiza en fichas. Cuando se abre C++ Builder o se crea un nuevo proyecto, se visualiza una ficha vacía en la pantalla. Sobre esta ficha puede empezar a construirse la interfaz de la aplicación, incluidas ventanas, menús y cuadros de diálogo. Para diseñar la interfaz gráfica de usuario de una aplicación, basta con colocar en la ficha los componentes visuales, como botones y cuadros de lista. C++ Builder se ocupa automáticamente de los detalles de programación subyacentes.

Los componentes suelen lograr un grado de encapsulación mayor que las clases estándar de C++. Por ejemplo, suponiendo que se dispone de un cuadro de diálogo que contiene un botón, en un programa de Windows desarrollado en C++, cuando el usuario hace click sobre el botón, Windows genera un mensaje que debe ser capturado y tratarlo con una rutina que se ejecutará como respuesta a dicho mensaje.

En C++ Builder, el componente botón está programado para responder a un click del ratón con el manejador de sucesos incorporado OnClick. Por ello no es necesario determinar en el programa que se ha hecho click con el ratón, sino que es suficiente con especificar la rutina que debe llamarse cuando se hace click con el ratón.

De un modo similar la mayoría de los componentes de C++ Builder manejan la mayoría de los mensajes de Windows. En la figura 2.3.4.1 se tiene una imagen del entorno de desarrollo C++ Builder.

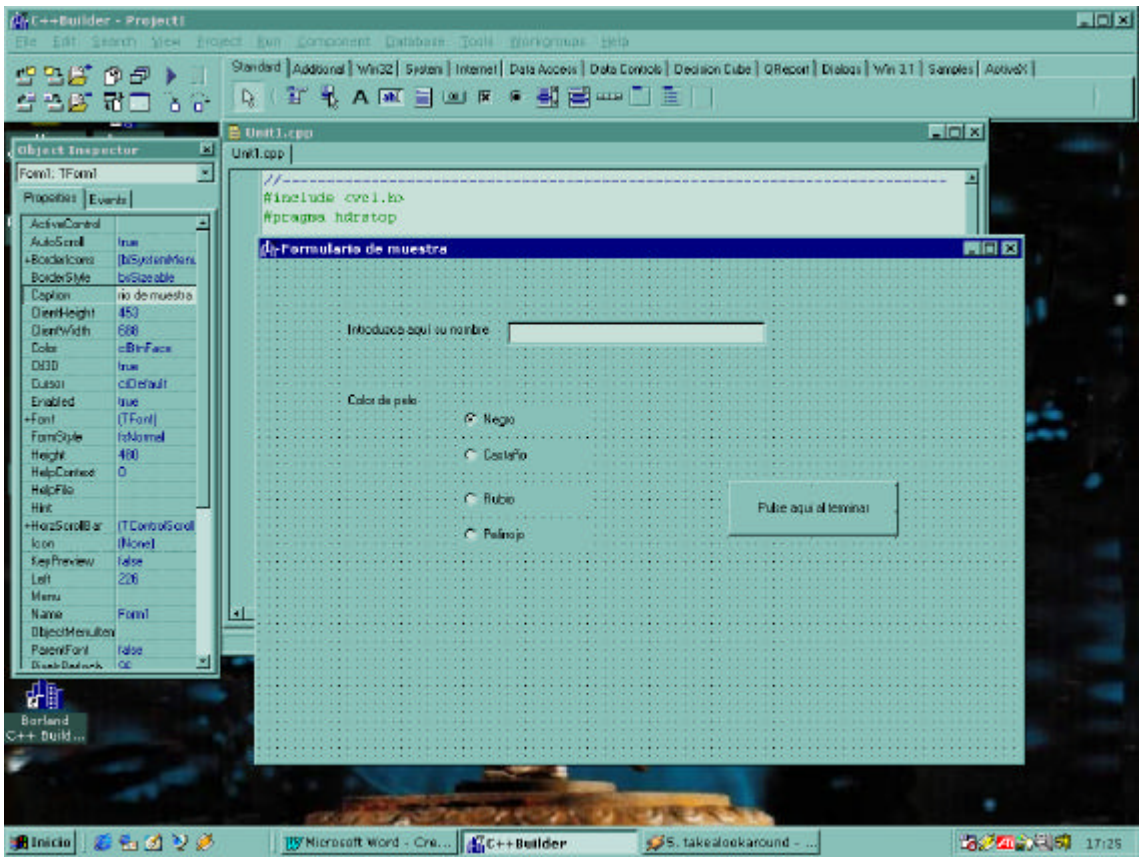

Figura 2.3.4.1: Entorno de desarrollo visual Borland C++ Builder

En la captura de pantalla que se muestra en la figura 1 se puede apreciar la ficha, donde se han añadido ya varios componentes visuales: dos etiquetas de texto (*Introduzca aquí su nombre* y *Color de pelo*), cuatro botones de radio (cada uno de los colores de pelo), un cuadro de diálogo simple (donde se debe introducir el nombre) y un botón (*Pulse aquí al terminar*) al que se le pueden asociar los distintos eventos, por ejemplo el evento producido por un click de ratón sobre él, a cualquier función que se codifique. La identificación de cada componente se aprecia claramente en las figuras 2.3.4.2, 2.3.4.3, 2.3.4.4, 2.3.4.5 y 2.3.4.6, ya que está delimitado en color rojo.

Para crear un objeto dentro de un formulario con C++ Builder, basta con crear un objeto de la clase deseada de la biblioteca de clases visuales de Borland (VCL, Visual Class Library). Este objeto puede crearse a mano, que es el proceso que se ha seguido en la aplicación o simplemente seleccionando la clase de la paleta e insertándola en el formulario. En este último caso, C++ Builder genera automáticamente el código necesario.

En el caso de que se cree un objeto a mano dentro de un formulario, no hay que olvidar asignar a la propiedad *Parent* del objeto creado el formulario sobre el que debe aparecer, ya que ésta propiedad determina quién recibe los eventos que se producirán sobre dicho objeto y se usa para establecer el origen de coordenadas al asignar las propiedades de tipo geométrico a dicho objeto.

Para crear un formulario basta con crear un objeto de la clase *TForm*.

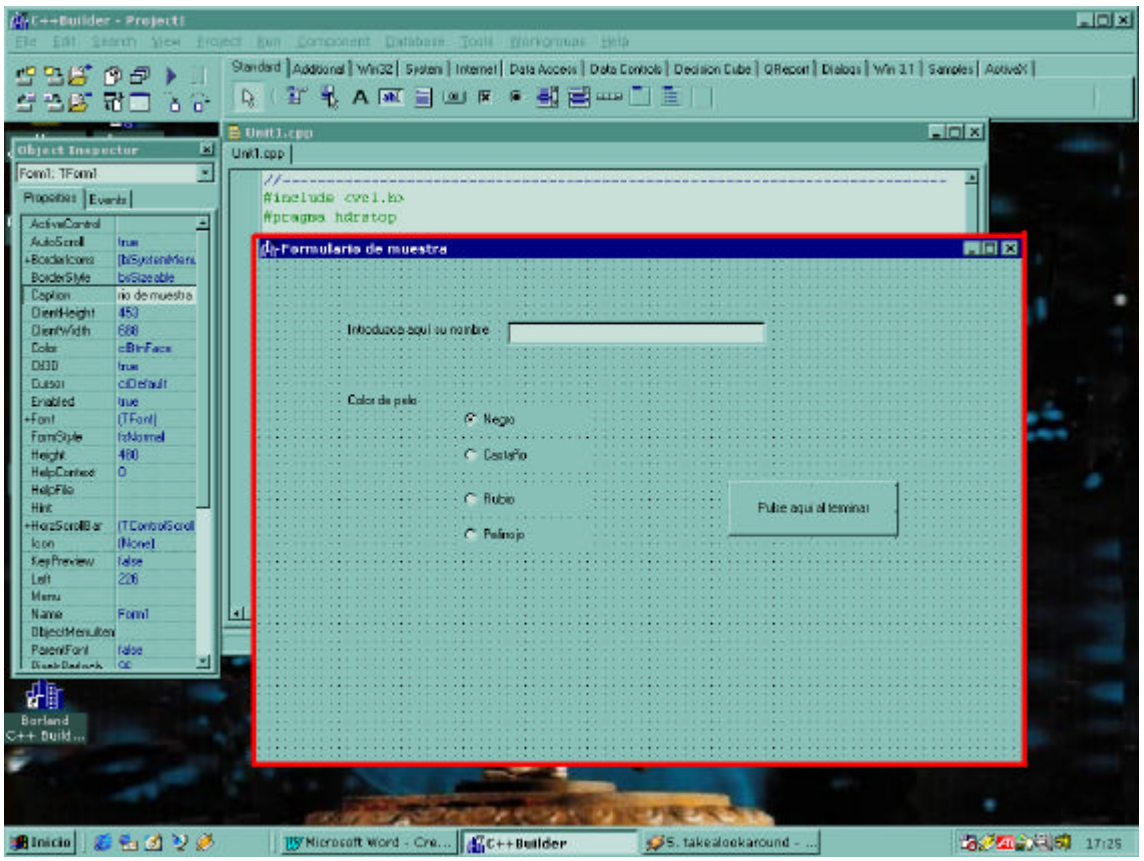

Figura 2.3.4.2: Formulario

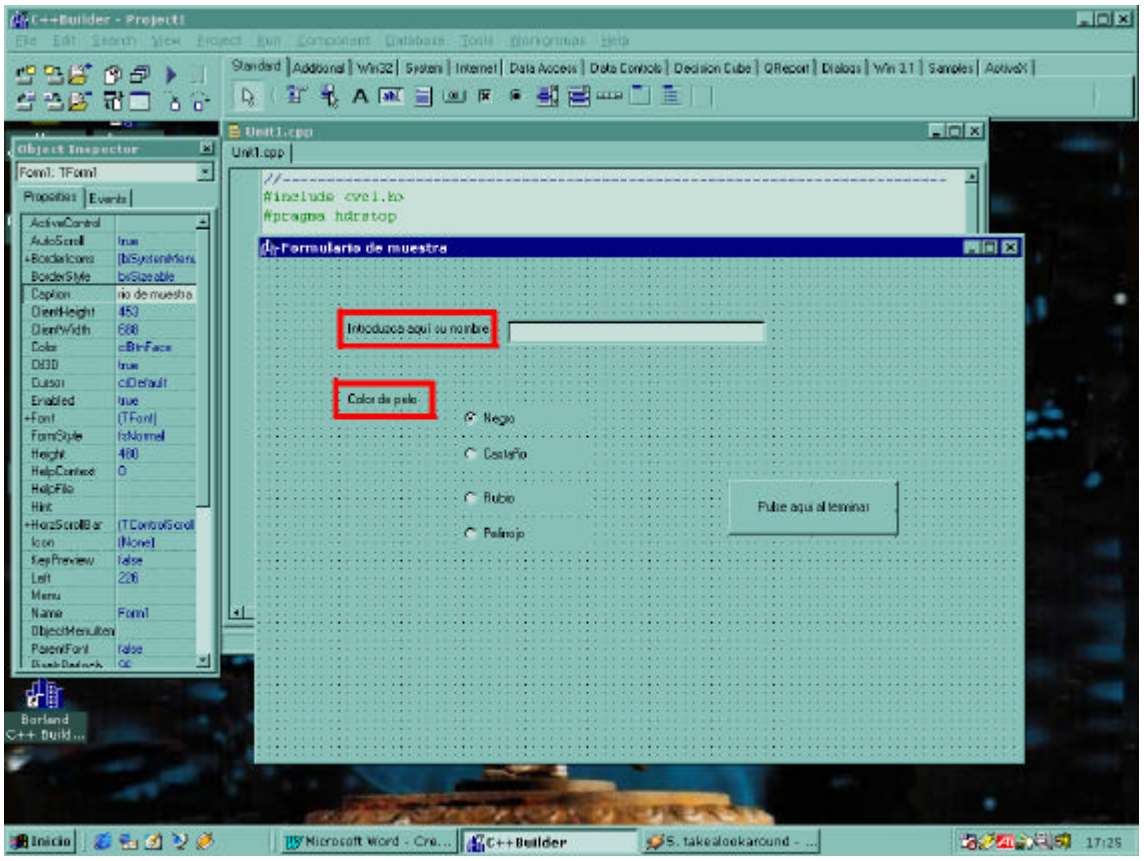

Figura 2.3.4.3: Cuadros de texto

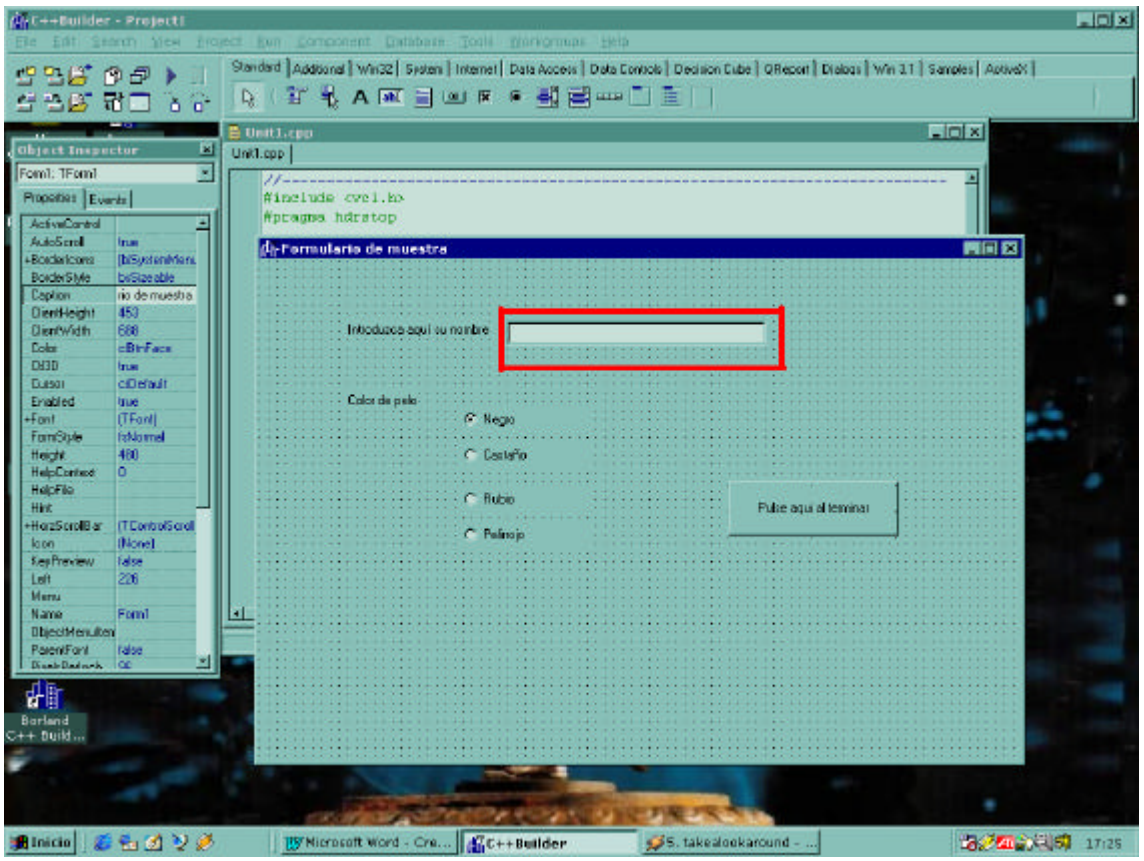

Figura 2.3.4.4: Cuadro de diálogo

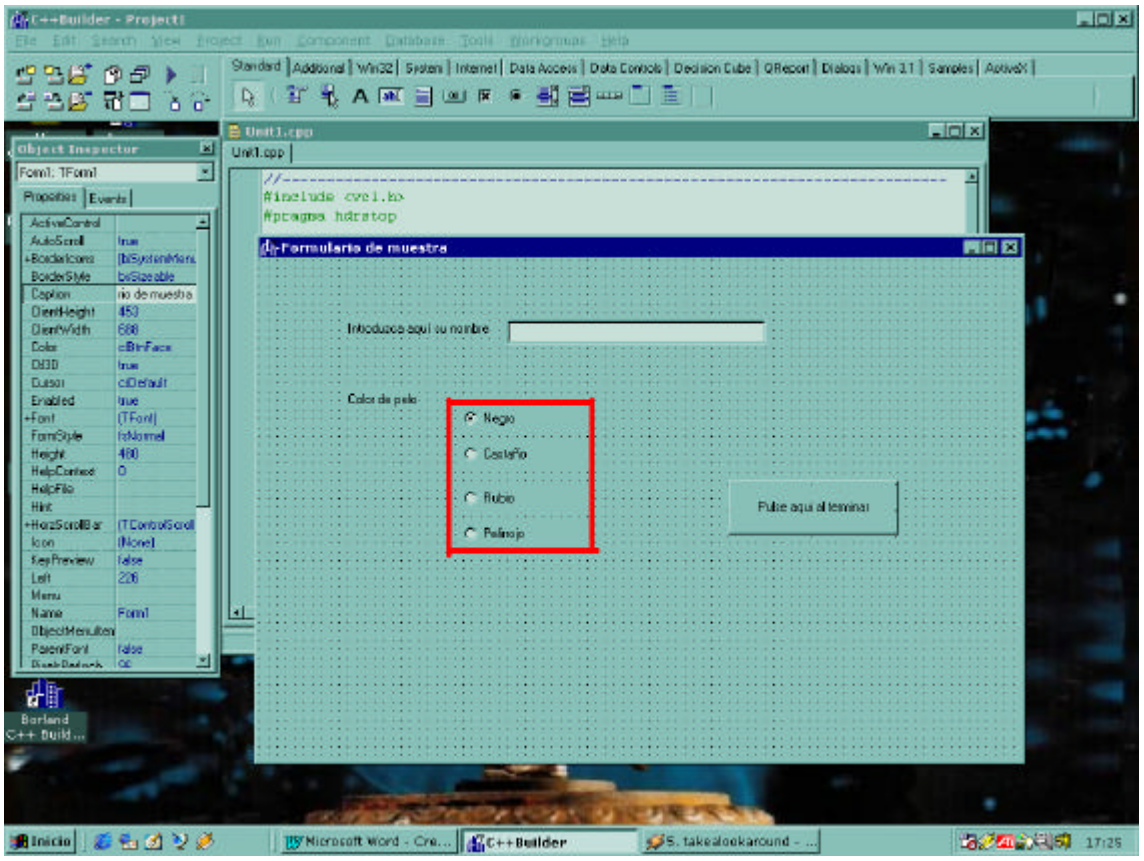

Figura 2.3.4.5: Botones de radio

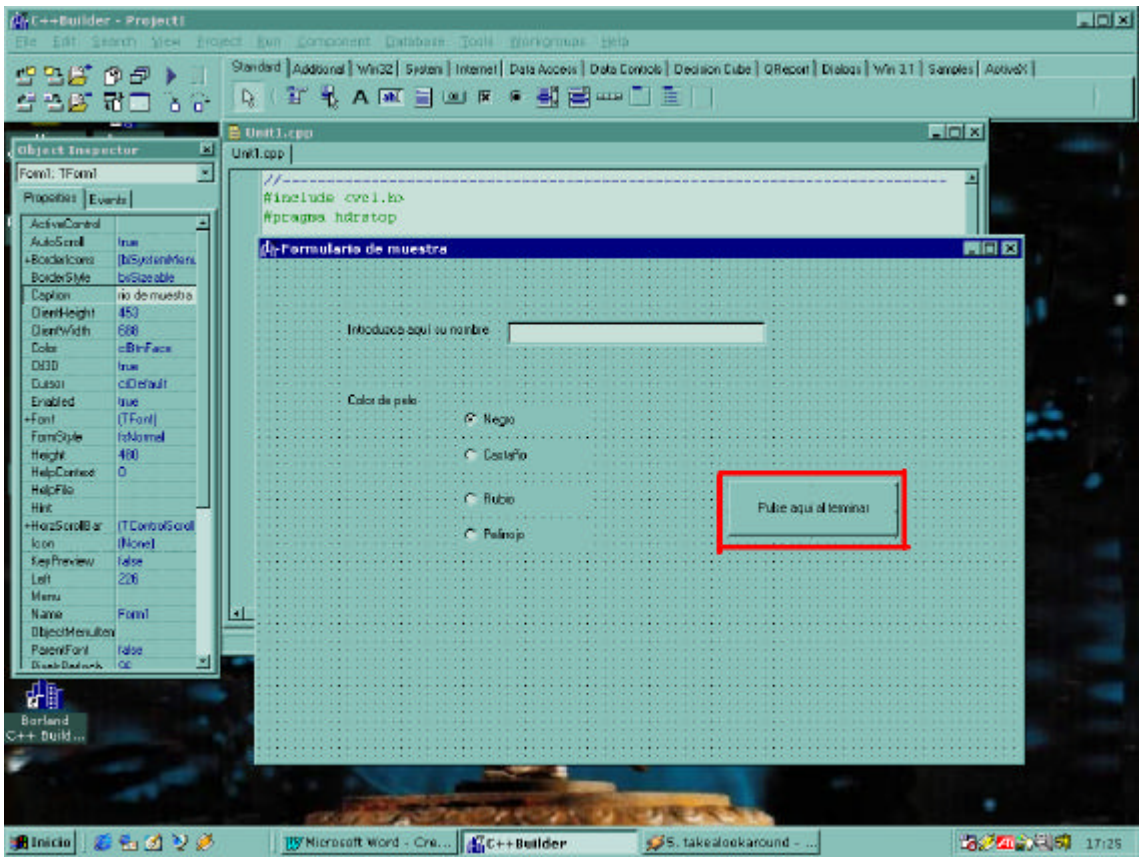

Figura 2.3.4.6: Botón

Para crear cuadros de texto se deben crear objetos de la clase *TLabel*. Los objetos de la clase *TEdit* se usan para crear cuadros de diálogo que pueden presentar texto o ser modificados por el usuario. Para crear un botón basta con crear un objeto de la clase *Tbutton* o *TbitButton*. La principal diferencia entre ellos dos es de tipo estética.

Los botones de radio se crean mediante objetos *TRadioButton*.

La VCL de Borland tiene prácticamente todos los objetos gráficos que pueden aparecer en cualquier aplicación. Pero además incluye una serie de clases que no son de tipo gráfico pero son muy útiles en cuanto a ahorro de tiempo y codificación. Un ejemplo claro es la clase *TList*. Esta clase implementa una lista enlazada cuyos miembros pueden ser de cualquier clase, aunque deben pertenecer todos a la misma clase. Además incluye una serie de métodos que permiten una sencilla gestión de dicha lista.

Volviendo a la figura, bajo la ficha está el editor de código, sobre el que C++ Builder va escribiendo automáticamente el código necesario. A la izquierda está el inspector de objetos, con el que se pueden modificar las propiedades y asociar los eventos a funciones para cada uno de los objetos que se usen en la aplicación.

Normalmente cualquier aplicación de cierta envergadura contendrá varias fichas que además no deberán mostrarse simultáneamente en pantalla, sino que deberán ir apareciendo según los eventos que se produzcan.

Cuando se crea una ficha en C++ Builder desde el IDE, se crea automáticamente la ficha en memoria incluyéndola en *WinMain()*, es decir, la ficha aparece cuando se inicia el programa y permanece en memoria mientras dura la ejecución de la aplicación.

Si no se desea que todas las fichas se muestren simultáneamente se puede optar por definir la propiedad *Show* de cada ficha como *False*. Cuando se desee mostrar una ficha basta con poner esta propiedad a *True* y ponerla de nuevo como *False* cuando se desee borrar.

Este método tiene un inconveniente, y es que la ficha, aunque no se ve, está en memoria, es decir, está usando recursos del sistema. Esto es inaceptable en grandes aplicaciones con muchas fichas. Afortunadamente, C++ Builder proporciona un método que permite que las fichas se carguen en memoria sólo cuando sea necesario.

Desafortunadamente hay otro problema relacionado con la creación de fichas desde el IDE. Hasta ahora no se ha tenido en cuenta que la resolución de la pantalla con la que trabaja el desarrollador puede ser distinta de la resolución de la que puede disponer el usuario. En la figura 2.3.4.1 puede observarse que el tamaño y la posición de cada uno de los objetos visuales se determina usando los píxeles de la pantalla como unidad de medida.

Esto implica que cuando un usuario con una resolución distinta a la que usó el desarrollador ejecute la aplicación la presentación de los interfaces de usuario en pantalla puede ser muy distinta de la que éste diseñó en un principio.

Esto implica que cada vez que se dibuje en pantalla un objeto visual hay que ajustar su posición y su tamaño en función de la resolución de la pantalla en la que se esté mostrando. Para ello habría que modificar cada uno de los objetos visuales que se crean.

Por otra parte, anteriormente se comentó que no es conveniente tener todas las fichas en memoria y que es aconsejable irlas cargando según sea necesario.

Recapitulando, habría que diseñar las fichas, implementar el método que se aconseja en el manual de Borland para hacer que las fichas se carguen en memoria según sea necesario, y a continuación modificar el tamaño y la posición de todos los objetos visuales con el fin de adecuarlos a la resolución de la pantalla.

En definitiva, el formulario que se presente en pantalla realmente no va a tener mucha relación con el que diseñó el desarrollador.

Por otra parte, todos los objetos visuales que se añaden a las fichas desde las paletas de componentes del IDE de C++ Builder no son más que instancias de clases. Se puede pensar en crear y destruir dichas clases con los comandos *new* y *delete* de C++, según sea necesario. Esto es, escribiendo directamente el código necesario, sin diseñar el formulario. De esta manera, además, al crearlos se puede calcular el tamaño y la posición de cada uno de ellos de acuerdo a la resolución de la pantalla, con lo que se mostrarían de una forma muy parecida independientemente de dicha resolución.

Otra ventaja adicional es que, dentro de una ficha, se puede seleccionar, según las decisiones que haya ido tomando el usuario, qué objetos mostrar en la ficha y cuáles no.

Por poner un ejemplo, se supone que un usuario abre cierto fichero de disco en modo sólo lectura. En este caso no es necesario mostrar un botón de *Guardar Cambios*. No solo no es necesario mostrarlo, sino que no es necesario crearlo, con lo que se ahorra memoria del sistema.

En realidad, al crear componentes en tiempo de ejecución escribiendo directamente el código no se está haciendo nada que no se pudiera hacer anteriormente con C++ Builder, pero dadas las condiciones particulares de la aplicación parece que este método es más intuitivo y ordenado: cuando hace falta cierto objeto se crea y cuando ya no sea necesario, se elimina. Además se pueden desarrollar funciones que creen dinámicamente el objeto con el tamaño adecuado y en la posición adecuada de manera que no sea necesario diseñar cada una de las fichas que se van usar. Se podría incluso crear fichas con un número arbitrario de objetos, totalmente en función de las condiciones o gustos del usuario, ya que no se tiene que diseñar previamente cada una de ellas.

Como se puede suponer, en la aplicación se han usado componentes que se crean y destruyen en tiempo de ejecución, y dichos componentes se han creado escribiendo directamente el código con el editor, sin usar el editor de fichas.

Pero además del problema del consumo de memoria que provocan las fichas si no se cargan dinámicamente, es decir, en tiempo de ejecución, había otro problema. En efecto, todavía no se ha expuesto el método usado en la aplicación para adaptar la posición y los tamaños de los objetos visuales a la resolución de la pantalla.

Si se recuerdan las figuras 2.3.4.2 a 2.3.4.6, donde se muestra cada uno de los objetos visuales que se dibujaron en la ficha de ejemplo de la figura 2.3.4.1, se puede apreciar como la ficha es un objeto de tipo formulario (*TForm*) y como todos los demás están situados sobre éste. En realidad son objetos “hijo” del objeto formulario, que es el padre (*Parent*).

Los objetos hijo se posicionan con coordenadas relativas respecto al padre, y es totalmente imposible que un objeto hijo sobresalga por los límites del padre. En ese caso sólo se representaría en pantalla la porción del objeto hijo que está dentro de los límites del padre.

El hecho de tener objetos que son padres de otros no tiene relación alguna con la herencia de clases de C++.

El método que se ha seguido para ajustar el tamaño y la posición de los objetos visuales según la resolución de la pantalla parte de una idea sencilla pero que se ajusta perfectamente a las necesidades de la aplicación.

La aplicación tiene varios formularios principales. Uno de ellos es el interfaz de la red vial, el que aparece al ejecutar la aplicación y sobre el que se representa la red vial. Los demás son cada uno de los interfaces de usuario que se presentan cuando se pulsa el botón adecuado en el formulario de herramientas

Al crear uno de estos formularios se pone el valor de la propiedad *WindowState* como *wsMaximized*. Esto hace que al crearse el formulario ocupe toda la pantalla. Con estos

formularios no hay problemas con la resolución, puesto que se adaptan a la de la pantalla.

El problema surge al añadir objetos hijo a uno de estos formularios. Lo que se hace es partir de un diseño proporcional de los interfaces. La idea es que cuando se diseñe un formulario, incluyendo sus imágenes, sus botones, sus cuadros de diálogo, no se usen coordenadas absolutas, es decir, el botón está a 200 píxeles del borde superior de la pantalla, a 50 píxeles del borde izquierdo y tiene 20 píxeles de alto y 50 de ancho. Si se diseña así el formulario es posible que al representarlo el aspecto no sea el deseado. Hay que usar de alguna manera coordenadas y dimensiones relativas, proporcionales al tamaño del formulario padre.

Lo que se hace es dividir el formulario en una serie de filas y columnas, de manera que cada objeto ocupe varias casillas. Cada una de estas filas y columnas tendrá unas ciertas dimensiones que se calculan como un porcentaje de las dimensiones totales del formulario, es decir, puede haber una fila cuyo alto sea el 10% del alto del formulario padre y cuyo ancho sea el 20% del ancho del formulario padre, por ejemplo. Es similar al método en que se ponen los barcos en el popular juego de los barquitos, donde cada jugador debe acertar las casillas en las que el contrario ha colocado sus barcos, con la diferencia de que en este juego el porcentaje para el tamaño de las casillas es igual para todas ellas.

El objeto padre de cada uno de los objetos visuales que se añadan es el formulario, del que se pueden conocer la anchura y la altura, ya que son propiedades que están accesibles. De esta manera se hace una división imaginaria del formulario en filas y columnas, de manera que los límites de cada casilla se corresponden con unas coordenadas relativas al formulario padre. Cuando se cree un objeto hijo, se le pasan como posición y tamaño las coordenadas adecuadas de manera que aparezca representado ocupando las casillas que se desean.

En la figura 2.3.4.7 se aprecia cómo un formulario se divide en casillas y columnas para luego representar los componentes.

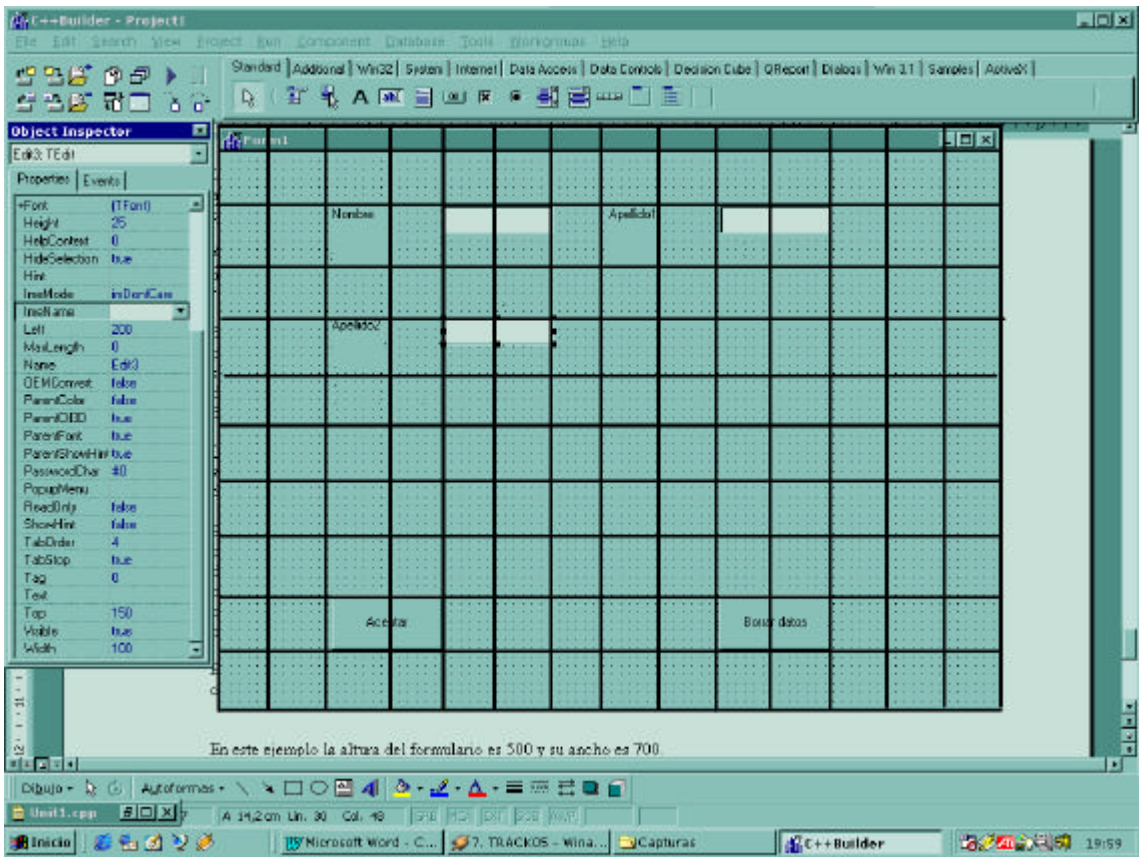

Figura 2.3.4.7: Cuadriculado para división de formulario

En la figura 2.3.4.7 el ancho del formulario es 700 y la altura es 500. Estos datos se refieren al área del formulario donde se pueden emplazar objetos, ya que se reserva una cierta cantidad para los bordes y otra para la barra de título

En este caso se ha optado por asignar el mismo porcentaje a cada una de las filas y columnas, pero no tiene que ser así. Lo importante es que todos los tamaños y posiciones estén dados en porcentajes respecto a las dimensiones totales del formulario.

Ejemplo de creación de componentes en tiempo de ejecución

A continuación se muestra un sencillo ejemplo de cómo crear componentes en tiempo de ejecución. Desde luego es mucho más sencillo que los que casos que aparecen en la aplicación, pero es suficiente para mostrar todo lo que es necesario saber para adentrarse un poco en la creación de componentes en tiempo de ejecución.

Cuando se crea una aplicación en Borland se crea una nueva ficha en blanco. Los componentes de esta ficha pueden crearse en tiempo de ejecución o usando la plantilla de componentes. La definición de las funciones a ejecutar ante ciertos eventos puede hacerse con el *Object Inspector* o de nuevo en tiempo de ejecución.

En el ejemplo siguiente se parte del formulario inicial (la ficha en blanco). En ella se creará un mensaje de texto y tres botones, todo en tiempo de ejecución. También en tiempo de ejecución, porque además no habría otra forma posible, se van a asignar a cada uno de los botones las funciones que deben ejecutar cuando tengan lugar una pulsación del ratón sobre ellos (evento *OnClick*) . Cuando se crean componentes en

tiempo de ejecución la asociación entre funciones a ejecutar y los eventos también debe hacerse en tiempo de ejecución. No puede hacerse con el *Object Inspector* puesto que el objeto no está aún creado ya que, se insiste, éste se crea en tiempo de ejecución.

El ejemplo imprime aproximadamente en el centro del formulario el mensaje “Pulsa alguno de los botones para moverme”. Debajo aparecen dos botones simétricos, uno de ellos con los símbolos “<<” y otro con los símbolos “>>”. Un poco más abajo y de nuevo centrado aparece un botón con el texto “Salir”.

La posición de cada uno de estos objetos (mensaje y botones) es independiente del tamaño del formulario, ya que su posición está calculada con el método de dividir el área del formulario en una matriz imaginaria de filas y columnas donde cada objeto tiene su posición determinada por la fila y columna que ocupa y el porcentaje del tamaño total que tiene cada una de las filas y columnas. El tamaño de estos objetos también se ajusta proporcionalmente al tamaño del formulario. Todo esto se verá con más detalle cuando se muestre el código. En la figura 2.3.4.8 se muestra el aspecto del formulario inicialmente.

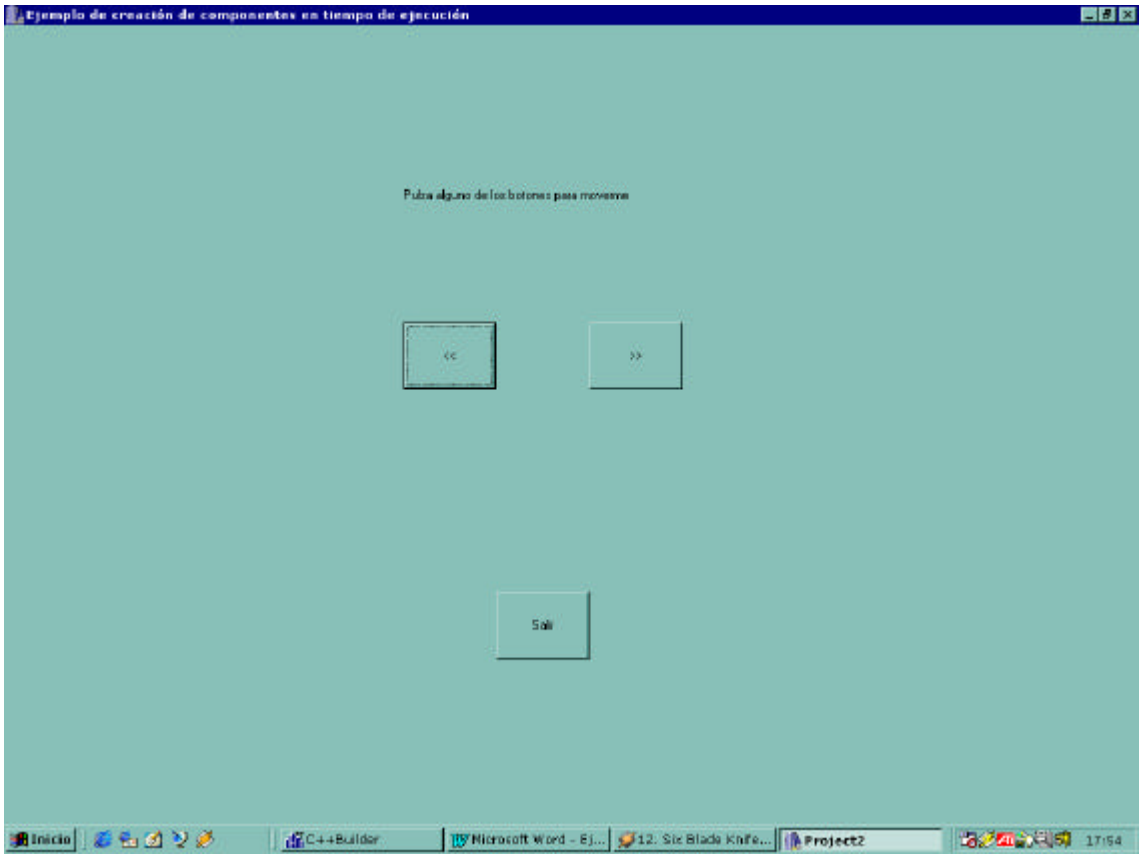

Figura 2.3.4.8: Programa de ejemplo

El funcionamiento de este ejemplo es bastante sencillo: al pulsar sobre el botón “<<” el mensaje se desplaza hacia la izquierda y al pulsar el botón “>>” se desplaza hacia la derecha. Si se pulsa el botón “Salir” se abandona el programa.

En la figura 2.3.4.9 se muestra el aspecto del formulario tras pulsar algunas veces el botón “>>”.

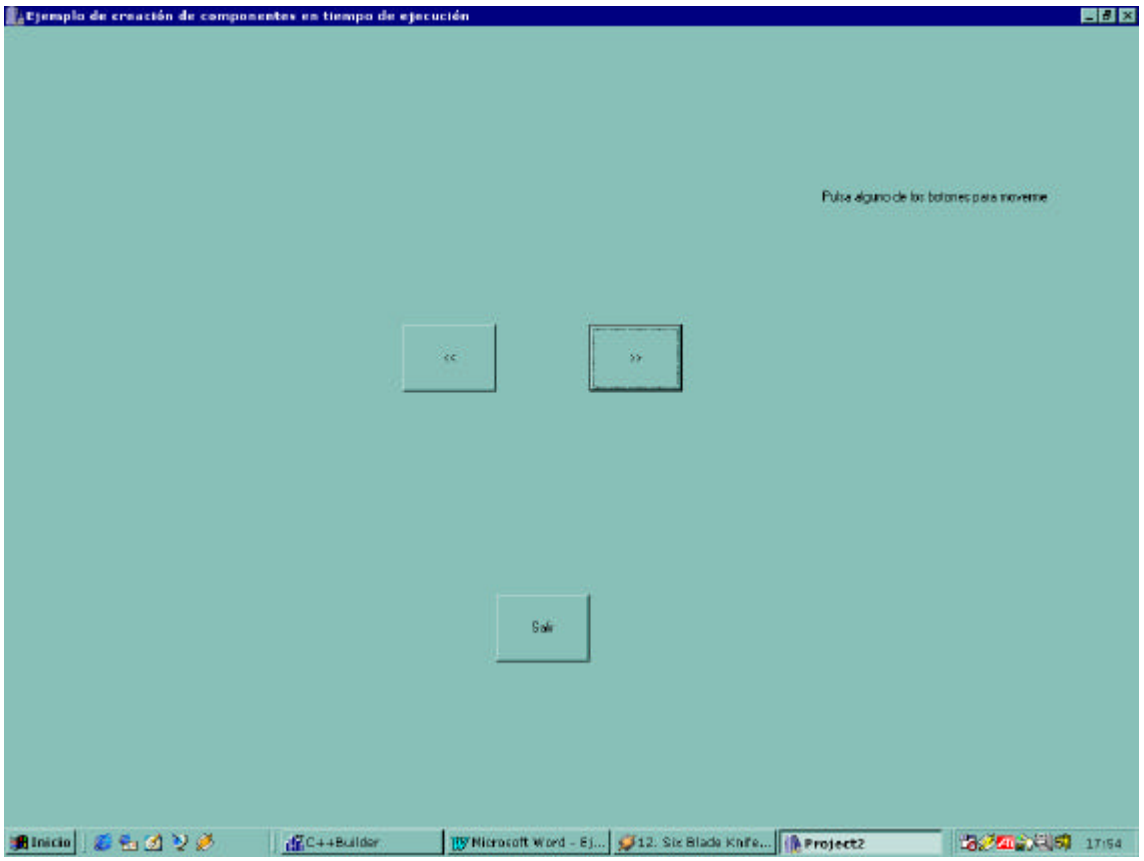

Figura 2.3.4.9: Formulario tras haber pulsado varias veces el botón “>>”

En este ejemplo no se ha especificado la posición ni el tamaño del formulario, sino que estos valores se ajustan automáticamente para adaptarse a las dimensiones de la pantalla. Esto implica que el aspecto de la aplicación de ejemplo es totalmente independiente de la resolución de la pantalla del equipo en el que se ejecuta dicha aplicación.

En el desarrollo del programa se han utilizado las mismas técnicas que en este ejemplo: los formularios aparecen todos maximizados y sus dimensiones no se pueden variar, si bien esto se consigue con métodos distintos según el formulario del que se trate. Esto también se verá más claramente sobre el código, que es expone a continuación.

El código necesario para programar esta aplicación en tiempo de ejecución se compone de tres ficheros.

Fichero principal del proyecto

En este fichero está la función *WinMain()*. Este fichero es creado íntegramente por C++ Builder y no es necesario modificarlo. En realidad, lo mejor sería no modificarlo a menos que se sepa con exactitud lo que se está haciendo.

```
//---------------------------------------------------------------
---
#include <vcl.h>
```

```
#pragma hdrstop
USERES("Project2.res");
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------------
---
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------
---
```

Fichero Unit1.h

Este fichero incluye las declaraciones necesarias y la clase *TForm1*, el formulario de partida.

```
//---------------------------------------------------------
---
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
---
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------
---
class TForm1 : public TForm
{
__published:   // IDE-managed Components

    void __fastcall FormCreate(TObject *Sender);
private:  // User declarations
public:        // User declarations
    __fastcall TForm1(TComponent* Owner);


    // Eventos para los botones
 void  __fastcall adios (TObject *Sender);
```

```
 void  __fastcall mueve_izquierda (TObject *Sender);
 void  __fastcall mueve_derecha (TObject *Sender);

};


//---------------------------------------------------------
---
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
---
#endif
```

Este fichero lo crea C++ Builder. Sin embargo, en este caso sí debe ser modificado. En concreto es necesario añadir las declaraciones de cada una de las tres funciones que se asociarán a los eventos *OnClick* de cada uno de los botones. Los eventos *OnClick* se producen cuando el usuario hace click con el ratón sobre el objeto en concreto, en este caso cuando se hace click sobre cada uno de los botones.

Las funciones deben declararse dentro de la clase del formulario, porque los eventos se producirán sobre dicho formulario (en realidad sobre objetos hijos del formulario), pero será el formulario el que reciba los eventos.

Fichero Unit1.cpp

Es el fichero donde hay que implementar todas las funciones.

```
//---------------------------------------------------------
---
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------
---
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;


//Se crean los punteros necesarios para los objetos
gráficos
TButton *boton1;
TButton *boton2;
TButton *boton3;
TLabel *mensaje;



float ancho;
```

```
float alto;

//---------------------------------------------------------
---
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------
---

void __fastcall TForm1::FormCreate(TObject *Sender)
{

// Cálculo de cuadricula para emplazar objetos, incluyendo
un borde
float columnas=8;
float filas=8;
//Se suponen todas las filas y columnas de igual tamaño
entre sí

Form1->Caption="Ejemplo de creación de componentes en
tiempo de ejecución";
Form1->WindowState=wsMaximized;
Form1->Position=poDefaultSizeOnly;
float borde=Form1->ClientHeight/30;
ancho=(Form1->ClientHeight-2*borde)/columnas;
alto=(Form1->ClientWidth-2*borde)/filas;

// Creación de los objetos gráficos
boton1=new TButton(Form1);
boton2=new TButton(Form1);
boton3=new TButton(Form1);
mensaje=new TLabel(Form1);

//Asignación del padre de cada uno
boton1->Parent=Form1;
boton2->Parent=Form1;
boton3->Parent=Form1;
mensaje->Parent=Form1;

// Tamaño de los objetos
boton1->Height=alto/2;
boton1->Width=ancho;
boton2->Height=alto/2;
boton2->Width=ancho;
boton3->Height=alto/2;
boton3->Width=ancho;
mensaje->Height=alto;
mensaje->Width=ancho;
```

```
// Emplazamiento de los objetos
mensaje->Left=borde+ancho*4;
mensaje->Top=borde+alto;
boton1->Left=borde+ancho*4;
boton1->Top=borde+alto*2;
boton2->Left=borde+ancho*6;
boton2->Top=borde+alto*2;
boton3->Left=borde+ancho*5;
boton3->Top=borde+alto*4;

//Leyendas de los botones
boton1->Caption="<<";
boton2->Caption=">>";
boton3->Caption="Salir";

// Texto del mensaje
mensaje->Caption="Pulsa alguno de los botones para
moverme";

// Debe asegurarse que se vean
mensaje->Visible=true;
boton1->Visible=true;
boton2->Visible=true;
boton3->Visible=true;

// Asignación de eventos
boton1->OnClick=mueve_izquierda;
boton2->OnClick=mueve_derecha;
boton3->OnClick=adios;
}
//---------------------------------------------------------
---
void  __fastcall TForm1::adios (TObject *Sender)
  {

   delete (boton1);
   delete (boton2);
   delete (boton3);
   delete (mensaje);
   Form1->Close();
  }
//---------------------------------------------------------
---
void  __fastcall TForm1::mueve_izquierda (TObject *Sender)
  {
  mensaje->Left=mensaje->Left-ancho/2;
  }
//---------------------------------------------------------
---
void  __fastcall TForm1::mueve_derecha (TObject *Sender)
  {
```

```
    mensaje->Left=mensaje->Left+ancho/2;
  }
```

Este es el fichero donde prácticamente se realiza toda la programación. En realidad todo el código aparece dentro del método de *TForm1* llamado *FormCreate.* Esto es así porque esta función se ejecuta al crearse cualquier formulario de la clase *TForm1,* y *Form1* pertenece a esa clase. *Form1* es el nombre del formulario que creó C++ Builder al iniciar el proyecto.

En primer lugar se definen dos variables de tipo *float* que en realidad funcionan como dos constantes, ya que sirven para determinar las filas y columnas imaginarias en que se va a dividir el área de cliente del formulario. El área de cliente del formulario es el área que se puede usar del mismo, ya que tiene un cierto espacio reservado implícitamente para los bordes y para la barra de título.

A continuación se le da el título al formulario asignando el valor adecuado a la propiedad *Caption* del mismo. Al asignar el valor *wsMaximized* a la propiedad *wsWindowState* se consigue que el formulario aparezca maximizado, es decir, ocupando toda la pantalla. Asignando el valor *poDefaultSizeOnly* a la propiedad *Position* se consigue que la posición del formulario sea lo más cercana posible a los bordes de la pantalla, pero con el tamaño que ya se le ha asignado, es decir, maximizado. Esto es necesario porque si sólo se asigna el valor *wsMaximized* a la propiedad *wsWindowState,* si bien se consigue que el formulario ocupe toda la pantalla, el área de cliente de dicho formulario no es toda la pantalla, sino un valor por defecto. Esto provocaría errores en el cálculo de las posiciones y tamaños de objetos, ya que el área de cliente del formulario estaría falseada y sería menor de lo que debe ser en realidad.

Como se ha dicho anteriormente, en la aplicación, al igual que en este ejemplo, todos los formularios ocupan toda la pantalla y sus dimensiones no se pueden modificar. Pero cuando los formularios se crean desde otro formulario ya existente, cosa que ocurre para todos los formularios de la aplicación excepto el primero, ya no es necesario realizar estos dos pasos, sino que es algo más sencillo y transparente.

En efecto, si el primer formulario ya está maximizado, se conocen las dimensiones necesarias para que un formulario aparezca maximizado. Estas dimensiones son los valores de las propiedades *Height* y *Width* de dicho formulario. Por lo tanto lo que se hace es crear el nuevo formulario con las dimensiones adecuadas que son conocidas.

Para evitar que estos valores puedan ser modificados se le asigna el valor *bsDialog* a la propiedad *BorderStyle* del formulario.

De esta manera todos los formularios de la aplicación aparecen maximizados y se ajustan automáticamente a las dimensiones (resolución) de la pantalla. Además, como ya se ha explicado, y se detallará posteriormente en el código de este ejemplo, todos los objetos que se creen sobre el formulario también se adaptarán a dichas dimensiones.

Al igual que las variables *filas* y *columnas* se define la variable *borde*. No es más que un porcentaje del área de cliente que se reservará como libre, para que los objetos no

aparezcan demasiado cerca de los límites del formulario. A continuación se almacena en las variables *alto* y *ancho* las dimensiones de cada una de las cuadrículas imaginarias en que queda dividido el formulario, teniendo en cuenta el espacio reservado a los bordes. Puede considerarse como una matriz de dimensión *filas* x *columnas* donde todos los elementos ocupan el mismo espacio. Cada uno de los objetos que se creen sobre el formulario puede ocupar el espacio de uno o varios elementos, la división en filas y columnas tiene como único propósito mantener la relación de proporcionalidad.

Una vez ajustada la geometría del formulario y de las cuadrículas se deben crear cada uno de los objetos. Se crean tres botones, que son objetos de la clase *TButton* y una línea de texto, que es un objeto de la clase *TLabel.* Además a cada uno de estos objetos se les debe asignar un padre (*Parent*) para que se sepa dónde deben emplazarse, en este caso sobre el formulario *Form1*.

A continuación se define la geometría de cada uno de los objetos creados, por supuesto en función del tamaño de la cuadrícula imaginaria, para preservar las proporciones.

Seguidamente se debe definir la posición donde se emplazará cada objeto. Para cada elemento se necesitan dos valores: uno indica el desplazamiento del objeto respecto al borde izquierdo del comienzo del área de cliente del formulario, y se debe almacenar en la propiedad *Left* de dicho objeto. El otro valor es el desplazamiento del objeto respecto al borde superior del área de cliente del formulario y debe almacenarse en la propiedad *Top* del mismo.

Puede verse en detalle la posición de, por ejemplo, uno de los botones:

```
boton1->Left=borde+ancho*4;
boton1->Top=borde+alto*2;
```

En primer lugar, puede observarse como tanto para la propiedad *Left* como para *Top*, se ha tenido en cuenta el espacio que se ha querido mantener despejado desde el borde del área de cliente del formulario. Este espacio libre viene dado por el valor de la variable *borde*, como puede observarse.

Y las coordenadas de cada una de las propiedades se ha calculado teniendo en cuenta la división en cuadrículas. Así, el botón estará en la segunda fila y en la cuarta columna.

No hay que olvidar el texto que se desea que aparezca sobre cada botón y el texto de que consta la frase. Todo esto se consigue modificando adecuadamente la propiedad *Caption* de cada uno de los objetos.

Para acabar con el diseño de todos estos objetos se debe hacer que sean visibles. Para asegurarlo se le asigna el valor *true* a las propiedad *Visible* de cada uno de ellos.

Y para finalizar, se le debe asignar una función a cada evento *OnClick* sobre cada uno de botones. La propiedad *OnClick* es un puntero a función. Dicha función se ejecutará cuando tenga lugar el evento *OnClick* sobre el botón apropiado.

Así, cuando se pulse el botón con el símbolo “>>” se ejecutará la función *mueve_derecha*, cuando se pulse el botón marcado como “<<” se ejecutará la función *mueve_izquierda* y cuando se pulse el botón “Salir” se ejecutará la función *adios*.

Tanto la función *mueve_derecha* como la función *mueve_izquierda* son bastante sencillas. Lo único que hacen es desplazar la posición de la línea de texto hacia la derecha o hacia la izquierda, respectivamente.

Es importante resaltar el hecho de que en todo el ejemplo no se ha utilizado nunca un valor fijo en píxeles para representar ni la posición ni el tamaño de cualquiera de los objetos que se crean. Por el contrario, todas las posiciones y dimensiones de cada uno de los objetos que aparecen se han calculado proporcionalmente a las dimensiones del área de cliente del formulario, en definitiva, proporcionalmente a la resolución de la pantalla, ya que el formulario se adapta a las dimensiones de ésta.

Esto permite asegurar que la aplicación mantendrá este aspecto independientemente de las características del sistema en que se ejecute.

Creación de componentes en tiempo de ejecución en la aplicación

En la aplicación todos los componentes se crean en tiempo de ejecución menos los dos formularios iniciales. En principio, esto provocaría que hubiese una gran cantidad de líneas de código dedicadas exclusivamente a estos quehaceres.

Aprovechando la sobrecarga de funciones de C++, se han creado una serie de funciones, llamadas *Crear_componente,* cuya función es la de crear componentes en tiempo de ejecución. Naturalmente que se ha tenido que desarrollar una función distinta para cada tipo de objeto que se desea crear, pero la sobrecarga de funciones permite poder usar la llamada a la función unifórmente, sin usar una función distinta para cada clase.

A continuación se listan las funciones *Crear_componente* sobrecargadas de las que se dispone:

*void Crear_componente (TWinControl *formulario,**TMaskEdit** **elemento,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TTreeView** **hijo, int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TStringGrid** **hijo, int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TImage** **lista, int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TPageControl** **hijo,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *formulario,**TShape** **elemento,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TListBox** **lista,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *semejante,**TTabSheet** **hijo);*

*void Crear_componente (TWinControl * padre,**TForm** **formulario,int left,int top,int Ancho,int Alto);*

*void Crear_componente(TWinControl *semejante,**TComboBox** **hijo,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *formulario,**TLabel** **label,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *formulario,**TBitBtn** **elemento,int left,int top,int Ancho,int Alto);*

*void Crear_componente (TWinControl *formulario,**TCheckBox** **elemento,int left,int top,int Ancho,int Alto);*
*void Crear_componente (TWinControl *formulario,**TGroupBox** **elemento,int left,int top,int Ancho,int Alto);*
*void Crear_componente (TWinControl *formulario,**TEdit** **elemento, int left,int top,int Ancho,int Alto);*
*void Crear_componente (TWinControl *formulario,**TMaskEdit** **elemento,int left,int top,int Ancho,int Alto);*
*void Crear_componente (TWinControl *semejante,**TUpDown** **hijo,int left,int top, int Ancho,int Alto);*
*void Crear_componente (TWinControl *formulario,**TScrollBar** **caja,int Left,int Top,int Ancho,int Alto,int estilo);*
*void Crear_componente (TWinControl *semejante,**TMemo** **hijo,int left,int top,int Ancho,int Alto);*
*void Crear_componente (TWinControl *formulario,**TToolButton** **caja,TToolBar *barra);*
*void Crear_componente (TWinControl *formulario,**TToolBar** **caja,int Left,int Top,int Ancho,int Alto,bool flat);*
*void Crear_componente (TWinControl *formulario,**TImageList** **caja);*
*void Crear_componente (TWinControl *formulario,**TBevel** **caja,int tipo,int Left,int Top,int Ancho,int Alto);*

Todas estas funciones reciben los mismos parámetros, excepto el segundo de ellos, que difiere en cada una de dichas funciones. Además la última de ellas recibe un parámetro extra: *int tipo*, y la penúltima recibe menos parámetros. Salvo estas excepciones, todas reciben los mismos argumentos.

El primero de los argumentos que reciben estas funciones es un puntero al objeto Padre al que se quiere crear. Esto es necesario porque tras crear un objeto hay que asignarle un objeto Padre, se trata de la propiedad *Parent*.

Los cuatro últimos argumentos, *Left, Top, Ancho* y *Alto*, definen las características geométricas del objeto a crear. Ya se ha expuesto anteriormente en esta misma sección el significado de cada una de estas propiedades.

Por último, el segundo argumento. Éste argumento es el que permite designar a C++ la función que se debe usar en cada caso, pues es el único argumento que puede ser diferente, salvo las excepciones ya enumeradas.

Este argumento es un puntero a puntero a la clase del objeto que se quiere crear. Esto quiere decir que mirando el segundo argumento puede saberse que objetos pueden crearse usando estas funciones. Por ejemplo, observando la primera de las funciones puede saberse que para crear un objeto de la clase *TMaskEdit* no es necesario crearlo con el operador **new** y a continuación modificar cada una de sus propiedades, sino que se puede crear usando la función *Crear_componente.* Esta función, como se ha comprobado, permite que se le pase como parámetro las características geométricas del objeto a crear, por lo que el desarrollador se ahorra algunas líneas de código.

El hecho de que el segundo argumento sea un puntero a puntero no es causal. Aunque en C++ se pueden pasar parámetros como referencia, al codificar la aplicación se decidió usar los punteros.

Normalmente que tiene como función crear un interfaz gráfico para el usuario debe crear un formulario y dentro de éste todos los objetos necesarios: botones, imágenes, barras de desplazamiento, etc... Se puede poner como ejemplo la clase *Contexto.* Se recuerdan las propiedades principales de esta clase:

```
TImage *imagen_min;
TImage *imagen_aux;
TImage *imagen;
TBevel *borde;
TScrollBar *horizontal;
TScrollBar *vertical;
```

La clase contiene punteros a objetos de tipo imagen (*TImage**), punteros a objetos de tipo “pedestal” (*TBevel**) y punteros a objetos de tipo barra de desplazamiento (*TScrollBar**). No contiene ninguno de estos objetos, sino punteros. ¿Por qué?

La respuesta es sencilla. Todos los componentes se crean en tiempo de ejecución. Así, para tener realmente una imagen que pertenezca a la clase *Contexto* basta con hacer:

```
imagen_min= new TImage;
```

Lo que ocurre es que prácticamente todos los objetos que hagan falta se van a crear usando la función *Crear_componente.* En C++, al igual que en C, por defecto los parámetros se pasan por valor.

El código de la función *Crear_componente* sobrecargada para el caso de *TImage* es:

```
void Crear_componente (TWinControl *semejante,TImage **hijo,
                       int left,int top,int Ancho,int Alto)
{
```

```
 *hijo=new TImage (semejante);
 (*hijo)->Parent=semejante;
 (*hijo)->Left=left;
 (*hijo)->Top=top;
 (*hijo)->Height=Alto;
 (*hijo)->Width=Ancho;
 (*hijo)->Visible=true;
}
```

Teniendo en cuenta que la llamada a la función sería:
Crear_componente (form, &imagen_min,0,0,500,700);

Donde *form* es un puntero al formulario Padre y 0,0,500 y 700 son parámetros geométricos.

Dentro de la función se crea la imagen y se hace que el puntero a la imagen apunte a la misma (el contenido de un puntero a puntero a imagen es un puntero a imagen, claro).

Por lo tanto *hijo es un puntero imagen que contiene la dirección de la imagen recién creada. Todo funciona perfectamente.

Si por el contrario la función Crear_componente hubiese sido así:
*void Crear_componente (TWinControl *semejante,**TImage** *lista, int left,int top,int Ancho,int Alto);*

La llamada a la función tendría que ser:
Crear_componente (form, imagen_min,0,0,500,700);

Y el código de la función debería ser el siguiente:

```
void Crear_componente (TWinControl *semejante,TImage *hijo, int left,int top,int Ancho,int Alto)
{
 hijo=new TImage (semejante);
 hijo->Parent=semejante;
 hijo->Left=left;
 hijo->Top=top;
 hijo->Height=Alto;
 hijo->Width=Ancho;
 hijo->Visible=true;
}
```

¿Qué ocurre? Pues que dentro de la función se modifica el valor del argumento hijo. El resultado es que tras regresar de la función el puntero *imagen_min* sigue apuntando a donde apuntaba inicialmente (y allí no hay ninguna imagen) y se tiene espacio reservado en memoria para un objeto *TImage* al que no se puede acceder de ninguna manera, pues no se dispone de un puntero y por lo tanto tampoco se puede liberar la memoria que ocupa.

Como se ha visto, al llamar a la función *Crear_componente* para crear un objeto, se le debe pasar como parámetro las dimensiones y el tamaño de dicho objeto. Esto es muy

útil para crear, por ejemplo, formularios que son hijos de otro. De esta manera, conociendo el tamaño del formulario padre, el formulario hijo se crea de forma que ocupe el porcentaje de pantalla que interesa.

Y esto es aplicable a cualquier otra clase de objetos.

De todas maneras, en la aplicación la mayoría de los formularios aparecen maximizados.