TEMA

Tema 3: Creación de componentes visuales

Desarrollo de aplicaciones multiplataforma

Desarrollo de interfaces

Autoría: David Forteza

Linkia FP

Tema 3: Creación de componentes visuales

¿Qué aprenderás?

- Creación de nuevos componentes visuales
- Definir las propiedades y valores de los nuevos componentes
- Crear eventos personalizados en los controles
- Empaquetar y publicar componentes creados
- Crear aplicaciones usando componentes personalizados

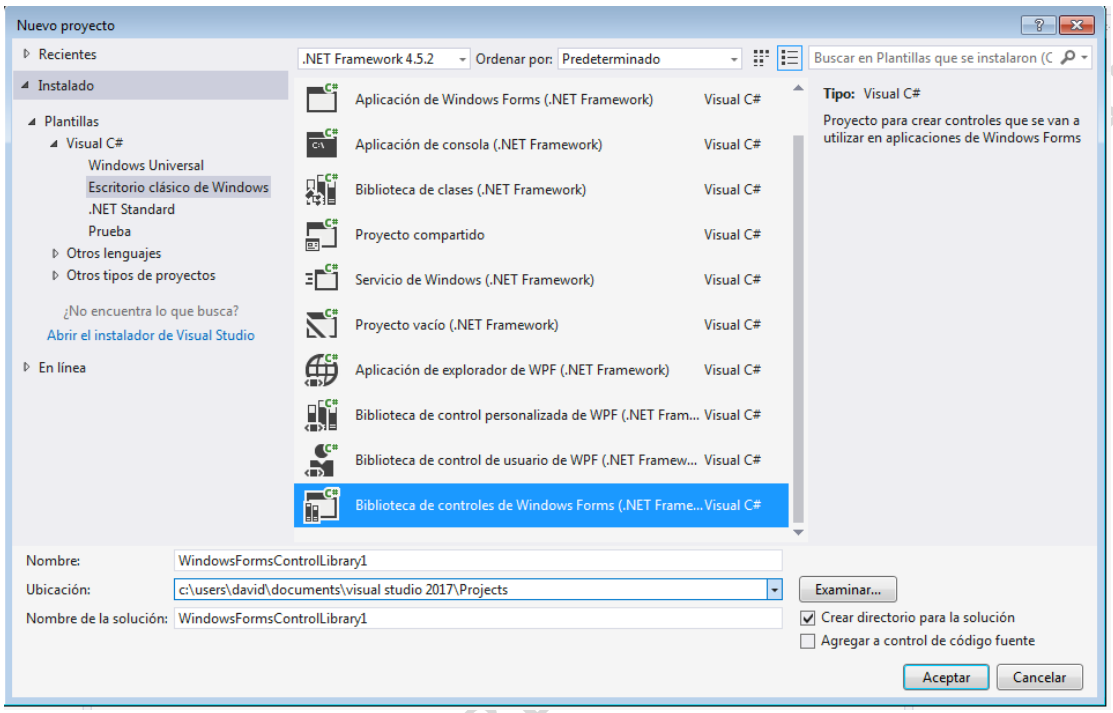
¿Sabías que…?

- Se pueden crear bibliotecas de controles para poder usarlos en cualquier proyecto.

3.1. Diseño de Controles

3.1.1. Creación de bibliotecas de controles

Para crear controles personalizados en Visual Studio .NET hay que crear un tipo especial de proyecto llamado **Biblioteca de controles de Windows Forms**.



Este tipo de proyectos presenta dos peculiaridades con respecto a los proyectos que hemos utilizado hasta este momento:

- No generan un archivo ejecutable (.exe), sino una librería dinámica (.dll) que podemos incluir en otros proyectos.
- No generan un formulario (o ventana), sino una superficie de trabajo vacía, ya que lo que desarrollamos es un control y no un interfaz.

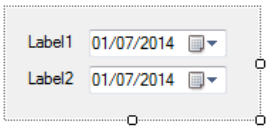
Una vez creado el proyecto podremos desarrollar en él tantos controles como queramos.

3.1.2. Creación de controles por agrupación

La primera de las técnicas de diseño de controles que vamos a ver es probablemente la más sencilla y rápida de todas ellas, ya que se basa en utilizar los controles ya existentes. Para ello crearemos grupos de uno o más controles reprogramando sus funcionalidades. El hecho de partir de los controles existentes facilita su desarrollo, pero limita sus capacidades.

Para ilustrar esta técnica lo haremos a través del desarrollo de un control que nos permita conocer el número de días que hay entre dos fechas. Este control nos sería de gran utilidad en el diseño del interfaz de una aplicación para gestionar las reservas en un hotel, por ejemplo. Para desarrollar este control utilizaremos dos controles del tipo **dateTimePicker** (control que nos permite elegir una fecha) y sincronizarlos.

Creamos el proyecto para el nuevo control y lo llamamos **RangoDias**. Añadimos los dos controles **dateTimePicker** junto con dos **Label** delante de ellos. Modificaremos la propiedad **Format** de ambos selectores de fecha de **Long** a **Short**.



Aspecto del control

Luego ponemos un nombre adecuado a cada elemento: fecha_inicio, fecha_fin, etiqueta_inicio, etiqueta_fin. Cuando ejecutamos el proyecto podemos ver que ambos controles funcionan correctamente, aunque de forma absolutamente independiente el uno del otro. Para sincronizar su funcionamiento lo que haremos será reprogramar los eventos **ValueChanged** de ambos controles.

```
private void fecha_inicio_ValueChanged(object sender, EventArgs e)
{
  if (fecha_fin.Value <= fecha_inicio.Value)
  {
    fecha_fin.Value = fecha_inicio.Value.AddDays(1);
  }
}


private void fecha_fin_ValueChanged(object sender, EventArgs e)
{
  if (fecha_incio.Value >= fecha_fin.Value)
  {
    fecha_incio.Value = fecha_fin.Value.AddDays(-11);
  }
}
```

Si cambiamos cualquiera de las fechas, el programa se asegura que haya al menos un día de diferencia entre ambas. En caso contrario, ajusta la fecha que no hemos tocado para que eso sea así.

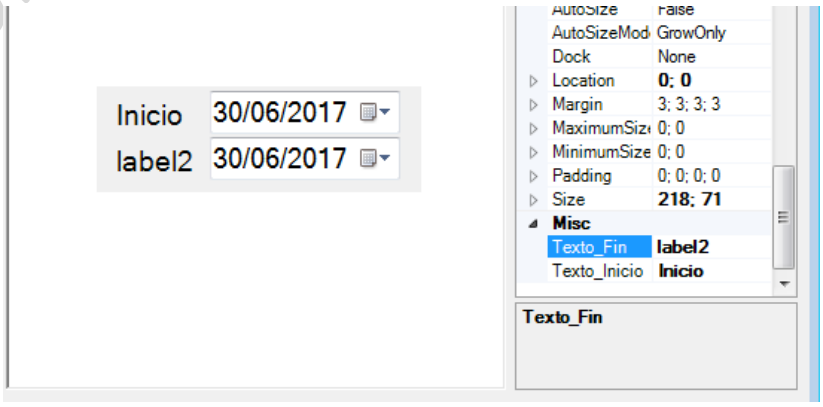
3.1.3. Creación de propiedades

Ahora que ya tenemos las fechas "sincronizadas" debemos proporcionar propiedades al usuario para que pueda configurar el control a las necesidades de su interfaz. Lo primero que haremos será crear dos propiedades que nos permitan configurar el texto de las etiquetas. Para ello, debemos definir dos propiedades de tipo **String**. Hay que tener en cuenta que las propiedades no son mas que **getters** y **setters** clásicos para acceder a las variables internas de clase que de no existir antes debemos crear dichas variables. En nuestro caso como la variable ya existe (la propiedad **Text** de ambos **labels**) podemos declarar directamente las propiedades.

```
public String Texto_Inicio
{
  get { return etiqueta_inicio.Text; }
  set { etiqueta_inicio.Text = value; }
}
```

En la sección **Get** hay que definir lo que devolverá el control al consultar el valor de la propiedad y en la sección **Set** recibimos, a través de la variable **value** (que lógicamente comparte tipo de dato con el de la propiedad), el valor que el usuario quiere dar a la propiedad. Más tarde realizaremos las comprobaciones pertinentes y, en caso que todo sea correcto, asignaremos el valor a la variable correspondiente.

En ese caso no hay que chequear nada puesto que cualquier texto será válido. Podemos evitar que una etiqueta demasiado larga se solape con el control de selección de fecha desactivando la propiedad **AutoSize** de la etiqueta y haciéndola más larga. Ahora si ejecutamos el programa podemos comprobar en la sección **Misc** el efecto de ambas propiedades:



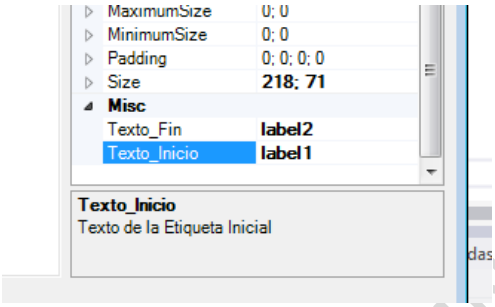
Propiedades

Podemos mejorar el interfaz del control si añadimos un valor por defecto a las propiedades, así como un texto de ayuda.

Es suficiente con añadir lo siguiente al principio de la propiedad:

```
[Description("Texto de la Etiqueta Inicial"),DefaultValue("Inicio")]
```

Con ello, el control quedará así:



Texto de Ayuda

A continuación vamos a añadir dos nuevas propiedades que permitan acceder a las fechas de los selectores de fecha, de lo contrario no podríamos manejar esos valores desde el código (solamente en tiempo de diseño).

```
[Description("Fecha Inicial")]
public DateTime Fecha_Inicio
{
  get { return fecha_inicio.Value; }
  get { fecha_inicio.Value = value; }
}
[Description("Fecha Final")]
public DateTime Fecha_Fin
{
  get { return fecha_fin.Value; }
  set { fecha_fin.Value = value; }
}
```

Para finalizar el bloque de propiedades, nos queda quizás la más importante de todas: la que contiene el número de días que hay entre ambas fechas. Como resulta lógico esta propiedad no la puede manejar el usuario ya que se calcula automáticamente a partir de los valores de los selectores de fecha; por consiguiente, el usuario solamente podrá consultar su valor. Por este motivo, esa propiedad no tiene método **Set**.

Lo que haremos es definir una variable interna de nuestro control de tipo entero y una función que calcule los días que hay entre ambas fechas y todo lo guarde en dicha variable. A continuación, la propiedad **Dias** solamente necesita acceder a la variable interna y devolver su valor. Para realizar el cálculo usaremos el objeto **TimeSpan** que gestiona periodos de tiempo.

```
private int dias=0;
public void recalcular()
{
   TimeSpan tiempo;
   tiempo = new TimeSpan(fecha_fin.Value.Ticks
      - fecha_inicio.Value.Ticks);
   días = tiempo.Days;
}
public int Dias
{
   get { return días; }
}
```

Notar que la función **recalcular** () actualiza el contenido de la variable **dias**, por este motivo hay que invocarla tras cada modificación de las fechas inicio y fin. Fijaos también que la propiedad **Días** no tiene **Set** lo que provoca que no sea una propiedad estrictamente de lectura. El usuario no puede alterar directamente su valor.

3.1.4. Creación de eventos

Ahora que ya tenemos las propiedades añadidas vamos a generar eventos personalizados para nuestros controles. En nuestro caso solo vamos a generar un evento que se activará al cambiar el valor de la variable **días.** de esta manera el programador puede reaccionar de forma automática a los cambios de dicha variable.

La generación de eventos es muy simple. Lo primero que hay que hacer es declarar el nombre del evento y los argumentos que devolverá. Para ello se usa la siguiente estructura al principio de nuestro control debajo de la declaración del espacio de nombres:

```
public delegate void CambioValor (object sender, EventArgs e);
```

A continuación, en el interior de la clase de nuestro control declaramos un evento del tipo **CambioValor** declarado antes:

```
public event CambioValor CambioDias;
```

Luego basta con lanzar el evento cuando sea necesario. En nuestro cas al final de la función que recalcula los días:

```
public void recalcular()
{
   TimeSpan tiempo;
   tiempo = new TimeSpan(fecha_fin.Value.Ticks
      - fecha_inicio.Value.Ticks);
   días = tiempo.Days;

   var evento = this.CambioDias;
   if (evento != null)
     evento(this, new EventArgs());
}
```

El resto del código no cambia. Ahora ya podemos proceder a la generación de la librería.

Lo que ocurre es que podemos querer que en los argumentos del evento **CambioDias** se pase el valor de la variable **días**. En ese caso hay que definir un nueva clase de argumentos heredando de la clase base **EventArgs**.

```
public class Argumentos : EventArgs
{
   private int dias;
   public Argumentos (int d)
   {
     this.dias = d;
   }
   public int Dias
   {
     get { return this.días; }
   }
```

Modificamos el evento delegado para reflejar esos cambios:

```
public delegate void CambioValor (object sender, Argumentos e);
```

Y para terminar modificamos la llamada al evento para pasarle el parámetro días.:

```
evento(this, new Argumentos(dias));
```

3.2. Diseño de controles desde cero

Hasta ahora hemos tomado como punto de partida controles existentes. Esto nos ahorra dos tareas:

- Programar la interacción del control con el usuario: Eventos de teclado y ratón.
- Dibujar el control.

Si en vez de partir de un control existente, partimos de cero hay que implementar el código asociado a las dos tareas anteriormente mencionadas, pero, a cambio, nos permite generar controles con un aspecto y funcionalidades completamente nuevas.

3.2.1. Dibujar el control

Lo primero que tenemos que programar es el dibujado del control. Para ello hay que sobrecargar el método **OnPaint** que tienen todos los controles y escribir en él nuestro propio código:

```
public partial class semaforo: UserControl
{
  public semaforo()
  {
    InitializeComponent();
  }
  protected override void OnPaint(PaintEventArgs e)
  {
    base.OnPaint(e);
  }
}
```

A continuación, tenemos que preparar un lienzo de dibujo asociado a nuestro control donde poder trabajar, para conseguirlo simplemente recuperamos su referencia de los argumentos del evento:

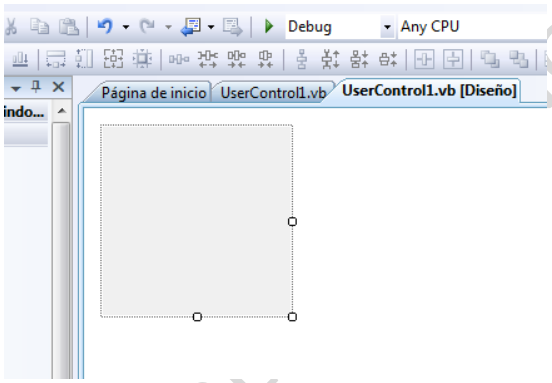
```
  protected override void OnPaint(PaintEventArgs e)
  {
    base.OnPaint(e);
    Graphics g = e.Graphics;
  }
```

Un vez hecho esto ya podemos empezar a trabajar.

Para ilustrar este capítulo, vamos a desarrollar un control que consiste en un semáforo. Para ello la fase de dibujo va a consistir en representar tres círculos uno debajo del otro de colores: rojo, amarillo y verde.

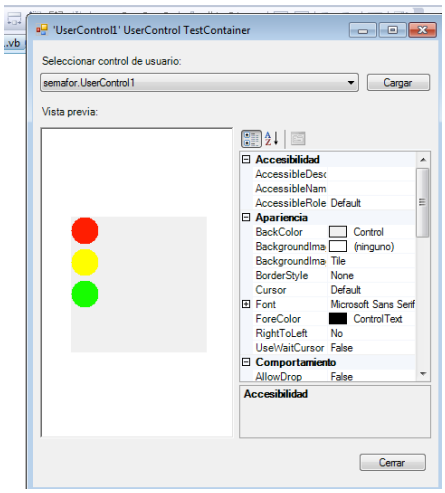
```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);
  Graphics g = e.Graphics;
  g.FillEllipse(Brushes.Red, New Rectangle(0, 0, 30, 30));
  g.FillEllipse(Brushes.Yellow, New Rectangle(0, 35, 30, 30));
  g.FillEllipse(Brushes.Lime, New Rectangle(0, 70, 30, 30));
}
```

Es importante que notéis, que ahora al cambiar a la pestaña de diseño no vemos nada dibujado.



Vista Diseño

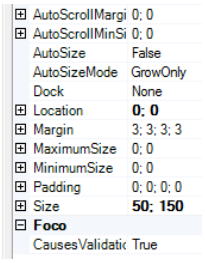
Esto ocurre porque el control se dibuja en ejecución, por lo tanto para ver cómo va a quedar dibujada, debemos iniciar nuestro proyecto. Cuando lo hacemos vemos algo así:



Control semáforo (I)

Ahora lo que observamos es que el control no tiene unas proporciones aceptables. Debería ser un rectángulo con orientación vertical de una altura 3 veces superior a la anchura, y los tres círculos deberían estar ubicados en su centro. Para ello lo primero que vamos a hacer es cambiar la propiedad **Size** de nuestro control en la vista de diseño.

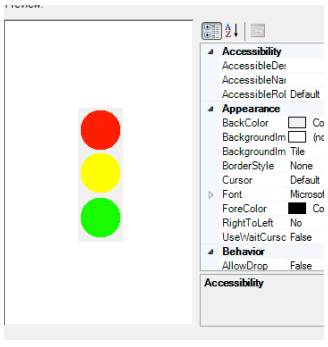
Si no hemos tocado nada en ese momento nuestro control mide 150 de ancho y 150 de alto. Ahora cambiamos el ancho por 50. Hemos solucionado la primera parte del problema, pero las luces del semáforo siguen mal ubicadas.



Control semáforo (II)

Vamos a modificar las líneas que dibujan los tres círculos para que estos salgan centrados y ocupen todo el espacio disponible.

```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);
  Graphics g = e.Graphics;
  int ancho = (int)(this.Width*0.9);
  int borde = (int)(this.Width*0.05);
  g.FillEllipse(Brushes.Red, New Rectangle(borde, borde, ancho, ancho);
  g.FillEllipse(Brushes.Yellow, New Rectangle(borde, this.Width, ancho, ancho);
  g.FillEllipse(Brushes.Lime, New Rectangle(borde, 2*this.Width, ancho, ancho);
}
```



Control semáforo (III)

Es importante que os fijéis que hemos dibujado los círculos tomando siempre como referencia el factor **this.Width** que representa la anchura actual del control; de esta forma podemos cambiar las dimensiones del control y los círculos de las luces mantendrán el mismo aspecto y proporción (siempre que el control mida 3 veces de alto por 1 de ancho).

Sabemos que nuestro semáforo no puede estar rojo, amarillo y verde a la vez, o sea, que debemos definir un atributo del control que contenga su estado actual. Como podemos tener tres estados distintos vamos a generar un tipo enumerado para definir el estado de nuestro control:

```
public enum EstadoSem
{
   Rojo,
   Amarillo,
   Verde
};
```

A continuación crearemos un atributo de tipo **EstadoSem** y lo vamos a predefinir en estado Rojo:

```
private EstadoSem estado = EstadoSem.Rojo;
```

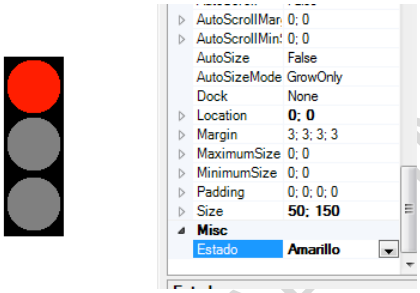
Seguidamente, modificamos el código de la función de dibujo para que "encienda" la luz que corresponde con el estado actual del semáforo. Lo mas practico es generar un array con los 3 pinceles en gris y luego encender el que toca (ya que solo puede ser uno a la vez):

```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);
  Graphics g = e.Graphics;
  g.Clear(Color.Black);
  int ancho = (int)(this.Width*0.9);
  int borde = (int)(this.Width*0.05);
  Brush[] luces = new Brush[]{
    new SolidBrush(Color.Gray),
    new SolidBrush(Color.Gray),
    new SolidBrush(Color.Gray)};
  switch(estado)
  {
     case EstadoSem.Rojo:
       luces[0]= new SolidBrush(Color.Red);
       break;
     case EstadoSem.Amarillo:
       luces[1]= new SolidBrush(Color.Yellow);
       break;
     case EstadoSem.Verde:
       luces[2]= new SolidBrush(Color.Lime);
       break;
  }
  g.FillEllipse(luces[0], New Rectangle(borde, borde, ancho, ancho);
  g.FillEllipse(luces[1], New Rectangle(borde, this.Width, ancho, ancho);
  g.FillEllipse(luces[2], New Rectangle(borde, 2*this.Width, ancho, ancho);
}
```

Lo siguiente es generar una propiedad accesible para el usuario para que este pueda cambiar el estado de nuestro semáforo.

```
[Description("Texto de la Etiqueta Inicial"),DefaultValue("Inicio")]
public EstadoSem Estado
{
  get { return estado; }
  set { estado = value; }
}
```

Si ejecutamos nuestro control y cambiamos el valor de la propiedad Estado, veremos que el semáforo no cambia de color como debería.

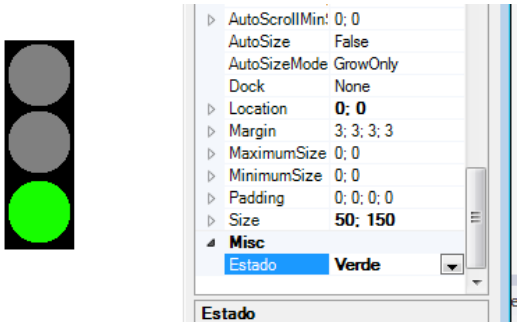


Control semáforo (IV)

Esto ocurre porque en controles cuyo dibujado controlamos nosotros, debemos invocar la función de redibujado de forma manual. Esta acción la hacemos mediante el comando **Invalidate**.

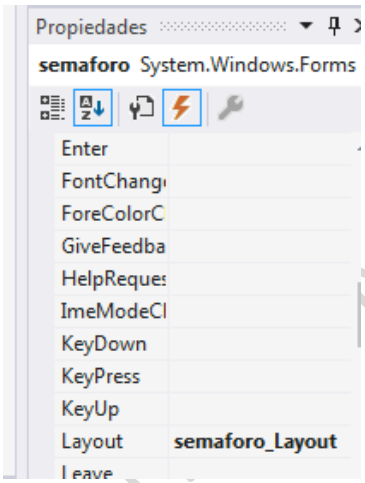
```
[Description("Texto de la Etiqueta Inicial"),DefaultValue("Inicio")]
public EstadoSem Estado
{
  get { return estado; }
  set { estado = value; Invalidate(); }
}
```

Ahora el control ya funciona perfectamente.



Control semáforo (V)

El último paso en el dibujado del control es permitir que el usuario pueda cambiar su tamaño sin alterar el proceso de dibujado. Eso resulta muy sencillo, ya que hemos tenido la precaución de dibujar todo el control tomando como base su anchura y suponiendo que mide el triple de alto que de ancho. Por lo tanto, cuando el usuario cambie la propiedad **Size** de nuestro control, lo único que debemos hacer es forzar que la anchura sea un tercio de la altura para mantener constante su relación ancho/alto. Para ello vamos a usar el evento **Layout**. Ese evento se produce cada vez que alguna propiedad del control es cambiada. En nuestro caso la propiedad que nos interesa es **Bounds** (significa contornos en inglés).



Evento Layout

El código será el siguiente.

```
private void Semaforo_Layout(object sender , LayoutEventArgs e )
{
  if (e.AffectedProperty == "Bounds")
  {
     this.Width = (int)(this.Height / 3);
     Invalidate();
  }
}
```

NOTA: No olvidar la llamada a Invalidate o de lo contrario al redimensionar el control las luces no se reajustaran al nuevo tamaño.

Con todo esto, damos por finalizado el bloque de dibujado del control.

3.2.2. Interacción con el usuario

En esta parte vamos a programar las interacciones que puede hacer el usuario con nuestro control. Lo que haremos será programar un evento de ratón que cuando se haga clic sobre una luz se active el estado correspondiente en nuestro semáforo.

```
private void Semaforo_MouseClick(object sender, MouseEventArgs e)
{
   if (e.Location.Y < this.Width)
     this.Estado = EstadoSem.Rojo;
   if ((e.Location.Y >= this.Width) && (e.Location.Y <= 2 * this.Width))
     this.Estado = EstadoSem.Amarillo;
   if (e.Location.Y > 2 * this.Width)
     this.Estado = EstadoSem.Verde;
}
```

Es importante que notéis que no hemos modificado directamente la variable **estado** sino que hemos usado la propiedad Estado de esa forma se invoca a **Invalidate** de forma automática. Esto también facilitara posteriormente la activación del evento de cambio de estado.

A continuación, para rubricar el trabajo, solo nos queda generar un evento, como vimos anteriormente, cada vez que el semáforo cambia de estado, para notificarlo y que el programador pueda reaccionar ante este. Aquí tenemos el código completo:

```
public partial class semaforo: UserControl
{

   public enum EstadoSem
   {
      Rojo,
      Amarillo,
      Verde
   };

   public class EstadoArgs : EventArgs
   {
      private EstadoSem est;
      public EstadoArgs (EstadoSem nuevoEstado)
      {
        this.est = nuevoEstado;
      }
      public EstadoSem Estado
      {
```

```
        get { return this.est; }
      }
    }
    public delegate void CamobioValor(object sender, EstadoArgs e);
    public event CamobioValor CambioEstado;


[Description("Texto de la Etiqueta Inicial"), DefaultValue("Inicio")]
    public EstadoSem Estado
    {
      get { return estado; }
      set {
        estado = value;
        Invalidate();
        var evento = this.CambioEstado;
        if (evento != null)
          evento(this, new EstadoArgs(estado));
      }
    }

    public semaforo()
    {
      InitializeComponent();
    }

    protected override void OnPaint(PaintEventArgs e)
    {
      base.OnPaint(e);
      Graphics g = e.Graphics;
      g.Clear(Color.Black);
      int ancho = (int)(this.Width*0.9);
      int borde = (int)(this.Width*0.05);
      Brush[] luces = new Brush[]{
        new SolidBrush(Color.Gray),
        new SolidBrush(Color.Gray),
        new SolidBrush(Color.Gray)};
      switch(estado)
      {
         case EstadoSem.Rojo:
           luces[0]= new SolidBrush(Color.Red);
           break;
         case EstadoSem.Amarillo:
           luces[1]= new SolidBrush(Color.Yellow);
           break;
```

```
        case EstadoSem.Verde:
          luces[2]= new SolidBrush(Color.Lime);
          break;
      }
      g.FillEllipse(luces[0], New Rectangle(borde, borde, ancho, ancho);
      g.FillEllipse(luces[1], New Rectangle(borde, this.Width, ancho, ancho);
      g.FillEllipse(luces[2], New Rectangle(borde, 2*this.Width, ancho, ancho);
    }

    private void Semaforo_Layout(object sender , LayoutEventArgs e )
    {
      if (e.AffectedProperty == "Bounds")
      {
         this.Width = (int)(this.Height / 3);
         Invalidate();
      }
    }

    private void Semaforo_MouseClick(object sender, MouseEventArgs e)
    {
       if (e.Location.Y < this.Width)
         this.Estado = EstadoSem.Rojo;
       if ((e.Location.Y >= this.Width) && (e.Location.Y <= 2 * this.Width))
         this.Estado = EstadoSem.Amarillo;
       if (e.Location.Y > 2 * this.Width)
         this.Estado = EstadoSem.Verde;
    }
}
```

3.2.3. Toques finales

En esta parte vamos a realizar los toques finales a nuestro control como elegir el icono que va a tener en la barra de herramientas así como el proceso que debemos seguir para incluirlo en otros proyectos.

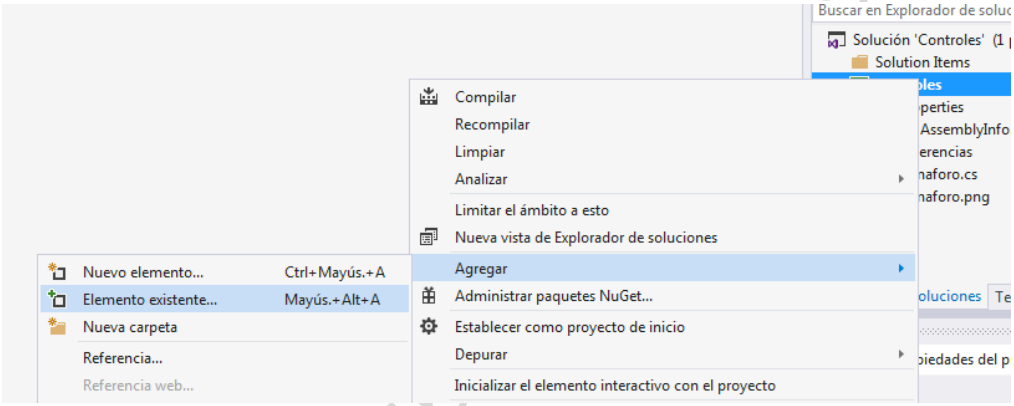
3.2.3.1. Icono personalizado

Si damos el control por terminado este aparecerá en la barra de herramientas con un icono genérico, por lo que si diseñamos vario controles va a ser difícil distinguirlos de forma rápida. Para ello lo mejor es personalizar dicho icono.

El primer paso, como es lógico será buscar y preparar el icono que queremos utilizar. En principio vale cualquiera, pero hay que tener en cuenta algunos aspectos:

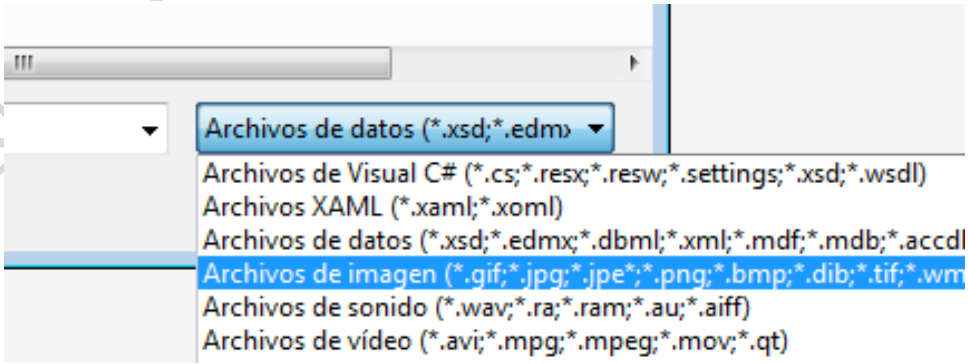
- Los formatos gif y png permiten fondos transparentes por lo que son los idóneos. Aunque otros formatos como bmp o jpg también se pueden usar
- El icono va a ser escalado a un tamaño de 16x16 en la paleta de herramientas por lo que es importante que el icono no sea muy rico en detalles purés esos se perderán.
- El número de colores también se verá reducido con lo que es bueno usar color planos sin demasiados gradados de tono.

Una vez preparado el icono hay que agregarlo a nuestro control. Para ello hacemos clic con el botón secundario del ratón en el nombre del control en el cuadro de explorador de soluciones y seleccionamos ***Agregar*** → ***Elemento Existente*** del menú contextual.



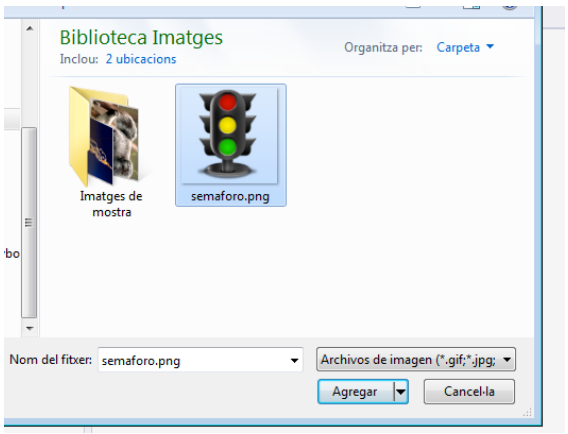
Agregando Icono

A continuación en la ventana de selección de archivo nos aseguramos de elegir el filtro para seleccionar archivos de imagen.



Filtros disponibles

Luego navegamos por el sistema de archivos hasta localizar el icono que hemos dejado preparado. En nuestro caso la imagen de un semáforo.

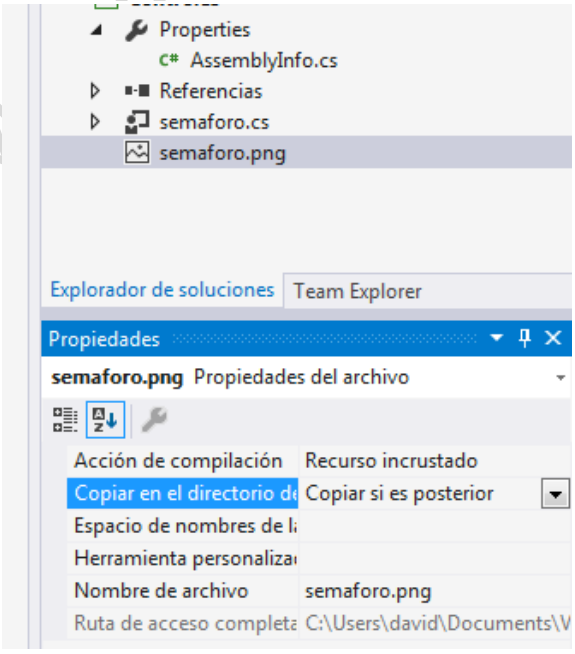


Icono elegido

Ahora podemos observar el nuevo elemento en la estructura de nuestro proyecto.

El paso siguiente es muy importante y muchas veces se olvida. Al compilar el proyecto los archivos de imagen no e incluyen dentro del archivo final por lo que si otra persona usara el icono que hemos diseñado no vería la imagen del semáforo puesto que el no la tiene. Por lo que debemos visualizar las propiedades del archivo de imagen y asegurarnos de cambiar el desplegable llamado ***Acción de Compilación*** y elegir la opción ***Recurso Incrustado***.

Con esto lo que logramos es que la imagen quede incluida dentro del archivo compilado por lo que no es necesario que el usuario que vaya a utilizar el control tenga el icono en su PC.

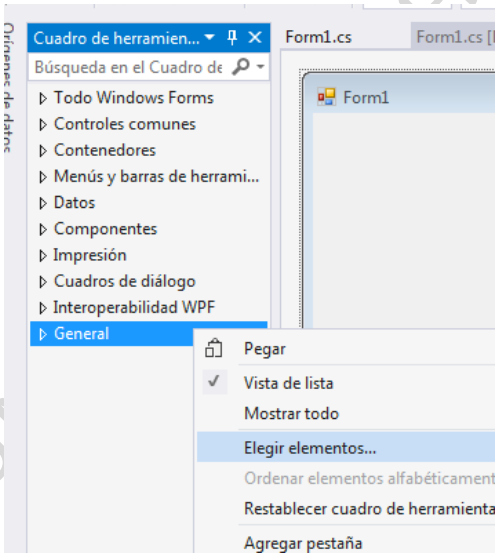


Incrustar Recurso

Ahora todo lo que queda es modificar el código del control para que se use la nueva imagen añadida. Basta con agregar una línea al principio de nuestro control:

```
[ToolboxBitmap(typeof(semaforo),"semáforo.png")]
public partial class semaforo: UserControl
{
...
```

Con eso ya hemos terminado. Por lo que compilamos la solución y cerramos en proyecto.

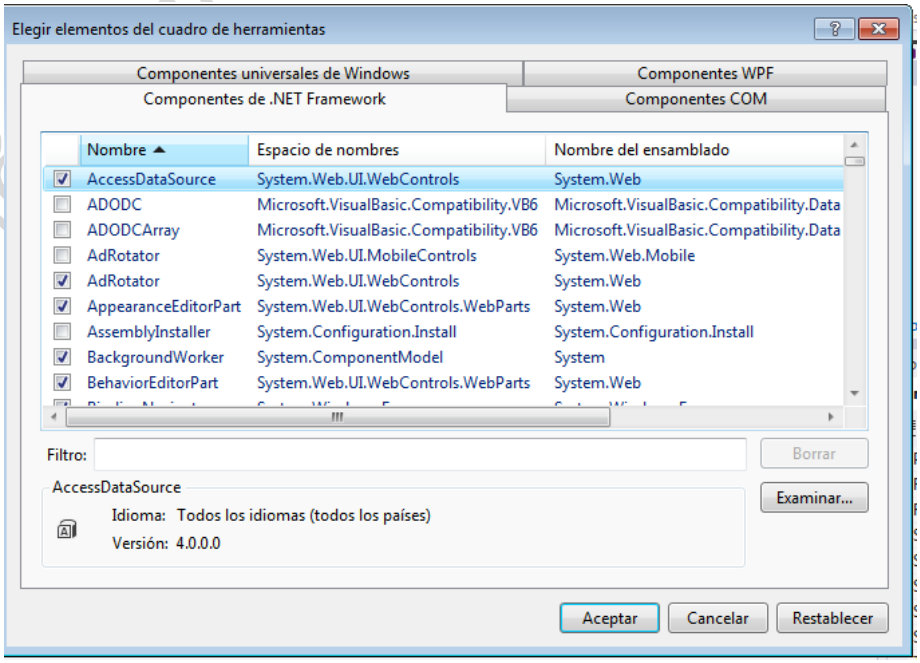


Nuevos controles

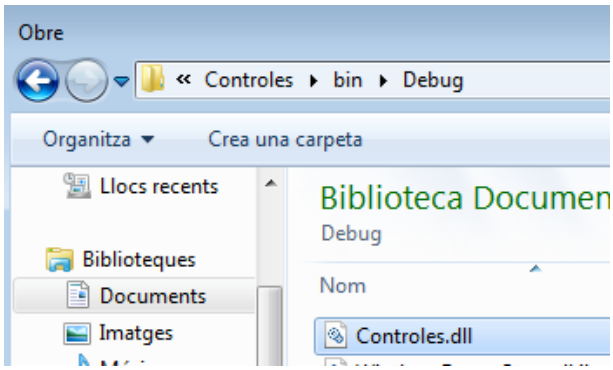
3.2.3.2. Agregar un control

Para utilizar un control personalizado lo primero que debemos hacer es añadirlo a la paleta de herramientas. Para ello debemos elegir la sección en la queremos colocarlo. En nuestro caso usaremos la sección ***General***. Luego desplegamos el menú contextual y seleccionamos la opción **Elegir Elementos**.

Ahora nos aparece una ventana con múltiples solapas. Nos aseguramos que tenemos elegida **Componentes de .NET Framework**. Pulsamos el botón **Examinar**.

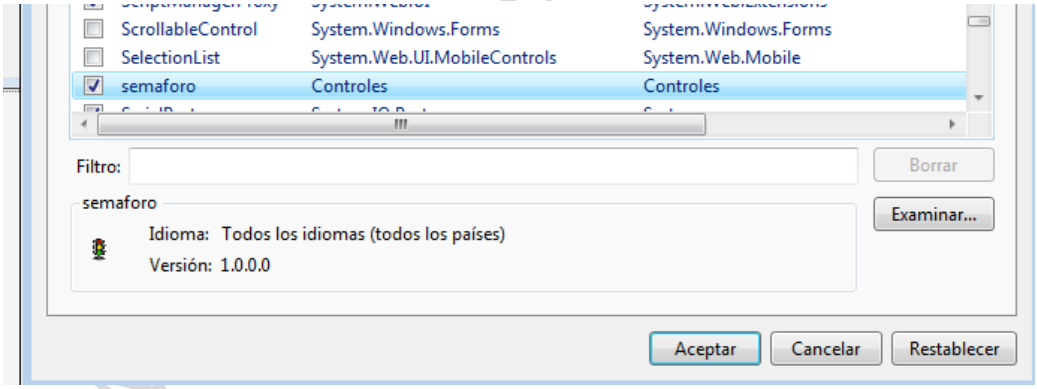


Luego nos desplazamos por el sistema de archivos para localizar el archivo compilado que contiene nuestro icono. En este punto es importante señalar que los proyectos de controles no generan ficheros .**exe** sino librerías .**dll**.



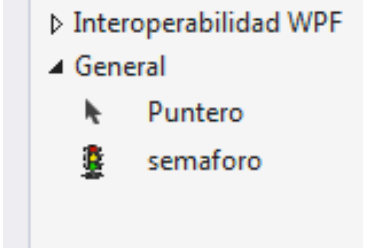
Archivo dll

Para localizar el archivo nos dirigimos a la carpeta del proyecto y dentro encontraremos un directorio llamado **bin** que contiene a su vez a otro llamado **Debug**. En el debemos encontrar el archivo que buscamos. Una vez elegido en la ventana anterior vemos resaltados los nuevos controles añadidos:



Control Semaforo

Nos aseguramos que la casilla de verificación esta marcada y aceptamos. Ahora ya podemos ver nuestro control en la paleta de herramientas.



Conceptos clave

- **Adaptación:** En este tercer tema, hemos aprendido a crear nuevos controles mejor adaptados a las necesidades de nuestra interfaz.
- **Propiedades:** También hemos aprendido a generar las propiedades y eventos necesarios para el óptimo funcionammiento del control.

Test de autoevaluación

1. Las bibliotecas de controles...
    a. Son archivos .lib
    b. Son archivos .dll
    c. Son archivos .exe
    d. Son archivos .sln

2. Sobre las propiedades:
    a. Deben tener siempre ambos métodos Get y Set.
    b. Pueden tener solamente método Get si son de solo lectura.
    c. Los métodos Get y Set se pueden generar de forma automatizada.
    d. Los métodos Get y Set son públicos.

3. Cuál es el método que nos permite definir como debe dibujarse un control:
    a. OnPaint.
    b. OnDraw.
    c. Paint.
    d. Show.

4. Cuál de los siguientes métodos debemos utilizar para volver a dibujar un control cuyas propiedades han camiado.
    a. Refresh
    b. Repaint
    c. Redraw
    d. Invalidate

5. Cómo se llama la propiedad que permite establecer el icono de un control el la paleta de componentes.
    a. IconTool.
    b. Icon.
    c. ToolBoxBitmap.
    d. Image.

6. Para asignar una imagen como icono a un control debemos:
    a. Especificar la ruta de disco absoluta a la imagen.
    b. Añadirla a una biblioteca de iconos.
    c. Colocarla en la misma carpeta que el ejecutable del control
    d. Añadirla al proyecto como recurso incrustado.

Ponlo en práctica

La actividad consiste en implementar un nuevo control partiendo de los existentes. El control será un potenciómetro que permitirá monitorizar un valor mediante una serie de elementos luminosos cuyo color simbolizará lo cerca que nos encontramos del valor máximo. Vamos a crear un grupo de 10 elementos luminosos (los seis primeros verdes, 3 amarillos y el último en rojo). Definiremos 3 propiedades:

- Max: Valor máximo del potenciómetro.
- Min: Valor mínimo del potenciómetro.
- Valor: Valor actual del potenciómetro.

En función del valor hay que iluminar los elementos correspondientes en proporción.

Se pueden utilizar todos los controles estudiados para implementar el potenciómetro.

Solucionario

Test de autoevaluación

1. Las bibliotecas de controles...
    a. Son archivos .lib
    b. Son archivos .dll
    c. Son archivos .exe
    d. Son archivos .sln

2. Sobre las propiedades:
    a. Deben tener siempre ambos métodos Get y Set.
    b. Pueden tener solamente método Get si son de solo lectura.
    c. Los métodos Get y Set se pueden generar de forma automatizada.
    d. Los métodos Get y Set son públicos.

3. Cuál es el método que nos permite definir como debe dibujarse un control:
    a. OnPaint.
    b. OnDraw.
    c. Paint.
    d. Show.

4. Cuál de los siguientes métodos debemos utilizar para volver a dibujar un control cuyas propiedades han camiado.
    a. Refresh
    b. Repaint
    c. Redraw
    d. Invalidate

5. Cómo se llama la propiedad que permite establecer el icono de un control el la paleta de componentes.
    a. IconTool.
    b. Icon.
    c. ToolBoxBitmap.
    d. Image.

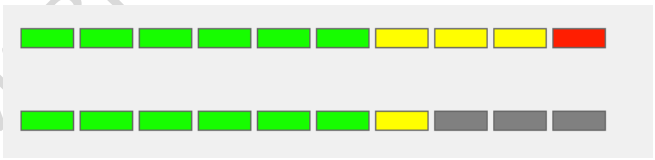
6. Para asignar una imagen como icono a un control debemos:
    a. Especificar la ruta de disco absoluta a la imagen.
    b. Añadirla a una biblioteca de iconos.
    c. Colocarla en la misma carpeta que el ejecutable del control
    d. **Añadirla al proyecto como recurso incrustado.**

Ponlo en práctica

La actividad consiste en implementar un nuevo control partiendo de los existentes. El control será un potenciómetro que permitirá monitorizar un valor mediante una serie de elementos luminosos cuyo color simbolizará lo cerca que nos encontramos del valor máximo. Vamos a crear un grupo de 10 elementos luminosos (los seis primeros verdes, 3 amarillos y el último en rojo). Definiremos 3 propiedades:

- Max: Valor máximo del potenciómetro.
- Min: Valor mínimo del potenciómetro.
- Valor: Valor actual del potenciómetro.

En función del valor hay que iluminar los elementos correspondientes en proporción.



Se pueden utilizar todos los controles estudiados para implementar el potenciómetro.

Los solucionarios están disponibles en la versión interactiva del aula.