Crear interfaces gráficas

En el momento del retorno a la fuerza de la línea de comandos, es absolutamente legítimo plantearse la cuestión: «¿Para que queremos interfaces gráficas si la filosofía de PowerShell es sustituir a estas últimas?». En efecto, esto podría parecer paradójico a primera vista, pero pensándolo un poco, es fácil encontrar muy buenas razones que hacen pensar que las interfaces gráficas sean finalmente complementarias a los scripts.

Entre las buenas razones, podemos encontrarnos al menos las siguientes:

- Necesidad de proporcionar un script a los usuarios. Así, por la vía de la interfaz gráfica, los usuarios con poca experiencia en scripting pueden interactuar de forma sencilla.

- Facilitar la utilización de un script que requiere de muchos parámetros. Todos los scripts no tienen necesariamente la necesidad de ser ejecutados exclusivamente como tarea planificada. Además si se puede evitar la necesidad de conocer todos los parámetros de un script, ¿por qué privarse de una interfaz gráfica que presentara estos últimos en forma de casillas de verificación? Se podría muy bien imaginar un script iniciado sin parámetros que muestre una interfaz gráfica que solicite introducirlos. Este mismo script iniciado por línea de comandos con todos sus parámetros funcionaría como cualquier otro script PowerShell clásico. Una cosa no impide la otra, ¡todo lo contrario!

- Disponer de una interfaz gráfica flexible. Gracias al hecho de que PowerShell sea un lenguaje interpretado, es decir no compilado, todo el mundo puede tener acceso al código fuente. Por consiguiente, es fácil modificar el código para obtener la interfaz gráfica que responda exactamente a la necesidad del momento. Esto no es factible con instrumentos convencionales realizados en lenguajes compilados como C, C# o VB ya que generalmente no se dispone de fuentes.

- Hacer progresar a los usuarios en su conocimiento de PowerShell. Por experiencia, cuando se da una herramienta flexible a los usuarios un poco curiosos, estos terminan por apropiársela y mejorarla. Lo que refuerza claramente su conocimiento del lenguaje.

1. ¿Qué tecnología escoger?

De hecho, todo depende del nivel de funcionalidad esperado por la interfaz gráfica, de la plataforma en la que su script debe ejecutarse, y del tiempo de que disponga. Si usted simplemente desea asegurarse que una determinada tarea se ejecute siempre perfectamente, entonces el aspecto de la interfaz gráfica tendrá relativamente poca importancia. Un caso típico es cuando, como responsable del sistema, le proporciona a las personas que declaran las cuentas en Active Directory un script con interfaz gráfica que, por ejemplo, obliga a introducir en mayúsculas el nombre y crea el directorio home siempre en el lugar adecuado y con los permisos correctos. En este caso la tecnología Windows Forms aportada por el Framework .NET 2.0 es por lo general suficiente. En efecto los *Windows Forms* poseen el aspecto Windows. En él se encuentran botones, casillas de verificación, etiquetas, zonas de entrada de texto, resumiendo, lo típico.

Si por el contrario, necesita desarrollar un script que debe generar gráficos 2D o 3D, como por ejemplo para hacer el seguimiento del espacio de disco de los servidores de archivos, entonces el resultado final tendrá una gran importancia. En este caso, por lo general será preferible recurrir a la tecnología **WPF** (*Windows Presentation Foundation*). Al igual que si quiere realizar interfaces gráficas extravagantes, que se salgan de lo común; WMF será un excelente candidato.

Una ventaja que tiene WPF sobre su hermano Windows Forms para crear interfaces gráficas es que se puede describir estas últimas con una gramática basada en el lenguaje XML. Se trata del formato XAML. XAML, es un lenguaje descriptivo, facilita la creación de interfaces y aporta gran flexibilidad en el momento de la modificación de estas últimas; pero veremos esto posteriormente...

Ahora que usted sabe más sobre tecnologías, veremos cuáles son los requerimientos necesarios para su aplicación:

Windows Forms	WPF
Framework .NET 2.0 mínimo	Framework .NET 3.0 mínimo - 3.5 recomendado
PowerShell 1.0 y 2.0	PowerShell 2.0

A la vista de estos requisitos, usted podría preguntarse si un script basado en Windows Forms es más manejable que un script basado en WPF. Esto viene del hecho de que la pareja Framework .NET 2.0/PowerShell 1.0 es ahora casi un estándar en los PC de una empresa. Si está planeando hacer funcionar sus scripts utilizando WPF en las plataformas Windows 7 o Windows Server 2008 R2, no debe inquietarse ya que los requisitos previos están instalados por defecto. En cambio, siempre para WPF, en Vista será necesario instalar PowerShell v2 y en Windows XP como mínimo el Framework .NET 3.0 y PowerShell v2.

2. Windows Forms

a. Introducción a Windows Forms

Microsoft Windows Forms, también denominado Winform es el nombre que se le ha dado a las interfaces gráficas aportadas con el Framework .NET. La utilización de estos Winforms, basados en un conjunto de tipos disponibles en el assembly `System.Windows.Form`, permite crear interfaces al estilo Windows gracias a un acceso a los elementos gráficos nativos de Windows.

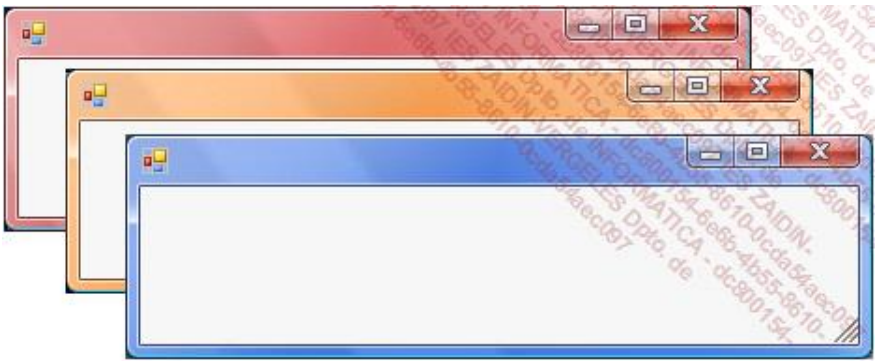
Para conocer los distintos tipos gráficos cuyo espacio de nombres es System.Windows.Form, puede volver a utilizar la función Get-TypeName que hemos desarrollado anteriormente. Utilizando el comando siguiente, podrá ver que existen más de un millar de tipos:

```
PS > (Get-TypeName System.Windows.Form).count
```

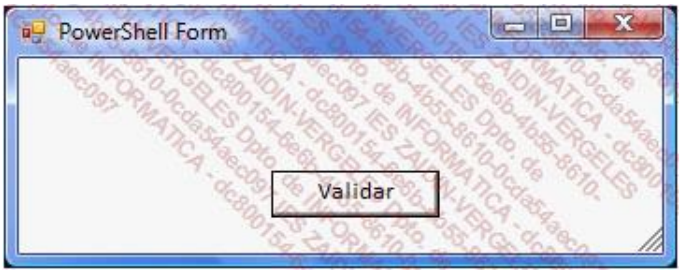
Sin embargo, tenga cuidado, ya que estos tipos no están cargados al inicio de PowerShell. Necesitará por lo tanto asegurarse de cargar el ensamblado `System.Windows.Forms` previamente con el comando:

```
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
```

El elemento básico en la construcción de una interfaz gráfica Windows se denomina un control; un control puede ser tanto un botón, un texto, un formulario, etc. Un formulario es una superficie visual parametrizable en la que podrá introducir otros componentes. Los formularios gozan de las mismas técnicas de desarrollo que las interfaces del sistema operativo Windows. Lo que permite a cada formulario gráfico creado, heredar las propiedades de visualización y del tema de su sistema operativo. Por ejemplo, los formularios desarrollados en Vista o Windows 7 se aprovechan del efecto de transparencia de las ventanas (véase la ilustración siguiente).



Windows Forms según los temas Windows 7



Formulario con botón

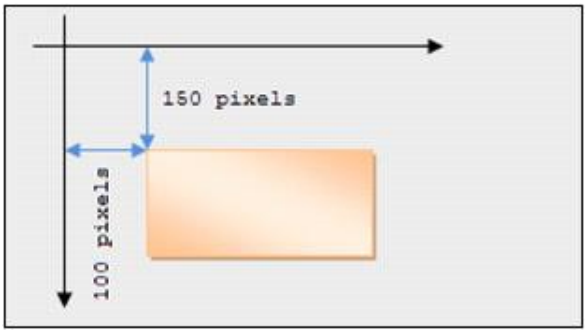
Cada control es un objeto, y por lo tanto, dispone de un determinado número de propiedades comunes que permiten ajustar la textura, la visibilidad, el tamaño, la posición, etc. La lista de las propiedades y métodos puede obtenerse aplicando el commandlet **`Get-Member`** al objeto.

El otro aspecto interesante de los Winforms, es la gestión de eventos (clic derecho, clic izquierdo del ratón, escribir por teclado, etc.). Existen decenas y permiten añadir un comportamiento a un control. Como por ejemplo, visualizar texto, cerrar un formulario. Todo control, se trate de un formulario o un elemento que lo componga, podrá ser sometido a uno o varios eventos. Y cada evento puede provocar una o varias acciones.

Puesto que PowerShell no dispone de un editor gráfico para la realización de los Winforms, el posicionamiento de los controles se hace de forma manual según dos constantes: el eje de ordenadas X y el eje de abscisas Y, con origen en la esquina superior izquierda de su contenedor y como unidad el píxel.

Ejemplo:

El posicionamiento de un control a 100 píxeles sobre el eje de abscisas y 150 sobre el eje de ordenadas, dará: location (100,150).



b. Creación de un formulario sencillo

Como acabamos de comentar, con el fin de utilizar los tipos gráficos disponibles en el Framework .NET, es indispensable cargar el ensamblado `System.Windows.Forms` que contiene todas las clases para permitir la creación de aplicaciones gráficas. La carga del ensamblado se hace de la manera siguiente:

```
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
```

Una vez estén disponibles los tipos gráficos, vamos a crear un formulario principal, sobre el cual podamos insertar componentes gráficos, menús, controles, ventanas de diálogo, etc.

```
# Creación del formulario principal
$form = New-Object Windows.Forms.Form
```

Observando los métodos y propiedades de nuestro objeto (¡¡¡más de 500!!!), vamos a buscar cómo personalizar nuestro formulario. Como por ejemplo añadirle un título utilizando la propiedad «Text», y asignándole un tamaño determinado:

```
#Mostrar un título
$form.Text = 'PowerShell Form'

# Dimensión del formulario
$form.Size = New-Object System.Drawing.Size(360,140)
```

Observe que la asignación de un tamaño a un formulario requiere asignar a la propiedad Size un objeto de tipo `System.Drawing.Size` al que atribuimos dos variables (anchura, altura).

Por defecto, el tamaño atribuido a un formulario es de 300 píxeles de anchura y 300 de altura.

Añadamos ahora un botón a nuestra interfaz creando un objeto de tipo `System.Windows.Forms.Button`:

```
# Creación del botón Salir
$boton_salir = New-Object System.Windows.Forms.Button
$boton_salir.Text = 'Salir'

# Posicionamiento del botón
$boton_salir.Location = New-Object System.Drawing.Size(135,60)

# Ajuste del tamaño
$boton_salir.Size = New-Object System.Drawing.Size(90,25)
```

Es necesario añadir nuestro botón al formulario principal, para ello utilizamos el método `Add` aplicado al miembro `Controls` del formulario.

```
$form.Controls.Add($boton_salir)
```

Después, para finalizar, debemos mostrar el formulario con el método `ShowDialog`:

```
$form.ShowDialog()
```

El método `Show` permite mostrar el formulario. Pero éste último desaparecerá inmediatamente, ya que ningún bucle de mensaje interno del sistema, está disponible para bloquear la ventana.

Utilizando el método ShowDialog, creará una ventana hijo del proceso PowerShell. Por contra, utilizando el método estático Run de la clase System.Windows.Forms.Application, creará esta vez una aplicación realmente independiente. Así, al cerrar la consola PowerShell, su formulario permanece. Lo que no ocurre con el método ShowDialog.

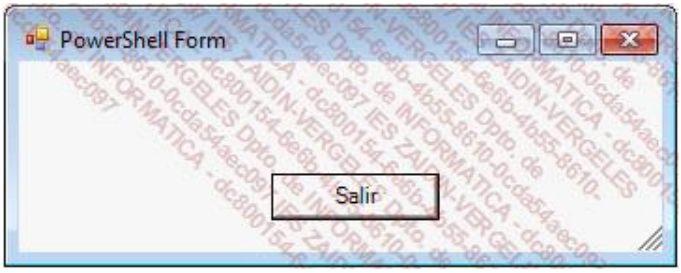
```
PS > $form = New-Object System.Windows.Form
PS > [System.Windows.Forms.Application]::Run($form)
```

Asociando las partes de código anteriores, obtenemos un script que crea una interfaz gráfica cuyo resultado se indica en la figura siguiente.

Script: Creación de una interfaz gráfica

```
#Form_1.ps1
#Creación de una interfaz gráfica

[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form
```

```
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,140)
$boton_salir = New-Object System.Windows.Forms.Button
$boton_salir.Text = 'Salir'
$boton_salir.Location = New-Object System.Drawing.Size(135,60)
$boton_salir.Size = New-Object System.Drawing.Size(90,25)
$form.Controls.Add($boton_salir)
$form.ShowDialog()
```



Gracias a este ejemplo, acabamos de crear una interfaz gráfica con PowerShell. Ahora debemos añadir otras funcionalidades a este formulario. Esto es lo que vamos a ver en el apartado siguiente.

c. Añadir funcionalidades a un formulario

Tras haber procedido a la creación de un formulario gráfico elemental, vamos ahora a progresar en la creación de las interfaces gráficas con la inserción de un menú así como la utilización de la gestión de eventos y del Timer.

Los eventos

Con el Framework .NET, cada componente gráfico puede reaccionar con uno o varios eventos, si está más o menos bien configurado. La adición de un acontecimiento a un componente, se efectúa aplicando el método Add_<nombre_del_evento>. Basta luego con incluir las acciones a emprender entre las llaves en el interior de los paréntesis de la metodología.

Ejemplo:

Evento clic izquierdo (`Add_Click`).

```
$<Nombre_del_boton>.Add_Click(
{
  # Bloque de instrucciones
})
```

Cuando utilice los eventos para hacer cambios gráficos en su interfaz, es posible que necesite actualizar la ventana para que las modificaciones se tengan en cuenta.

Ejemplo:

Adición de evento a un formulario.

En este ejemplo, vamos a basarnos en el formulario desarrollado en la sección anterior (creación de un formulario sencillo) y añadir un evento en el botón **Salir** para cerrar la ventana cuando pulsemos el botón izquierdo del ratón. Para ello utilizaremos el método `Close` definido en el evento `Add_Click` del botón **Salir**.

Script: Creación de una interfaz gráfica con el botón Salir activo.

```
#Form_2.ps1
#Creación de una interfaz gráfica con un boton Salir activo

[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,160)
$boton_salir = New-Object System.Windows.Forms.Button
$boton_salir.Text = 'Salir'
$boton_salir.Location = New-Object System.Drawing.Size(135,80)
$boton_salir.Size = New-Object System.Drawing.Size(90,25)

#  Añadir el evento 'clic izquierdo' al botón
$boton_salir.Add_Click(
{
  $form.Close()
})

$form.Controls.Add($boton_salir)
$form.ShowDialog()
```

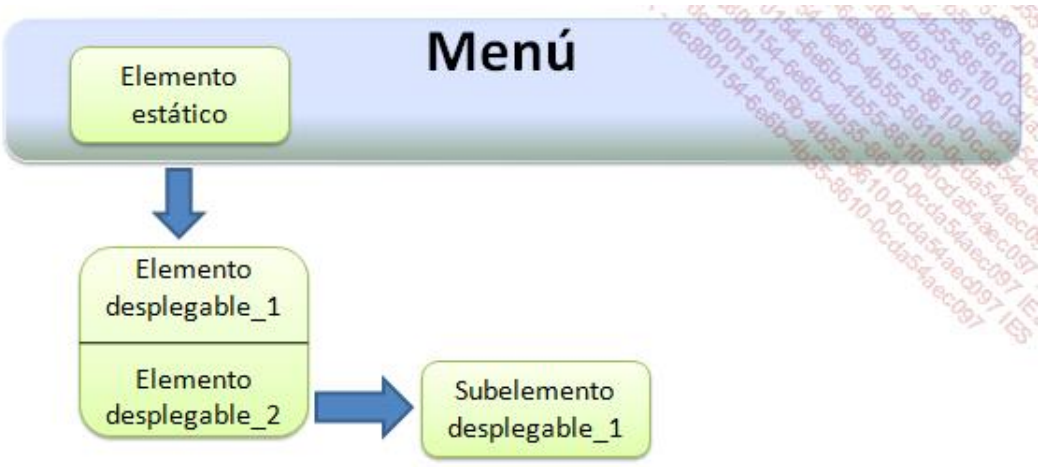
Resultado, la ventana se cierra haciendo un clic con el botón izquierdo en el botón **Salir**.

Los menús

Para reforzar aún un poco más una interfaz, es posible crear una barra de herramientas y de menús. Un componente de almacenamiento sirve para crear un árbol de botones a los que es también posible añadir eventos.

La creación de un menú se realiza con la instanciación de la clase MenuStrip (`System.Windows.Forms.MenuStrip`) que hizo su aparición a partir del Framework 2.0.

A este menú, pueden agregarse los componentes de tipo `System.Windows.Forms.ToolStripMenuItem`, pudiendo ellos también a su vez contener otros componentes y formar un árbol de botones.



La adición de un elemento estático en el menú requiere la llamada del método `Add` que se encuentra en el miembro Items aplicado al menú propiamente dicho, mientras que la adición de un elemento desplegable (o subelemento) requiere la metodología `Add` que se encuentra en el miembro DropDownItems aplicado al elemento padre.

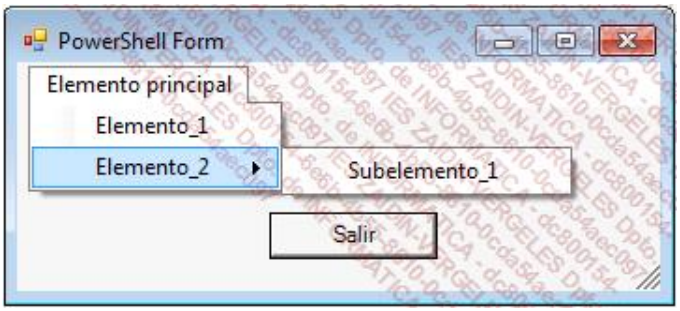
Ejemplo:

Correspondiente a la creación del menú superior.

```
# Creación del objeto Menú
$Menu = New-Object System.Windows.Forms.MenuStrip

# Declaración de los elementos
$elementos =
New-Object System.Windows.Forms.ToolStripMenuItem('Elemento principal')
$elemento_1 =
New-Object System.Windows.Forms.ToolStripMenuItem('Elemento_1')
$elemento_2 =
New-Object System.Windows.Forms.ToolStripMenuItem('Elemento_2')
$elemento_3 =
New-Object System.Windows.Forms.ToolStripMenuItem('Subelemento_1')

# Adición de los elementos
[void]$elementos.DropDownItems.Add($elemento_1)
[void]$elementos.DropDownItems.Add($elemento_2)
[void]$elemento_2.DropDownItems.Add($elemento_3)
[void]$Menu.Items.Add($elementos)

# Adición del menú al formulario
$form.Controls.Add($Menu)
```

La utilización de [void] permite no tener que mostrar sobre la consola el resultado de los comandos. Podríamos haber hecho lo mismo con la asignación de una variable en lugar de [void].

Añadiendo el código anterior a nuestra pequeña interfaz, el resultado será el siguiente:



Modelización de los menús gráficos

He aquí el script completo de nuestra interfaz gráfica:

```
#Form_3.ps1
#Creación de una interfaz gráfica
#con un botón salir activo y un menú

[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,160)
$boton_salir = New-Object System.Windows.Forms.Button
$boton_salir.Text = 'Salir'
$boton_salir.Location = New-Object System.Drawing.Size(135,80)
$boton_salir.Size = New-Object System.Drawing.Size(90,25)

#  Añadir el evento 'clic izquierdo' al botón
```

```
$boton_salir.Add_Click(
{
  $form.Close()
})

$form.Controls.Add($boton_salir)

# Creación del objeto Menú
$Menu = New-Object System.Windows.Forms.MenuStrip

# Declaración de los elementos
$elementos = New-Object System.Windows.Forms.ToolStripMenuItem('Elemento
principal')
$elemento_1 = New-Object System.Windows.Forms.ToolStripMenuItem('Elemento_1')
$elemento_2 = New-Object System.Windows.Forms.ToolStripMenuItem('Elemento_2')
$elemento_3 = New-Object System.Windows.Forms.ToolStripMenuItem
('Subelemento_1')

# Adición de los elementos
[void]$elementos.DropDownItems.Add($elemento_1)
[void]$elementos.DropDownItems.Add($elemento_2)
[void]$elemento_2.DropDownItems.Add($elemento_3)
[void]$Menu.Items.Add($elementos)

# Adición del menú al formulario
$form.Controls.Add($Menu)

# Visualización del formulario
$form.ShowDialog()
```

Timer

La implementación de un Timer en un Winform permite el inicio automático de acontecimientos en intervalos de tiempo determinado. Resultante de la clase `System.Windows.Forms.Timer`, el Timer es un complemento invisible en un formulario donde el intervalo de tiempo entre cada ejecución está fijado en milisegundos por la propiedad Interval. La declaración de las instrucciones a realizar en cada iniciación se efectúa aplicando el método `Add_Tick`. A continuación bastará, como para los eventos, incluir las acciones a emprender entre las llaves en el interior de los paréntesis del método como se muestra a continuación.

```
# creación del objeto
$timer = New-Object System.Windows.Forms.Timer

# Definición del intervalo a 1 segundo
$timer.Interval = 1000

$timer.Add_Tick({
   <Bloque de instrucciones>
})
```

Con los comandos anteriores, acabamos de configurar nuestro Timer. Pero queda por determinar cuando va a iniciarse y parar. Para ello, basta con aplicar los métodos `Start` y `Stop` para respectivamente iniciar y parar el Timer.

Podrá también iniciar/parar el Timer asignando el valor `True/False` a la propiedad `Enabled` del objeto `Timer`.

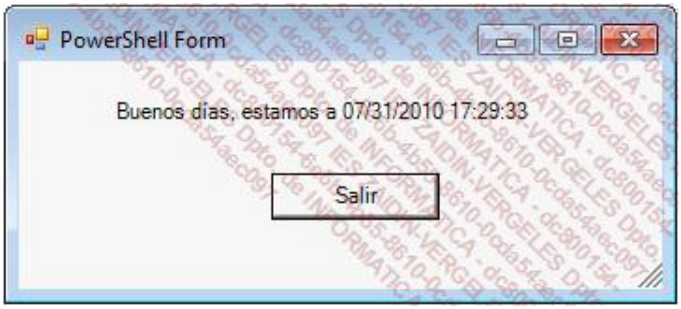
Con el fin de asimilar mejor el modo de utilizar un Timer, he aquí un ejemplo donde intervendrá la interfaz desarrollada en la parte «Creación de un formulario sencillo». Vamos esta vez a añadir dos nuevos controles:

- Un Label que va a mostrar vía el commandlet **`Get-Date`**, la fecha y la hora actual.
- Un Timer, que cada segundo, actualizará el valor contenido en el Label.

Script: Creación de una interfaz con Timer incorporado.

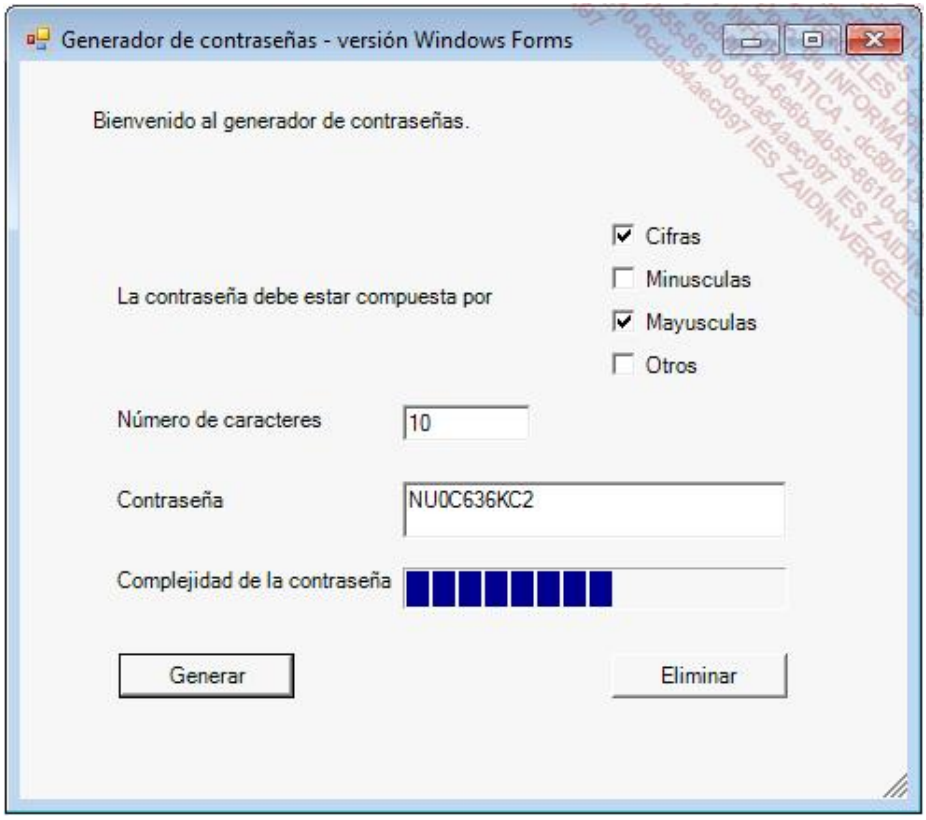
```
#Form_4.ps1
# Creación de una interfaz con Timer incorporado
[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
$form = New-Object Windows.Forms.Form

# Creación del objeto Timer
$timer = New-Object System.Windows.Forms.Timer
$form.Text = 'PowerShell Form'
$form.Size = New-Object System.Drawing.Size(360,160)
$boton_salir = New-Object System.Windows.Forms.Button
$boton_salir.Text = 'Salir'
$boton_salir.Location = New-Object System.Drawing.Size(135,60)
$boton_salir.Size = New-Object System.Drawing.Size(90,25)
$boton_salir.Add_Click( {$form.Close()} )

# Creación de un Label
$label1 = New-Object System.Windows.Forms.label
$label1.Location = New-Object System.Drawing.Point(50,20)
$label1.Autosize = $true

# Creación del objeto Timer
$timer = New-Object System.Windows.Forms.Timer

$timer.Interval = 1000  # Definición del intervalo a 1 segundo
$timer.Add_Tick({
   $label1.Text = "Buenos días, estamos a $(Get-Date)"
})

$timer.Start()
$form.Controls.Add($boton_salir)
$form.Controls.Add($label1)
$form.ShowDialog()
```

Resultado, nuestra interfaz muestra cada segundo la fecha y la hora actual.



Ejemplo: generador de contraseñas complejas

Para concluir con la creación de interfaces gráficas, y para mostrarle mejor las capacidades gráficas de PowerShell gracias al Framework .NET, le proponemos un script que recoge algunos elementos descritos en este capítulo, tales como la inserción de los componentes de un formulario, las casillas de verificación, zonas de texto, etc. El script que le presentaremos crea una interfaz (véase la figura siguiente) que permite generar las contraseñas más o menos complejas mediante diferentes criterios, como su composición y su longitud.



Interfaz del generador de contraseñas

El script es el siguiente:

Script: Script de generación de contraseñas.

```
# WinForms_pwdgenerator.ps1

[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')

$textBox_resultado = New-Object System.Windows.Forms.TextBox
$progressBar = New-Object System.Windows.Forms.ProgressBar
$button_generar = New-Object System.Windows.Forms.Button
$checkBox_cifras = New-Object System.Windows.Forms.CheckBox
$checkBox_minusculas = New-Object System.Windows.Forms.CheckBox
$checkBox_mayusculas = New-Object System.Windows.Forms.CheckBox
$button_eliminar = New-Object System.Windows.Forms.Button
$label1 = New-Object System.Windows.Forms.Label
$checkBox_otros = New-Object System.Windows.Forms.CheckBox
$label2 = New-Object System.Windows.Forms.Label
$label3 = New-Object System.Windows.Forms.Label
$textBox_Nb_caracteres = New-Object System.Windows.Forms.TextBox
$label4 = New-Object System.Windows.Forms.Label
$label_principal = New-Object System.Windows.Forms.Label
#
```

```
# textBox_resultado
#
$textBox_resultado.Location = New-Object System.Drawing.Point(205, 225)
$textBox_resultado.Multiline = $true
$textBox_resultado.Name = 'textBox_resultado'
$textBox_resultado.Size = New-Object System.Drawing.Size(206, 31)
$textBox_resultado.TabIndex = 2
#
# progressBar
#
$progressBar.Location = New-Object System.Drawing.Point(205, 271)
$progressBar.Name = 'progressBar'
$progressBar.Size = New-Object System.Drawing.Size(206, 23)
$progressBar.TabIndex = 3
$progressBar.set_forecolor('darkblue')
#
# button_generar
#
$button_generar.Location = New-Object System.Drawing.Point(53, 317)
$button_generar.Name = 'button_generar'
$button_generar.Size = New-Object System.Drawing.Size(94, 24)
$button_generar.TabIndex = 4
$button_generar.Text = 'Generar'
$button_generar.UseVisualStyleBackColor = $true

$button_generar.Add_Click({
   [int]$len = $textBox_Nb_caracteres.get_text()
   $textBox_resultado.Text = ''
   $complex = 0
   $progressBar.Value = 0
   [string]$chars = ''

   if ($checkBox_cifras.Checked)
        {$chars += '0123456789';$complex += 1}
   if ($checkBox_mayusculas.Checked)
        {$chars += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';$complex += 1}
   if ($checkBox_minusculas.Checked)
        {$chars += 'abcdefghijklmnopqrstuvwxyz';$complex += 1}
   if ($checkBox_otros.Checked)
        {$chars += '_!@#$%';$complex += 1}

   if($chars -ne ''){
      $bytes = New-Object System.Byte[] $len
      $rnd =
      New-Object System.Security.Cryptography.RNGCryptoServiceProvider
      $rnd.GetBytes($bytes)
      $resultado = ''
      for( $i=0; $i -lt $len; $i++ )
      {
         $resultado += $chars[ $bytes[$i] % $chars.Length ]
      }
      $complex *= $(2.57*$len)
      if($complex -gt 100){ $complex = 100 }
      $progressBar.Value = $complex
      $textBox_resultado.Text = $resultado
```

```
   }
})
#
# checkBox_cifras
#
$checkBox_cifras.AutoSize = $true
$checkBox_cifras.Location = New-Object System.Drawing.Point(317, 85)
$checkBox_cifras.Name = 'checkBox_cifras'
$checkBox_cifras.Size = New-Object System.Drawing.Size(61, 17)
$checkBox_cifras.TabIndex = 5
$checkBox_cifras.Text = 'Cifras'
$checkBox_cifras.UseVisualStyleBackColor = $true
#
# checkBox_minusculas
#
$checkBox_minusculas.AutoSize = $true
$checkBox_minusculas.Location = New-Object System.Drawing.Point(317, 108)
$checkBox_minusculas.Name = 'checkBox_minusculas'
$checkBox_minusculas.Size = New-Object System.Drawing.Size(79, 17)
$checkBox_minusculas.TabIndex = 6
$checkBox_minusculas.Text = 'Minúsculas'
$checkBox_minusculas.UseVisualStyleBackColor = $true
#
# checkBox_mayusculas
#
$checkBox_mayusculas.AutoSize = $true
$checkBox_mayusculas.Location = New-Object System.Drawing.Point(317, 131)
$checkBox_mayusculas.Name = 'checkBox_mayusculas'
$checkBox_mayusculas.Size = New-Object System.Drawing.Size(79, 17)
$checkBox_mayusculas.TabIndex = 7
$checkBox_mayusculas.Text = 'Mayúsculas'
$checkBox_mayusculas.UseVisualStyleBackColor = $true
#
# button_eliminar
#
$button_eliminar.Location = New-Object System.Drawing.Point(317, 317)
$button_eliminar.Name = 'button_eliminar'
$button_eliminar.Size = New-Object System.Drawing.Size(94, 24)
$button_eliminar.TabIndex = 8
$button_eliminar.Text = 'Eliminar'
$button_eliminar.UseVisualStyleBackColor = $true
$button_eliminar.Add_Click({$Form1.Close()})
#
# label1
#
$label1.AutoSize = $true
$label1.Location = New-Object System.Drawing.Point(50, 271)
$label1.Name = 'label1'
$label1.Size = New-Object System.Drawing.Size(139, 13)
$label1.TabIndex = 9
$label1.Text = 'Complejidad de la contraseña'
#
# checkBox_otros
#
$checkBox_otros.AutoSize = $true
```

```
$checkBox_otros.Location = New-Object System.Drawing.Point(317, 154)
$checkBox_otros.Name = 'checkBox_otros'
$checkBox_otros.Size = New-Object System.Drawing.Size(56, 17)
$checkBox_otros.TabIndex = 10
$checkBox_otros.Text = 'Otros'
$checkBox_otros.UseVisualStyleBackColor = $true
#
# label2
#
$label2.AutoSize = $true
$label2.Location = New-Object System.Drawing.Point(50, 119)
$label2.Name = 'label2'
$label2.Size = New-Object System.Drawing.Size(227, 15)
$label2.TabIndex = 11
$label2.Text = 'La contraseña debe estar compuesta por'
#
# label3
#
$label3.AutoSize = $true
$label3.Location = New-Object System.Drawing.Point(50, 185)
$label3.Name = 'label3'
$label3.Size = New-Object System.Drawing.Size(129, 15)
$label3.TabIndex = 12
$label3.Text = 'Número de caracteres'
#
# textBox_Nb_caracteres
#
$textBox_Nb_caracteres.Location = New-Object System.Drawing.Point(205, 184)
$textBox_Nb_caracteres.Name = 'textBox_Nb_caracteres'
$textBox_Nb_caracteres.Size = New-Object System.Drawing.Size(69, 20)
$textBox_Nb_caracteres.TabIndex = 13
$textBox_Nb_caracteres.Text = '10'
#
# label4
#
$label4.AutoSize = $true
$label4.Location = New-Object System.Drawing.Point(50, 228)
$label4.Name = 'label4'
$label4.Size = New-Object System.Drawing.Size(71, 13)
$label4.TabIndex = 14
$label4.Text = 'Contraseña'
#
# label_principal
#
$label_principal.AutoSize = $true
$label_principal.Location = New-Object System.Drawing.Point(37, 25)
$label_principal.Name = 'label_principal'
$label_principal.Size = New-Object System.Drawing.Size(355, 20)
$label_principal.TabIndex = 15
$label_principal.Text = 'Bienvenido al generador de contraseñas.'
#
$Form1 = New-Object System.Windows.Forms.form
# Form1
#
```

```
$Form1.ClientSize = New-Object System.Drawing.Size(475, 395)
$Form1.Controls.Add($label_principal)
$Form1.Controls.Add($label4)
$Form1.Controls.Add($textBox_Nb_caracteres)
$Form1.Controls.Add($label3)
$Form1.Controls.Add($label2)
$Form1.Controls.Add($checkBox_otros)
$Form1.Controls.Add($label1)
$Form1.Controls.Add($button_eliminar)
$Form1.Controls.Add($checkBox_mayusculas)
$Form1.Controls.Add($checkBox_minusculas)
$Form1.Controls.Add($checkBox_cifras)
$Form1.Controls.Add($button_generar)
$Form1.Controls.Add($progressBar)
$Form1.Controls.Add($textBox_resultado)
$Form1.Name = 'Form1'
$Form1.Text = 'Generador de contraseñas - versión Windows Forms'
$Form1.ShowDialog()
```

d. El conversor de formularios

PowerShell no dispone de editor gráfico para generar interfaces, por lo que todo debe realizarse por línea de comandos. Esto suele ser largo y tedioso, sobre todo en el momento de precisar la posición de los componentes.

Para responder a esta necesidad, encontrará en nuestro sitio Web www.powershell-scripting.com un script denominado `CSForm2PS.ps1` capaz de transcribir un formulario gráfico desarrollado con el software Visual C# en un script PowerShell.


Este script conversor de formularios no es un editor gráfico, pero permite pasar de C# al PowerShell por transformación de líneas de código. La utilización del conversor se limita a la conversión de los componentes gráficos. No tendrá en cuenta los eventos ni la configuración específica de los componentes.

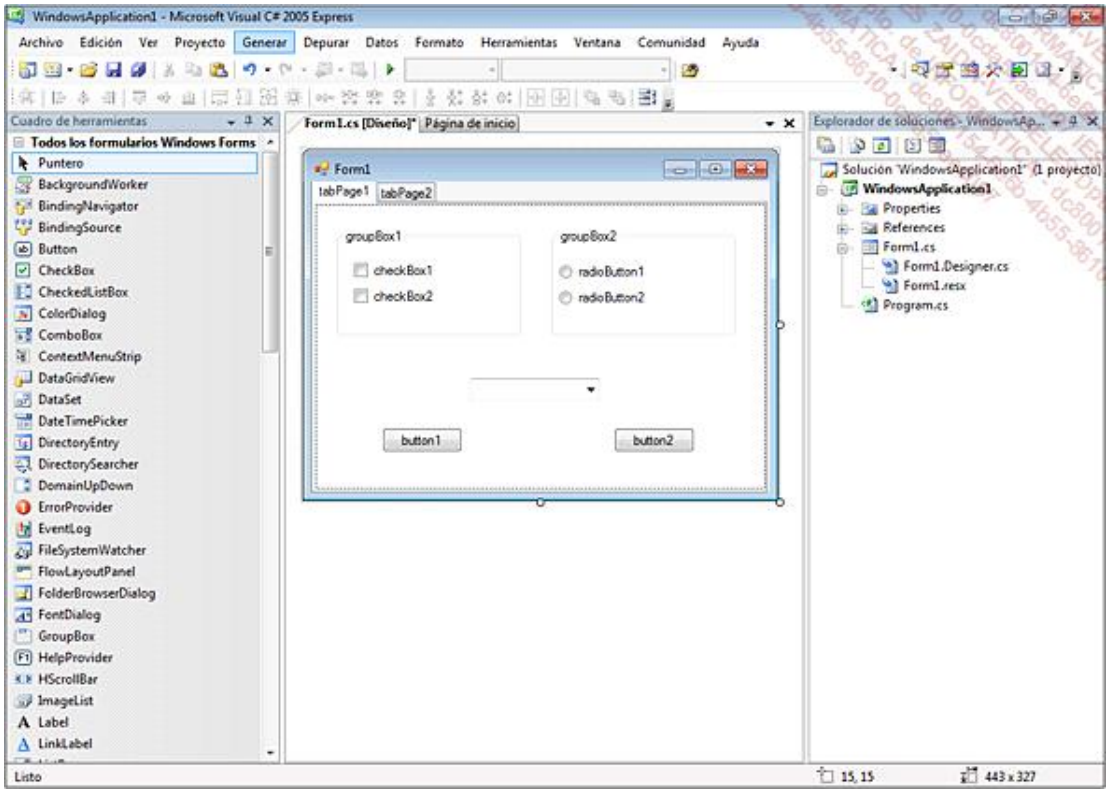
Sencillo y rápido: ya no volverá a perder tanto tiempo midiendo el tamaño y la ubicación de cada componente. Ahora bastará con algunos clics en Visual C#.


Si no desea adquirir una licencia para el software visual C#, sepa que éste último se puede descargar gratuitamente en su versión Express. Versión descargable en Internet y que es más que suficiente para la realización de formularios que podrá después convertir en PowerShell.

- Para crear un formularios gráfico, inicie Visual C#, y seleccione crear un nuevo proyecto pulsando en **Archivo** y luego **Nuevo proyecto...**.

Ventana de inicio de la aplicación Visual C#

Después de haber escogido el modelo **Aplicación para Windows**, le será posible modificar el formulario que se crea por defecto y añadirle otros componentes disponibles en la barra de herramientas, ver la figura siguiente.



Creación de una interfaz en Visual C#

Atención, el conversor de formulario no permite tratar todos los tipos de componentes. Para más información sobre los componentes aceptados, puede consultar el sitio web de MSDN y hacer una búsqueda por Visual C#

(http://msdn.microsoft.com/es-es/vcsharp/default.aspx)

Después de haber definido la interfaz según sus necesidades, guarde el proyecto con el nombre que haya elegido y recupere el archivo *<nombre de formulario>.Designer.cs* (por defecto, este archivo se denomina *form1.Designer.cs*).

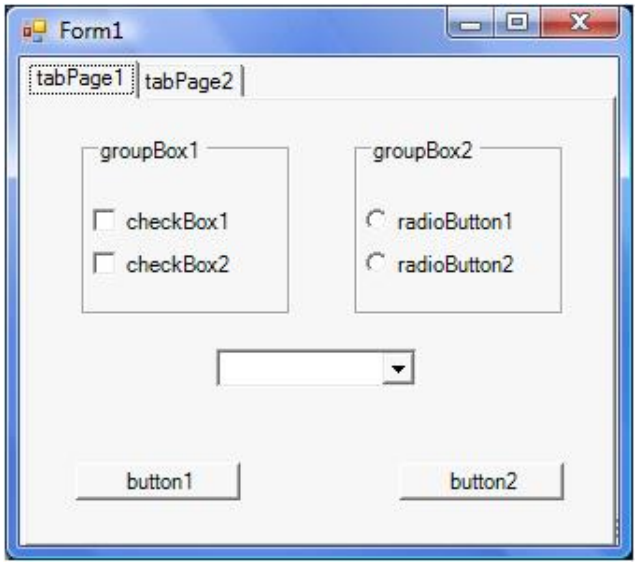
Sólo faltará utilizar el script `CSForm2PS.ps1` con los parámetros **`-source`** y **`-dest`** que especifican respectivamente el nombre del archivo creado con Visual C#, y el nombre del script PowerShell que desea generar.

Ejemplo:

```
PS > ./CSForm2PS.ps1 -source <Archivo *.designer.cs> -dest <Archivo.ps1>
```

El archivo *.designer.cs se transformará para convertirse en un script PowerShell.

Ejemplo del resultado gráfico obtenido al ejecutar un script generado por **`CSForm2PS.ps1`**:

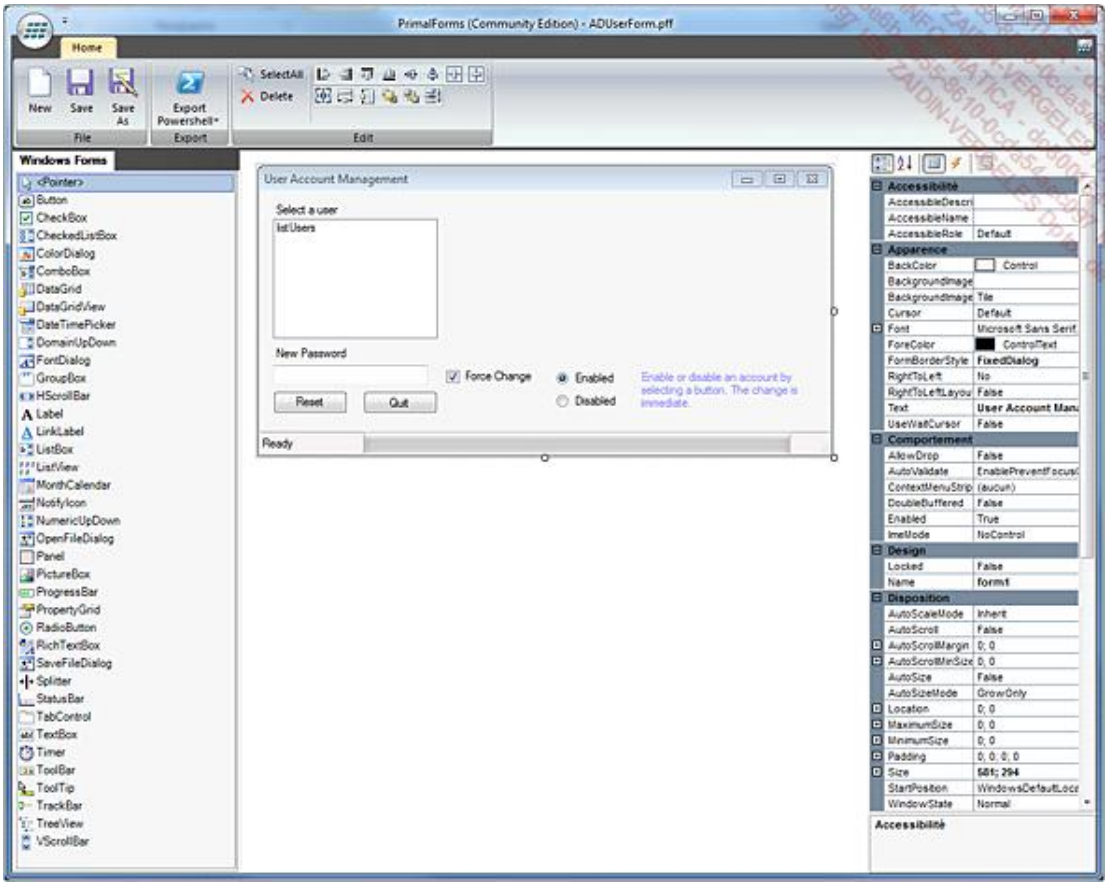


El conversor de formularios era la herramienta final para la creación de interfaces de tipo Windows forms hasta que la herramienta **Primal Forms** lo destronó. Por ello, vale de la pena hacer mención a su existencia, al menos para poner de relieve el logro técnico que supone convertir al vuelo un código C# en un código PowerShell, lo que demuestra de que finalmente estos dos lenguajes no son tan diferentes...

e. Sapien Primal Forms Community Edition

PrimalForms Community Edition de la sociedad Sapien Technologies (http://www.primaltools.com/downloads/) es una utilidad gratuita que permite crear increíbles interfaces gráficas Windows Forms en un tiempo récord.

Con PrimalForms, usted tendrá acceso a la lista de los componentes gráficos (tales como las listas desplegables, las zonas de inserción de texto, las casillas de selección, etc.) y bastará, a semejanza de un Visual Studio, de *arrastrar* los componentes hasta el formulario para que aparezcan. Una vez creada la interfaz, no hay más que exportarla al portapapeles o bien directamente a un archivo PS1.



Primal Forms Community Edition

Como puede suponer, una herramienta como esta le evita tener que escribir muchas líneas de script. Es una ayuda inestimable en términos de ganancia de tiempo en la definición de las interfaces.

A continuación, una vez el script PowerShell se ha generado es preciso modificarlo para añadir eventos tales como las acciones a realizar al hacer clic sobre uno de los botones de la interfaz.

Veamos un extracto de un script exportado:

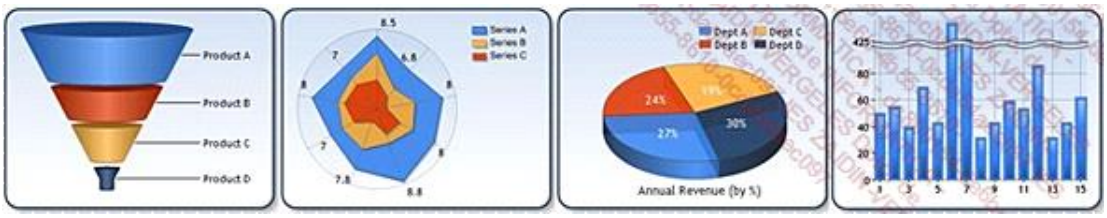
```
#----------------------------------------------
#Generated Event Script Blocks
#----------------------------------------------
#Provide Custom Code for events specified in PrimalForms.
...
$handler_button1_Click=
{
   #TODO: Place custom script here
}
$handler_listBox1_SelectedIndexChanged=
{
   #TODO: Place custom script here
}
#----------------------------------------------
#region Generated Form Code
$button1.TabIndex = 2

$button1.Name = "button1"
$System_Drawing_Size = New-Object System.Drawing.Size
$System_Drawing_Size.Width = 75
```

```
$System_Drawing_Size.Height = 23
$button1.Size = $System_Drawing_Size
$button1.UseVisualStyleBackColor = $True
$button1.Text = "Commander"
$System_Drawing_Point = New-Object System.Drawing.Point
$System_Drawing_Point.X = 45
$System_Drawing_Point.Y = 245
$button1.Location = $System_Drawing_Point
$button1.DataBindings.DefaultDataSourceUpdateMode = 0
$button1.add_Click($handler_button1_Click)

$form.Controls.Add($button1)
```

Mientras que **PrimalForms Community Edition** es gratuito, sepa que existe una versión de pago llamada **PrimalForms 2009**. La versión de pago proporciona acceso a nuevas funcionalidades tales como: un editor de scripts con coloración de sintaxis, la ayuda en línea contextual (al estilo *IntelliSense*), un explorador de objetos .NET, y una herramienta increíble que hace de **PrimalForms** un producto único: un empaquetador.

El empaquetador permite transformar cualquier script PowerShell en un ejecutable. Es posible, incluso indicar una cuenta y una contraseña de un usuario con privilegios para ejecutar scripts con permisos diferentes a los de usuario. Una condición: tener PowerShell instalado en los puestos de trabajo que llamen a los scripts transformados en ejecutables.

f. Creación de gráficos con los MS Charts Controls



MS Charts Controls for .NET Framework 3.5 en acción

MS Charts Controls for .NET Framework 3.5 es una biblioteca de objetos gráficos, proporcionada gratuitamente por Microsoft, diseñada para las tecnologías **Windows Forms** y **ASP.NET**. Estos objetos, llamados controles, son muy agradables a la vista y por lo tanto están muy bien adaptados para la presentación de resultados.

El único requisito previo es el Framework .NET 3.5 SP1, y por supuesto PowerShell (v1 o v2).

He aquí el enlace para descargar esta biblioteca (1.8 Mb): http://www.microsoft.com/downloads/details.aspx?FamilyId=130F7986-BF49-4FE5-9CA8-910AE6EA442C&displaylang=es

Microsoft Chart Controls for Microsoft .NET Framework 3.5 instalará nuevos ensamblados que contienen los controles gráficos ASP.NET y Windows Forms. La instalación es muy sencilla: se resume a pulsar **Siguiente**, **Siguiente** y **Finalizar**.

Tomemos un ejemplo para tratar de mostrar en forma de gráfico de barras 3D los cinco procesos con mayor consumo de memoria.

Para comenzar, instanciamos un control gráfico de tipo `Chart` o más concretamente de tipo `System.Windows.Forms.DataVisualization.Charting.Chart`. Y le damos una cierta dimensión, como se muestra a continuación:

```
# Creación del objeto Chart
$Chart = New-Object System.Windows.Forms.DataVisualization.Charting.Chart
$Chart.Width = 500
$Chart.Height = 400
$Chart.Left = 40
```

```
$Chart.Top = 30
```

Podemos también aprovechar para ponerle un título a nuestro gráfico:

```
[void]$Chart.Titles.Add('Top 5 de los procesos con mayor consumo de memoria')
```

Definimos seguidamente la propiedad ChartAreas del gráfico como se muestra a continuación:

```
$ChartArea =
New-Object System.Windows.Forms.DataVisualization.Charting.ChartArea
$ChartArea.AxisX.Title = 'Procesos (PID)'
$ChartArea.AxisY.Title = 'Memoria de paginación utilizada (en Mb)'
$Chart.ChartAreas.Add($ChartArea)
```

Gracias a esta propiedad definimos las leyendas en los ejes X e Y.

Acto seguido llega la generación de datos. El objetivo para llegar a nuestro fin es pasar dos variables de tipo tabla a nuestra objeto gráfico. En el eje X pasaremos la tabla que contiene los nombres de los procesos y en el eje Y la tabla de valores asociados.

Para ello empezamos por recuperar los datos :

```
$Procesos = Get-Process | Sort-Object -Property PM | Select-Object -Last 5
```

Se recuperará por tanto 5 objetos de tipo proceso clasificados por valor creciente de la propiedad PM (*Paged Memory* - memoria de paginación). Si se muestra el contenido de la variable `$Procesos`, veamos lo que podemos obtener:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    325      18    65284      22640   148    58,50   3632 svchost
    964      34    69132      55576   262    59,65   3436 iexplore
    297      16    71708      57788   271     1,59   2944 powershell_ise
   1355      42    80912      75584   370   231,38    908 explorer
    429      18   202204     184028   770   351,50   1200 javaw
```

En la variable `$ProcNames` ponemos el nombre de los procesos, así como su número de identificación entre paréntesis y lo convertimos todo en una tabla gracias a la arroba:

```
$ProcNames = @(foreach($Proc in $Procesos){$Proc.Name + '(' + $Proc.ID +')'})
```

La línea de script anterior es la forma condensada de:

```
$ProcNames = @(
          foreach($Proc in $Procesos)
          {
              $Proc.Name + '(' + $Proc.ID +')'
          }
)
```

`$ProcNames` contiene actualmente los valores siguientes:

```
svchost(3632)
iexplore(3436)
powershell_ise(2944)
explorer(908)
java
```

En la variable $PM almacenamos los valores correspondientes a las propiedades PM de nuestros procesos y convertimos el resultado en una tabla:

```
$PM = @(foreach($Proc in $Procesos){$Proc.PM/1MB})
```

A fin de obtener un resultado más comprensible convertimos los valores en MB gracias al cuantificador de bytes MB. Veamos el contenido de la variable $PM:

```
63,75390625
67,51171875
70,02734375
79,015625
197,46484375
```

Pasamos las tablas de valor a la propiedad Series en el formato siguiente:

```
[void]$Chart.Series.Add('Data')
$Chart.Series['Data'].Points.DataBindXY($ProcNames, $PM)
```

Definimos la apariencia del gráfico (columnas 3D):

```
# Definición del tipo de gráfico (columnas)
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column

# Definición del estilo del gráfico (cilindros 3D)
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'
```

Aprovechamos para dar un poco de color a los valores mínimo y máximo:

```
# Encontrar los valores mínimo y máximo y aplicación de colores
$maxValuePoint = $Chart.Series['Data'].Points.FindMaxByValue()
$maxValuePoint.Color = [System.Drawing.Color]::Red
$minValuePoint = $Chart.Series['Data'].Points.FindMinByValue()
$minValuePoint.Color = [System.Drawing.Color]::Green
```

Y por último creamos el formulario, con un nombre, su tamaño y le asignamos el objeto correspondiente a nuestro gráfico:

```
# Creación del formulario Windows Forms
$Form = New-Object Windows.Forms.Form
$Form.Text = 'PowerShell MS Charts Demo'
$Form.Width = 600
$Form.Height = 500
$Form.Controls.Add($Chart)
```

Después para concluir lo visualizamos:

```
# Visualización del formulario
$Form.Add_Shown({$Form.Activate()})
$Form.ShowDialog()
```

Y obtendremos el resultado siguiente:

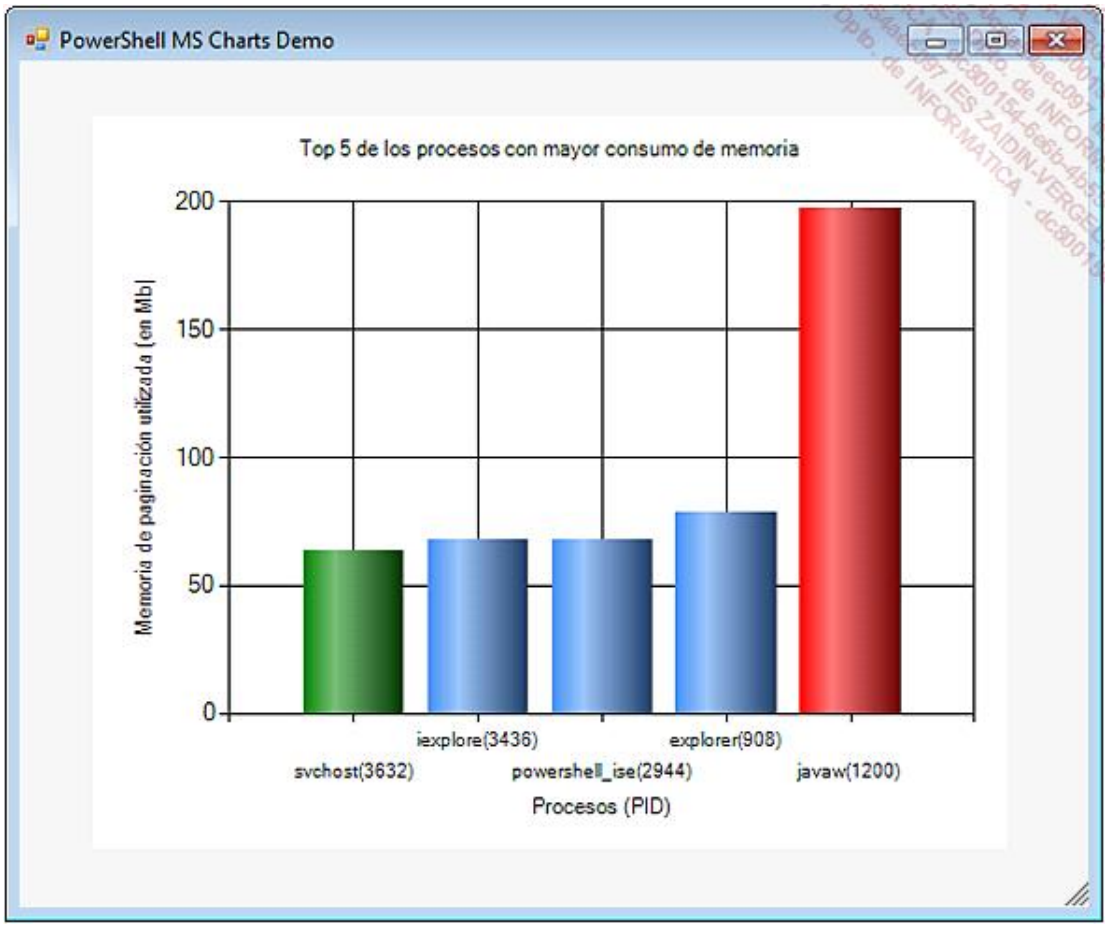


Gráfico de barras 3D con los MS Charts Controls

Veamos el script en su totalidad:

```
# Gráfico de barras 3D con los MS Charts Controls
# Carga de los ensamblados
[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms')
[void][Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms.
DataVisualization')

# Creación del objeto Chart
$Chart = New-object System.Windows.Forms.DataVisualization.Charting.Chart
$Chart.Width = 500
$Chart.Height = 400
$Chart.Left = 40
$Chart.Top = 30
# Añadimos el título
[void]$Chart.Titles.Add('Top 5 de los procesos con mayor consumo
de memoria')

# Creación de una zona de diseño y adición del objeto Chart a esta zona
$ChartArea = New-Object System.Windows.Forms.DataVisualization.Charting.
ChartArea
$ChartArea.AxisX.Title = 'Procesos (PID)'
$ChartArea.AxisY.Title = 'Memoria de paginación utilizada (en Mb)'
$Chart.ChartAreas.Add($ChartArea)

# Adición de datos
```

```
$Procesos = Get-Process | Sort-Object -Property PM | Select-Object -Last 5
$ProcNames = @(foreach($Proc in $Procesos){$Proc.Name + '(' + $Proc.ID +')'})
$PM = @(foreach($Proc in $Procesos){$Proc.PM/1MB})
[void]$Chart.Series.Add('Data')
$Chart.Series['Data'].Points.DataBindXY($ProcNames, $PM)

# Definición del tipo de gráfico (columnas)
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column

# Definición del estilo del gráfico (cilindros 3D)
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'

# Encontrar los valores mínimo y máximo y aplicación de colores
$maxValuePoint = $Chart.Series['Data'].Points.FindMaxByValue()
$maxValuePoint.Color = [System.Drawing.Color]::Red
$minValuePoint = $Chart.Series['Data'].Points.FindMinByValue()
$minValuePoint.Color = [System.Drawing.Color]::Green

# Creación del formulario Windows Forms
$Form = New-Object Windows.Forms.Form
$Form.Text = 'PowerShell MS Charts Demo'
$Form.Width = 600
$Form.Height = 500
$Form.Controls.Add($Chart)

# Visualización del formulario
$Form.Add_Shown({$Form.Activate()})
$Form.ShowDialog()
```

Por último, el resultado podría comprenderse mejor si se presenta en forma circular.

Reemplazar las líneas siguientes:

```
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Column
$Chart.Series['Data']['DrawingStyle'] = 'Cylinder'
```

Por estas:

```
$Chart.Series['Data'].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::pie
$Chart.Series['Data']['PieLabelStyle'] = 'inside'
$Chart.Series['Data']['PieDrawingStyle'] = 'concave'
```

¡Aquí tenemos el resultado!

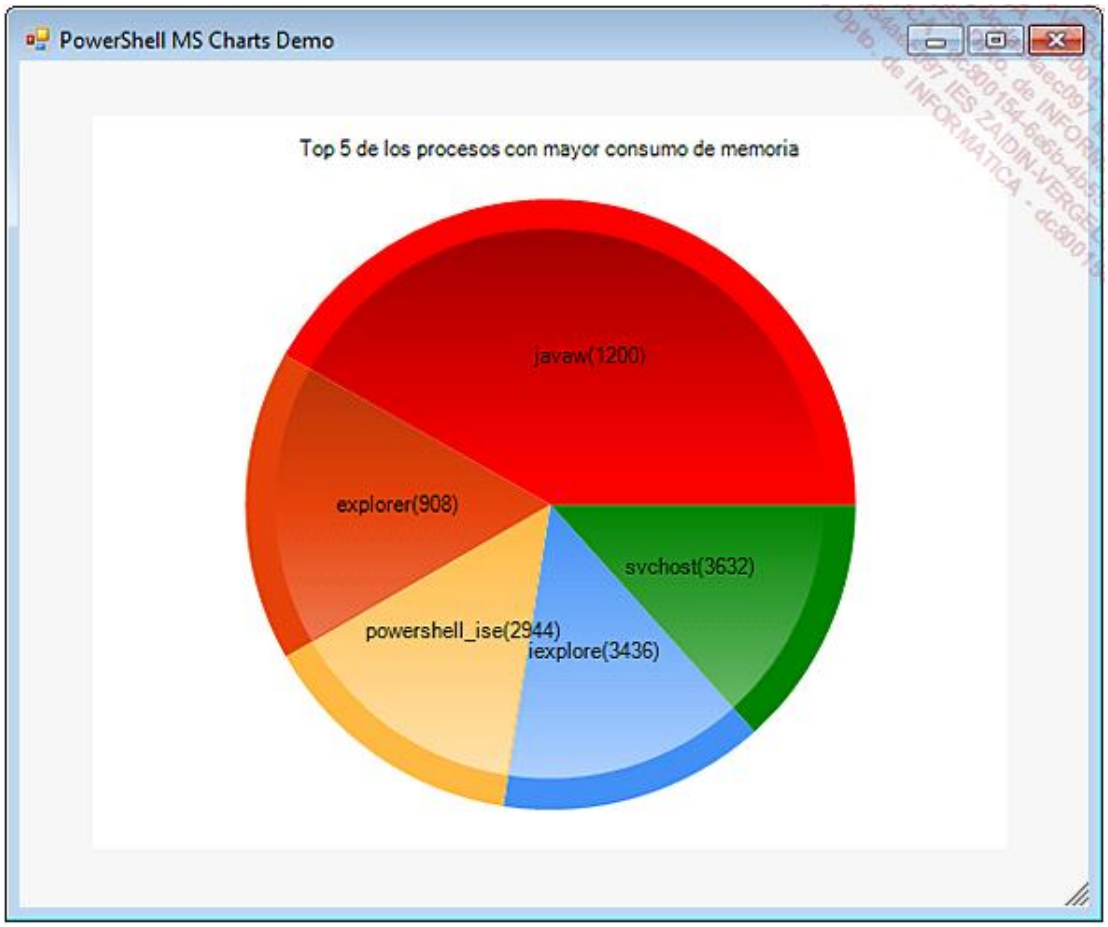


Gráfico circular 3D con los MS Charts Controls

g. Biblioteca LibraryChart

La biblioteca **LibraryChart**, creada por *Chad Miller*, y que puede encontrarse en esta dirección: http://poshcode.org/1330 tiene como objetivo simplificar la utilización de los **MS Charts Controls**. **LibraryChart** propone las funcionalidades siguientes:

- Envío del resultado de uno o varios comandos PowerShell vía la tubería directamente en un gráfico,
- Visualización de los gráficos en un **Windows Form**,
- Guardar los gráficos como una imagen,
- Elección variada de gráficos: barras horizontales, columnas, circular, etc.
- Actualización de gráficos en tiempo real, gracias a la utilización de bloques de scripts; estos últimos se ejecutarán con un intervalo de tiempo determinado,
- Compatibilidad PowerShell v1

Un vez descargada la librería, bastará con cargarla de la siguiente forma:

```
PS >. ./libraryChart.ps1
```

No olvide poner un punto seguido de un espacio antes de la llamada del script (DotSourcing - ver capítulo Fundamentos) sino la librería no se importará en la sesión del script (el ámbito) actual. Por tanto las funciones de la

librería no se cargará en memoria.

Una vez hecho esto, podemos empezar a crear los gráficos muy fácilmente.

Ejemplo:

Visualización en modo columnas de los 10 procesos que ocupan la mayor parte de la memoria de paginación.

```
PS > Get-Process | Sort-Object -Property PM | Select-Object -Last 10 |
Out-Chart -x Name -y PM
```

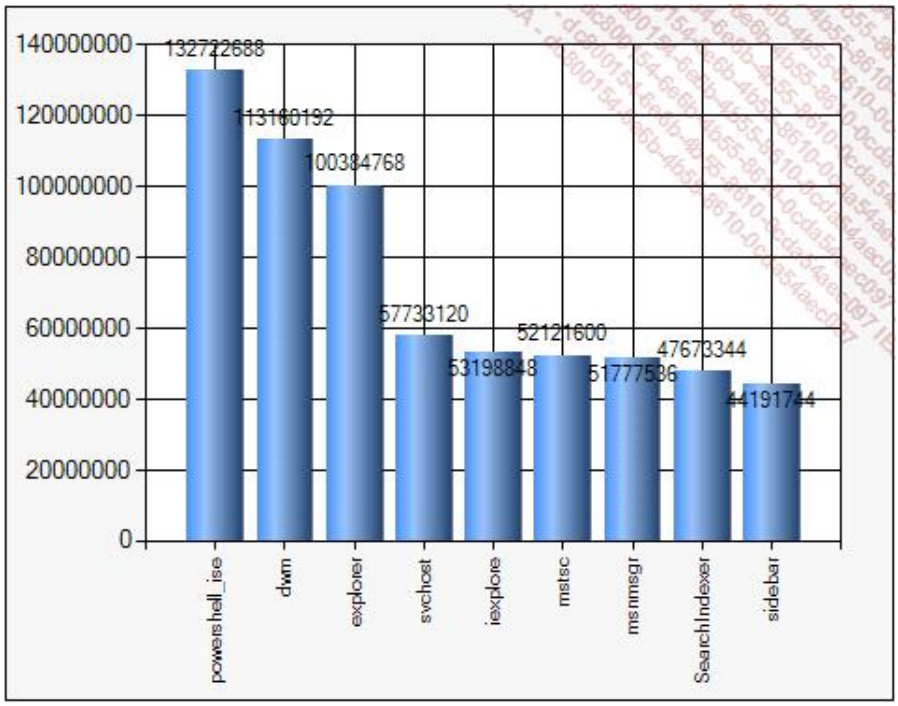


Gráfico de barras con los MS Charts Controls y LibraryChart

Ejemplo:

Muestra en modo circular de los 10 procesos que ocupan la mayor parte de la memoria de paginación.

```
PS > Get-Process | Sort-Object -Property PM | Select-Object -Last 10 |
Out-Chart -x Name -y PM -chartType pie
```

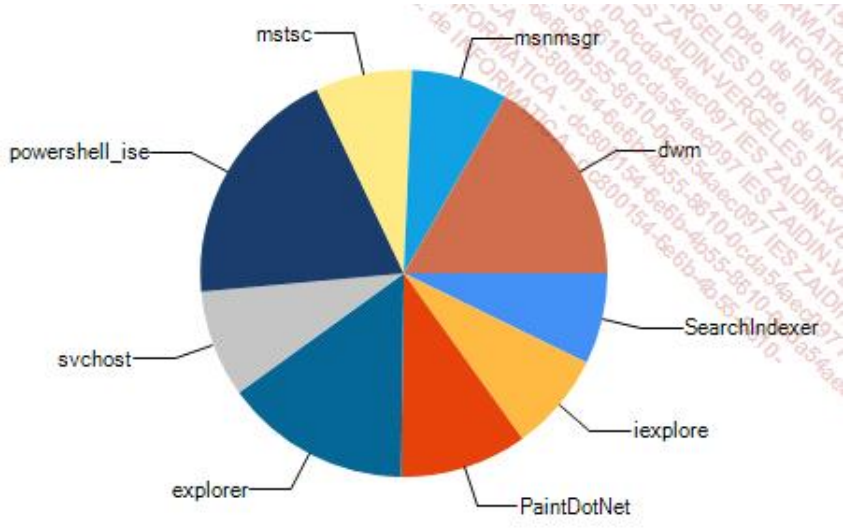


Gráfico circular con los MS Charts Controls y LibraryChart

Como verá el rendimiento líneas de script/resultado obtenido es más que rentable... Si necesita los efectos 3D u otras funcionalidades, bastará con modificar el script PowerShell correspondiente a la librería.

3. Windows Presentation Foundation

Las interfaces gráficas **WPF** representan, desde el punto de vista de Microsoft y de la comunidad de desarrolladores, la vía del futuro. Microsoft está claramente orientado a invertir (después de 2006) en **WPF** y dejar de hacerlo en **Windows Forms**, como futura plataforma de presentación.

Hay muchas razones para ello; en primer lugar el aspecto vectorial de **WPF** hace que las interfaces gráficas **WPF** sean independientes de la resolución de las pantallas. De este modo una interfaz creada en una pantalla de 14 pulgadas, tendrá el mismo rendimiento que en una pantalla de 50 pulgadas. Además, **WPF** es una tecnología nueva, por lo que sabe sacar partido de la aceleración por hardware basándose en la API Direct 3D. En fin, los elementos que componen **WPF** presentan una increíble riqueza y una gran flexibilidad de utilización. Por ejemplo, incluso sin estar recomendado, sería fácil crear un desplegable lleno de animaciones 2D o de vídeo clips.

Pero **WPF** tiene la enorme ventaja, en comparación con **Windows Forms,** de disponer del lenguaje XAML. Gracias a XAML, la interfaz gráfica ya no se construirá en PowerShell mediante la llamada a los métodos .NET (incluso si continuase siendo posible), sino que estará descrita con una gramática XML. Esto es un punto muy importante para el mantenimiento de los scripts con interfaz gráfica. Con los **Windows Forms** la interfaz es una parte integrante del script, y por lo tanto no será fácil modificarla. En efecto, la lógica del script y la definición de la interfaz están mezcladas. Ahora, con **WPF** la interfaz gráfica está almacenada en un archivo externo, lo que permitirá poder modificarla posteriormente con los instrumentos adecuados sin tener que tocar el script PowerShell que hace la llamada.

Se puede sin embargo, si se desea, integrar el código XAML en un script PowerShell almacenándolo en una variable de tipo *Here String*. Lo que en algunos casos puede resultar útil con el fin de evitar las dependencias entre archivos para tener un script PowerShell autosuficiente.

a. Antes de empezar...

Antes de lanzarse en cuerpo y alma en el scripting con **WPF**, debe saber que existen algunos requisitos previos. El primero, ya lo conoce, y es que será necesario utilizar como mínimo la versión 2 de PowerShell. El segundo, es que será necesario:

- Ejecutar el script en PowerShell ISE (*Integrated Scripting Environment*), el editor gráfico PowerShell,
- Ejecutar el script en la consola PowerShell, pero no olvidando lanzar éste último con el parámetro -STA, es decir:

```
PowerShell.exe -STA
```

Si usted ha optado por el segundo método, entonces necesitará siempre empezar los scripts por la carga de los ensamblados PresentationFramework, PresentationCore, y WindowsBase como mostramos a continuación:

```
Add-Type -assemblyName PresentationFramework
Add-Type -assemblyName PresentationCore
Add-Type -assemblyName WindowsBase
```

Observe que si usa PowerShell ISE la carga de los ensamblados no será necesaria, ya que ya lo habrán hecho por usted; PowerShell ISE es ella misma una aplicación **WPF**. En los ejemplos que veremos a continuación, a fin de limitar el número de líneas de script, omitiremos cargar los ensamblados. No olvide por tanto añadir esas líneas al inicio de los scripts si los ejecuta a partir de la consola PowerShell clásica.

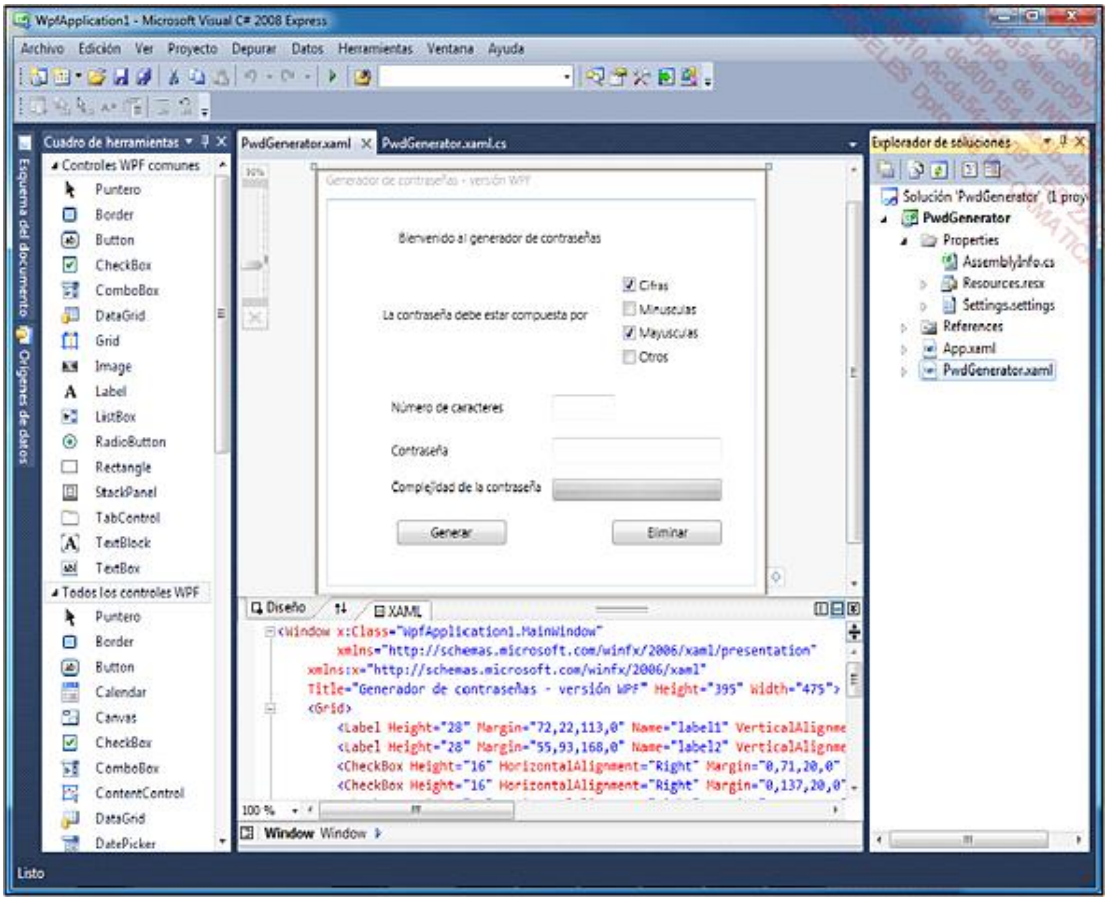
b. Creación asistida de interfaces gráficas WPF

WPF, en comparación con PowerShell es una tecnología reciente, en efecto **WPF** sólo se utiliza después de la versión 2 de PowerShell. Por consiguiente, no existe todavía un instrumento integrado tal como **PrimalForms** para construir las interfaces gráficas. Esto nos hará por tanto, en cierto modo, aventureros... Y como tales, tendremos que actuar un poco con astucia.

En primer lugar, incluso aunque algunos administradores de sistemas puedan no estar muy de acuerdo, tendremos que utilizar una herramienta para diseñar nuestra interfaz gráfica. Y para ello, nuestra elección se centrará en **WPF Designer** incluido en el conjunto de desarrollo **Visual Studio 2008**. **Visual Studio 2008** se desglosa en diferentes versiones; para nuestro uso **Visual C# 2008 Express Edition** (disponible en la dirección http://www.microsoft.com/express/vcsharp) nos sirve a la perfección; se trata de una versión gratuita.

Una vez instalado **Visual C# 2008 Express Edition**, e iniciado, debe ir al menú Archivo/Nuevo proyecto... y seleccionar Aplicación WPF. Ahora, usted acaba de iniciar el **WPF Designer**; y de forma clásica no tiene más que construir su interfaz arrastrando los elementos gráficos.

Por ejemplo, tratando de rehacer el generador de contraseñas que nos sirvió de ejemplo para ilustrar los formularios Windows Forms:



Concepción de una interfaz gráfica WPF en WPF Designer de Visual Studio 2008.

Puede ver en la parte inferior de la ventana, el código XAML correspondiente a nuestra interfaz. Una vez haya finalizado su interfaz, bastará con copiar/pegar el contenido de la ventana en nuestro script PowerShell y ¡ya tendremos la mitad ganada!

También podrá guardar su trabajo y buscar en el directorio donde lo haya guardado el archivo con la extensión .xaml para copiar su contenido.

Deberá obtener un código XAML similar a este:

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Generador de contraseñas - versión WPF" Height="395"
Width="475">
    <Grid>
        <Label Height="28" Margin="72,22,113,0" Name="label1"
VerticalAlignment="Top">Bienvenido al generador de contraseñas</Label>
        <Label Height="28" Margin="55,93,168,0" Name="label2"
VerticalAlignment="Top">La contraseña debe estar compuesta por</Label>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,71,20,0" Name="checkBox1" VerticalAlignment="Top"
Width="120" IsChecked="True">Cifras</CheckBox>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,137,20,0" Name="checkBox4" VerticalAlignment="Top"
Width="120">Otros</CheckBox>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,115,20,0" Name="checkBox3" VerticalAlignment="Top"
Width="120" IsChecked="True">Mayusculas</CheckBox>
```

```
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,93,20,0" Name="checkBox2" VerticalAlignment="Top"
Width="120">Minusculas</CheckBox>
        <Label Height="28" HorizontalAlignment="Left"
Margin="64,0,0,149" Name="label3" VerticalAlignment="Bottom"
Width="149">Número de caracteres</Label>
        <Label Height="28" HorizontalAlignment="Left"
Margin="64,0,0,110" Name="label4" VerticalAlignment="Bottom"
Width="149">Contraseña</Label>
        <Label Height="28" Margin="0,0,221,76" Name="label5"
VerticalAlignment="Bottom" HorizontalAlignment="Right"
Width="168">Complejidad de la contraseña</Label>
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,149,152" Name="textBox1" VerticalAlignment="Bottom" Width="66" />
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,37,113" Name="textBox2" VerticalAlignment="Bottom"
Width="178" />
        <ProgressBar Height="20" HorizontalAlignment="Right"
Margin="0,0,37,78" Name="progressBar1" VerticalAlignment="Bottom"
Width="178" />
        <Button Height="23" HorizontalAlignment="Left" Margin="75,0,0,37"
Name="button1" VerticalAlignment="Bottom" Width="114">Generar</Button>
        <Button Height="23" HorizontalAlignment="Right" Margin="0,0,37,37"
Name="button2" VerticalAlignment="Bottom" Width="114">Eliminar</Button>
    </Grid>
</Window>
```

Para tener un código XAML que se pueda utilizar con PowerShell, elimine simplemente la primera línea «`x:Class="WpfApplication1.Window1"`».

Sírvase señalar la declaración de los espacios de nombres XML gracias a la sintaxis siguiente: aunque la sintaxis puede parecer un poco extraña, está de acuerdo con la norma XML. Estas dos líneas son necesarias, ya que indican que el archivo se ajusta a la gramática XAML.

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Ahora, para utilizar nuestra interfaz, tenemos que crear un *Here String* que contendrá el código XAML, después habrá que convertir este último en un objeto XML. Para ello necesitaremos cuatro líneas de PowerShell. Observe el ejemplo siguiente:

```
[xml]$xaml = @'

Insertar aquí el código XAML

'@
$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)

$Form.ShowDialog() | Out-Null
```

Con este script podremos ya mostrar nuestro interfaz. Todo y que no será funcional hasta que no definamos en el script las acciones a realizar, tales como el clic en los botones.

Podríamos también externalizar de forma sencilla en un archivo de texto toda la parte XAML, cargarla con el commandlet `Get-Content`, después convertirla en XML como se muestra a continuación:

```
[XML]$xaml = Get-Content 'C:\scripts\generadorMDP.xaml'
$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)

$Form.ShowDialog() | Out-Null
```

c. Gestión de eventos

Todo está muy bien, disponemos de una hermosa interfaz gráfica, pero no hemos definido ninguna acción. Para ello, es preciso gestionar los eventos como con los **Windows Forms**. Procederemos a añadir algunas líneas de scripts adicionales.

En primer lugar debemos ver como conectarnos a los objetos del formulario. Tenga cuidado ya que con **WPF** los nombres de los objetos son sensibles a mayúsculas y minúsculas, al contrario de lo que ocurre con los comandos y las variables PowerShell.

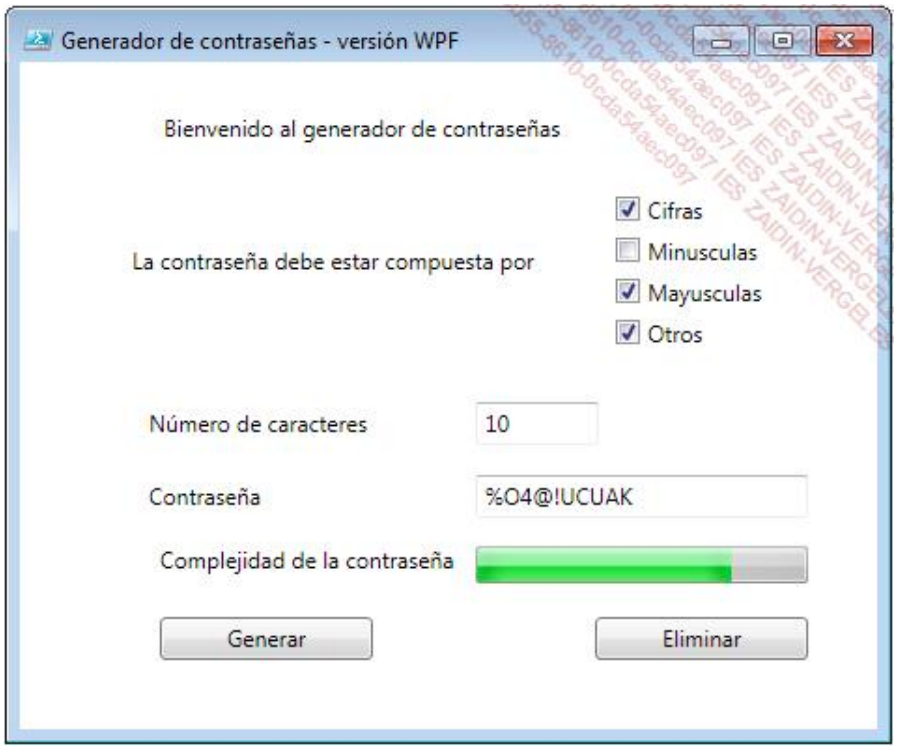
En segundo lugar, una vez conectados por ejemplo a un botón, hay que añadir a este último un gestor de evento como «add_click». Éste, como su nombre indica, reacciona en caso de pulsarlo con el ratón. En nuestro ejemplo, en caso de pulsar en el botón **Generar**, iniciaremos la generación de la contraseña. Y en caso de pulsar el botón **Salir**, entonces se cerrará el formulario.

Ejemplo:

```
$btnSalir = $form.FindName('button2')
$btnSalir.add_click({  $Form.close()  })
```

Estas pocas líneas de script se insertarán justo antes de mostrar el formulario, es decir justo antes de llamar el método `ShowDialog()`.

Ya tenemos entonces nuestro generador de contraseña en versión **WPF**; podrá observar que no hay prácticamente ninguna diferencia visible en términos gráficos en comparación a la versión realizada con los **Windows Forms**:



Generador de contraseñas en versión WPF

Veamos a continuación el script completo del generador de contraseñas revisado:

```
# WPF_PWDGenerator.ps1
[xml]$XAML = @'
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Generador de contraseñas - versión WPF" Height="395"
Width="475">
    <Grid>
        <Label Height="28" Margin="72,22,113,0" Name="label1"
VerticalAlignment="Top">Bienvenido al generador de contraseñas</Label>
        <Label Height="28" Margin="55,93,168,0" Name="label2"
VerticalAlignment="Top">La contraseña debe estar compuesta por</Label>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,71,20,0" Name="checkBox1" VerticalAlignment="Top"
Width="120" IsChecked="True">Cifras</CheckBox>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,137,20,0" Name="checkBox4" VerticalAlignment="Top"
Width="120">Otros</CheckBox>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,115,20,0" Name="checkBox3" VerticalAlignment="Top" Width="120"
IsChecked="True">Mayúsculas</CheckBox>
        <CheckBox Height="16" HorizontalAlignment="Right"
Margin="0,93,20,0" Name="checkBox2" VerticalAlignment="Top"
Width="120">Minúsculas</CheckBox>
        <Label Height="28" HorizontalAlignment="Left" Margin="64,0,0,149"
Name="label3" VerticalAlignment="Bottom"
Width="149">Número de caracteres</Label>
        <Label Height="28" HorizontalAlignment="Left" Margin="64,0,0,110"
Name="label4" VerticalAlignment="Bottom" Width="149">Contraseña</Label>
        <Label Height="28" Margin="0,0,221,76" Name="label5"
VerticalAlignment="Bottom" HorizontalAlignment="Right"
Width="168">Complejidad de la contraseña</Label>
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,149,152" Name="textBox1" VerticalAlignment="Bottom" Width="66" />
        <TextBox Height="23" HorizontalAlignment="Right"
Margin="0,0,37,113" Name="textBox2" VerticalAlignment="Bottom"
Width="178" />
        <ProgressBar Height="20" HorizontalAlignment="Right"
Margin="0,0,37,78" Name="progressBar1" VerticalAlignment="Bottom"
Width="178" />
        <Button Height="23" HorizontalAlignment="Left" Margin="75,0,0,37"
Name="button1" VerticalAlignment="Bottom" Width="114">Generar</Button>
        <Button Height="23" HorizontalAlignment="Right" Margin="0,0,37,37"
Name="button2" VerticalAlignment="Bottom" Width="114">Eliminar</Button>
    </Grid>
</Window>
'@

$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)

# Atención: los nombres de los objetos a buscar son sensibles a mayúsculas
```

```
y minúsculas
$textBox_Nb_caracteres = $form.FindName('textBox1')
$textBox_resultado = $form.FindName('textBox2')
$checkBox_cifras = $form.FindName('checkBox1')
$checkBox_minusculas = $form.FindName('checkBox2')
$checkBox_mayusculas = $form.FindName('checkBox3')
$checkBox_otros = $form.FindName('checkBox4')
$btnGenerar = $form.FindName('button1')
$btnEliminar = $form.FindName('button2')
$progressBar =  $form.FindName('progressBar1')

$btnGenerar.add_click({
   [int]$len = $textBox_Nb_caracteres.get_text()
   $textBox_resultado.Text = ''
   $complex = 0
   $progressBar.Value = 0
   [string]$chars = ''

  if ($checkBox_cifras.isChecked)
       {$chars += '0123456789';$complex += 1}
  if ($checkBox_mayusculas.isChecked)
       {$chars += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';$complex += 1}
  if ($checkBox_minusculas.isChecked)
       {$chars += 'abcdefghijklmnopqrstuvwxyz';$complex += 1}
  if ($checkBox_otros.isChecked)
       {$chars += '_!@#$%';$complex += 1}

  if($chars -ne ''){
    $bytes = New-Object System.Byte[] $len
    $rnd = New-Object System.Security.Cryptography.RNGCryptoServiceProvider
    $rnd.GetBytes($bytes)
    $resultado = ''
    for( $i=0; $i -lt $len; $i++ )
    {
       $resultado += $chars[ $bytes[$i] % $chars.Length ]
    }
    $complex *= $(2.57*$len)
    if($complex -gt 100){ $complex = 100 }
    $progressBar.Value = $complex
    $textBox_resultado.Text = $resultado
  }

}) # fin del bloque del botón "Generar"

$btnEliminar.add_click({  $Form.close()  })

$Form.ShowDialog() | Out-Null
```

Una cosa sorprendentemente en comparación con la versión **Windows Forms** de este mismo script es la concisión. En efecto hemos pasado de más de 200 líneas de código a menos de 80. Un número de líneas dividido por dos y medio.

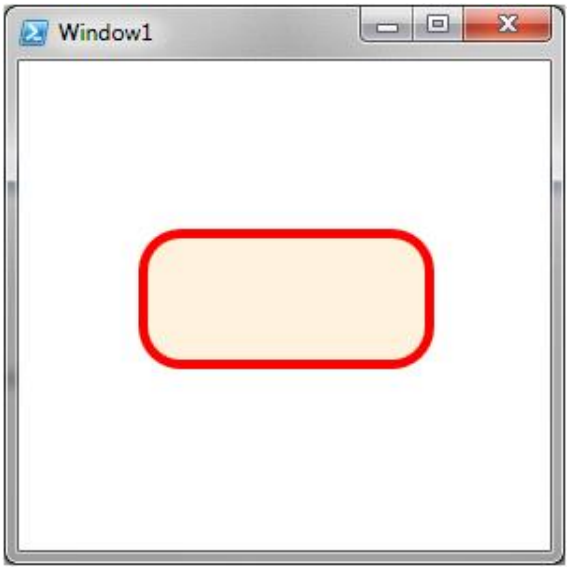
Esto no sólo hace que el script sea más legible sino que permite sobre todo disociar la parte de diseño de la interfaz, de la parte de gestión. Así, si un día precisa rediseñar completamente la interfaz, sólo necesitará copiar/pegar el código XAML en su editor XAML preferido, modificarlo y pegar el nuevo código en el script. Si no modifica el nombre de los objetos, todo funcionará a la perfección. ¿No es magnífico?

d. Creación de formas gráficas sencillas

WPF posee un gran número de primitivas básicas para lograr formas gráficas sencillas: elipses, rectángulos, líneas, polígonos, etc.

Probemos de mostrar un rectángulo con ángulos redondeados:

```
# WPF_rectangle.ps1

[xml]$XAML = @'
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300" >
    <Grid>
      <Rectangle Margin="64,90,62,97" Name="rectangle1" Stroke="Red"
       Fill="AntiqueWhite" StrokeThickness="5" RadiusX="20" RadiusY="20" />
    </Grid>
</Window>
'@

$reader=New-Object System.Xml.XmlNodeReader $xaml
$Form=[Windows.Markup.XamlReader]::Load($reader)
$Form.ShowDialog() | Out-Null
```



*Creación de un rectángulo redondeado con **WPF***

Siempre con la idea de separar el script de la parte de presentación, hemos establecido en el script un *Here-String* conteniendo el código XAML correspondiente al rectángulo que acabamos de generar con **Visual Studio 2008 Express**.

Todo y que **WPF** es una tecnología rica, no existen lamentablemente, nativamente, formas avanzadas de tipo gráfico de barras o circular. Pero Microsoft ha pensado en todo ya de manera similar a los **MS Charts Controls** para **Windows Forms**, Microsoft proporciona (gratuitamente) el **WPF ToolKit** (descargable en la dirección: http://wpf.codeplex.com). Este último contiene todo lo necesario para realizar gráficos de calidad llamados «*charts*».

e. Realización de gráficos evolucionados con el WPF Toolkit

La realización de gráficos con **WPF** es relativamente reciente ya que las funcionalidades de «*charting*» se han incluido

en la última versión del **WPF ToolKit**, sólo a partir de Junio de 2009. Antes de eso, era necesario utilizar bibliotecas de pago de terceros, para integrar un **Windows Form** conteniendo los **MS Charts Controls** en el interior de un formulario **WPF**.

Un vez instalado **WPF Toolkit** vamos a poder verificar el ejemplo siguiente. Este último muestra en formato circular los cinco procesos con más consumo de memoria:

```
# WPF_circular.ps1
# Gráfico circular en WPF con el WPF ToolKit

$dataVisualization = 'C:\Program Files\WPF Toolkit
        \v3.5.50211.1\System.Windows.Controls.DataVisualization.Toolkit.dll'

$wpfToolkit = 'C:\Program Files\WPF Toolkit
        \v3.5.50211.1\WPFToolkit.dll'

Add-Type -Path $dataVisualization
Add-Type -Path $wpfToolkit

function ConvertTo-Hashtable
{
    param([string]$key, $value)
    Begin {
        $hash = @{}
    }
    Process {
        $thisKey = $_.$Key
        $hash.$thisKey = $_.$Value
    }
    End {
        Write-Output $hash
    }
} #ConvertTo-Hashtable

# Utilización de un Here-String que contenga el código XAML
$xamlString = @'
<Window
xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
xmlns:toolkit='http://schemas.microsoft.com/wpf/2008/toolkit'
xmlns:charting='clr-
namespace:System.Windows.Controls.DataVisualization.Charting;assembly=
System.Windows.Controls.DataVisualization.Toolkit'
Title="WPF Toolkit"
Background="LightBlue"
Height="400" Width="500" >
<charting:Chart x:Name="MyChart">
<charting:Chart.Series>
<charting:PieSeries ItemsSource="{Binding}"
DependentValuePath="Value"
IndependentValuePath="Key" />
</charting:Chart.Series>
</charting:Chart>
</Window>
'@
```

```
[xml]$xaml = $xamlString
$reader=New-Object System.Xml.XmlNodeReader $xaml
$control=[Windows.Markup.XamlReader]::Load($reader)

# Creación de una tabla de control que contenga los datos
$myHashTable = Get-Process | Sort-Object -Property PM |
Select-Object Name, PM -Last 5
$myHashTable = $myHashTable | ConvertTo-Hashtable Name PM

# Búsqueda del control que contenga la forma gráfica
$chart = $control.FindName('MyChart')
$chart.DataContext = $myHashTable
$chart.Title = 'Top 5 de los procesos con mayor consumo de memoria'

# Visualización del resultado
$null = $control.ShowDialog()
```

Gráfico en formato circular con el WPF Toolkit

El script recupera los cinco procesos con mayor consumo de memoria, los almacena en un cuadro asociativo (llamado también «tabla de control») después los transfiere a la propiedad `DataContent` del gráfico. La transferencia de datos en PowerShell es posible ya que en la parte XAML de la interfaz, hemos indicado la palabra clave «`{Binding}`».

f. La biblioteca PowerBoots

Como puede darse cuenta, la realización de gráficos necesita un cierto número líneas de código XAML y de PowerShell. Para hacer honor a la reputación de «perezosos» que tenemos los informáticos, vamos a hacer una llamada a una biblioteca, llamada «**PowerBoots**», que va a potenciar nuestra productividad limitando, naturalmente, el número de líneas de código que debemos escribir...

En el momento en que escribimos estas líneas, **PowerBoots** está disponible en versión Beta.

PowerBoots podrá imponerse sin problemas a corto plazo como LA biblioteca gráfica elegida por PowerShell.

Joel 'Jaykul' Bennet (MVP PowerShell) es la persona que ha originado este proyecto, proyecto que se puede encontrar en el sitio de la comunidad de intercambio Codeplex (http://powerboots.codeplex.com). La idea de **PowerBoots** es facilitar la creación de interfaces gráficas en PowerShell, ofreciendo una sintaxis simple. **PowerBoots** se basa en **WPF** y soporta la gestión de eventos, de threads, y muchas otras cosas. «Boots» es un juego de palabras inventado por Jaykul para hacer un guiño a la biblioteca gráfica del lenguaje **Ruby** llamada «shoes». Sorprendentemente, la mayoría de las funcionalidades de **PowerBoots** se utiliza también con PowerShell v1.

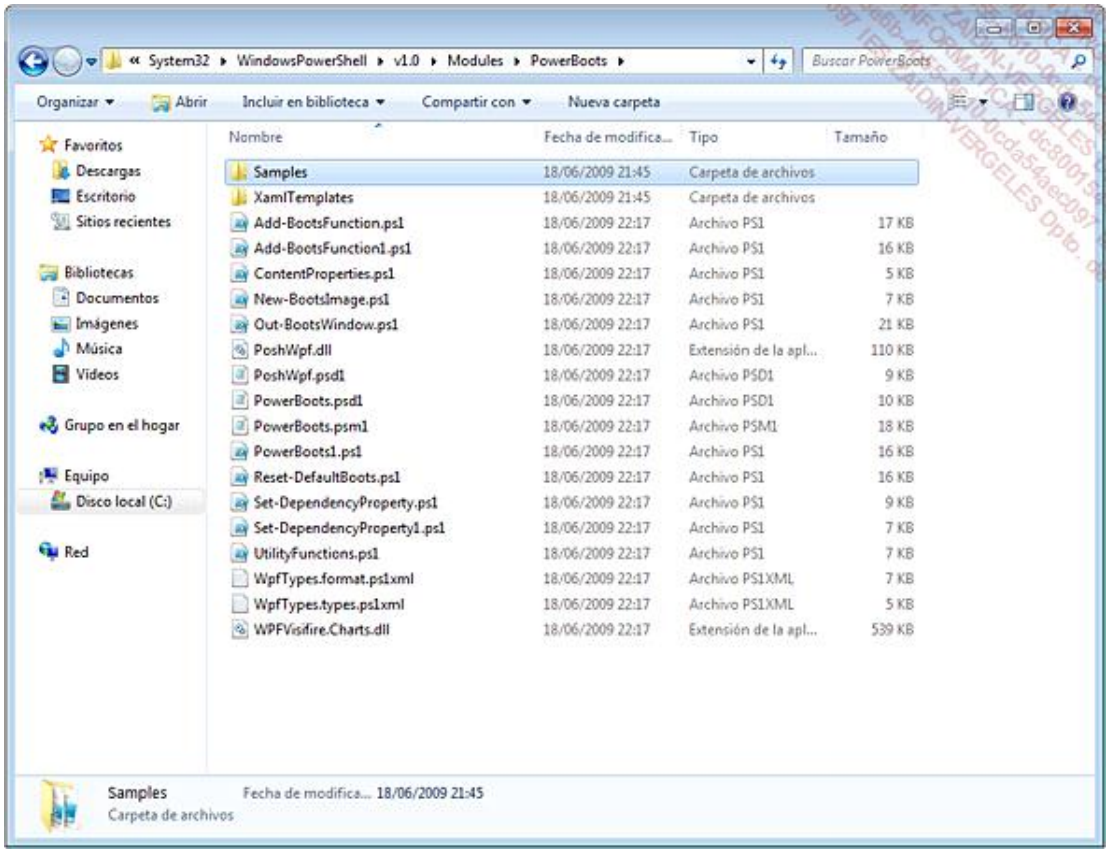
Instalación

PowerBoots se instala como un módulo.

Un vez descargado el archivo `PowerBoots.GPL.zip`, es importante desbloquearlo (pulsar el botón secundario del ratón/Propiedades y luego Desbloquear). Si usted no está familiarizado con los mecanismos de seguridad, lea el capítulo La seguridad.

Acto seguido descomprima el archivo en uno de los directorios de almacenamiento de los módulos (directorio del usuario C:\Users\Oscar\Documents\WindowsPowerShell\Modules\PowerBoots o el directorio de la máquina C:\Windows\system32\WindowsPowerShell\v1.0\Modules\PowerBoots).

Debería obtener una estructura de directorios similar a la siguiente:



Directorio de instalación de PowerBoots

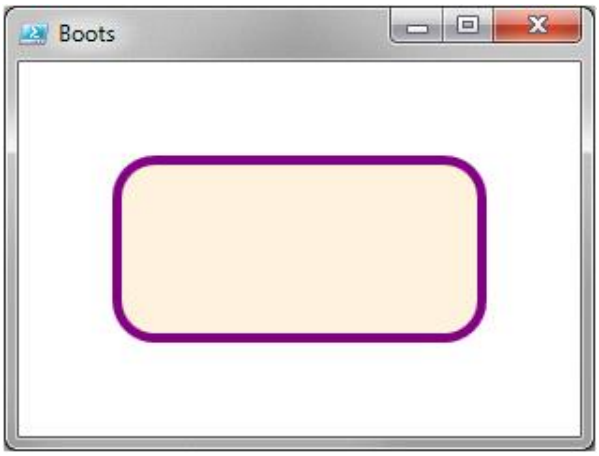
Seguidamente, abra una consola PowerShell en modo Administrador, posiciónese en el directorio de instalación de **PowerBoots** y ejecute el script `PowerBoots.ps1`. Esto creará la carpeta `Functions` y generará en su interior un gran número de scripts utilizados por la biblioteca **PowerBoots**.

Utilización de Boots

La última etapa necesaria para poder utilizar PowerBoots es cargar el módulo. Para efectuar esta etapa, ya no es necesario estar en una consola PowerShell en modo Administrador:

```
PS > Import-Module Powerboots
```

Ahora, utilizamos **PowerBoots** para crear un rectángulo:



Creación de un rectángulo redondeado con PowerBoots

```
Boots {
     Rectangle -Margin ’50,50,50,50’ -Width 200 -Height 100 -Stroke Purple `
-StrokeThickness 5 -Fill AntiqueWhite -RadiusX 20 -RadiusY 20
}
```

2-3 líneas de código para crear un rectángulo en **WPF**, ¡increíble!

Para visualizar una imagen en una ventana, es también muy sencillo:

```
Boots {
   Image -Source C:\Users\Oscar\Pictures\moto.jpg -MaxWidth 400
} -Title ’Cuando yo sea grande...’ -Async
```

Visualizar una imagen con PowerBoots

Finalmente, nuestro «clásico» gráfico de barras de los procesos con mayor consumo de memoria en 6 líneas:

```
Boots {
      Chart -MinWidth 200 -MinHeight 150 -Theme Theme3 {
```

```
        DataSeries {
        get-process| sort PM -Desc | Select -First 5 | %{
           DataPoint -AxisXLabel ($_.ProcessName) -YValue ($_.PM/1MB)
           }
        }
     }
} -Title 'Boots - Top 5 de los procesos con mayor consumo'
```

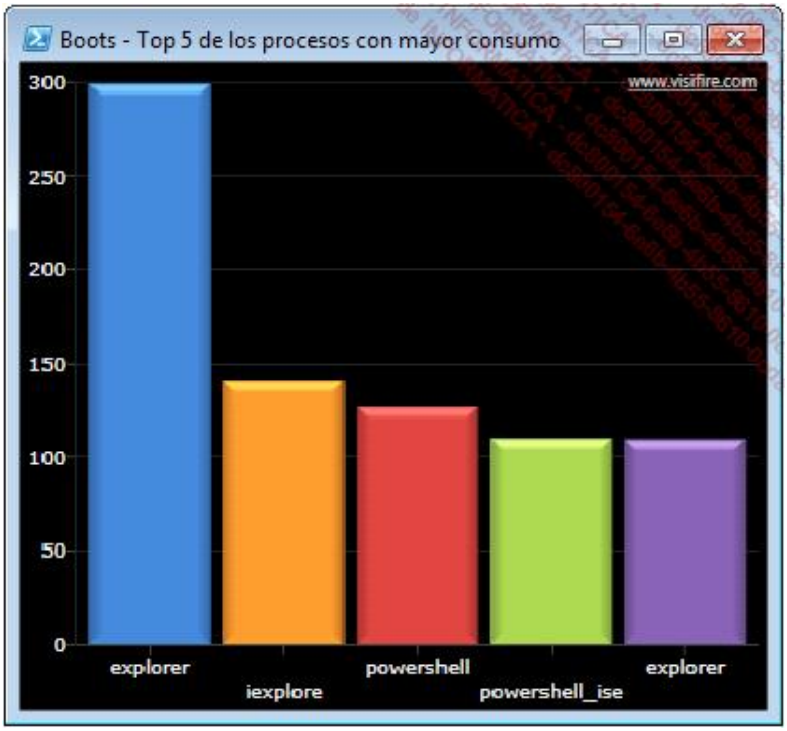


Gráfico en modo columnas con PowerBoots

4. Resumen

Como usted ha podido ver, hay una multitud de posibilidades para crear tanto gráficos sencillos como complejos con PowerShell. Existen muchos otros, como por ejemplo, los **PowerGadgets** (véase el capítulo Recursos complementarios), pero este tema es tan amplio que podría ser objeto de un libro entero.

Para resumir este apartado sobre la creación de interfaces gráficas, no es posible decir cual de entre las dos tecnologías **Windows Forms** o **WPF** escogeríamos. En efecto, usted podrá realizar a nivel global lo mismo con estas dos tecnologías, y sólo su aplicación es diferente. Su elección se realizará, con toda probabilidad, en función del tiempo y los conocimientos que usted ya dispone.

WPF es una tecnología verdaderamente poderosa y su aplicación es relativamente rápida. El aspecto más interesante es el hecho de poder disociar la interfaz gráfica de la parte de negocio. Usted no debe perder nunca de vista que **WPF** es la tecnología que sucede a **Windows Forms**, por lo tanto, si es usted principiante, le recomendamos optar directamente por **WPF** sin pasar por **Windows Forms**. Dicho esto, seamos honestos, ya que como cualquier cosa nueva, encontrará escasa ayuda en Internet para poder superar los problemas que se le presenten. Lo cierto en cualquier caso, es que **WPF** representa el futuro a corto o medio plazo...

Una última cosa: no olvide que para utilizar **WPF**, es necesario PowerShell v2. La versión 2 de PowerShell es relativamente reciente y es probable que todavía esté poco extendida. Por consiguiente, si debe proporcionar un script con interfaz gráfica que deba ejecutarse en un gran número de computadoras, será preferible utilizar el tándem PowerShell v1/**Windows Forms**.