Desarrollo de controles Windows propios (II)

Luis Miguel Blanco Ancos

Manipulación del foco y creación de nuevas características

Retomando el camino tras el descanso

En la primera parte de este artículo tratamos los puntos básicos de la creación de controles Windows partiendo de cero. Revisamos su creación inicial, el dibujo de elementos en la superficie de trabajo, la escritura de código para dotarles de funcionalidad y algunos otros aspectos. En esta segunda parte completaremos el desarrollo de nuestro control, abordando los puntos que nos quedaron pendientes, así que sin más dilación, comencemos.

Establecimiento de una imagen personalizada para el control en el Cuadro de herramientas del IDE

Antes de proseguir con la siguiente fase de construcción del control, vamos a asignarle una imagen para que aparezca en el Cuadro de herramientas de VS.NET cuando utilicemos el control en un formulario.

Tras seleccionar un archivo de tipo bitmap, icono, etc., que contenga la mencionada imagen, y copiarlo en el directorio de nuestro proyecto, deberemos aplicar a la clase del control el atributo ToolboxBitmap, que será el encargado de asociar la imagen al control.

La vía más fácil de uso de este atributo consiste en pasar a su constructor una cadena con la ruta en la que reside el archivo de imagen. De esta forma, si la imagen está en un archivo de icono con el nombre Rombo.ico, en la ruta *E:\PruebasVBNET\ControlScratch*, usaríamos la siguiente instrucción.

```
**Atributo para asignar a un control una imagen para el Cuadro de herramientas**
<ToolboxBitmap("E:\PruebasVBNET\ControlScratch\Rombo.ico")> _
Public Class CheckRomboBox
   Inherits Control

'....
'....

End Class
```

Esta técnica tiene el inconveniente de que debemos utilizar la ruta absoluta en la que está la imagen, por lo que si necesitáramos cambiar la ubicación del proyecto en el disco duro, o copiarlo en otro equipo con una unidad de disco diferente, deberemos modificar la cadena con la ruta o se producirá un error al no encontrar el archivo con la imagen.

Para solucionar esta traba debemos encarar el problema con un enfoque ligeramente distinto: en lugar de acceder a la imagen a través de la ruta de su archivo, debemos incluir el propio archivo como un recurso incrustado en nuestro proyecto y utilizarlo directamente; veamos como hacer esta tarea.

En primer lugar, mediante la opción de menú de VS.NET *Proyecto + Agregar elemento existente*, seleccionaremos el archivo con la imagen y lo añadiremos al proyecto del control; en nuestro ejemplo este archivo es Rombo.ico.

Esta acción por defecto no establece el archivo como un recurso incrustado, por lo que tenemos que ir a la ventana de propiedades del archivo, y en la propiedad *Acción de generación* seleccionar el valor *Recurso incrustado*, como vemos en la siguiente figura.

Añadir un archivo de imagen como recurso incrustado al proyecto

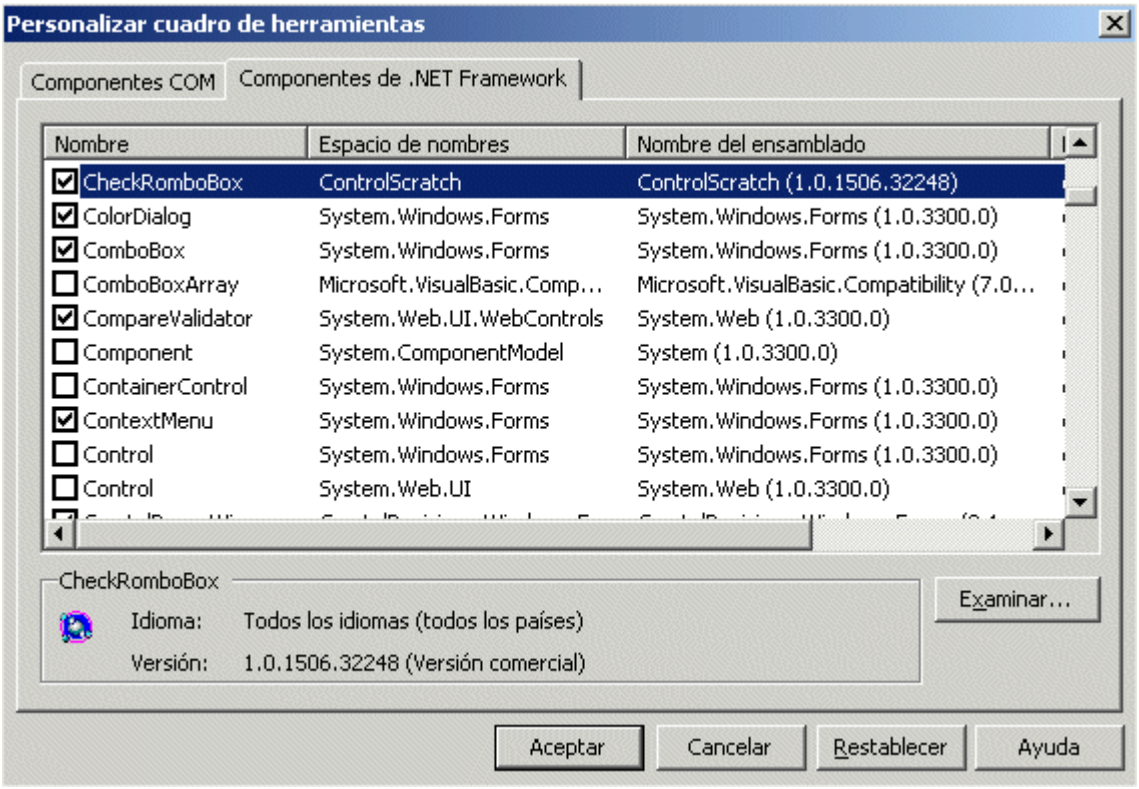
Ahora aplicaremos el atributo ToolboxBitmap a la clase CheckRomboBox como vemos en el siguiente código.

Atributo para asignar a control propio un recurso de imagen para el Cuadro de herramientas
<ToolboxBitmap(GetType(CheckRomboBox), "Rombo.ico")> _

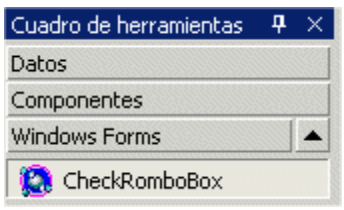
```
Public Class CheckRomboBox
   Inherits Control

'....
'....

End Class
```

En el primer parámetro obtenemos mediante la función GetType el tipo que contiene el recurso incrustado, y en el segundo parámetros indicamos con una cadena el nombre de dicho recurso.

Compilaremos el control y pasaremos al diseñador del formulario. Haciendo clic derecho sobre el Cuadro de herramientas, elegiremos la opción de menú para añadir o quitar elementos, y seleccionaremos el archivo .DLL correspondiente al control, que ya mostrará la imagen incrustada como recurso, según vemos en las siguientes figuras.



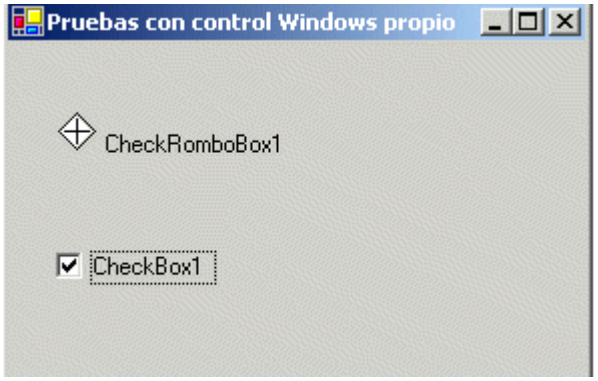
Añadir el control al Cuadro de herramientas



Control personalizado con imagen agregado al Cuadro de herramientas

Gestionando la captura del foco

Volviendo a las comparaciones –odiosas- con ChekBox, el control que estamos utilizando como modelo a imitar, si ejecutamos un formulario que tenga uno de estos controles, nos percataremos que cuando este control recibe el foco mediante la pulsación de la tecla [Tab], queda rodeado por un rectángulo de puntos, que Windows utiliza como guía estándar para indicar al usuario que todas las entradas de teclado del programa serán a partir de ese momento recibidas por el control.



Cuando un control CheckBox recibe el foco es rodeado por un rectángulo de puntos

Nuestro control también recibe el foco de entrada de la aplicación, consecuencia de heredar de la clase Control, que implementa toda la maquinaria de gestión del foco, pero sin embargo no proporciona automáticamente la funcionalidad de señalar visualmente esta circunstancia. Las motivaciones son muy claras: supongamos que el diseño de un control propio exige que el foco se indique mediante un sistema de colores, o quizá que no se indique en absoluto, en esta situación el rectángulo de puntos constituye un estorbo.

Por estas causas el programador tiene que implementar él mismo su sistema de guía visual para indicar al usuario que el control ha obtenido el foco; supone más trabajo de codificación, pero como beneficio obtenemos una gran flexibilidad, ya que podemos diseñar el método de aviso que mejor se adapte a nuestros requerimientos. En nuestro ejemplo, puesto que vamos a intentar simular en lo posible un tipo de control ya existente, utilizaremos la técnica clásica en Windows del rectángulo de puntos.

Primeramente declararemos un nuevo campo en la clase con el nombre mbTieneFoco, de tipo Boolean, que nos servirá para guardar el estado del foco. Podríamos utilizar la propiedad ContainsFocus, que se hereda de la clase Control, y proporciona la misma información; sin embargo, esta propiedad llama internamente al API de Windows para obtener el estado del foco, por lo que podría suponer una penalización en el rendimiento.

A continuación tenemos que detectar los momentos en los que el control gana y pierde el foco, para ello disponemos respectivamente de los métodos OnGotFocus y OnLostFocus, que reemplazaremos en la forma mostrada a continuación.

```
**Métodos para la detección de la captura y liberación del foco**
Private mbTieneFoco As Boolean

'---------------------------------
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
   ' al ganar el foco
   MyBase.OnGotFocus(e)
   mbTieneFoco = True
   Me.Invalidate()
End Sub

'---------------------------------
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
   ' al perder el foco
   MyBase.OnGotFocus(e)
   mbTieneFoco = False
   Me.Invalidate()
End Sub
```

Para detectar que el control recibe el foco también al hacer clic con el ratón, añadiremos en el método OnClick, una llamada al método base Focus, que asigna directamente el foco al control.

```
**Método de pulsación del ratón sobre el control**
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
   MyBase.OnClick(e)

   ' al hacer clic asignamos directamente el foco al control
   MyBase.Focus()
   ' cambiamos el estado de marcado
   mbMarcado = Not mbMarcado
   Me.DibujarMarca()
End Sub
```

Ahora se nos plantea el siguiente problema: ¿Cómo dibujamos el rectángulo de puntos que tienen los controles Windows cuando reciben el foco?. Bien, el dibujo del rectángulo en sí no representa un problema, ya que la clase ControlPaint, diseñada para automatizar ciertas tareas con el dibujo de controles nos provee del método

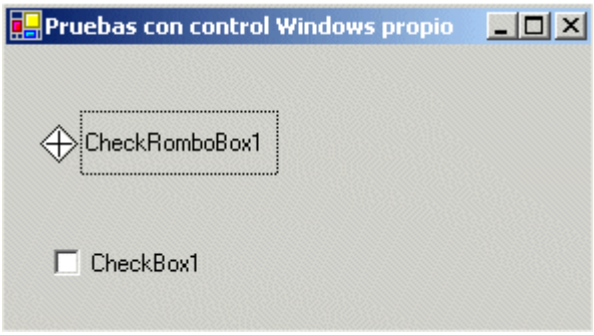
DrawFocusRectangle, que como su propio nombre desvela, sirve para pintar el rectángulo de puntos estándar, indicativo de que un control ha recibido el foco de entrada de la aplicación.

Con esta información escribiremos un nuevo método al que daremos el nombre DibujarRectanguloFoco, y que utilizaremos para llamar a ControlPaint.DrawFocusRectangle, siempre y cuando el control haya ganado el foco. A este nuevo método lo llamaremos desde OnPaint.

```
**Dibujo de un rectángulo de puntos sobre el control**
Private Sub DibujarRectanguloFoco(ByVal oGraphics As Graphics)
   ' si el control tiene el foco
   If mbTieneFoco Then
      Dim oRectFoco As Rectangle
      oRectFoco = New Rectangle(21, 0, Me.Width - 21, Me.Height)
      ' dibujar el rectángulo de puntos que lo indica
      ControlPaint.DrawFocusRectangle(oGraphics, oRectFoco)
   End If
End Sub

'---------------------------------
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
   '....
   '....
   Me.DibujarRectanguloFoco(oGraphics)
   '....
End Sub
```

Como cada vez que ocurren los eventos OnGotFocus y OnLostFocus invalidamos el área del control, produciéndose por consiguiente, una nueva llamada a OnPaint, las operaciones de pintar y borrar el rectángulo de puntos se simplifican.

A partir de ahora al cambiar de foco entre los controles del formulario, podremos comprobar gráficamente cuándo nuestro control recibe el foco, como muestra la siguiente figura.

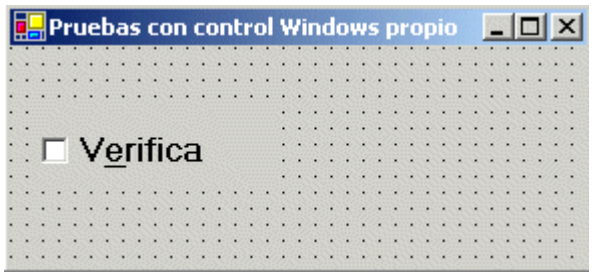


Control personalizado rodeado del rectángulo de puntos indicativo de captura del foco

Asignación y captura de atajos de teclado

Un atajo de teclado o hotkey consiste en una combinación de teclas que provoca una acción sobre el control aunque este no tenga el foco. Por ejemplo, en un control Button o CheckBox provocarían su pulsación, lanzando el evento Click.

Para definir un atajo de teclado, el programador en la fase de diseño debe anteponer el carácter & a la letra que quiere definir como la disparadora del evento asociado. El reflejo visual inmediato de esta acción sobre el control será que dicha letra quedará subrayada en su título. Si en la propiedad Text de un control CheckBox escribimos el valor V&erifica, dicho control se mostrará igual que en la siguiente figura.



CheckBox mostrando el carácter subrayado de atajo de teclado

Al ejecutar el formulario, cada vez que pulsemos la combinación de teclas [Alt] + [E] se producirá el evento Click sobre este control.

Implementar esta funcionalidad en un control propio no es tarea baladí, ya que requiere el acople preciso de diversas piezas; pero sí resulta muy interesante; vamos por lo tanto a describir los pasos necesarios para llevar a cabo este proceso.

En primer lugar, para que el texto del control muestre el carácter designado como atajo de teclado (subrayado), en el método DibujarTexto de nuestro control tomaremos el objeto StringFormat que habíamos creado previamente, y asignaremos a su propiedad HotKeyPrefix el valor Show. Esta propiedad acepta los valores de la enumeración HotKeyPrefix que indicamos a continuación.

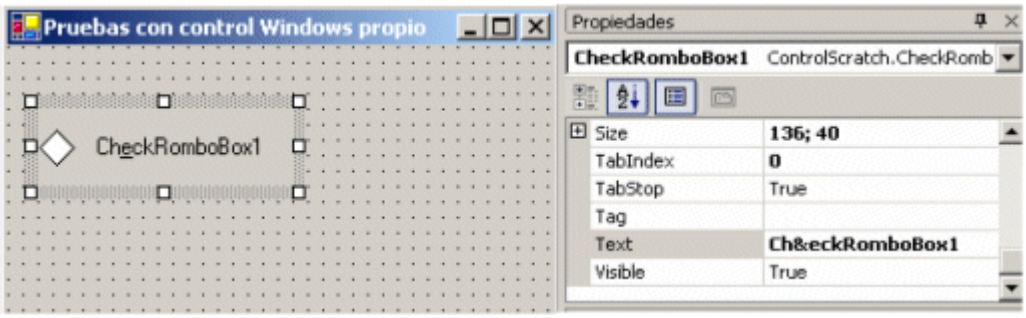
- **Hide**. No muestra el carácter subrayado.
- **None**. No utiliza indicaciones gráficas para tecla de atajo.
- **Show**. Muestra el carácter subrayado.

También debemos reemplazar el método generador de evento OnTextChanged, para que cada vez que cambiemos el atajo de teclado, invalidemos el área del control

provocando una llamada a OnPaint, de manera que se vuelvan a dibujar todos sus elementos, reflejando así la nueva hotkey.

```
**Dibujo del texto o título del control**
Public Sub DibujarTexto(ByVal oGraphics As Graphics)
   '....
   '....
   oStrFormat.HotkeyPrefix = Drawing.Text.HotkeyPrefix.Show
   '....
End Sub

Protected Overrides Sub OnTextChanged(ByVal e As System.EventArgs)
   MyBase.OnTextChanged(e)
   Me.Invalidate()
End Sub
```

Realizados estos cambios y tras la correspondiente compilación, nuestro control visualizará, subrayado, el carácter elegido como hotkey.



Control propio que implementa la funcionalidad visual de atajo de teclado

El uso de un atajo de teclado va a requerir la realización de llamadas y declaraciones al API de Windows, por lo que para tener este código mejor ubicado, añadiremos al proyecto un módulo con el nombre DeclaracionesAPIWin, en el que incluiremos los siguiente elementos.

- La declaración de la constante WM_HOTKEY para capturar el mensaje producido al pulsar un atajo de teclado.
- Una enumeración que llamaremos HotKeyModificadores, cuyos miembros nos servirán para identificar la tecla especial pulsada en la combinación de la hotkey.
- La función RegisterHotKey, que usaremos para registrar en el sistema la combinación de teclas que compondrán nuestro atajo de teclado.

- La función GlobalAddAtom, cuya finalidad consiste en crear un identificador único a partir de una cadena, y que necesitaremos para registrar nuestro hotkey.

El código de estas declaraciones se muestra en el siguiente fuente.

```
**Declaraciones del API de Windows para manejar atajos de teclado**
Imports System.Runtime.InteropServices

Module DeclaracionesAPIWin

   ' constante que identifica el mensaje
   ' de Windows correspondiente a la pulsación de
   ' un hotkey
   Public Const WM_HOTKEY = &H312

   ' enumeración con los códigos de tecla
   ' especiales que se combinan para
   ' componer un hotkey
   Public Enum HotkeyModificadores
      MOD_ALT = &H1
      MOD_CONTROL = &H2
      MOD_SHIFT = &H4
      MOD_WIN = &H8
   End Enum

   ' registra un hotkey,
   ' parámetros:
   ' - hWnd: manipulador de ventana o control al que se asigna
   ' el atajo de teclado
   ' - id: identificador único para el hotkey
   ' - fsModifiers: tecla (Alt, Control, etc.) especial
   ' que es pulsada en combinación con la tecla
   ' definida en el parámetro vkey
   ' - vkey: tecla definida para el atajo de teclado
   <DllImport("user32", EntryPoint:="RegisterHotKey", _
   SetLastError:=True, _
   ExactSpelling:=True, _
   CallingConvention:=CallingConvention.StdCall)> _
   Public Function RegisterHotkey(ByVal hWnd As IntPtr, _
      ByVal Id As Int32, _
      <MarshalAs(UnmanagedType.U4)> ByVal fsModifiers As Int32, _
      <MarshalAs(UnmanagedType.U4)> ByVal vkey As Int32) As Boolean

   End Function
```

```
    ' crea un identificador único en el sistema
    ' parámetros:
    ' - lpString: cadena usada para crear el identificador
    <DllImport("kernel32", EntryPoint:="GlobalAddAtom", _
    SetLastError:=True, _
    ExactSpelling:=False)> _
    Public Function GlobalAddAtom(<MarshalAs(UnmanagedType.LPTStr)> ByVal
lpString As String) As Int32

    End Function

End Module
```

A continuación crearemos un nuevo método al que daremos el nombre RegistrarHotKey, y en el que realizaremos las llamadas correspondientes al API de Windows que hemos declarado anteriormente, veamos su código fuente.

```
**Registro de la tecla hotkey para el control**
Public Sub RegistrarHotKey()
    ' comprobar si hay un carácter hotkey
    ' en el título del control
    Dim aCaracteres() As Char
    Dim bHayMnemonic As Boolean
    Dim chrCaracter As Char
    Dim nCodigoChar As Integer
    Dim hWnd As IntPtr
    Dim nIDHotKey As Integer
    Dim oCodificador As System.Text.ASCIIEncoding
    Dim aCodigosCar() As Byte

    ' obtener los caracteres del título del control
    aCaracteres = Me.Text.ToCharArray()

    ' recorrer el array de caracteres y comprobar
    ' si alguno está definido como atajo de teclado
    bHayMnemonic = False
    For Each chrCaracter In aCaracteres
        If Me.IsMnemonic(chrCaracter, Me.Text) Then
            bHayMnemonic = True
            Exit For
        End If
    Next

    ' si hay atajo de teclado
    If bHayMnemonic Then
```

```
        ' crear un objeto para obtener el código numérico
        ' del carácter correspondiente a la tecla pulsada
        oCodificador = New System.Text.ASCIIEncoding

        ' convertir a mayúsculas la tecla pulsada
        ' y obtener su código
        aCodigosCar = oCodificador.GetBytes(CType(chrCaracter.ToString(),
String).ToUpper())

        nCodigoChar = aCodigosCar(0)

        ' obtener el manipulador del control
        hWnd = Me.Handle

        ' crear un identificador único para el atajo de teclado
        nIDHotKey = GlobalAddAtom("HotKeyRombo")

        RegisterHotkey(hWnd, _
           nIDHotKey, _
           HotkeyModificadores.MOD_ALT, _
           nCodigoChar)

    End If

End Sub
```

La mecánica utilizada en este método es la siguiente: en primer lugar obtenemos la cadena del título del control para transformarla en un array de caracteres, que recorremos después para comprobar si hay alguno definido como hotkey; esta comprobación la efectuamos valiéndonos del método IsMnemonic, que heredamos de la clase Control, y que devuelve un valor lógico al buscar en una cadena si un determinado carácter corresponde a un atajo de teclado.

Si hemos encontrado un carácter que actúa como hotkey, debemos extraer del mismo su código numérico, lo que logramos utilizando el método GetBytes del objeto ASCIIEncoding, perteneciente al espacio de nombres System.Text; nótese que antes de recuperar el código del carácter lo convertimos a mayúsculas.

Debido a que el resultado del código se devuelve en forma de array de Bytes, tenemos que recuperar de dicho array el primer y único elemento.

El siguiente paso consiste en obtener el manipulador de ventana del control. Para aquellos lectores que no se encuentren familiarizados con los conceptos de gestión interna de ventanas, debemos aclarar que Windows trata a todos los elementos de una aplicación como ventanas, de ahí que asigne un código identificador –genéricamente denominado *manipulador de ventana* (hWnd)- tanto a la ventana del

programa como a todos sus controles; esto permite tener identificados a todos los componentes de una ventana para realizar con ellos las operaciones necesarias. En la clase Control, el manipulador lo obtenemos gracias a la propiedad Handle, de tipo IntPtr, que representa un puntero o identificador del sistema.

A continuación creamos un identificador para el atajo de teclado llamando a la función del API GlobalAddAtom, y finalmente, con todos los valores que acabamos de obtener, registraremos nuestra combinación de teclas ejecutando la función del API RegisterHotKey.

La siguiente fase consiste en determinar cuándo y dónde debemos llamar a nuestro método RegistrarHotKey, para lo que debemos buscar un método generador de evento que se produzca una vez que el control se encuentre completamente construido y disponga de manipulador de ventana, elemento que como sabemos, resulta imprescindible para registrar adecuadamente el atajo de teclado.

En este ejemplo hemos optado por reemplazar el método InitLayout de la clase Control, que se produce una vez que el control se ha añadido a su contenedor –el formulario-. El código fuente se muestra a continuación.

```
**Punto del control en el que registramos el atajo de teclado**
Protected Overrides Sub InitLayout()
   ' cuando el control se agregue a su contenedor
   ' llamar al registro de hotkey
   MyBase.InitLayout()
   Me.RegistrarHotKey()
End Sub
```

Una vez registrada una combinación de teclas como hotkey dentro de un control, cada vez que el usuario pulse dicha combinación, Windows creará un mensaje de tipo WM_HOTKEY, y lo enviará al control en cuestión. El control por su parte, deberá encontrarse preparado para recibir dicho mensaje y tratarlo adecuadamente.

La forma personalizada de procesar los mensajes que recibe una ventana o control pasa por escribir una función que contenga una estructura de estilo Select Case, en la que, dependiendo del mensaje recibido, se ejecutará una determinada operación sobre la ventana o control. Esta función, denominada WindowProc (procedimiento de ventana) en la documentación del API de Windows, deberá ser conectada al control para que el sistema operativo envíe los mensajes que le correspondan.

Para facilitar el trabajo al programador, la plataforma .NET ya incluye en la clase Control un método con este comportamiento: WndProc, que es llamado internamente cada vez que se produce un mensaje dirigido al control. Si lo reemplazamos e implementamos una estructura de captura de mensajes personalizada como hemos indicado en el párrafo anterior, podremos manipular los mensajes que necesitemos.

Cómo en nuestro caso sólo nos interesa manipular el mensaje producido al pulsar una hotkey escribiremos el siguiente código.

```
**Procedimiento de captura de mensajes en el control**
' procesador de los mensajes que Windows envía a este control
Protected Overrides Sub WndProc(ByRef m As System.Windows.Forms.Message)
   ' estructura de proceso de mensajes
   Select Case (m.Msg)
      Case WM_HOTKEY
         ' cuando se detecte el mensaje de pulsación
         ' del atajo de teclado, marcar/desmarcar la casilla
         ' del control
         mbMarcado = Not mbMarcado
         Me.DibujarMarca()
   End Select

   ' llamar a la estructura de proceso de mensajes por defecto
   ' para que el mensaje sea tratado con el comportamiento base
   MyBase.WndProc(m)
End Sub
```

En el anterior fuente, al pulsar la combinación de teclas para el control y producirse el mensaje WM_HOTKEY, actualizaremos el campo de la clase que indica el estado de la casilla y llamaremos al método DibujarMarca, encargado de reflejar dicho estado.

Un control, no obstante, recibe innumerables mensajes de todo tipo durante el tiempo que la aplicación permanece en ejecución. ¿Qué ocurre con el resto de mensajes a los que no estamos interesados en dar soporte?. Debemos tener presente que todos los mensajes deben ser procesados, o en caso contrario, el control presentará un comportamiento irregular.

Esta cuestión la resolvemos de una manera muy sencilla: al final de este método llamamos a la implementación de WndProc existente en la clase base, que se encargará de ejecutar el comportamiento por defecto del control.

Llegados a este punto, lo natural es pensar que ya podemos situar nuestro control en el formulario, y tras asignarle una tecla hotkey, ejecutar la aplicación marcando y desmarcando la casilla mediante dicha combinación de teclas. Bien, pues en el estado actual del control esto es así pero sólo hasta cierto punto.

Supongamos que después de añadir el control al formulario, definimos el carácter R como hotkey; a continuación ejecutamos el proyecto procurando que tanto el formulario en ejecución como el diseñador del formulario de VS.NET estén visibles; pulsamos la combinación de teclas [Alt] + [R] y… ¡sorpresa!, el control que reacciona

ante dicha pulsación no es el del formulario en ejecución sino el del diseñador. ¿Perplejo estimado lector?, pues no se asuste que esto tiene la debida explicación.

Cada vez que en tiempo de diseño –y recalco porque es importante, en tiempo de diseño- interactuamos con un control, este ejecuta su código. Quizá es algo que nos ha pasado inadvertido hasta ahora, pero tomemos como ejemplo el método OnTextChanged, ¿qué le ocurre al control cuando asignamos o cambiamos su propiedad Text para establecer un carácter como hotkey?, pues que en tiempo de diseño se ejecuta OnTextChanged para actualizar el título del control.

Esto nos lleva a la conclusión de que el control funciona bajo dos contextos (host) de ejecución: el del IDE de Visual Studio .NET, y el del ejecutable correspondiente al proyecto que estamos desarrollando. Como actualmente no hacemos distinción de cuál es dicho contexto de ejecución, siempre toma por defecto el del entorno de desarrollo de VS.NET, registrándose el atajo de teclado para este en lugar de hacerlo para el ejecutable real, que es el que de verdad nos interesa.

¿Cómo solucionamos este entuerto?, pues comprobando el valor de la propiedad DesignMode, que nos indicará si el control está ejecutándose en modo de diseño o no. Esta propiedad pertenece a Component, que es la clase base o padre de Control.

Vamos a efectuar esta comprobación desde InitLayout, el mismo método en el que llamamos a RegistrarHotKey, quedando como vemos en el siguiente fuente.

```
**Registro del atajo de teclado usando DesignMode**
Protected Overrides Sub InitLayout()
   ' cuando el control se agregue a su contenedor
   MyBase.InitLayout()

   ' si el control no se ejecuta en modo de diseño,
   ' registrar la hotkey
   If Not (Me.DesignMode) Then
      Me.RegistrarHotKey()
   End If
End Sub
```

Aunque la anterior técnica es la oficialmente reconocida para comprobar si un control se encuentra en tiempo de diseño o ejecución, existe un medio algo más rudimentario y artificioso para efectuar esta operación y que requiere más código, por lo que no es recomendable su uso; veamos en qué consiste.

Si toda aplicación Windows es un ejecutable, el IDE de VS.NET no va a ser menos, y en efecto así es, ya que el archivo devenv.exe corresponde al ejecutable del entorno de desarrollo; lo que debemos hacer por lo tanto es detectar cuándo el control se ejecuta bajo devenv.exe, es decir, dentro del IDE, y cuándo lo hace bajo el ejecutable del

proyecto. Para ello usaremos el método ExecutablePath del objeto Application, que devuelve una cadena con la ruta y nombre del ejecutable que actúa como contexto de ejecución del formulario; cuando constatemos que no se trata del IDE de desarrollo, sólo entonces, registraremos el atajo de teclado correspondiente al control.

```
**Registro del atajo de teclado detectando el ejecutable**
Protected Overrides Sub InitLayout()
   ' cuando el control se agregue a su contenedor
   MyBase.InitLayout()

   Dim sRutaEjecutable As String
   Dim sNombreEjecutable As String

   ' obtener la ruta del ejecutable contenedor
   ' del control
   sRutaEjecutable = Application.ExecutablePath()

   ' extraer el nombre del archivo ejecutable
   sNombreEjecutable = System.IO.Path.GetFileName(sRutaEjecutable)

   ' si el ejecutable no es el IDE
   ' registrar la hotkey
   If sNombreEjecutable <> "devenv.exe" Then
      Me.RegistrarHotKey()
   End If
End Sub
```

Como detalle a destacar del anterior fuente, ya que sólo nos interesa el nombre del archivo ejecutable devuelto por ExecutablePath, lo extraemos usando el método GetFileName del objeto Path, situado en el espacio de nombres System.IO.

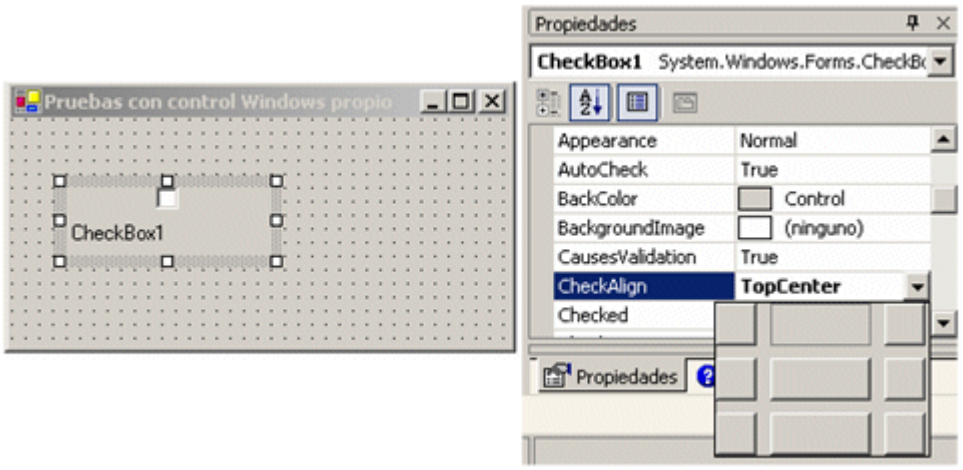
Llegados a este punto, damos por finalizada la tercera fase de refinamiento del control.

Cambiando la posición de la casilla

Como estará comprobando el lector por el trabajo desarrollado hasta el momento, emular al completo la apariencia y comportamiento de un control ya existente no es una tarea menospreciable, pero si muy apasionante, ya que estamos descubriendo detalles que de otra manera podrían pasarnos inadvertidos. No obstante .NET Framework provee al programador con los elementos necesarios para llevar a cabo su objetivo.

Otro de los aspectos pertenecientes al CheckBox estándar que nos puede interesar adaptar en nuestro control radica en la posibilidad de cambiar la alineación de la casilla de verificación dentro del área del control.

El control CheckBox dispone para ello de la propiedad CheckAlign, que utilizada desde la ventana de propiedades del IDE nos permite establecer la mencionada posición usando una guía visual como muestra la siguiente figura.



Guía de alineación de la casilla de un CheckBox

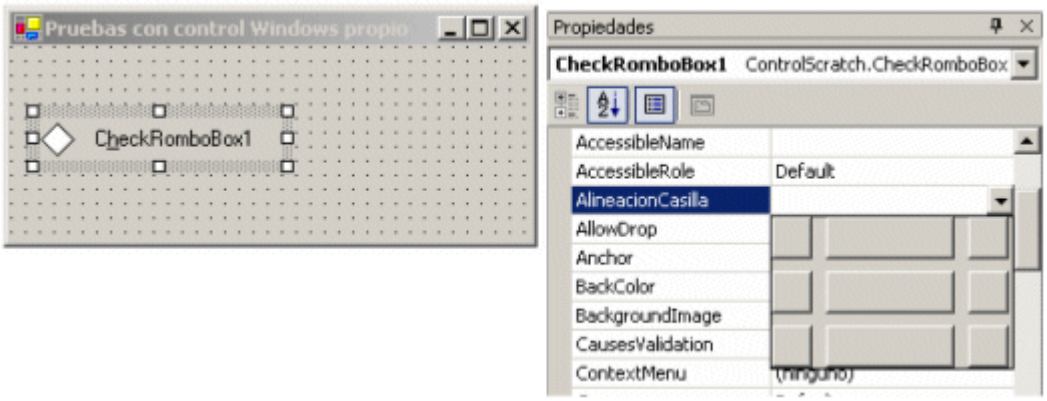
Posiblemente estaremos pensando que implementar una propiedad de estas características para nuestro control es algo muy complejo, ¿cómo me las voy a apañar para construir la utilidad de selección visual?. Pues aunque parezca increíble esto es lo más fácil, veámoslo a continuación.

La propiedad CheckAlign del control CheckBox pertenece al tipo enumerado ContentAlignment, por lo que si añadimos a la clase de nuestro control un campo de este tipo con su correspondiente procedimiento Property, automáticamente dispondremos en el diseñador de esta guía visual. El código a escribir sería el siguiente.

```
**Creación de propiedad para alineación de la casilla del control**
' campos
Private moAlineacionCasilla As ContentAlignment
'....
'....
Public Property AlineacionCasilla() As ContentAlignment
   Get
      Return moAlineacionCasilla
   End Get
   Set(ByVal Value As ContentAlignment)
      moAlineacionCasilla = Value
      Me.Invalidate()
```

```
    End Set
End Property
```

Después de compilar el control, volveremos al formulario de pruebas, y al seleccionar la nueva propiedad AlineacionCasilla, ya dispondremos de esta guía visual como muestra la siguiente figura.



Control propio implementando una guía de alineación de la casilla

Ahora nos queda el trabajo más importante, conseguir que el control reaccione coordinadamente cada vez que seleccionamos una nueva posición para alinear la casilla de verificación.

Desafortunadamente, en aras de la flexibilidad, toda la lógica que teníamos articulada para el dibujo de la casilla y el texto del título acaba de quedar inservible. Si antes, la posición de los elementos del control era estática, ahora debe ser dinámica, lo que significa que al cambiar el estado de la propiedad AlineacionCasilla, deberemos invalidar el área del control y volverlo a dibujar en base a las coordenadas de la nueva posición.

Para adaptarnos a este nuevo comportamiento, vamos a añadir a la clase dos arrays de tipo Point para guardar las coordenadas del rombo exterior e interior; un array también Point para las posiciones de las líneas que forman la marca de la casilla; y un objeto RectangleF que contendrá el área del control en el que se dibujará el título.

```
**Campos para almacenar coordenadas**
Private maCoorRomboExt(4) As Point
Private maCoorRomboInt(4) As Point
Private maCoorMarca(3) As Point
Private moRectTexto As RectangleF
```

Seguidamente vamos a escribir un constructor para la clase en el que especificaremos una posición por defecto para la casilla.

```
**Constructor del control**
Public Sub New()
   MyBase.New()

   ' alineación de casilla inicial
   moAlineacionCasilla = ContentAlignment.MiddleLeft
End Sub
```

A continuación crearemos dos métodos con los nombres CalcularCoorRombo y CalcularCoorTexto, en los que en función de la alineación de la casilla, calcularemos las posiciones en las que dibujaremos los rombos, marca y título, almacenando los valores resultantes en los arrays de objetos Point y el RectangleF. Estos métodos serán llamados al comienzo de OnPaint.

```
**Métodos para calcular coordenadas de los elementos del control**
Public Sub CalcularCoorRombo()
   ' calcular las coordenadas del rombo exterior
   ' en función de la alineación del rombo
   Dim nIzquierda As Integer
   Dim nCentroX As Integer
   Dim nDerecha As Integer
   Dim nArriba As Integer
   Dim nCentroY As Integer
   Dim nAbajo As Integer

   Select Case moAlineacionCasilla
      Case ContentAlignment.TopLeft
         nCentroX = 10
         nIzquierda = 1
         nDerecha = 19
         nCentroY = 10
         nArriba = 1
         nAbajo = 19

      Case ContentAlignment.MiddleLeft
         nCentroX = 10
         nIzquierda = 1
         nDerecha = 19
         nCentroY = Me.Height / 2
         nArriba = (Me.Height / 2) - 9
         nAbajo = (Me.Height / 2) + 9
```

```
Case ContentAlignment.BottomLeft
   nCentroX = 10
   nIzquierda = 1
   nDerecha = 19
   nCentroY = Me.Height - 10
   nArriba = Me.Height - 19
   nAbajo = Me.Height - 1

Case ContentAlignment.TopCenter
   nCentroY = 10
   nArriba = 1
   nAbajo = 19
   nCentroX = Me.Width / 2
   nDerecha = (Me.Width / 2) + 9
   nIzquierda = (Me.Width / 2) - 9

Case ContentAlignment.MiddleCenter
   nCentroY = Me.Height / 2
   nArriba = (Me.Height / 2) - 9
   nAbajo = (Me.Height / 2) + 9
   nCentroX = Me.Width / 2
   nDerecha = (Me.Width / 2) + 9
   nIzquierda = (Me.Width / 2) - 9

Case ContentAlignment.BottomCenter
   nCentroY = Me.Height - 10
   nArriba = Me.Height - 19
   nAbajo = Me.Height - 1
   nCentroX = Me.Width / 2
   nDerecha = (Me.Width / 2) + 9
   nIzquierda = (Me.Width / 2) - 9

Case ContentAlignment.TopRight
   nCentroY = 10
   nArriba = 1
   nAbajo = 19
   nCentroX = Me.Width - 10
   nDerecha = Me.Width - 1
   nIzquierda = Me.Width - 19

Case ContentAlignment.MiddleRight
   nCentroY = Me.Height / 2
   nArriba = (Me.Height / 2) - 9
   nAbajo = (Me.Height / 2) + 9
   nCentroX = Me.Width - 10
   nDerecha = Me.Width - 1
```

```
            nIzquierda = Me.Width - 19

        Case ContentAlignment.BottomRight
            nCentroY = Me.Height - 10
            nArriba = Me.Height - 19
            nAbajo = Me.Height - 1
            nCentroX = Me.Width - 10
            nDerecha = Me.Width - 1
            nIzquierda = Me.Width - 19

    End Select

    ' pasar los valores calculados a los arrays que
    ' contienen las coordenadas del rombo exterior, interior
    ' y líneas de la marca
    maCoorRomboExt(0) = New Point(nIzquierda, nCentroY)
    maCoorRomboExt(1) = New Point(nCentroX, nArriba)
    maCoorRomboExt(2) = New Point(nDerecha, nCentroY)
    maCoorRomboExt(3) = New Point(nCentroX, nAbajo)
    maCoorRomboExt(4) = New Point(nIzquierda, nCentroY)

    maCoorRomboInt(0) = New Point(nIzquierda + 1, nCentroY)
    maCoorRomboInt(1) = New Point(nCentroX, nArriba)
    maCoorRomboInt(2) = New Point(nDerecha - 1, nCentroY)
    maCoorRomboInt(3) = New Point(nCentroX, nAbajo)
    maCoorRomboInt(4) = New Point(nIzquierda + 1, nCentroY)

    maCoorMarca(0) = New Point(nCentroX, nArriba + 3)
    maCoorMarca(1) = New Point(nCentroX, nAbajo - 3)
    maCoorMarca(2) = New Point(nIzquierda + 3, nCentroY)
    maCoorMarca(3) = New Point(nDerecha - 3, nCentroY)

End Sub

'---------------------------------
Public Sub CalcularCoorTexto()
    ' calcular el rectángulo para dibujar el título
    ' del control en función de la alineación del rombo
    Select Case moAlineacionCasilla
        Case ContentAlignment.TopLeft, _
            ContentAlignment.MiddleLeft, _
            ContentAlignment.BottomLeft

            moRectTexto = New RectangleF(21, 0, Me.Width - 21, Me.Height)

        Case ContentAlignment.TopRight, _
            ContentAlignment.MiddleRight, _
```

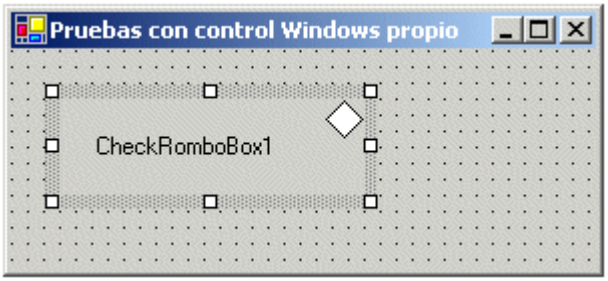
```
            ContentAlignment.BottomRight

            moRectTexto = New RectangleF(0, 0, Me.Width - 21, Me.Height)

        Case ContentAlignment.BottomCenter

            moRectTexto = New RectangleF(0, 0, Me.Width, Me.Height - 21)

        Case ContentAlignment.TopCenter

            moRectTexto = New RectangleF(0, 21, Me.Width, Me.Height - 21)

        Case ContentAlignment.MiddleCenter

            moRectTexto = New RectangleF(0, 0, Me.Width, Me.Height)

    End Select

End Sub

'---------------------------------
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    ' llamar a la versión de esté método de la clase base
    MyBase.OnPaint(e)

    ' calcular coordenadas para dibujar el
    ' rombo del control y el texto del título
    Me.CalcularCoorRombo()
    Me.CalcularCoorTexto()
    '....
    '....
End Sub
```

Por causa de estas nuevas operaciones de cálculo de coordenadas, todos aquellos métodos de la clase que se encargaban de dibujar los diferentes elementos del control deberán ser readaptados. Veamos seguidamente dichos métodos.

```
**Métodos de dibujo del control**
Private Sub DibujarRomboExterior(ByVal oGraphics As Graphics)
    ' dibujar el rombo exterior
    oGraphics.DrawPolygon(New Pen(Color.Black, 1), _
        maCoorRomboExt)
End Sub

' ----------------------------
```

```
Private Sub DibujarRomboInterior(ByVal oGraphics As Graphics)
   ' dibujar el rombo interior
   oGraphics.FillPolygon(New SolidBrush(Color.White), _
      maCoorRomboInt)
End Sub

' ----------------------------
Private Overloads Sub DibujarMarca(ByVal oGraphics As Graphics)
   ' dibujar líneas que indican el control marcado
   ' si el campo mbMarcado es True
   If mbMarcado Then
      ' línea vertical
      oGraphics.DrawLine(New Pen(Color.Black, 1), _
         maCoorMarca(0), maCoorMarca(1))

      ' línea horizontal
      oGraphics.DrawLine(New Pen(Color.Black, 1), _
         maCoorMarca(2), maCoorMarca(3))
   End If
End Sub

' ----------------------------
Public Sub DibujarTexto(ByVal oGraphics As Graphics)
   ' dibujar el título del control
   Dim oStrFormat As New StringFormat()
   oStrFormat.LineAlignment = StringAlignment.Center
   oStrFormat.Alignment = StringAlignment.Center
   oStrFormat.HotkeyPrefix = Drawing.Text.HotkeyPrefix.Show

   oGraphics.DrawString(Me.Text, _
      Me.Font, _
      New SolidBrush(Me.ForeColor), _
      moRectTexto, _
      oStrFormat)
End Sub

' ----------------------------
Private Sub DibujarRectanguloFoco(ByVal oGraphics As Graphics)
   ' dibujar el rectángulo de puntos que
   ' indican que el control tiene el foco
   If mbTieneFoco Then
      Dim focusRect As Rectangle = Me.ClientRectangle
      ControlPaint.DrawFocusRectangle(oGraphics, _
         Rectangle.Ceiling(moRectTexto))
   End If
End Sub
```

El campo de la clase moRectTexto, de tipo RectangleF, es el que usamos para dibujar el texto del control y el rectángulo del foco; para esto último, sin embargo, necesitamos que el rectángulo sea de tipo Rectangle, por lo que para evitar el uso de dos variables separadas que tengan estas coordenadas, usaremos el método compartido Rectangle.Ceiling, que convierte a tipo Rectangle, el objeto RectangleF que se le pasa como parámetro. La siguiente figura muestra un ejemplo de alineación de la casilla distinta de la habitual.



Control propio con capacidad para alinear la casilla de verificación

Coloreando el rombo

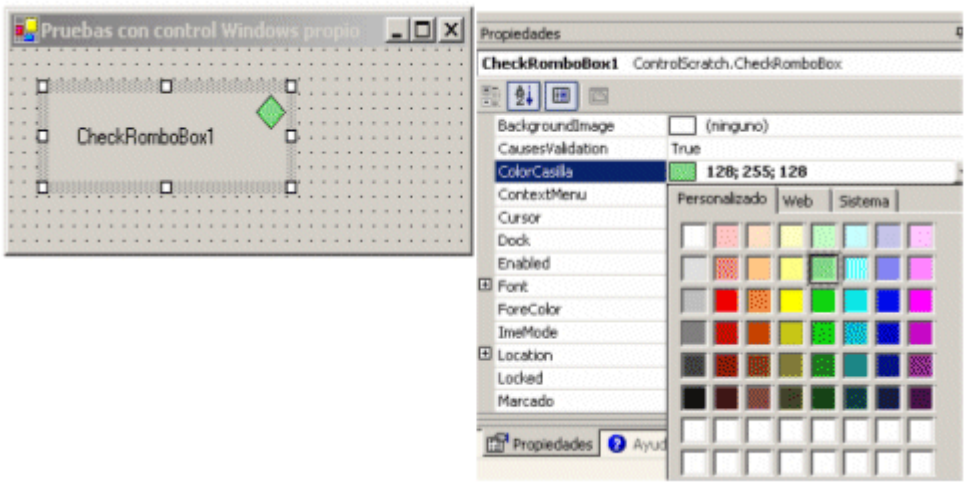
Para finalizar el desarrollo de nuestro control, vamos a dotarle de una nueva característica consistente en aplicar color sobre la casilla de marcado.

A poco que repasemos los pasos seguidos para crear la anterior propiedad, veremos que implementar esta es todavía más fácil. En primer lugar, definiremos en la clase un nuevo campo, moColorCasilla, y un procedimiento de propiedad, ColorCasilla, para almacenar el color de la casilla; seguiremos con el establecimiento por defecto del color blanco para la casilla, en el constructor de la clase; por último, al dibujar el rombo interior utilizaremos el color contenido en el campo moColorCasilla. Repasemos a continuación el código necesario.

Código para dibujar casilla del control con color
```
Private moColorCasilla As Color

'----------------------------
Public Property ColorCasilla() As Color
  Get
    Return moColorCasilla
  End Get
  Set(ByVal Value As Color)
    moColorCasilla = Value
    Me.Invalidate()
  End Set
```

```
End Property

'-----------------------------
Public Sub New()
   MyBase.New()

   ' color de casilla inicial
   moColorCasilla = Color.White

   ' alineación de casilla inicial
   moAlineacionCasilla = ContentAlignment.MiddleLeft
End Sub

'-----------------------------
Private Sub DibujarRomboInterior(ByVal oGraphics As Graphics)
   ' dibujar el rombo interior
   oGraphics.FillPolygon(New SolidBrush(moColorCasilla), _
      maCoorRomboInt)
End Sub
```

Al igual que ocurre con la propiedad creada en el anterior apartado, la paleta de selección de colores que se muestra en la ventana de propiedades es automática, por el simple hecho de haber asignado el tipo Color a la propiedad. Veamos el control en la siguiente figura mostrando un nuevo color para la casilla.



Paleta de selección de color para la casilla del control

Llegados a este punto, damos por finalizada la cuarta y última fase de refinamiento del control.

Final del viaje

Y con esto querido lector, acabamos nuestro periplo por el emocionante mundo del desarrollo de controles desde cero. Si bien no hemos examinado todos y cada uno de los detalles en la fase de construcción de un control de estas características, creemos que sí hemos tratado los puntos primordiales, que sirvan para sentar una base suficiente que los lectores puedan emplear como punto de partida en la creación de sus propios controles.