Argentina $8,90 (recargo al interior $0,20) / México: $45

USERS

Microsoft

Curso teórico y práctico de programación

Desarrollador .net

Con toda la potencia de **Visual Basic .NET** y **C#**

La mejor forma de aprender a programar desde cero

★★★★★

Basado en el programa Desarrollador Cinco Estrellas de Microsoft

13

Enlace de datos

ADO.NET - El objeto BindingSource

Distribución de aplicación

XCOPY - Windows Installer - ClickOnce

ISBN 978-987-1347-43-8
00013
9 789871 347438

RedUSERS

COMUNIDAD DE TECNOLOGIA

¡SUMATE!

EL SITIO Nº1 DE TECNOLOGIA

Noticias al instante // Entrevistas y coberturas exclusivas // Análisis y opinión de los máximos referentes // Reviews de productos // Trucos para mejorar la productividad // Registrate, participa, y comparte tus opiniones

SUSCRIBITE

SIN CARGO A CUALQUIERA DE NUESTROS **NEWSLETTERS** Y RECIBÍ EN TU CORREO ELECTRÓNICO TODA LA INFORMACIÓN DEL UNIVERSO TECNOLÓGICO ACTUALIZADA **AL INSTANTE**

INGRESÁ A
redusers.com/suscribirse-al-newsletter
¡Y REGISTRATE YA!

Foros · Encuestas · Tutoriales · Agenda de eventos · Videos · ¡Y mucho más!

redusers.com

Seguinos en:
www.facebook.com/redusers
www.twitter.com/redusers
www.youtube.com/redusersvideos

Diseño de interfaz de usuario

Todas las claves, técnicas y secretos para hacer que nuestra aplicación sea sencilla y fácil de utilizar.

El diseño de la interfaz de usuario es de fundamental importancia en el uso de una aplicación. ¿Qué significa esto? Bueno, si los programas que usamos habitualmente estuvieran desorganizados, con los botones siempre en distinto lugar, con colores poco consistentes, con textos poco claros, se nos haría más difícil aprender a usarlos. Tomemos cualquier programa de los que tenemos instalados en nuestra computadora. Si observamos un poco, veremos que, por ejemplo, en las ventanas que tienen botones Aceptar y Cancelar, éstos siempre se encuentran en la misma ubicación y acomodados del mismo modo (en general, en la parte inferior derecha, con Aceptar más a la izquierda y Cancelar a su derecha). De este modo, cuando nos empezamos a acostumbrar, usamos la aplicación casi "de memoria", ya que sabemos que si queremos anular la acción actual, debemos hacer clic en el botón que está en la esquina inferior derecha, y estaremos seguros de ése es Cancelar.
Como programadores, siempre debemos tratar de imitar estas buenas prácticas de diseño de interfaces. Para hacerlo, Visual Studio nos proporciona un conjunto de herramientas visuales, propiedades y controles que nos permiten generar interfaces estándar y consistentes. Veamos las más importantes.

> El diseño de la interfaz de usuario es de fundamental importancia en el uso de una aplicación.

Snaplines

Una de las herramientas más sencillas, y más útiles, es *Snapline*. Las snaplines son líneas que nos permiten alinear los controles unos con otros y con respecto a su contenedor, de manera tal que queden bien posicionados en cuanto a sus márgenes y al alineamiento del texto que se va a ingresar en cada uno de ellos (si es aplicable). Estas líneas se dibujan automáticamente en el diseñador de formularios al momento de posicionar el control y arrastrarlo dentro de él. En Windows Forms .NET 2.0 todos los controles tienen una propiedad **Margin**, que es la distancia que existe alrededor del control.

Cuando dos controles se mueven uno cerca de otro, y la distancia es igual a la suma de sus márgenes, se muestra una snapline roja. Ésta es la distancia que recomiendan las guías de diseño de interfaces de usuario.
Además, los controles contenedores tienen

FIGURA 032 | Las líneas azules nos permiten alinear los controles entre sí. La roja es para la alineación de los textos respecto a otros controles.

una nueva propiedad llamada **Padding**, que es la distancia interna desde el borde del control contenedor hasta el del control que se está ubicando dentro de él. Una snapline también se muestra cuando se alcanza esta distancia.

Anchor y Docking

Así como las snaplines nos permiten alinear los controles en tiempo de diseño, con anchor y docking podemos fijarlos en tiempo de ejecución. Las propiedades **Anchor** y **Dock** de los controles nos permiten anclar o fijar los controles en una determinada posición y permitir su redimensionamiento de manera automática cuando el contenedor o el formulario cambian de tamaño. Por ejemplo, si colocamos un control TextBox, tal que su borde derecho quede junto al borde derecho del formulario, con Anchor podemos hacer que, cuando el usuario estire el formulario para hacerlo más grande, el TextBox también crezca, para mantenerse siempre junto al borde derecho de él.
Con Anchor, "anclamos" o fijamos un control con respecto a los bordes de su contenedor. Podemos controlar aspectos como que un botón que está en el extremo inferior derecho del formulario se ubique siempre ahí, independientemente del tamaño que el usuario le dé al formulario en tiempo de ejecución. En cambio, con Docking "pegamos" el control a alguno de los bordes (si lo "pegamos" a los cuatro bordes simultáneamente, ocupará toda la superficie de su contenedor). La diferencia entre Anchor y Docking radica, principalmente, en que con el primero podemos colocar los controles a cualquier distancia del borde de su contenedor y hacer que siempre la mantenga, mientras que con el segundo el control siempre se pega a los bordes.

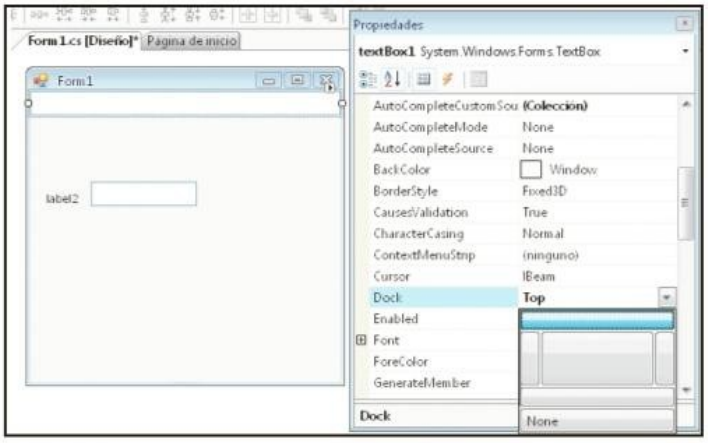


FIGURA 033 | Docking en acción. Al cambiar la propiedad Dock del control, éste se "pega" al borde seleccionado; en este caso, el superior.

Guías de diseño

Microsoft ha publicado un extenso documento con las Guías de Diseño Estándar; esto es, un conjunto de reglas, procedimientos y especificaciones que nos sirven como referencia para construir aplicaciones totalmente compatibles con el estándar de Windows. Hay desde reglas que indican la distancia en píxeles que debe haber entre cada botón, hasta las que determinan la paleta de colores que debemos usar para crear los iconos.

Layout Panels

Así como podemos utilizar Anchor y Docking para acomodar los controles en tiempo de ejecución en el formulario, a veces esto no es suficiente si las interfaces son complejas. Para ayudar en esta tarea, tenemos los paneles de disposición de controles o Layout Panels. Estos controles nos permiten armar complejos formatos de pantallas, de manera similar a como se haría en una página web: ubicándolos de forma que "fluyan",

es decir, que los controles se coloquen uno a continuación de otro mientras exista espacio disponible y, cuando se acabe, lo hagan en la línea siguiente. Además, al cambiar el tamaño del formulario, los controles se mueven para adaptarse al nuevo espacio. Para este caso utilizaríamos el control FlowLayoutPanel. En contraste con este modelo, y usando un **TableLayoutPanel**, podemos armar una disposición tabular; es decir, un esquema de filas y columnas en las cuales alinear los controles. Estas columnas pueden tener anchos absolutos (un tamaño fijo en píxeles) o relativos (un porcentaje del tamaño de su contenedor). De esta manera, al redimensionar el formulario, los controles también se irán reacomodando, pero respetando la estructura tabular. Para alinear internamente los controles dentro de las "celdas" del TableLayoutPanel debemos usar las propiedades Anchor y Docking mencionadas anteriormente.

Document Outline

Cuando utilizamos muchos controles en nuestras interfaces, a veces es necesario acceder a ellos de manera rápida y visualizar cómo están organizados en relación con el formulario. Esa funcionalidad la proporciona la ventana Document Outline, de Visual Studio. Con ella es posible ver la jerarquía de controles del formulario en forma de TreeView y, además, editar el nombre de esos controles. Para acceder a esta ventana, debemos ir al menú Ver/Otras Ventanas/Esquema del Documento.

Esta práctica herramienta también nos permite cambiar la ubicación de los controles, desplazándolos hacia arriba o hacia abajo e intercambiándolos con los demás. Incluso, si un control

> Con Anchor "anclamos" o fijamos un control con respecto a los bordes de su contenedor.

⊕ Un truco práctico

Si dentro de un FlowLayoutPanel colocamos varios controles contenedores (por ejemplo, Panels), todos con la propiedad Dock en Top, podremos ocultar cualquiera de esos contenedores, de modo que aquellos que le siguen subirán automáticamente, y así lograremos que no quede un espacio vacío en el formulario. Esto es muy útil cuando tenemos muchos controles, pero no siempre deben estar todos visibles.

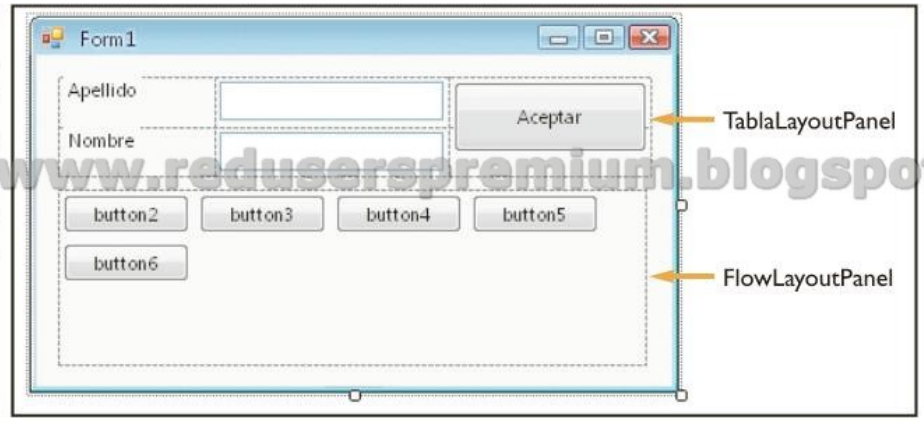


FIGURA 034 | Mediante estos dos controles, podemos trabajar más cómodamente con la disposición de los demás controles en pantalla.

Los controles de Layout nos permiten armar complejos formatos de pantallas.

se encuentra dentro de un contenedor, por medio de la opción Quitar del siguiente contenedor (flecha a la izquierda) podemos eliminarlo y colocarlo fuera, sin tener que cortar y pegar el control manualmente.

Herencia visual

Otro de los elementos que podemos usar para realizar interfaces de usuario consistentes, con el mismo comportamiento en uno o varios proyectos, es la herencia visual.

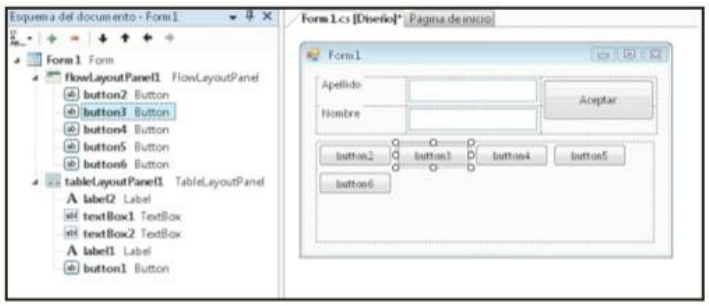


FIGURA 035 | Con esta herramienta, podemos ver rápidamente toda la jerarquía de controles del formulario actual, y moverlos, si es necesario.

Combinar Anchor y snaplines

Como vimos anteriormente, las Guías de Diseño especifican cómo deberíamos crear las interfaces. El uso combinado de snaplines para ajustar los márgenes y las distancias de separación apropiadamente, y de la propiedad Anchor nos permitirá cumplir muy bien con esas guías, y así lograr aplicaciones muy profesionales.

Un formulario no es más que una instancia de una clase System.Windows.Forms.Form, por lo que nada nos impide crear uno con los controles y comportamientos básicos que deseemos.

Podríamos crear un formulario con un par de botones predeterminados (Aceptar y Cancelar, por ejemplo), aplicarle algún diseño de tipo de letra y colores, y agregar métodos con funcionalidad predeterminada (seguridad, validaciones de campos, acceso a bases de datos, etc.), para luego utilizarlo como un formulario base a partir del cual heredarán todos los demás de la aplicación.

La manera más sencilla y práctica de realizar esta tarea es crear un proyecto Class Library y generar nuestro formulario allí. Luego, en nuestro proyecto, lo referenciamos y lo empleamos para crear los formularios basados en él.

Veamos cómo realizar herencia visual:

- Con Visual Basic Express Edition, creamos un nuevo proyecto del tipo Class Library (Biblioteca de Clases) y lo denominamos Base.
- Acto seguido, en el proyecto creado eliminamos la clase Class1, que se creó automáticamente, ya que sólo contendremos un formulario que nos servirá de base.
- Hacemos clic derecho del mouse sobre el proyecto en el Explorador de Soluciones, elegimos la opción Agregar/Windows Form… y, en la ventana que aparece, ponemos como nombre frmBase.vb (o frmBase.cs si trabajamos con C#).
- Ya tenemos nuestro formulario base. Ahora le agregaremos dos controles Button, que serán nuestros botones de Aceptar y Cancelar. Cambiamos su propiedad Text y Name para que coincidan. En este

momento, podemos utilizar las propiedades Anchor de cada control para anclarlos abajo a la derecha (Figura 36).

- En el evento Click de cada botón, ponemos un mensaje indicando la opción seleccionada (en una aplicación real, tendríamos el código necesario para la tarea que realice dicho botón).

```
'VB.Net
Private Sub btnAceptar_Click(ByVal sender As
Object, ByVal e As EventArgs) Handles
btnAceptar.Click
      MessageBox.Show("Aceptar del formulario
      base")
    End Sub

Private Sub btnCancelar_Click(ByVal sender As
Object, ByVal e As EventArgs) Handles
btnCancelar.Click
         MessageBox.Show("Cancelar del
         formulario base")
    End Sub

//C#
private void btnAceptar_Click(object sender,
EventArgs e)
{
    MessageBox.Show("Aceptar del formulario
    base");
}

private void btnCancelar_Click(object sender,
EventArgs e)
{
    MessageBox.Show("Cancelar del formulario
    base");
}
```

- Una vez creada la interfaz, haciendo clic derecho en el proyecto, dentro del Solution Explorer, seleccionamos la opción **Build** para compilar el proyecto. Este paso es importante, porque si no se compila, Visual Studio no es capaz de reconocer los formularios para utilizarlos como base de otros. Si efectuamos algún cambio, tendremos que compilar otra vez.

> Un formulario no es más que una instancia de una clase System.Windows.Forms.Form.

- Para utilizar el formulario base, agregamos otro proyecto a la solución, pero del tipo Windows Application, y le ponemos como nombre HerenciaVisual. Lo marcamos como Proyecto de Inicio de la solución

FIGURA 036 | En nuestro formulario base colocamos los botones que queremos tener en todos los formularios comunes.

Herencia y formularios

Si bien es bastante sencillo realizar herencia visual, es fundamental tener un buen conocimiento de las técnicas de orientación a objetos para poder efectuar el correcto manejo de las clases, la sobrescritura de métodos y el acceso a diferentes elementos de las clases base. Así podremos aprovechar y explotar al máximo esta excelente característica de .NET.

Visual Studio no es capaz de reconocer los formularios para utilizarlos como base de otros hasta que no los compilamos.

eligiendo la opción Set as StartUp Project, en el menú contextual.

Rescritura de métodos

Un buen diseño de herencia visual prevé la rescritura de los métodos que se ejecutan ante determinados eventos. Para esto, debemos tener en cuenta no escribir directamente las acciones en el manejador del evento en el formulario base, sino invocar un método y hacer que sea Overrideable en VB.NET, o virtual en C#. Así, podremos rescribirlo en los formularios derivados.

- Al igual que en el proyecto anterior, eliminamos el formulario creado por defecto y seleccionamos Agregar/Nuevo elemento, en el menú contextual del proyecto.
- En la ventana que aparece seleccionamos Inherited Form, para crear un formulario con herencia visual; le ponemos como nombre frmHerencia.vb.
- Al aceptar, se abre una ventana para elegir de qué formulario queremos heredar. De forma automática, nos muestra el formulario creado en el otro proyecto, ya que está en la misma solución (Figura 37). Si no deseamos alguno de ellos, presionamos el botón Examinar… y elegimos algún assembly que contenga los forms que queremos. En nuestro caso, presionamos Aceptar directamente, porque ése es el que queremos.
- Eso es todo. Así creamos nuestro formulario heredado junto con los controles de la base. Si lo ejecutamos, al presionar los botones, se muestran los mensajes incorporados en el código del formulario base.

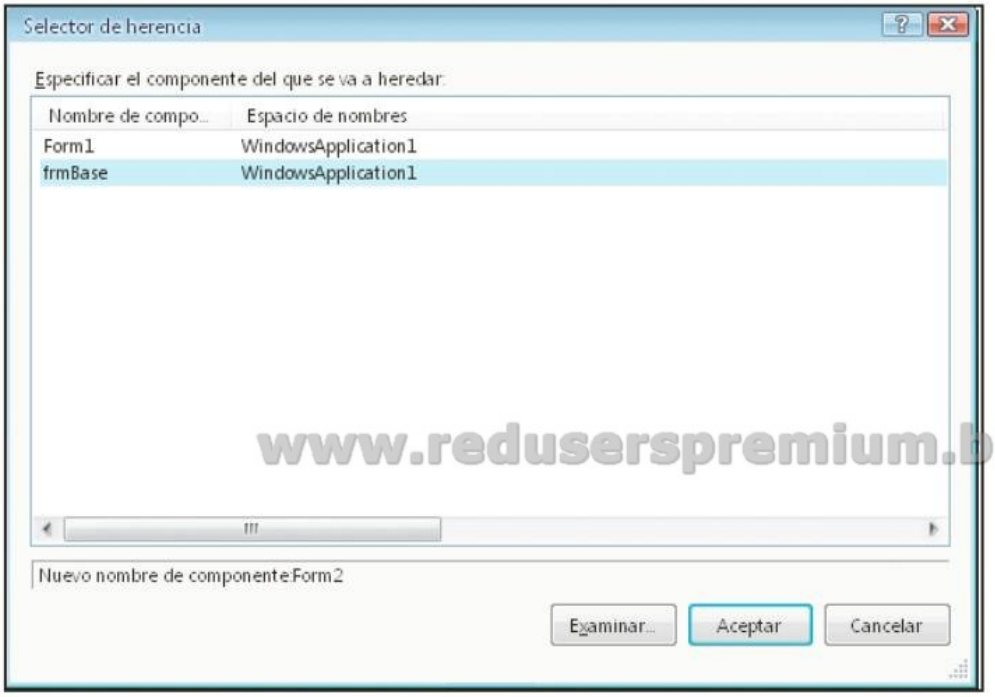


FIGURA 037 | Al crear un nuevo formulario heredado, Visual Studio nos muestra la lista de los disponibles para que seleccionemos el de base.

Configuración

Muchas veces sucede que cada usuario tiene preferencias distintas en cuanto a aspectos de la interfaz, como colores de fondo o textos de los controles. Obviamente, gestionar las personalizaciones por código puede ser una tarea muy trabajosa. Pero si lográramos brindar estas opciones, generaríamos una aplicación muy completa. Por suerte, una de las nuevas características que proporciona la plataforma .NET 2.0 es la posibilidad de almacenar las preferencias del usuario y los elementos de configuración de manera dinámica para utilizar en la aplicación. Para realizar esto, proporciona herramientas que nos ahorran casi el 100% del código. Por medio de estas funcionalidades, es posible personalizar las aplicaciones Windows con propiedades tales como color de fondo, color de texto, nombre de la compañía, nombre del producto, cadenas de conexión a bases de datos, nombre del servidor, y más. A continuación, veremos en detalle cómo aprovechar esta funcionalidad en nuestra aplicación.

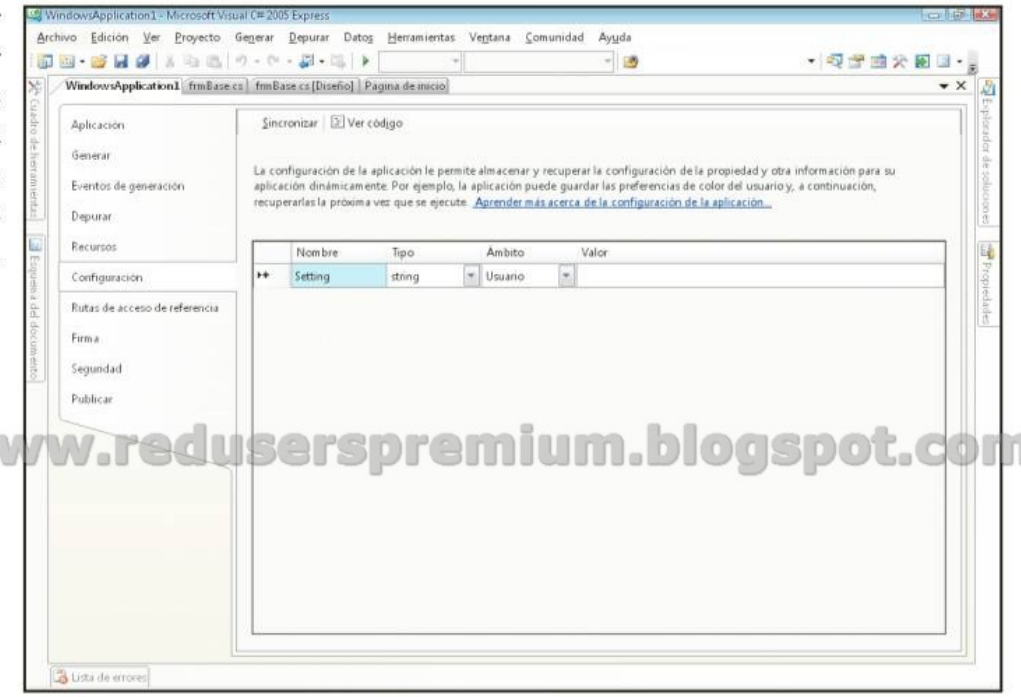
> Tengamos en cuenta que cada usuario tiene preferencias distintas en cuanto a aspectos de la interfaz.

Estas propiedades de configuración se almacenan en un archivo específico llamado app.settings. Cuando éste es compilado, cambia su nombre por el del ejecutable más la

ⓘ Leer y almacenar

Si bien la posibilidad de trabajar con configuraciones e, incluso, de asignarlas a controles ya existe desde la primera versión de .NET, la 2.0 incorporó la posibilidad de modificar y almacenar los valores en tiempo de ejecución. Antes, desde el código fuente sólo podíamos leer los valores configurados, y si queríamos hacer algún cambio, debíamos hacerlo desde fuera de la aplicación, editando el archivo de configuración.

FIGURA 038 | Para acceder a la configuración y editar los valores, debemos seleccionar la solapa Configuración (Settings, en inglés) dentro de las propiedades del proyecto.



Las propiedades de configuración se almacenan en un archivo de configuración llamado "app.settings".

extensión .config. Por ejemplo, si nuestro ejecutable se llama demo.exe, el archivo de configuración será demo.exe.config. Para acceder a la configuración desde Visual Studio, debemos abrir las propiedades del proyecto (hacemos clic derecho en el Explorador de soluciones y elegimos Propiedades) y, luego, elegir la solapa Configuración. Allí podremos crear las propiedades que serán dinámicas y darles el alcance que corresponda: a nivel de aplicación o a nivel de usuario. Si optamos por nivel de aplicación, cualquier cambio afectará a todos los usuarios que la utilicen, en tanto que si elegimos nivel de usuario, el cambio sólo afectará al usuario en particular que esté utilizando la aplicación.

El entorno de desarrollo de Visual Studio Express crea automáticamente una clase **Settings**, que se usa para leer y grabar esos valores en tiempo de ejecución con pocas o ninguna línea de código y de manera transparente. Esta clase posee una propiedad por cada atributo que hemos configurado. Para ver cómo funcionan, realizaremos el siguiente ejemplo:

FIGURA 039 | Vemos la interfaz que usaremos para probar el funcionamiento de la configuración y las propiedades dinámicas de los controles de nuestra aplicación.

- Creamos un proyecto Windows Application, al que llamamos PruebaConfiguración.
- Abrimos las propiedades del proyecto y seleccionamos la solapa Configuración.
- Se presentará una grilla en la que vamos a crear las propiedades dinámicas. Cada una tiene nombre, tipo de dato, alcance (aplicación o usuario) y valor por defecto. Creamos las propiedades tal como se muestran en la Tabla 13.
- Una vez creadas las propiedades, cerramos la ventana y agregamos al formulario tres controles Button y un Label. Esto nos permitirá demostrar la funcionalidad

Tabla 13 | Propiedades de configuración para nuestro ejemplo

Propiedad	Descripción	Valor por defecto
Titulo	Contendrá el título de la aplicación, será del tipo String y alcance a nivel de aplicación.	Configuración 1.0
ColorFondo	Contendrá el color de fondo del formulario, será de tipo Color y estará a nivel de usuario.	Control
ColorTexto	Contendrá el color de texto del formulario y también será de tipo Color. El alcance lo definimos a nivel usuario.	ControlText

EL ENTORNO VISUAL STUDIO EXPRESS CREA AUTOMÁTICAMENTE UNA CLASE SETTINGS, QUE SE USA PARA LEER Y GRABAR LOS VALORES DE CONFIGURACIÓN EN TIEMPO DE EJECUCIÓN CON POCAS O NINGUNA LÍNEA DE CÓDIGO Y DE MANERA TRANSPARENTE.

de la aplicación. El Label contendrá el título de la aplicación y se cargará dinámicamente por enlace o "binding". Los botones servirán para cambiar el color de fondo del formulario, el del texto y volver a los valores predefinidos, en caso de ser necesario. Los nombres para cada control son los siguientes: lblTitulo, btnColorFondo, btnColorTexto y btnReset. La interfaz debe quedar similar a como se muestra en la Figura 39.

- Ahora comenzamos con el enlace. Seleccionamos Label, en la ventana de propiedades, elegimos Application Settings y, luego, Properties Binding. En la ventana que se abre, seleccionamos la propiedad Text del control, que es la que deseamos asociar; en el valor escogemos la propiedad Titulo, creada en el editor anteriormente. Al confirmar la opción, nuestro Label muestra el valor por defecto asociado.
- El resto de las propiedades las utilizaremos por código. Creamos dos procedimientos: uno para seleccionar el color que el usuario desee utilizar, y otro para actualizar las propiedades al modificar sus preferencias. En los eventos de cada Button llamamos a las rutinas creadas, y en los eventos Load y FormClosing del formulario leemos y grabamos, respectivamente, las opciones seleccionadas por el usuario. El código que vamos a crear en el formulario es el siguiente:

```
Private Sub btnReset_Click(ByVal sender
As System.Object, ByVal e As
System.EventArgs) Handles btnReset.Click
    'El usuario solicita volver a
    'lo valores por defecto.
    'Recargamos los valores iniciales
    nuevamente
    My.Settings.Reset()
    My.Settings.Reload()
    CargarConfiguracion()
End Sub

Private Sub btnColorFondo_Click(ByVal
sender As System.Object, ByVal e As
System.EventArgs) Handles
btnColorFondo.Click
    'Cambiamos el color de fondo
    Dim oColor As Color = Me.BackColor
    Me.BackColor = SeleccionarColor
    (oColor)
End Sub

Private Sub btnColorTexto_Click(ByVal
sender As Object, ByVal e As
System.EventArgs) Handles
btnColorTexto.Click
    'Cambiamos el color de texto
    Dim oColor As Color = Me.ForeColor
    Me.ForeColor = SeleccionarColor
    (oColor)
End Sub

Private Sub Form1_FormClosing(ByVal
```

```
    sender As Object, ByVal e As
    System.Windows.Forms.FormClosingEventArgs)
    Handles Me.FormClosing
        'Al cerrar actualizo el archivo de
        configuración con las
        'preferencias del usuario
        My.Settings.ColorFondo =
        Me.BackColor
        My.Settings.ColorTexto =
        Me.ForeColor
        My.Settings.Save()
    End Sub

    Private Sub Form1_Load(ByVal sender As
    Object, ByVal e As System.EventArgs)
    Handles Me.Load
        'Carga los valores provistos por el
        archivo de configuración
        CargarConfiguracion()
    End Sub
    'Asigna los valores desde el archivo de
    configuracion
    Private Sub CargarConfiguracion()
        Me.BackColor =
        My.Settings.ColorFondo
        Me.ForeColor =
        My.Settings.ColorTexto
    End Sub

    'Permite seleccionar un color
    Private Function SeleccionarColor(ByVal
    Color As Color) As Color
        Dim oColor As Color = Color
        Dim oColorDialog As New ColorDialog
        oColorDialog.Color = oColor
        oColorDialog.FullOpen = True
        If oColorDialog.ShowDialog =
        Windows.Forms.DialogResult.OK Then
            oColor = oColorDialog.Color
        End If
        oColorDialog = Nothing
        Return oColor
    End Function
```

Una vez escrito el código, podemos probarlo. Al seleccionar algún color, éste cambia; y al cerrar y volver a abrir la aplicación, la selección del usuario permanece.

Diálogos comunes

En la mayoría de las aplicaciones, debemos permitir que el usuario abra o grabe un archivo, elija una carpeta para hacer alguna tarea o, incluso, seleccione un color para asignarlo a un elemento de la interfaz. Si hacemos ventanas propias para estos diálogos, nos estaremos saliendo un poco de los estándares de Windows, con la consecuencia de que al usuario le tomará más tiempo familiarizarse con la aplicación. Lo ideal es usar los mismos cuadros de diálogo que emplean todos los programas. Para esto, .NET nos provee de unos controles muy útiles: los

EN LA MAYORÍA DE LAS APLICACIONES, DEBEMOS PERMITIR QUE EL USUARIO ABRA O GRABE UN ARCHIVO, ELIJA UNA CARPETA PARA HACER ALGUNA TAREA O, INCLUSO, SELECCIONE UN COLOR PARA ASIGNARLO A UN ELEMENTO DE LA INTERFAZ.

diálogos comunes, o *Common Dialogs*. Estos controles permiten interactuar con el usuario para la ejecución de tareas comunes, como abrir un archivo, configurar la impresión, seleccionar un color del sistema, etc. Para lograrlo, basta con configurar algunas propiedades e invocar su método ShowDialog(). Estos controles pueden arrastrarse directamente desde el Cuadro de Herramientas o bien crearse a mano por código en la medida en que se necesiten.
Los principales diálogos con los que contamos para hacer esto son:

- ColorDialog: Muestra un diálogo que permite la selección de colores por parte del usuario. El valor elegido se obtiene de la propiedad **Color**.
- FontDialog: Muestra un diálogo para la selección de fuentes. El valor seleccionado se obtiene de la propiedad **Font**.
- FolderBrowserDialog: Permite la selección de una carpeta. El nombre de la carpeta seleccionada se obtiene de su propiedad **SelectedPath**.
- OpenFileDialog: Permite la selección de un archivo para abrirlo. El nombre se obtiene de su propiedad **Filename**.
- SaveFileDialog: Similar al diálogo anterior, pero permite la selección de un nombre para grabar un archivo. La propiedad con el nombre del archivo es **Filename**.
- PrintDialog: Presenta las opciones para la impresión de documentos.

Como acabamos de ver, cada control tiene un propósito bien definido y provee una propiedad para acceder al valor seleccionado por el usuario. En todos los casos, podemos utilizar el resultado del método ShowDialog (que es de tipo DialogResult) para saber si el usuario aceptó o canceló el diálogo.

Los Common Controls permiten la interacción con el usuario para la ejecución de tareas comunes.

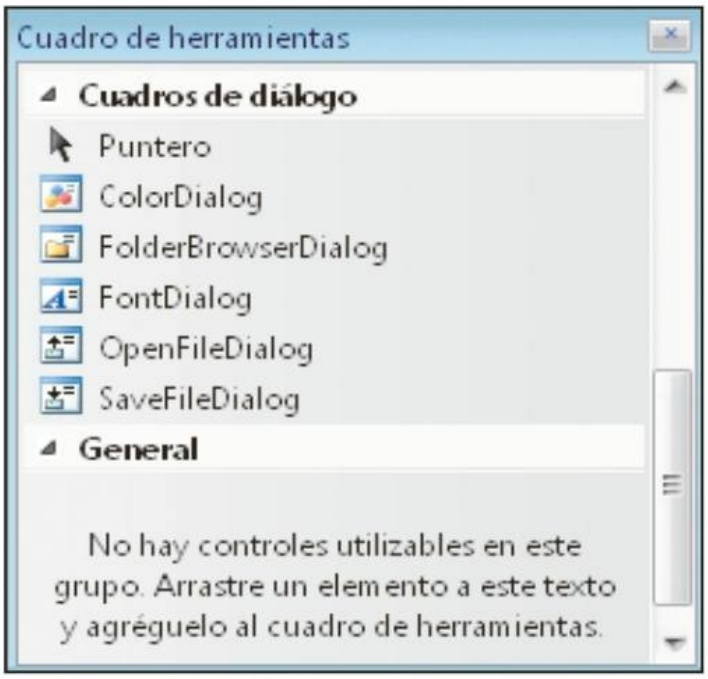


FIGURA 040 | Los diálogos comunes se encuentran en la sección Cuadros de Diálogo del Cuadro de Herramientas de Visual Studio Express.

ⓘ Exploremos las propiedades

Los cuadros de diálogo de .NET poseen una gran cantidad de propiedades. Conocerlas todas excede el alcance de esta obra, pero podemos hacer el ejercicio de recorrerlas en la ventana de propiedades y leer el breve texto de documentación que se muestra al pie. Veremos que hay propiedades para alterar completamente el comportamiento de los controles y hacer exactamente lo que necesitemos, con el fin de obtener una gran flexibilidad.

Enlace a datos

Veremos diferentes formas de vincular y asociar nuestra aplicación con una base de datos.

Muchas veces, tendremos que mostrar en pantalla datos de una tabla de la base o el valor de alguna propiedad de un objeto. Ante esta situación, tenemos dos alternativas: escribir el código necesario para asignar a las propiedades de los controles los valores por mostrar, leyéndolos de donde corresponda; o apelar a las técnicas de DataBinding, o enlace de datos. Esta técnica permite asociar un origen de datos o **DataSource** a los controles que mostrarán la información contenida en ellos a los usuarios. Así como en el capítulo de ASP.NET vimos que podemos enlazar un control DropDownList con los registros de una tabla, también podemos hacerlo en las aplicaciones Windows. Los orígenes de datos para realizar DataBinding pueden ser de diferente naturaleza:

- Objetos ADO.NET: DataReader, DataTable, DataSet, DataView, etc.
- Documentos XML: Creados en memoria, obtenidos desde el sistema de archivos o provenientes de un WebService.
- Colecciones: Cualquier colección puede ser un origen de datos (tal como ArrayList, CollectionBase, etc.) o bien cualquier clase que implemente la interfaz IList.

FIGURA 041 | DataBinding en funcionamiento. Al enlazar un ArrayList con un ComboBox, los elementos del primero se muestran como ítem del combo.

DataBinding con colecciones

Las colecciones como orígenes de datos son sencillas de utilizar. Para entender mejor cómo funcionan, veamos un ejemplo: crearemos una colección de colores y la enlazaremos a un control ComboBox en el evento Load del formulario. El resultado de la ejecución se muestra en la Figura 41.

```
' VB.Net
Private Sub Form1_Load(ByVal sender As
Object, ByVal e As System.EventArgs) Handles
Me.Load

        'Creamos coleccion
        Dim colColores As ArrayList
        colColores = New ArrayList

        'creamos contenido de la coleccion
        colColores.Add("Rojo")
        colColores.Add("Verde")
        colColores.Add("Azul")
        colColores.Add("Amarillo")

        'asignamos la colección como origen
        de datos al combobox
        cmbColores.DataSource = colColores

End Sub
```

```
// C#
private void Form1_Load(object sender,
EventArgs e)
{
      //Creamos coleccion
     System.Collections.ArrayList colColores;
     colColores = new System.Collections.
     ArrayList();

     //creamos contenido de la coleccion
     colColores.Add("Rojo");
     colColores.Add("Verde");
     colColores.Add("Azul");
     colColores.Add("Amarillo");

    //asignamos la colección como origen de
    datos al combobox
    cmbColores.DataSource = colColores;
}
```

En el ejemplo anterior enlazamos sólo un elemento, pero, en general, enlazaremos múltiples campos en la visualización, como código y descripción. Un ComboBox sólo permite visualizar un campo del origen de datos a la vez, en tanto que un DataGridView resuelve esta situación creando columnas por cada uno de ellos. Para el caso del ComboBox y de otros controles que no permitan múltiples campos, cada uno proporciona dos propiedades adicionales: **DisplayMember** y **MemberValue**. En la primera asignamos el campo del DataSource que queremos que el usuario vea, mientras que en la segunda asignamos el valor que necesitamos en nuestro sistema (en general, será un valor que nos permita identificar de manera unívoca el registro seleccionado). En el caso de tablas, el campo es el nombre de una columna; en el caso de enlazar objetos, puede ser el nombre de una propiedad del objeto o el de una función que retorne algún valor.

En el siguiente ejemplo, vamos a crear una clase Color con dos propiedades (código y descripción), generaremos una colección de estos objetos y la enlazaremos al ComboBox. En este caso, para obtener el valor seleccionado debemos utilizar la propiedad **SelectedValue** del combo. Para ahorrar espacio, sólo veremos el código en C#.

> El DataBinding permite asociar un origen de datos o DataSource a los controles.

Primero creamos la clase Color con las propiedades que hemos mencionado:

```
class Color
{

    /*constructor*/
    public Color(int codigo,string
    descripcion)
    {
        _codigo = codigo;
```

El método DataBind

Cuando aprendimos a enlazar controles de ASP.NET con datos para aplicaciones Web, vimos que es fundamental llamar al método DataBind() para que el enlace se haga efectivo. En las aplicaciones Windows esto no es necesario y, de hecho, los controles no poseen ese método. La diferencia radica, fundamentalmente, en el modelo de ejecución y el ciclo de vida de las aplicaciones Web, que es bastante más complejo que el de las aplicaciones Windows, donde todo ocurre en la misma computadora, hecho que simplifica bastante las cosas cuando trabajamos con las técnicas de DataBinding.

En general, enlazaremos múltiples campos en la visualización.

```
            _descripcion = descripcion;
        }

        /*propiedad y campo de codigo*/
        private int _codigo;
        public int Codigo
        {
            get { return _codigo; }
            set { _codigo = value; }
        }

        /*propiedad y campo de descripcion*/
        private string _descripcion;
        public string Descripcion
        {
            get { return _descripcion; }
            set { _descripcion = value; }
        }

    }
```

Luego, en el formulario, escribimos el código necesario para enlazar una lista de objetos de tipo Color con el ComboBox:

```
        private void Form1_Load(object
        sender, EventArgs e)
        {
            //Creamos colección
            System.Collections.ArrayList
            colColores;
            colColores = new System.
            Collections.ArrayList();
            //creamos contenido de la colección
            colColores.Add(new Color(1,
            "Rojo"));
            colColores.Add(new Color(2,
            "Verde"));
            colColores.Add(new Color(3,
            "Azul"));
            colColores.Add(new Color(4,
            "Amarillo"));

            //asignamos la colección como
            origen de datos al combobox
            cmbColores.DataSource =
            colColores;
            cmbColores.DisplayMember =
            "Descripcion";
```

GridView

Como ya mencionamos, el entorno .NET provee un control muy útil llamado GridView, que permite mostrar información a modo de tabla, colocando una columna por cada campo o propiedad y una fila por cada registro o elemento de la colección que le enlacemos. Además de mostrar la información, permite al usuario modificar los datos, y brinda propiedades y métodos para obtener los cambios realizados y volcarlos al origen de datos.

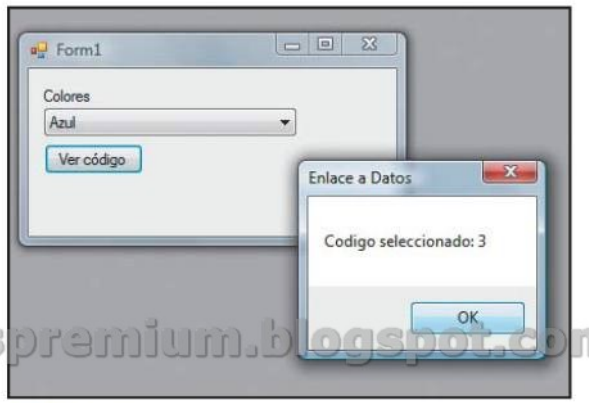


FIGURA 042 | Resultado de la selección al enlazar datos a un ComboBox. El valor obtenido es el que brinda la propiedad SelectedValue, que obtiene su valor de la propiedad ValueMember.

```
            cmbColores.ValueMember =
            "Codigo";
        }

        private void btnCodigo_Click(object
        sender, EventArgs e)

        {
          MessageBox.Show("Código
          seleccionado: " + cmbColores.
          SelectedValue, "Enlace a Datos");
        }
```

El objeto BindingSource

Durante el DataBinding, a veces es necesario intercambiar diversos DataSources, y si enlazamos cada uno directamente a los controles, dependiendo de ellos, deberemos alterar el código de actualizaciones de los datos. Para centralizar estos cambios y abstraernos de la fuente, podemos utilizar el objeto **BindingSource**. Éste proporciona una capa de abstracción entre el origen de datos y los controles, y nos da flexibilidad a la hora de controlar el enlace de los datos y su manipulación. La manera de realizar el enlace es sencilla: basta con asignar la propiedad DataSource con el origen de datos y enlazar el BindingSource a los controles.

Veamos la modificación que debemos implementar en el código del ejemplo anterior para utilizar un BindingSource:

```
BindingSource BindingSourceCtl = new
BindingSource();
BindingSourceCtl.DataSource = colColores;
cmbColores.DataSource = BindingSourceCtl;
```

Como podemos notar, se realiza un paso intermedio entre la asignación del DataSource al control, utilizando como intermediario el objeto BindingSource. El objeto BindingSource tiene varios métodos, propiedades y eventos útiles que nos sirven para mantener el control

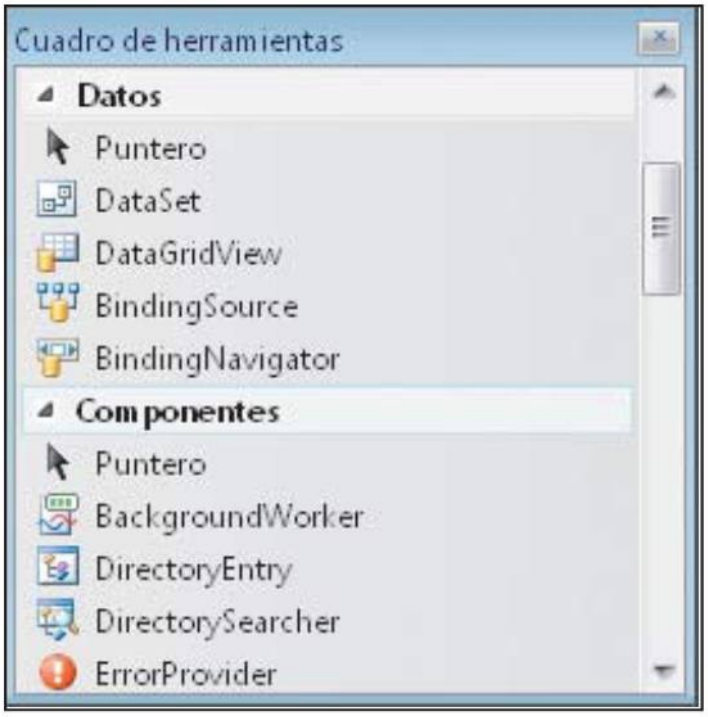


FIGURA 043 | Los controles para trabajar con DataBinding se encuentran en la sección Datos del Cuadro de herramientas de Visual Studio.

Ventajas del control BindingSource

En las versiones anteriores de .NET (la 1.0 y la 1.1) no existía el control BindingSource. Si bien era posible enlazar controles a datos, la ausencia de este control implicaba la necesidad de escribir código para hacer muchas de las tareas relacionadas con el manejo de las asociaciones entre los datos y los controles, como hacer los refrescos necesarios cuando el usuario navegaba por los distintos registros del origen de datos.

del enlace a datos. En la Tabla 14 podemos ver algunos de ellos.

Con estos métodos, propiedades y eventos, podemos movernos a través del DataSource de manera independiente del control al que esté enlazado y obtener efectivamente el objeto activo que está seleccionado en el control, sea cual fuera. Podemos incluir un sistema de navegación de registros de forma manual e, incluso, permitir el filtrado de datos. También tenemos bajo control la forma en que se editan los datos y hasta podemos cancelar su edición.

Uso de DataBinding con otros controles

Otra forma de realizar DataBinding es recurrir a la propiedad DataBindings de los controles y asignarles las propiedades por enlazar. Esta propiedad es una colección de objetos Binding. La clase Binding permite enlazar una propiedad de un control con un campo o propiedad del origen de datos. Siguiendo con el ejemplo anterior, podríamos visualizar las propiedades Código y Descripción de cada objeto Color en cajas de texto. Para navegar entre ellos, utilizamos un control llamado Binding-Navigator, que nos proporciona toda la funcionalidad para realizar la navegación entre los elementos de un DataSource. En la Tabla 15 vemos los controles que utilizamos en este ejemplo.
Una vez que colocamos en el formulario los controles listados en la tabla, el diseño nos quedará como muestra la Figura 44.

Ahora nos resta poner el código en el evento Load del formulario, para hacer el enlace de datos con los controles. ¿Cómo lo logramos? Es muy sencillo:

Tabla 14 | Principales elementos del objeto BindingSource

Tipo	Nombre	Descripción
Método	MoveFirst	Se posiciona en el primer registro del DataSource.
Método	MoveLast	Se posiciona en el último registro del DataSource.
Método	MovePrevious	Se posiciona en el registro anterior al actual.
Método	MoveNext	Se posiciona en el registro siguiente al actual.
Método	CancelEdit	Cancela la edición en curso.
Propiedad	Position	Permite obtener o asignar la posición del registro actual sobre la base del índice de la colección.
Propiedad	List	Retorna la lista asociada al DataSource asignado.
Propiedad	Filter	Permite filtrar los registros según una condición.
Propiedad	Current	Retorna el objeto activo del DataSource.
Propiedad	AllowEdit	Determina si el DataSource permitirá editar los datos.
Propiedad	AllowAddNew	Determina si el DataSource permitirá agregar nuevos datos.
Propiedad	AllowRemove	Determina si el DataSource permitirá la eliminación de datos.
Evento	AddingNew	Ocurre antes de que un ítem sea agregado a la lista interna.
Evento	CurrentChanged	Ocurre cuando el ítem actual ha cambiado.
Evento	CurrentItemChanged	Ocurre cuando una propiedad del ítem actual ha cambiado.
Evento	PositionChanged	Ocurre después de que el valor de la propiedad position ha cambiado.

VB.NET:

```
Private Sub Form1_Load(ByVal sender As
Object, ByVal e As System.EventArgs) Handles
Me.Load
        'Creamos coleccion
        Dim colColores As ArrayList
        colColores = New ArrayList

        'Creamos contenido de la coleccion
        colColores.Add(New Color(1, "Rojo"))
        colColores.Add(New Color(2, "Verde"))
        colColores.Add(New Color(3, "Azul"))
        colColores.Add(New Color(4,
        "Amarillo"))

        'Asignamos la colección como origen
        de datos al BindingSource
        BindingSourceCtl.DataSource =
        colColores

        'Enlazamos el BindingNavigator con
        el BindingSource para permitir
        'la navegación de los datos de
        manera automática
        BindingNavigatorCtl.BindingSource =
        BindingSourceCtl

        'Enlazamos cada control de texto con
        lo la propiedad a visualizar
        txtCodigo.DataBindings.Add("Text",
        BindingSourceCtl, "Codigo")
        txtDescripcion.DataBindings.Add
        ("Text", BindingSourceCtl,
        "Descripcion")

End Sub
```

C#:

```
private void Form1_Load(object sender,
EventArgs e)
{
    //Creamos coleccion
    System.Collections.ArrayList colColores;
    colColores = new System.Collections.
    ArrayList();

    //creamos contenido de la coleccion
    colColores.Add(new Color(1, "Rojo"));
    colColores.Add(new Color(2, "Verde"));
    colColores.Add(new Color(3, "Azul"));
    colColores.Add(new Color(4, "Amarillo"));

    //Asignamos la colección como origen de
    datos al BindingSource
    BindingSourceCtl.DataSource = colColores;

    //Enlazamos el BindingNavigator con el
    BindingSource para permitir
    //la navegación de los datos de manera
    automática
    BindingNavigatorCtl.BindingSource =
    BindingSourceCtl;

    //Enlazamos cada control de texto con lo
    que deasemos visualizar en cada uno
    txtCodigo.DataBindings.Add("Text",
    BindingSourceCtl, "Codigo");
    txtDescripcion.DataBindings.Add("Text",
    BindingSourceCtl, "Descripcion");
}
```

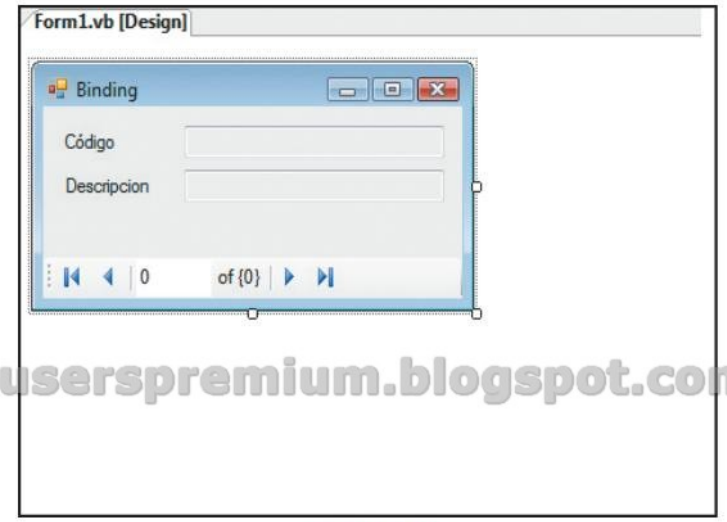


FIGURA 044 | Interfaz que diseñaremos para el ejemplo de enlace a datos.

ES POSIBLE REALIZAR DATABINDING DIRECTAMENTE SIN UTILIZAR UN BINDINGSOURCE.

FIGURA 045 | DataBinding permite una visualización rápida de los datos de diferentes DataSources. BindingNavigator nos permite recorrerlos de manera automática.

§ BindingSource

Durante el enlace de datos, es fundamental utilizar BindingSource para tener control de la colección o el origen de datos utilizado en el DataBinding. Este objeto nos proporciona una abstracción frente a la visualización de los datos con diferentes controles, y si en algún momento deseamos cambiarlos, prácticamente no tendremos que modificar el código de la aplicación.

Primero creamos la colección de colores como lo habíamos hecho anteriormente. Luego, en vez de asignar directamente el origen de datos a los controles, utilizamos el BindingSource para esa tarea. Esto nos provee de un intermediario entre las diferentes operaciones que realizaremos con el origen de datos, y mantendremos bajo control la colección, por si la necesitamos en algún momento. Para permitir la navegación de los ítem de la colección, recurrimos al control BindingNavigator y lo enlazamos con el BindingSource para sincronizarlos (.NET se encarga del resto). Para terminar, enlazamos las cajas de texto con lo que queremos visualizar en cada una. Para enlazar los controles, usamos la propiedad DataBindings de cada uno. De esta manera, le indicamos cómo y qué enlazar. El primer parámetro especifica qué propiedad del control enlazar; el segundo, el origen de datos (en nuestro caso, BindingSource); y por último, el campo por visualizar: si la colección es de objetos,

Tabla 15 | Elementos que debemos colocar en el formulario

Control	Cantidad	Descripción
Labels	2	Indican etiquetas de código y descripción.
TextBox	2	Enlazan las propiedades Código y Descripción del objeto Color.
BindingSource	1	Permite asignar el origen de datos. Puede crearse por código o utilizarse como control. En este último caso, se crea una instancia de este objeto de manera automática. Es la forma que utilizaremos en este ejemplo.
BindingNavigator	1	Permite la navegación del DataSource de manera automática. Pondremos su propiedad Dock en Bottom.

LA PROPIEDAD DATABINDINGS DE LOS CONTROLES PERMITE ENLAZARLOS A DATOS.

aquí va el nombre de una propiedad del objeto de la colección, mientras que si es una tabla, va el nombre de la columna por enlazar. Eso es todo. Al ejecutar el programa, veremos que los datos se muestran automáticamente y podemos recorrerlos con el BindingNavigator ubicado en la parte inferior del formulario.

DataBinding directo

También es posible realizar DataBinding directamente, sin utilizar un BindingSource con los datos obtenidos de la base, asignándolos a la propiedad DataSource del control. Si bien esto es posible, el hecho de hacerlo de esta manera implica que el código sea complejo de modificar y mantener, por lo que las buenas prácticas de desarrollo evitan utilizar esta técnica tal como se la expone. Sin embargo, puede usarse en algunos casos puntuales para realizar testing, demos, pruebas, etc. Por ejemplo:

VB.NET:

```
Dim oCnn As New SqlConnection("....")
Try
  oCnn.Open
  Dim da As New SqlDataAdapter("Select * from
  Customers", oCnn)
  Dim dt As New DataTable
  da.Fill(dt)
  Me.dataGridView1.DataSource = dt
Finally
  cn.Close
End Try
```

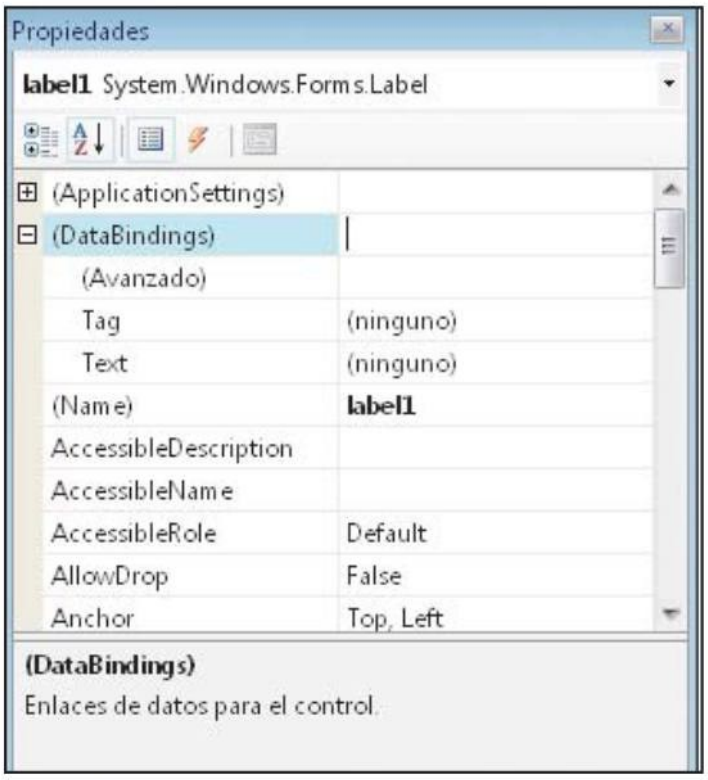


FIGURA 046 | Visual Studio nos permite asociar visualmente las propiedades de los controles con los campos del origen de datos.

DataBinding y DataSets

Si creamos un origen de datos y lo agregamos al proyecto (mediante la opción Datos/Agregar nuevo origen de datos), tenemos la posibilidad de enlazar las tablas con los controles BindingSource y, luego, asociar las propiedades de los controles con los campos, todo de manera visual y sin escribir una sola línea de código. Esto acelerará bastante el desarrollo en aplicaciones simples y sin mucha lógica de negocios.

Distribución de aplicaciones

Cuáles son las diferentes alternativas que tenemos a nuestra disposición para distribuir una aplicación una vez terminada.

Una vez que nuestra aplicación está desarrollada y probada, debemos distribuirla para que los usuarios puedan utilizarla. Con .NET disponemos de varias alternativas para hacerlo, y deberemos elegir la más adecuada según los requerimientos de cada caso. Básicamente, .NET proporciona tres formas para distribuir las aplicaciones: **XCopy, Windows Installer** y **ClickOnce**.

XCopy

La distribución por XCopy se relaciona con un escenario donde todo lo que hay que hacer es copiar el directorio (incluyendo sus subdirectorios) a la computadora en la que se desea ejecutar el programa. En muchos artículos y bibliografía de .NET, esta técnica se menciona como XCopy Deployment. Este tipo de distribución fue uno de los detalles más aplaudidos con la llegada de .NET. Anteriormente, las aplicaciones desarrolladas con Visual Basic estaban basadas en el modelo COM o ActiveX, y la distribución implicaba la registración de componentes, sin mencionar que esto podía traer aparejados problemas de versionamiento, sobrescritura de archivos, etc. (el famoso "infierno de las dlls").

El nombre de esta práctica deriva del comando DOS XCopy, que se utiliza para copiar una estructura de directorios de un lugar a otro. Este tipo de distribución asume que todas las dependencias de nuestra aplicación pueden encontrarse en la estructura de directorios de la aplicación. El CLR buscará todas sus dependencias allí e intentará ejecutarla. Como vimos al comienzo de la obra, si alguna dependencia no está en la carpeta de la aplicación, el CLR buscará en la GAC.

Obviamente, la ventaja de este método de distribución radica en su sencillez, ya que basta con ejecutar algunas instrucciones desde la línea de comandos, para tener la aplicación lista para ejecutar en la PC del usuario. Incluso, si debemos hacerlo en muchas computadoras, podemos automatizar el proceso creando un archivo BAT que se encargue de generar las carpetas y copiar los archivos necesarios.

FIGURA 047 | El comando XCopy nos permite distribuir una aplicación simplemente copiando los archivos necesarios para su ejecución a la PC del usuario.

Si la aplicación utiliza elementos más avanzados –como Registry, permisos especiales, GAC, etc.–, debemos dejar todo perfectamente configurado e inicializado durante el proceso de

instalación o deployment de la aplicación. Para lograrlo, este tipo de distribución es bastante limitado, y se deben realizar las tareas de forma manual o emplear un esquema de instaladores basado en Windows Installer.

Windows Installer

Microsoft introdujo el servicio de Windows Installer como parte de Windows 2000, como una manera de simplificar la instalación de aplicaciones. Windows Installer también se instala de manera automática al hacer lo propio con las aplicaciones de Microsoft Office, por ejemplo. El servicio de Windows Installer es lo que Microsoft llama un componente del sistema operativo. Implementa todos los requerimientos que el instalador necesita, como no sobrescribir archivos de sistema con versiones más antiguas. En vez de crear un ejecutable que contenga todas las reglas de instalación, se crea un archivo llamado Windows Installer Package, que describe lo que debe hacerse durante la instalación. Estos archivos tienen una extensión .msi, que deriva de Microsoft Installer.
Una aplicación de Windows Installer Package consta de tres partes: components, features y products. Cada una está constituida por un conjunto de los elementos anteriores. Por ejemplo, un product está compuesto de varias features, y cada feature puede tener uno o varios components. Un component es la parte más pequeña de la instalación, y contiene un grupo de archivos y otros recursos que necesitan instalarse juntos.

Al ser tratado de esta manera, el archivo **.msi** hace que se componga de diferentes paquetes **.cab**, y permitirá al usuario determinar qué elementos instalar y cuáles no.
Cuando el usuario desee instalar una aplicación, deberá ejecutar este archivo, asumiendo que el servicio de Windows Installer ya está instalado.

> Windows Installer es un componente que implementa los requerimientos que el instalador necesita.

Si no lo está, por lo general, hay junto a él un archivo Setup.exe, que instalará el servicio en caso de que no se encuentre o lo actualizará si es una versión más nueva. Al comenzar la

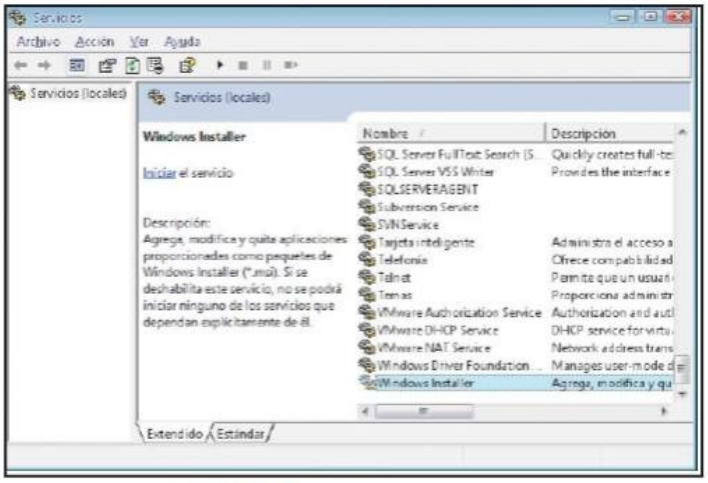


FIGURA 048 | Cuando utilizamos Windows Installer, lo que realmente distribuimos es un archivo MSI, y el servicio de Windows Installer se encarga de hacer la instalación.

Windows Installer y los permisos

Como la instalación mediante Windows Installer es llevada a cabo por un servicio que se encarga de realizar las acciones del archivo MSI, debemos tener en cuenta que el usuario que instale la aplicación deberá tener permisos para hacerlo. En la mayoría de las instalaciones de Windows, sólo el administrador del equipo local y los administradores de dominio tienen los privilegios suficientes para correr archivos MSI.

Microsoft introdujo el servicio de Windows Installer como parte de Windows 2000.

instalación, el servicio leerá el archivo y determinará qué cosas debe realizar, tales como copiar un archivo del sistema, modificar el Registry, crear directorios y accesos directos, etc.

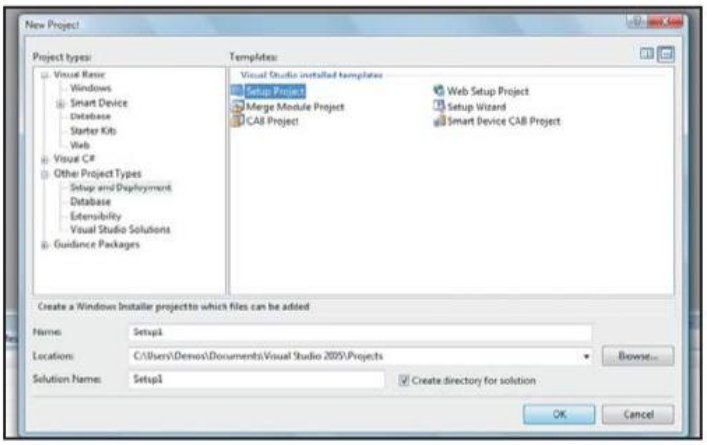


FIGURA 049 | Desde aquí podemos elegir el tipo de proyecto de instalación que se va a utilizar, dependiendo de la aplicación. Todos hacen uso de la tecnología Windows Installer.

⊛ Problemas durante la instalación

Al problema de los permisos que mencionamos en la página anterior, se suma otro relacionado con los servicios. Es muy común que algunos usuarios avanzados (o no tanto) detengan algunos servicios que consideran en desuso para liberar recursos del sistema operativo y optimizar el equipo. En caso de que el servicio de Windows Installer esté detenido, no se podrá realizar la instalación. Tengamos esto en cuenta si un usuario nos llama reclamando que no puede instalar un programa.

El servicio de Windows Installer también proporciona una API para que los desarrolladores puedan utilizarlo para incluir instalaciones por demanda dentro de las aplicaciones. También ofrece un mecanismo de Rollback (vuelta atrás) si la instalación falla, de modo que es posible regresar al estado anterior de la instalación, por lo que podemos decir que ésta es transaccional. Esto es muy importante, sobre todo, porque nos asegura que, pase lo que pase durante la instalación, no quedarán inconsistencias que pongan en peligro la ejecución de la aplicación: si algo falla, la aplicación no se instalará; en caso contrario, se instalará por completo.

Para crear un Package de Windows Installer, podemos utilizar las Windows Installer SDK Tools, pero esta opción no es "muy amigable" en cuanto a su uso. Por eso, Microsoft ha integrado la creación de este tipo de instaladores dentro del entorno de desarrollo de Visual Studio, en una categoría de proyectos denominados **Setup and Deployment**.

Con este tipo de proyectos, podemos tener un control completo sobre la instalación de las aplicaciones, y es el que deberíamos utilizar en la mayorías de los casos, debido al grado de control que tendremos sobre ellas. No obstante, es preciso contar con Visual Studio Professional, como mínimo, para usar esta clase de proyectos.

ClickOnce

Otro tipo de distribución más avanzado que XCopy y más eficiente para la distribución de aplicaciones por Internet e intranets es una nueva tecnología incorporada en el .NET Framework 2.0, llamada ClickOnce. Su intención inicial es simplificar la distribución de aplicaciones sin intervención del usuario, y lograr que sea robusta y aplicable a una gran variedad de casos. La principal característica de ClickOnce es que,

para realizar la instalación, el usuario accede a una página Web y hace clic en un link, tal como si bajara un archivo, pero en realidad, está instalando la aplicación. Una vez que lo hace, si vuelve a hacer clic en el link, ya no la instalará, sino que directamente la ejecutará. Además, y ésta es una de las ventajas fundamentales, si subimos una nueva versión, cuando el usuario vuelva a ejecutar la aplicación, ésta se actualizará automáticamente desde el servidor, hecho que facilitará muchísimo la distribución de las actualizaciones y correcciones. Entre otras ventajas de este sistema, podemos mencionar:

- Las aplicaciones que se instalan o ejecutan con ClickOnce son independientes entre sí. Cada una es instalada y ejecutada por un usuario y aplicación, de manera que no hay conflicto con otras.
- Se puede elegir que la aplicación funcione de manera offline u online; es decir, que se instale en el cliente y pueda trabajar sin estar conectado a la red, o que se comporte como una aplicación Web, de forma tal que cuando el usuario seleccione un enlace de una página Web, la aplicación se ejecute directamente. Para la ejecución offline se instalará un acceso directo en el menú de Inicio de Windows.
- Contiene una API para controlar el proceso de actualización si no deseamos que sea automático, y así poder controlarlo por código. En la primera instalación, ClickOnce instala sólo los archivos obligatorios, mientras que aquellos que no lo son podrán bajarse posteriormente con esta API, con lo cual se optimiza el proceso de instalación.
- Las actualizaciones en ClickOnce son transaccionales, es decir que se puede volver atrás si no se realiza completamente y de manera satisfactoria.
- Los manifiestos de aplicación e instalación que genera ClickOnce están firmados digitalmente. Esto proporciona la seguridad de que la aplicación y sus actualizaciones provienen de una fuente segura y confiable.
- ClickOnce tiene la opción de volver a la versión anterior de la aplicación desde el

> El servicio de Windows Installer también proporciona un mecanismo de Rollback.

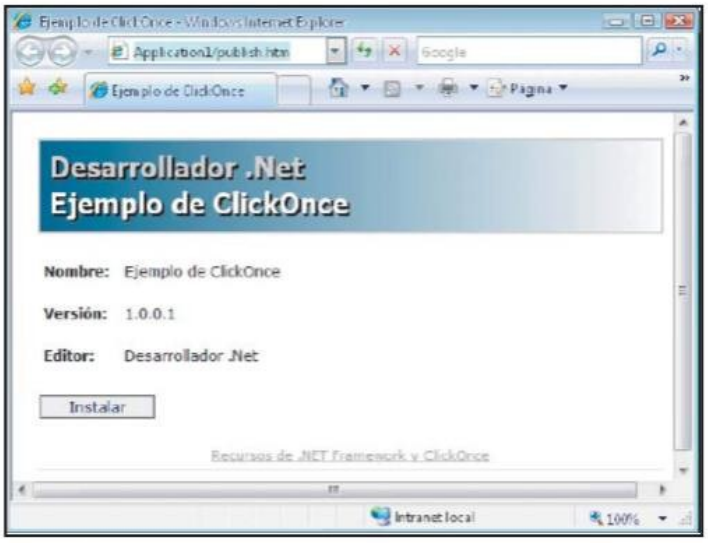


FIGURA 050 | Cuando publicamos una aplicación con ClickOnce, se crea una página a la que el usuario deberá acceder para instalarla.

No necesitamos permisos

A diferencia de lo que sucede con Windows Installer, no se necesitan permisos de administrador para instalar aplicaciones distribuidas mediante la técnica de ClickOnce. Tampoco se crean iconos en el Escritorio ni entradas en el Registro. Al igual que con los MSI, la aplicación sí aparecerá en la lista de programas instalados, en el Panel de control.

Argentina $8,90 (recargo al interior $0,20) / México: $45

USERS

Microsoft

Curso teórico y práctico de programación

Desarrollador .net

Con toda la potencia de **Visual Basic .NET** y **C#**

La mejor forma de aprender a programar desde cero

★★★★★

Basado en el programa Desarrollador Cinco Estrellas de Microsoft

13

Enlace de datos

ADO.NET - El objeto BindingSource

Distribución de aplicación

XCOPY - Windows Installer - ClickOnce

ISBN 978-987-1347-43-8
00013
9 789871 347438