Tema 9. Interfaces gráficas de usuario: GUI

Principales conceptos que se abordan en este tema

Hasta ahora, en el libro nos hemos concentrado en escribir aplicaciones con interfaces basadas en texto. La razón no es que estas interfaces ofrezcan, en principio, ninguna gran ventaja, la única ventaja que tienen es que son más fáciles de crear.

Además, no queríamos distraer demasiado la atención de las cuestiones más importantes del desarrollo software, en estas primeras etapas de aprendizaje de la programación orientación a objetos. Hemos preferido concentrarnos en problemas como la interacción y estructura de los objetos, el diseño de clases y la calidad del código.

Las **interfaces gráficas de usuario** (a partir de ahora **GUI**, del inglés *Graphical User Interface*) también se construyen a partir de objetos que interactúan, pero tienen una estructura muy especializada, y hemos preferido evitar presentarlas antes de analizar las estructuras de los objetos en términos más generales. Sin embargo, ahora sí que estamos listos para echar un vistazo a las técnicas de construcción de interfaces GUI.

Las interfaces GUI proporcionan a nuestras aplicaciones una interfaz compuesta de ventanas, menús, botones y otros componentes gráficos. Hacen que las aplicaciones se parezcan mucho más a la “típica" aplicación que la mayoría de las personas utilizan hoy día.

Observe que nos topamos de nuevo aquí con el doble significado de la palabra interfaz. Las interfaces de las que estamos hablando ahora no son ni las interfaces de las clases ni la estructura `interface` de Java. Ahora estamos hablando de *interfaces* de usuario - la parte de una aplicación que es visible en pantalla para que el usuario pueda interaccionar con ella.

Número de sesiones

Se estiman un total de 20 horas.

9.1 Componentes, diseño y tratamiento de sucesos

Los detalles implicados en la creación de interfaces GUI son muy prolijos. En este libro no nos será posible cubrir todos los detalles de todas las posibles cosas que pueden hacerse con esas interfaces, pero sí que expondremos los principios generales e incluiremos un buen número de ejemplos.

Toda la programación de interfaz GUI en Java se realiza utilizando librerías estándar de clases dedicadas. Una vez que comprendamos los principios, nos resultará fácil encontrar todos los detalles necesarios, trabajando con la documentación de la librería estándar.

Los principios que necesitamos comprender pueden dividirse en tres áreas temáticas:

- ¿Qué tipo de elementos podemos mostrar en pantalla?
- ¿Cómo colocamos esos elementos?
- ¿Cómo podemos reaccionar a la entrada del usuario?

Estas cuestiones se corresponden con tres conceptos clave: ***componentes, diseño gráfico y tratamiento de sucesos.***

Los ***componentes*** son las partes individuales con las que se construye una GUI. Son cosas como botones, menús, elementos de menú, casillas de verificación, barras de desplazamiento, campos de texto, etc. La librería Java contiene un buen número de componentes prefabricados, y además podemos escribir nuestros propios componentes. Tendremos que aprender cuáles son los componentes más importantes, cómo crearlos y cómo hacer que tengan el aspecto que deseamos.

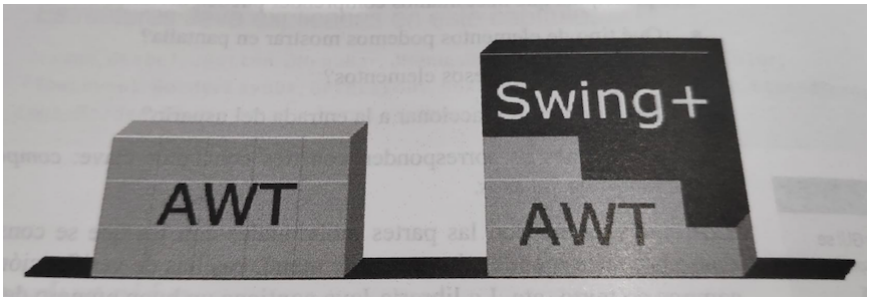
El ***diseño gráfico*** se ocupa de la cuestión de cómo disponer los componentes en pantalla. Los sistemas GUI mas antiguos y primitivos solucionaban este tema mediante coordenadas bidimensionales: el programador especificaba coordenadas x e y (en píxeles) para indicar la posición y el tamaño de cada componente. En los sistemas GUI mas mordernos, sin embargo, esto es demasiado simplista. Tenemos que tener en cuenta las diferentes resoluciones de pantalla, los distintos tipos de fuente, el hecho de que los usuarios pueden cambiar el tamaño de las ventanas y muchos aspectos que hacen que el diseño gráfico de la GUI resulte más difícil. La solución será un esquema en el que podemos especificar ese diseño en términos más generales. Por ejemplo, podemos especificar que un determinado componente debe estar debajo de otro o que el tamaño de un cierto componente debe ser reducido si se redimensiona la ventana, mientas que tal otro componente debe tener siempre un tamaño fijo. Más adelante veremos que esta tarea se lleva a cabo utilizando ***gestores de diseño gráfico***.

El tratamiento de sucesos hace referencia a la técnica que utilizaremos para tratar con la entrada del usuario. Una vez que hayamos creado nuestros componentes y los hayamos colocado en pantalla, también tendremos que asegurarnos de que ocurra algo cuando el usuario haga clic en un botón. El modelo utilizado por la librería Java para resolver este tema está basado en sucesos: si un usuario activa un componente (por ejemplo, hace clic en un botón o selecciona un elemento de menú) el sistema generará un suceso. Nuestra aplicación puede entonces recibir una notificación del suceso (haciendo que sea invocado uno de sus métodos), con lo que podremos llevar a cabo la acción apropiada.

9.2 AWT y Swing

Java tiene dos librerías GUI. La mas antigua se denomina AWT (*Abstract Window Toolkit*) y fue introducida como parte de la API Java original. Mas tarde se añadió a Java una librería GUI muy mejorada, denominada *Swing*.

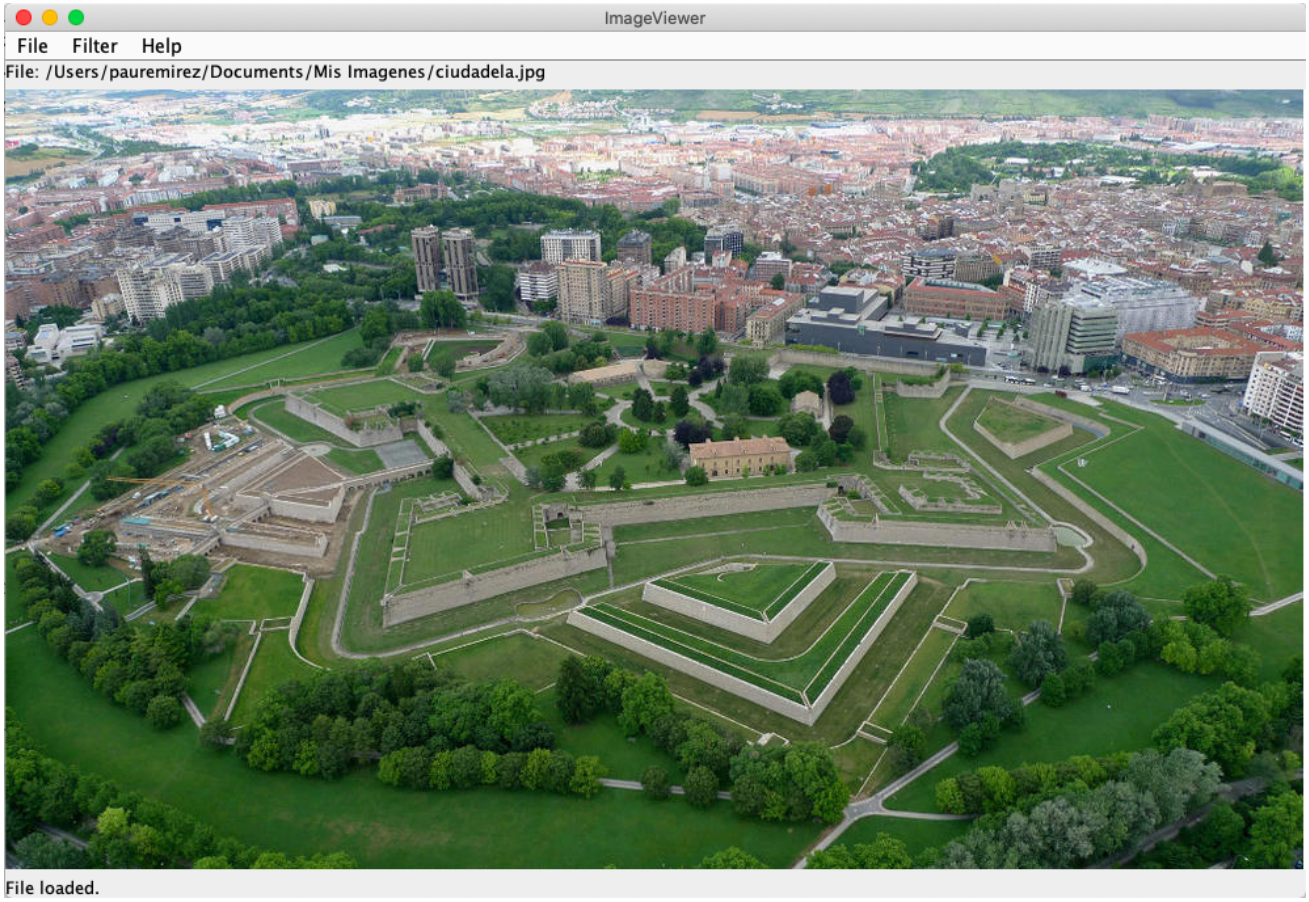
Swing hace uso de algunas de las clases de AWT, sustituye algunas de las clases de AWT por su propia versión y añade muchas nuevas clases.



Cuando existen clases equivalentes en AWT y Swing, las versiones Swing se han identificado añadiendo la letra mayúscula J al principio del nombre de la clase. Por ejemplo, en la documentación podrá encontrar clases denominadas *Button* y *JButton*, *Frame* y *JFrame*, *Menu* y *JMenu*, etc. Las clases que comienzan con J son las versiones Swing; esas son las que usaremos y no deben mezclarse las dos versiones en una misma aplicación.

9.3 El ejemplo ImageViewer

Como siempre, explicaremos los conceptos nuevos analizando un ejemplo. La aplicación que construiremos en este capítulo es un visualizador de imágenes (Figura 11.2). Es un programa que puede abrir y mostrar archivos de imagen en formatos JPEG y PNG, realizar algunas transformaciones en las imágenes y guardarlas en disco.



Como parte de esta tarea, utilizaremos nuestra propia clase para representar una imagen mientras se encuentra en memoria, implementaremos varios filtros para modificar la apariencia de la imagen y emplearemos componentes Swing para construir una interfaz de usuario. Mientras hacemos esto, centraremos las explicaciones en los aspectos del programa relativos a la GUI.

Si tiene curiosidad por lo que vamos a construir, puede abrir y probar el proyecto imageviewer1-0, que es la versión mostrada anteriormente; pruebe a crear un objeto ImageViewer. Vamos a comenzar poco a poco, empezando con algo mucho mas simple y progresando paso a paso hacia la aplicación final.

9.3.1 Primeros experimentos: creación de un marco

Casi todo lo que podemos ver en una GUI está contenido en una ventana de nivel superior. Una ventana de nivel superior es aquella que se encuentra bajo control del gestor de ventanas del sistema operativo y que normalmente puede moverse, redimensionarse, minimizarse y maximizarse de forma independiente.

Java denomina a estas ventanas de nivel superior **marcos**. En Swing, se representan mediante una clase llamada **JFrame**.

Para conseguir tener una interfaz GUI en pantalla, lo primero que tenemos que hacer es crear y visualizar un marco.

Ejercicio 9.1

Abra el siguiente proyecto. Cree una instancia de la clase ImageViewer. Redimensiona el marco resultante (hágalo más grande). ¿Qué es lo que puede observar acerca de la posición del texto dentro del marco?

Mostrar retroalimentación

Se mantiene aunque se redimensione el marco.

Ahora vamos a analizar con cierto detalle la clase Image Viewer mostrada en el código del ejercicio anterior.

Las tres primeras líneas de esa clase son instrucciones para la importación de todas las clases de los paquetes *java.awt*, *java.awt.event* y *javax.swing*. Necesitaremos muchas de las clases de estos paquetes para todas las aplicaciones Swing que creemos, por lo que en nuestros programas GUI siempre importaremos los tres paquetes completos.

Si examinamos el resto de la clase, vemos rápidamente que todo lo interesante se encuentra dentro del método *makeframe*. Este método se encarga de construir la GUI. El constructor de la clase contiene únicamente una llamada a este método. Hemos hecho esto para que todo el código de construcción de la GUI se encuentre en un lugar bien definido y sea fácil de encontrar posteriormente (¡cohesión!). Haremos esto mismo en todos nuestros ejemplos de interfaces GUI.

La clase tiene una variable de instancia de tipo JFrame. Esta variable se emplea para almacenar el marco que el visualizador de imágenes quiere mostrar en pantalla. Examinemos ahora más detenidamente el método makeFrame.

La primera línea de este método es *frame - new JFrame("ImageViewer");*

Esta instrucción crea un nuevo marco y lo almacena en nuestra variable de instancia para su uso posterior.

Ejercicio 9.2

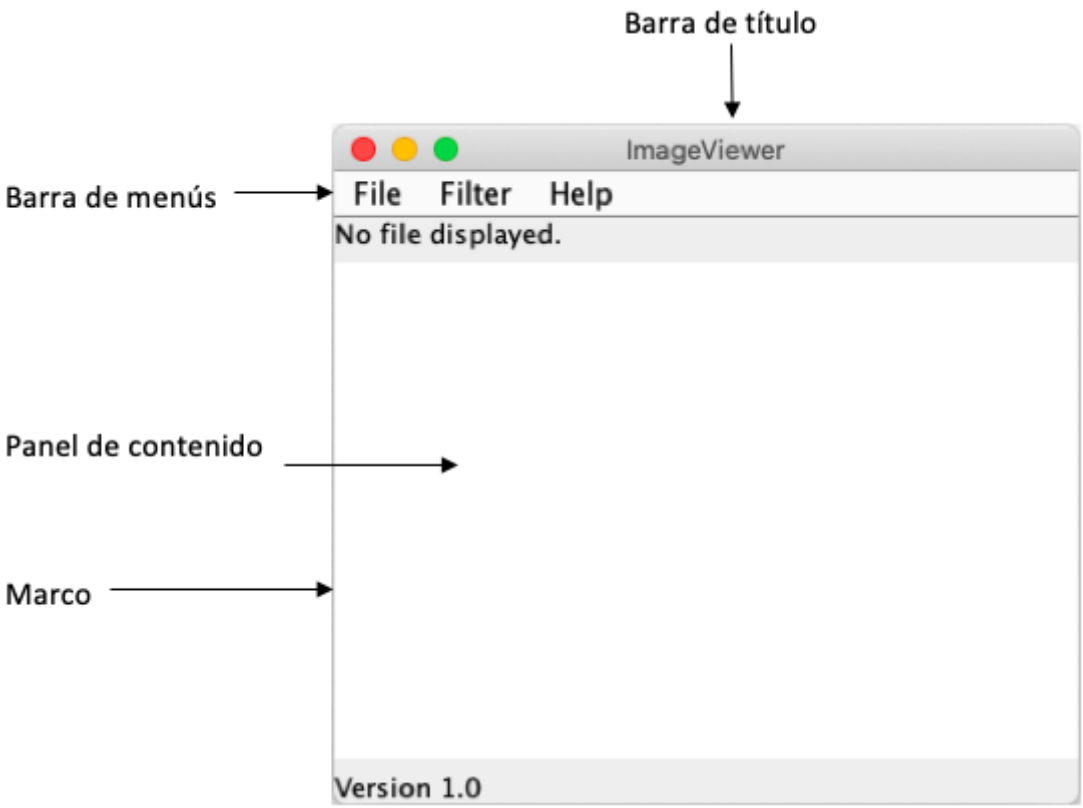
Localice la documentación de la clase JFrame, ¿Cuál es el propósito del parámetro "ImageViewer" que hemos utilizado en la llamada anterior al constructor?

Mostrar retroalimentación

Proporciona un título al Frame creado. Es este caso el título es "ImageViewer".

Un marco está compuesto por tres partes: la barra de titulo, una barra de menú opcional y el panel de contenido. La apariencia exacta de la barra de título depende del sistema operativo subyacente. Normalmente, contiene el título de la ventana y unos cuantos controles de ventana.

La barra de menú y el panel de contenido están bajo el control de la aplicación. Para crear una GUI, podemos añadir algunos componentes a ambas partes del marco. Vamos a concentrarnos primero en el panel de contenido.



9.3.2 Adición de componentes simples

Inmediatamente después de la creación del JFrame, el marco será invisible y su panel de contenido estará vacío. Vamos a continuar añadiendo una etiqueta al panel de contenido:

```
Container contentPane =  frame.getContentPane();
JLabel label =  new JLabel("I am a label.");
contentPane.add(label);
```

La primera linea extrae una referencia al panel de contenido del marco, Siempre vamos a tener que hacer esto; los componentes de una GUI se añaden a un marco agregándolos al panel de contenido del marco.

El propio panel de contenido es de tipo Container. Un contenedor es un componente Swing que puede almacenar grupos arbitrarios de otros componentes -más o menos de la misma forma que un Arraylist puede almacenar una colección arbitraria de objetos. Hablaremos más detalladamente de los contenedores más adelante.

Después creamos un componente etiqueta (tipo JLabel) y lo añadimos al panel de contenido.
Una etiqueta es un componente que puede mostrar texto y/o una imagen.
Para terminar, tenemos las dos lineas

```
frame.pack();
frame.setVisible(true);
```

La primera línea hace que el marco disponga apropiadamente los componentes que contiene y que se dote asimismo del tamaño adecuado. Siempre vamos a tener que invocar el método *pack* sobre el marco después de añadir o redimensionar componentes.

Finalmente, la última línea hace que el marco sea visible en pantalla. Siempre vamos a comenzar con el marco invisible, para poder disponer todos los componentes en su interior sin que el proceso de construcción sea visible en pantalla.

Después, cuando el marco esté ya construido, podemos mostrarlo en un estado terminado.

Ejercicio 9.3

Otro componente Swing muy utilizado es el botón (tipo JButton). Sustituya la etiqueta del ejemplo anterior por un botón.

Mostrar retroalimentación

Container contentPane = frame.getContentPane(); **JButton button = new JButton();**contentPane.add(**button**):

Ejercicio 9.4

Qué sucede cuando añadimos dos etiquetas (o dos botones) al panel de contenido? ¿Puede explicar lo que observe? Experimente redimensionando el marco.

Mostrar retroalimentación

Container contentPane = frame.getContentPane();

JLabel label1 = new JLabel("I am the label 1."); JLabel label2 = new JLabel("I am the label 2.");

contentPane.add(label1):contentPane.add(label2):

* Se muestra el último elemento añadido al contenedor. El último elemento "machaca" al anterior.

9.3.3 Una estructura alternativa

Hemos elegido desarrollar nuestra aplicación creando un objeto *JFrame* como un atributo o *ImageViewer* y rellenándolo con componentes GUI adicionales, que se crean fuera del objeto marco. Una estructura alternativa a esta consistiría en definir ImageViewer como subclase de *JFrame* y rellenarlo internamente. Este estilo también es bastante común sería el equivalente en este estilo al código que estamos utilizando como ejemplo. Esta versión está disponible en el siguiente proyecto. Aunque merece la pena estar familiarizado con ambos estilos, ninguno de los dos es necesariamente mejor que el otro, y nosotros vamos a continuar con nuestra versión original en el resto del tema.

9.3.4 Adición de menús

Nuestro siguiente paso para la construcción de una GUI consiste en añadir menús v elementos de menú. Esto es conceptualmente muy sencillo, pero hay un detalle que tiene sus complicaciones: ¿cómo nos la arreglamos para reaccionar a las acciones del usuario, como por ejemplo la selección de un elemento de menú? Hablaremos de esto más adelante.

Primero, vamos a crear los menús. Son tres las clases implicadas:

- **JMenuBar** Un objeto de esta clase representa una barra de menús que puede mostrarse debajo de la barra de titulo ubicada en la parte superior de la ventana.
- **JMenu** Los objetos de esta clase representan un único menú (como por ejemplo los habituales menús *File*, *Edit* o *Help -Archivo, Edición o Ayuda*-). Los menús suelen estar contenidos en una barra de menús. También pueden aparecer en los menús emergentes, pero por el momento no vamos a hacer uso de esa posibilidad.
- **JMenuItem** Los objetos de esta clase representan un único elemento de menú dentro de un menú (como por ejemplo *Open* o *Save-Abrir* o *Guardar*-).

Para nuestro visualizador de imágenes, vamos a crear una barra de menús y varios menús y elementos de menú.

La clase *JFrame* tiene un método denominado *setJMenuBar*. Podemos crear una barra de menú y utilizar este método para asociar la barra de menús al marco:

```
JMenuBar menubar - new JMenuBar();
frame.setJMenuBar(menubar);
```

Ahora estamos listos para crear un menú y añadirlo a la barra de menús:

```
JMenu fileMenu - new JMenu("File");
menubar.add(fileMenu);
```

Estas dos líneas crean un menú etiquetado como *File* y lo insertan en la barra de menús. Por último, podemos añadir elementos de menú al menú. Las siguientes líneas añaden al menú *File* dos elementos, etiquetados como *Open* y *Quit*:

```
JMenuItem openItem - new JMenuItem("Open");
fileMenu.add (openItem);
JMenuItem quitItem - new JMenuItem("Quit");
fileMenu.add (quitItem);
```

Ejercicio 9.5

Añada el menú y los elementos de menú que acabamos de exponer a su proyecto del visualizador de imágenes. ¿Qué sucede cuando se selecciona un elemento de menú?

Mostrar retroalimentación

(Proyecto con la implementación)

Al seleccionar el menú se despliegan los items del menú.

Ejercicio 9.6

Añada otro menú denominado ***Help*** que contenga un elemento de menú llamado ***About ImageViewer***.

Mostrar retroalimentación

Enlace la proyecto con la solución.

Hasta ahora, hemos conseguido completar tan solo la mitad de la tarea: podemos crear y visualizar menús, pero nos falta la otra mitad -todavía no sucede nada cuando un usuario selecciona un menú. Ahora tenemos que añadir código para reaccionar a las selecciones de menú.

9.3.5 Tratamiento de sucesos

Swing utiliza un modelo muy flexible para manejar la entrada de la GUI: un modelo de *tratamiento de sucesos* con escuchas de sucesos.

El propio entorno de trabajo Swing y algunos de sus componentes generan sucesos cuando sucede algo en lo que puedan estar interesados otros objetos. Existen diferentes tipos de sucesos, provocados por distintos tipos de acciones. Cuando se hace clic en un botón o se selecciona un elemento de menú, el componente genera un suceso *ActíonEvent*. Cuando se hace clic con el ratón o se mueve este se genera un suceso *MouseEvent*. Cuando se cierra o se reduce a un icono un marco, se genera un suceso *WindowEvent*. Existen asimismo muchos otros tipos de sucesos.

Cualquiera de nuestros objetos puede convertirse en escucha de sucesos, es decir, puede ponerse a la escucha de cualquiera de estos sucesos. Cuando se pone a la escucha, recibirá una notificación por cada uno de los sucesos que esté escuchando. Un objeto se convierte en un escucha de sucesos implementando una de las diversas interfaces de escucha existentes. Si implementa la interfaz correcta, puede registrarse ante un componente del cual quiera quedarse a la escucha.

Veamos un ejemplo. Un elemento de menú (clase *JMenuItem*) genera un suceso *ActionEvent* al ser activado por un usuario. Los objetos que quieran escuchar esos sucesos tienen que implementar la interfaz *ActionListener* del paquete *java.awt.event*.

Hay dos estilos alternativos para la implementación de escuchas de sucesos: o bien un único objeto escucha los sucesos de muchas fuentes de sucesos distintas, o bien a cada una de las fuentes de sucesos se le asigna su propio escucha.

9.3.6 Recepción centralizada de sucesos

Para hacer que nuestro objeto ImageViewer sea el único escucha de todos los sucesos del menú, tenemos que hacer tres cosas:

1- Debemos declarar en la cabecera de la clase que implementa la interfaz ActionListener.

2- Tenemos que implementar un método con la signatura

public void actionPerformed (ActionEvent e)

Este es el único método declarado en la interfaz ***Action Listener***.

3- Tenemos que invocar el método ***addActionListener*** del elemento de menú, para registrar como escucha el objeto ***ImageViewer***.

Los puntos 1 y 2 (implementar la interfaz y definir su método) garantizan que nuestro objeto sea un subtipo de ActionListener. El punto 3 registra nuestro propio objeto como escucha de los elementos de menú.

El siguiente código muestra cómo hacer esto en nuestro contexto:

```
public class ImageViewer implements ActionListener {

    // Se omiten los campos y el constructor.

    public void actionPerformed(ActionEvent event) {

        System.out.println("Menu item: " +
        event.getActionCommand());

    }

    /**
    * Crear el marco Swing y su contenido.
    */

    private void makeFrame(){

        frame = new JFrame("ImageViewer");
        makeMenuBar(frame);


        // Se omite parte del código de construcción de la
        GUI.

    }

    /** Crear la barra de menús del marco principal.
    * @param frame El marco al que hay que añadir la barra de
    menus
    */
```

```
    private void makeMenuBar(JFrame frame) {

        JMenuBar menubar = new JMenuBar();
        frame.setJMenuBar(menubar);
        // crear el menú File
        JMenu fileMenu = new JMenu("File");
        menubar.add(fileMenu);


        JMenuItem openItem = new JMenuItem("Open");
        openItem.addActionListener(this);
        fileMenu.add(openItem);


        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(this);
        fileMenu.add(quitItem);

    }

}
```

En el ejemplo de código anterior, fíjese especialmente en las líneas:

```
JMenuItem openItem - new JMenuItem("Open");
open Item.addActionListener (this);
```

En ellas, se crea un elemento de menú y se registra el objeto actual (el propio objeto *ImageViewer*) como un escucha de acción, pasando this como parámetro al método *addActionListener*.

El efecto de registrar nuestro objeto como escucha ante el elemento de menú es que el elemento de menú invocará nuestro propio método ***actionPerformed*** cada vez que se active el elemento. Cuando se invoque nuestro método, el elemento de menú le pasará un parámetro de tipo ***ActionEvent*** que proporcionará algunos detalles acerca del suceso que ha tenido lugar. Estos detalles incluyen el instante exacto del suceso, el estado de las teclas modificadoras (las teclas Mayús, Ctrl y meta), una "cadena de comandos" y otros detalles.

La cadena de comandos es una cadena que identifica de alguna manera el componente que ha provocado el suceso. Para los elementos de menú, esta será, de forma predeterminada, el texto de la etiqueta del elemento.

En nuestro ejemplo, registramos el mismo objeto de acción para ambos elementos de menú. Esto significa que ambos elementos de menú invocarán el mismo método ***actionPerformed*** cuando sean activados.

En el método ***actionPerformed***, simplemente imprimimos la cadena de comando del elemento, para demostrar que este esquema funciona. Obviamente, podemos ahora añadir código para gestionar apropiadamente la invocación del menú.

Este ejemplo de código, tal como lo hemos explicado hasta el momento, está disponible en este enlace.

Ejercicio 9.7

Implemente el código de gestión de menú que acabamos de explicar en el proyecto base que estamos utilizando desde el principio del tema. Alternativamente, abra el último proyecto propuesto y examine cuidadosamente el código fuente. Describa por escrito y detalladamente la secuencia de sucesos que se produce al activar el elemento de menú *Quit*.

Ejercicio 9.8

Añada otro elemento de menú denominado Save.

Mostrar retroalimentación

Añadiremos el siguiente código dentro del método *makeMenuBar (JFrame frame):*

```
JMenuItem saveItem = new JMenuItem("Save");
saveItem.addActionListener(this); fileMenu.add(saveItem);
```

Ejercicio 9.9

Añada tres métodos privados a su clase, denominados openFile, saveFile y quit.Modifique el método actionPerformed de modo que invoque el método correspondiente cuando se active un elemento de menú.

Mostrar retroalimentación

Proyecto con la solución

Ejercicio 9.10

Si ya hecho el Ejercicio 9.6 (añadir un menú Help), asegúrese de que también se gestioneapropiadamente su elemento de menú.

Mostrar retroalimentación

Proyecto con la solución

Podemos comprobar que este enfoque funciona, e implementar a continuación métodos para gestionar los elementos de menú, con el fin de llevar a cabo las diversas tareas del programa.

Existe, sin embargo, otro aspecto que debemos investigar: la solución actual no es muy elegante en términos de ampliabilidad y mantenibilidad.

Examine el código que ha escrito en el método ***actionPerformed*** para resolver el Ejercicio 9.9. Hay varios problemas:

- Probablemente ha utilizado una instrucción if y el método ***getActionCommand*** para averiguar qué elemento fue activado. Por ejemplo, se podría escribir

  ```
  if(event.getActionCommand().equals("Open"))
  ```

  Depender de la cadena de caracteres que forma la etiqueta del elemento para realizar la función no es una buena idea. ¿Qué pasaría si ahora tradujéramos la interfaz a otro idioma? Con solo cambiar el texto del elemento del menú, el programa dejaría de funcionar (o tendríamos que localizar todos los lugares en el código donde se ha utilizado esa cadena de caracteres y cambiarla -un procedimiento tedioso y proclive a errores).

- Disponer de un método de despacho central (como nuestro ***actionPerformed***) no proporciona una estructura elegante. Básicamente, lo que hacemos es que todos los elementos llamen a un mismo método, tan solo para a continuación escribir un tedioso código en este método que invoque diferentes métodos para cada elemento. Esto es muy molesto en términos de mantenimiento (para cada elemento de menú adicional, tendríamos que añadir una nueva instrucción if en ***actionPerformed***); también es un desperdicio de esfuerzos. Sería mucho más conveniente que pudiéramos hacer que cada elemento de menú invocara directamente un método distinto.

9.3.7 Clases internas

Para resolver el problema que acabamos de mencionar relativo al despacho centralizado de sucesos, utilizaremos una nueva estructura que aun no hemos presentado: las clases internas. Las clases internas son clases que se declaran textualmente dentro de otra clase:

```
class ClaseCircundante {

    class ClaseInterna {

    ...
    }

}
```

Las instancias de la clase interna están asociadas a instancias de la clase circundante; solo pueden existir junto con una instancia circundante, y existen conceptualmente dentro de la instancia circundante. Un detalle interesante es que las instrucciones contenidas en los métodos de la clase interna pueden ver los campos y métodos privados de la clase circundante y acceder a ellos. Por tanto, existe, obviamente, un acoplamiento muy estrecho entre las dos clases. La clase interna se considera una parte de la clase circundante, de la misma forma que cualquiera de los métodos contenidos en la clase circundante.

Ahora podemos utilizar esta estructura para definir una clase separada de escucha de acción para cada elemento de menú que queramos escuchar. Como se trata de clases separadas, cada una puede disponer de un método ***actionPerformed*** distinto, de modo que cada uno de esos métodos gestionará la activación de un único elemento. La estructura es la siguiente:

```
class ImageViewer {

    class OpenActionListener implements ActionListener {

        public void actionPerformed(ActionEvent event) {

            //realizar la acción correspondiente a open

        }

    }
    class QuitActionListener implements ActionListener {

        public void actionPerformed (ActionEvent event) {

            // realizar la acción correspondiente a quit

        }

    }

}
```

Recomendación

(Como guía de estilo, solemos escribir las clases internas al final de la clase circundante, después de los métodos.)

Una vez que hemos hecho esto, podemos ahora crear instancias de estas clases internas exactamente de la misma forma que lo haríamos para cualquier otra clase. Observe también que ***ImageViewer*** no implementa ya ***ActionListener*** (eliminamos su método ***actionPerformed***), pero las dos clases internas sí lo hacen. Esto nos permite ahora utilizar instancias de las clases internas como escuchas de acción para los elementos de menú.

```
JMenuItem openItem - new JMenuItem("0pen"):
openItem.addActionListener(new OpenActionListener());
...
JMenuItem quitItem - new JMenuItem("Quit"); SN
quitItem.addActionListener(new QuitAction Listener());
```

En resumen, en lugar de hacer que el objeto visualizador de imágenes escuche todos los sucesos de acción, hemos creado objetos escucha separados para cada posible suceso, dedicando cada uno de ellos a escuchar un único tipo de suceso. Puesto que todo escucha tiene su propio método ***actionPerformed***, ahora podemos escribir en esos métodos el código específico para el tratamiento del suceso que deseemos. Asimismo, puesto que las clases escucha se encuentran dentro del ámbito de la clase circundante (pueden acceder a los campos
y métodos privados de la clase circundante), pueden hacer un uso completo de la clase circundante en la implementación de los métodos ***actionPerformed***.

Observe un par de características de estos objetos escucha:

- No nos preocupamos de almacenarlos en variables -por tanto, en la práctica, se trata de objetos anónimos. Solo los elementos de menú disponen de una referencia a los objetos escucha, para poder invocar sus métodos actionPerformed.
- Creamos un único objeto para cada una de las clases internas, ya que cada clase está altamente especializada para un elemento de menú concreto.

Estas características nos llevarán a explorar otra característica adicional de Java en la siguiente sección.

Ejercicio 9.11

Implemente la gestión de los elementos de menu con clases internas, como hemos explicado aquó, en su propia versión del visualizador de imágenes.

Mostrar retroalimentación

Proyecto con la solución

En algunos casos, las clases internas pueden utilizarse con carácter general para mejorar la cohesión en proyectos de gran tamaño.

9.3.8 Clases internas anónimas

La solución al problema del despacho de acciones mediante clases internas es bastante buena, pero nos gustaría llevarla un paso más allá: podemos usar ***clases internas anónimas.*** El proyecto imageviewer0-3 muestra una implementación utilizando esta estructura.

Ejercicio 9.12

Abra el proyecto imageviewer0-3 y examínelo; es decir, pruébelo y lea su código fuente. No se preocupe si no entiende todos los aspectos, porque algunas nuevas características son precisamente el tema de esta sección. ¿Qué es lo que observa acerca del uso de clases internas a la hora de permitir que *ImageViewer* escuche los sucesos y los trate?

Ejercicio 9.13

Observará que al activar el elemento de menú Quit, salimos del programa. Examine cómo se hace esto. Busque la documentación de la librería para las clases y métodos implicados.

Uno de los aspectos fundamentales de los cambios incluidos en esta versión es la forma en que se configuran los escuchas de acción para que escuchen los sucesos de acción de los elementos de menú. El código relevante tiene el siguiente aspecto:

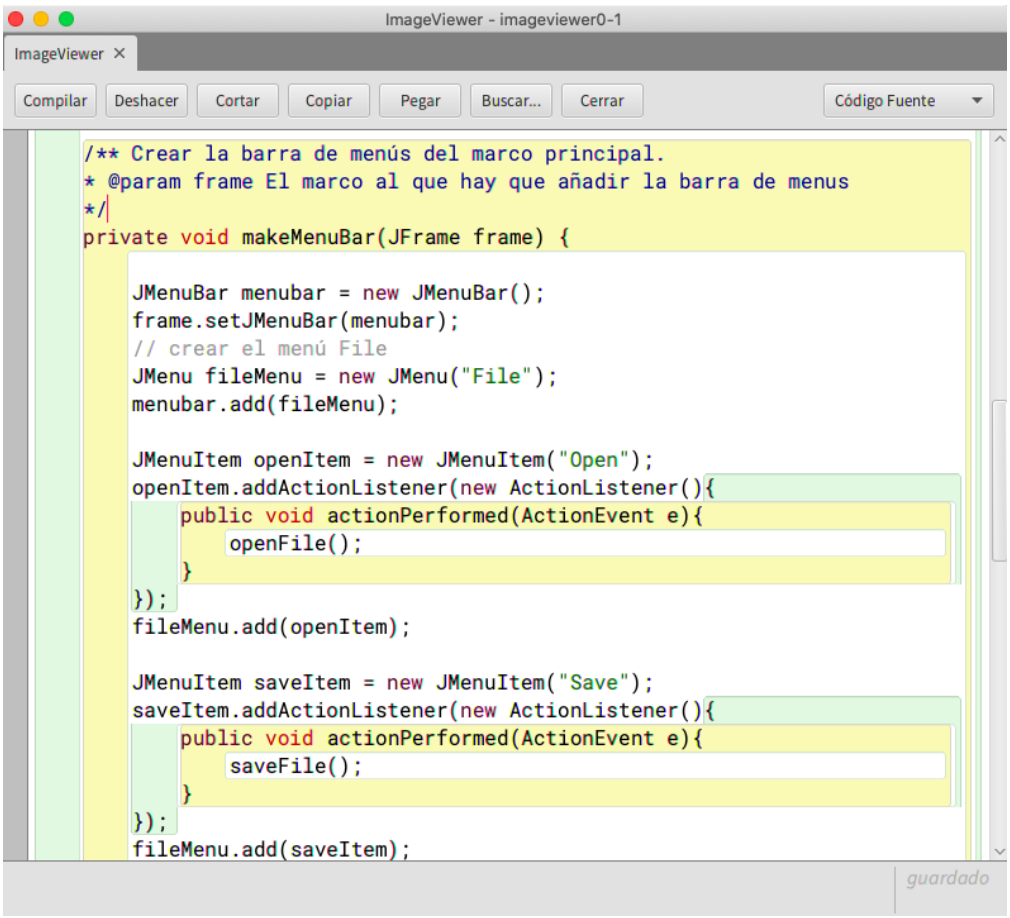
```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener()(

   public void actionPerformed(ActionEvent e) { openFile(): }

});
```

Este fragmento de código parece bastante misterioso cuando nos encontramos con él por primera vez, y probablemente tenga problemas en interpretarlo aunque haya comprendido todo lo que hemos explicado en el libro hasta ahora, Esta estructura es, probablemente, el ejemplo más confuso desde el punto de vista sintáctico que jamás se encontrará en el lenguaje Java. Pero no se preocupe, vamos a analizar detalladamente esa estructura.

Lo que podemos ver es una clase interna anónima. La idea de esta estructura está basada en las observaciones efectuadas en la versión anterior, donde vimos que solo utilizamos cada clase interna exactamente una vez para crear una única instancia sin nombre. Para esta situación las clases internas anónimas proporcionan un atajo sintáctico: nos permiten definir

una clase y crear una única instancia de esa clase, todo en un único paso. El efecto es idéntico a la version con clases internas anterior, con la diferencia de que no necesitamos definir clases con nombre separadas para los escuchas; asimismo, la definición del método escucha está situada más próxima al registro del escucha ante el elemento de menú.

El coloreado del ámbito en el editor de BlueJ nos proporciona algunas indicaciones que nos pueden ayudar a comprender esta estructura. El sombreado verde (gris más oscuro en la figura) indica una clase, el de color amarillo (gris mas claro en la figura) muestra una definición de método y el fondo blanco identifica un cuerpo de método. Podemos ver que el cuerpo del método makeMenuBar contiene, muy densamente empaquetadas, dos definiciones de clase (de aspecto extraño), cada una de las cuales tiene una única definición de método dentro de un cuerpo de corta longitud.



Cuando utilizamos una clase interna anónima, creamos una clase interna sin asignarla un nombre e inmediatamente creamos una única instancia de esa clase. En el código del escucha de acción anterior, esto se hace mediante el fragmento de código:

```
new ActionListener() {

    public void actionPerformed(ActionEvent e) ( openFile(); }

}
```

La manera de crear una clase interna anónima es mediante el nombre de un a menudo suele ser un clase abstracta o una interfaz -aquí *ActionListener*), seguido de un bloque que contiene una implementación para sus métodos abstractos. Esto tiene un aspecto inusual, porque no hay ningún otro caso en el que se permita crear directamente una instancia o de una clase abstracta o interfaz.

En este ejemplo, hemos creado un nuevo subtipo de ActionListener que implementa el método *actionPerformed*. Esta nueva clase no recibe ningún nombre. En lugar de ello, le anteponemos la palabra clave new para crear una única instancia de dicha clase.

En nuestro ejemplo, esta única instancia es un objeto escucha de acción (es de un subtipo de *ActionListener*). Se puede pasar al método ***addActionListener*** de un elemento de menú y este invocará el método ***openFile*** de su clase circundante al ser activado. Cada subtipo de *ActionListener* creado de esta forma representa una clase anónima distinta.

Al igual que las clases internas con nombre, las clases internas anónimas pueden acceder a los método, pueden acceder a las variables locales y parámetros de dicho método. Sin embargo, campos y métodos de su clase circundante. Además, puesto que están definidas dentro de un método, pueden acceder a las variables locales y parámetros de dicho método. Sin embargo, una regla importante es que las variables locales a las que se acceda de esta forma deben estar declaradas como variables ***final***.

Merece la pena recalcar algunas observaciones sobre las clases internas anónimas. En primer lugar, para nuestro problema concreto, resulta muy útil emplear clases internas anónimas. Nos permiten eliminar completamente el método ***actionPerformed*** central de nuestra clase *ImageViewer*. Y lo que hacemos en lugar de ello es crear un escucha de acción (clase y objeto) independiente y personalizado para cada elemento de menú. Este escucha de acción puede invocar directamente el método que implementa la correspondiente función.

Esta estructura tiene un alto grado de cohesión y ampliabilidad. Si necesitamos un elemento de menú adicional, nos basta con añadir el código para crear el elemento y su escucha, así como el método que gestiona su función. No hace falta ninguna enumeración en ningún método central.

En segundo lugar, la utilización de clases internas anónimas puede hacer que el código sea bastante difícil de leer. Se recomienda vivamente utilizarlas tan solo para clases muy cortas y para estructuras de código muy comunes.

En tercer lugar, a menudo utilizamos clases anónimas cuando solo se va a requerir una única instancia de la implementación -cuando las acciones asociadas con cada elemento de menú son características de ese elemento concreto. Además, todas las referencias a la instancia serán siempre a través de su supertipo. Ambas razones implican que existe una menor necesidad de dotar de un nombre a la nueva clase; de aquí que pueda ser anónima.

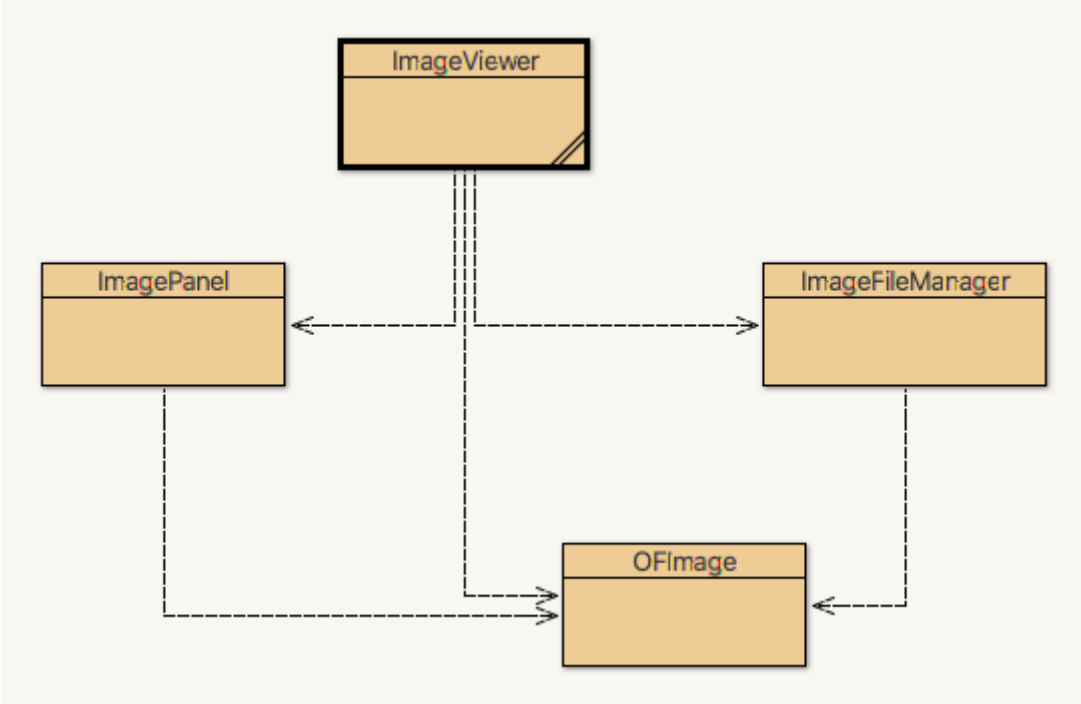
> Vamos a evitar emplear el método ***actionPerformed*** central y usaremos en su lugar **clases internas anónimas**.

9.4 ImageViewer 1.0: La primera versión completa

Ahora vamos a trabajar en la primera versión completa, una que realmente pueda llevar a cabo la tarea principal: mostrar algunas imágenes.

9.4.1 Clases de procesamiento de imágenes

Como paso intermedio para la solución vamos a investigar otra versión provisional: imageviewer0-4. Su estructura es la siguiente:



Como puede ver, hemos añadido tres nuevas clases: *OFImage, ImagePanel* y *ImageFileManager*.

OFImage es una clase que sirve para representar una imagen que queramos Visualizar y manipular. *ImageFileManager* es una clase auxiliar que proporciona métodos estáticos para leer un archivo de imagen (en formato JPEG o PNG) de disco y devolverlo en formato *OFImage* y luego guardar de nuevo en disco el objeto *OFImage*. *ImagePanel* es un componente Swing personalizado para mostrar la imagen en nuestra GUI.

Vamos a analizar brevemente los aspectos más importantes de cada una de estas clases con algo más de detalle. Sin embargo, no las vamos a explicar completamente -eso lo dejamos como ejercicio para el estudiante curioso.

La clase *OFImage* es nuestro propio formato personalizado para representar una imagen en memoria. Podemos pensar en *OFImage* como en una matriz bidimensional de píxeles. Cada uno de los píxeles puede tener un color. Utilizamos la clase estándar *Color* (del paquete *java.awt*) para representar el color de cada píxel. Eche un vistazo también a la documentación de la clase *Color*; la necesitaremos mas adelante.

OFImage está implementada como subclase de la clase estándar Java *BufferedImage* (de| paquete *java.awt.image*). *BufferedImage* nos proporciona la mayor parte de la funcionalidad que necesitamos (también representa una imagen como una matriz bidimensional), pero no dispone de métodos para configurar o consultar un píxel utilizando un objeto *Color* (utiliza diferentes formatos para esto que no queremos emplear). Por tanto, hemos definido nuestra propia subclase, que añade estos dos métodos.

Para este proyecto, podemos tratar *OFImage* como una clase de librería; no necesitamos, modificar dicha clase.

Los métodos más importantes de *OFImage* para nosotros son:

- *getPixel* y *setPixel* para leer y modificar píxeles de uno en uno.
- *getHeight* y *getWidth* para determinar el tamaño de la imagen.

La clase *ImageFileManager* ofrece tres métodos: uno para leer un archivo de imagen con hombre desde disco y devolverlo como un objeto *OFImage*, otro para escribir un archivo *OFImage* en disco y otro para abrir un cuadro de diálogo de selección de archivo, para que el usuario pueda seleccionar la imagen que desea abrir. Los métodos pueden leer archivos en los formatos JPEG y PNG estándar, y el método save escribirá los archivos en formato JPEG. Esto se lleva a cabo utilizando los método de E/S de imágenes estándar de Java, de la clase ImageIO (paquete *javax.imageio*).

La clase *ImagePanel* implementa un componente Swing personalizado para visualizar nuestra imagen. Los componentes Swing personalizados pueden crearse fácilmente escribiendo una subclase de un componente existente. Como componentes que son pueden insertarse en un contenedor Swing y mostrarse en nuestra GUI, al igual que cualquier otro componente Swing *ImagePanel* es una subclase de *JComponent*. El otro punto importante que hay que destacar aquí es que *ImagePanel* dispone de un método *setImage* que toma un objeto *OFImage* como parámetro, con el fin de visualizar cualquier objeto *OFImage* que queramos.

9.4.2 Adición de la imagen

Ahora que hemos preparado las clases para manejar las imágenes, resulta sencillo añadir la imagen a la interfaz de usuario. El siguiente código muestra las diferencias importantes con respecto a las versiones anteriores.

```
public class ImageViewer{

    private JFrame frame;
    private ImagePanel imagePanel;

    //Se omiten el constructor y el método para salir de la
    aplicación

    /**
    * Función Open: abre un selector de archivos para elegir
    * un nuevo archivo de imagen
    */

    private void openFile() {

        OFImage image = ImageFileManager.getImage();
        imagePanel.setImage(image);
        frame.pack();

    }

    /**
    * Crear el macro Swing y su contenido
    */

    private void makeFrame() {

        frame = new JFrame("ImageViewer");
        makeMenuBar(frame);
        Container contentPane = frame.getContentPane();
        imagePanel = new ImagePanel();
        contentPane.add(imagePanel);
        //terminada la construcción - colocar los componentes
        y mostrar
        frame.pack();
        frame.visible(true);

    }

    //Método makeMenuBar omitido

}
```

Al comparar este código con la versión anterior, observamos que solo hay dos pequeños cambios:

- En el método ***makeFrame***, ahora creamos y añadimos un componente ***ImagePanel*** en lugar de un ***JLabel***. Hacer esto no es más complicado que añadir la etiqueta. El objeto ImagePanel se almacena en un campo de instancia, para que podamos acceder a él de nuevo posteriomente.
- Hemos modificado nuestro método ***openFile*** para que abra y muestra un archivo de imagen. Utilizando nuestras clases para el procesamiento de imágenes, esta tarea también resulta sencilla. La clase ***ImageFileManager*** tiene un método para seleccionar y abrir una imagen, y el objeto ***ImagePanel*** dispone de un método para visualizar esa imagen. Un aspecto que hay que resaltar es que necesitamos invocar ***frame.pack()*** al final del método ***openFile***, ya que el tamaño de nuestro componente de imagen habrá cambiado. El método pack recalculará la colocación del marco y lo redibujará para que se gestione adecuadamente el cambio de tamaño.

Ejercicio 9.14

Abra y pruebe el proyecto imagewiewer0-4.

Ejercicio 9.15

¿Qué sucede cuando abrimos una imagen y luego redimensionamos el marco? ¿Qué sucede si redimensionamos primero el marco y luego abrimos una imagen?

Mostrar retroalimentación

- ¿Qué sucede cuando abrimos una imagen y luego redimensionamos el marco?
  - La imagen ocupa de inicio todo el marco. Al redimensionar el marco, la imagen no lo hace. Se mantiene en su tamaño original. El marco de amplía con un marco blanco.
- ¿Qué sucede si redimensionamos primero el marco y luego abrimos una imagen?
  - El marco se redimensiona al tamaño de la imagen

Con esta versión, hemos resuelto la tarea central: ahora podemos abrir un archivo de imagen almacenado en el disco y mostrarlo en pantalla. Sin embargo, antes de declarar a nuestro proyecto como finalizado, queremos añadir unas cuantas mejoras más:

- Queremos añadir dos etiquetas: una para mostrar el nombre del archivo de imagen en la parte superior y un texto de estado en la parte inferior.
- Queremos añadir un menú *Filter* que contenga algunos filtros para modificar la apariencia de la imagen.
- Queremos añadir un menú *Help* que contenga un elemento *About ImageViewer.* Al seleccionar este elemento de menú, debe mostrarse un cuadro de diálogo con el nombre de la aplicación, el número de versión e información acerca del autor.

9.4.3 Diseño gráfico

En primer lugar, trabajaremos en la tarea de añadir dos etiquetas de texto a nuestra interfaz una en la parte superior, que se utilizará para visualizar el nombre del archivo de la imagen qUe se esté mostrando, y otra en la parte inferior, que se empleará para diversos mensajes de estado.

La creación de estas etiquetas es sencilla -se trata simplemente de instancias *JLabel*. Lag almacenaremos en campos de instancia, para poder acceder posteriormente a ellas con el fm de modificar el texto que muestran. La única cuestión pendiente es cómo colocarlas en la pantalla

Un primer intento (simplista e incorrecto) podría ser así:

```
Container contentPane = frame.getContentPane( );

filenameLabel = new JLabel();
contentPane.add(filenameLabel);

imagePanel = new ImagePanel();
contentPane.add(imagePanel);

statusLabel = new JLabel("Version 1.0");
contentPane.add(statusLabel);
```

La idea aquí es muy simple: obtenemos el panel de contenido del marco y añadimos, uno tras otro, los tres componentes que queremos visualizar. El único problema es que no hemos especificado exactamente cómo queremos colocar esos tres componentes. Puede que queramos que aparezcan uno a continuación del otro, o uno debajo del otro o en cualquier otra posible disposición. Cómo no hemos especificado ningún diseño gráfico, el contenedor (el panel de contenido) recurrirá a un comportamiento predeterminado. Y resulta que esto no es lo que deseamos.

Swing utiliza gestores de diseño gráfico para colocar los componentes en una GUI. Cada contenedor que alberga componentes, como el panel de contenido, tiene un gestor de diseño gráfico asociado que se encarga de colocar los componentes dentro de ese contenedor.

Ejercicio 9.16

Continuando con su última versión del proyecto (imageviewer0-4), utilice el fragmento de código mostrado anteriormente para añadir las dos etiquetas. Pruebe esta solución. ¿Qué es lo que observa?

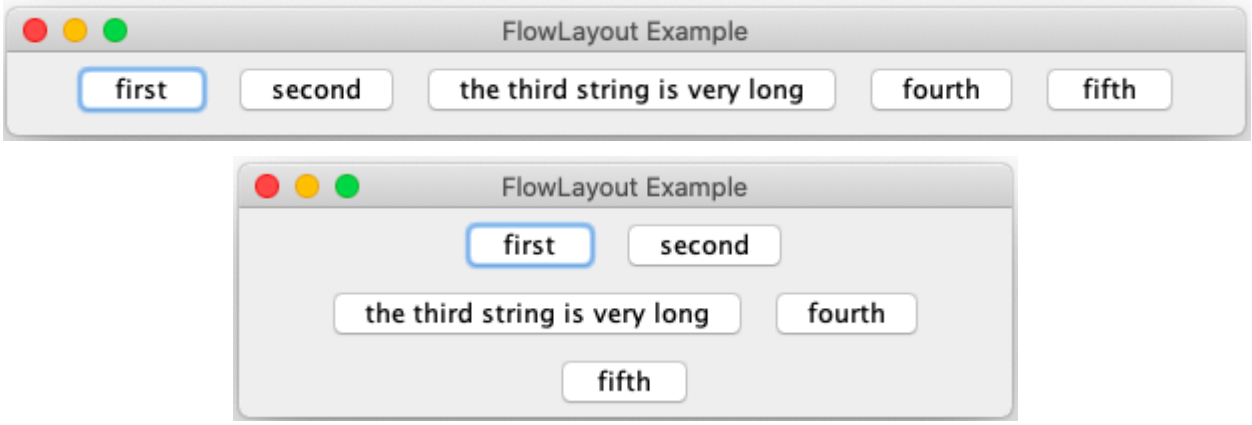
Mostrar retroalimentación

El último elemento añadido al JPanel es el que prevalece. Así que si queremos añadir una JLabel, se mantendrá la etiqueta en vez de el elemento ImagePanel.

Swing proporciona varios gestores de diseño diferentes, para dar soporte a las distintas preferencias de colocación. Los mas importantes son: *FlowLayout*, *BorderLayout*, *GridLayout* y *BoxLayout*. Cada uno de ellos está representado por una clase Java en la librería Swing y cada uno coloca de diferentes maneras los componentes que tiene bajo su control.

He aquí un breve descripción de cada uno de esos posibles diseños. Las diferencias principales entre ellos son la forma en la que se colocan los componentes y cómo se distribuye el espacio disponible entre los mismos. Puede encontrar los ejemplos ilustrados aquí.

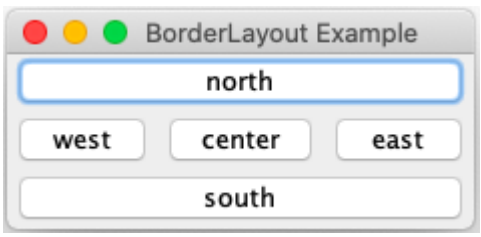
FlowLayout dispone todos los componentes secuencialmente de izquierda a derecha. Dejará cada componente con su tamaño preferido y los centrará horizontalmente. Si el espacio horizontal no es suficiente como para encajar todos los componentes, algunos de ellos saltarán a una segunda línea. El gestor *FlowLayout* puede también configurarse para alinear los componentes a la izquierda o a la derecha. Dado que los componentes no se redimensionan para rellenar el espacio disponible, habrá espacio libre alrededor de ellos si se cambia el tamaño de la ventana.

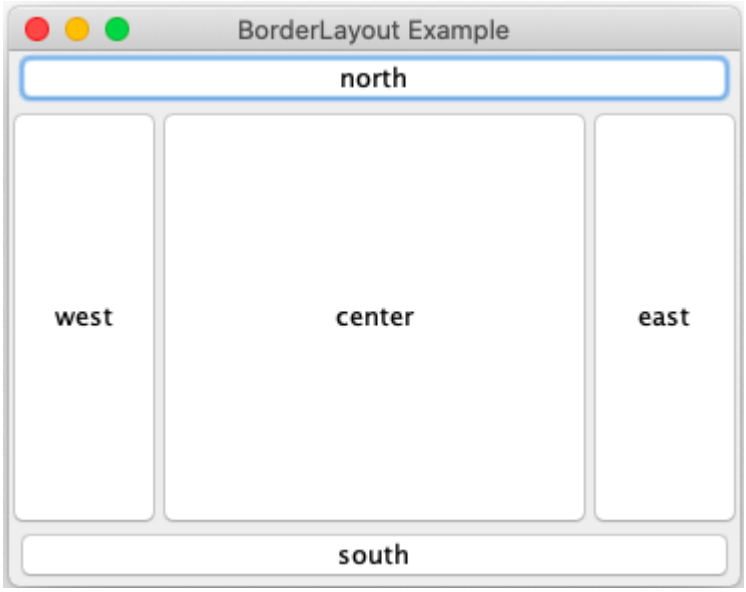


Un *BorderLayout* coloca hasta cinco componentes en un patrón en forma de cruz: uno en el centro y los otros cuatro componentes en las partes superior, inferior, derecha e izquierda. Cada de estas posiciones puede estar vacía, por lo que podría en total contener menos de cinco componentes. Las cinco posiciones se denominan CENTER, NORTH, SOUTH, EAST y WEST. Con un gestor *BorderLayout* no se deja ningún espacio vacío al cambiar el tamaño de la ventana; todo el espacio se distribuye (de manera no equitativa) entre los componentes.

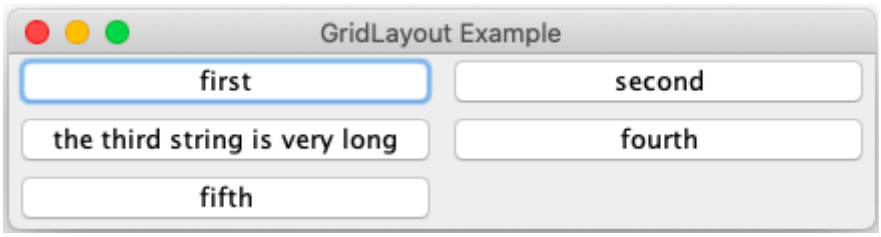
Este diseño gráfico puede parecer bastante especializado a primera vista, lo que nos hace preguntarnos si se suele utilizar muy a menudo. Pero en la práctica es un diseño gráfico sorprendentemente útil que se utiliza en muchas aplicaciones. En BlueJ, por ejemplo, tanto la ventana principal como el editor utilizan un *BorderLayout* como gestor de diseño gráfico principal.

Cuando se redimensiona un *BorderLayout*, el componente central es el que se estira o encoge en ambas direcciones. Los componentes situados al este y al oeste cambian de altura, pero mantienen su anchura. Los componentes situados en las posiciones al norte y al sur mantienen su altura, por lo que solo varía su anchura.

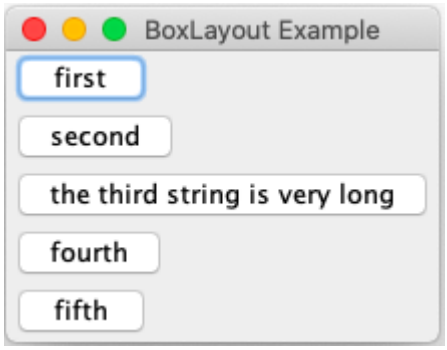




Como el nombre sugiere, un *GridLayout* resulta útil para disponer los componentes en una cuadrícula (grid) equiespaciada. Pueden espaciarse los números de filas y columnas y el gestor *GridLayout* siempre mantendrá todos los componentes con el mismo tamaño. Esto puede ser útil, por ejemplo, para obligar a que los botones tengan la misma anchura… La anchura de las instancias de *JButton* está inicialmente determinada pa el texto contenido en el botón, cada botón se hace lo suficientemente ancho como para poder mostrar su texto. Insertar los botones en un *GridLayout* hará que se redimensionen todos los botones hasta adquirir la anchura del botón más ancho. Si un número impar de componentes del mismo tamaño no pueden rellenar una cuadrícula 2D, puede haber algo de espacio libre en algunas configuraciones.



Un *BoxLayout* coloca múltiples componentes en forma vertical u horizontal. Los componentes no se redimensionan y el gestor no hará que salten de línea o de columna al cambiar el tamaño de la ventana. Anidando varios gestores *BoxLayout* pueden construirse diseños gráficos sofisticados, alineados bidimensionalmente.



Ejercicio 9.17

Utilizando el proyecto layouts, experimente con los ejemplos ilustrados en esta sección. Añada y elimine componentes de las clases existentes, para ver cuáles son las características claves de los diferentes estilos de diseño gráfico. ¿Qué sucede, por ejemplo, si no hay ningún componente CENTER con *BorderLayout*?

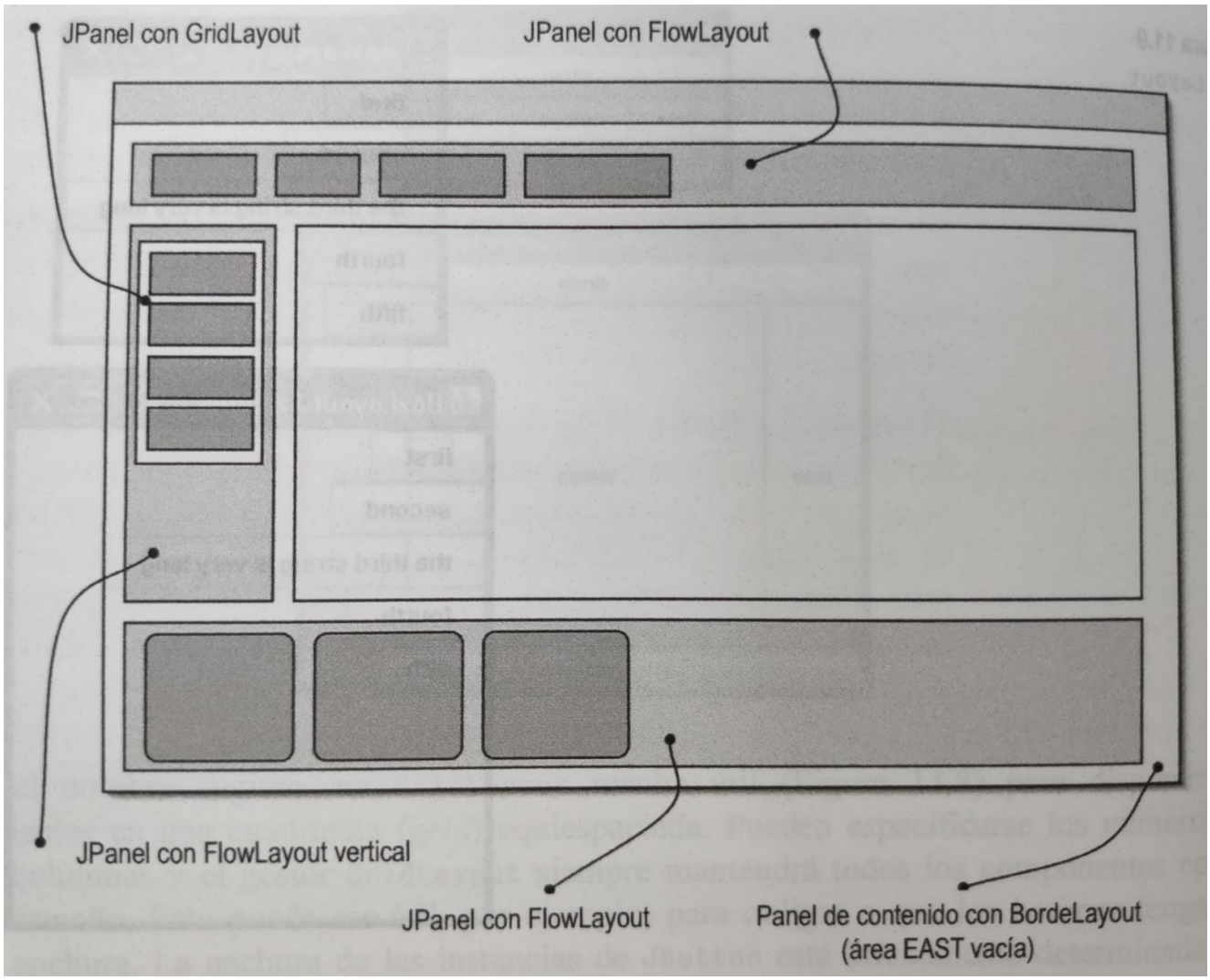
Mostrar retroalimentación

El espacio que debería ocupar el componente CENTER queda vacío. El resto de elementos pueden ocupar ese espacio.

9.4.4 Contenedores anidados

Todas las estrategias de diseño gráfico que acabamos de explicar son bastante simples. La clave para construir interfaces con un buen aspecto y un comportamiento correcto radica en un último detalle: los gestores de diseño pueden anidarse. Muchos de los componentes de Swing son contenedores. Los contenedores parecen ser de cara al exterior un único componente, pero pueden contener otros muchos componentes. Cada contenedor tiene asociado su propio gestor de diseño gráfico.

El contenedor más utilizado es la clase *JPanel*. Puede insertarse un *JPanel* como componente en el panel de contenido del marco, después de lo cual pueden colocarse más componentes dentro del *JPanel*. Por ejemplo, la siguiente imagen muestra una disposición de interfaz similar a la de la ventana principal de BlueJ. El panel de contenido de este marco utiliza un *BorderLayout*, en el que la posición EAST no está utilizada. El área NORTH de este *BorderLayout* contiene un *JPanel* con un *FlowLayout* horizontal que dispone sus componentes (por ejemplo, botones de la barra de tareas) en una fila. El área SOUTH es similar: otro *JPanel* con un *FlowLayout*.



El grupo de botones en el área WEST se coloca primero en un *JPanel* con un *GridLayout* de una sola columna, para proporcionar a todos los botones la misma anchura. Este *JPanel* fue entonces colocado dentro de otro *JPanel* con un *FlowLayout* de modo que la cuadrícula no ocupara toda la altura del área WEST. El *JPanel* externo se insertó entonces dentro del am WEST del marco.

Observe cómo cooperan el contenedor y el gestor de diseño gráfico a la hora de colocar los componentes. El contenedor almacena los componentes, pero es el gestor de diseño el que decide su posición exacta en la pantalla. Cada contenedor tiene un gestor de diseño gráfico; si no configuramos explícitamente uno, utilizará un gestor de diseño predeterminado.

El gestor predeterminado es distinto para los diferentes contenedores: el panel de contenido de un ***JFrame***, por ejemplo, tiene de forma predeterminada un *BorderLayout*, mientras que ***JPanel*** utiliza un ***FlowLayout*** por omisión.

Ejercicio 9.18

Examine la GUI de esta calculadora.

¿Qué tipo de contenedores/gestores de diseño gráfico cree que se utilizaron para crearlo? Después de responder a esta pregunta, abra el proyecto calculator-gui y compruebe su respuesta leyendo el código.

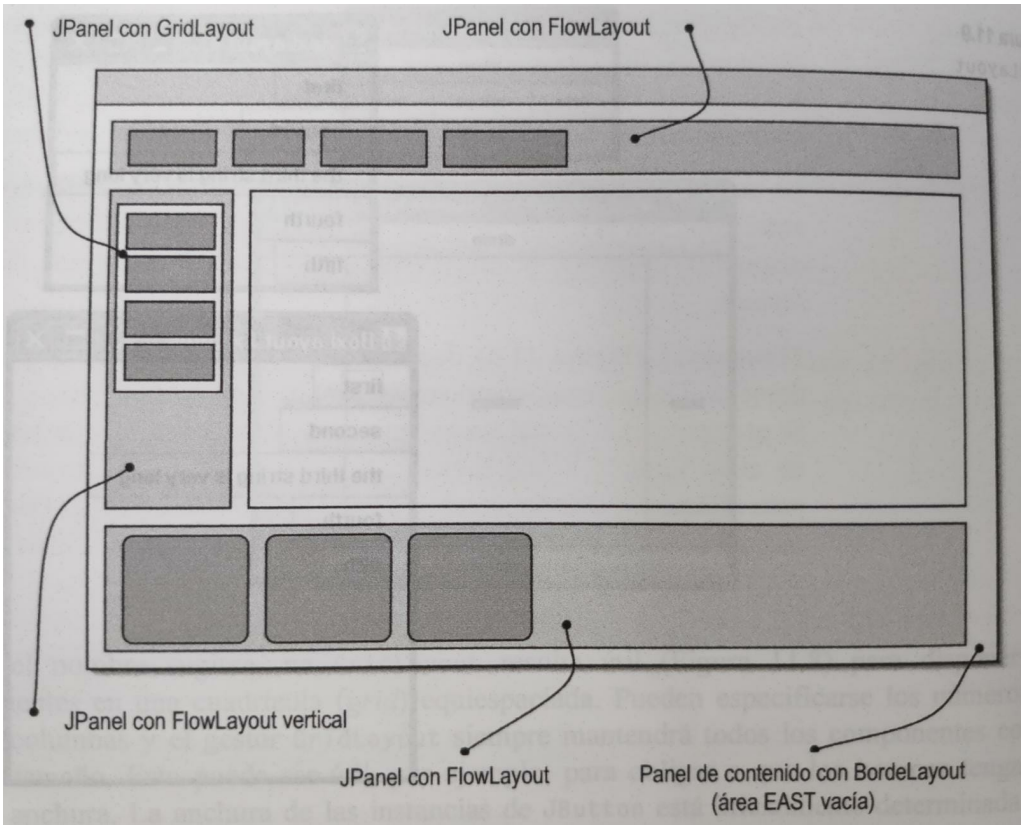
Mostrar retroalimentación

Utiliza varios *BorderLayout* anidados y un *GridLayout*

Ejercicio 9.19

¿Qué tipo de gestores de diseño gráfico podrían haberse utilizado para crear el diseño de la ventana del editor de BlueJ?

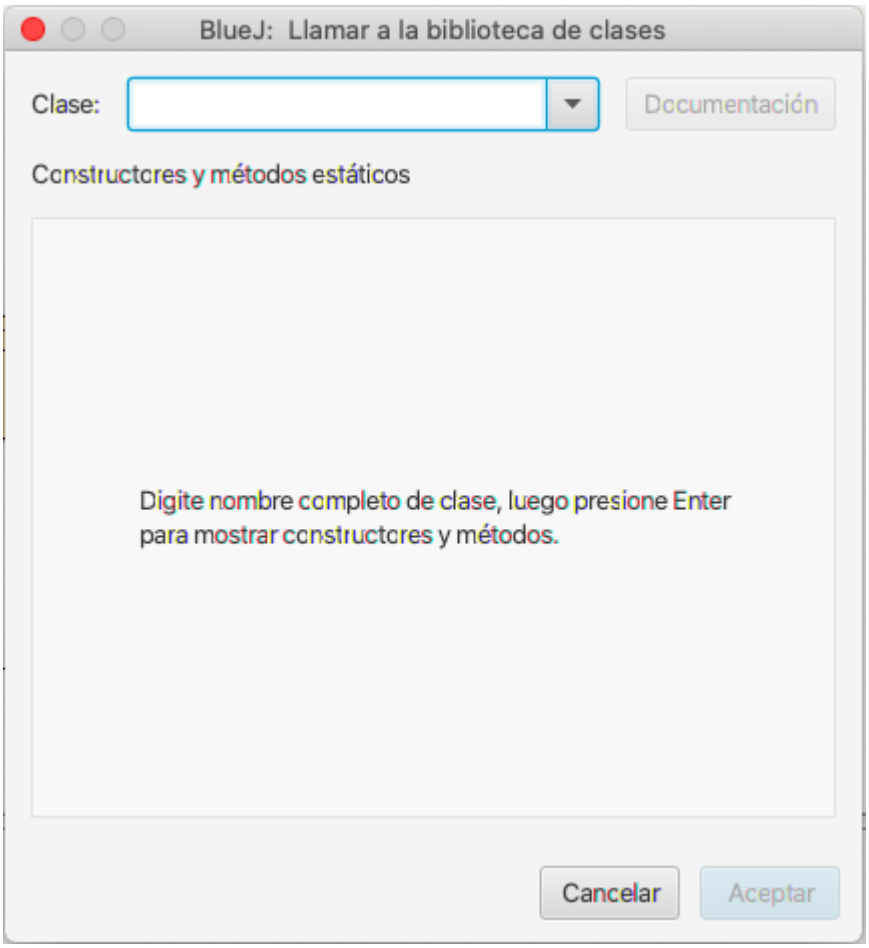
Mostrar retroalimentación



Ejercicio 9.20

En BlueJ. invoque la función *Usar de bilblioteca de clases (Library Class)* del menú Herramientas (Tools). Examine el cuadro diálogo que aparece en pantalla. ¿Qué contenedores/gestores de diseño gráfico se han utilizado para crearlo? Redimensione el cuadro de diálogo y observe el comportamiento del mismo ante el cambio de tamaño, para obtener información adicional.

Mostrar retroalimentación



Es el momento de examinar algo de código para nuestra aplicación *ImageViewer*. Nuestro objetivo es muy simple. Queremos ver tres componentes uno encima de otro: una etiqueta en la parte superior, la imagen en la parte central y otra etiqueta en la parte inferior. Son varios los gestores de diseño gráfico que permiten hacer esto y estará más claro cuál debemos elegir después de pensar acerca del comportamiento en caso de redimensionamiento. Cuando agrandemos la ventana, nos gustaría que las etiquetas mantuvieran su anchura y que todo el espacio adicional se asignara a la imagen. Esto sugiere un *BorderLayout*: las etiquetas pueden estar en las áreas NORTH y SOUTH, y la imagen en el área CENTER. El siguiente código muestra cómo implementar esto.

```
contentPane = frame.getContentPane();

contentPane.setLayout(new BorderLayout());

filenameLabel = new JLabel();
contentPane.add(filenameLabel, BorderLayout.NORTH);

imagePanel = new ImagePanel();
contentPane.add(ImagePanel, BorderLayout.CENTER);
```

```
statusLabel = new JLabel("Versíon 1.0");
contentPane.add(statusLabel, BorderLayout.SOUTH);
```

Merece la pena resaltar dos detalles. En primer lugar, se utiliza el método ***setLayout*** en el panel de contenido para configurar el gestor de diseño gráfico deseado. El propio gestor de diseño gráfico es un objeto, así que creamos una instancia de ***BorderLayout*** y lo pasamos al método ***setLayout***.

En segundo lugar, cuando añadimos un componente a un contenedor con un ***BorderLayout***, utilizamos un método ***add*** diferente que tiene un segundo parámetro. El valor del segundo parámetro es una de las constantes públicas NORTH, SOUTH, EAST, HEST y CENTER, que están definidas en la clase ***BorderLayout***.

Ejercicio 9.21

Implemente y pruebe el código mostrado más arriba en su versión del proyecto.

Ejercicio 9.22

Experimente con otros gestores de diseño gráfico. Pruebe en su proyecto todos los gestores de diseño mencionados anteriormente y compruebe que se comportan como cabe esperar.

9.4.5 Cuadros de diálogo

Nuestra última tarea para esta versión consiste en añadir un menú Help que tenga un elemento de menú denominado *About ImageViewer*... Al seleccionar este elemento de menú, aparecerá un cuadro de diálogo que mostrará un pequeño texto informativo. Ahora tenemos que implementar el método *showAbout* para que muestre un cuadro de diálogo "About".

Ejercicio 9.23

De nuevo, añada un menú denominado Help. En él, añada un elemento de menú etiquetado como About ImageViewer.

Mostrar retroalimentación

```
//Menu Help
JMenu helpMenu = new JMenu("Help");
menubar.add(helpMenu);
JMenuItem aboutItem = new JMenuItem("About
ImageViewer");
helpMenu.add(aboutItem);
```

Ejercicio 9.24

Añada un esqueleto de método (un método con un cuerpo vacío) denominado showAbout, y añada un escucha de acción al elemento de menú About ImageViewer. .. en el que se invoque a ese método.

Mostrar retroalimentación

```
//Menu Help
JMenu helpMenu = new JMenu("Help");
menubar.add(helpMenu);
JMenuItem aboutItem = new JMenuItem("About
ImageViewer");
```

```
aboutItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        showAbout();
    }});
helpMenu.add(aboutItem);
```

Una de las principales características de un cuadro de diálogo es si se trata de un cuadro de diálogo modal o no. Un cuadro de diálogo modal bloquea todas las interacciones con otras partes de la aplicación hasta que el cuadro de diálogo se haya cerrado. Obliga al usuario a tratar primero con lo que ese cuadro de diálogo le esté comunicando. Los cuadros de diálogo no modales permiten interactuar en otros marcos mientras están visibles.

Los cuadros de diálogo pueden implementarse de forma similar a nuestro *JFrame* principal. A menudo utilizan la clase *JDialog* para mostrar el marco.

Sin embargo, para los cuadros de diálogo modales con una estructura estándar, existen algunos métodos cómodos en la clase *JOptionPane* que hacen que resulte muy sencillo mostrar dichos cuadros de diálogo. *JOptionPane* tiene, entre otras cosas, métodos estáticos para mostrar tres tipos de cuadros de diálogo estándar. Estos tres tipos son:

- Cuadro de diálogo de mensajes: este cuadro de diálogo muestra un mensaje y tiene un botón OK para cenar el cuadro.
- Cuadro de diálogo de confirmación: este cuadro de diálogo suele plantear una pregunta y tiene botones para que el usuario haga una selección; por ejemplo, Yes, No y Cancel.
- Cuadro de diálogo de entrada: este cuadro de diálogo incluye un indicativo y un campo de texto para que el usuario introduzca un texto.

Nuestro recuadro “About” es un cuadro de diálogo de mensaje simple. Buscando en la documentación de *JOptionPane*, encontramos que hay métodos estáticos, *showMessageDialog*, que permiten hacer esto.

Ejercicio 9.25

Localice la documentación de *showMessageDialog*. ¿Cuántos métodos hay con este nombre? ¿Cuáles son las diferencias entre ellos? ¿Cuál deberíamos usar para el recuadro 'About'? ¿Por qué?

Mostrar retroalimentación

En la API (Java SE 11 & JDK 11) hay 3 métodos con el mismo nombre:

static void	showMessageDialog(Component parentComponent, Object message)	Brings up an information-message dialog titled "Message".
static void	showMessageDialog(Component parentComponent, Object message, String title, int messageType)	Brings up a dialog that displays a message using a default icon determined by the messageType parameter.
static void	showMessageDialog(Component parentComponent, Object message, String title, int messageType, Icon icon)	Brings up a dialog displaying a message, specifying all parameters.

Las diferencias entre ellos es el número y tipo de parámetros que aceptan.

Podríamos usar la segunda opción, ya que podríamos poner un mensaje y un título a esa ventana emergente que nos aparecería. Si queremos a demás añadirle un icono personalizado, podríamos usar la tercera opción.

Ejercicio 9.26

Implements el método showAbout en su clase *ImageViewer*, utilizando una llamada a un método *showMessageDialog*.

Mostrar retroalimentación

```
private void showAbout() {
    JOptionPane.showMessageDialog(frame,
    "ImageViewer\n","About
    ImageViewer", JOptionPane.INFORMATION_MESSAGE);
}
```

Ejercicio 9.27

Los métodos *showInputDialog* y *JOptionPane* permiten solicitar al usuario que introduzca una cierta entrada mediante un cuadro de diálogo, cuando sea necesario. Por otro lado, el componente *JTextField* permite mostrar un área permanente de texto dentro de una GUI. Localice la documentación para esta clase. ¿Qué entrada provoca que sea notificada un ActionListener asociado con un *TextField*? ¿Se puede impedir a un usuario que edite el texto del campo? ¿Es posible que se notifique a un escucha la realización de cambios arbitrarios en el texto del campo? Sugerencia: ¿qué uso hace un *JTextField* de un objeto Document?

Puede ver un ejemplo de un *JTextField* en el proyecto calculator.

Después de estudiar la documentación, podemos implementar nuestro recuadro “About” haciendo una llamada al método showMessageDialog. La solución de código se muestra en el siguiente código. Observe que hemos introducido una constante de cadena denominada VERSION para almacenar el número de versión actual.

```
private void showAbout() {

    JOptionPane.showMessageDialog(frame,   "ImageViewer\n"   +
    VERSION,
    “About ImageViewer”, JOptionPane.INFORMATION_MESSAGE);

}
```

Esta era la última tarea que nos quedaba por hacer para completar la “versión 1.0” de nuestra aplicación del visualizador de imágenes. Si ha hecho todos los ejercicios, debería disponer ahora de una versión del proyecto que puede abrir imágenes, mostrar mensajes de estado y mostrar un cuadro de diálogo.

El proyecto imageviewer1-0 contiene una implementación de toda la funcionalidad que hemos presentado hasta el momento, incluidos filtros de imagen. Debería estudiar cuidadosamente este proyecto y compararlo con sus propias soluciones.

En este proyecto, hemos mejorado también el método openFile para incluir una mejor notificación de los errores. Si el usuario selecciona un archivo que no sea un archivo de imagen válido, debemos mostrar ahora un mensaje de error apropiado. Ahora que sabemos como diseñar cuadros de diálogo de mensajes, esto resulta fácil de hacer.

9.5 Ampliaciones adicionales

La programación de interfaces GUI con Swing es un área enormemente extensa. Swing dispone de muchos tipos distintos de componentes y de muchos contenedores y gestores de diseño gráfico distintos. Cada uno de ellos tiene muchos atributos y métodos.

Familiarizarse con la toda la librería Swing requiere su tiempo y no es algo que se pueda hacer en unas pocas semanas. Normalmente, a medida que vamos trabajando con interfaces GUI, lo que hacemos es continuar leyendo acerca de detalles que no conocíamos antes para terminar convirtiéndonos en expertos con el paso del tiempo.

El ejemplo expuesto en este capítulo, aunque contiene un montón de detalles, es solamente una breve introducción a la programación de interfaces GUI. Hemos explicado la mayor parte de los conceptos importantes, pero sigue habiendo una gran cantidad de funcionalidad que descubrir, la mayor parte de la cual cae fuera del alcance de este libro. Hay disponibles varias fuentes de información para ayudarle a continuar. Tendrá que consultar con frecuencia la documentación de la API para las clases Swing; es imposible trabajar sin ella. También hay disponibles muchos tutoriales GUI/Swing, tanto en forma de libro como en la Web.

Un muy buen punto de partida para esto, como suele suceder a menudo, es el tutorial Java disponible públicamente en el sitio web de Oracle. Contiene una sección titulada Creating a GUI with JFC/Swing.

En esta sección, hay muchas subsecciones interesantes. Una de las más útiles puede ser la sección titulada Using Swing Components y dentro de ella la subsección How To... Contiene las siguientes entradas: How to Use Buttons, Check Boxes, and Radio Buttons; How to Use Labels; How to Make Dialogs; How to Use Panels; etc.

De forma similar, la sección Laying Out Components within a Container también contiene una sección How To..., que informa acerca de todos los gestores de diseño gráfico (layout manager) disponibles.

Debes conocer

Si bien es cierto que en este tema hemos abordado la creación de interfaces gráficas de usuario sin la ayuda del IDE, existen muchos Entornos de Desarrollo que integran la opción "drag & drop" para crear interfaces gráficos de forma más rápida. La creación de éste tipo de GUI no presenta una estructura tan "pura" como las estudiadas, ya que no se basan en ninguno de los diseños gráficos vistos en el tema.

La rapidez es una de las ventajas a la hora de elegir éste método para crear una interfaz, pero por contra, muchos IDEs no permitirán modificar parte del código de esos componentes creados automáticamente.

A la hora de programar tendremos que ver qué es lo que nos compensa utilizar.

Aquí tenéis un enlace a un pequeño tutorial de **NetBeans** y el uso del **GUI Builder** para construir la interfaz.

Número de sesiones

Se estiman un total de 18 horas