3.1.5. Administrador de distribución GridBagLayout

El administrador de distribución **`GridBagLayout`** divide el contenedor **`JPanel`** como si fuera una rejilla de celdas. Pero, a diferencia del administrador **`GridLayout`**, las celdas no tienen por qué tener el mismo tamaño. Cada componente puede ocupar una celda entera o sólo una parte. Si el componente va a ocupar un tamaño menor que la celda, puede estar centrado o alineado a algún borde de la celda. También es posible que un componente ocupe varias celdas. Al conjunto de celdas ocupadas por un componente se le denomina **área de visualización** de ese componente o **bolsa**.

Las características más importantes de este administrador se resumen a continuación:

Configuración asociada al componente al añadirle al panel – Se utiliza para especificar cosas como posición de la celda en la que incluir el componente, número de celdas del área de visualización, alineación, etc.… Toda esta configuración se establece con objetos de la clase **`java.awt.GridBagConstraints`**.

Configuración del administrador – No tiene. Se usa el constructor sin parámetros.

Configuración en el propio componente – Dependiendo de la configuración pasada con el componente al añadirlo al panel, se puede necesitar el tamaño mínimo.

Tamaño preferido del contenedor – La anchura se calcula como la suma de la máxima de las anchuras mínimas de cada componente de cada fila. La altura se calcula de forma equivalente. Es decir, el tamaño preferido del contenedor es aquel que hace que ningún componente se muestre más pequeño que su tamaño mínimo.

En las instancias de la clase **`GridBagConstraints`** se pueden configurar muchos atributos públicos para decidir el tamaño y posición del componente. También se deciden el número de filas y columnas de las celdas y el tamaño de las mismas. A continuación se describe como configurar cada uno de estos aspectos:

Número de filas y columnas de la tabla – No es necesario indicar las filas y columnas de la rejilla al construir el administrador como ocurre con `GridLayout`. Al añadir cada componente al `JPanel` se indica la fila y la columna de la celda en las que debe estar este y el número de celdas que ocupa (tanto de ancho como de alto). Con esta información se determina el número de filas y columnas de la rejilla.

Tamaño de las celdas y de la rejilla – Existen unos atributos públicos de la clase **`GridBagConstraints`** que controlan el tamaño de cada una de las celdas de la rejilla. Con sus valores por defecto la anchura de cada columna es la anchura mínima más grande de los componentes de esa columna. La altura de cada fila es la altura mínima más grande de los componentes de esa fila. Y la rejilla aparecería centrada dentro del contenedor como se puede apreciar en la figura 3.6.

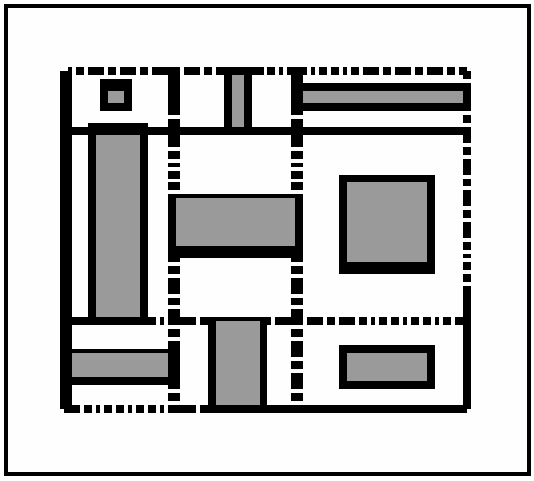

Figura 3.6: Forma de la rejilla de un administrador de distribución **`GridBagLayout`** con los valores de los atributos que controlan el tamaño de las celdas a su valor por defecto

A continuación se presentan y describen los diferentes atributos que se pueden configurar en una instancia de la clase **`GridBagConstraints`**, y cómo se usan para configurar la posición y tamaño de los componentes.

Los atributos o parámetros **`public double weightx`** y **`public double weighty`** sirven para especificar pesos a las columnas y filas respectivamente. Estos pesos serán los que influyan en el tamaño de las filas y las columnas. Aunque los pesos se especifican en los objetos **`GridBagConstraints`** y por tanto se asocian a cada componente, a cada fila y a cada columna corresponde un único valor. El valor de **`weightx`** para una columna se calcula como el máximo **`weightx`** de los componentes de esa columna. El valor de **`weighty`** de una fila se calcula como el máximo **`weighty`** de todos los componentes de esa fila. En la figura 3.7 se puede ver una representación gráfica de estos atributos.

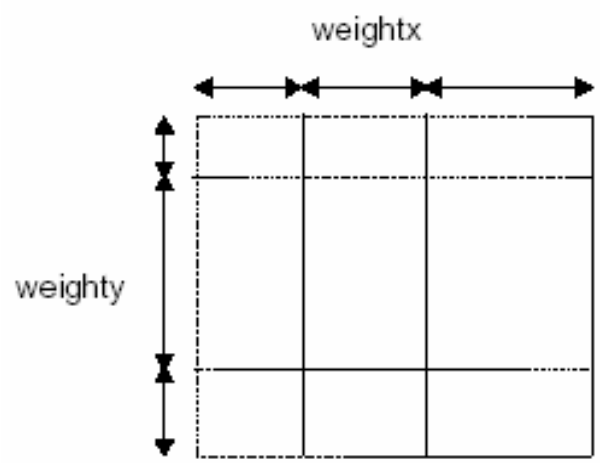

Figura 3.7: Atributos de pesos que influyen en el tamaño de las filas y columnas de la rejilla en un **`JPanel`** con administrador de distribución **`GridBagLayout.`**

Estos atributos pueden tomar valores entre 0.0 y 1.0. Cuando su valor es 0.0 (que es el valor por defecto) la rejilla aparecerá centrada en el `JPanel`. Si el valor del atributo **`weightx`** de alguna columna es mayor que 0.0, la rejilla tendrá la anchura del `JPanel`.

Todas las columnas seguirán con el mismo tamaño de antes excepto la columna con el valor mayor que 0.0, que ocupará **todo el espacio sobrante**. Esto se refleja de forma gráfica en la figura 3.8.

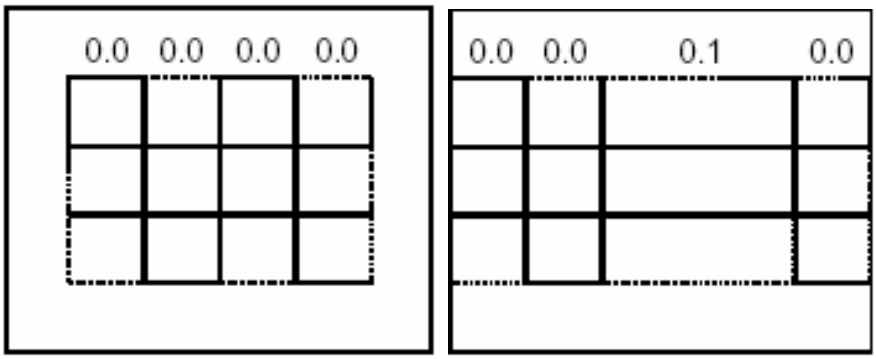

Figura 3.8: Ejemplo de distribución de las columnas de una rejilla según el atributo **`weightx`** de la clase **`GridBagConstraints`**

Para el caso de las filas ocurre algo similar. Si el valor del atributo **`weighty`** de alguna fila es mayor que 0.0, la tabla tendrá la altura del `JPanel`. Todas las filas seguirán con el mismo tamaño de antes excepto la fila con el valor mayor que 0.0, que ocupará **todo el espacio sobrante**. Esto se refleja de forma gráfica en la figura 3.9.

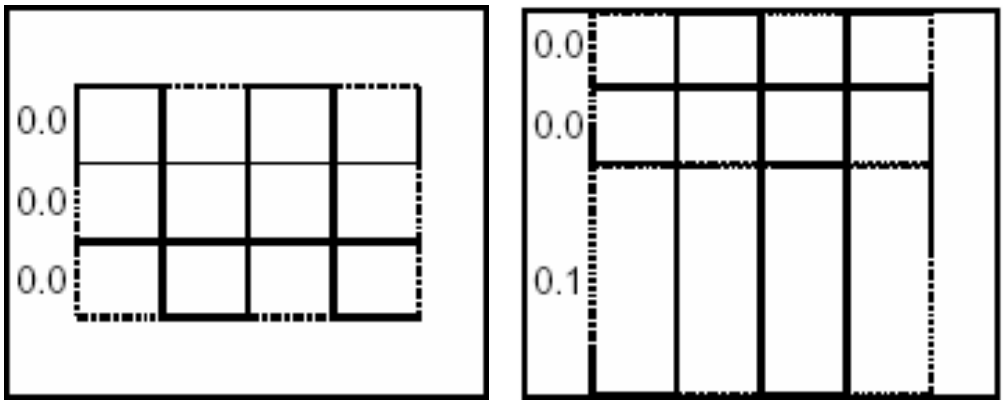

Figura 3.9: Ejemplo de distribución de las filas de una rejilla según el atributo **`weighty`** de la clase **`GridBagConstraints`**

Si existen dos filas o dos columnas, por ejemplo, con valores mayores que 0.0, se repartirán el tamaño sobrante en proporción a los valores. Si las dos tienen el mismo valor (sea cual sea) se **repartirán el tamaño sobrante a partes iguales**. Si una tiene un valor doble que la otra, implica que se llevará el doble del espacio sobrante. Es importante recalcar que si una columna tiene doble de peso que otra columna no tiene porque tener el doble de tamaño,

quiere decir que cada una tiene el tamaño preferido más la mitad del tamaño que sobra. Es decir, si el tamaño preferido de una columna es diferente que el tamaño preferido de otra, aunque tengan el mismo peso, nunca serán iguales.

El **área de visualización ocupada por el componente dentro de la rejilla** se determina con los siguientes atributos de la clase **`GridBagConstraints`**:

Los atributos **`public int gridx`** y **`public int gridy`** especifican la fila y la columna de la esquina superior izquierda de la bolsa del componente. El valor de estos atributos comienza con 0. Se puede especificar mediante un número entero o bien con la constante **`GridBagConstraints.RELATIVE`**, que es el valor por defecto de estos atributos. Usando esta constante el componente se coloca debajo (si se usa en el atributo **`gridy`**) o a la derecha (si se usa en el atributo **`gridx`**) del componente anteriormente incluido al `JPanel`.

Con el fin de facilitar la comprensión del código no se recomienda el uso de esta constante. La razón es porque al depender la colocación del componente anterior, no se sabría muy bien dónde se colocará. Mientras que si los valores de estos atributos se proporcionan con posiciones absolutas, se sabrá en todo momento el lugar dónde se coloca cada componente y, aunque alguno de ellos cambie, se seguirá sabiendo el valor exacto; sin embargo, si hay cambios y se utiliza en todos esta constante, no se sabrá dónde se encuentran.

Los atributos **`public int gridwith`** y **`public int gridheight`** especifican el número de celdas que ocupa la bolsa. El valor se puede especificar mediante un entero, siendo 1 el valor por defecto, o bien a través de las siguientes constantes:

`GridBagConstraints.REMAINDER`, la cual indica que el área de visualización abarca hasta el final de la fila (si al atributo **`gridwith`** se le da esta constante como valor) o hasta el inal de la columna (si al atributo **`gridheight`** se le da esta constante como valor).

`GridBagConstraints.RELATIVE`, la cual indica que el área de visualización abarca hasta el siguiente componente de la fila (si al atributo **`gridwith`** se le da esta constante como valor) o de la columna (si al atributo **`gridheigth`** se le da esta constante como valor).

La **posición del componente dentro del área de visualización** se determina usando los siguientes atributos de la clase **`GridBagConstraints`**:

El atributo **`public int anchor`** especifica la posición del componente dentro de su área de visualización cuando ésta es más grande que el tamaño del componente. El valor de este atributo se especifica a través de una de las siguientes constantes (cada constante se presenta acompañada de su correspondiente representación gráfica a modo de explicación):

- **`GridBagConstraints.NORTH`** 

- **`GridBagConstraints.SOUTH`** 
- **`GridBagConstraints.WEST`** 
- **`GridBagConstraints.EAST`** 
- **`GridBagConstraints.NORTHWEST`** 
- **`GridBagConstraints.NORTHEAST`** 
- **`GridBagConstraints.SOUTHWEST`** 
- **`GridBagConstraints.SOUTHEAST`** 
- **`GridBagConstraints.CENTER`** 

El valor por defecto del atributo **`anchor`** es **`GridBagConstraints.CENTER`**.

El **tamaño del componente dentro del área de visualización** se configura usando los siguientes atributos de la clase **`GridBagConstraints`**:

El atributo **`public int fill`** específica qué tamaño tiene el componente dentro del área de visualización. El valor de este atributo se especifica a través de una de las siguientes constantes (cada constante se presenta acompañada de su correspondiente representación gráfica a modo de explicación):

- **`GridBagConstraints.NONE`**, indica que el componente será del tamaño preferido.

- **`GridBagConstraints.HORIZONTAL`,** undica que la anchura será como la anchura del rea de visualización y su altura será la preferida.

- **`GridBagConstraints.VERTICAL`**, indica que la altura será como la altura del área de visualización y su anchura será la preferida.

- **`GridBagConstraints.BOTH`**, indica que es del mismo tamaño que el área de visualización.

El valor por defecto del atributo **`fill`** es **`GridBagConstraints.NONE`**.

El atributo **`public Insets insets`** especifica un espacio alrededor del componente. Por defecto no existe tal espacio. Para dar valor a este atributo se utiliza la clase **`java.awt.Insets`**. Esta clase tiene cuatro atributos públicos (**`bottom`**, **`left`**, **`right`**, **`top`**) para especificar el tamaño del espacio en cada lado y un constructor **`public Insets(int top, int left, int bottom, int right).`**

Tiene sentido usar el atributo **`insets`** cuando se usa el atributo **`fill`** o bien cuando el área de visualización es próximo al tamaño preferido del componente. Siempre se mantiene este espacio alrededor del componente. Una posible representación gráfica de ejemplo se muestra en la figura 3.10.

Figura 3.10: Ejemplo de espaciado alrededor de un componente del atributo **`insets`** de la clase **`GridBagConstraints`**

Los atributos **`public int ipadx`** y **`public int ipady`** se utilizan para aumentar la anchura preferida del componente en dos veces el valor de **`ipadx`** y la altura en dos veces el valor de **`ipady`**. Se aumenta en dos veces porque se aplica en los dos lados del componente. Se mide en píxeles.

El **tamaño preferido del contenedor** será aquél que hace que el tamaño de las filas y columnas sea el mayor tamaño mínimo de los componentes; es decir, el tamaño preferido es el que ocuparía la tabla si todos los pesos tuvieran valor 0.0.

El siguiente código muestra un esquema básico de un programa que usa el administrador de distribución **`GridBagLayout`**:

```
...
JPanel panelContenido = new JPanel();
GridBagLayout administrador = new GridBagLayout();
panelContenido.setLayout(administrador);
GridBagConstraints config = new GridBagConstraints();
config.gridx = ...;
config.gridy = ...;
config. ...
panelContenido.add(componente, config);
GridBagConstraints config2 = new GridBagConstraints();
config2.gridx = ...;
config2.gridy = ...;
config2. ...
panelContenido.add(componente2, config2);
...
```

A continuación se mostrarán varios ejemplos que muestren cómo construir algunas interfaces gráficas de usuario con las opciones estudiadas para posicionar y dimensionar un componente dentro de su área de visualización.

Ejemplo 3.2

Se van a crear varias aplicaciones con interfaz gráfica de usuario, donde cada una tendrá una ventana cuyo panel de contenido se distribuirá con un administrador de distribución **`GridBagLayout`**. Tan sólo existirá una celda con valor 1.0 para los atributos **`weightx`** y **`weighty`** de la clase **`GridBagConstraints`**, y un botón dentro de ella. Este botón se configurará de una manera diferente en cada una de las aplicaciones.

- **Ejemplo 3.2.1: Aplicación con el botón centrado en la celda y con su tamaño preferido.**

En la figura 3.11 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.11: Interfaz gráfica de usuario del ejemplo 3.2.1

El código fuente de la aplicación es el siguiente:

`Ejemplo321.java`

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ejemplo321 {

   public Ejemplo321() {

      JFrame ventana = new JFrame("Ejemplo 3.2.1");
      ventana.setSize(300, 200);
      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JButton boton = new JButton("Botón");

      JPanel panelDeContenido = new JPanel();

      panelDeContenido.setLayout(new GridBagLayout());

       GridBagConstraints config = new GridBagConstraints();

       config.weightx = 1.0;
       config.weighty = 1.0;

       panelDeContenido.add(boton, config);

       ventana.setContentPane(panelDeContenido);

       ventana.setVisible(true);
   }

   public static void main(String[] args) {
      new Ejemplo321();
   }
}
```

En la figura 3.12 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.12: Interfaz gráfica de usuario del ejemplo 3.2.2

El código fuente de la aplicación es el siguiente:

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ejemplo322 {

    public Ejemplo322() {

        JFrame ventana = new JFrame("Ejemplo 3.2.2");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton boton = new JButton("Botón");

        JPanel panelDeContenido = new JPanel();

        panelDeContenido.setLayout(new GridBagLayout());

        GridBagConstraints config = new GridBagConstraints();

        config.weightx = 1.0;
        config.weighty = 1.0;

        config.anchor = GridBagConstraints.EAST;
        config.insets = new Insets(5,5,5,5);

        panelDeContenido.add(boton, config);

        ventana.setContentPane(panelDeContenido);

        ventana.setVisible(true);
    }

    public static void main(String[] args) {
        new Ejemplo322();
    }
}
```

- **Ejemplo 3.2.3: Aplicación con el botón cuya altura sea la del panel de contenido y su anchura sea la preferida más 8 píxeles.**

En la figura 3.13 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.13: Interfaz gráfica de usuario del ejemplo 3.2.3

El código fuente de la aplicación es el siguiente:

Ejemplo323.java

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ejemplo323 {

   public Ejemplo323() {

      JFrame ventana = new JFrame("Ejemplo 3.2.3");
      ventana.setSize(300, 200);
      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JButton boton = new JButton("Botón");

      JPanel panelDeContenido = new JPanel();

      panelDeContenido.setLayout(new GridBagLayout());

      GridBagConstraints config = new GridBagConstraints();

      config.weightx = 1.0;
      config.weighty = 1.0;

      config.fill = GridBagConstraints.VERTICAL;
      config.ipadx = 4;

      panelDeContenido.add(boton, config);

      ventana.setContentPane(panelDeContenido);

      ventana.setVisible(true);
   }
   public static void main(String[] args) {
      new Ejemplo323();
   }
}
```

- **3.2.4: Aplicación con el botón alineado en la parte superior y que ocupe la anchura del panel de contenido.**

En la figura 3.14 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.14: Interfaz gráfica de usuario del ejemplo 3.2.4

El código fuente de la aplicación es el siguiente:

`Ejemplo324.java`

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class Ejemplo324 {
public Ejemplo324() {
JFrame ventana = new JFrame("Ejemplo 3.2.4");
ventana.setSize(300, 200);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JButton boton = new JButton("Botón");
JPanel panelDeContenido = new JPanel();
panelDeContenido.setLayout(new GridBagLayout());
GridBagConstraints config = new GridBagConstraints();
config.weightx = 1.0;
config.weighty = 1.0;
config.fill = GridBagConstraints.HORIZONTAL;
config.anchor = GridBagConstraints.NORTH;
panelDeContenido.add(boton, config);
ventana.setContentPane(panelDeContenido);
ventana.setVisible(true);
public static void main(String[] args) {
new Ejemplo324();
}}
```

En los ejemplos anteriores se ha mostrado cómo colocar componentes dentro de su área de visualización. Ahora, a través del siguiente ejemplo, se pretende ver cómo configurar el tamaño de las celdas cuando hay más espacio del preferido.

Ejemplo 3.3

Se van a crear varias aplicaciones con interfaz gráfica de usuario, donde cada una tendrá una ventana cuyo panel de contenido se distribuirá con un administrador de distribución **`GridBagLayout`**.

- **Ejemplo 3.3.1: Aplicación con cuatro botones, todos ellos con el mismo tamaño y que se encuentren agrupados en el centro del panel de contenido.**

En la figura 3.15 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.15: Interfaz gráfica de usuario del ejemplo 3.3.1

El código fuente de la aplicación es el siguiente:

Ejemplo331.java

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class Ejemplo331 {
public Ejemplo331() {
JFrame ventana = new JFrame("Ejemplo 3.3.1");
ventana.setSize(300, 200);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JButton boton1 = new JButton("Botón 1");
JButton boton2 = new JButton("Botón 2");
JButton boton3 = new JButton("Botón 3");
JButton boton4 = new JButton("Botón 4");
```

```
JPanel panelDeContenido = new JPanel();
panelDeContenido.setLayout(new GridBagLayout());
GridBagConstraints config1 = new GridBagConstraints();
config1.gridx = 0;
config1.gridy = 0;
panelDeContenido.add(boton1, config1);
GridBagConstraints config2 = new GridBagConstraints();
config2.gridx = 1;
config2.gridy = 0;
panelDeContenido.add(boton2, config2);
GridBagConstraints config3 = new GridBagConstraints();
config3.gridx = 0;
config3.gridy = 1;
panelDeContenido.add(boton3, config3);
GridBagConstraints config4 = new GridBagConstraints();
config4.gridx = 1;
config4.gridy = 1;
panelDeContenido.add(boton4, config4);
ventana.setContentPane(panelDeContenido);
ventana.setVisible(true);
public static void main(String[] args) {
new Ejemplo331();
}}
```

- **3.3.2: Aplicación con cuatro botones. Los botones de la primera columna tendrán el tamaño preferido, pero los botones de la segunda columna deberán ocupar todo el tamaño restante del panel de contenido.**

En la figura 3.16 se representa la interfaz gráfica de usuario con diferentes tamaños.

Figura 3.16: Interfaz gráfica de usuario del ejemplo 3.3.2

El código fuente de la aplicación es el siguiente:

Ejemplo332.java

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class Ejemplo332 {
public Ejemplo332() {
JFrame ventana = new JFrame("Ejemplo 3.3.2");
ventana.setSize(300, 200);
ventana.setDefaultCloseOpe

ration(JFrame.EXIT_ON_CLOSE);
JButton boton1 = new JButton("Botón 1");
JButton boton2 = new JButton("Botón 2");
JButton boton3 = new JButton("Botón 3");
JButton boton4 = new JButton("Botón 4");
JPanel panelDeContenido = new JPanel();
panelDeContenido.setLayout(new GridBagLayout());
GridBagConstraints config1 = new GridBagConstraints();
config1.gridx = 0;
config1.gridy = 0;
config1.weightx = 0.0;
panelDeContenido.add(boton1, config1);
GridBagConstraints config2 = new GridBagConstraints();
config2.gridx = 1;
config2.gridy = 0;
config2.weightx = 1.0;
config2.fill = GridBagConstraints.HORIZONTAL;
panelDeContenido.add(boton2, config2);
GridBagConstraints config3 = new GridBagConstraints();
config3.gridx = 0;
config3.gridy = 1;
config3.weightx = 0.0;
panelDeContenido.add(boton3, config3);
GridBagConstraints config4 = new GridBagConstraints();
config4.gridx = 1;
config4.gridy = 1;
config4.weightx = 1.0;
config4.fill = GridBagConstraints.HORIZONTAL;
panelDeContenido.add(boton4, config4);
ventana.setContentPane(panelDeContenido);
ventana.setVisible(true);
public static void main(String[] args) {
new Ejemplo332();
  }
}
```

A continuación se dan algunos detalles de parte de la API Swing que pueden resultar muy útiles, como se viene haciendo a lo largo de algunos de los capítulos del presente libro.

Aprendiendo la API. Barras de desplazamiento

Cuando el tamaño preferido de un componente es muy grande, pero se desea que se muestre completamente, se puede hacer uso de las barras de desplazamiento. Para que un componente se muestre con barras de desplazamiento si su tamaño preferido es mayor que el tamaño disponible, se usa el contenedor **`JScrollPane`** usando el siguiente esquema:

```
...
JScrollPane componenteConScroll = new JScrollPane(componente);
...
JPanel panel = ...
...
panel.add(componenteConScroll);
...
```

Aprendiendo la API. Texto multilínea

Un componente que permite introducir texto de una línea es **`JTextField`**. Y para textos de varias líneas se usa el componente **`JTextArea`**. Los métodos más usados de este último componente son los siguientes:

- `public void setText(String texto)`: Este método se utiliza para insertar texto en el componente.
- `public String getText()`: Este método se utiliza para obtener el texto del componente.
- `public void append(String texto)`: Este método se utiliza para añadir texto al ya existente dentro del componente.

Se suele usar este componente dentro de un contenedor **`JScrollPane`**, para el caso particular en el que el número de líneas haga que el componente sea más grande que el espacio asignado por el administrador de distribución.

Aprendiendo la API. Botones de radio

Para crear botones de radio es necesario instanciar objetos de la clase **`JRadioButton`** para cada botón de radio que se desee crear. Los métodos más importantes de la clase **`JRadioButton`** se describen a continuación:

- `public boolean isSelected()`**:** Este método determina si el botón está o no seleccionado.
- `public void setSelected(boolean seleccionado)`**:** Método que se usa para establecer la selección o no del botón.

Para que un conjunto de botones de radio se agrupen de forma que sólo uno de ellos pueda estar seleccionado, es necesario usar un objeto de una clase auxiliar, concretamente un

objeto de la clase **`ButtonGroup`**. De manera que, para agrupar los botones de radio, simplemente habrá que invocar el método de la clase **`ButtonGroup`**:

- `public void add(AbstractButton botón)`**:** Este método añade el botón al grupo de botones, de forma que sólo uno de los botones del grupo podrá estar seleccionado en un momento dado.

Es importante destacar que la clase **`ButtonGroup`** no ofrece muchas facilidades para determinar el botón de radio que se encuentra seleccionado en cada momento. Si se quiere disponer de esta funcionalidad de forma cómoda y reutilizable, será necesario programarla. Además, **`ButtonGroup`** no es un componente y no debe ser añadido a ningún contenedor, su única funcionalidad es asegurarse de quitar la selección al botón de radio que la tuviera cuando se selecciona uno nuevo.

Por otra parte, si se desea ejecutar código cuando el usuario seleccione un nuevo botón de radio y, es decir, cuando cambie el botón seleccionado, habrá que asociar código al evento de tipo **Action**.

Pero si se desea seleccionar un botón de radio desde el código del programa, no desde la interfaz, no se generará este evento. Por este motivo, los botones de radio generan eventos de tipo **Item** cuando su estado pasa de seleccionado a no seleccionado o viceversa.

Practica7 – Aplicación de traducción 8
Para mostrar tanto las capacidades de los administradores de distribución como los nuevos elementos de la API mencionados se pretende ampliar la aplicación de traducción 7. Permitirá la traducción de textos formados por varias palabras. Para ello, se añadirá una ventana que contenga un área de texto para poder introducir el texto que se desea traducir, y otra área de texto para mostrar el texto resultante de la traducción.

Además, se cambiará el botón y la etiqueta, que marcaba el sentido de la traducción en versiones anteriores de la aplicación, por dos botones de radio que se encargarán ahora de esa funcionalidad. La función de rollover será suprimida de la aplicación.

La interfaz gráfica de usuario de la aplicación se presenta en la figura 3.17.

Figura 3.17: Interfaz de usuario de la aplicación de traducción de textos formados por varias palabras

Algo importante que debe cumplir la interfaz gráfica es que si la ventana de aplicación es redimensionada, los componentes se deben distribuir correctamente. Un ejemplo de esta posible redistribución se puede ver en la figura 3.18, cuyo tamaño de ventana de aplicación ha cambiado sustancialmente con respecto al tamaño de la ventana de aplicación mostrada en la figura 3.17.

Figura 3.18: Interfaz de usuario de la aplicación de traducción de textos formados por varias palabras redimensionada

Para implementar este ejemplo se modificarán diversas clases. La clase **`VentanaTraductor`** se modificará para no establecer el tamaño de forma fija, sino que se calculará para que el panel de contenido tenga su tamaño preferido. Por supuesto, la otra clase que cambiará será **`PanelTraductor`**. Se pide implementar el código de ambas clases.