Manual de Gtk Sharp

De MonoHispano, la enciclopedia libre.

Tabla de contenidos

Prologo

Este tutorial describe el uso GTK# para mono. Esta basado en el tutorial escrito con John Finlay y traducido por Lorenzon Gil Sánchez

Introducción

Gtk-sharp es un conjunto de ensamblajes(assemblys) que componen la interfaz de Lenguaje intermedio(IL) para GTK+ 2.0. A través del resto de este documento Gtk-sharp se refiere a la versión 1.0 de Gtk-sharp y GTK y GTK+ se refieren a la versión 2.0 de GTK+, El principal sitio web de Gtk-sharp es gtk-sharp.sf.net. Además mcs se refiere al Compilador C# de mono, versión 1.0

C# (pronunciado "ce sharp") es un lenguaje de programación orientado a objetos desarrollado por Microsoft como parte de su iniciativa .NET. Microsoft basó C# en C++ y Java. C# fue diseñado para combinar potencia(la influencia de C++) y velocidad de programación(Las influencias de Visual Basic y java).

Microsoft ha entregado C# a ECMA para la estandarización formal. En diciembre de 2001, ECMA lanzó ECMA-334 "Especificación de Lenguaje C#" además C# se hizo un estandard ISO en 2003 (ISO/IEC 23270). Hay algunas implementaciones independientes que están siendo desarrolladas, por ejemplo:

Mono, La implementación .NET libre de Ximian dotGNU y Portable.NET, de la Free Software Foundation

GTK (GIMP Toolkit) es una librería para crear interfaces de usuario gráficas. Esta licenciada usando la licencia LGPL, por lo que puedes desarrollar software abierto, software libre, o incluso software no libre usando GTK sin tener que pagar nada en licencias o derechos.

Se llama el toolkit de GIMP porque originariamente fue escrita para desarrollar el Programa de Manipulación de Imágenes GNU (GIMP), pero GTK se usa ahora en un amplio número de proyectos de software, incluyendo el proyecto de Entorno de Modelo de Objetos orientados a Red (GNOME). GTK está diseñada encima de GDK (Kit de Dibujo de GIMP) que básicamente es una abstracción de las funciones de bajo nivel para acceder a las funciones del sistema de ventanas (Xlib en el caso del sistema de ventanas X). Los principales autores de GTK son:

Peter Mattis petm@xcf.berkeley.edu

Spencer Kimball spencer@xcf.berkeley.edu

Josh MacDonald jmacd@xcf.berkeley.edu

Actualmente GTK es mantenida por:

Owen Taylor otaylor@redhat.com

Tim Janik timj@gtk.org

Básicamente GTK es un interfaz orientada a objetos para programadores de aplicaciones (API). Aunque está escrita completamente en C, está implementada usando la idea de clases y funciones de retro-llamada (punteros a función).

También hay un tercer componente, llamado Glib, que contiene unas cuantas funciones que reemplazan algunas llamadas estandard, así como funciones adicionales para manejar listas enlazadas, etc. Las funciones de reemplazo se usan para aumentar la portabilidad de GTK ya que algunas de las funciones que implementa no están disponibles o no son estandard en otros Unix tales como g_strerror(). Algunas también incluyen mejoras a las versiones de libc, tales como g_malloc que tiene utilidades de depuración mejoradas.

En la versión 2.0, GLib ha incluido el sistema de tipos que forma la base para la jerarquía de clases de GTK, el sistema de señales usado en GTK, una API de hebras que abstrae las diferentes APIs nativas de hebras de las diversas plataformas y una facilidad para cargar módulos.

Como último componente, GTK usa la librería Pango para la salida de texto internacionalizado.

Este tutorial describe Gtk-sharp desde el lenguaje c-sharp y esta basado en el tutorial de PyGTK 2.0 escrito por John Finlay y Traducido al español por Lorenzo Gil Sánchez Este tutorial intenta documentar todo lo posible Gtk-sharp, pero en ningún caso es completo.

Este tutorial asume algún conocimiento previo de c-sharp, y de cómo crear y ejecutar programas escritos en c-sharp. Si no estas familiarizado con c-sharp, por favor lee el Mono Handbook primero. Este tutorial no asume ningún conocimiento previo de GTK; si estás aprendiendo Gtk-sharp para aprender GTK, por favor comenta cómo encuentras este tutorial, y con qué has tenido problemas. Este tutorial no describe cómo compilar o instalar Mono, GTK+ o Gtk-sharp.

Este tutorial está basado en:

GTK+ 2.4

Mono 1.0

Gtk-sharp 1.0

Los ejemplos fueron escritos y probados en una Debian 3.1 (Sarge/Testing)

Primeros Pasos

Hola Mundo en Gtk#

Ahora haremos un programa con un control (un botón). Es la versión GTK# del clásico programa "Hola Mundo" helloworld.cs (*http://www.monohispano.org/tutoriales/man_gtksharp/examples/helloworld.cs*).

```
// ejemplo helloworld.cs
using System;
using Gtk;

public class HelloWorldWindow: Window {

    // Creamos un nuevo botón con la etiqueta "Hola Mundo".
    private Button button = new Button("Hola Mundo");

   // Esta es un método de retrollamada. Se usará posteriormente para
   // manejar el evento Clicked del botón. En este ejemplo los
   // argumentos son ignorados. Más adelante se hablará sobre retrollamadas.
   private static void OnButtonClicked(object obj, EventArgs args)
   {
       Console.WriteLine("Hola Mundo");
   }

   // Otro manejador del evento Clicked
   private static void OnButtonClicked2(object obj, EventArgs args)
   {
       Application.Quit();
   }

   //Manejador de evento para el evento DeleteEvent
   private static void OnDelete(object obj, DeleteEventArgs args)
   {
       Console.WriteLine("Ha ocurrido un \"delete event\"");

       // Si se le asigna false a la propiedad RetVal del segundo argumento
       // del manejador de evento de DeleteEvent, GTK destruirá la ventana.
       // Asignarle true significa que no se desea que la ventana sea destruida.
       // Esto es útil para mensajes del tipo "¿Esta seguro que desea salir?"
       args.RetVal = true;
   }

   public HelloWorldWindow()   :   base("Hola Mundo")
   {
       // Añádimos el método WindowDelete al evento DeleteEvent de la ventana.
       // Cuando la ventana emita un evento DeleteEvent (el cual es dado por
       // el administrador de ventanas, generalemente por la opción "cerrar"
       // o por presionar el boton "X" en la barra de título), llamará al
       // método WindowDelete definido arriba
       this.DeleteEvent += new DeleteEventHandler(OnDelete);
       this.BorderWidth = 10;

       // Cuando se dé clic en el boton, se llamará al método OnButtonClicked
definido
       // más arriba
       button.Clicked += new EventHandler(OnButtonClicked);

       // También al hacer clic sobre el boton, se llamará al método
OnButtonClicked2
```

```
            // el cual finalizará la aplicación
            button.Clicked += new EventHandler(OnButtonClicked2);

            // Esto agrega el botón a la ventana (un contenedor GTK+)
            this.Add(button);

            // Muestra la ventana y su contenido
            this.ShowAll();
        }
    }

       class MainClass {
        public static void Main(){

            // Se inicializa GTK+
            Application.Init();

            // Creamos la ventana HelloWorldWindow
            new HelloWorldWindow();

            // Entra en un ciclo en el cual la aplicación duerme, esperando por
eventos.
            Application.Run();
        }
    }
```

La Figura 2-2 muestra la ventana creada por helloworld.cs (*http://www.monohispano.org/tutoriales/man_gtksharp/examples/helloworld.cs*).

Figura 2-2. Programa de ejemplo Hola Mundo

Las clases, enumeraciones y delegados de GTK# se definen en el espacio de nombres Gtk y se llaman de la forma Gtk.*. Por ejemplo, el programa helloworld.cs (*http://www.monohispano.org/tutoriales/man_gtksharp/examples/helloworld.cs*) usa:

```
Gtk.Application
Gtk.Button
Gtk.Window
```

Esto se puede facilitar importando el espacio de nombres Gtk, como en la línea 4 del programa helloworld.cs (*http://www.monohispano.org/tutoriales/man_gtksharp/examples/helloworld.cs*).

Compilando

La compilación de un programa que usa solo GTK# bajo Mono es bastante simple:

```
mcs -pkg:gtk-sharp-2.0 helloworld.cs
```

La opción -pkg será llamativa para los usuarios de .NET. Eso es porque .NET no soporta el mecanismo pkgconfig que sí es soportado por Mono. Si estuviesemos corriendo .NET, precisaríamos sacar esta opción y agregarle *algo hací como*:

```
-r:gtk-sharp-2.0.dll -r:atk-sharp-2.0.dll -r:pango-sharp-2.0.dll ....
```

Ejecutando

```
mono helloworld.exe
```

O, si hemos configurado nuestro Linux apropiadamente:

```
./helloworld.exe
```

Teoría de Eventos

En la versión 2.0 de GTK+, el sistema de señales se ha movido de GTK+ a GLib. No entraremos en detalles sobre las extensiones que GLib 2.0 tiene en relación con el sistema de señales de GTK+ 1.2. Las diferencias no deberían notarse entre los usuarios de GTK#.

Antes de que entremos en detalle en helloworld.cs, discutiremos las señales y las retrollamadas. GTK+ es una librería orientada a eventos, lo que significa que se dormirá en el método Application.Run() hasta que un evento ocurra y el control pase a la función apropiada.

Esta delegación del control se realiza usando la idea de "señales". (Nótese que estas señales no son las mismas que las señales de los sistemas Unix, y no se implementan usando estas, aunque la terminología es casi idéntica). Cuando un evento ocurre, como cuando presionamos un botón del ratón, la señal apropiada será "emitida" por el el control que fue presionado. Así es cómo GTK+ hace la mayoría de su trabajo útil. En Gtk# las señales son representadas mediante el uso de eventos event del leguaje C#. Hay señales que todos los controles heredan, como Destroy, y hay señales que son específicas a cada control, como Toggled en un botón de activación.

Para hacer que un botón realice una acción, tenemos que configurar un manejador de señales que capture estas señales y llame a la función apropiada. Esto se hace añadiendo un método uno de los eventos del objeto dado. Por ejemplo:

```
object.Event +=  EventHandler(HandlerMetod);
```

donde object es la instancia de Widget (o una de sus derivadas) que estará emitiendo la señal, y el miembro Event es el evento que deseas capturar. EventHandler es el delegado que se usa para la mayoría de los eventos. El argumento HandlerMetod es el método que se quiere que sea llamado cuando ocurre el evento. El método recibe como segundo parámetro un objeto EventArgs, cuya propiedad RetVal puede usarse para desconectar o bloquear el manejador.

El método especificado en el constructor de EventHandler se llama "método de retrollamada", y generalmente es de la forma:

```
public static void HandlerMetod(object obj, EventArgs args)
{
    ...
}
```

donde el primer argumento será una referencia al Widget (control) que emitió el evento, y el segundo (args)

una referencia varios datos útiles que podrían servir para el manejador del evento.

La forma anterior de declaración de un método de retrollamada para manejar eventos es sólo una guía general, ya que señales específicas de controles generan diferentes parámetros de llamada.

Eventos

Todos las clases que derivan de Widget poseen un conjunto de eventos que reflejan el mecanismo del sistema de ventanas. Métodos de retrollamada se pueden asignar a cualquiera de estos eventos. Estos son:

```
ButtonPressEvent
ButtonReleaseEvent
ScrollEvent
MotionNotifyEvent
DeleteEvent
DestroyEvent
ExposeEvent
KeyPressEvent
KeyReleaseEvent
EnterNotifyEvent
LeaveNotifyEvent
ConfigureEvent
FocusInEvent
FocusOutEvent
MapEvent
UnmapEvent
PropertyNotifyEvent
SelectionClearEvent
SelectionRequestEvent
SelectionNotifyEvent
ProximityInEvent
ProximityOutEvent
VisibilityNotifyEvent
ClientEvent
NoExposeEvent
WindowStateEvent
```

Como vimos anteriormente, Para conectar una retrollamada a un determinado evento, simplemente es necesario asignarle a este un método, el cual queremos que se ejecute cuando ocurra el evento. Este método que se asigna puede ser ligeramente diferente dependiendo del tipo de evento, pero en general tiene la siguiente forma:

```
public static void HandlerMetod(object obj, EventArgs args)
```

En algunos casos, las variaciones en el método de retrollamada incluyen cambions en el argumento EventArgs args. Por lo general es cambiado por un derivado de la clase EventArgs. Por ejemplo, para manejar el evento que ocurre al presionar un botón del mouse se debe usar un manejador de como:

```
public static void ButtonPressHandler(object obj, ButtonPressEventArgs args)
```

En donde ButtonPressEventArgs es una clase derivada de EventArgs. La clase ButtonPressEventArgs, al igual muchas otras clases de argumentos Gtk#, añade una nueva propiedad Gtk.Event (o alguna de sus clases derivadas) a EventArgs.

Gdk.Event es una clase cuya propiedad Type indicará con más exactitud cuál de los eventos ha ocurrido. Otras propiedades pueden ser añadidas a Gdk.Event en clases derivadas, como es el caso de Gdk.EventButton que define, entre otras, una propiedad Button que especifica cual botón fue oprimido.

```
EventType.Nothing
EventType.Delete
EventType.Destroy
EventType.Expose
EventType.MotionNotify
EventType.ButtonPress
EventType.TwoButtonPress
EventType.ThreeButtonPress
EventType.ButtonRelease
EventType.KeyPress
EventType.KeyRelease
EventType.EnterNotify
EventType.LeaveNotify
EventType.FocusChange
EventType.Configure
EventType.Map
EventType.Unmap
EventType.PropertyNotify
EventType.SelectionClear
EventType.SelectionRequest
EventType.SelectionNotify
EventType.ProximityIn
EventType.ProximityOut
EventType.DragEnter
EventType.DragLeave
EventType.DragMotion
EventType.DragStatus
EventType.DropStart
EventType.DropFinished
EventType.ClientEvent
EventType.VisibilityNotify
EventType.NoExpose
EventType.Scroll
EventType.WindowState
EventType.Setting
```

Para usar un evento Gdk.Event deberíamos usar algo como:

```
using Gdk;

widget.ButtonPressEvent += new ButtonPressEventHandler(ButtonPressHandler);

private void ButtonPressHandler(object obj, ButtonPressEventArgs args) {
    // si el evento es un click simple
    if (args.Event.Type == EventType.ButtonPress) {
        ...
    }
    // si es un doble click
    if (args.Event.Type == EventType.TwoButtonPress) {
        ...
    }

    // si se presionó el botón izquierdo
    if (args.Event.Button == 1) {
        ...
    }
}
```

De esta forma, cuando se presiona un boton, podemos obtener más detalles acerca del evento, como por ejemplo, si fue un click simple o un doble click, o si se presiono el botón izquierdo o el derecho.

Hola Mundo Paso a Paso

Ahora que ya conocemos la teoría que hay detrás de esto, vamos a aclarar el programa de ejemplo helloworld.cs paso a paso.

Las líneas 5-62 definen la clase HelloWorldWindow, la cual deriva de la clase de Gtk# Window y contiene todas las retrollamadas como métodos de la clase y el constructor para la inicialización de objetos. Vamos a examinar los métodos de retrollamada.

La línea 8, al inicio de la declaración de clase, se crea un nuevo botón. El botón tendrá la etiqueta "Hola Mundo", cuando se muestre.

```
private Button button = new Button("Hola Mundo");
```

Las líneas 13-16 definen el método de retrollamada OnButtonClicked(), el cual se llamará cuando el botón sea pulsado. Cuando se llama a este método, se imprime "Hola Mundo" en la consola. En este ejemplo ignoramos los parámetros objeto y argumentos de evento, pero la mayoría de las retrollamadas los usan. En el siguiente ejemplo usaremos el argumento de datos para saber que botón fue pulsado.

```
private static void OnButtonClicked(object obj, EventArgs args)
{
    Console.WriteLine("Hola Mundo");
}
```

En las líneas 19-22 se define el método de retrollamada OnButtonClicked2(), el cual será también cuando el boton sea pulsado. Lo que hace este método es simplemente cerrar la aplicación haciendo una llamada al método estático Quit de la clase Application.

```
private static void OnButtonClicked2(object obj, EventArgs args)
{
    Application.Quit();
}
```

La siguiente retrollamada (líneas 25-34) es un poco especial. El evento DeleteEvent se produce cuando el administrador de ventanas manda este evento al programa. Tenemos varias posibilidades en cuanto a qué hacer con estos eventos. Podemos ignorarlos, realizar algun tipo de respuesta, o simplemente cerrar el programa.

En este método se utiliza el argumento args. El valor que se le asigne a la propiedad RetVal de args le permite a GTK+ saber qué acción realizar. Devolviendo true, le hacemos saber que no queremos que se emita el evento Destroy, y asi nuestra aplicación sigue ejecutandose. Devolviendo false, pedimos que la ventana sea destruida. Nótese que se han quitado los comentarios para una mayor claridad.

```
private static void OnDelete(object obj, DeleteEventArgs args)
{
    Console.WriteLine("Ha ocurrido un \"delete event\"");
    args.RetVal = true;
}
```

Las líneas 30-61 definen el constructor de la clase HelloWorldWindow, el cual crea la ventana y los

controles que se usan en el programa.

La línea 43 ilustra un ejemplo de cómo asignar un manejador de evento a un objeto, en este caso, a la ventana. Se captura el evento DeleteEvent, el cual se emite cuando cerramos la ventana a través del manejador de ventanas, o cuando usamos la llamada al método Destroy() de Widget.

```
this.DeleteEvent += new DeleteEventHandler(OnDelete);
```

La línea 46 establece un atributo nuestra ventana para que tenga un área vacía de 10 píxeles de ancho alrededor de él donde ningún control se situará. Hay otras funciones similares que se tratarán en la sección Poniendo Atributos de Controlesrm -r

```
this.BorderWidth = 10;
```

En la línea 50 asignamos un manejador de evento al botón para que cuando emita la señal "Clicked", nuestro método de retrollamada OnButtonClicked() se llame. Obviamente la señal "Clicked" se emite cuando hacemos clic en el botón con el cursor del ratón. Este método imprimirá la frase "Hola Mundo" en la consola.

```
button.Clicked += new EventHandler(OnButtonClicked);
```

También vamos a usar este botón para salir de nuestro programa. La línea 54 muestra como es posible agregar muchos métodos diferentes para que actuen como manejadores de evento. Cuando hacemos clic en el botón, al igual que antes, se llama al método de retrollamada OnButtonClicked() primero, y después al siguiente en el orden en el que son asignados (o sumados). Puedes tener todos los métodos de retrollamada que necesistes y se ejecutarán en el orden en el que los asignaste.

Entonces asignamos el método OnButtonClicked2(), el cual definimos más atrás, al evento clicked del botón. De esta manera se va a cerrar la aplicación, justo despues de haber escrito "Hola Mundo" en la consola, que es lo que se hacía en OnButtonClicked().

```
button.Clicked += new EventHandler(OnButtonClicked2);
```

La línea 57 es una llamada de colocación, que será explicada en profundidad más tarde en Colocando ontroles . Pero es bastante fácil de entender. Simplemente le dice a GTK que el botón debe situarse en la ventana donde se mostrará. Ten en cuenta que un contenedor GTK sólo puede contener un control. Hay otros controles, que se describen después, que están diseñados para posicionar varios controles de diferentes maneras.

```
this.Add(button);
```

Ahora lo tenemos todo configurado como queremos. Con todos los manejadores de señales, y el botón situado en la ventana donde debería estar, le pedimos a GTK (línea 60) que muestre la ventana con todos los controles en la pantalla.

```
this.ShowAll();
```

Las líneas 64-75 definen una nueva clase llamada MainClass, la cual contrendrá en punto de entrada a nuestra aplicación, es decir, el método Main() (líneas 65-74). Dentro del método Main() se llama al método estático Init() de la clase Application, el cual se encarga inicializar GTK+. Despues se crea una instancia implícita de la clase HelloWorldWindow (nuestra ventana) definida más arriba, con lo cual esta se construirá y se mostrará en pantalla. Por último, se hace un llamado al método estático Run() de la clase Application, con lo cual nuesta aplicación esperará esperando a que ocurran eventos de GTK+.

```
public class MainClass
{
```

```
        public static void Main()
        {
            Application.Init();
            new HelloWorldWindow();
            Application.Run();
        }
    }
```

Ahora, cuando hagamos clic con el botón del ratón en el botón GTK, el control emitirá una señal "Clicked". Para poder usar esta información, nuestro programa configura un evento para que capture esta señal, la cual llama al método que decidamos. En nuestro ejemplo, cuando el botón que hemos creado es pulsado, se llama al método OnButtonClicked() de la clase HelloWorldWindow, y después se llama el siguiente manejador para este evento. El siguiente manejador es el método OnButtonClick2(), el cual hace que la aplicación se cierre.

Otra función de los eventos es usar el manejador de ventanas para matar la ventana, lo cual causará que se emita DeleteEvent. Esto llamará nuestro manejador de DeleteEvent. Si asignamos true a arg.RetVal aqui, la ventana se quedará como si nada hubiera pasado. Asignando false GTK+ destruirá la ventana

Avanzando

Más sobre manejadores de señales

Veamos otra vez la llamada a EventHandler.

```
objeto.señal += new EventHandler (función);
```

Como ya se ha dicho, se pueden tener tantas retrollamadas por señal como necesites, y cada una se ejecutará por turnos, en el orden en el que fueron conectadas.

Puedes eliminar la retrollamada de la lista mediante:

```
object.disconnect(id)
```

También puedes deshabilitar temporalmente los manejadores de señal con los métodos signal_handler_block() y signal_handler_unblock() .

```
object.signal_handler_block(handler_id)

object.signal_handler_unblock(handler_id)
```

Un Hola Mundo Mejorado

Empaquetamiento de Controles

Perspectiva General de Controles

Cuando crees un programa, querrás poner más de un control en una ventana. Nuestro primer ejemplo helloworld.cs solo usaba un control para que pudieramos usar simplemente el método Add() de la clase Container para "empaquetar" el control en la ventana. Pero cuando quieres poner más de un control en la ventana, ¿cómo controlas el sitio donde ese control se coloca? Aqui es donde la colocación entra en juego.

Teoría de Cajas Empaquetadoras

La mayoría del empaquetamiento se realiza utilizando cajas. Estos contenedores invisibles de controles pueden ser de dos tipos, una caja horizontal, y una caja vertical. Cuando empaquetamos controles en una caja horizontal, los objetos se insertan horizontalmente de izquierda a derecha o de derecha a izquierda dependiendo de la llamada que se use. En una caja vertical, los controles se empaquetan de arriba a abajo o viceversa. Puedes usar una combinación de cajas dentro de cajas para obtener el efecto deseado.

Para crear una nueva caja horizontal, usamos un objeto de HBox, y para cajas verticales, VBox. Los métodos PackStart() y PackEnd() se utilizan para colocar objetos dentro de estos contenedores. El método PackStart() empezará en la parte de arriba e irá bajando en una vbox, y de izquierda a derecha en una hbox. El método PackEnd() hará lo contrario, empaquetará de abajo a arriba en una vbox, y de derecha a izquierda en una hbox. Usando estos métodos, podemos alinear a la derecha o a la izquierda nuestros controles y se pueden mezclar de la forma necesaria para obtener el efecto deseado. Usaremos PackStart() en la mayoría de nuestros ejemplos. Un objeto puede ser otro contenedor o un control. De hecho, muchos controles son en realidad contenedores por ellos mismos, incluyendo el botón, pero normalmente sólo usamos una etiqueta dentro de un botón.

Usando estas llamadas, GTK sabe donde quieres colocar tus controles y asi puede cambiar el tamaño automáticamente y otras cosas interesantes. Como puedes imaginar, este método nos da bastante flexibilidad al colocar y crear controles.

Detalles de Cajas

A causa de esta flexibilidad, el empaquetamiento de cajas puede ser confuso al principio. Hay muchas opciones, y no es obvio al principio cómo enacajan todas ellas. Al final, de cualquier forma, hay básicamente cinco estilos. La Figura 4-1 muestra el resultado de ejecutar el programa packbox.cs con un argumento de 1:

Figura 4-1. Empaquetamiento: Cinco variaciones

Cada línea contiene una caja horizontal (hbox) con varios botones. La llamada a pack es una copia de la llamada a pack en cada uno de los botones de la hbox. Cada botón es empaquetado en la hbox de la misma manera (por ejemplo, con los mismos argumentos al método PackStart().

Esto es un ejemplo del método PackStart().

```
box.PackStart(child, expand, fill, padding);
```

box es la caja donde estas empaquetando el objeto; el primer argumento, child, es el objeto que se va a empaquetar. Los objetos serán botones por ahora, por lo que estaremos empaquetando botones dentro de cajas.

El argumento expand de PackStart() y PackEnd() controla si los controles se disponen para ocupar todo el espacio extra de la caja y de esta manera la caja se expande hasta ocupar todo el área reservada para ella ("true"); o si la caja se encoge para ocupar el espacio justo de los controles ("false"). Poniendo expand a "false" te permitirá justificar a la derecha y a la izquierda tus controles. Si no, se expandirán para llenar la caja, y el mismo efecto podría obtenerse usando sólo o PackStart() o PackEnd().

El argumento fill controla si el espacio extra se utiliza en los propios objetos ("true"), como expacio extra en la caja alrededor de los objetos ("false"). Sólo tiene efecto si el argumento expand también es "true".

En Gtk# también se define una versión sobrecargada de PackStart() y PackEnd() para los cuales expand, fill y padding toman valores por defecto.

```
Box.PackStart(child);
Box.PackEnd(child);
```

Al usar esta versión de PackStart() y PackEnd(), expand, fill y padding tomarán los valores: "true", "true" y 0 respectivamente. El argumento child es el único que debe especificarse

Al crear una caja nueva, se debe crear un objeto de las clases HBox o VBox:

```
hbox = new HBox(homogeneous, spacing)
vbox = new VBox(homogeneous, spacing)
```

También contamos con la versión sobrecargada con homogeneous = false y spacing = 0.

```
hbox = new HBox()
vbox = new VBox()
```

El argumento homogeneous del contructor de HBox y VBox() controla si cada objeto en la caja tiene el mismo tamaño (por ejemplo el mismo ancho en una hbox, or el mismo alto en una vbox). Si se usa, las rutinas de empaquetado funcionan basicamente como si el argumento expand estuviera siempre activado.

¿Qué diferencia hay entre spacing (se pone cuando la caja se crea) y padding (se pone cuando los elementos se empaquetan)? El spacing se añade entre objetos, y el padding se añade a cada lado de un objeto. La Figura 4-2 ilustra la diferencia; pasa un argumento de 2 a packbox.cs :

Figura 4-2. Empaquetando con Spacing y Padding

La Figura 4-3 ilustra el uso del método PackEnd() (pasa un argumento de 3 a packbox.cs). La etiqueta "end" se empaqueta con el método PackEnd(). Se mantendrá en el borde derecho de la ventana cuando esta sea redimensionada.

Figura 4-3. Empaquetando con PackEnd()

Programa de Demostración de Empaquetamiento

Uso de Tablas para Empaquetar

Veamos otra manera de empaquetar: Tablas. Pueden ser extremadamente útiles en determinadas situaciones.

Al usar tablas, creamos una rejilla donde podemos colocar los controles. Los controles puede ocupar tantos espacios como especifiquemos.

Lo primero que hay que mirar es obviamente la clase Table y su constructor:

```
table = new Table(rows, columns, homogeneous);
```

El primer argumento es el número de filas de la tabla, mientras que el segundo, obviamente, es el número de columnas.

El argumento homogeneous tiene que ver en el tamaño de las celdas de la tabla. Si homogeneous es "true", las celdas de la tabla tienen el tamaño del mayor control en la tabla. Si homogeneous es "false", el tamaño de las celdas viene dado por el control más alto en su misma fila, y el control más ancho en su columna.

Las filas y las columnas se disponen de 0 a n, donde n es el número que se especificó en la llamada a gtk.Table(). Por tanto, si especificas rows (filas) = 2 y columns (columnas) = 2, la dispoisición quedaría así:

```
0          1          2
0+---------+----------+
|          |          |
1+---------+----------+
|          |          |
2+---------+----------+
```

Fijate que el sistema de coordenadas empieza en la esquina superior izquierda. Para meter un control en una caja, usa el siguiente método:

```
table.Attach(child,
             left_attach, right_attach, top_attach, bottom_attach,
             xoptions, yoptions,
             xpadding, ypadding);
```

table es la instancia de la clase Table que creaste. El primer parámetro (child) es el control que quieres meter en la tabla.

Los argumentos left_attach, right_attach, top_attach y bottom_attach especifican donde colocar el control, y cuantas cajas usar. Si quieres un botón en la esquina inferior derecha de una tabla 2x2, y quieres que ocupe SÓLO ese espacio, left_attach sería = 1, right_attach = 2, top_attach = 1, bottom_attach = 2.

Ahora, si quieres que un control ocupe la fila entera de nuestra tabla 2x2, pondrías left_attach = 0, right_attach = 2, top_attach = 0, bottom_attach = 1.

Los argumentos xoptions e yoptions se usan para especificar opciones de la enumeración AttachOptions de colocación y pueden ser unidas mediante la operación OR (|) permitiendo así múltiples opciones.

Estas opciones son:

AttachOptions.Fill Si la caja es más grande que el control, y especificas AttachOptions.Fill, el control se expandirá hasta usar todo el espacio disponible. AttachOptions.Shrink Si se le asigna menos espacio a la tabla del que solicitó (normalmente porque el usuario ha redimensionado la ventana), entonces los controles normalmente sería empujados a la parte inferior de la ventana y desaparecerían. Si especificas AttachOptions.Shrink, los controles se encojeran con la tabla. AttachOptions.Expand Esto hará que la tabla se expanda para usar el espacio sobrante en la ventana.

El Padding es igual que en las cajas, ya que crea un espacio vacío especificado en pixeles alrededor del control.

También contamos con la versión sobrecargada del constructor de Table

```
table.Attach(child, left_attach, right_attach, top_attach, bottom_attach);

// Valores por defecto:
// xoptions = AttachOptions.Fill | AttachOptions.Expand;
// yoptions = AttachOptions.Fill | AttachOptions.Expand;
// xpadding = 0;
// ypadding = 0;
```

También tenemos los métodos SetRowSpacing() y SetColSpacing() . Añaden espacio entre las filas en la columna o fila especificada.

```
table.set_row_spacing(row, spacing);
table.set_col_spacing(column, spacing);
```

Fijate que para las columnas, el espacio va a la derecha de la columna, y para las filas, el espacio va debajo de la fila.

También puedes poner un espacio consistente para todas las filas y/o columnas con las siguientes propiedades:

```
table.RowSpacing = spacing;
table.ColSpacing = spacing;
```

Fijate que con estas propiedades, la última fila y la última columna no obtienen ningún espacio.

Ejemplo de Empaquetamiento con Tablas

Perspectiva General de Controles

Jerarquía de Controles

Controles sin Ventana

El Control de Botón

Botones Normales

Botones Biestado

Botones de Activación

Botones de Exclusión Mútua

Ajustes

GTK# tiene varios controles que pueden ser ajustados visualmente por el usuario usando el ratón o el teclado, tales como los controles de rango, descritos en la sección Controles de Rango. También hay unos cuantos controles que visualizan una parte ajustable de un área de datos mayor, tales como el control de texto y el control de puerto.

Obviamente, una aplicación necesita ser capaz de reaccionar ante los cambios que el usuario realiza en los controles de rango. Una forma de hacer esto sería que cada control emitiera su propio tipo de señal cuando su ajuste cambiara y, o bien pasa el nuevo valor al manejador de señal, o requiere que se mire dentro de la estructura de datos del control para ver el nuevo valor. Pero puede que también quieras conectar los ajustes de varios controles juntos, para que ajustando uno se ajusten los otros. El ejemplo más obvio de esto es conectar una barra de desplazamiento a un puerto o a un área de texto desplazable. Si cada control tuviera su propia manera de manipular el valor del ajuste, entonces el programador tendría que escribir sus propios manejadores de señales para traducir entre la salida de la señal de un control y la entrada del método de ajuste de otro control.

GTK arregla este problema usando el objeto Adjustment , que no es un control sino una manera de que los controles almacenen y pasen la información de ajuste de una forma abstracta y flexible. El uso más obvio de Adjustment es almacenar los parámetros de configuración y los valores de los controles de rango como las barras de desplazamiento y los controles de escala. Sin embargo, como la clase Adjustment deriva de Object, también tiene unas características especiales más alla de ser estructuras de datos normales. La más importante es que pueden emitir señales, como los controles, y estas señales no sólo pueden ser usadas para permitir a tus programas reaccionar a la entrada de usuario en controles ajustables, sino que pueden propagar valores de ajuste de una forma transparente entre controles ajustables.

Verás como los ajustes encajan entre sí cuando veas otros controles que los incorporan: Barras de Progreso, Puertos, Ventanas de Desplazamiento, y otros.

Crear un Ajuste

Muchos de los controles que usan ajustes lo hacen automáticamente, pero más tarde se mostrarán casos en los que puedes necesitar crearlos por ti mismo. Puedes crear un ajuste usando:

```
Adjustment adjustment = new Adjustment (double value, double lower,
double upper, double step_increment,
double page_increment, double page_size);
```

El argumento value es el valor inicial que quieres darle al ajuste, normalmente corresponde a la posición superior o la posición más a la izquierda de un control ajustable. El argumento lower especifica el valor más bajo que puede tomar el ajuste. El argumento step_increment especifica el incremento más pequeño de los dos incrementos por los que el usuario puede cambiar el valor, mientras que el argumento page_increment es el más grande de los dos. El argumento page_size normalmente se corresponde de alguna manera con el área visible de un control desplazable. El argumento upper se usa para representar la coordenada inferior o la más a la derecha en el hijo de un control desplazable. Por tanto no es siempre el número más grande que el valor puede tomar, ya que el page_size de tales controles normalmente es distinto de cero.

Usar los Ajustes de la Forma Fácil

Los controles ajustables pueden dividirse más o menos en aquellos que usan y requieren unidades específicas para estos valores, y aquellos que los tratan como número arbitrarios. El grupo que trata los valores como números arbitrarios incluye los controles de rango (barras de desplazamiento y escalas, la barra de progreso y los botones de aumentar/disminuir). Todos estos controles normalmente se ajustan directamente por el usuario con el ratón o el teclado. Tratarán los valores inferior y superior de un ajuste como un rango dentro del cual el usuario puede manipular el valor del ajuste. Por defecto, solo modificarán el valor de un ajuste.

El otro grupo incluye el control de texto, el control de puerto, el control de lista compuesta y el control de ventana de desplazamiento. Todos estos controles usan valores de píxeles para sus ajustes. Todos estos controles normalmente se ajustan indirectamente usando barras de desplazamiento. Aunque todos los controles que usan ajustes pueden crear sus propios ajustes o usar los que les proporciones, normalmente querrás dejarles a ellos la tarea de crear sus propios ajustes. Normalmente, sobreescribirán todos los valores de los ajustes que les proporciones, excepto el propio valor, pero los resultado son, en general, impredecibles (lo que significa que tendrás que leer el código fuente para descubrirlo, y puede ser diferente entre los

controles).

Ahora, probablemente estes pensando, ya que los controles de texto y los puertos insisten en establecer todos los parámetros de sus ajustes excepto el valor, mientras que las barras de desplazamiento solo tocan el valor del ajuste, si compartes un objeto ajuste entre una barra de desplazamiento y un control de texto, al manipular la barra de desplazamiento, ¿se ajustará automágicamente el control de texto? ¡Por supuesto que lo hará! Tal y como esto:

```
# crea sus propios ajustes
  Viewport viewport = new Viewport();

# usa los ajustes recién creados para la barra de desplazamiento también
  VScrollbar vscrollbar = new VScrollbar(viewport.Vadjustment);
```

El Interior del Ajuste

Vale, dirás, eso está bien, pero ¿qué pasa si quiero crear mis propios manejadores que respondan cuando el usuario ajuste un control de rango o un botón aumentar/disminuir, y cómo obtengo el valor de un ajuste en estos manejadores? Para contestar a estas preguntas y a otras más, vamos a empezar mirando los atributos de la propia clase Gtk.Adjustment :

```
  lower
  upper
  value
  step_increment
  page_increment
  page_size
```

Sea adj una instancia de la clase Adjustment, cada uno de los atributos se obtienen o modifican usando adj.Lower, adj.Value, etc.

Como se ha dicho antes, Adjustment es una subclase de Object igual que los demás controles, y por tanto, es capaz de emitir señales. Esto es la causa, claro está, de por qué las actualizaciones ocurren automágicamente cuando compartes un objeto ajuste entre una barra de desplazamiento y otro control ajustable; todos los controles ajustables conectan manejadores de señales a la señal "ValueChanged" de sus ajustes, como podría hacerlo tu programa. Aqui tienes la definición de la retrollamada de esta señal:

```
public event EventHandler ValueChanged;
```

Los diversos controles que usan el objeto Adjustment emitirán esta señal en un ajuste siempre que cambien su valor. Esto ocurre tanto cuando el usuario hace que el deslizador se mueva en un control de rango, como cuando el programa explícitamente cambia el valor con la propiedad value. Así, por ejemplo, si tienes un control de escala, y quieres que cambie la rotación de una imagen siempre que su valor cambie, podrías crear una retrollamada como esta:

```
  void cb_rotate_picture (object sender, EventArgs e)
  {
    set_picture_rotation ( ((Adjustment) sender).value);
```

y conectarla al ajuste del control de escala así:

```
adj.ValueChanged += new EventHandler(cb_rotate_picture)
```

¿Y qué pasa cuando un control reconfigura los campos upper (superior) o lower (inferior) de su ajuste, tal y como cuando un usario añade más texto al control de texto? En este caso, se emite la señal "Changed", que es así:

```
public event EventHandler Changed;
```

Los controles Range normalmente conectan un manejador para esta señal, el cual cambia su apariencia para reflejar el cambio - por ejemplo, el tamaño del deslizador de una barra de desplazamiento crecerá o encojerá en proporción inversa a la diferencia entre el valor superior e inferior de su ajuste.

Probablemente nunca tendrás que conectar un manejador a esta señal, a menos que estes escribiendo un nuevo tipo de control de rango. En cualquier caso, si cambias alguno de los valores de un Adjustment directamente, deberías emitir esta señal para reconfigurar los controles que lo están usando, tal que así:

¡Ahora adelante y ajusta!

Miscelánea de Controles

Etiquetas

Flechas

El Objeto Pistas

Barras de Progreso

Diálogos

Imágenes(*)

Reglas

Barras de Estado

Entradas de Texto

Botones Aumentar/Disminuir

Lista Desplegable

Calendario

Selección de Color

Selectores de Fichero

Diálogo de Selección de Fuentes

Controles Contenedores

[editar]

La Caja de Eventos

El Control Alineador

Contenedor Fijo

Contenedor de Disposición

Marcos

Marcos proporcionales

Controles de Panel

Puertos de Visión

Ventanas de Desplazamiento

Cajas de Botones

Barra de herramientas

Fichas

Control Menú

Creación de Menús Manual

Ejemplo de Menú Manual

Usando la Factoria de Elementos

Ejemplo de Factoria de Elementos

Arrastrar y Soltar

Perspectiva General de Arrastrar y Soltar

Propiedades de Arrastrar y Soltar

Métodos de Arrastrar y Soltar

Configuración del Control Orígen

Señales en el Control Orígen

Configuración de un Control Destino

Señales en el Control Destino

Ficheros rc de GTK

Funciones para Ficheros rc

Cuando comience tu aplicación, debes incluir una llamada a:

```
Gtk.Rc.Parse(string filename)
```

Pasándole el nombre de tu fichero rc en filename. Esto hace que GTK analice este fichero, y use los valores de estilo para los tipos de controles que definas alli.

Si quieres tener un conjunto especial de controles que puedan tener un estilo diferente a los demás, o cualquier otra división lógica de controles, esta es la propiedad del widget

```
string Widget.Name
```

El nombre que especifiques se le asignará a tu control recién creado. Esto te permitirá cambiar los atributos de este control en el fichero rc.

Si usamos una llamada parecida a:

```
Gtk.Button boton = new Gtk.Button("Boton especial");
button.Name = "special button";
```

Entonces se le dará el nombre "special button" a este botón y se podrá localizar en el fichero rc como "special button.GtkButton".

El ejemplo de fichero rc que hay más abajo, cambia las propiedades de la ventana principal, y deja a todos sus hijos que hereden el estilo descrito para el "main button" (botón principal). El código usado por la aplicación es:

```
Gtk.Window window = new Gtk.Window("Ventana Principal");
Window.Name = "main window"
```

Y el estilo se define en el fichero rc usando:

```
widget "main window.*GtkButton*" style "main_button"
```

Lo cual le pone a todos los controles Button en la "main window" el estilo "main_buttons" tal y como se define en el fichero rc.

Como puedes ver, este es un sistema bastante potente y flexible. Usa tu imaginación para sacarle el mejor provecho.

Formato de los Ficheros rc de GTK

El formato del fichero rc se muestra en el siguiente ejemplo . Este es el fichero testgtkrc de la distribución GTK, pero le he añadido unos cuantos comentarios y cosas. Puedes incluir esta explicación en tu programa para permitirle al usuario afinar su aplicación.

Hay varias directivas para cambiar los atributos de un control.

fg - Cambia el color de frente de un control.

bg - Cambia el color de fondo de un control.

bg_pixmap - Cambia el fondo de un control para que sea un mosaico con el pixmap dado.

font - Cambia la fuente que usará el control.

Además de esto, hay varios estados en los que un control puede estar, y puedes cambiar los diferentes colores, pixmaps y fuentes para cada estado. Los estados son:

NORMAL (Normal) El estado normal de un control, sin que el ratón esté encima de él, y sin que haya sido pulsado, etc.

PRELIGHT (Preiluminado) Cuando el ratón está encima del control, los colores definidos en este estado tendrán efecto.

ACTIVE (Activo) Cuando el control está presionado o se ha hecho clic en él estará activo y los atributos asignados con este valor tendrán efecto.

INSENSITIVE (Insensitivo) Cuando un control está insensitivo, y no puede ser activado, tendrá estos atributos.

SELECTED (Seleccionado) Cuando un objeto está seleccionado, se usan estos atributos.

Cuando se usan las palabras "fg" y "bg" para cambiar los colores de los controles, el formato es:

```
fg[<STATE>] = { Rojo, Verde, Azul }
```

Donde STATE es uno de los estados anteriores (PRELIGHT, ACTIVE, etc), y el Rojo, Verde y Azul son valores en el rango de 0 - 1.0, siendo { 1.0, 1.0, 1.0 } el blanco. Deben estar en formato float, o se les asignará 0, por tanto un simple "1" no funcionará, debe ser "1.0". Un simple "0" esta bien ya que da igual si no se reconoce porque los valores no reconocidos se ponen a 0.

bg_pixmap es muy similar a lo anterior excepto que los colores se sustituyen por un nombre de fichero.

pixmap_path es una lista de caminos separados por ":" . Cuando especifiques un pixmap, se buscará en esta lista.

La directiva "font" es simple:

```
font = ""
```

Lo único dificil es saber la cadena de la fuente. El programa xfontsel o una utilidad similar ayuda.

La directiva "widget_class" fija el estilo para una clase de controles. Estas clases se listan en la perspectiva general de controles en la jerarquía de clases.

La directiva "widget" fija el estilo de un conjunto especifico de controles con un nombre, reemplazando cualquier estilo de la clase del control en cuestión. Estos controles se registran en la aplicación usando el método set_name() . Esto te permite cambiar los atributos de un control de una forma mucho más concreta, en vez de especificar los atributos de toda su clase. Te recomiendo que documentes todos estos controles especiales para que los usuarios puedan personalizarlos.

Cuando se usa la palabra parent (padre) en un atributo, el control usará los atributos de su padre en la aplicación.

Al definir un estilo, puedes asiganr los atributos de un estilo definido previamente al estilo actual.

```
style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

Este ejemplo usa el estilo "button", y crea un nuevo estilo "main_button" simplemente cambiando la fuente y el color de fondo del estado preiluminado del estilo "button".

Por supuesto, muchos de estos atributos no funcionan con todos los controles. Solo es cuestión de sentido común. Todo lo que podría aplicarse, se debe aplicar.

Ejemplo de fichero rc

pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."

pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"

```
style <name> [= <name>]
{
  <option>
}
```

widget <widget_set> style <style_name> widget_class <widget_class_set> style <style_name>

Esto crea un estilo llamado "window". El nombre no es importante ya que se le asigna a los controles reales al final del fichero.

```
 style "window"
 {
   //Esto le pone el pixmap especificado como margén alrededor de la ventana.
   //bg_pixmap[<STATE>] = "<pixmap filename>"
   bg_pixmap[NORMAL] = "warning.xpm"
 }

 style "scale"
 {
   //Pone el color de frente (el color de la fuente) a rojo en el estado
   //"NORMAL".

   fg[NORMAL] = { 1.0, 0, 0 }

    //Pone el pixmap de fondo de este control al mismo de su padre.
   bg_pixmap[NORMAL] = "<parent>"
 }

 style "button"
 {
    // Esto muestra los posibles estados de un botón. El único que no tiene es
    // el estado SELECTED .

 fg[PRELIGHT] = { 0, 1.0, 1.0 }
 bg[PRELIGHT] = { 0, 0, 1.0 }
 bg[ACTIVE] = { 1.0, 0, 0 }
 fg[ACTIVE] = { 0, 1.0, 0 }
 bg[NORMAL] = { 1.0, 1.0, 0 }
 fg[NORMAL] = { .99, 0, .99 }
 bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
 fg[INSENSITIVE] = { 1.0, 0, 1.0 }
 }
```

En este ejemplo, heredamos los atributos del estilo "button" y reemplazamos el color de fondo y la fuente del estado PRELIGHT para crear el estilo "main_button".

```
style "main_button" = "button"
{
 font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
 bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
 fg[NORMAL] = { 1.0, 0, 0 }
 fg[ACTIVE] = { 1.0, 0, 0 }

 // Esto le pone el pixmap de fondo al botón biestado al que esté usando el
 // padre (tal y como se defina en la aplicación)
 bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
 bg_pixmap[NORMAL] = "marble.xpm"
 fg[NORMAL] = { 1.0, 1.0, 1.0 }
```

```
}

style "ruler"
{
 font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}
```

pixmap_path "~/.pixmaps"

Le decimos que estos tipos de controles usen los estilos definidos arriba. Los tipos de controles se listan en la jerarquía de controles, pero probablemente se podrían listar en este documento para referencia del usuario.

```
widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"
```

Le decimos que todos los botones que sean hijos de la ventana "main window" tengan el estilo "main_button". Esto debe ser documentado para que se le saque provecho.

```
  widget "main window.*GtkButton*" style "main_button"
```

Trucos para escribir aplicaciones GTK#

A. Señales GTK

Gtk.Object

Gtk.Widget

Gtk.Container

Gtk.Calendar

Gtk.Editable

Gtk.Notebook

Gtk.ListStore

Gtk.MenuShell

Gtk.Toolbar

Gtk.Button

Gtk.Item

Gtk.Window

Gtk.HandleBox

Gtk.ToggleButton

Gtk.MenuItem

Gtk.CheckMenuItem

Gtk.InputDialog

Gtk.ColorSelection

Gtk.Statusbar

Gtk.Curve

Gtk.Adjustment

B. Ejemplos de Código

scribblesimple.cs

Lista de tablas

Tabla de figuras