Desarrollo de controles Windows propios (I)

Luis Miguel Blanco Ancos

Creación de controles a partir de cero (*from scratch*)

La creación de controles con Visual Basic antes de .NET

Cuando hace algunas versiones de VB, Microsoft anunció que proporcionaba la capacidad al programador para la creación de sus propios controles, se marcó un importante hito en la evolución de esta herramienta de desarrollo.

Desde ese momento, partiendo de una plantilla de diseño inicial (al estilo de la plantilla de diseño del clásico formulario), podíamos crear un nuevo control compuesto por uno o varios de los controles ya existentes en el IDE de VB, añadiendo código para ampliar o alterar su funcionalidad y comportamiento, que lo adaptara a los requerimientos de nuestras aplicaciones.

A pesar de estas facilidades, en determinados casos nos quedaba la sensación de que podíamos haber llegado *"un poco más allá"* en la creación del control, es decir, que podíamos haber desarrollado un control mejor (o más a nuestro gusto al menos), si globalmente hubiéramos dispuesto de un mayor nivel de dominio en su proceso de creación. En resumen, nos faltaba la capacidad de crear un control partiendo de cero (*from scratch*); este poder todavía estaba reservado al exclusivo círculo de C++, y conseguirlo desde VB era tarea casi imposible, dada su complejidad y las limitaciones impuestas a VB en aquellos tiempos.

Aunque la situación que acabamos de describir se podía dar en un menor número de ocasiones, debido a que la funcionalidad base proporcionada por los controles existentes era, en general, suficiente para nuestros requisitos; el tener pleno dominio en la capacidad de dibujo sobre el interfaz del control es un aspecto que siempre nos ha atraído.

Pues bien, ya no es necesario convertirnos a la fe de C++, nuestra espera ha terminado; con VB.NET podemos crear controles como siempre habíamos deseado: manipulando todas sus fases de creación.

Controles con VB.NET. Vía libre a la creatividad

La plataforma .NET Framework rompe con todas las barreras que limitaban nuestra capacidad creativa de controles, ofreciéndonos, bajo un modelo de desarrollo más abierto y orientado a objetos, la posibilidad de crear controles Windows partiendo de

cero, o controles propios, como también les denominaremos a lo largo de este artículo.

A través de un conjunto de clases que representan desde el propio control hasta los elementos necesarios para dibujarlo, podremos controlar el más mínimo detalle en su fase de desarrollo. Uno de los beneficios que .NET proporciona con este sistema de trabajo, reside en que todas las clases están adecuadamente organizadas en jerarquías y espacios de nombre, lo que facilita su localización y uso.

Ante un nivel de control tan elevado hay, como contrapartida, un precio que pagar, y esto se traduce en el hecho de que cuanto más queramos que nuestro control tenga un comportamiento estándar con respecto a los controles intrínsecos de Windows, más esfuerzo tendremos que emplear en su desarrollo, por lo que nos veremos obligados a escribir una mayor cantidad de código que contemple todas las características de un control estándar.

No obstante, estamos seguros de que el trabajo y tiempo invertido en la creación del control merecerá la pena, ya que obtendremos como resultado un componente hecho a medida, que se adapta a nuestros requerimientos, y no al revés.

Manos a la obra. ¿Qué control queremos crear?

Supongamos que nos gustaría tener en nuestro formulario un control consistente en una casilla de verificación, al estilo del CheckBox tradicional, pero en lugar de mostrar la casilla como un recuadro, lo hiciera como un rombo.

Partiendo de esta idea base, comencemos nuestra singladura por el fascinante mundo de la creación de controles.

El enfoque del caso a desarrollar

Podríamos basar la explicación del artículo en dos bloques secuenciales: en el primero describiríamos la escritura del código para el control, y a continuación, en el segundo, su modo de utilización desde un formulario. Como autor del artículo este enfoque me resulta más fácil, pero creo que de esta forma perdemos la perspectiva de un importante factor: las sucesivas fases de refinamiento por las que pasa el desarrollo del control, y que nos pueden dar una mejor idea de su proceso evolutivo.

Por este motivo, abordaremos la creación del ejemplo mostrando gradual y alternativamente una parte del desarrollo del control, y cómo este va mostrándose en el lado cliente, al ser usado por un formulario.

Durante el transcurso de todo este proceso, hemos establecido varios hitos o fases de refinamiento, que consideramos ciclos independientes en el periodo de creación del

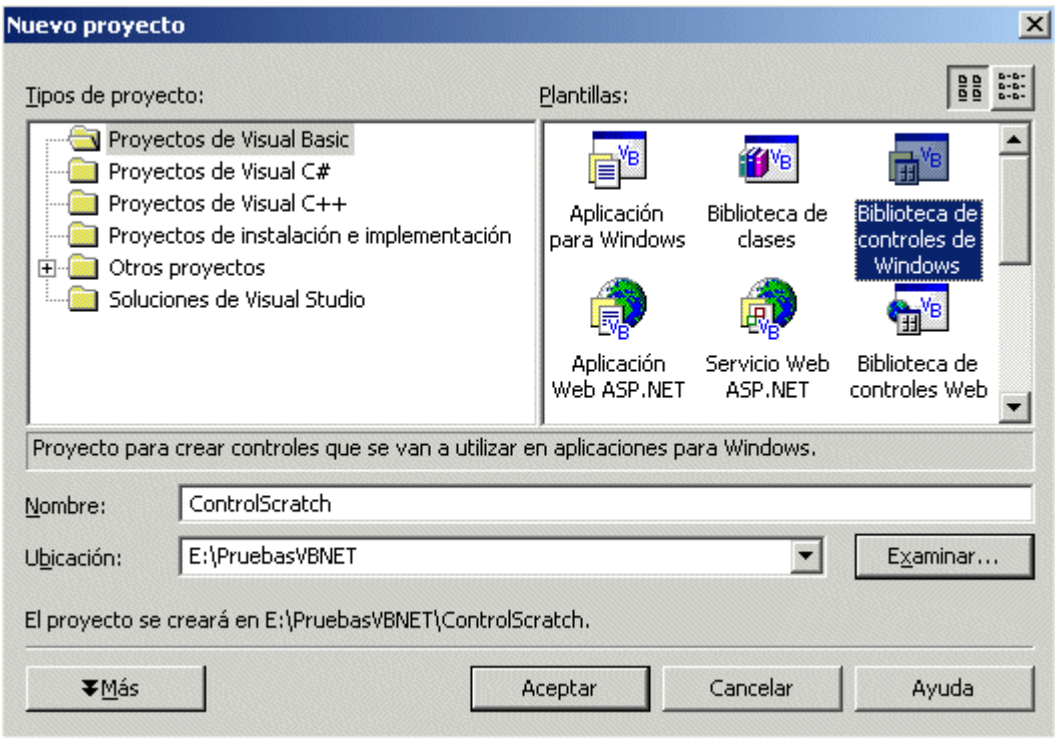
control. No pretenden establecer ninguna norma, sino que se proponen como guía al lector para ayudarle en el proceso de escritura de sus propios controles.

A lo largo del artículo, cada vez que uno de estos ciclos o fases de refinamiento sea completado, se le indicará oportunamente al lector para que tenga constancia del estado en el que se encuentra en ese momento el desarrollo del control.

Para facilitar el seguimiento de toda la exposición, se proporcionan como materiales adjuntos al artículo, la solución desarrollada en VS.NET conteniendo los proyectos del control y la aplicación Windows de pruebas, así como varios archivos de texto, con el código fuente correspondiente a cada una de las fases de refinamiento.

Comenzamos el desarrollo. Creación del proyecto

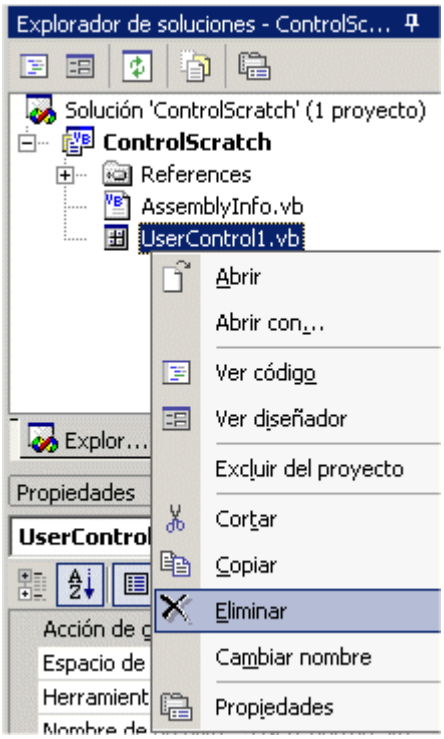
Para empezar a trabajar, en primer lugar abriremos el entorno de desarrollo de Visual Studio .NET, y crearemos un proyecto de tipo Biblioteca de controles Windows, al que daremos el nombre ControlScratch, como vemos en la siguiente figura.



Creación de un proyecto de tipo Biblioteca de controles

Un proyecto de estas características, asume por defecto que vamos a crear un control basado en una plantilla base de controles o UserControl, por lo que añade un elemento de este tipo con el nombre UserControl1, y nos posiciona en el mismo. Dado que esta no va a ser nuestra técnica de trabajo en el presente ejemplo, haremos clic

sobre este elemento en el Explorador de soluciones, y pulsaremos la tecla [Supr], eliminándolo del proyecto, como vemos en la siguiente figura.



Eliminar el diseñador UserControl

La clase del control y la clase Control, muchas cosas en común

A continuación mediante la opción de menú *Proyecto + Agregar clase,* debemos añadir a nuestro proyecto una clase, que será la que utilicemos para escribir el código de nuestro control. A esta clase le daremos el nombre CheckRomboBox.

Como hemos indicado anteriormente, vamos a crear un control Windows partiendo de cero, pero esto no significa que tengamos que desarrollar absolutamente todo el trabajo de creación del control, tarea de titanes donde las haya; aunque si hay algún valiente que se atreva, ¡ánimo!.

En .NET Framework la situación es más fácil de lo que parece. Los controles Windows heredan de la clase Control, situada en el espacio de nombres System.Windows.Forms; con ello obtienen la funcionalidad base o cimientos comunes a todos los controles, independientemente de la naturaleza o finalidad con la que vayan a ser destinados. Entre estos aspectos se encuentran las propiedades referentes a colores, tamaño, ubicación, acople, anclaje, etc.; los métodos para mostrar, proporcionar foco, etc.; y los eventos para detectar la pulsación del ratón, teclado, captura / liberación del foco, etc.

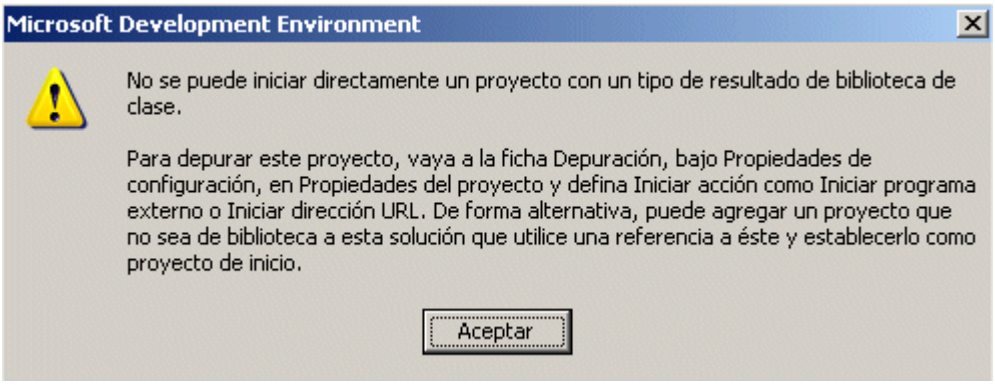
Por tanto, lo que nosotros debemos hacer tras crear nuestra clase es heredar también de Control, como vemos en el siguiente código fuente.

```
**Declaración de la clase del control personalizado**
Public Class CheckRomboBox
   Inherits Control


End Class
```

A partir de este momento, nuestro control también tendrá todas las características del modelo básico de controles.

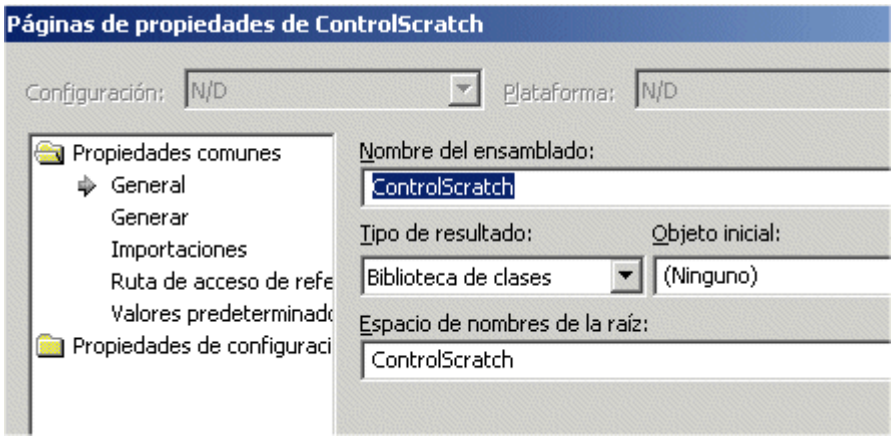
Creando el banco de pruebas para el control

Si en este momento compilamos el proyecto e intentamos ejecutar nuestro control obtendremos el mensaje de advertencia de la siguiente figura.



No es posible ejecutar directamente el proyecto del control

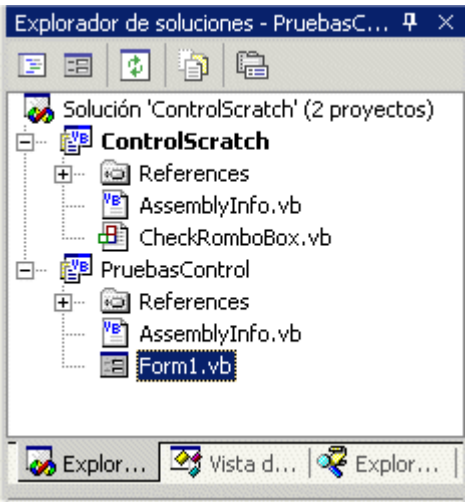
Como vemos, no es posible ejecutar directamente un proyecto de tipo biblioteca de clases, y antes de que el lector replique diciendo que ha creado un proyecto de biblioteca de controles y no de clases, debemos aclarar que, internamente, este tipo de proyecto es una biblioteca de clases; para comprobarlo tan sólo necesita hacer clic en el nombre del proyecto del Explorador de soluciones, y seleccionar la opción Propiedades del menú contextual; en la ventana Propiedades observe que el elemento Tipo de resultado tiene el valor Biblioteca de clases, como muestra la siguiente figura.



El proyecto del control es internamente una Biblioteca de clases

Lo que necesitamos para probar el control es un proyecto de tipo *Aplicación para Windows,* en el que podemos utilizar un formulario que realice las funciones de contenedor (host) del control.

Si bien podemos utilizar un proyecto totalmente separado del control, es mucho más cómodo añadir un proyecto a la misma solución que contiene el control –sobre la que estamos trabajando-; para ello seleccionaremos la opción de menú de VS.NET *Archivo + Agregar proyecto + Nuevo proyecto*, seleccionaremos como plantilla de proyecto Aplicación para Windows, y daremos el nombre PruebasControl al nuevo proyecto. La siguiente figura muestra cómo quedaría estructurada nuestra solución multiproyecto.



Agregando un proyecto de pruebas para el control

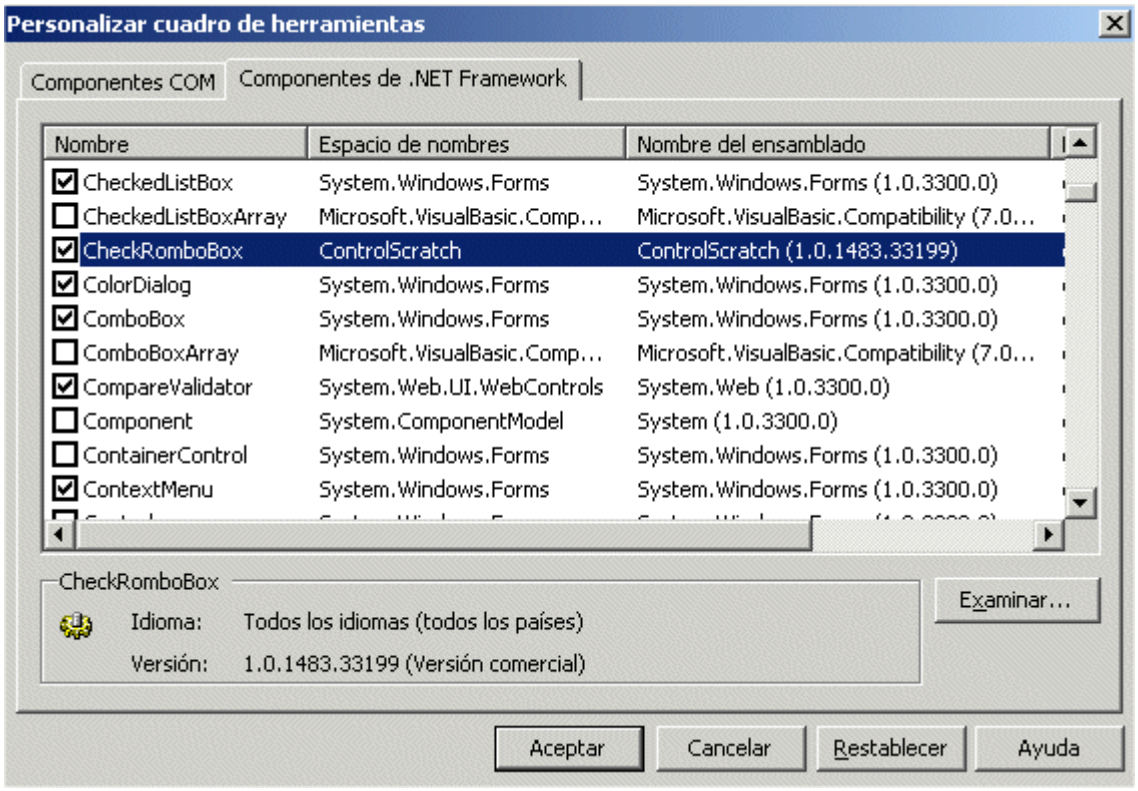
A continuación, en el Explorador de soluciones haremos clic derecho sobre el nuevo proyecto, seleccionando la opción *Establecer como proyecto de inicio*, por lo que a partir de ahora, cuando comencemos la ejecución, será este proyecto el que se inicie mostrando su formulario.

Usando el control desde el formulario

Todavía nos queda un paso más para poder usar nuestro control en el formulario, ya que a pesar de que los proyectos del control y de aplicación Windows se encuentran ambos en la misma solución, no hay ningún tipo de referencia que permita al formulario acceder al control, cosa que vamos a solucionar ahora mismo.

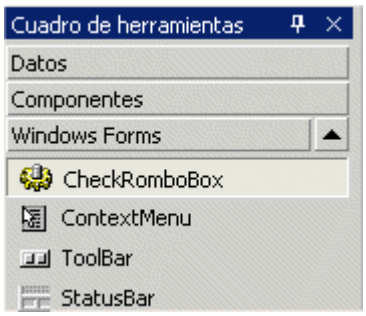
Dado que el control es un componente visual, lo normal es que deseemos tenerlo disponible -al igual que el resto de controles- en el Cuadro de herramientas del IDE de VS.NET; para ello, situados en el diseñador del formulario, haremos clic derecho sobre el Cuadro de herramientas, seleccionando la opción de menú contextual *Personalizar cuadro de herramientas* o *Agregar y quitar elementos*, dependiendo de la versión de Visual Studio que tengamos. Se abrirá el cuadro de diálogo para personalizar el cuadro de herramientas, en el que podremos añadir y quitar los controles que componen nuestra paleta de diseño.

Pulsando la pestaña *Componentes de .NET Framework*, haremos clic en el botón Examinar y nos desplazaremos hasta el directorio \bin correspondiente al proyecto del control. A continuación seleccionaremos el archivo ControlScratch.dll (si no hemos compilado el proyecto del control debemos hacerlo para obtener la librería que contiene el control compilado) y pulsaremos Aceptar; seguidamente volveremos al cuadro de diálogo que contiene la lista de controles, en la que ordenados alfabéticamente podremos ver nuestro control como muestra la siguiente figura.



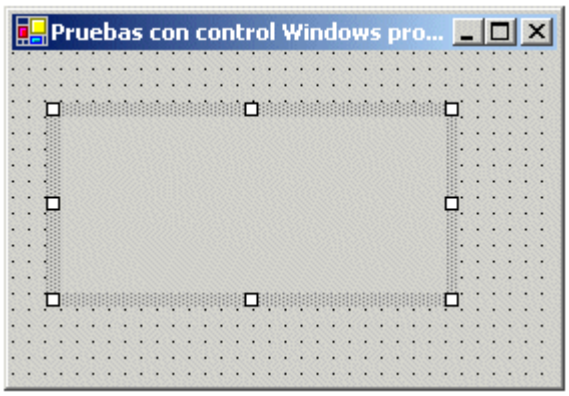
Agregar el control al Cuadro de herramientas

Para hacer que el control se agregue al Cuadro de herramientas, marcaremos la casilla que se encuentra a su izquierda y aceptaremos esta ventana.



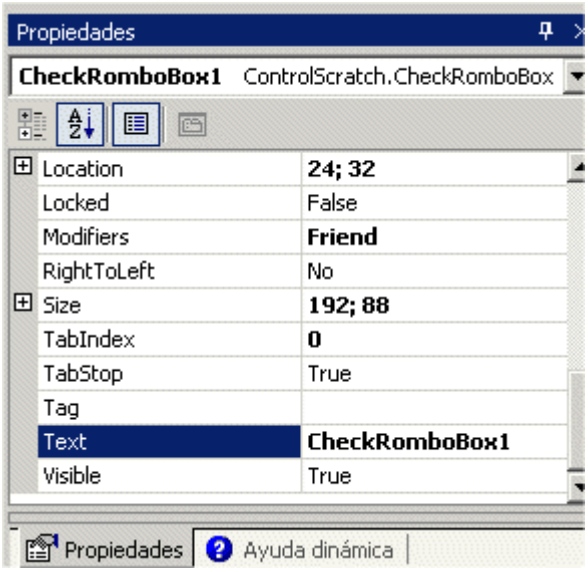
Cuadro de herramientas reflejando el control personalizado

Y ahora, la prueba definitiva para verificar que ya podemos trabajar con el control: seleccionándolo del Cuadro de herramientas, lo arrastraremos y dibujaremos sobre el formulario al igual que hacemos con cualquiera de los controles típicos de Windows. En el diseñador de la ventana observaremos cómo nuestro control, aunque todavía vacío de contenido, ocupa el área del formulario que le hayamos asignado.



Formulario conteniendo el control propio

A pesar de que apenas hemos comenzado a escribir nuestro control, una vez situado en el formulario, vemos como la ventana Propiedades ya refleja un buen número de las mismas, causado ello por heredar de la clase Control.



Propiedades heredadas del control

Ahora es cuando comienza el trabajo divertido: escribir código.

El rombo para la casilla de marcado

Ya que el motivo del control es que muestre la casilla de verificación en forma romboide, ¿qué mejor punto de partida para la clase del control que comenzar dibujando dicha casilla?.

Dentro de un control, cualquier operación que implique el dibujo de los elementos del control debe ser realizada reemplazando el método OnPaint, que será llamado por el sistema cada vez que el control necesite ser dibujado. El código a escribir será el que mostramos a continuación.

```
**Método para operaciones de dibujo sobre el control**
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 punto central del rombo
    Dim nCentro As Integer = Me.Height / 2
    ' calcular punto superior del rombo
    Dim nSuperior As Integer = nCentro - 9
    ' calcular punto inferior del rombo
    Dim nInferior As Integer = nCentro + 9

    ' calcular array de puntos para dibujar el rombo exterior
    Dim aPuntos(4) As Point
```

```
    aPuntos(0) = New Point(1, nCentro)
    aPuntos(1) = New Point(10, nSuperior)
    aPuntos(2) = New Point(19, nCentro)
    aPuntos(3) = New Point(10, nInferior)
    aPuntos(4) = New Point(1, nCentro)

    ' obtener un objeto para dibujar sobre el control
    Dim oGraphics As Graphics = e.Graphics

    ' dibujar el rombo exterior
    oGraphics.DrawPolygon(New Pen(Color.Black, 1), _
       aPuntos)

    ' liberar los recursos gráficos
    oGraphics.Dispose()

End Sub
```

Antes de pasar a comentar el anterior código, es preciso aclarar que para efectuar cualquier operación que implique el dibujo de elementos relacionados con el interfaz gráfico de Windows, necesitaremos unos conocimientos previos, aunque sean básicos, relacionados con GDI+, el subsistema gráfico de .NET Framework. Una explicación de la programación gráfica en la plataforma .NET queda fuera del ámbito de este artículo, por lo que recomendamos al lector que consulte la documentación que a tal efecto se encuentra en la ayuda de Visual Studio .NET.

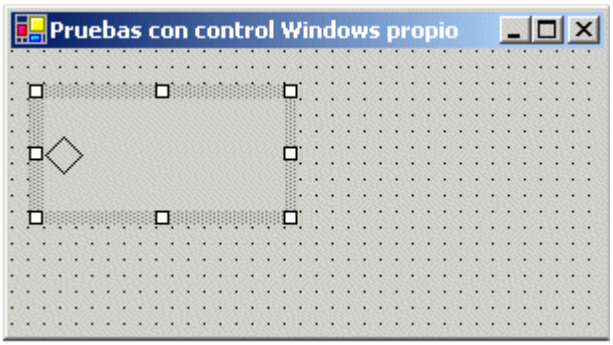
Ahora pasemos al método OnPaint. En primer lugar, y puesto que lo estamos reemplazando, debemos llamar a la implementación de este mismo método existente en su clase padre mediante la palabra clave MyBase (esta es la norma a seguir siempre que reemplacemos cualquiera de los métodos de este tipo), y pasar como parámetro el argumento e, de tipo PaintEventArgs que estamos recibiendo en este método, y que contiene la información relativa a las operaciones de dibujo para este evento. De esta forma, nos aseguramos que las operaciones por defecto se van a realizar, añadiendo después nuestro código personalizado.

A continuación calculamos, usando la altura (propiedad Height) del control, los valores clave que nos van a servir como referencia para establecer los puntos de dibujo. Después, y partiendo de los anteriores valores, creamos un array de objetos Point, que contendrán las coordenadas para dibujar el rombo. Un objeto Point está compuesto por dos valores Integer, que representan las coordenadas *x* e *y* en el plano de dibujo. El conjunto de coordenadas que acabamos de obtener nos servirán para situar el rombo en el control, alineado a la izquierda horizontalmente, y centrado verticalmente.

En el siguiente paso obtenemos del parámetro *e* del método un objeto Graphics, que contiene los métodos para dibujar los elementos gráficos del control; en este caso usaremos el método DrawPolygon, que dibuja el rombo pasándole como parámetros

un objeto Pen, y el array de objetos Point. El objeto Pen representa la herramienta de dibujo o pincel, y al instanciarlo debemos asignarle un color mediante la estructura Color de la plataforma .NET, y un grosor para el trazo mediante un valor Integer. Finalmente, liberaremos los recursos gráficos que hemos utilizado llamando al método Graphics.Dispose.

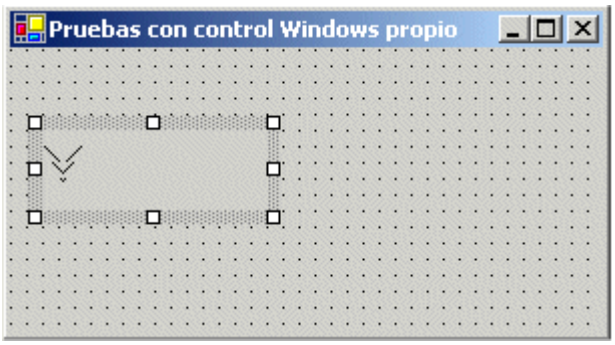
Cada vez que hagamos un cambio en el código del control, y antes de proceder a probarlo en el formulario, debemos compilar el proyecto del control o bien la solución entera. Una vez compilado, pasaremos al diseñador del formulario y situaremos sobre él una copia del control, que debería tener un aspecto similar a la siguiente figura.



Control propio mostrando el rombo para la casilla de marcado

Cambiando el tamaño del control en tiempo de diseño

Después de haber situado el control en el formulario, si lo redimensionamos, observaremos que las coordenadas de posición de la casilla no se actualizan con respecto al nuevo tamaño del control, por lo que al dibujarse el rombo, posiblemente lo haga de manera irregular, como vemos en la siguiente figura.



Problemas de visualización al redimensionar el control

En este tipo de situaciones, lo que necesitamos es detectar cuándo se está cambiando el tamaño del control, para que se ejecute adecuadamente el método OnPaint, y que

dibuje el control en los puntos adecuados. El método OnResize de la clase Control es el que deberemos utilizar, reemplazándolo y añadiendo nuestro propio código.

No es conveniente sin embargo, hacer desde OnResize una llamada directa a OnPaint, sino utilizar el método Invalidate, el cual, al ser ejecutado, invalida el área gráfica del control, y provoca internamente una nueva llamada a OnPaint, que se encarga de dibujar el control. Veámoslo en el siguiente código fuente.

```
**Método para redimensión del control**
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
   ' llamamos a la versión de este método existente
   ' en la clase base
   MyBase.OnResize(e)

   ' marcamos el área gráfica del control
   ' como no válida, para forzar un repintado
   Me.Invalidate()
End Sub
```

Dibujando el resto de elementos de la casilla

Una vez que hemos conseguido dibujar el rombo exterior y mantenerlo estable al redimensionar el control, debemos rellenar su interior de color blanco, al igual que ocurre con el CheckBox estándar. El cálculo de las coordenadas para esta zona del control será muy similar al anterior, debiendo tener en cuenta que la superficie a dibujar es menor.

Otro detalle importante radica en que no vamos a trazar el contorno de una figura, sino a rellenarla de un color sólido; por ello, y a diferencia del dibujo anterior, ahora utilizaremos el método Graphics.FillPolygon, pasándole un objeto SolidBrush con el color a aplicar y el array de coordenadas Point. El objeto SolidBrush representa a la herramienta de dibujo utilizada en este caso, que será un pincel grueso (brocha).

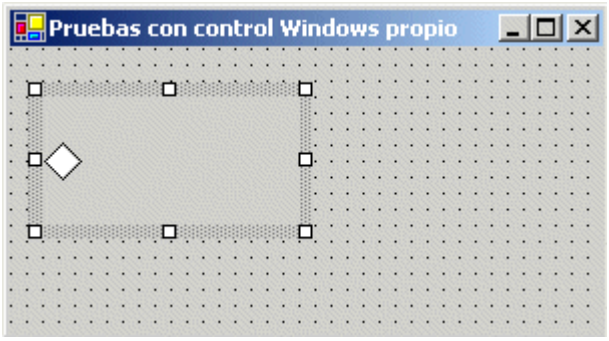
Estas operaciones las codificaremos en el método OnPaint a continuación del dibujo del rombo exterior, siendo el código a añadir el siguiente.

```
**Dibujo del interior del rombo**
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
   '....
   '....
   ' calcular array de puntos para dibujar el rombo interior
   aPuntos(0) = New Point(2, nCentro)
   aPuntos(1) = New Point(10, nSuperior + 1)
   aPuntos(2) = New Point(18, nCentro)
```

```
    aPuntos(3) = New Point(10, nInferior - 1)
    aPuntos(4) = New Point(2, nCentro)

    ' dibujar-rellenar el rombo interior
    oGraphics.FillPolygon(New SolidBrush(Color.White), aPuntos)

    ' liberar los recursos gráficos
    oGraphics.Dispose()

End Sub
```

Observe el lector que la llamada a Graphics.Dispose la hemos trasladado al final; esto es necesario ya que mientras hagamos uso de recursos gráficos no podremos liberar este objeto para que sea recuperado por el recolector de memoria del CLR.

Al volver a compilar, el control que habíamos situado anteriormente en el diseñador del formulario se actualizará, mostrando la parte interior del rombo con el nuevo color.



Control mostrando el dibujo externo e interno del rombo

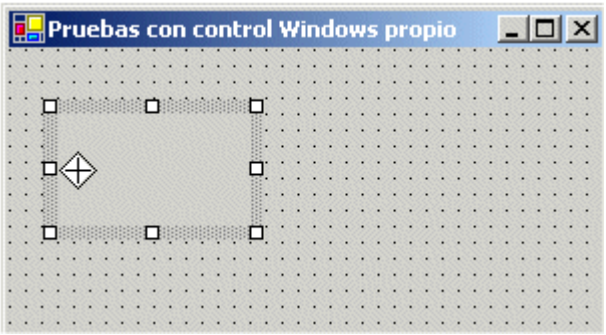
Para terminar con la casilla de marcado, ampliaremos el contenido del método OnPaint, escribiendo el código que se ocupe de dibujar el signo de marcado sobre el rombo. Este signo estará compuesto por dos líneas cruzadas dentro del rombo.

Al igual que en los casos anteriores, el objeto Graphics nos provee del método DrawLine, que usaremos para dibujar una línea en modo gráfico, pasándole como parámetro un objeto Pen y dos objetos Point, delimitadores estos del comienzo y final de línea. Los cálculos de posición de las líneas los haremos usando los valores de referencia calculados al comienzo de OnPaint. Las siguientes líneas de código muestran esta parte del método.

```
**Dibujo de la marca para la casilla del control**
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    '....
```

```
    '....
    ' dibujar el signo de marcado de la casilla:
    ' línea vertical
    oGraphics.DrawLine(New Pen(Color.Black, 1), _
       New Point(10, nSuperior + 3), _
       New Point(10, nInferior - 3))

    ' línea horizontal
    oGraphics.DrawLine(New Pen(Color.Black, 1), _
       New Point(4, nCentro), _
       New Point(16, nCentro))

    ' liberar los recursos gráficos
    oGraphics.Dispose()

End Sub
```

El resultado de este nuevo elemento se muestra en la siguiente figura.



Control mostrando la marca de verificación

El título asociado al control

El texto que se sitúa junto a la casilla de verificación lo dibujaremos añadiendo el siguiente código al método OnPaint.

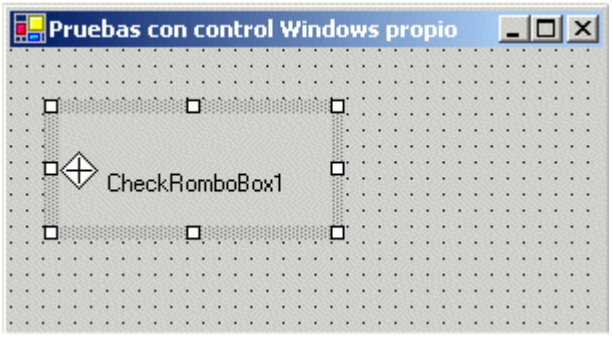
```
**Dibujo del título del control**
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    '....
    '....
    ' dibujar el título del control
    oGraphics.DrawString(Me.Text, Me.Font, _
       New SolidBrush(Me.ForeColor), 22, nCentro)
```

```
    ' liberar los recursos gráficos
    oGraphics.Dispose()

End Sub
```

Por el hecho de heredar de Control, hay valores para dibujar el texto que obtenemos directamente de las propiedades de la clase base. Estas propiedades son las siguientes.

- **Text**. Título del control.
- **Font**. Típo de letra del texto.
- **ForeColor**. Color de primer plano para el texto.

Para plasmar el texto en el control, tal y como acabamos de ver en el anterior fuente, usaremos el método Graphics.DrawString, pasándole las propiedades antes mencionadas, y los valores que representan la coordenada a partir de la que se dibujara el título. La siguiente figura muestra el control, incorporando ya su correspondiente título.



Control mostrando el título

La alineación del texto con respecto a la casilla no ha quedado todo lo ajustada que nos hubiera gustado, por lo que vamos a intentar solucionar este pequeño inconveniente añadiendo algunas líneas de código adicionales a OnPaint.

Instanciaremos un objeto StringFormat, gracias al cual podremos manipular el formato del texto a dibujar, en concreto su alineación, a través de las propiedades Alignment y LineAlignment; el valor que asignaremos será Center, perteneciente a la enumeración StringAlignment, y nos permitirá centrar el texto al ser dibujado.

En cuanto a DrawString, usaremos una versión distinta de este método, en la que en vez de especificar la ubicación de dibujo mediante dos valores de posición, usaremos un objeto RectangleF con el área del control en la que se dibujará el texto, y el objeto StringFormat para que el texto sea pintado en el centro del rectángulo. Veamos ahora el código para dibujar el texto con estas modificaciones.

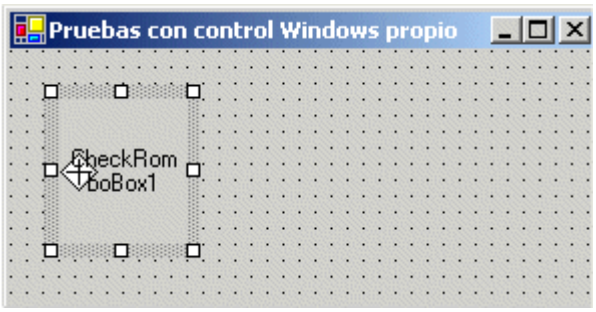
```
**Dibujo del título centrado en el control**
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
   '....
   '....
   ' crear un objeto para formato de cadenas
   ' y configurarlo para centrar la cadena a utilizar
   Dim oStrFormat As New StringFormat
   oStrFormat.LineAlignment = StringAlignment.Center
   oStrFormat.Alignment = StringAlignment.Center

   ' instanciar un objeto RectangleF con la zona
   ' del control en la que vamos a dibujar el texto
   Dim oRectTexto As RectangleF
   oRectTexto = New RectangleF(21, 0, Me.Width - 21, Me.Height)

   ' dibujar el título del control en el rectángulo
   ' que acabamos de crear y centrado
   oGraphics.DrawString(Me.Text, _
      Me.Font, _
      New SolidBrush(Me.ForeColor), _
      oRectTexto, _
      oStrFormat)

   ' liberar los recursos gráficos
   oGraphics.Dispose()

End Sub
```

Observemos que a la hora de definir el rectángulo, no utilizamos toda la superficie del control, ya que excluimos el área ocupada por la casilla de verificación.

Si usáramos toda la zona del control creando el rectángulo de la siguiente manera.

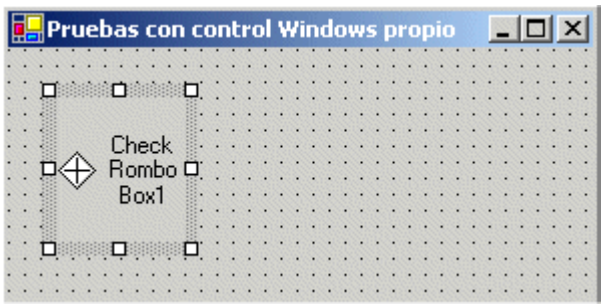
```
**Creación del rectángulo para dibujar el título en todo el área del control**
oRectTexto = New RectangleF(0, 0, Me.Width, Me.Height)
```

Podríamos encontrarnos con un resultado como el que vemos a continuación cuando el tamaño del control fuera muy pequeño.



Control con el título superpuesto sobre la casilla

Sin embargo, dejando fuera del área de dibujo del texto a la casilla, siempre nos aseguramos que al redimensionar el control a un pequeño tamaño, el texto no la tapará. La siguiente figura muestra el control resultante tras aplicar esta técnica.



Control con el título situado junto a la casilla

Por otro lado, la diferencia entre la clase Rectangle y RectangleF reside en que esta última utiliza valores Single, por lo que podemos emplear precisión decimal. Aunque en nuestro caso no necesitamos este tipo de precisión, debemos emplear esta clase, ya que la versión sobrecargada de DrawString que estamos usando ahora requiere un tipo RectangleF.

Organizando las operaciones de dibujo en la clase

Repasando el código resultante que hemos escrito en OnPaint, y de cara a una mejor organización de las futuras operaciones que tendremos que codificar, vamos a crear un método en la clase para cada uno de los elementos a dibujar del control: rombo exterior, interior, marca y título. Igualmente, las variables usadas para almacenar los puntos superior, inferior y central las declararemos ahora –como campos- en la zona de declaraciones de la clase. El siguiente código fuente refleja estos cambios sobre la clase.

Reorganización del código para el dibujo del control

```
' campos
Private nCentro As Integer
Private nSuperior As Integer
Private nInferior As Integer

'---------------------------------
' reemplazo de métodos generadores de eventos
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 punto central del rombo
   nCentro = Me.Height / 2
   ' calcular punto superior del rombo
   nSuperior = nCentro - 9
   ' calcular punto inferior del rombo
   nInferior = nCentro + 9

   ' obtener un objeto para dibujar sobre el control
   Dim oGraphics As Graphics = e.Graphics

   Me.DibujarRomboExterior(oGraphics)
   Me.DibujarRomboInterior(oGraphics)
   Me.DibujarMarca(oGraphics)
   Me.DibujarTexto(oGraphics)

   ' liberar los recursos gráficos
   oGraphics.Dispose()

End Sub

'---------------------------------
' métodos de dibujo del control
Private Sub DibujarRomboExterior(ByVal oGraphics As Graphics)
   ' calcular array de puntos para dibujar el rombo exterior
   Dim aPuntos(4) As Point
   aPuntos(0) = New Point(1, nCentro)
   aPuntos(1) = New Point(10, nSuperior)
   aPuntos(2) = New Point(19, nCentro)
   aPuntos(3) = New Point(10, nInferior)
   aPuntos(4) = New Point(1, nCentro)

   ' dibujar el rombo exterior
   oGraphics.DrawPolygon(New Pen(Color.Black, 1), _
      aPuntos)
End Sub
```

```
'---------------------------------
Private Sub DibujarRomboInterior(ByVal oGraphics As Graphics)
   ' calcular array de puntos para dibujar el rombo interior
   Dim aPuntos(4) As Point
   aPuntos(0) = New Point(2, nCentro)
   aPuntos(1) = New Point(10, nSuperior + 1)
   aPuntos(2) = New Point(18, nCentro)
   aPuntos(3) = New Point(10, nInferior - 1)
   aPuntos(4) = New Point(2, nCentro)

   ' dibujar-rellenar el rombo interior
   oGraphics.FillPolygon(New SolidBrush(Color.White), aPuntos)
End Sub

'---------------------------------
Private Sub DibujarMarca(ByVal oGraphics As Graphics)
   ' dibujar el signo de marcado de la casilla:
   ' línea vertical
   oGraphics.DrawLine(New Pen(Color.Black, 1), _
      New Point(10, nSuperior + 3), _
      New Point(10, nInferior - 3))

   ' línea horizontal
   oGraphics.DrawLine(New Pen(Color.Black, 1), _
      New Point(4, nCentro), _
      New Point(16, nCentro))
End Sub

'---------------------------------
Public Sub DibujarTexto(ByVal oGraphics As Graphics)
   ' crear un objeto para formato de cadenas
   ' y configurarlo para centrar la cadena a utilizar
   Dim oStrFormat As New StringFormat()
   oStrFormat.LineAlignment = StringAlignment.Center
   oStrFormat.Alignment = StringAlignment.Center

   ' instanciar un objeto RectangleF con la zona
   ' del control en la que vamos a dibujar el texto
   Dim oRectTexto As RectangleF
   oRectTexto = New RectangleF(21, 0, Me.Width - 21, Me.Height)

   ' dibujar el título del control en el rectángulo
   ' que acabamos de crear y centrado
   oGraphics.DrawString(Me.Text, _
      Me.Font, _
      New SolidBrush(Me.ForeColor), _
      oRectTexto, _
```

```
        oStrFormat)
End Sub
```

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

Controlando por código el marcado de la casilla

Si compilamos el control y ejecutamos el ejemplo, notaremos que al hacer clic sobre el rombo no se marca o desmarca la casilla, tal y como sucede con un CheckBox típico. Esto es lógico, pues a pesar de haber escrito el código que dibuja la marca, no controlamos cuándo ocurre este suceso para proceder a actualizar el estado de la casilla. Adicionalmente, tampoco contamos con una propiedad que nos permita asignar directamente dicho estado por código durante la ejecución, o en la ventana de propiedades en tiempo de diseño. Vamos por lo tanto a añadir dicha funcionalidad a la clase.

En lo que concierne a la creación de la propiedad, definiremos un campo privado en la clase con el nombre mbMarcado, de tipo Boolean, que almacene el valor True cuando la casilla esté marcada, y False cuando esté vacía.

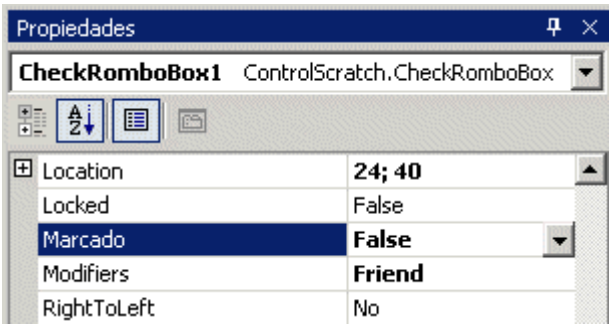
Después escribiremos una propiedad pública con el nombre Marcado, que permita asignar / recuperar el valor del campo mbMarcado desde el formulario que utilice el control. En el bloque Set de la propiedad, según el valor que tenga mbMarcado, llamaremos al método que dibuja la marca o el rombo interior. Veamos el código fuente correspondiente a estos pasos.

```
**Propiedad para guardar el estado de la casilla del control**
Private mbMarcado As Boolean

Public Property Marcado() As Boolean
  Get
    Return mbMarcado
  End Get
  Set(ByVal Value As Boolean)
    mbMarcado = Value

    Dim oGraphics As Graphics = Me.CreateGraphics()

    If mbMarcado Then
      Me.DibujarMarca(oGraphics)
    Else
      Me.DibujarRomboInterior(oGraphics)
    End If
```

```
    End Set
End Property
```

Nótese que debido a que el método DibujarMarca requiere que pasemos un objeto Graphics, en este caso -y a diferencia de OnPaint en donde lo obteníamos del parámetro e- vamos a crearlo explícitamente usando el método CreateGraphics que heredamos de la clase base.

Tras compilar la clase con estos nuevos cambios, comprobaremos cómo al seleccionar el control en diseño, la ventana Propiedades tiene la propiedad Marcado, que nos permite cambiar el estado de la marca, como vemos en la siguiente figura.



Ventana de propiedades con la nueva propiedad Marcado

También podemos manejar esta propiedad desde el código del formulario. Si por ejemplo, añadimos un botón y escribimos la siguiente línea, cambiaremos, alternativamente, el estado de la casilla.

```
**Cambio del estado de la marca del control**
Me.CheckRomboBox1.Marcado = Not Me.CheckRomboBox1.Marcado
```

Gracias al operador Not, cambiamos el valor actual de una propiedad lógica en una sola línea de código de forma muy simple.

Puliendo el marcado inicial de la casilla

Existe otro pequeño, pero importante detalle acerca del dibujo de la marca en la casilla del control, ya que cada vez que añadimos al formulario un nuevo control, este aparece siempre con el rombo marcado, a pesar de que la propiedad Marcado tenga el valor False.

Este comportamiento se debe a que al ser llamado el método DibujarMarca, no se verifica el valor del campo mbMarcado, que nos informa del estado que debería tener la casilla.

Para solventar este inconveniente vamos a utilizar este campo de la clase, de modo que sólo marcaremos el rombo cuando su valor sea True, como vemos en el siguiente fuente.

```
**Dibujo de la marca en el control previa comprobación de su estado**
Private Overloads Sub DibujarMarca(ByVal oGraphics As Graphics)
   If mbMarcado Then
      ' dibujar el signo de marcado de la casilla:
      ' línea vertical
      oGraphics.DrawLine(New Pen(Color.Black, 1), _
         New Point(10, nSuperior + 3), _
         New Point(10, nInferior - 3))

      ' línea horizontal
      oGraphics.DrawLine(New Pen(Color.Black, 1), _
         New Point(4, nCentro), _
         New Point(16, nCentro))
   End If
End Sub
```

Ahora nos queda permitir al usuario interaccionar con el control, para que él también pueda marcar y desmarcar la casilla, pero esto lo veremos en el siguiente apartado.

Marcando la casilla con el ratón

Hasta el momento, la única forma que tenemos de marcar o desmarcar el control consiste en asignar valor a su propiedad Marcado, tal y como acabamos de ver en el apartado anterior. Pero como es natural, también necesitamos esta funcionalidad al hacer clic sobre el control en tiempo de ejecución.

Valiéndonos del método OnClick, que se produce cada vez que se pulsa con el ratón en el área del control, vamos a reemplazarlo y añadir el código mostrado a continuación

```
**Método producido al pulsar el control con el ratón**
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
   MyBase.OnClick(e)

   mbMarcado = Not mbMarcado
```

```
    Dim oGraphics As Graphics = Me.CreateGraphics()

    If mbMarcado Then
        Me.DibujarMarca(oGraphics)
    Else
        Me.DibujarRomboInterior(oGraphics)
    End If
End Sub
```

En este código cambiamos el valor del campo que indica el estado de la casilla, y según esto, dibujamos la marca o la casilla interior vacía.

Actualmente hay más de un lugar en el código de la clase que tiene un proceso con esta lógica de funcionamiento, punto negativo en nuestro código a efectos de futuros mantenimientos. Por ello, vamos a centralizar estas operaciones en un único método, al que llamaremos tanto desde la propiedad Marcado como desde el método OnClick. A este nuevo método podemos llamarle DibujarMarca, sobrecargando al método que con este mismo nombre ya existe en la clase. El código afectado por estos cambios quedaría como vemos a continuación.

```
**Revisiones al código relacionado con la pulsación del control**
Private Overloads Sub DibujarMarca()
    Dim oGraphics As Graphics = Me.CreateGraphics()

    If mbMarcado Then
        Me.DibujarMarca(oGraphics)
    Else
        Me.DibujarRomboInterior(oGraphics)
    End If
End Sub

'---------------------------------
Public Property Marcado() As Boolean
    Get
        Return mbMarcado
    End Get
    Set(ByVal Value As Boolean)
        mbMarcado = Value
        Me.DibujarMarca()
    End Set
End Property

'---------------------------------
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
    MyBase.OnClick(e)
```

```
    mbMarcado = Not mbMarcado
    Me.DibujarMarca()

End Sub
```

Recuerde el lector que a la primera versión del método DibujarMarca tenemos que aplicarle también el modificador Overloads para que no se produzca un error de compilación.

La barra espaciadora también cambia el estado de la casilla

Cuando un control CheckBox estándar recibe el foco de entrada del programa, podemos cambiar alternativamente a marcado / desmarcado el estado de la casilla pulsando la barra espaciadora. Para lograr este comportamiento en nuestro control debemos reemplazar el método OnKeyDown, y añadir las siguientes líneas de código.

```
**Detectando la pulsación de la barra espaciadora**
Protected Overrides Sub OnKeyDown(ByVal e As
System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyDown(e)

    ' si la tecla pulsada es la barra espaciadora...
    If e.KeyCode = Keys.Space Then
        '...cambiar el valor de la variable
        ' que indica el estado de la casilla
        mbMarcado = Not mbMarcado
        ' y dibujar la marca o el rombo interior
        Me.DibujarMarca()
    End If

End Sub
```

OnKeyDown recibe un parámetro de tipo KeyEventArgs con toda la información necesaria acerca del evento de pulsación de la tecla. Para saber qué tecla se ha pulsado sólo tenemos que preguntar a la propiedad KeyCode, que nos devolverá un valor que se corresponde con uno de los miembros de la enumeración Keys de la plataforma .NET, en la que están catalogadas las teclas existentes. Si el valor obtenido es igual a Keys.Space –la tecla espacio-, cambiaremos el estado de la casilla a marcado / desmarcado según corresponda.

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

Permanezcan atentos a nuestra sintonía, continuará…

Y como merced del caminante tecnológico es hacer una parada en la senda de su investigación, buscar cálida posada en la que de viandas a su carnal presencia proveer, y descanso frugal a su cansada alma dar; con la licencia de vuesas mercedes, lo propio vamos a hacer hasta la próxima entrega de esta aventura sin par.