COPIA LITERAL

Extracto del libro “GAMBAS, programación visual con Software Libre”, de la editorial EDIT LIN EDITORIAL S.L., cuyos autores son Daniel Campos Fernández y José Luis Redrejo.

LICENCIA DE ESTE DOCUMENTO

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia, añadiendo, si es copia literal, la mención “copia literal”. Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expuestos en esta licencia.

8. Herencia

8. Herencia
8. Herencia
8. Herencia

8.1 Lenguajes orientados a objetos y herencia

Ya hemos visto una breve introducción al mundo de la programación orientada a objetos. Podemos decir que un objeto es una herramienta, por ejemplo, un túnel de lavado. La clase son los planos de esa herramienta, en los cuales están descritos los distintos cepillos giratorios, los conductos de agua y detergente, los secadores de aire caliente, los distintos sensores, el proceso automático paso a paso desde la entrada del coche hasta su salida, etc.

Cuando creamos un objeto, el sistema nos devuelve una herramienta, en este caso, un túnel de lavado dispuesto para funcionar, que se ha creado a partir de los planos que nosotros habíamos escrito en la clase.

Ahora bien, supongamos que como vendedores de túneles de lavado deseamos innovar y añadir una fase de encerado a nuestro túnel. Venderemos máquinas baratas sin encerado y otras más caras con esta funcionalidad añadida. La primera opción sería ponerse manos a la obra con nuevos planos desde cero, pero esto es costoso en tiempo y dinero. Lo mejor sería tomar los planos ya existentes y, a partir de ellos, añadir la nueva función.

En esto consiste precisamente la herencia: partimos de una clase ya escrita, probablemente grande y con mucha funcionalidad, y a partir de ella creamos otra clase (otros planos) que tiene el mismo rendimiento del original, más nuevas funciones que añadimos, la cual *hereda*, de ahí viene el término, la funcionalidad de su clase padre.

Además de añadir nuevas funcionalidades, podremos modificar también el comportamiento de las ya existentes para adaptarlas a la nueva *máquina a crear*, por ejemplo, los cepillos del túnel de lavado con cera tendrán que ser de un material que no se disuelva con los productos químicos que componen nuestra cera.

En resumen, vamos a explicar cómo crear clases que provienen de otras, que se comportan en primera instancia como la original y que añaden nuevas funcionalidades o modifican algunas de las existentes, eliminando así la necesidad de escribir las mismas líneas de código una y otra vez.

8. 2 Conceptos necesarios

La clase padre

Empecemos creando los planos de la máquina original, es decir, creemos una clase *padre*. Vamos a escribir una clase a partir de la cual crearemos objetos. Dichos objetos almacenan varios números reales y, una vez introducidos, nos devuelve la media aritmética de todos ellos.

Ahora, vamos a crear un nuevo proyecto de consola, lo llamamos **EstudioHerencia** y dentro de él un módulo de inicio llamado *modMain*, que nos servirá para las pruebas.

Además, creamos también una clase, llamada *ClsCalculo*, en la que escribiremos el código principal a estudiar.

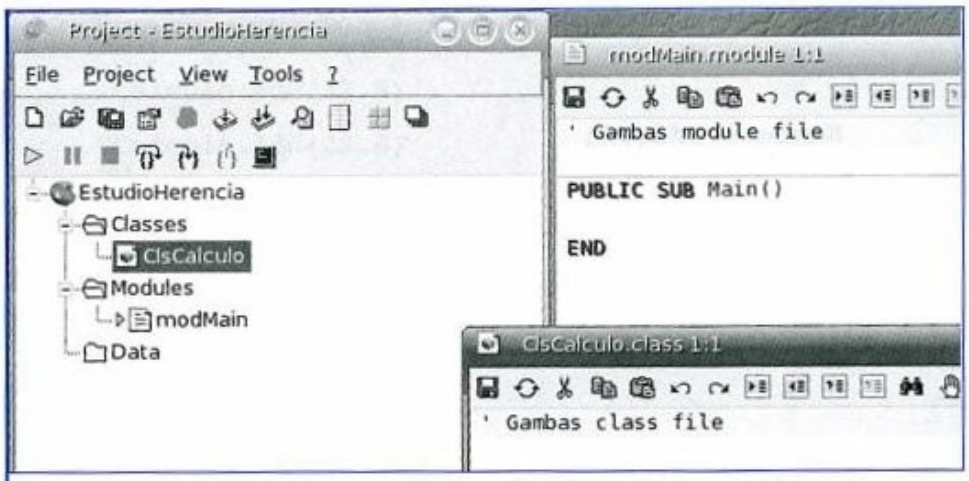

Figura 1. Proyecto nuevo de consola **EstudioHerencia**.

Dentro de *ClsCalculo* escribiremos el código necesario: tendremos una matriz privada de números reales, donde añadimos cada uno de los números a calcular y cuatro métodos: el primero, _New, es el constructor que sirve para inicializar la matriz; el segundo, _Free, libera la matriz al destruir el objeto; el tercero, **Add**, añade un nuevo valor a la serie; y el cuarto, **Average**, calcula la media aritmética de los números almacenados:

```
' Gambas class file
PUBLIC _Numbers AS Float[]

PUBLIC SUB _new()

  _Numbers = NEW Float[]

END
```

Sin embargo, los métodos públicos que comienzan por el signo de subrayado no se muestran nunca en el sistema de autocompletado ni en el sistema de ayuda, cuando creamos componentes.

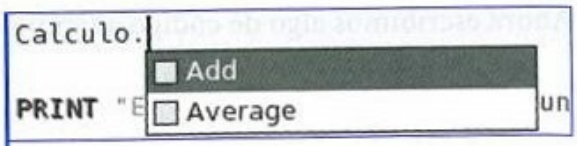

Figura 3. Variables públicas no visibles.

Puesto que en Gambas no existe el concepto de C++ y otros lenguajes de propiedad, variable o función *Protegida* (accesible sólo desde la clase y sus clases hijas), esta funcionalidad nos permite declarar variables públicas no visibles desde el código principal, manteniendo así una interfaz más coherente con nuestros propósitos.

Los nombres de variables, propiedades o funciones que comienzan con un subrayado, no se muestran ni en el sistema de autocompletado, ni en el de ayuda de Gambas.

NOTA PARA PROGRAMADORES DE C++

En Gambas existen propiedades, métodos y variables privadas o públicas, pero no protegidas.

La clase hija. Palabra clave INHERITS

Supongamos ahora que deseamos tener una nueva clase que se comporte como la inicial, pero tenga una propiedad adicional, de sólo lectura, que nos devuelve el número de elementos que hemos almacenado.

Crearemos, entonces, una nueva clase llamada *ClsCalculo2*, en la cual introduciremos al inicio la palabra clave *INHERITS* seguida del nombre de la clase *padre* (ver Figura 4).

Esto es todo lo necesario para tener una clase que hereda todas las propiedades de su clase *padre*. Probemos a modificar el código del método **Main**() de modo que creemos un objeto de la segunda clase, en lugar de la original.

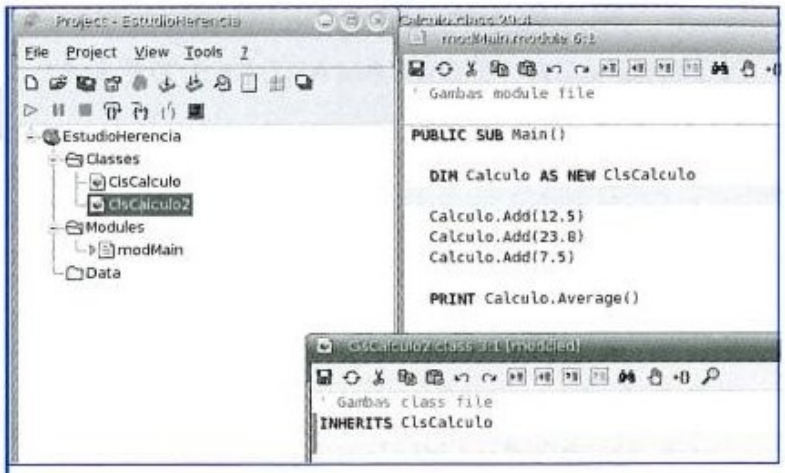

Figura 4. Nueva clase *ClsCalculo2*.

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo2

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT Calculo.Average()
END
```

Si ejecutamos ahora el programa, observaremos que el resultado *14.6* es exactamente el mismo: la nueva clase ya dispone de todos los métodos, propiedades y eventos de la original, sin necesidad de escribir el código que los implemente.

Para escribir una clase que hereda las características de una clase *padre*, tecleamos **INHERITS** seguido del nombre de la clase *padre*, al inicio del código de la nueva clase:

```
' Gambas class file
INHERITS ClsCalculo
```

NOTA PARA PROGRAMADORES DE C++

En Gambas cada clase *hija* tiene una sola clase *padre*, no existe el concepto de herencia múltiple.

□□□□□ Extendiendo funcionalidades. Palabra clave SUPER

La nueva clase *ClsCalculo* tiene poco interés por ahora: hace lo mismo que la original. Añadamos entonces en la clase *hija* la nueva propiedad que nos devuelve el número de elementos almacenados:

```
' Gambas class file
INHERITS ClsCalculo

PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END
```

Estamos empleando una nueva palabra clave por primera vez, **SUPER**. Sabemos que cuando nos referimos a un objeto podemos utilizar, bien el nombre del objeto, o bien la palabra clave *ME* cuando nos referimos al objeto actual dentro de la propia clase. Pues bien, en este caso, la matriz **_Numbers** no se encuentra dentro de la clase *ClsCalculo2*, sino dentro de la clase *padre* **ClsCalculo**.

Con la palabra clave *SUPER* no nos referimos al objeto actual, sino al padre del objeto actual.

Escribamos ahora el código de la función **Main()** para aprovechar la nueva característica de nuestra clase *hija*:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo2

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()

END
```

Como es fácil adivinar, el resultado de la ejecución es:

```
Elementos: 3 - Media: 14.6.
```


Modificando funcionalidades

Hasta ahora tenemos dos clases, una de las cuales aporta una propiedad adicional *Count*, y sigue siendo poco útil ya que podríamos, sin más, haber añadido esa funcionalidad a la clase original *ClsCalculo*, evitándonos el mantenimiento de dos clases diferentes. Vamos ahora a explorar la verdadera potencia del concepto de *Herencia*, y es la posibilidad de reemplazar los métodos y propiedades originales por nuevas implementaciones que den lugar a resultados distintos: en este caso, la nueva clase, cuando el método **Average** sea llamado, borrará todos los elementos de la matriz para quedarse en blanco. De esta forma, tendremos dos clases con diferente funcionalidad partiendo de una base común: los objetos de la clase *padre* acumulan números sin fin, mientras que en la segunda cada llamada a **Average** los limpia y comienza de cero.

La clase *ClsCalculo2* queda ahora como sigue:

```
' Gambas class file
INHERITS ClsCalculo
```

```
PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END

PUBLIC FUNCTION Average() AS Float

  DIM Nm AS Float
  DIM Vl AS Float
  DIM Total AS Integer

  IF SUPER._Numbers.Count = 0 THEN RETURN 0

  FOR EACH Vl IN SUPER._Numbers
    Nm += Vl
    Total += 1
  NEXT

  SUPER._Numbers.Clear()

  RETURN Nm / Total

END
```

Ahora existe un método *Average* dentro de *ClsCalculo2*, y este método reemplaza en la clase *hija* el método *Average* del *padre*. Cuando llamemos al método *Average* de un objeto de la clase *ClsCalculo2*, el intérprete no llamará a la función original del *padre*, sino a la implementación de la clase *hija*.

Modificamos la función Main() de pruebas con este nuevo código para comprobar el resultado:

```
PUBLIC SUB Main()


  DIM Calculo AS NEW ClsCalculo2

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()

  Calculo.Add(17.5)
  Calculo.Add(31.8)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()


END
```

Como resultado, obtenemos en consola dos líneas que muestran el funcionamiento de la nueva clase:

```
Elementos: 3 - Media: 14.6
Elementos: 2 - Media: 24.65
```

Si en la primera línea de Main() definimos ahora que el objeto Calculo pertenece a la clase *padre* ClsCalculo, eliminamos el uso de *Count*, que no existe en el padre, y volvemos a ejecutar el programa, observaremos claramente la diferencia:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Media: " & Calculo.Average()

  Calculo.Add(17.5)
  Calculo.Add(31.8)

  PRINT "Media: " & Calculo.Average()

END
```


En este caso, se calcula primero la media de los tres primeros números y luego la media de los cinco números:

```
Media: 14.6
Media: 18.62
```

Podemos aprovecharnos más de la herencia, empleando la palabra *SUPER* que vimos anteriormente. La clase original *ClsCalculo* ya tenía el código necesario para calcular la media, y en la clase *hija ClsCalculo2* tan sólo necesitamos borrar los elementos de la matriz una vez calculada la media.

Podemos, entonces, modificar el código de la nueva función, **Average**, para que llame a la original y después borre los elementos, ahorrando, como siempre, tiempo y código:

```
' Gambas class file
INHERITS ClsCalculo
```

```
PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END

PUBLIC FUNCTION Average() AS Float

  DIM Nm AS Float

  Nm = SUPER.Average()
  SUPER._Numbers.Clear()

  RETURN Nm

END
```


La nueva función **Average** ahora se limita a llamar a la original, implementada en la clase *padre*, a almacenar ese valor, borrar los elementos de la matriz y devolver el valor.

Reemplazando métodos especiales: _New y _Free

Vamos a crear una nueva clase *hija* de *ClsCalculo*, que se comporte como la original, pero que, en este caso, en el momento de inicializarse un objeto de la clase, almacene unos valores. Pongamos de ejemplo una toma de muestras de temperatura.

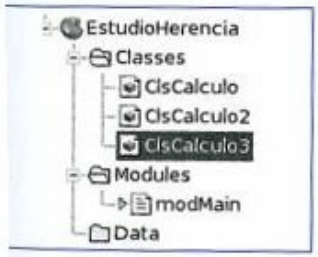

Figura 5. Nueva clase hija *ClsCalculo3*.

Los objetos, al inicializarse, pueden tomar los valores de temperaturas de la semana anterior de forma automática para que, después, el programador introduzca los de la semana en curso y se obtenga la media de la quincena. Para ello creamos la clase *ClsCalculo3*, con este código:

```
' Gambas class file
INHERITS ClsCalculo

PUBLIC SUB _New()

  SUPER.Add(12.5)
  SUPER.Add(15.3)
  SUPER.Add(18.4)
  SUPER.Add(19.12)
  SUPER.Add(21.15)
  SUPER.Add(20.4)
  SUPER.Add(19.5)

END
```


Igualmente escribimos y ejecutamos el código de Main() para comprobar el resultado:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo3

  Calculo.Add(29.2)
  PRINT "Media: " & Calculo.Average()

END
```

El resultado de la ejecución es:

```
Media: 19.44625
```

Como era de esperar, por lo explicado anteriormente, se ha ejecutado el método _New() de nuestra clase *hija*. Ahora bien, en este método tan sólo añadimos elementos,

pero no inicializamos la matriz, por lo que de haberse comportado el intérprete como en los casos anteriores, habríamos obtenido un error en tiempo de ejecución, al tener en el inicio la matriz con valor *Null*.

Lo que ha ocurrido en realidad, es que al reemplazar el método especial _New(), el intérprete ejecuta siempre el método original de la clase *padre* y luego la nueva implementación del *hijo*.

El método especial *_New()* no se reemplaza en las clases *hijas* como en el resto de métodos o propiedades, en su lugar se llama siempre al método original *_New()* del *padre* para que inicialice todo lo necesario y, después, si existe, se llama al método *_New()* de la clase *hija*. De este modo, el objeto siempre está listo e inicializado cuando se comienza a trabajar con él.

Añadamos ahora un destructor _Free() personalizado para nuestra clase heredera, en el cual indicamos el valor de la variable _Numbers:

```
' Gambas class file
INHERITS ClsCalculo

PUBLIC SUB _New()

  SUPER.Add(12.5)
  SUPER.Add(15.3)
  SUPER.Add(18.4)
  SUPER.Add(19.12)
  SUPER.Add(21.15)
  SUPER.Add(20.4)
  SUPER.Add(19.5)

END
```

```
PUBLIC SUB _Free()

  if SUPER._Numbers=NULL then print "NULO"

END
```

La ejecución del programa da ahora como resultado lo que vemos a continuación:

```
Media: 19.44625
NULO
```

Con esto se demuestra que el intérprete, como en el caso de *_New*, ha llamado primero al método *_Free* de la clase original, para llamar luego al método *_Free* de la clase *hija*.

Si nos ha surgido alguna duda respecto al orden de ejecución de algunas funciones, recordemos que siempre podremos ejecutar el código paso a paso para comprobarlo.

Limitaciones

La principal limitación de la herencia en Gambas, es que una clase no puede ser *hija* de otra que, a su vez, era *hija* de una tercera.

En otras palabras, sólo hay un nivel de herencia y en nuestros ejemplos anteriores no podríamos crear una clase *ClsCalculo4* que proviniera de *ClsCalculo2* o *ClsCalculo3*, puesto que éstas están empleando ya la palabra clave *INHERITS* para indicar que heredan las propiedades de *ClsCalculo*.

Esto implica, por ejemplo, que no podemos crear clases que sean herederas de *Form*, pues la clase *Form* proviene de *Window*.

8. 3 Crear un nuevo control con Gambas

Planteando el código

Vamos a examinar la posibilidad de crear nuevos controles gráficos directamente desde Gambas, aprovechando el concepto de herencia.

Crear controles nuevos programados en Gambas aporta dos ventajas fundamentales: su desarrollo es rápido al trabajar en un lenguaje de alto nivel, y el mismo código sirve tanto si programamos con *gb.gtk*, como si lo hacemos con *gb.qt*, ahorrándonos dos implementaciones para un mismo control con diferentes *toolkits*.

Las desventajas son una velocidad menor del código al ser interpretado, cuestión que tendrá más o menos importancia según la función del nuevo control, así como menor flexibilidad a la hora de moldear el control, dado que trabajaremos sobre los controles previamente aportados con Gambas y no sobre un *toolkit* para C o C++ como GTK+ o QT, ni a bajo nivel con X-Windows.

Para crear un control con Gambas, tenemos que partir de uno que exista ya. Si estamos acostumbrados a trabajar con otros entornos RAD de BASIC o C++, nos será familiar partir de un control *plantilla* con unas propiedades y métodos básicos para crear uno nuevo.

Los controles creados con Gambas han de partir de un control ya existente y creable, a su vez. No se puede partir directamente de la clase base *Control*, es decir, un control creado con *INHERITS CONTROL* no funcionará, ya que *Control* no tiene una representación gráfica, sino que es simplemente la base que define las propiedades, métodos y eventos mínimos para otros controles reales (*Label, Button*...).

Nuestro nuevo control será una etiqueta a la que llamaremos *ColorLabel*, similar al *Label*, pero que muestra el texto con un gradiente de color.

Para dibujar texto en diferentes colores, trazaremos cada letra por separado, desde un color inicial *Color1* hasta un color final *Color2*, utilizando un algoritmo algo rudimentario: tomamos la diferencia de valor de los dos colores y vamos aumentando el valor del color actual por cada nueva letra que dibujamos.

Con el fin de trazar cada letra por separado, tendremos que usar la clase *Draw* sobre un control base que será *DrawingArea*.

Implementación básica

Creamos un nuevo proyecto gráfico, que llamaremos *ColorLabel*, para nuestras pruebas. Dentro de él, originaremos un formulario *FMain*, también para nuestras pruebas, así como una clase llamada *ColorLabel* para implementar el control. Al igual que una etiqueta normal, definiremos una propiedad **Text** con el texto a mostrar, y añadiremos dos propiedades nuevas **Color1** y **Color2** que contienen el valor inicial y final del gradiente. Para manejar las tres propiedades, tendremos tres variables privadas: **hText** para el texto, **hColor1** para el primer color y **hColor2** para el segundo color.

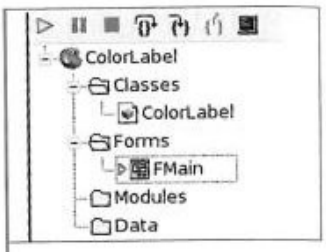

Figura 6. Nuevo proyecto gráfico *ColorLabel*.

Empezamos, pues, a escribir el código de la clase:

```
' Gambas class file
INHERITS DrawingArea

PRIVATE hText AS String
PRIVATE hColor1 AS Integer
PRIVATE hColor2 AS Integer

PROPERTY Text AS String
PROPERTY Color1 AS Integer
PROPERTY Color2 AS Integer
```

Implementar las funciones de lectura de las tres propiedades es trivial, basta con devolver los valores de nuestras variables privadas (recordemos que estamos escribiendo el código siempre dentro de la clase *ColorLabel*):

```
PRIVATE FUNCTION Color1_Read() AS Integer

  RETURN hColor1

END


PRIVATE FUNCTION Color2_Read() AS Integer

  RETURN hColor2

END

PRIVATE FUNCTION Text_Read() AS String

  RETURN hText

END
```

En cuanto a las funciones de escritura, hemos de asignar el valor que el usuario pasa a nuestras propiedades, y hemos de actualizar el control cada vez que una de ellas cambia, tarea que realizaremos en una función llamada **Redraw**(), aún por definir:

```
PRIVATE SUB Text_Write(Vl AS String)

  hText = Vl
  Redraw()

END
```

```
PRIVATE SUB Color1_Write(Vl AS Integer)

  hColor1 = Vl
  Redraw()

END

PRIVATE SUB Color2_Write(Vl AS Integer)

  hColor2 = Vl
  Redraw()

END
```

Llega el punto de implementar **Redraw**(), para lo cual recorreremos carácter por carácter la cadena almacenada en la variable **hText** e iremos representando letra por letra usando los distintos colores del gradiente. Para calcular la posición en el eje X de cada letra, nos servimos del método **Font.Width**() que determina el ancho de cada letra con la fuente actual del control:


```
PRIVATE SUB Redraw()

  DIM xPos AS Integer
  DIM Bucle AS Integer
  DIM hColor AS Integer
  DIM hDiff AS Integer

  ME.Clear
  hColor = hColor1
  IF Len(hText) THEN hDiff = (hColor2 - hColor1) /
  Len(hText)

  Draw.Begin(ME)
```

```
    FOR Bucle = 1 TO Len(hText)

      Draw.ForeColor = hColor
      Draw.Text(Mid(hText, Bucle, 1), xPos, 0)
      xPos = xPos + SUPER.Font.Width(Mid(hText, Bucle, 1))
      hColor = hColor + hDiff

    NEXT
    Draw.End()

  END
```

Finalmente, nos interesa que el control se encargue por sí mismo del redibujado de la ventana. Por defecto, todo lo dibujado sobre un *DrawingArea* desaparece si la ventana se oculta tras otra ventana o se minimiza. Para evitarlo, utilizaremos la propiedad **Cached** de *DrawingArea*, que se encarga de mantener un buffer y redibujar las partes expuestas al usuario del control:

```
  PUBLIC SUB _New()

    SUPER.Cached = TRUE

  END
```

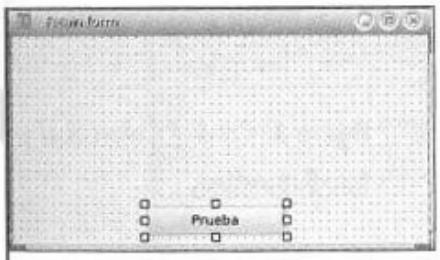

Figura 7. Texto de prueba sobre el formulario *FMain*.

Probemos ahora el control. Nuestro formulario *FMain* tendrá unas dimensiones de 225x240 píxeles y contendrá un botón llamado *BtnPrueba* con el texto **Prueba**.

El código del formulario consiste en crear un control **ColorLabel** en tiempo de ejecución, situándolo en el formulario con un texto de ejemplo y dos colores definidos para el gradiente:

```
PUBLIC SUB BtnPrueba_Click()

  DIM hLabel AS ColorLabel

  hLabel = NEW ColorLabel(ME)
  hLabel.Font.Size = 18
  hLabel.Move(0, 0, ME.Width, 100)
  hLabel.Color1 = Color.Blue
  hLabel.Color2 = Color.DarkBlue
  hLabel.Text = "Hola desde el ColorLabel"

END
```

Al ejecutar el programa y pulsar el botón, podremos ver ya nuestro control en funcionamiento. Ha sido una tarea rápida y sencilla, si bien se puede depurar bastante el código del control. Por ejemplo, este control tendrá problemas con caracteres con código ASCII mayor al 127, ya que en la codificación de Gambas (UTF-8) ocupan más de un byte, rompiendo el algoritmo que recoge el texto letra por letra.

Se puede solucionar convirtiendo la cadena a una codificación que emplee siempre un byte por letra (por ejemplo, ISO-8859-1) y reconvertirlo a UTF-8 a la hora de representarlo. También podríamos incluir una propiedad para alinear el texto a derecha, izquierda o centro, así como mejorar el algoritmo para obtener el gradiente, separando los componentes RGB de cada color, tareas todas ellas que dejamos si estamos interesados en practicar.

Figura 8. Control *ColorLabel* en funcionamiento.

Este nuevo control se puede reutilizar en todos los programas que diseñemos añadiendo la clase *ColorLabel* al proyecto.

8. 4 Nuevos componentes con Gambas

Cuando abrimos las propiedades de un proyecto y pulsamos la pestaña **Componentes**, observamos los módulos adicionales que podemos emplear en un programa escrito con Gambas, los cuales aportan nuevas funcionalidades.

Desde Gambas, tenemos también la posibilidad de crear nuevos componentes. Estos pueden ser de cualquier índole, aunque en el próximo ejemplo será un componente que aporta un control adicional: el *ColorLabel* que creamos en el apartado anterior.

Si bien vamos a crear aquí un nuevo componente gráfico, los componentes se limitan a aportar nuevas clases que no tienen por qué estar relacionadas con la interfaz gráfica.

Figura 9. Componentes del proyecto.

Preparación del código. Palabra clave EXPORT

Dentro de un programa Gambas, tendremos diversos módulos, clases y formularios. Implementar un control en Gambas supone tan sólo crear un programa que exportará algunas de sus clases. Preparar un programa Gambas para ser un componente, simplemente implica definir qué clases serán visibles y cuáles no.

Para indicar que una clase será visible desde el programa principal, se usa la palabra clave EXPORT al principio de la misma antes del resto del código. Por tanto, en nuestro programa *ColorLabel* indicamos esta palabra clave al inicio de la clase *ColorLabel*:

```
' Gambas class file
EXPORT
INHERITS DrawingArea

PRIVATE hText AS String
...
```


Ahora es necesario compilar el programa para tener un ejecutable con la nueva clase exportada.

Archivos necesarios, ubicaciones

Cada componente Gambas necesita, en primer lugar, del programa en sí, que tendrá siempre como nombre *gb.x.gambas*, donde *x* será un nombre que nosotros elijamos.

A partir de ahora, vamos a llamar a nuestro nuevo componente *controles* y, por tanto, el nombre del componente dentro del sistema de archivos será *gb.controles.gambas*.

Es necesaria también una ubicación específica para dicho archivo. Todos los componentes de Gambas han de estar situados en una carpeta que, salvo que hayamos compilado Gambas desde los fuentes empleando una opción - -*prefix* específica, será */usr/lib/gambas2*.

Si hemos compilado Gambas usando una opción - -*prefix* distinta de */usr* en el script *configure* (por ejemplo, */usr/local*), tendremos que buscar las rutas indicadas a partir de la que seleccionamos, como */usr/local/lib/gambas2* en este caso.

Copiaremos nuestro ejecutable a dicha carpeta, para lo cual, desde la consola y como *root*, entraremos en la carpeta del proyecto y copiaremos el ejecutable con el nuevo nombre (por supuesto, si lo preferimos, podemos hacerlo con Konqueror o Nautilus):

```
cp ColorLabel.gambas /usr/lib/gambas2/gb.controles.gambas
```

Los componentes en Gambas necesitan también de un archivo con extensión **.component** para ser reconocidos como tales. Vamos a crear este archivo a mano con cualquier editor. Nosotros vamos a usar **gedit**, pero puede ser cualquiera (recordemos que debemos ser root):

```
gedit /usr/lib/gambas2/gb.controles.component
```

El contenido de dicho fichero incluye varios parámetros que se exponen a continuación:

```
[Component]
Key=gb.controles
Name=Additional controls
Author=Daniel Campos
Group=Adicionales
Controls=ColorLabel
Require=gb.qt
```

La clave [Component] siempre debe estar al inicio del archivo. En el valor Key hay que situar el nombre del componente sin extensión (en este caso, el fichero *gb.component.gambas* que hemos copiado, pero sin la extensión *gambas*). En la clave Name

se indica el nombre del componente que aparecerá luego como descripción a la hora de elegir los componentes de un proyecto. En **Author** situamos el nombre del autor del componente.

Después vienen una serie de claves opcionales: **Group** sólo se usa si queremos añadir controles gráficos y supone la pestaña de controles donde queremos que se añada el nuevo. Lo mismo se aplica para **Controls**, que se referirá a cada uno de los controles que queremos que aparezcan en dicha pestaña. **Require** se utiliza si nuestro componente depende de otros, en este caso, de **gb.qt**. Si hubiéramos compilado el proyecto para *gb.gtk*, tendría que ser ésa la dependencia indicada.

Una vez creado y guardado el archivo *gb.controles.component*, tenemos finalmente que copiar dos archivos dentro de */usr/share/gambas2/info*, que proporcionan información adicional para el intérprete de Gambas.

Dentro de nuestra carpeta del proyecto se han creado también tras compilar, dos archivos llamados **.list** e **.info**. Si deseamos verlos, recordemos que los archivos que comienzan por un punto son ocultos, por tanto, hemos de usar *ls -a* desde la consola o bien activar la opción de ver archivos ocultos desde Konqueror o Nautilus. Dichos archivos han de copiarse a su ubicación definitiva con el nombre del componente (en este caso, **gb.controles.info** y **gb.controles.list**):

```
cp .info /usr/share/gambas2/info/gb.controles.info
cp .list /usr/share/gambas2/info/gb.controles.list
```

Probando el nuevo componente

Tras estos pasos cerraremos el IDE de Gambas si estaba abierto y lo ejecutaremos de nuevo, creando un proyecto gráfico llamado *MiPrueba* en el que añadiremos un formulario.

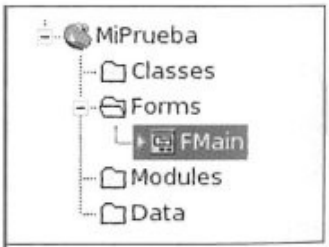

Figura 10. Formulario *FMain* sobre el proyecto *MiPrueba*.

Dentro de **Proyecto | Propiedades**, seleccionaremos la pestaña de **Componentes**.

Buscaremos y marcaremos el componente *gb.controles* y pulsaremos Aceptar para tenerlo como dependencia.

Figura 11. Selección del componente *gb.controles* como dependencia.

Figura 12. Pestaña **Adicionales** de la ventana de selección.

Ahora, en la ventana de selección de controles aparece una nueva pestaña **Adicionales**, tal y como habíamos indicado en el archivo *component* (**Figura 12**).

Al situarnos en esta nueva pestaña, aparecen los controles disponibles. Desgraciadamente, en este momento no se contempla la posibilidad de que esos controles tengan un icono propio, por lo que aparecen sin un dibujo representativo (**Figura 13**).

No obstante, al situarnos sobre él con el ratón, se muestra el botón y el *ToolTip* indicándonos el nuevo control.

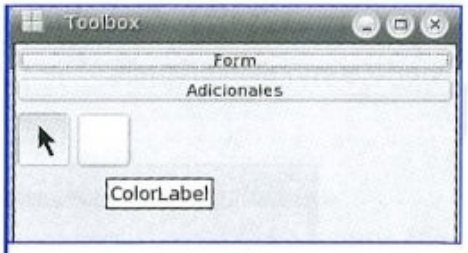

Figura 13. Controles **Adicionales**.

Como hacemos con el resto de controles, lo arrastraremos al formulario y lo situaremos como deseemos.

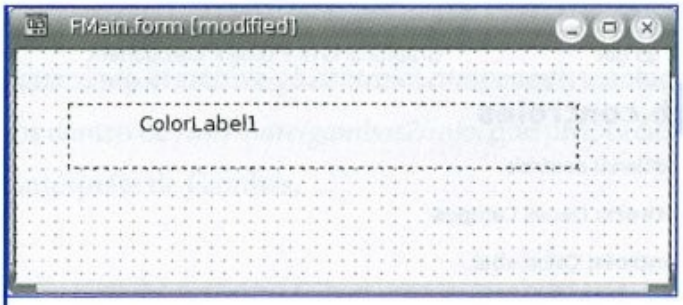

Figura 14. Control *ColorLabel1*.

Finalmente, en el código del formulario indicamos el texto y los colores de la etiqueta:

```
PUBLIC SUB Form_Open()

  ColorLabel1.Font.Size = 16
  ColorLabel1.Text = "Hola desde el componente"
  ColorLabel1.Color1 = Color.Blue
  ColorLabel1.Color2 = Color.Black

END
```

Al ejecutarlo, observamos el resultado. Ya disponemos de un nuevo componente reutilizable en muchos proyectos y que se puede distribuir entre distintos equipos siguiendo los pasos indicados.

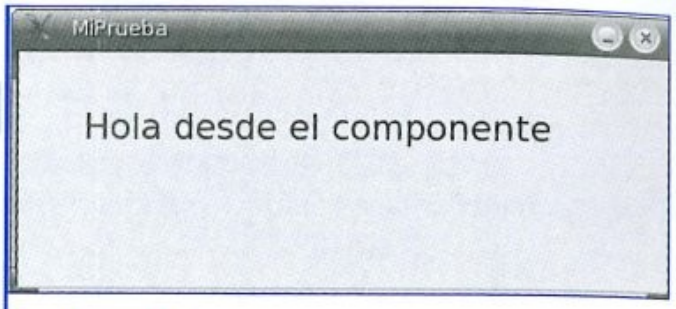

Figura 15. Resultado del nuevo componente.

Un componente escrito en Gambas es tan sólo un programa que exporta clases, por tanto, se puede depurar fácilmente haciendo las pruebas como en cualquier otro proyecto para, después, distribuirlo como componente.