Layout Managers

Martín Pérez Mariñán
martin@kristalnetworks.com

Los layout managers son uno de los conceptos más útiles que podemos encontrar en Java. Gracias a ellos podremos organizar todos los componentes de nuestra interfaz gráfica de modo que sea más sencillo añadirlos, eliminarlos o recolocar su posición. Los layout managers automatizan una gran cantidad de trabajo y eliminan al programador la necesidad de realizar tediosas tareas de control del interfaz. En este artículo veremos los layout managers más comunes y aprenderemos a dominarlos.

Introducción a los *layout managers*

Una traducción libre del término *layout manager* sería manejador de contenido y en realidad eso es lo que es. Un *layout manager* no es más que un delegado[5] que se encarga de organizar los componentes que forman parte de un contenedor como por ejemplo pueda ser una ventana. El *layout manager* es el encargado de decidir en que posiciones se renderizarán los componentes, que tamaño tendrán, que porción del contenedor abarcarán, etc... Todo esto se realiza de una manera transparente al programador que por lo tanto se ahorra el tener que escribir una gran cantidad de líneas de control.

Ventajas y desventajas

Los *layout managers* tienen una gran cantidad de ventajas:

- Encapsulan parte de la lógica de presentación de nuestro interfaz gráfico de modo que evitan al programador tener que escribir una gran cantidad de líneas de código. Además hacen este código mucho más sencillo de leer y por lo tanto más mantenible.
- Reorganizan automáticamente los componentes del interfaz de modo que siempre se ajuste a las directivas que hemos establecido previamente. Si el usuario en un momento dado decide maximizar el interfaz gráfico éste mantendrá su aspecto original en la medida de lo posible. De este modo no limitamos al usuario a un formato de pantalla determinado.
- Hacen más sencillo la labor de añadir, modificar y eliminar componentes. En un diseño tradicional cuando nos vemos obligados a añadir un componente en un lugar donde ya

existen varios, seguramente tengamos que mover el resto de componentes del interfaz gráfico para acomodar a nuestro nuevo inquilino. Utilizando *layout managers* lo único que tenemos que hacer es agregar el componente y el *layout manager* se encarga automáticamente de reorganizar todo el interfaz.

- Hacen nuestro interfaz mucho más portable. Esto se debe a que los componentes gráficos no tienen las mismas propiedades en todos los sistemas operativos. Un botón que muestre la cadena “Hola Mundo” en Mac no tendrá las mismas dimensiones que su homónimo en Linux o Windows. Al realizar nuestro programa con *layout managers*, éstos ya se encargan de ajustar los componentes adecuadamente y nos evitamos problemas inesperados.

Ciertamente existen también una serie de desventajas asociadas a los *layout managers*:

- Requieren una buena dosis de aprendizaje y práctica. Muchos programadores odiarán los *layout managers* ya que pueden resultar una experiencia traumática sin un buen aprendizaje. De todos modos, una vez dominados, son pocos los programadores que dejan de utilizarlos.

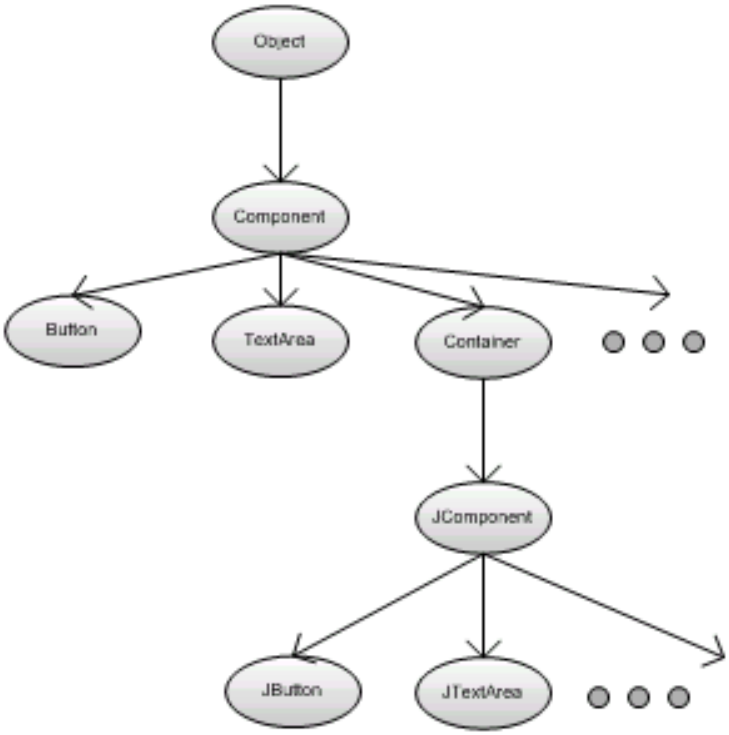
- Pueden generar interfaces muy pesadas. A menudo las primeras veces que creemos nuestros *layouts* nos encontraremos con que acabamos con un gran número de paneles anidados. Los paneles son objetos bastante pesados por lo que hay que tener cuidado de no sobrecargar innecesariamente nuestra interfaz gráfica.

Contenedores y Componentes

Para poder entender el funcionamiento de los *layout managers* es necesario una pequeña base sobre lo que son los contenedores y los componentes.

Un contenedor es un componente Java que puede contener otros componentes[1]. La clase principal es *java.awt.Component* de la cual se heredan componentes como *java.awt.Button, java.awt.Label*, etc..., y también se hereda la clase *java.awt.Container* que representa a un objeto contenedor. En la siguiente figura podemos ver un extracto de la jerarquía de clases de *AWT* y *swing*.

1.Implementación clásica del patrón Composite[5].



Un aspecto muy importante a tener en cuenta es que cada contenedor tiene un *layout manager* establecido por defecto. En la siguiente tabla podemos ver los *layout managers* asignados por defecto a cada contenedor.

Contenedor	Layout Manager
Panel	FlowLayout
Applet	FlowLayout
Frame	BorderLayout
Dialog	BorderLayout
ScrollPane	FlowLayout

Tabla 1-1: *layouts* por defecto de los contenedores *AWT*

Los layouts por defecto para los contenedores de *swing* son los mismos que para sus homónimos de *AWT*.

Si queremos cambiar el *layout manager* de un contenedor en un momento dado, tan sólo tendremos que llamar al método:

```
contenedor.setLayout(LayoutManager layout);[1]
```

Si en cualquier momento decidimos que estamos hartos de un *layout manager* y queremos encargarnos nosotros mismos de la gestión de componentes tan sólo tendremos que escribir:

```
contenedor.setLayout(null);
```

Los componentes como hemos dicho son objetos que forman parte de nuestro interfaz gráfico. Cada componente tiene asignada una coordenada horizontal, una coordenada vertical, una longitud y una anchura determinadas. Estos valores serán los que se utilizarán para renderizar el componente en pantalla.

La clase *java.awt.Component* nos ofrece una serie de métodos para poder modificar los valores de los atributos anteriores:

```
public void setSize(Dimension size);
public void setBounds(Rectangle r);
```

Hay pequeñas variaciones de estos métodos pero su función es la misma. Por su parte la clase *javax.swing.JComponent* tiene métodos diferentes:

```
public void setPreferredSize(Dimension size);
public void setMinimumSize(Dimenstion size);
public void setMaximumSize(Dimension size);
```

Es muy frustrante, cuando conoces el funcionamiento de los layouts, ver que aunque utilizas estos métodos, el intérprete Java no les hace ningún caso y los objetos muestran un tamaño que nosotros no queríamos. Veamos un ejemplo:

```
....
JButton button = new JButton();
button.setMaximumSize(new Dimension(80,25));
JFrame frame = new JFrame();
frame.getContentPane().add(button);
frame.setSize(400,300);
frame.setVisible(true);
....
```

1.Aunque no lo he dicho, todos los *layout managers* implementan la interfaz *LayoutManager* directa o indirectamente (a través de *LayoutManager2*).

Si creamos un pequeño programa que contenga las líneas de código anteriores y lo ejecutamos, inicialmente podríamos pensar que se verá un botón de 80 puntos de largo por 25 de ancho. Sin embargo, tras ejecutar el programa veremos como nuestro botón es considerablemente más grande. ¿Qué es lo que ha pasado? ¿Por qué no nos ha hecho caso? La realidad es que cuando utilizamos *layout managers* no sirve de nada intentar establecer a la fuerza el tamaño de los componentes ya que será el *layout* el que siempre tenga la última palabra[1]. En el ejemplo anterior, si que se establece el tamaño máximo de 80x25 para el botón, lo que pasa es que después de añadirlo al *frame* es el *layout manager* de éste el que toma el control del interfaz y el que pasa a decidir el nuevo tamaño del botón, por lo que nuestra llamada a *setMaximumSize()* no hace ningún efecto.

El último aspecto importante a tener en cuenta es el significado del atributo *preferredSize*. Este atributo indica el tamaño que a un componente le gustaría tener. Este tamaño suele ser dependiente de la plataforma y el *layout manager* intentará respetarlo siempre que sea posible y siempre que lo permitan sus políticas de *layout*.

FlowLayout

El primer *layout manager* que vamos a ver es también el más simple. *FlowLayout* coloca los componentes en filas horizontales. *FlowLayout* es el *layout manager* por defecto de paneles y *applets*.

FlowLayout respeta siempre el tamaño preferido de cada componente. Cuando queremos insertar un componente y no hay más espacio en la fila actual, el elemento se insertará en la fila siguiente. Los componentes de cada fila se encuentran equiespaciados. Podemos controlar la alineación de los elementos en las filas utilizando los atributos estáticos *FlowLayout.LEFT*, *FlowLayout.CENTER* y *FlowLayout.RIGHT*.

Por defecto *FlowLayout* deja un espacio de cinco puntos tanto horizontal como vertical entre componentes. *FlowLayout* tiene varios constructores con los que podemos modificar este espaciado y también la alineación de los componentes.

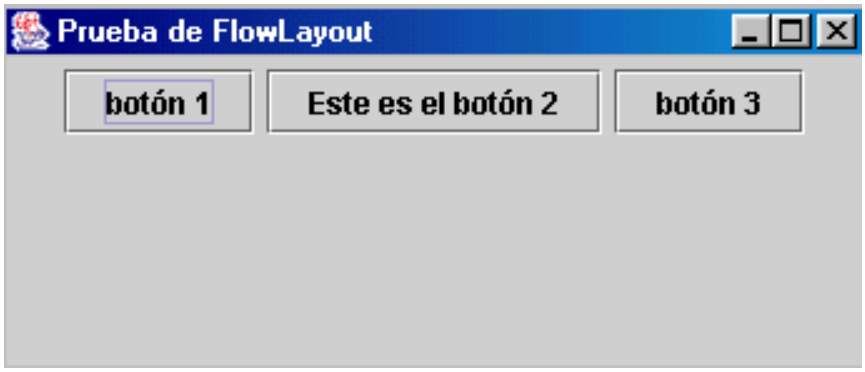
Veamos un ejemplo de funcionamiento de este *layout manager*:

```
import javax.swing.*;
```

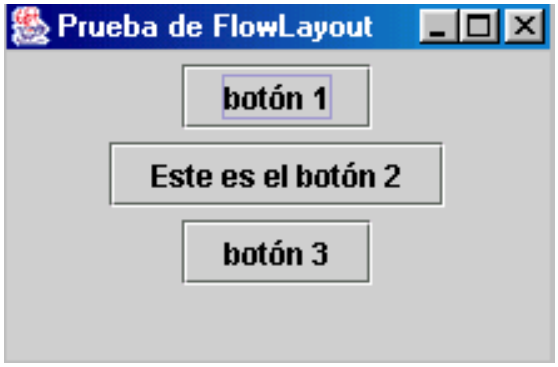
1.Técnicamente si que existen formas de forzar un tamaño a los componentes aunque se estén utilizando *layouts* pero no tiene ningún sentido el hacerlo ya que perderíamos la ventaja que éstos nos ofrecen.

```
public class TestFlowLayout extends JFrame {

    public static void main(String[] args) {

        TestFlowLayout frame = new TestFlowLayout();
        JPanel panel = new JPanel();
        JButton boton1 = new JButton("botón 1");
        JButton boton2 = new JButton("Este es el botón 2");
        JButton boton3 = new JButton("botón 3");
        panel.add(boton1);
        panel.add(boton2);
        panel.add(boton3);
        frame.setContentPane(panel);
        frame.setSize(350,150);
        frame.setTitle("Prueba de FlowLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Al compilar y ejecutar este programa nos debería salir una ventana como la de la figura 2.



Si modificamos las dimensiones de la ventana podemos ver como los elementos se reorganizan según el espacio disponible. Por ejemplo si hacemos que la ventana sea muy estrecha veremos en la figura siguiente como cada botón se sitúa en su propia fila.



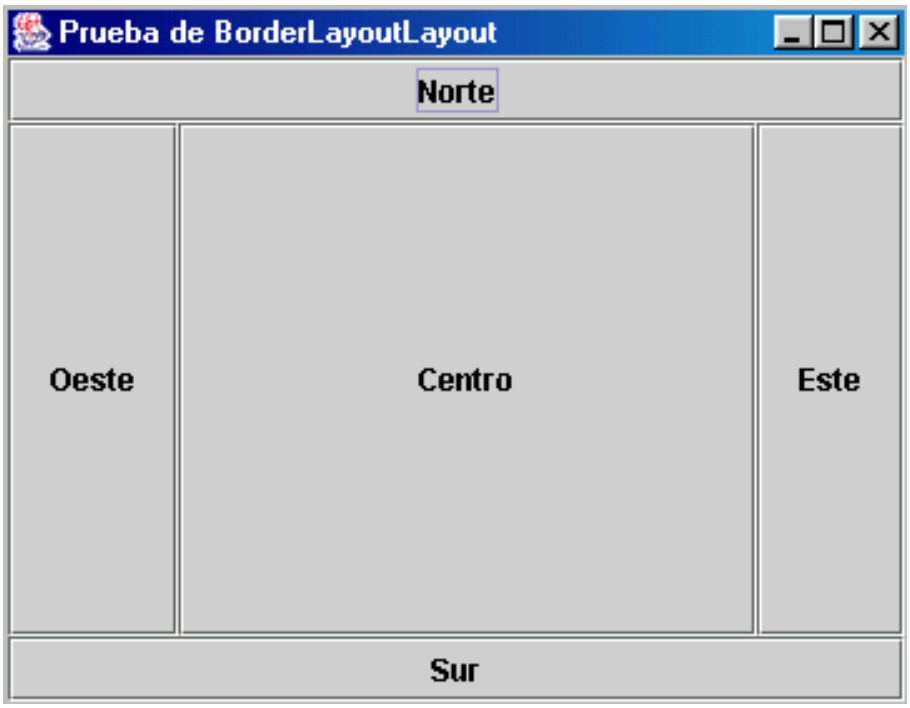
BorderLayout

BorderLayout es el *layout manager* por defecto para *frames* por lo que al igual que *FlowLayout* su aprendizaje es indispensable. *BorderLayout* divide el espacio de un contenedor en cinco regiones diferentes. Estas regiones son: *North*, *South*, *East*, *West* y *Center*, y se corresponden con su situación dentro del contenedor en el que se encuentran.

Veamos más claramente lo que quiere decir esto con un ejemplo sencillo:

```
import javax.swing.*;
import java.awt.*;

public class TestBorderLayout extends JFrame {

    public static void main(String[] args) {

        TestBorderLayout frame = new TestBorderLayout();
        Container panel = frame.getContentPane();
        JButton norte = new JButton("Norte");
        JButton sur= new JButton("Sur");
        JButton este= new JButton("Este");
        JButton oeste = new JButton("Oeste");
        JButton centro = new JButton("Centro");
        panel.add(norte, BorderLayout.NORTH);
        panel.add(sur, BorderLayout.SOUTH);
        panel.add(este, BorderLayout.EAST);
        panel.add(oeste, BorderLayout.WEST);
        panel.add(centro, BorderLayout.CENTER);
        frame.setSize(400,300);
```

```
                frame.setTitle("Prueba de BorderLayoutLayout");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        }
```



Como podemos ver en el código fuente anterior, al añadir un elemento al *BorderLayout* tenemos que especificar la región en la cual lo queremos añadir. Si no especificamos ninguna región por defecto el componente se inserta en el centro del contenedor. Aunque ya las hemos visto en el código las resumiremos de nuevo:

```
BorderLayout.NORTH
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.CENTER // Por defecto
```

Existe una manera alternativa de especificar donde queremos colocar los componentes y es pasando como parámetros al método *add* en lugar de los atributos anteriores las cadenas siguientes:

```
"North"
"South"
```

```
"East"
"West"
"Center"
```

Este método no está recomendado porque es demasiado propenso a errores, ya que si nos equivocamos en una simple letra el compilador no nos avisará de nuestro error y por lo tanto podremos llevarnos sorpresas al ejecutar el programa.

Si insertamos un componente en una región donde había otro componente previamente, el que se encontraba en el contenedor desaparecerá y el nuevo ocupará su lugar. Por lo tanto tenemos que tener cuidado con donde insertamos los componentes.

Para finalizar con este *layout manager* vamos a hablar de como trata el tamaño de los componentes. Cuando añadimos un componente en las posiciones norte o sur, *BorderLayout* respeta su alto mientras que la longitud del mismo se ajusta hasta ocupar todo el ancho del contenedor. Con los componentes de las posiciones este y oeste pasa lo contrario, *BorderLayout* respeta su longitud y ajusta su altura hasta que ocupe la totalidad de la altura del contenedor o hasta que se encuentre con los componentes del norte o del sur. Por último el objeto que se situe en la zona central ocupará el resto de espacio disponible.

BorderLayout, por su estructura, es muy útil para muchas aplicaciones. Por ejemplo, sería bastante normal colocar una barra de herramientas en el panel norte de nuestra ventana, una barra de estado en el panel sur y quizás un árbol de navegación en el panel izquierdo o derecho, dejando el panel central para el resto del interfaz.

CardLayout

CardLayout es un *layout manager* ligeramente diferente a todos los demás ya que tan sólo muestra en un instante dado un único componente. Un contenedor que tenga asignado un *CardLayout* podrá tener cualquier número de componentes en su interior pero sólo uno se verá en un instante dado.

En este *layout manager* los componentes ocuparán todo el tamaño disponible en el contenedor. Los componentes a medida que se insertan en el contenedor van formando una secuencia. Para seleccionar el componente que queremos mostrar en cada momento disponemos de varios métodos:

```
public void first(Container contenedor);
public void last(Container contenedor);
public void next(Container contenedor);
```

```
public void previous(Container contenedor);
public void show(Container container, String nombre);
```

El método más común para añadir un componente es:

```
public void add(Component componente, String nombre);
```

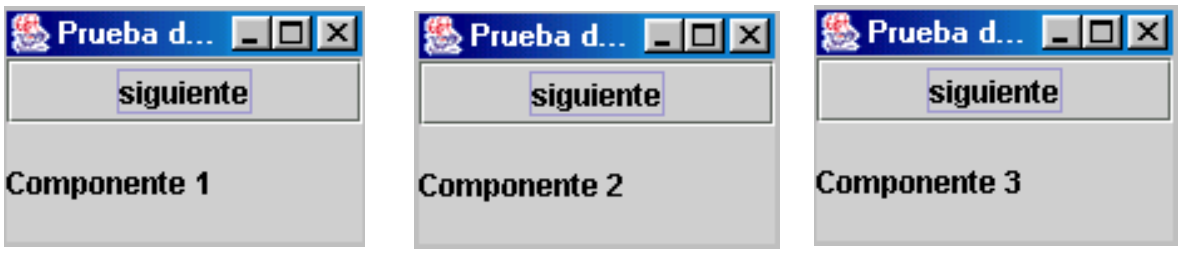
Este método inserta un componente dentro de un contenedor y le asigna un nombre, este nombre lo podremos utilizar con el método *show* para mostrar el componente directamente.

Por último, al añadir componentes tendremos que fijarnos en que el orden en el que los añadamos al contenedor será el orden en el que serán recorridos por el *layout manager*.

Veamos un ejemplo simple:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestCardLayout extends JFrame {

    public static void main(String[] args) {

        TestCardLayout frame = new TestCardLayout();
        Container container = frame.getContentPane();
        JButton siguiente = new JButton("siguiente");
        container.add(siguiente, BorderLayout.NORTH);

        JLabel label1 = new JLabel("Componente 1");
        JLabel label2 = new JLabel("Componente 2");
        JLabel label3 = new JLabel("Componente 3");
        JLabel label4 = new JLabel("Componente 4");

        JPanel panelComponentes = new JPanel();
        CardLayout layout = new CardLayout();

        panelComponentes.setLayout(layout);
        panelComponentes.add(label1,"1");
        panelComponentes.add(label2,"2");
        panelComponentes.add(label3,"3");
        panelComponentes.add(label4,"4");
        container.add(panelComponentes, BorderLayout.CENTER);

        siguiente.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layout.next(panelComponentes);
            }
        });
```

```
                frame.setSize(400,300);
                frame.setTitle("Prueba de BorderLayoutLayout");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        }
```



El ejemplo es muy sencillo y muestra un contenedor con varias etiquetas. Con un botón podemos ir avanzando de etiqueta. Al pulsar el botón se muestra la etiqueta siguiente llamando al método *next()* de *CardLayout*.

Este *layout manager* es muy sencillo y muy útil especialmente cuando tenemos un panel que variará su contenido en función de alguna parte de nuestro interfaz (una *combo box* por ejemplo). En lugar de eliminar el panel e insertar otro nuevo, o en lugar de eliminar los componentes e insertar otros nuevos, podemos utilizar un *CardLayout* que nos ahorra gran cantidad de trabajo.

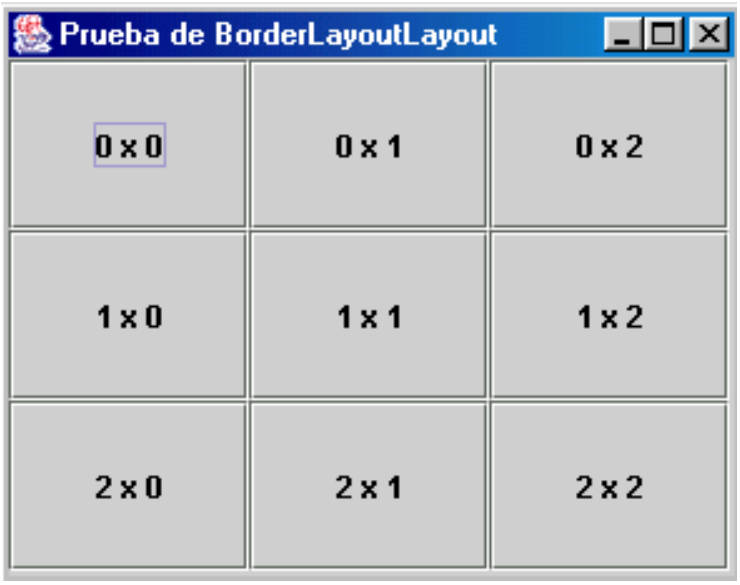
GridLayout

GridLayout divide el espacio de un contenedor en forma de tabla, es decir, en un conjunto de filas y columnas. Cada fila y cada columna tiene el mismo tamaño y el área del contenedor se distribuye equitativamente entre todas las celdas. De todo esto se deduce que *GridLayout* no respetará el tamaño preferido de los componentes que insertemos en cada una de las celdas.

El número de filas y columnas se especifica en el constructor. Si pasamos cero como el número de filas o columnas el *layout manager* irá creando las filas y columnas en función del número de componentes y del valor de la otra dimensión, es decir, si creamos un *GridLayout* con cero filas y tres columnas e insertamos cuatro componentes el *GridLayout* será lo suficientemente inteligente como para saber que tiene que crear dos filas.

Veamos un ejemplo simple de funcionamiento:

```
import javax.swing.*;
import java.awt.*;

public class TestGridLayout extends JFrame {

      public static void main(String[] args) {

            TestGridLayout frame = new TestGridLayout();
            Container container = frame.getContentPane();

            int X = 3; int Y = 3;
            container.setLayout(new GridLayout(X,Y));
            for (int i = 0; i < X; i++) {
                  for (int j = 0; j < Y; j++) {
                        container.add(new JButton(i + " x " + j));
                  }
            }

            frame.setSize(400,300);
            frame.setTitle("Prueba de BorderLayoutLayout");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
}
```



En la figura anterior vemos el resultado de ejecutar el ejemplo. Como se puede ver todas las celdas tienen el mismo tamaño y los botones ocupan la totalidad de la celda.

GridLayout es un *layout manager* realmente muy simple y como tal su utilidad se encuentra bastante reducida. Suele ser útil a la hora de crear partes del interfaz de usuario que necesiten representar una matriz de componentes o incluso para interfaces que tengan en si una forma matricial.

Lo realmente interesante sería que algunasceldas pudiesen tener tamaños diferentes a las demás, o que una celda pudiese ocupar varias posiciones, o dejar celdas vacías, etc... Afortunadamente, esto es justamente lo que nos permite el próximo *layout manager* que veremos, *GridBagLayout*.

GridBagLayout

GridBagLayout es el *layout manager* más poderoso y eficaz con mucha diferencia. Con *GridBagLayout* podemos imitar facilmente el comportamiento del resto de *layout managers* a parte de poder crear con el interfaces mucho más complejas.

Ventajas y desventajas

GridBagLayout es el *layout manager* que más pavor causa entre los programadores Java. Odiado por unos y alabado por otros, este *layout manager* proporciona una serie de ventajas sobre el resto:

- Permite la creación de interfaces de usuario complejos. Con este *layout manager* tenemos control absoluto sobre las posiciones que ocuparán los objetos en el interfaz final.
- Las interfaces construidas son más ligeras. Cuando queremos crear un interfaz de usuario combinando el resto de *layout managers* vistos hasta el momento a menudo terminamos con un número grande de paneles anidados. Los paneles son objetos bastante pesados y tener una gran cantidad de los mismos puede influir perjudicialmente en el rendimiento de nuestro programa. Con *GridBagLayout* se pueden crear interfaces exactamente iguales pero con un único panel con lo que nuestra interfaz será mucho más ligera.

Pero como todo, también tiene sus inconvenientes:

- Requiere un tiempo de aprendizaje bastante grande. No sólo es necesario comprender su funcionamiento sino que también es necesario haber hecho bastantes ejemplos para

llegar a dominarlo.

- El código necesario para crear un interfaz de usuario es considerablemente más grande que con el resto de *layout managers* y además suele ser un código bastante complicado y díficil de comprender y por lo tanto de mantener.

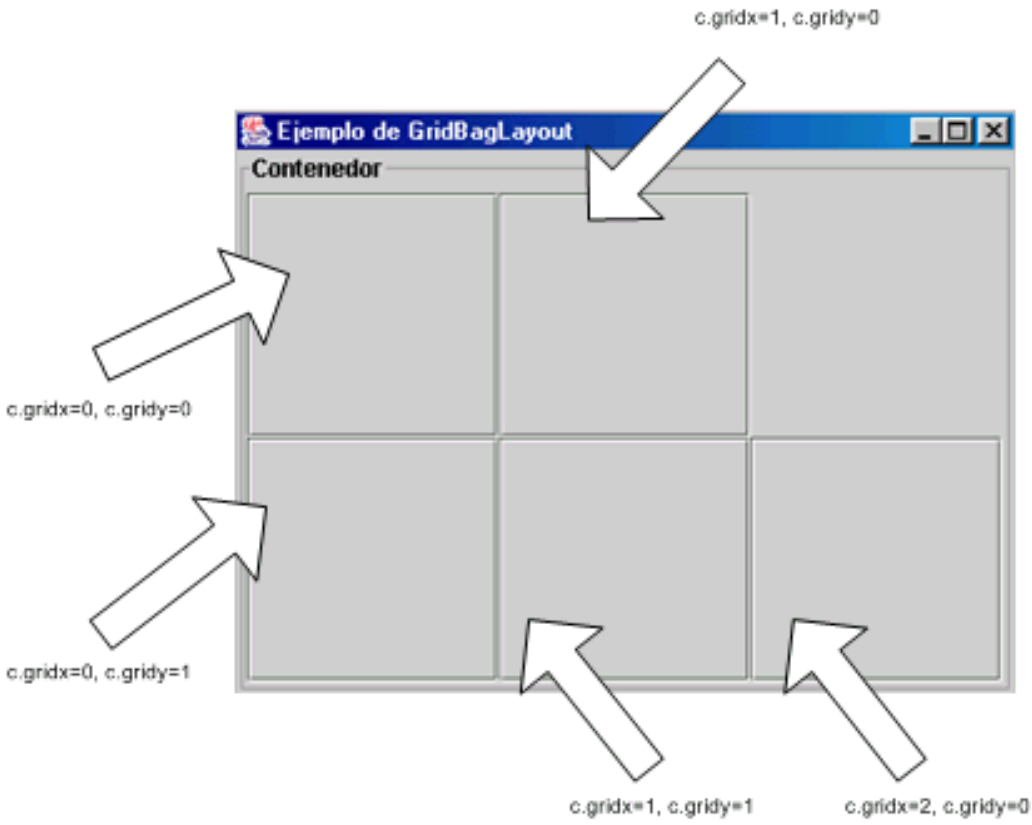
En la última parte de este apartado veremos como podemos crear una serie de clases auxiliares con las que solucionaremos estos dos inconvenientes y que harán que utilizar *GridBagLayout* sea un juego de niños.

GridBagLayout a fondo

GridBagLayout basa su funcionamiento en una clase auxiliar que establece restricciones a los componentes, *GridBagConstraints*. Estas restricciones especifican exactamente como se mostrará cada elemento dentro del contenedor. La clase *GridBagConstraints* posee bastantes atributos que nos permiten configurar el *layout* de un contenedor a nuestro gusto. A continuación vamos a ver los atributos importantes; no voy a mostrar la totalidad de atributos para no complicar las cosas a los menos experimentados. Vamos a ir viendo los atributos más importantes de esta clase y de una forma gráfica para que se entienda mejor:

gridx y gridy

Estos dos atributos especifican las coordenadas horizontal y vertical del componente que vamos a insertar en el grid. Realmente no siempre es necesario establecer su valor ya que en los casos más simples nos llegaría con *gridwidth* y *gridheight*, sin embargo la experiencia dice que poner este atributo es recomendable ya que permite saber en que elemento nos encontramos de una manera visual. La siguiente figura muestra gráficamente lo que indican los atributos *gridx* y *gridy*:

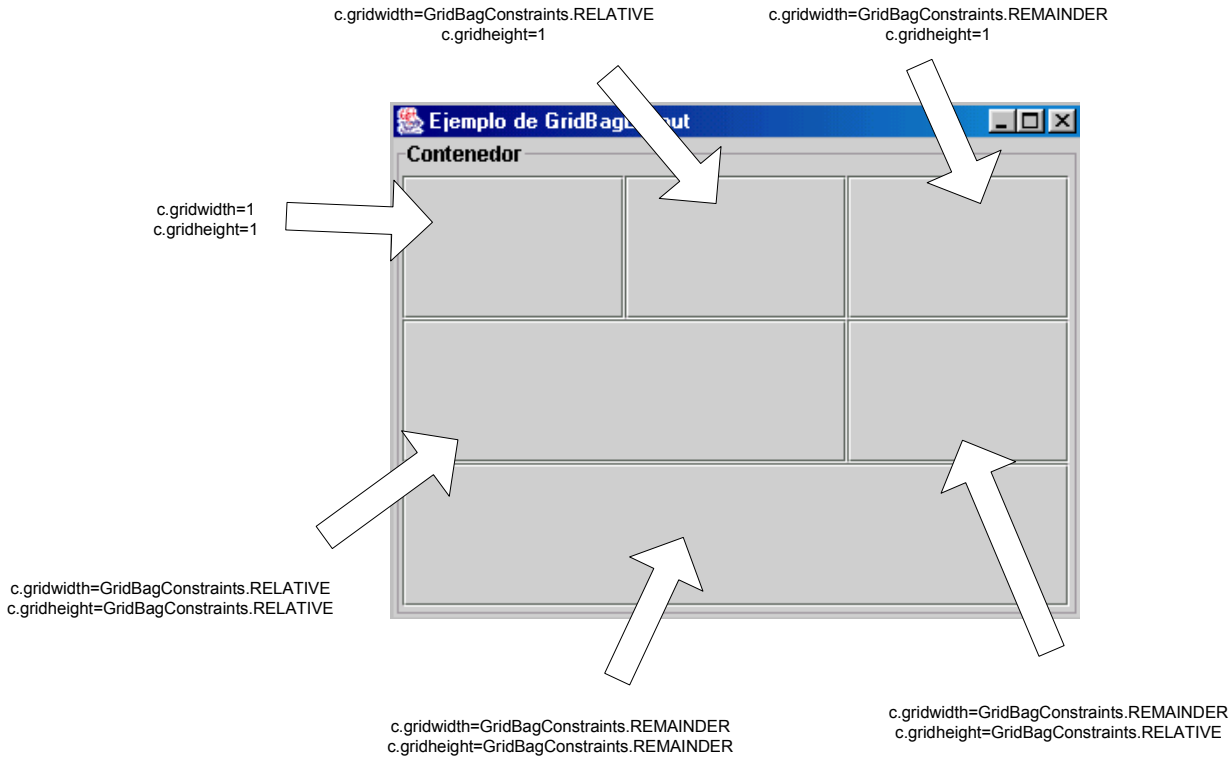


gridwidth y gridheight

Este otro par de elementos junto con *gridx* y *gridy* son la base de *GridBagLayout*. Comprendiendo a la perfección su significado no habrá ningún interfaz que se nos resista. Básicamente lo que indican *gridwidth* y *gridheight* es el número de celdas que ocupará un componente dentro del *GridBagLayout*, su valor puede ser:

- Un número cardinal, en este caso indica exactamente el número de filas o columnas que ocupará el componente.
- *GridBagConstraints.RELATIVE*, indica que el componente ocupará el espacio disponible desde la fila o columna actual hasta la última fila o columna disponibles.
- *GridBagConstraints.REMAINDER*, indica que el componente es el último de la fila actual o columna actual.

Veamoslo gráficamente:



Analicemos la figura anterior para comprender el significado de estos dos atributos.

En la primera fila tenemos tres componentes. Todos los componentes ocupan una celda en horizontal y en vertical luego su *gridwidth* y *gridheight* ha de ser igual a uno. En el segundo y tercer componente en lugar de ponerle uno como valor de *gridwidth* hemos de poner *RELATIVE* y *REMAINDER.* Técnicamente sólo sería necesario poner *REMAINDER* ya que es el único que necesita conocer el *GridBagLayout* para saber que se ha acabado la fila.[1]

En la segunda fila tenemos dos componentes. Ambos tienen como *gridheight RELATIVE* ya que se encuentran en la penúltima fila. El primer componente tiene de *gridwidth REMAINDER*, es decir, ocupará todo el espacio hasta el último componente. En la figura se puede ver como se cumple esto que acabo de decir ya que el componente abarca dos celdas y la última se deja para el último cuyo *gridwidth* es *RELATIVE*.

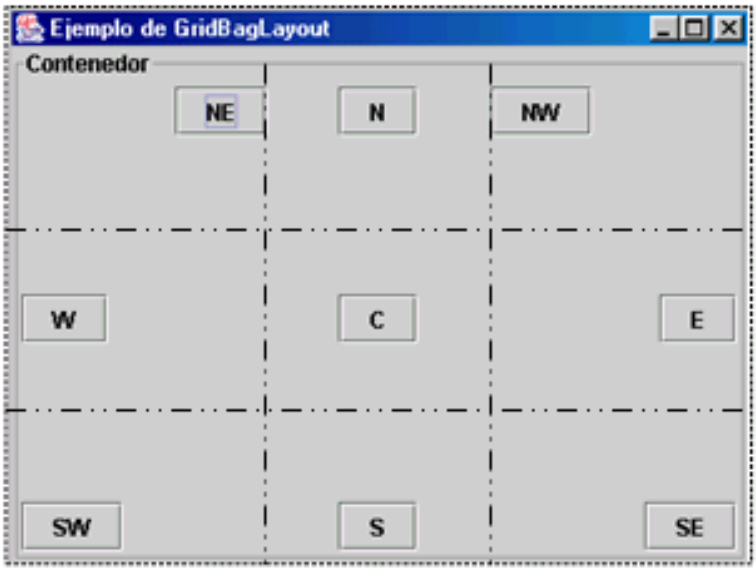
1. Siempre y cuando no se use *gridx* y *gridy*.

Por último en la tercera fila tan sólo tenemos un componente con *gridwidth REMAINDER* y *gridheight REMAINDER*.

Como veis es bastante sencillo, *gridwidth* y *gridheight* especifican el número de celdas horizontal y vertical que abarcará un componente. Además podemos utilizar los valores especiales *REMAINDER* y *RELATIVE* para indicar que un componente ha de ocupar todo el espacio restante o todo el espacio hasta el último componente.

anchor

Este atributo es mucho más sencillo; *anchor* especifica la posición que ocupará un componente dentro de una celda. Los valores que puede tomar este atributo están definidos como variables estáticas dentro de la clase *GridBagConstraints* y son: *NORTH, SOUTH, EAST, WEAST, NORTHWEST, SOUTHWEST, NORTHEAST, SOUTHEAST* y *CENTER*. Como intuiréis indican la orientación de los componentes dentro de la celda que ocupan. Veamos una figura que nos aclare las cosas:



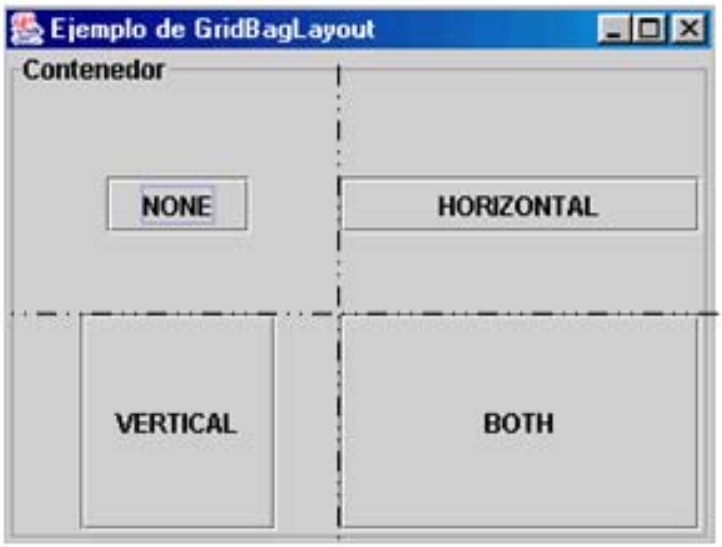
En la imágen anterior las líneas punteadas marcan el espacio que ocupa cada una de las celdas del contenedor. En este ejemplo he colocado cada uno de los componentes con un *anchor* diferente para que podáis apreciar fácilmente el efecto que tiene este atributo.

fill

El atributo *fill* especifica el espacio que ocupará el componente dentro de la celda. Los valores que puede tomar también son variables estáticas de la clase *GridBagConstraints*:

- *NONE*: El componente ocupará exactamente el tamaño que tenga como preferido
- *HORIZONTAL:* El componente ocupará todo el espacio horizontal de la celda mientras que su altura será la que tenga como preferida.
- *VERTICAL*: El componente ocupará todo el espacio vertical de la celda mientras que su longitud será la que tenga como preferida.
- *BOTH*: El componente ocupará la totalidad de la celda

Veámoslo en una imágen:

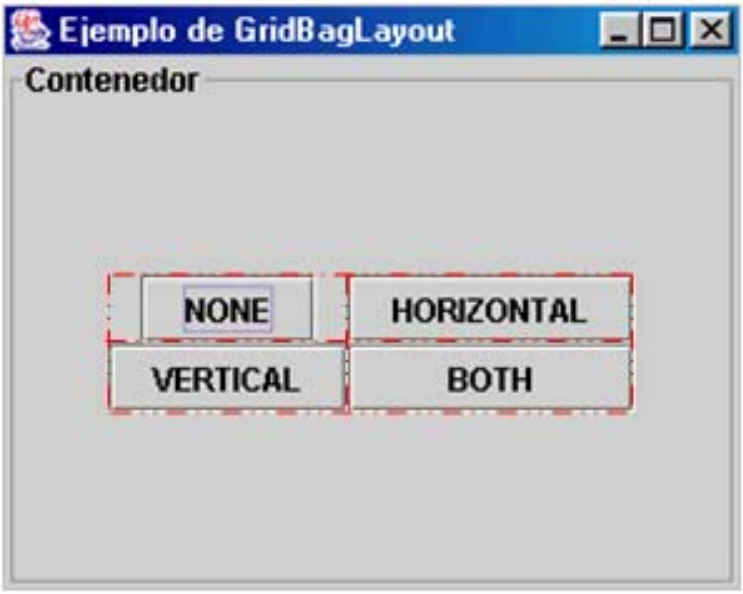


Como antes, las líneas marcan los bordes de las celdas. Como se puede apreciar en la figura según el valor del atributo *anchor* los componentes abarcarán más o menos espacio.

weightx y weighty

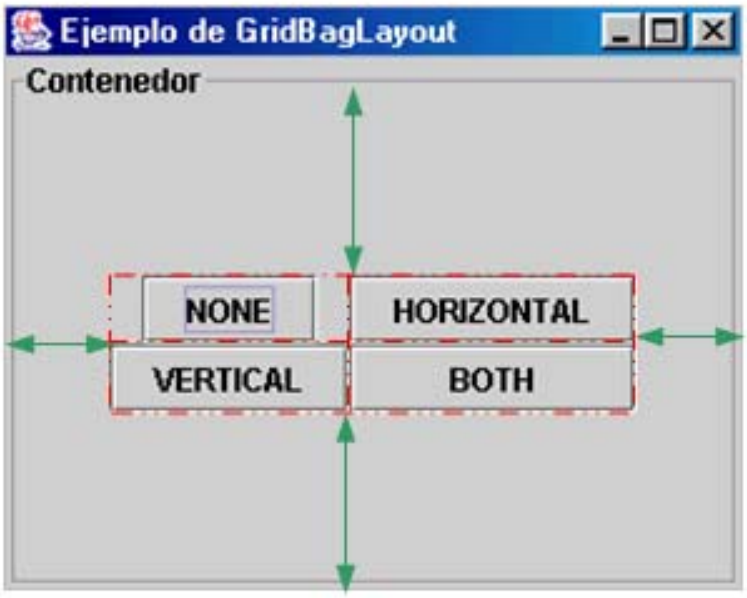
Estos dos atributos son la clave de los dolores de cabeza que *GridBagLayout* le da a tanta gente. Como habéis podido observar hasta ahora no ha habido nada complicado. Todos los atributos que hemos visto son bastante sencillos de comprender y prácticamente ya estaríamos en disposición de crear un interfaz complejo con *GridBagLayout*. Sin embargo existe un pequeño problema que veremos a continuación.

A medida que vamos añadiendo componentes a un contenedor el *layout manager* va determinando en función del tamaño de los componentes el espacio que ocupan las celdas. Hay que tener mucho cuidado porque al contrario de lo que pueda parecer si no indicamos nada las celdas no ocuparán la totalidad del contenedor. Fijaros en la figura anterior, en este caso las celdas si que ocupan todo el contenedor, pero eso es debido a que he utilizado los atributos *weightx* y *weighty*. Mirar lo que sucede si hago el mismo ejemplo anterior sin estos atributos:



He puesto las líneas que marcan el espacio ocupado por las celdas en rojo para que se vea mejor. Como veis las celdas tienen de largo la longitud máxima de los componentes y de alto la altura máxima de los componentes. Esto suele desconcertar a la gente ya que por mucho que se utilicen correctamente los otros atributos este comportamiento por defecto acaba por hacer que nuestros interfaces no salgan como planeabamos. ¿Cómo hacemos entonces para que las celdas ocupen la totalidad del contenedor?, la respuesta como cabía esperar son los atributos *weightx* y *weighty*.

Los atributos *weightx* y *weighty* especifican el porcentaje de espacio libre que ocupará una celda determinada. En el ejemplo anterior una vez añadidos los componentes queda una determinada cantidad de espacio libre tanto horizontal como vertical. Veamoslo:



Este espacio libre (flechas verdes) se dividirá entre todas las celdas que especifiquen valores dentro de los atributos *weightx* y *weighty*. La forma de especificar el espacio que quiere ocupar cada componente es mediante un número entre 0.0 y 1.0. Este número representa al porcentaje de espacio libre que ocupará cada celda.

Un ejemplo:

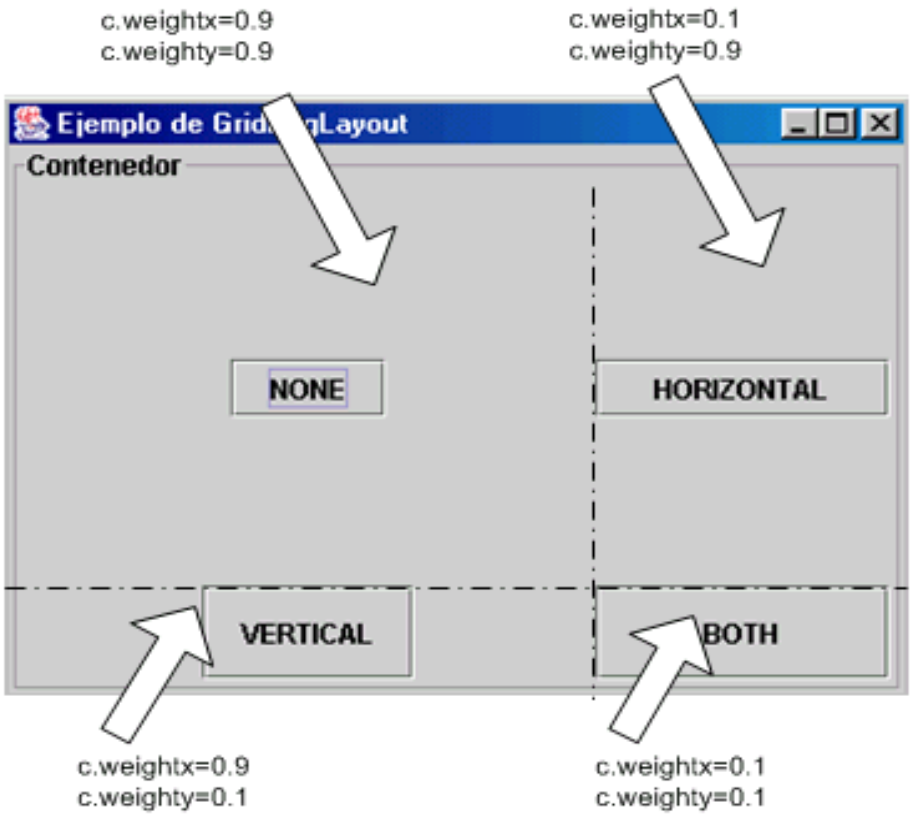
- Espacio libre 250 puntos en horizontal y 150 en vertical
- Componente 1: c.weightx=1.0, c.weighty=1.0
- Componente 2: c.weightx=0.4, c.weighty=1.0

Veamos como se asigna el espacio horizontal. Como ambos componentes han pedido espacio libre, éste se divide entre ambos, por lo tanto a cada uno le tocan 125 puntos. Sin embargo como el componente 2 tan sólo quiere el 40% del espacio libre se le asignan 50 puntos y el resto de los puntos pasan al otro componente que recibirá sus 125 puntos más los 75 puntos que sobraron del segundo componente, en total 200 puntos.

El espacio vertical es más sencillo. Como vimos antes se divide el espacio libre entre los componentes que lo han pedido. En este caso como los dos componentes han pedido la totalidad de su parte a ambos les corresponden 75 puntos.

Obviamente cuando estemos diseñando el interfaz no estaremos pensando en si este componente va a tener unos puntos y otro otros, sin embargo los atributos *weightx* y *weighty* son de una ayuda inestimable para hacer que determinadas partes de nuestra interfaz sean más grandes que las otras.

Veamos como queda el ejemplo anterior pero utilizando los atributos *weightx* y *weighty*:



Como se puede apreciar en la figura anterior cuanto mayor sea el porcentaje más espacio libre ocupará nuestra celda. Lo más importante es comprender que se los porcentajes se refieren al espacio libre y que el espacio libre se determina después de insertar los componentes. Por lo tanto, que no os extrañe que poniendo 0.1 como valor de *weightx* las celdas de la derecha ocupen tanto espacio. En este caso, las celdas de la derecha ocuparán el tamaño preferido del botón más grande (“HORIZONTAL”) más su porcentaje del espacio libre horizontal.

insets

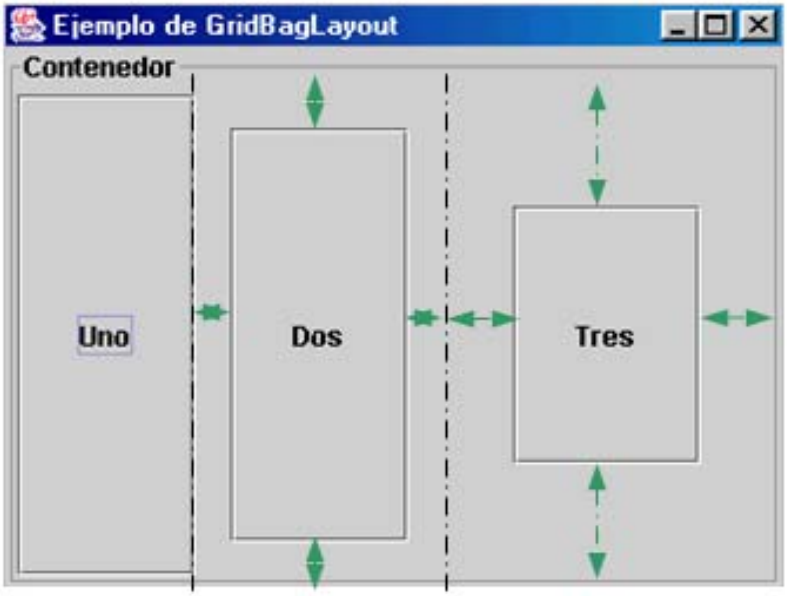
Con todo lo que hemos visto hasta ahora tenemos más que de sobra para crear una interfaz compleja. Vamos a ver un último atributo que nos servirá de gran ayuda, *insets*.

Si os fijáis en la figura anterior, cuando se insertan componentes que ocupan la totalidad de la celda ya sea en horizontal, en vertical o en ambas direcciones, el componente se pega literalmente al borde de la celda. Muy comúnmente desearemos que los componentes no estén tan pegados, es decir, que haya un márgen entre el borde de la celda y los componentes. Esto lo conseguimos con el atributo *insets*.

El atributo *insets* es un objeto de la clase *java.awt.Insets* cuyo constructor es:

Insets(int top, int left, int bottom, int right)

Como intuiréis los parámetros del constructor especifican el espacio que se dejará de márgen. Veamos un ejemplo:



Como podéis ver según el valor de *insets* con que insertemos el componente se pondrá uno u otro márgen.

Diseño de interfaces complejos

Ya hemos visto un montón de atributos de la clase *GridBagConstraints*. Vamos a ver ahora como utilizar todo lo aprendido para realizar un interfaz complejo.

La mejor forma para diseñar un interfaz es la más antigua de todas, lápiz y papel. Una vez que tengamos claro los elementos que forman nuestro interfaz de usuario en base a los casos de uso de nuestra aplicación llegará el momento de crear un boceto de dicho interfaz. A menudo nos encontraremos con que lo que en un principio nos gustaba una vez terminado el boceto ya no nos gusta tanto por lo que además de lápiz y papel, vale la pena tener a mano una goma de borrar.

Antes de comenzar con los ejemplos de *GridBagLayout* me gustaría resaltar algunos puntos:

- Crear todo el interfaz de usuario con *GridBagLayout* es un error. Normalmente en nuestro interfaz de usuario tendremos una serie de contenedores, por ejemplo paneles, que de por si forman unidades lógicas (un panel que muestra información de cliente, un panel de búsqueda, ...). Estos paneles nos ayudan a hacer nuestro código más sencillo y estructurado, prescindir de ellos y crear todo el interfaz con *GridBagLayout* lo único que haría sería hacer más complejo nuestro código.

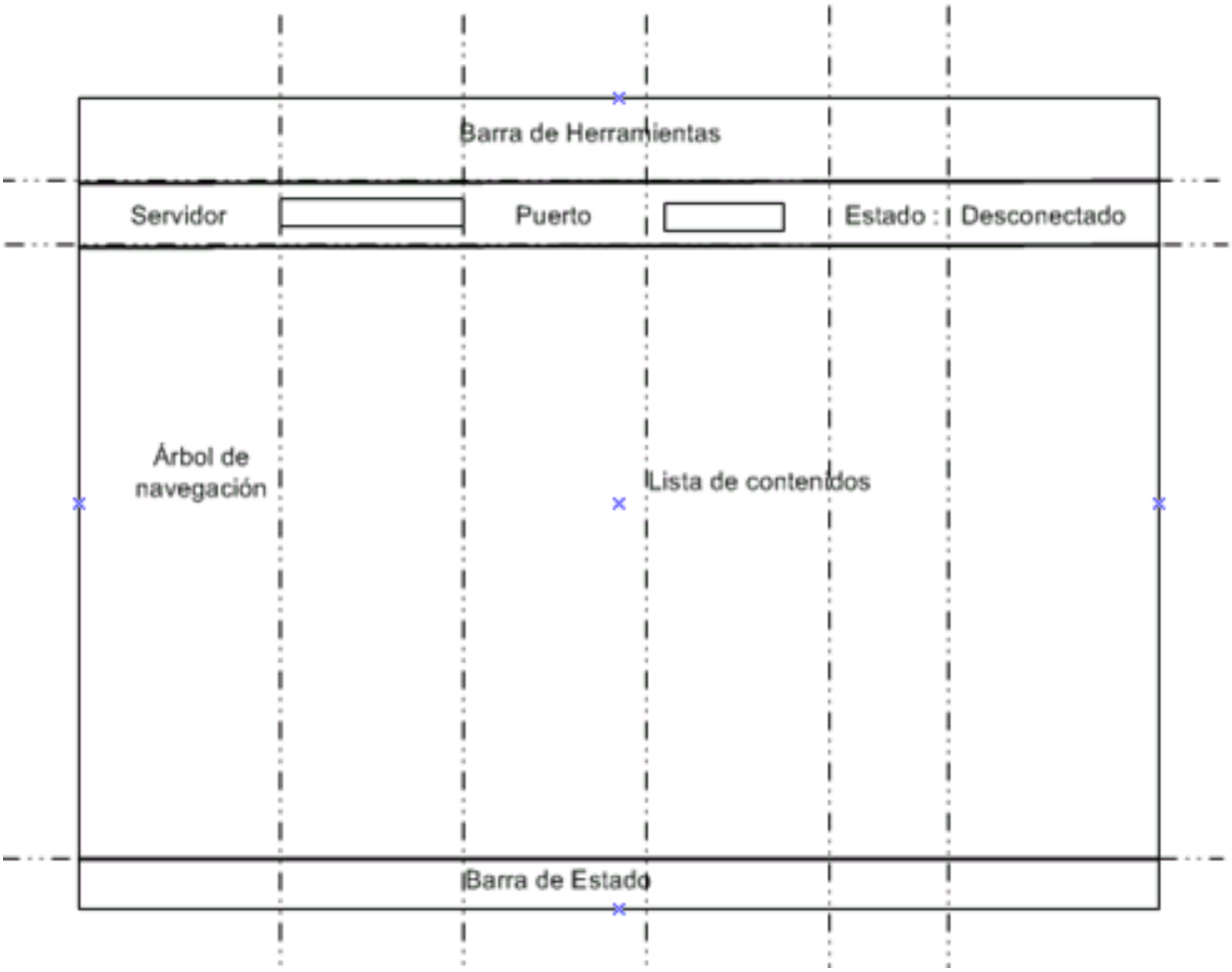
- Siempre que sea más sencillo utilizar los otros *layouts* hacedlo. Por supuesto teniendo cuidado de no entrar en los casos donde *GridBagLayout* era ventajoso, es decir, si por ejemplo estamos cargando demasiado la interfaz de usuario porque anidamos muchos paneles quizás nos tengamos que replantear nuestro diseño.

- En muchas ocasiones para crear ese primer nivel de contenedores no necesitaremos ningún *layout manager*. Por ejemplo, es muy común utilizar *JSplitPanes* para crear áreas de nuestra interfaz que el usuario pueda redimensionar a su gusto.

- Típicamente utilizaremos *GridBagLayout* para modelar los contenedores principales de nuestra aplicación de los que hemos hablado antes, siempre y cuando sus elementos no se puedan crear de manera mucho más sencilla con uno de los otros interfaces de usuario.

Una vez aclarados estos aspectos vamos a ver como crear un par de interfaces de usuario bastante comunes. La primera es un típico diálogo de entrada a un sistema, algo tan sencillo pero que se convierte en un quebradero de cabeza si no estás acostumbrado a los *layout managers*. El segundo ejemplo es algo más complejo y representa al interfaz de un cliente de ftp. Veamos los bocetos de ambas interfaces:

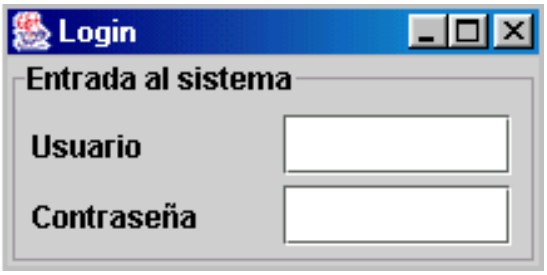

Usuario
Contraseña


Barra de Herramientas
Servidor
Puerto
Estado : Desconectado
Árbol de navegación
Lista de contenidos
Barra de Estado

Vamos a empezar con el más sencillo, la ventana de entrada al sistema. Veamos como sería el código para crear este interfaz:

```
import javax.swing.*;
import java.awt.*;

public class A {

    public static void main(String[] args) {

        JFrame f = new JFrame();
        Container container = f.getContentPane();
        container.setLayout(new GridBagLayout());
        ((JPanel)container).setBorder(BorderFactory.createTitledBorder(
                                        "Entrada al sistema"));
        GridBagConstraints c = new GridBagConstraints();

        c.weightx=0.4; c.weighty=1.0;
        c.gridwidth=GridBagConstraints.RELATIVE;
        c.gridheight=GridBagConstraints.RELATIVE;
        c.fill=GridBagConstraints.BOTH;
        c.anchor = GridBagConstraints.WEST;
        c.insets = new Insets(2,5,2,0);
        container.add(new JLabel("Usuario"),c);
        c.gridwidth=GridBagConstraints.REMAINDER;
        c.gridheight=GridBagConstraints.RELATIVE;
        c.weightx=1.0;
        c.insets = new Insets(2,0,2,5);
        container.add(new JTextField(),c);
        c.gridwidth=GridBagConstraints.RELATIVE;
        c.gridheight=GridBagConstraints.REMAINDER;
        c.weightx=0.4;
        c.insets = new Insets(2,5,2,0);
        container.add(new JLabel("Contraseña"),c);
        c.gridwidth=GridBagConstraints.REMAINDER;
        c.gridheight=GridBagConstraints.REMAINDER;
        c.weightx=1.0;
        c.insets = new Insets(2,0,2,5);
        container.add(new JTextField(),c);

        f.setSize(220,110);
        f.setTitle("Login");
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Como se puede ver cada vez que añadimos un componente al contenedor hemos de pasar una variable de tipo *GridBagConstraints* al método *add*. Tened mucho cuidado de no olvidaros pasar la variable de *constraints* porque en otro caso el interfaz no saldrá como esperábais. Esta variable puede ser reutilizada, tal como se ve en el código, para no tener que estar creandolas continuamente. Si compilais y ejecutáis el ejemplo os deberíais encontrar con la ventana siguiente:



Un último punto que quería resaltar del código fuente anterior es que como podéis observar no he utilizado para nada los atributos *gridx* y *gridy*. Como ya os he comentado, estos atributos no son estrictamente necesarios al igual que tampoco lo son *gridwidth* o *gridheight*. A menudo con utilizar una de las dos alternativas será suficiente pero habrá ocasiones en las que tendremos que utilizar ambos atributos.

Veamos como haríamos ahora para crear el interfaz de usuario del boceto de cliente de ftp que vimos anteriormente.

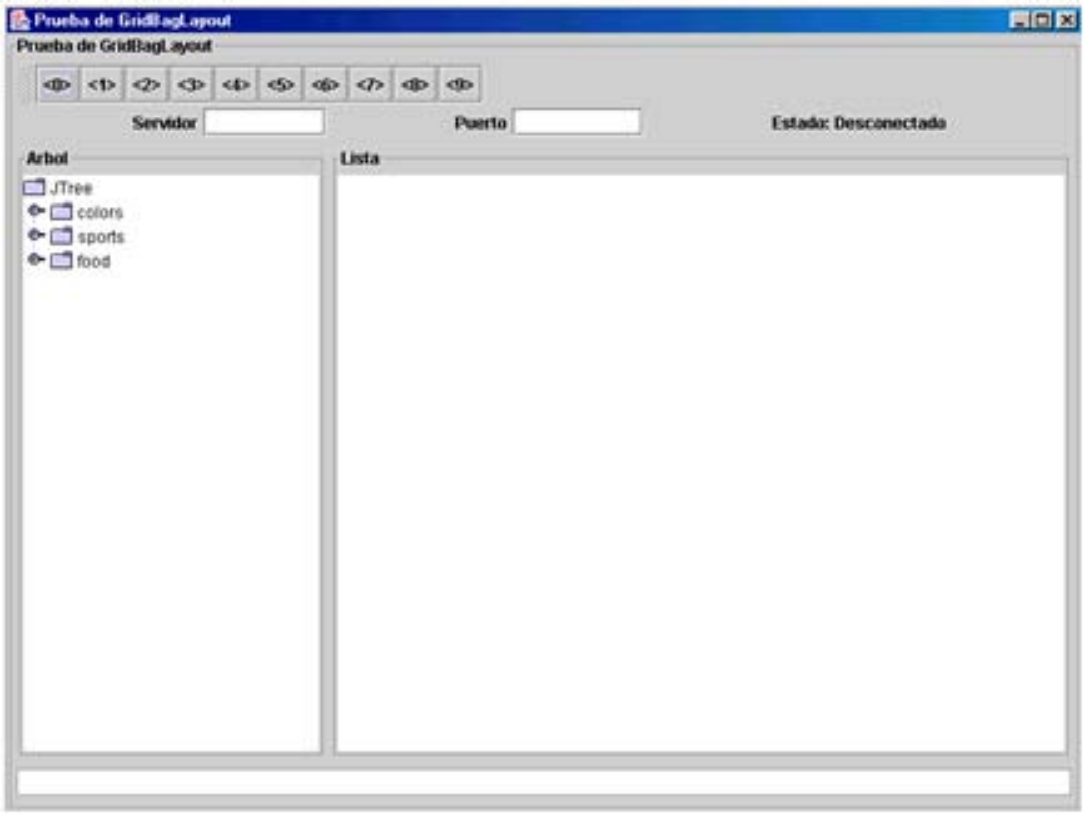
```
import javax.swing.*;
import java.awt.*;

public class A {

  public static void main(String[] args) {

    JFrame f = new JFrame();
    Container container = f.getContentPane();
    container.setLayout(new GridBagLayout());
    ((JPanel)container).setBorder(
            BorderFactory.createTitledBorder("Prueba de GridBagLayout"));
    GridBagConstraints c = new GridBagConstraints();

    JToolBar toolbar = new JToolBar();
    for (int i = 0; i<10; i++)
       toolbar.add(new JButton("<" + i + ">"));
```

```
JScrollPane panelArbol = new JScrollPane(new JTree());
panelArbol.setBorder(BorderFactory.createTitledBorder("Arbol"));

JScrollPane panelLista = new JScrollPane(new JList());
panelLista.setBorder(BorderFactory.createTitledBorder("Lista"));

JTextField statusBar = new JTextField();
statusBar.setEnabled(false);

c.fill=GridBagConstraints.HORIZONTAL;
c.weightx = 1.0; c.weighty = 0.0;
c.gridx = 0; c.gridy = 0;
c.gridwidth = GridBagConstraints.REMAINDER; c.gridheight = 1;
container.add(new JToolBar(),c);

c.fill = GridBagConstraints.NONE;
c.insets = new Insets(2,2,6,2);

c.gridx = 0; c.gridy = 1;
c.gridwidth = 1; c.gridheight = 1;
c.anchor = GridBagConstraints.EAST;
container.add(new JLabel("Servidor"),c);

c.gridx = 1;
c.fill = GridBagConstraints.HORIZONTAL;
c.anchor = GridBagConstraints.WEST;
container.add(new JTextField(),c);

c.gridx = 2;
c.fill = GridBagConstraints.NONE;
c.anchor = GridBagConstraints.EAST;
container.add(new JLabel("Puerto"),c);

c.gridx = 3;
c.anchor = GridBagConstraints.WEST;
c.fill = GridBagConstraints.HORIZONTAL;
container.add(new JTextField(),c);

c.gridx = 4;
c.fill = GridBagConstraints.NONE;
c.gridwidth = GridBagConstraints.RELATIVE;
c.anchor = GridBagConstraints.EAST;
container.add(new JLabel("Estado:"),c);

c.gridx = 5;
```

```
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.anchor = GridBagConstraints.WEST;
        container.add(new JLabel("Desconectado"),c);


        c.gridx = 0;c.gridy = 2;
        c.weighty = 1.0;
        c.fill = GridBagConstraints.BOTH;
        c.gridwidth = 2; c.gridheight = GridBagConstraints.RELATIVE;
        container.add(panelArbol,c);

        c.gridx = 2;
        c.gridwidth = GridBagConstraints.REMAINDER;
        container.add(panelLista,c);

        c.weighty = 0.0;
        c.gridx = 0; c.gridy = 3;
        c.gridheight = GridBagConstraints.REMAINDER;
        container.add(statusBar,c);


        f.setSize(800,600);
        f.setTitle("Prueba de GridBagLayout");
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      }
    }
```

Como veis la creación de este interfaz ya no ha sido tan sencilla. La cantidad de código necesario ha sido ya considerable y peor aún, a simple vista no podemos tener una idea clara de como es la interfaz de esta aplicación. Por otra parte el interfaz de la misma es muy ligero ya que no tenemos paneles anidados y tardará muy poco en cargarse.

Tened en cuenta que esto es tan sólo un ejemplo de como crear un interfaz complejo con *GridBagLayout*. En la práctica tendremos que decidir por nosotros mismos cuando estamos sobrecargando nuestro interfaz. Probablemente si nos vemos obligados a crear tres o más paneles anidados deberíamos considerar el deshacer dicho anidamiento. Sin embargo el interfaz anterior sería verdaderamente sencillo el crearlo con un panel con *BorderLayout* y un *JSplitPane* para el árbol y la lista.



Con esto termina este apartado. Hemos aprendido a dominar este *layout manager* y a realizar interfaces de usuario complejas y ligeras. Además hemos superado una de las limitaciones del *GridBagLayout*, ¡¡ hemos aprendido a utilizarlo !!. Sin embargo queda una limitación más, el código resulta ilegible debido a todas esas líneas de *constraints* que hace díficil el seguir la creación del mismo. En el siguiente apartado veremos como utilizando clases auxiliares podremos crear interfaces complejas con *GridBagLayout* pero con la simplicidad del resto de *layout managers.*

Mejorando *GridBagLayout*

Básicamente trataremos de resolver dos aspectos derivados del problema que nos quedó por resolver en el apartado anterior:

- Gran cantidad de líneas de código
- No nos da una idea de como es el interfaz gráfico

Para intentar solucionar[1] la limitación de la ilegibilidad de los interfaces de usuario creados con *GridBagLayout* lo que haremos será encapsular el objeto *GridBagConstraints* dentro de una clase que contendrá las restricciones para un grupo de componentes. La clase auxiliar que crearemos se llama *ConstraintGroup* y la podéis ver a continuación:

```
package com.canalejo.protocolos.ui;

import java.awt.GridBagConstraints;
import java.awt.Insets;

public class ConstraintGroup extends GridBagConstraints {

   private static final int NUM_CONSTRAINTS = 4;
   private int number = 0;
   private int[][] constraints;
   private double[][] weights;

   private ConstraintGroup() {}

   public ConstraintGroup(int[][] constraints, double[][] weights) {

      super();
      this.constraints = constraints;
      this.weights = weights;
   }

   public void setConstraints(int element) {

      int[] location = constraints[element * NUM_CONSTRAINTS];
      int[] area = constraints[element * NUM_CONSTRAINTS + 1];
      int[] size = constraints[element * NUM_CONSTRAINTS + 2];
      int[] insets = constraints[element * NUM_CONSTRAINTS + 3];
      double[] weights = this.weights[element];

      this.weightx = weights[0];
      this.weighty = weights[1];
      this.gridx = location[0];
      this.gridy = location[1];
      this.gridwidth = area[0];
      this.gridheight = area[1];
      this.fill = size[0];
      this.anchor = size[1];
      if (insets != null) {
```

1.Esta solución es una implementación mejorada de la solución que se esboza para este problema en el libro *Complete Java 2 Certification Study Guide*[4]

```
            this.insets = new Insets(insets[0],insets[1],insets[2],insets[3]);
        }
    }

    public GridBagConstraints getConstraints(int element) {

        setConstraints(element);
        return this;
    }
}
```

Como podéis ver en el código fuente, esta clase es una ampliación de *GridBagConstraints* cuyo objetivo es poder almacenar todas las restricciones de un interfaz en un único lugar. El proceso de creación de un interfaz utilizando esta clase es muy sencillo:

1. Crear un objeto *ConstraintGroup* inicializado con todas las restricciones de nuestro interfaz gráfico
2. Utilizar el método *getConstraints(int element)* para obtener las restricciones para cada uno de los elementos del interfaz en el momento de añadirlos al contenedor.

Veamos un ejemplo de como se utilizaría esta clase:

```
ConstraintGroup cg =  new ConstraintGroup(....);

container.add(new JLabel("Usuario"),cg.getConstraints(0));
container.add(new JTextField(),cg.getConstraints(1));
container.add(new JLabel("Contraseña"), cg.getConstraints(2));
container.add(new JTextField(),cg.getConstraints(3));
.....
```

Como se puede apreciar, ahora el código es considerablemente más limpio que todo el amasijo de líneas que tendríamos que haber escrito utilizando *GridBagConstraints*.

Ahora bien, muchos os estaréis preguntando ¿dónde se especifican las restricciones?. Bien, para especificar las restricciones tenemos muchas opciones: podemos crear los arrays de restricciones como atributos de nuestra clase, podemos utilizar una clase interna, podemos utilizar clases separadas o incluso guardarlas en un fichero *XML* para aislar más la creación del interfaz. En los ejemplos que pondré a continuación las crearé en una clase interna. Veamos un ejemplo:

```
private final class GUIConstraints {

    private final int NONE = GridBagConstraints.NONE;
    private final int BOTH = GridBagConstraints.BOTH;
    private final int SOUT = GridBagConstraints.SOUTH;
```

```
    private final int CENT = GridBagConstraints.CENTER;
    private final int WEST = GridBagConstraints.WEST;
    private final int SOEA = GridBagConstraints.SOUTHEAST;
    private final int HORI = GridBagConstraints.HORIZONTAL;
    private final int RELA = GridBagConstraints.RELATIVE;
    private final int REMA = GridBagConstraints.REMAINDER;

    final int[][] gMain = new int[][] {
        {0, 0}, {RELA, 5   }, {BOTH, CENT}, {4, 2, 4, 2},
        {1, 0}, {REMA, 1   }, {HORI, CENT}, {4, 2, 2, 4},
        {0, 6}, {RELA, REMA}, {HORI, SOUT}, {2, 4, 4, 2},
        {1, 1}, {REMA, 1   }, {HORI, CENT}, {2, 2, 2, 4},
        {1, 2}, {REMA, 1   }, {HORI, CENT}, {2, 2, 2, 4},
        {1, 3}, {REMA, 1   }, {NONE, CENT}, {2, 2, 2, 4},
        {1, 4}, {REMA, 1   }, {NONE, CENT}, {2, 2, 2, 4},
        {1, 5}, {REMA, 1   }, {NONE, CENT}, {2, 2, 2, 4},
        {1, 6}, {REMA, RELA}, {HORI, CENT}, {2, 2, 2, 4},
        {1, 7}, {REMA, REMA}, {HORI, CENT}, {2, 2, 4, 4}
    };
    final double[][] wMain = new double[][] {
        {1.0, 0.8},
        {0.0, 0.1},
        {1.0, 0.2},
        {0.0, 0.1},
        {0.0, 0.1},
        {0.0, 0.1},
        {0.0, 0.1},
        {0.0, 0.1},
        {0.0, 0.1},
        {0.0, 0.1}
    };
}
```

Como veis la clase consta de dos arrays, uno de enteros y otro de números decimales que especifican los valores de las restricciones para cada uno de los componentes del interfaz gráfico. Cada línea representa a un componente del interfaz. Habrá tantos componentes como líneas de restricción tengamos. El significado de las líneas es el siguiente:

- **línea de enteros: {#, #}, {#, #}, {#, #}, {#,#,#,#}**

{gridx, gridy}, {gridwidth, gridheight}, {fill, anchor}, {top, bottom,left right}(insets)

- **línea de decimales: {#,#}**

{weightx, weighty}

El agrupar todas las restricciones en una serie de arrays tal como se ve en el código anterior resuelve el segundo problema que teníamos. Ahora con un simple vistazo podemos saber como se sitúan exactamente los componentes del interfaz, su posición, su ancho, su alineación, etc... Esto lejos de ser tan sencillo como la creación de interfaces con una herramienta *RAD* al menos nos ofrece una visión mucho mejor que la que teníamos antes. Además, el tener las restricciones agrupadas en un único lugar hace que el introducir modificaciones o cambios en el interfaz sea mucho más sencillo y nos evita el tener que andar navegando entre multitud de líneas de código.

Por último, vamos a ver como crear los dos interfaces que habíamos visto anteriormente. Empecemos por el diálogo de entrada al sistema:

```
import javax.swing.*;
import java.awt.*;

public class B {

    private static final class GUIConstraints {

        private final int NONE = GridBagConstraints.NONE;
        private final int BOTH = GridBagConstraints.BOTH;
        private final int SOUT = GridBagConstraints.SOUTH;
        private final int CENT = GridBagConstraints.CENTER;
        private final int WEST = GridBagConstraints.WEST;
        private final int SOEA = GridBagConstraints.SOUTHEAST;
        private final int HORI = GridBagConstraints.HORIZONTAL;
        private final int RELA = GridBagConstraints.RELATIVE;
        private final int REMA = GridBagConstraints.REMAINDER;

        final int[][] gMain = new int[][] {
              {0, 0}, {RELA, RELA}, {BOTH, WEST},  {2, 5, 2, 0},
              {1, 0}, {REMA, RELA}, {BOTH, WEST}, {2, 0, 2, 5},
              {0, 1}, {RELA, REMA}, {BOTH, WEST}, {2, 5, 2, 0},
              {1, 1}, {REMA, REMA}, {BOTH, WEST}, {2, 0, 2, 5}
        };
        final double[][] wMain = new double[][] {
              {0.4, 1.0},
              {1.0, 1.0},
              {0.4, 1.0},
              {1.0, 1.0}
        };
    }

    public static void main(String[] args) {
```

```
                JFrame f = new JFrame();
                Container container = f.getContentPane();
                container.setLayout(new GridBagLayout());
                ((JPanel)container).setBorder(BorderFactory.createTitledBorder(
                                                "Entrada al sistema"));


                GUIConstraints cons = new GUIConstraints();
                ConstraintGroup cg =
                            new ConstraintGroup(cons.gMain,cons.wMain);

                container.add(new JLabel("Usuario"),cg.getConstraints(0));
                container.add(new JTextField(),cg.getConstraints(1));
                container.add(new JLabel("Contraseña"),cg.getConstraints(2));
                container.add(new JTextField(),cg.getConstraints(3));

                f.setSize(220,110);
                f.setTitle("Login");
                f.setVisible(true);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            }
        }
```

Hemos sustituido las 22 líneas de formato del interfaz que teníamos originalmente por 4 líneas para añadir los elementos más la definición del formato. Aunque la definición del formato ocupa bastantes líneas, muchas de éstas son de definición de constantes[1] y como ya hemos reseñado el objetivo de esta definición es una presentación visual del contenido, cosa que no teníamos en el código original.

Como ya he sugerido anteriormente, sería posible trasladar toda esta definición de constantes a un fichero *XML* en el que tuviésemos definidos el número de componentes junto con sus restricciones lo que minimizaría todavía más el código.

Veamos el código para el segundo interfaz gráfico, el del cliente *FTP:*

```
import javax.swing.*;
import java.awt.*;

public class A {

    private static final class GUIConstraints {

        private final int NONE = GridBagConstraints.NONE;
        private final int BOTH = GridBagConstraints.BOTH;
```

1.No se han incluido todas las constantes posibles por brevedad

```
    private final int SOUT = GridBagConstraints.SOUTH;
    private final int CENT = GridBagConstraints.CENTER;
    private final int WEST = GridBagConstraints.WEST;
    private final int SOEA = GridBagConstraints.SOUTHEAST;
    private final int HORI = GridBagConstraints.HORIZONTAL;
    private final int RELA = GridBagConstraints.RELATIVE;
    private final int REMA = GridBagConstraints.REMAINDER;

    final int[][] gMain = new int[][] {
       {0, 0}, {REMA, 1   }, {HORI, CENT}, {0, 0, 0, 0},
       {0, 1}, {1   , 1   }, {NONE, EAST}, {2, 2, 6, 2},
       {1, 1}, {1   , 1   }, {HORI, WEST}, {2, 2, 6, 2},
       {2, 1}, {1   , 1   }, {NONE, EAST}, {2, 2, 6, 2},
       {3, 1}, {1   , 1   }, {HORI, WEST}, {2, 2, 6, 2},
       {4, 1}, {RELA, 1   }, {NONE, EAST}, {2, 2, 6, 2},
       {5, 1}, {REMA, 1   }, {NONE, WEST}, {2, 2, 6, 2},
       {0, 2}, {2   , RELA}, {BOTH, WEST}, {2, 2, 6, 2},
       {2, 2}, {REMA, RELA}, {BOTH, WEST}, {2, 2, 6, 2},
       {0, 3}, {REMA, REMA}, {BOTH, CENT}, {2, 2, 6, 2},
    };
    final double[][] wMain = new double[][] {
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 0.0},
       {1.0, 1.0},
       {1.0, 1.0},
       {1.0, 0.0},
    };
  }

public static void main(String[] args) {

  JFrame f = new JFrame();
  Container container = f.getContentPane();
  container.setLayout(new GridBagLayout());
  ((JPanel)container).setBorder(BorderFactory.createTitledBorder(
                                        "Prueba de GridBagLayout"));

  JToolBar toolbar = new JToolBar();
  for (int i = 0; i<10; i++)
     toolbar.add(new JButton("<" + i + ">"));
```

```
        JScrollPane panelArbol = new JScrollPane(new JTree());
        panelArbol.setBorder(BorderFactory.createTitledBorder("Arbol"));

        JScrollPane panelLista = new JScrollPane(new JList());
        panelLista.setBorder(BorderFactory.createTitledBorder("Lista"));

        JTextField statusBar = new JTextField();
        statusBar.setEnabled(false);

        GUIConstraints cons = new GUIConstraints();
        ConstraintGroup cg =  new ConstraintGroup(cons.gMain,cons.wMain);

        container.add(toolbar,cg.getConstraints(0));
        container.add(new JLabel("Servidor"),cg.getConstraints(1));
        container.add(new JTextField(),cg.getConstraints(2));
        container.add(new JLabel("Puerto"),cg.getConstraints(3));
        container.add(new JTextField(),cg.getConstraints(4));
        container.add(new JLabel("Estado:"),cg.getConstraints(5));
        container.add(new JLabel("Desconectado"),cg.getConstraints(6));
        container.add(panelArbol,cg.getConstraints(7));
        container.add(panelLista,cg.getConstraints(8));
        container.add(statusBar,cg.getConstraints(9));


        f.setSize(800,600);
        f.setTitle("Prueba de GridBagLayout");
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      }
    }
```

En este interfaz más complejo si que podemos apreciar mejor las ventajas de este método. El número de líneas utilizadas para añadir los elementos a nuestro interfaz ha sido de tan sólo 10, y no tenemos ese amasijo de restricciones mezcladas dentro de nuestro código. Como antes, aunque las líneas necesarias para la creación de la clase interna son bastantes, nos da una idea visual de como están situados los componentes sin la necesidad de atarnos a un editor gráfico.

Con esto termina el apartado dedicado a *GridBagLayout*. Espero que este extenso material os haya servido para aprender más sobre el *layout manager* más complejo y que a partir de ahora los que no os veiais capaces de utilizarlo hayáis aprendido lo suficiente como para usarlo en vuestros proyectos y aprovecharse de sus ventajas.

OverlayLayout

Hasta ahora todos los *layout managers* que hemos visto se encontraban dentro del paquete *java.awt* aunque eso no evita que los podamos utilizar en aplicaciones que utilicen *swing*. De aquí al final del artículo veremos una serie de *layout managers* que se encuentran dentro del paquete *javax.swing* y que son también bastante interesantes.

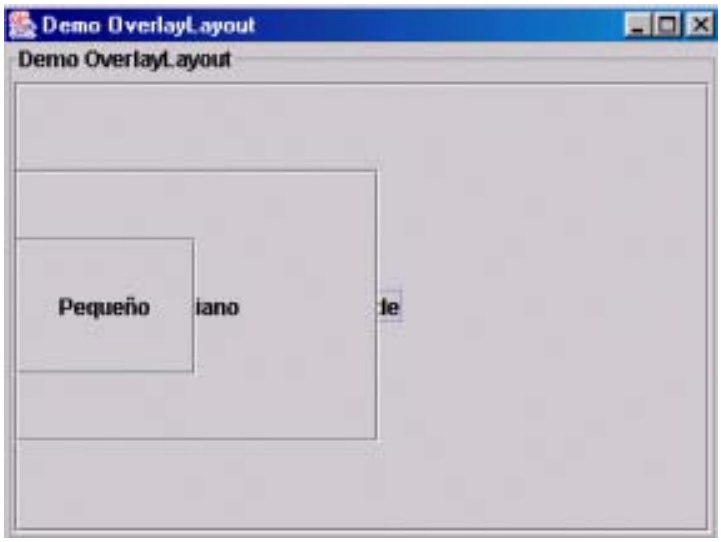
OverlayLayout organiza los componentes en capas al estilo de *CardLayout* pero con varias diferencias:

- El tamaño de los componentes no es fijo sino que puede ser variable.
- El tamaño del contenedor será el tamaño del mayor de los componentes.
- Debido a que el tamaño de los componentes es diferente se pueden ver varios a la vez.
- A diferencia de *CardLayout* no existen métodos para navegar por los componentes.

Como puede resultar díficil entender que es lo que hace este *layout manager*, nada mejor que un ejemplo sencillo para aclarar las cosas:

```
import java.awt.*;
import javax.swing.*;

public class OverlayLayoutDemo {

    public static void main(String[] args) {

        JFrame f = new JFrame();
        Container container = f.getContentPane();
        container.setLayout(new OverlayLayout(container));
        ((JPanel)container).setBorder(
            BorderFactory.createTitledBorder("Demo OverlayLayout"));

        JButton jbGrande = new JButton("Grande");
        JButton jbMediano = new JButton("Mediano");
        JButton jbPequeño = new JButton("Pequeño");

        jbGrande.setMaximumSize(new Dimension(400,300));
        jbMediano.setMaximumSize(new Dimension(200,150));
        jbPequeño.setMaximumSize(new Dimension(100,75));

        container.add(jbPequeño);
        container.add(jbMediano);
        container.add(jbGrande);
```

```
                f.setSize(400,300);
                f.setTitle("Demo OverlayLayout");
                f.setVisible(true);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            }
        }
```



Los componentes se muestran en el orden en el que se van insertando en el contenedor y además se ajustan al tamaño máximo que prefieran. Si como en nuestro ejemplo los componentes traseros ocupan más espacio que los de delante entonces se verán. Hay que tener cuidado al utilizar este *layout manager* de que los componentes que coloquemos al principio no sean más grandes que los de detrás porque en ese caso no veremos nada.

BoxLayout

Este *layout manager* es uno de los más sencillos y de los más útiles. Aquí los componentes son agrupados horizontal o verticalmente dentro del contenedor que los contiene. Los componentes no se solapan de ningún modo. La anidación de paneles utilizando este *layout manager* nos puede permitir crear interfaces muy complejos como nos permite *GridBagLayout* pero a costa de la creación de objetos pesados como son paneles.

El constructor de *BoxLayout* es muy simple:

```
public BoxLayout(Container objetivo, int eje);
```

Donde el parámetro entero *eje* puede tomar los valores:

- **X_AXIS**, los componentes se organizan de izquierda a derecha y en horizontal.
- **Y_AXIS**, los componentes se organizan de arriba a abajo y en vertical.
- **LINE_AXIS**, los componentes se organizan como si estuviesen en una línea. Para ello se tiene en cuenta la propiedad *ComponentOrientation* del contenedor. Si esta propiedad es horizontal entonces los componentes se organizarán horizontalmente y además según su valor lo harán de izquierda a derecha o de derecha a izquierda. En otro caso se organizarán verticalmente de arriba a abajo.
- **PAGE_AXIS**, los componentes se organizan como si estuvieran en una página. Para ello se tiene en cuenta la propiedad *ComponentOrientation* del contenedor. Si esta propiedad es horizontal entonces los componentes se organizarán verticalmente y en otro caso lo harán horizontalmente. Igual que antes podremos decidir si queremos que se alineen de izquierda a derecha o de derecha a izquierda en caso de que tengamos una organización horizontal mientras que en la vertical siempre lo harán de arriba a abajo.

Los componentes se organizan en el orden en el que se añaden al contenedor. En el caso de que los organicemos horizontalmente se intentará respetar la longitud preferida por el componente y en caso de organizarlos verticalmente se intentará respetar su altura.

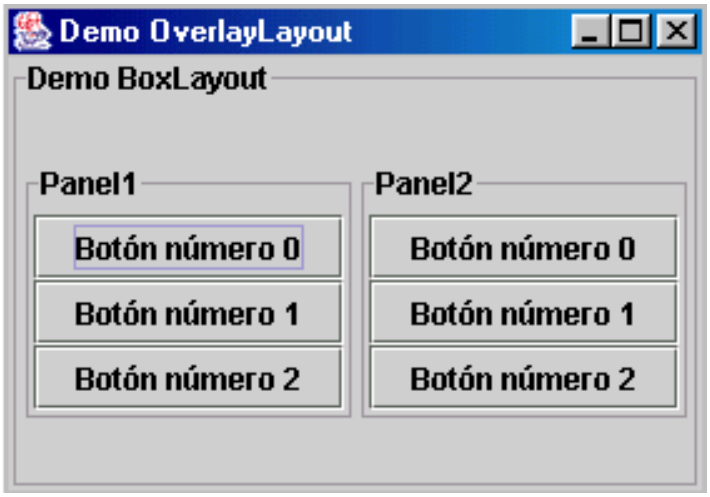
En el caso de organización horizontal, *BoxLayout* intentará que todos los componentes del contenedor tengan la misma altura, siendo esta la máxima de los elementos del contenedor. En caso de que no sea posible, *BoxLayout* intentará alinearlos a todos horizontalmente de modo que sus centros coincidan en una línea horizontal imaginaria que los atraviese.

De manera similar, si la organización es vertical, *BoxLayout* intentará que todos los componentes tengan la misma longitud, siendo esta la máxima longitud de los elementos del contenedor. Si eso falla, intentará alinearlos verticalmente de modo que sus centros coincidan en una línea vertical imaginaria que los atraviese.

A menudo, en lugar de utilizar *BoxLayout* directamente, se utiliza la clase *Box* que es un contenedor ligero que tiene como *layout manager* un *BoxLayout* y que ofrece métodos que hacen que su manejo sea muy sencillo. A continuación vamos a ver un ejemplo de como utilizaríamos *BoxLayout* y de como realizar el mismo programa con la clase *Box*.

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutDemo {
```

```
    public static void main(String[] args) {

        JFrame f = new JFrame();
        Container container = f.getContentPane();
        container.setLayout(new
                        BoxLayout(container, BoxLayout.X_AXIS));
        ((JPanel)container).setBorder(
            BorderFactory.createTitledBorder("Demo BoxLayout"));

        JPanel panel1 = new JPanel();
        panel1.setBorder(BorderFactory.createTitledBorder("Panel1"));
        JPanel panel2 = new JPanel();
        panel2.setBorder(BorderFactory.createTitledBorder("Panel2"));
        panel1.setLayout(new BoxLayout(panel1, BoxLayout.Y_AXIS));
        panel2.setLayout(new BoxLayout(panel2, BoxLayout.Y_AXIS));

        for (int i = 0; i < 3; i++) {
            panel1.add(new JButton("Botón número " + i));
            panel2.add(new JButton("Botón número " + i));
        }
        container.add(panel1);
        container.add(panel2);

        f.setSize(285,300);
        f.setTitle("Demo OverlayLayout");
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```



Como veis su uso es muy sencillo y su flexibilidad es bastante grande. Veamos ahora el código anterior pero utilizando la clase *Box*:

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutDemo {

    public static void main(String[] args) {

        JFrame f = new JFrame();
        Box bigBox = Box.createHorizontalBox();
        bigBox.setBorder(
            BorderFactory.createTitledBorder("Demo BoxLayout"));
        f.setContentPane(bigBox);

        Box box1 = Box.createVerticalBox();
        box1.setBorder(BorderFactory.createTitledBorder("Box 1"));
        Box box2 = Box.createVerticalBox();
        box2.setBorder(BorderFactory.createTitledBorder("Box 2"));

        for (int i = 0; i < 3; i++) {
            box1.add(new JButton("Botón número " + i));
            box2.add(new JButton("Botón número " + i));
        }
        bigBox.add(box1);
        bigBox.add(box2);
```

```
                f.setSize(285,200);
                f.setTitle("Demo OverlayLayout");
                f.setVisible(true);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            }
        }
```

Los dos ejemplos anteriores producen exactamente el mismo resultado y como veis quizás este sea incluso si cabe un poco más sencillo.

Además de encapsular un contenedor con un *BoxLayout*, la clase *Box* contiene una serie de métodos factoría que nos permiten obtener componentes invisibles con los que podremos ajustar al máximo el aspecto de nuestro interfaz gráfico. Los métodos útiles son los siguientes:

- **public Component createGlue()**
- **public Component createHorizontalGlue()**
- **public Component createVerticalGlue()**

Estos métodos crean lo que podríamos traducir como un pegamento entre los componentes. Este pegamento separa varios componentes y absorbe todo el espacio extra que se encuentre disponible en el contenedor. Quizás un ejemplo aclare más las cosas. Imaginaros que tenemos un contenedor con dos componentes cuyo tamaño es fijo. ¿Qué sucede si alargamos horizontalmente el contenedor?. Lo más probable es que ese espacio extra quede a la derecha de los componentes y el interfaz quede bastante feo. Si insertamos un pegamento entre estos dos componentes será él el que absorba todo el espacio extra. Veamos el ejemplo:

En la parte superior podemos ver el resultado de alargar un panel que contiene dos botones de tamaño fijo. Fijémonos en la siguiente imágen en lo que sucede al añadir un objeto de pegamento entre ambos botones:

Como se puede apreciar, al alargar el contenedor el espacio lo ha absorbido el objeto de pegamento que insertamos. Los más descuidados podéis ver a continuación como sería el proceso de añadir un componente de pegamento entre los dos botones (recordad que al menos su máximo ha de estar fijo):

```
....
contenedor.add(boton1);
contenedor.add(Box.createHorizontalGlue());
contenedor.add(boton2);
....
```

Veamos más métodos factoría útiles de la clase *Box*:

- **public Component createRigidArea(Dimension d)**, este método crea un componente invisible que siempre tomará el mismo tamaño. Es muy útil para añadir espacios invariables a nuestros interfaces.
- **public Component createHorizontalStrut(int longitud)**, crea un elemento con una longitud fija que pasamos como parámetro y sin altura. Este método es útil cuando queremos que entre dos componentes exista siempre una longitud fija. En caso de que el contenedor se amplie verticalmente, este componente tomará la altura que le corresponda justo como cualquier otro componente que no tenga fijada su altura.
- **public Component createVerticalStrut(int altura)**, crea un elemento con una altura fija que pasamos como parámetro y sin longitud. Este método es útil cuando queremos que entre dos componentes exista siempre una altura fija. En caso de que el contenedor se amplie horizontalmente, este componente tomará la longitud que le corresponda justo como cualquier otro componente que no tenga fijada su longitud.

BoxLayout es un *layout manager* bastante desconocido y que desde luego como espero que hayáis podido comprobar puede dar mucho juego dentro de nuestros interfaces gráficos.

SpringLayout

Este *layout manager* está únicamente disponible a partir de la versión 1.4 del *JDK*. Su funcionamiento es muy similar a utilizar *struts* y áreas rígidas con la clase *Box* que vimos anteriormente. *SpringLayout* define los componentes que estarán dentro del interfaz y la relación entre estos componentes, es decir, el espacio que habrá entre ellos. *SpringLayout* intentará respetar siempre que pueda el tamaño preferido de los componentes.

El espacio entre los componentes se define utilizando objetos *Spring*. Cada objeto *Spring* tiene cuatro propiedades, sus tamaños máximo, mínimo y preferido junto con su tamaño actual. Todos estos objetos *Spring* se acumulan formando un conjunto de restricciones que estarán en un objeto de tipo *SpringLayout.Constraints*.

La mejor forma de comprender el funcionamiento de este *layout manager* es con un ejemplo explicado paso a paso:

```
import java.awt.*;
import javax.swing.*;

public class SpringLayoutDemo {

    public static void main(String[] args) {

        JFrame f = new JFrame();
        Container container = f.getContentPane();

        JButton[] botones = new JButton[]{
            new JButton("botón 1"),
            new JButton("botón 2"),
            new JButton("un botón grande"),
            new JButton("botón 4")
        };
        SpringLayout layout = new SpringLayout();
        container.setLayout(layout);
```

Como véis la primera parte es muy sencilla y por ahora sólo hemos creado los botones y establecido el *layout manager*.

```
        Spring xPad = Spring.constant(5);
        Spring ySpring = Spring.constant(5);
        Spring xSpring = xPad;
        Spring maxHeightSpring = Spring.constant(0);
```

Ahora lo que hemos hecho ha sido definir las variables *xSpring* e *ySpring* que representarán la posición del componente dentro del contenedor, la variable auxiliar *xPad* y una variable *maxHeightSpring* donde almacenaremos la altura máxima.

```
for (int i = 0; i < botones.length; i++) {
        container.add(botones[i]);
        SpringLayout.Constraints cons = layout.getConstraints(
                                        botones[i]);
        cons.setX(xSpring);
        xSpring = Spring.sum(xPad, cons.getConstraint("East"));
        cons.setY(ySpring);
        maxHeightSpring = Spring.max(maxHeightSpring,
                                cons.getConstraint("South"));
}
```

Esto ya es un poco más complejo. El primer paso que hacemos es añadir el botón a la interfaz; a continuación obtenemos sus *constraints* y establecemos su posición horizontal. Después ctualizamos la posición horizontal para el siguiente componente añadiendo *xPad* a *xSpring*. Para finalizar establecemos la posición vertical del componente y actualizamos la altura máxima.

Como habréis visto, tal y como dijimos anteriormente, el objeto *SpringLayout.Constraints* va acumulando las *constraints* para cada uno de los componentes.

```
        SpringLayout.Constraints pCons =
                                layout.getConstraints(container);
        pCons.setConstraint("East", xSpring);
        pCons.setConstraint("South",
                                Spring.sum(maxHeightSpring, ySpring));

        f.setTitle("Demo OverlayLayout");
        f.pack();
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

En el último paso establecemos el tamaño del contenedor en base al tamaño de sus componentes. El tamaño horizontal lo establecemos utilizando la variable *xSpring* que ha ido incrementando su valor mientras que el vertical se corresponderá con la altura del componente más alto del contenedor más un pequeño espacio de cinco pixels.

El resultado de todo esto es el siguiente:

¿Mucho trabajo para un resultado pobre? En este caso si, hubiera sido mucho más sencillo utilizar un *FlowLayout*, pero existen interfaces más complejos para los que *SpringLayout* nos puede ser bastante útil. Lamentablemente, todavía no existe la suficiente documentación sobre este *layout manager* salvo lo poco que se puede encontrar en el tutorial de *SUN*[3] y que parece que ni siquiera ellos saben como funciona.[1]

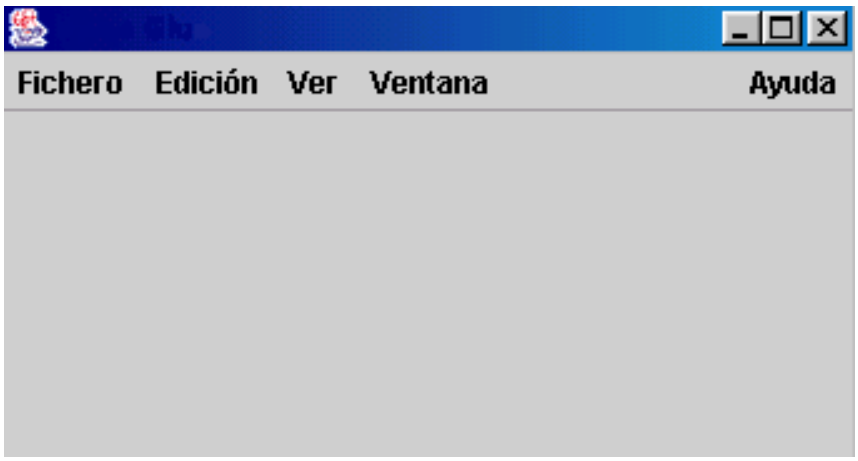
Usando *layout managers* con nuestros menus

Antes de finalizar este artículo voy a poner en este apartado un par de trucos que no todo el mundo conoce y que pueden ser de utilidad.

Como sabréis *javax.swing.JMenu* y *javax.swing.JMenuBar* heredan ambos de la clase *JComponent* y como todos los objetos de esta clase, son contenedores. Como contenedores, aunque mucha gente no lo sabe, son también susceptibles de aplicarseles *layout managers*.

El primer truco[2] sirve para crear un menu como los antiguos, en el cual el elemento de ayuda salga en el extremo más a la derecha. Para conseguir dicho efecto nada más sencillo que añadir un componente de pegamento entre el penúltimo de los elementos y el último:

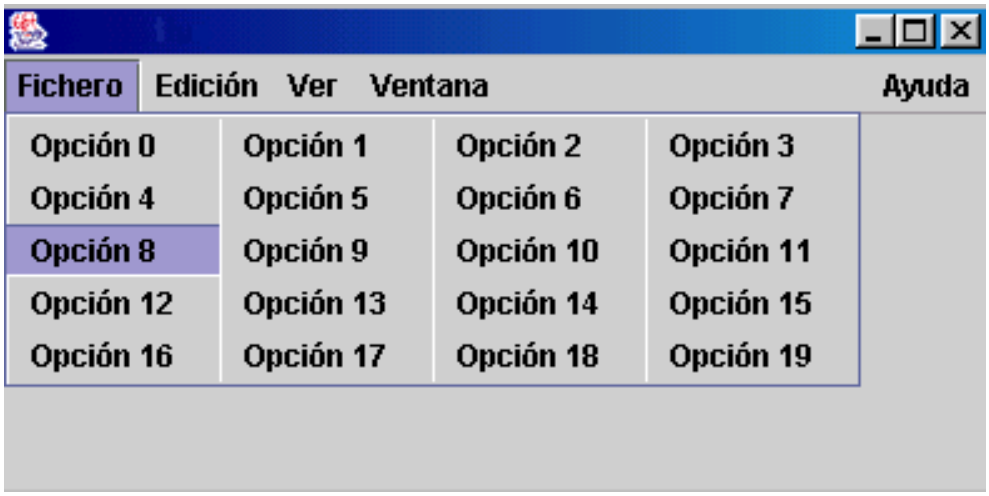
1.Es sorprendente ver como en el código fuente de los ejemplos había frases como "¿por qué no funciona esto?" o "Esto funciona. ¿Por qué?"
2.Este truco es bastante conocido ya que aparece en el tutorial de java de *SUN*[3]



Para los más despistados, veamos parte del código fuente:

```
....
JMenuBar menubar = new JMenuBar();
menubar.add(new JMenu("Fichero"));
menubar.add(new JMenu("Edición"));
menubar.add(new JMenu("Ver"));
menubar.add(new JMenu("Ventana"));
menubar.add(Box.createHorizontalGlue());
menubar.add(new JMenu("Ayuda"));
....
```

El segundo truco nos sirve para cuando tenemos un menú demasiado grande como para que quepa en pantalla. En un caso como este, lo más normal es que los elementos que no quepan dentro del contenedor simplemente permanezcan invisibles. Para solucionar este problema podemos añadirle al *JMenu* problemático un layout que cree varias columnas, por ejemplo *GridLayout*.



Como se puede ver así podemos aprovechar un poco más el espacio. El código necesario para realizar este efecto es el siguiente:

```
....
JMenu menu = new JMenu();
for (int i = 0; i < 20; i++) {
      menu.add(new JMenuItem("Opción " + i);
}
GridLayout layout = new GridLayout(5, 4);
menu.getPopupMenu().setLayout(layout);
....
```

Conclusiones y despedida

Con esto termina este tutorial de *layout managers*. Aunque confieso que en un principio estaba enfocado única y exclusivamente a explicar el funcionamiento del tan temido *GridBag-Layout* al final decidí intentar crear algo así como un "guía definitiva" de los *layout managers* y por supuesto en castellano.

En esta modesta "guía definitiva" hemos aprendido los secretos de *layout managers* tan sencillos como *FlowLayout* hasta los de *layout managers* tan complejos como *GridBagLayout*.

Como conclusión más importante destacaría que se deben utilizar *layout managers* siempre que sea posible ya que hacen nuestras aplicaciones más legibles, mantenibles y mucho más extensibles. Además correctamente usados pueden ofrecernos un buen rendimiento si

evitamos el recurso común de anidar paneles unos detrás de otros. Espero que después de haber leido todo este tutorial no tengáis demasiados problemas como para decidir el *layout manager* adecuado en cada momento.

Asimismo espero que os haya gustado a todos este tutorial, he intentado enfocarlo tanto a los más inexpertos como a la gente que ya tenía experiencia utilizando *layout managers*. Ahora ya es tarea vuestra el estimar si he conseguido mi objetivo.

Gracias por vuestro tiempo.

Referencias

[1] The Java Tutorial, a Practical Guide for Programmers, Mari Campione, Addison-Wesley Pub Co; ISBN: 0201703939; 3rd edition (15 Enero, 2000)

[2] The JFC Swing Tutorial: A Guide to Constructing GUIs, Kathy Walrath y Mari Campione, Addison-Wesley Pub Co; ISBN: 0201433214; Bk&Cd Rom edition (Julio 1999)

[3] The Java Tutorial, a Practical Guide for Programmers, SUN MicroSystems, online version http://java.sun.com/docs/books/tutorial/

[4] Complete Java 2 Certification Study Guide, Simon Roberds, Philip Heller y Michael Emest, Ed. Sybex; ISBN: 0782128254; 2nd edition, 2000

[5] Design Patterns: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson y J. Vlissides, Addison-Wesley, 1995

Contacto

Para cualquier duda sobre el contenido de este tutorial o cualquier otro tema relacionado, así como para el envio de comentarios, críticas, errores, alabanzas o simplemente saludos podéis dirigiros a la dirección de correo que figura en la cabecera o a **martin@javahispano.com**. Para cualquier duda sobre este y otros temas también podéis acudir a los foros de **http://www.javahispano.com**.