CAPITULO 11. CONSTRUIR INTERFACES GRAFICAS DE USUARIO

11.1 Introducción

La *Interfaz de usuario*, es la parte de una aplicación que está visible en la pantalla y que permite que un usuario interactúe con ella. GUI es la Graphic User Interface.

11.2 Componentes, gestores de disposición y captura de eventos

En Java, toda la programación de una GUI se realiza mediante el uso de bibliotecas de clases estándar especializadas.

Los principios que necesitamos comprender se pueden dividir en tres áreas:

- ¿Qué clase de elementos podemos mostrar en una pantalla?
- ¿Cómo podemos acomodar estos elementos?
- ¿Cómo podemos reaccionar ante una entrada del usuario?

Discutiremos estas cuestiones mediante los términos *componentes*, *gestores de disposición* y *manejo de eventos.*

Una GUI se construye mediante **componentes** que se ubican en la pantalla. Los componentes se representan mediante objetos. Son cosas tales como botones, menús, campos de texto, etc. La biblioteca de Java contiene una buena cantidad de componentes listos para usar y también podemos escribir los propios.

La distribución de los componentes en la pantalla se *lleva* a cabo mediante gestores de disposición. Los sistemas de GUI más viejos y primitivos manejaban coordenadas bidimensionales. En los sistemas de GUI más modernos, debemos tener en cuenta distintas resoluciones de pantalla, diferentes fuentes, ventanas que los usuarios pueden redimensionar, y muchos otros aspectos que vuelven mucho más dificultosa la distribución de los componentes. La solución será un esquema en el que podamos especificar la disposición de los componentes en términos más generales.

La terminología **manejo de eventos** hace referencia a la tarea de reaccionar a los *eventos* que produce el usuario como por ejemplo, hacer clic sobre el botón del ratón o ingresar algo por teclado. El modelo que usa la biblioteca de Java para lograr esto se basa en eventos: si un usuario activa un componente (por ejemplo, presiona un botón) el sistema generará un evento. Entonces, nuestra aplicación puede recibir una notificación del evento (mediante la invocación de uno de sus métodos) y podemos llevar a cabo la acción adecuada.

11.3 AWT y Swing

Java tiene dos bibliotecas para la construcción de interfaces gráficas de usuario. La más antigua se denomina *AWT (Abstract Window Toolkit)* y fue introducida con el primer sistema Java original; más tarde, se agregó una biblioteca mucho mejor de nombre *Swing*.

Swing utiliza algunas de las clases de la biblioteca AWT, reemplaza alguna de las clases de AWT con sus propias versiones y agrega muchas clases nuevas. Usaremos la biblioteca Swing. Como existen clases equivalentes en AWT y en Swing, las versiones Swing han sido identificadas mediante el agregado de la letra “J” al comienzo del nombre de la clase, como Button y JButton,

11.4 El ejemplo visor de imágenes

Formato de imagen. Las imágenes se pueden almacenar en diferentes formatos. Las diferencias tienen que ver principalmente con el tamaño del archivo y con la información que contienen.

11.4.1 Primeros experimentos: crear una ventana

Se usarán objetos de las clases de los paquetes java.awt, java.awt.event y javax.swing

Casi todo lo que se puede ver en una GUI está contenido en un tipo de ventana del más alto nivel. Una ventana del nivel más alto es una ventana que está bajo el control del administrador de ventanas del sistema operativo y que típicamente puede moverse, cambiar de tamaño, minimizarse y maximizarse de manera independiente. En Java, esta ventanas del más alto nivel se denominan *frames* y en Swing se representan mediante la clase de nombre JFrame.

Con esta línea de código: ventana = new JFrame("Visor de Imágenes"); se crea una nueva ventana y es almacenada en nuestra variable de instancia, para poder usarla más adelante.

Una ventana consta de tres partes: la *barra del título,* una *barra de menú* opcional y un *panel contenedor*. La apariencia exacta de la barra del título depende del sistema operativo que se esté usando. Generalmente, contiene el título de la ventana y unos pocos controles para la ventana. La barra de menú y el panel contenedor están bajo el control de la aplicación. Los componentes se ubican en una ventana agregándolos a la barra de menú o al panel contenedor.

11.4.2 Agregar componentes simples

Inmediatamente después de la creación del JFrame, la ventana no estará visible y su panel contenedor estará vacío. Continuamos el trabajo agregando una etiqueta al panel contenedor:

```
Container panelContenedor = ventana. getContentPane();
JLabel etiqueta = new JLabel( "Soy una etiqueta. " );
panelContenedor.add(etiqueta);
```

La primera línea obtiene el panel contenedor de la ventana. Siempre debemos hacer esto: los componentes de la GUI se agregan a la ventana agregándolos al panel contenedor de la misma. El panel contenedor es en sí mismo de tipo Container. Un contenedor es un componente Swing que puede contener un grupo arbitrario de otros componentes, prácticamente de la misma manera en que un ArrayList puede contener una colección arbitraria de objetos.

Luego, creamos un componente etiqueta (de tipo JLabel) y lo agregamos al panel contenedor. Una etiqueta es un componente que puede mostrar texto o alguna imagen, o ambas cosas a la vez.

Finalmente, tenemos las dos líneas ventana.pack(); ventana.setVisible(true); La primera línea hace que la ventana distribuya adecuadamente los componentes dentro de ella y le asigne el tamaño apropiado. Siempre tenemos que invocar el método pack sobre la ventana después de haber agregado o modificado el tamaño de sus componentes. La última línea finalmente hace que la ventana se vuelva visible en la pantalla. Siempre comenzamos con una ventana que inicialmente es invisible, por lo que podemos acomodar todos los componentes dentro de ella sin que este proceso sea visible en la pantalla. Luego, cuando la ventana esté construida, podemos mostrarla en u estado completo.

11.4.3 Agregar menús

Primero, creamos los menús. Las tres clases involucradas en esta tarea son:

- JMenuBar - Un objeto de esta clase representa una barra de menú que se puede mostrar debajo de la barra de título, en la parte superior de una ventana Cada ventana tiene un JMenuBar como máximo.
- JMenu - Los objetos de esta clase representan un solo menú (como por ejemplo, los menús comunes «Archivo», «Edición» o «Ayuda»). Los menús frecuentemente están contenidos en una barra de menú; también pueden aparecer en menús emergentes.
- JMenuItem - Los objetos de esta clase representan un solo elemento de menú dentro de un menú, como por ejemplo, “Abrir” o “Grabar”.

La clase JFrame tiene un método de nombre setJMenuBar. Podemos crear una barra de menú y usar este método para adjuntar nuestra barra de menú a la ventana:

```
JMenuBar barraDeMenu = new JMenuBar();
ventana.setJMenuBar(barraDeMenu);
```

Ahora estamos listos para crear un menú y agregarl o a la barra de menú:

```
JMenu menuArchivo = new JMenu(“Archivo”) ;
barraDeMenu.add(menuArchivo);
```

Estas dos líneas crean un menú con la etiqueta “Archivo” y lo insertan en la barra de menú. Finalmente, podemos agregar elementos al menú. Las siguientes líneas agregan dos elementos con las etiquetas “Abrir” y “Salir” al menú “Archivo”.

```
JMenuItem elementoAbrir = new JMenuItem(“Abrir”);
menuArchivo.add(elementoAbrir);
JMenuItem elementoSalir = new JMenuItem(“Salir”);
menuArchivo.add(elementoSalir);
```

Ahora tenemos que agregar código para reaccionar a las selecciones del menú.

11.4.4 Manejo de eventos

Swing usa un modelo muy flexible para reaccionar ante los ingresos que se producen en la GUI: un modelo de *manejo de eventos* mediante *oyentes de eventos*.

El marco de trabajo Swing y algunos de sus componentes disparan eventos cuando ocurre algo en que otros objetos pueden estar interesados. Existen diferentes tipos de eventos provocados por diferentes tipos de acciones: cuando se presiona un botón o se selecciona un elemento de un menú, el componente dispara un *ActionEvent;* cuando se presiona un botón del ratón o se

mueve el ratón, se dispara un *MouseEvent;* cuando se cierra una ventana o se la transforma en icono, se genera un *WindowEvent.* Existen muchos otros tipos de eventos.

Cualquiera de nuestros objetos puede convertirse en oyente de cualquiera de estos eventos. Un objeto oyente se notificará de cualquiera de los eventos que es capaz de oír.

Un objeto puede escuchar los eventos de los componentes implementando una interfaz oyente de eventos. Si implementa la interfaz correcta, puede registrarse a sí mismo como uno de los componentes al que quiere oír.

Los elementos del menú (clase JMenuItem) disparan eventos de acción (ActionEvents) cuando son activados por un usuario. Los objetos que desean oír estos eventos deben implementar la interfaz ActionListener del paquete java.awt.event. Hay dos estilos alternativos para la implementación de oyentes de eventos: un único objeto oye los eventos provenientes de varias fuentes diferentes o bien, a cada fuente de eventos diferente se le asigna su propio y único oyente.

11.4.5 Recepción centralizada de eventos

Para lograr que nuestro objeto VisorDeImagen se convierta en el único oyente de todos los eventos que provienen del menú tenemos que hacer tres cosas:

l. Debemos declarar, en el encabezado de la clase, que implementa la interfaz ActionListener:

```
public class VisorDeImagen implements ActionListener{…}
```

2. Tenemos que implementar un método con la signatura

```
public void actionPerformed (ActionEvent e){…}
```

Este es el único método que se define en la interfaz ActionListener.

3. Debemos invocar al método addActionListener del elemento del menú para registrar al objeto VisorDeImagen como un oyente.

```
elementoAbrir.addActionListener(this);
```

Los números 1 y 2, la implementación de la interfaz y la definición de su método, aseguran que nuestro objeto es un subtipo de ActionListener. Luego, el número 3 registra nuestro propio objeto como un oyente de los elementos del menú.

El efecto de registrar nuestro objeto como un oyente a través del elemento del menú, es que se invocará nuestro propio método actionPerformed mediante el elemento del menú, cada vez que se active este elemento. Cuando se invoque nuestro método, el elemento del menú será pasado como un parámetro de tipo ActionEvent que proporciona algunos detalles sobre el evento que ha ocurrido. Estos detalles incluyen el momento exacto del evento, el estado de las teclas modificadoras (control, shift y metateclas) y una "cadena de comando", entre otras cosas. La cadena de comando es una cadena que, de alguna manera, identifica al componente que produjo el evento. Para los elementos del menú, esta identificación se realiza, por defecto, mediante el texto de la etiqueta del elemento.

La dependencia de la cadena de la etiqueta del elemento para llevar a cabo la función correspondiente no es una buena idea. Si cambia la etiqueta, por ejemplo por una traducción, se dejará de identificar quien ha provocado la acción.

El hecho de que el despacho de métodos esté centralizado (tal como lo hace nuestro actionPerformed) no es una buena estructura para nada. Esencialmente, construimos un único método sólo para luego escribir código tedioso en el que invocamos a los métodos separados correspondientes a cada elemento del menú. Esto no tiene sentido en términos de mantenimiento: para cada elemento adicional del menú tendremos que agregar una nueva sentencia if en el método actionPerformed. También parece ser un esfuerzo en vano. Sería mucho mejor si pudiéramos hacer que cada elemento del menú invoque directamente a cada método por separado.

11.4.6 Clases internas

Para solucionar los problemas que presenta el despacho centralizado de métodos que mencionamos anteriormente, usamos una nueva construcción que no hemos tratado con anterioridad: las *clases internas.* Las clases internas son clases que se declaran textualmente dentro de otra clase:

```
class ClaseEnvolvente
{…
      Class ClaseInterna{…}
…}
```

Las instancias de la clase interna se adjuntan a las instancias de la clase envolvente: sólo pueden existir dentro de una instancia que las envuelva. Las instancias de las clases internas conceptualmente existen en el *interior* de una instancia que las envuelve. Un detalle interesante es que las sentencias de los métodos de la clase interna pueden ver y acceder a los campos y

métodos privados de la clase envolvente. La clase interna se considera una parte de la clase externa que la envuelve, al igual que cualquiera de los métodos de la clase envolvente.

```
class VisorDelmagen{…
    class AbrirActionListener implements ActionListener{
        public void actionPerformed (ActionEvent evento){…}/ / lleva a cabo la acción abrir
    }
    class salirActionListener implements ActionListener{
        public void actionPerformed (ActionEvent evento){…}/ / lleva a cabo la acción de salir
    }
}
```

Como una guía de estilo, generalmente escribimos las clases internas al final de la clase envolvente, a continuación de los métodos.

Una vez que hemos escrito estas clases internas, podemos crear instancias de estas clases internas exactamente de la misma manera en que lo hacemos a partir de cualquier otra clase. Observe que VisorDelmagen ya no implementa ActionListener pues hemos eliminado su método actionPerformed, pero sí lo hacen las dos clases internas. Esto nos permite usar instancias de las clases internas como oyentes de acción de los elementos del menú:

```
JMenuItem elementoAbrir = new JMenuItem("Abrir");
elementoAbrir.addActionListener(new AbrirActionListener());
```

Generalmente, se pueden usar las clases internas en algunos casos para mejorar la cohesión de proyectos grandes.

11.4.7 Clases internas anónimas

La solución al problema del despacho de las acciones que utiliza clases internas es bastante buena pero queremos avanzar un poco más: podemos usar *clases internas anónimas*.

El código relevante es como sigue:

```
JMenuItem elementoAbrir = new JMenuItem( "Abrir");
elementoAbrir.addActionListener(new ActionListener() {
        public void actionPerformed (ActionEvent e) {
        archivoAbrir();
        }
} ) ;
```

Lo que está viendo en este fragmento de código es una clase interna anónima. La idea de esta construcción está basada en la observación de nuestra versión anterior, que usó cada clase interna exactamente una y sólo una vez para crear una única instancia. En esta situación, las clases internas anónimas ofrecen un atajo sintáctico: nos permiten definir una clase y crear una sola instancia de ella, todo en un solo paso. El efecto es idéntico al de la clase interna de la versión anterior, con la diferencia de que no es necesario definir nombres para cada una de las clases oyentes, y que la definición del método oyente está más cerca de la registración del oyente del elemento del menú. Cuando usamos una clase interna anónima, creamos una clase interna que no tiene *ningún nombre* e inmediatamente creamos una sola instancia de esa clase.

Se crea una clase interna anónima nombrando un supertipo(frecuentemente, una clase abstracta o una interfaz, en este caso, ActionListener), seguido de un bloque que contiene una implementación para sus métodos abstractos.

En este ejemplo, creamos una nueva subclase de ActionListener que implementa el método actionPerformed. Esta nueva subclase no recibe un nombre; en su lugar, la prefijamos con la palabra clave new para crear una sola instancia de esta clase.

En nuestro ejemplo, esta instancia es un objeto oyente de acción, ya que es un subtipo de ActionListener. Puede pasarse al método addActionListener del elemento del menú y luego invocar al método archivoAbrir de su clase envolvente, cuando se active el elemento del menú.

De la misma manera que las clases internas que tienen nombre, las clases internas anónimas pueden acceder a los campos y métodos de su clase envolvente. Además, dado que están definidas dentro de un método, pueden acceder a las variables locales y a los parámetros de dicho método. Sin embargo, una regla importante es que las variables locales accedidas de esta manera deben ser declaradas como final.

El uso de clases internas anónimas hace que el código resulte bastante más difícil de leer. Recomendamos fuertemente usar estas clases sólo dentro de clases muy cortas y sólo para modismos de código bien determinados.

Con frecuencias usamos clases anónimas en los lugares en los que se requiere la implementación de una sola instancia: las acciones asociadas con cada elemento del menú son únicas para ese elemento en particular. Además, siempre se hará referencia

a la instancia mediante su supertipo. Ambas razones quieren decir que el nombre de la nueva clase no es tan necesario, por lo tanto, puede ser anónima.

Las clases internas anónimas son construcciones muy útiles para implementar oyentes de eventos.

Algo que queremos destacar es que necesitamos invocar a ventana.pack () en el final del método archivoAbrir pues se modificó el tamaño de nuestro componente para mostrar la imagen. El método pack recalculará la disposición de los componentes en la ventana y dibujará nuevamente la ventana, por lo tanto, el cambio de tamaño se maneja adecuadamente.

11.5.3 Esquemas de disposición

Primeramente, trabajaremos en la tarea de agregar dos etiquetas en la interfaz: una ubicada en la parte superior de la imagen que se usa para mostrar el nombre del archivo de la imagen que se muestra actualmente y otra ubicada en la parte inferior que se usa para mostrar varios mensajes de estado.

La creación de estas etiquetas es fácil: ambas son simples instancias de JLabel. La única cuestión que nos falta resolver es cómo acomodarlas en la pantalla.

Un primer intento (simplificado e incorrecto) podría ser este:

```
Container panelContenedor = ventana. getContentPane () ;
etiquetaNombreDeArchivo = new Jlabel();
panelContenedor.add(etiquetaNombreDeArchivo);
panelDelmagen = new PanelDelmagen () ;
panelContenedor.add(panelDelmagen);
etiquetaEstado = new Jlabel( "Versión 1.0");
panelContenedor.add (etiquetaEstado);
```

La idea de este código es simple: tomamos el panel contenedor de la ventana y agregamos uno tras otro, los tres componentes que queremos mostrar. El único problema es que no hemos especificado exactamente cómo se ubicarán estos tres componentes. Podríamos querer que aparezcan uno cerca del otro, o uno debajo del otro, o alguna otra disposición posible. Como no hemos especificado ninguna disposición en especial, el contenedor (el panel contenedor) utiliza un comportamiento por defecto, y esto no es lo que queremos.

Swing usa *gestores de disposición* para acomodar los componentes en una GUI. Cada contenedor que contiene componentes, por ejemplo, un panel, tiene un gestor de disposición asociado que se encarga de acomodar los componentes dentro del contenedor.

Swing proporciona varios gestores de disposición diferentes que soportan las diferentes preferencias de ubicación de los componentes. Los esquemas de disposición más importantes son: FlowLayout, BorderLayout, GridLayout y BoxLayout, cada uno de los cuales está representado por una clase Java en la biblioteca Swing y cada uno de los mismos dispone los componentes que tiene bajo su control de diferentes maneras.

El gestor de disposición FlowLayout acomoda todos los componentes secuencialmente, de izquierda a derecha. Deja cada componente en su tamaño preferido y los centra horizontalmente. Si el espacio horizontal no es suficiente para ajustar todos los componentes, los ubica en una segunda línea. También se puede configurar el esquema FlowLayout para alinear los componentes a la izquierda o a la derecha.

BorderLayout ubica cinco componentes con una disposición específica: uno en el centro y cada uno de los restantes en la parte superior, en la parte inferior, a la izquierda y a la derecha. Cada una de estas posiciones puede quedar vacía de modo que podría contener menos de cinco componentes. Los nombres de las cinco posiciones son: CENTER, NORTH, SOUTH, EAST Y WEST.

Cuando se cambia el tamaño de un BorderLayout, el componente central es el único que se modifica en ambas dimensiones. Los componentes ubicados al este y al oeste cambian su alto, pero no su ancho. Los componentes ubicados al norte y al sur mantienen su alto y sólo modifican su ancho.

El esquema GridLayout, tal como su nombre sugiere, es muy útil para ubicar componentes en una rejilla. Se puede especificar el número de filas y de columnas y el gestor de disposición GridLayout mantendrá siempre todos los componentes con el mismo tamaño. Puede ser útil por ejemplo, para forzar a que los botones tengan el mismo ancho. El ancho de las instancias de JButton se determina inicialmente mediante el texto del botón: cada botón se construye suficientemente ancho como para mostrar su texto completo. La inserción de botones en un GridLayout dará por resultado que todos los botones cambiarán de tamaño para que coincidan con el del botón más ancho.

BoxLayout ubica varios componentes vertical y horizontalmente. No arma otra línea cuando cambia el tamaño de los componentes. Mediante el anidado de varios esquemas BoxLayout, es decir, colocar uno dentro del otro, se pueden construir disposiciones de componentes en dos dimensiones, sofisticadas y alineadas.

11.5.4 Contenedores anidados

Todas las estrategias de disposición de componentes discutidas previamente son sumamente simples. La clave para construir interfaces que tengan un buen aspecto y un buen comportamiento reside en un último detalle: se deben anidar los esquemas de disposición. Algunos de los componentes Swing son *contenedores.* Desde afuera, los contenedores se presentan como si fueran componentes simples, pero pueden contener muchos otros componentes. Cada contenedor tiene asociado su propio gestor de disposición.

El contenedor que más se usa es el de la clase JPanel. Se puede insertar un JPanel en el panel contenedor de una ventana de la misma manera que un componente y luego, se pueden colocar más componentes dentro del JPanel.

El contenedor contiene a los componentes, pero el gestor de disposición decide su ubicación exacta en la pantalla. Cada contenedor tiene un gestor de disposición que usa un esquema por defecto si es que no establecemos alguno explícitamente. El esquema por defecto es diferente para los diferentes contenedores: por ejemplo, el panel contenedor de un JFrame tiene asociado por defecto un BorderLayout mientras que un JPanel usa por defecto un FlowLayout.

El método setLayout se utiliza sobre el panel contenedor para establecer el gestor de disposición que se pretende usar. El gestor de disposición es en sí mismo un objeto, de modo que creamos una instancia de BorderLayout y se la pasamos al método setLayout.

Cuando agregamos un componente en un contenedor con un BorderLayout, usamos un método add diferente, que tiene un segundo parámetro. El valor del segundo parámetro es una de las constantes públicas NORTH, SOUTH, EAST, WEST o CENTER, que están definidas en la clase BorderLayout.

```
Container panelContenedor = ventana. getContentPane();
panelContenedor.setLayout(new BorderLayout());
etiquetaNombreDeArchivo = new JLabel();
panelConenedor.add(etiquetaNombreDeArchivo,BorderLayout.NORTH);
panelDelmagen = new PanelDelmagen () ;
panelContenedor.add(panelDelmagen, BorderLayout.CENTER);
etiquetaEstado = new JLabel( "Versión 1.0");
panelContenedor.add(etiquetaEstado, BorderLayout.SOUTH);
```

11.5.6 Diálogos

Una de las principales características de un diálogo es si es *modal* o no. Un diálogo modal bloquea todas las interacciones con las restantes partes de una aplicación hasta que se cierre. Esto obliga al usuario a que trate primero con el diálogo. Los diálogos no modales permiten la interacción con otras ventanas mientras están visibles.

Los diálogos se pueden implementar de manera similar a nuestro JFrame principal, aunque para mostrar la ventana usan con frecuencia la clase Jdialog. Sin embargo, para los diálogos modales con una estructura estándar, existen algunos métodos convenientes en la clase JOptionPane que facilitan mucho el trabajo de mostrar estos tipos de diálogos. JOptionPane tiene, entre otras cosas, métodos estáticos para mostrar tres tipos estándar de diálogos que son:

- *Diálogo de mensaje:* es un diálogo que muestra un mensaje y que tiene un botón OK para cerrar el diálogo.
- *Diálogo de confirmación:* este diálogo, generalmente, permite hacer preguntas al usuario y posee botones que el usuario puede utilizar para responder, por ejemplo: *Sí, No* y *Cancelar.*
- *Diálogo de entrada:* este diálogo incluye un campo de texto para que el usuario escriba algún texto.

11.7.1 Botones

Hasta ahora, nuestra ventana usa un BorderLayout, en donde la zona WEST está vacía. Podemos usar esta zona para agregar nuestros botones de la barra de herramientas. Sin embargo, hay un pequeño problema: la zona WEST de un BorderLayout puede contener sólo un componente, pero en este caso, tenemos dos botones. La solución es simple. Agregamos un Jpanel en el área WEST de la ventana (como ya sabemos, un Jpanel es un contenedor) y luego pegamos los dos botones dentro de él. Un Jpanel usa por defecto un FlowLayout y un FlowLayout dispone los componentes horizontalmente. Cuando se crea un GridLayout, los parámetros del constructor determinan cuántas filas y columnas queremos que tenga. Un valor cero tiene un significado especial, es interpretado como «tantas filas y columnas como sea necesario». Por lo tanto, podemos crear un GridLayout con una sola columna usando 0 como el número de filas y 1 como el número de columnas. Luego, podemos aplicar este GridLayout en nuestro Jpanel usando el método setLayout del panel, inmediatamente después de crearlo. En el esquema BorderLayout, un contenedor (en este caso, nuestra barra de herramientas Jpanel) siempre cubre el área completa en que está

ubicado (el área WEST en nuestra ventana) y un GridLayout siempre modifica el tamaño de sus componentes hasta rellenar la totalidad del contenedor.

Terminamos teniendo un panel GridLayout dentro de un panel FlowLayout dentro de un BorderLayout. Las construcciones de este estilo son muy comunes. Frecuentemente deberá anidar varios contenedores dentro de otros contenedores para crear exactamente lo que desea ver.

Todos los componentes Swing cuentan con un método setEnabled(boolean) que habilita o deshabilita el componente. Los componentes inhabilitados se muestran generalmente agrisados y no reaccionan.

11.7.2 Bordes

Se pueden usar bordes para agrupar componentes o sólo para agregar espacio entre ellos. Cada componente Swing puede tener un borde. Algunos gestores de disposición también aceptan parámetros en el constructor que definen sus espacios y luego, el gestor de disposición se encarga de crear el espacio requerido entre los componentes.

Los bordes más usados son BevelBorder(borde biselado de dos líneas), CompoundBorder (borde compuesto de dos bordes anidados), EmptyBorder(borde vacío y transparente que ocupa espacio pero no pinta nada), EtchedBorder (un borde simple con aspecto de grabado hacia dentro o hacia fuera) y TitledBorder(un borde arbitrario con una cadena de título con una posición y justificación especificadas)

Podemos hacer tres cosas para mejorar el aspecto de nuestra GUI:

- agregar espacio alrededor de la parte exterior de la ventana;
- agregar espacio entre los componentes de la ventana y
- agregar una línea alrededor de la imagen.

La llamada al método setBorder del panel contenedor con el parámetro EmptyBorder agrega espacio alrededor del borde exterior de la ventana. Observe que ahora convertimos el panelContenedor en un JPanel pues el supertipo Container no posee el método setBorder.