

# Índice General

Capítulo 1.- Introducción .....	19
1.1. Interfaces de usuario .....	19
1.1.1. Aplicación de consola vs. Aplicación con Interfaz Gráfica de Usuario .....	21
1.1.2. Elementos gráficos .....	22
1.1.3. Gestión de los elementos gráficos .....	27
1.1.4. Gestión de Eventos .....	28
1.2. Interfaces Gráficas de Usuario en Java .....	32
1.2.1. Tipos de aplicaciones con interfaz gráfica de usuario .....	33
1.3. Pasos básicos para la creación de una interfaz gráfica de usuario .....	34
1.3.1. Creación de una ventana de aplicación .....	37
1.3.2. Creación de los componentes de una ventana de aplicación ..	38
1.3.3. Creación de un contenedor .....	39
1.3.4. Asociación de componentes a un contenedor .....	39
1.3.5. Asociación de un contenedor a la ventana de aplicación .....	40
1.3.6. Hacer visible la ventana .....	40
1.4. Eventos .....	42
1.5. La primera aplicación con interfaz gráfica de usuario .....	45
Capítulo 2.- Organización interna de una aplicación con interfaz gráfica .....	51
2.1. Introducción .....	51
2.2. Estructura general. Interfaz y Datos .....	52
2.3. Organización del código de elementos gráficos .....	59
2.4. Organización del código de eventos .....	62
2.4.1. Varios componentes generan eventos para la misma acción ..	62
2.4.2. Varios componentes generan eventos del mismo tipo para acciones distintas .....	65
2.4.3. Varios componentes generan eventos de distinto tipo .....	69
2.4.4. Gestores de eventos reutilizables .....	75
2.4.5. Clases anónimas en la gestión de eventos .....	79
2.4.6. Gestión de eventos cuando el número de componentes es dinámico .....	87

2.4.7. Conclusiones en la organización del código de gestión de eventos .....	92
Capítulo 3.- Organización de los componentes: administradores de distribución y bordes .....	93
3.1. Introducción .....	93
3.2. Administradores de distribución .....	93
3.2.1. Implementación de los administradores de distribución .....	96
3.2.2. Administrador de distribución FlowLayout.....	97
3.2.3. Administrador de distribución BorderLayout.....	99
3.2.4. Administrador de distribución GridLayout.....	103
3.2.5. Administrador de distribución GridBagLayout.....	104
3.2.6. Otros administradores de distribución: BoxLayout, SpringLayout y CardLayout .....	129
3.3. Administrador de distribución nulo. Distribución estática .....	130
3.4. Administración por anidamiento de contenedores.....	130
3.5. Bordes .....	131
Capítulo 4.- Visión general de la librería Swing.....	92
4.1. Introducción .....	139
4.2. Los componentes de Swing .....	140
4.3. Los eventos de Swing .....	145
4.4. Imágenes.....	153
4.4.1. Iconos.....	153
4.5. Paneles de opciones .....	156
4.6. Diálogos.....	161
4.7. Menús.....	164
4.7.1. Menús en barras de menús .....	165
4.7.2. Menús independientes .....	170
Capítulo 5.- Creación de componentes personalizados.....	175
5.1. Introducción .....	175
5.2. Recordatorio de eventos .....	176
5.3. Creación de eventos propios .....	177
Capítulo 6.- Diseño de aplicaciones con interfaz gráfica: separación de datos e interfaz.....	185
6.1. Introducción .....	185

---

6.2. Construcción interna de componentes.....	186
6.2.1. Ventajas de la arquitectura modelo – delegado.....	189
6.2.2. Implementación de la arquitectura modelo – delegado en Swing .....	191
6.2.3. Varios componentes representan los mismos datos.....	195
6.2.4. Separación de responsabilidades entre el componente y el modelo.....	195
6.2.5. Eventos en el modelo.....	198
6.3. Construcción de una aplicación con interfaz gráfica de usuario.....	201
Capítulo 7.- Técnicas para evitar el bloqueo de la interfaz gráfica .....	184
7.1. Introducción .....	209
7.2. Hilo de despacho de eventos de Swing .....	210
7.3. Ejecución de tareas en segundo plano (background).....	212
7.4. Actualización de la interfaz gráfica desde una tarea en segundo plano .....	212
Capítulo 8.- Tipos de aplicaciones con interfaz gráfica de usuario y su distribución.....	225
8.1. Introducción .....	225
8.2. Aplicación autónoma .....	226
8.3. Java Web Start .....	229
8.4. Applet.....	235
8.5. Sumario de la distribución de aplicaciones Java.....	242

## Índice de Tablas

Tabla 1.1: Direcciones de Internet donde encontrar información sobre Swing .....	33
Tabla 2.1: Diccionario de la aplicación de traducción .....	54
Tabla 4.1: Componentes simples .....	140
Tabla 4.2: Componentes complejos.....	140
Tabla 4.3: Componentes de texto .....	141
Tabla 4.4: Contenedores .....	141
Tabla 4.5: Ventanas .....	141
Tabla 4.6: Menús .....	142
Tabla 4.7: Otro componente .....	142
Tabla 4.8: Eventos para la clase Component.....	146
Tabla 4.9: Eventos para la clase AbstractButton.....	146
Tabla 4.10: Eventos para la clase JComboBox.....	146
Tabla 4.11: Eventos para la clase Window .....	147
Tabla 4.12: Eventos para la clase JTextComponent.....	147
Tabla 4.13: Eventos para la clase JEditorPane .....	147
Tabla 4.14: Eventos para la clase JTextField .....	148
Tabla 4.15: Eventos para la clase JInternalFrame.....	148
Tabla 4.16: Eventos para la clase JList .....	148
Tabla 4.17: Eventos para la clase JMenu .....	149
Tabla 4.18: Eventos para la clase JMenuItem.....	149
Tabla 4.19: Eventos para la clase JPopupMenu .....	149
Tabla 4.20: Eventos para la clase JProgressBar .....	150
Tabla 4.21: Eventos para la clase JSlider .....	150
Tabla 4.22: Eventos para la clase JSpinner.....	150
Tabla 4.23: Eventos para la clase JTabbedPane .....	150
Tabla 4.24: Eventos para la clase JTree .....	151
Tabla 4.25: Eventos para la clase JTable .....	151
Tabla 6.1: Interfaces de los modelos de los componentes.....	192
Tabla 6.2: Listado de delegados existentes .....	194
Tabla 8.1: Tipos de aplicaciones con interfaz gráfica y sus características.....	243

## Índice de Figuras

Figura 1.1: Aplicación de ejemplo con interfaz gráfica de usuario.....	23
Figura 1.2: Ejecución de una aplicación de consola.....	29
Figura 1.3: Ejecución de una aplicación con interfaz gráfica de usuario .....	31
Figura 1.4: Jerarquía de herencia del componente JLabel .....	35
Figura 1.5: Jerarquía de componentes .....	36
Figura 1.6: Interfaz de usuario simple .....	40
Figura 1.7: Varios códigos asociados a un mismo tipo de evento de un mismo componente.....	42
Figura 1.8: Un único código de eventos para varios componentes .....	43
Figura 1.9: Clases relacionadas con el evento de tipo Window.....	45
Figura 1.10: Interfaz gráfica de la aplicación de conversión.....	47
Figura 1.11: Diagrama de clases y objetos de la aplicación de conversión de euros a pesetas (ejemplo 1.2) .....	49
Figura 2.1: Clases básicas de una aplicación con interfaz gráfica de usuario .....	52
Figura 2.2: Diagrama de clases y objetos de la aplicación de traducción 1.0 .....	57
Figura 2.3: Interfaz de la aplicación de traducción.....	57
Figura 2.4: Diagrama de clases y objetos de la aplicación de traducción 2.0 .....	60
Figura 2.5: Diagrama de clases y objetos de la aplicación de traducción 3.0 .....	63
Figura 2.6: Diagrama de clases y objetos de la aplicación de traducción 4.0 .....	66
Figura 2.7: Interfaz gráfica de la aplicación de traducción 4.0 .....	67
Figura 2.8: Diagrama de clases y objetos de la aplicación de traducción 5.0 .....	72
Figura 2.9: Diagrama de clases y objetos de la aplicación de traducción 6.0 .....	76
Figura 2.10: Relación entre los objetos de la clase anónima y contenedora.....	83
Figura 2.11: Diagrama de clases y objetos de la aplicación de conversión de euros a pesetas con clases anónimas .....	84

Figura 2.12: Diagrama de clases y objetos de la aplicación de traducción 7.0 .....	85
Figura 2.13: Diagrama de clases y objetos de la aplicación con número dinámico de componentes .....	88
Figura 2.14: Interfaz gráfica de usuario de la aplicación con número dinámico de componentes .....	89
Figura 3.1: Interfaz gráfica de ejemplo con administrador de distribución FlowLayout .....	99
Figura 3.2: División de zonas del contenedor por el administrador BorderLayout .....	100
Figura 3.3: Asignación de tamaño por regiones en el administrador BorderLayout .....	100
Figura 3.4: Interfaz gráfica de ejemplo con administrador de distribución BorderLayout .....	101
Figura 3.5: Interfaz gráfica de ejemplo con administrador de distribución GridLayout .....	104
Figura 3.6: Forma de la rejilla de un administrador de distribución GridBagLayout con los valores de los atributos que controlan el tamaño de las celdas a su valor por defecto.....	105
Figura 3.7: Atributos de pesos que influyen en el tamaño de las filas y columnas de la rejilla en un JPanel con administrador de distribución GridBagLayout.....	106
Figura 3.8: Ejemplo de distribución de las columnas de una rejilla según el atributo weightx de la clase GridBagConstraints .....	107
Figura 3.9: Ejemplo de distribución de las filas de una rejilla según el atributo weighty de la clase GridBagConstraints .....	107
Figura 3.10: Ejemplo de espaciado alrededor de un componente del atributo insets de la clase GridBagConstraints .....	111
Figura 3.11: Interfaz gráfica de usuario del ejemplo 3.2.1 .....	112
Figura 3.12: Interfaz gráfica de usuario del ejemplo 3.2.2 .....	114
Figura 3.13: Interfaz gráfica de usuario del ejemplo 3.2.3 .....	115
Figura 3.14: Interfaz gráfica de usuario del ejemplo 3.2.4 .....	117
Figura 3.15: Interfaz gráfica de usuario del ejemplo 3.3.1 .....	118
Figura 3.16: Interfaz gráfica de usuario del ejemplo 3.3.2 .....	120
Figura 3.17: Interfaz gráfica de usuario de la aplicación de traducción 8.0 .....	124

Figura 3.18: Interfaz gráfica de usuario redimensionada de la aplicación de traducción 8.0.....	124
Figura 3.19: Ejemplo de interfaz gráfica de usuario con bordes .....	131
Figura 3.20: Interfaz de usuario de la aplicación de traducción 9.0.....	132
Figura 4.1: Vista de una jerarquía de herencia razonable para los componentes Swing .....	143
Figura 4.2: Jerarquía de clases real de los componentes Swing .....	144
Figura 4.3: Clases <code>XXListener</code> correspondientes a cada uno de los eventos .....	151
Figura 4.4: Clases <code>XXEvent</code> pasadas como parámetro en cada uno de los métodos de <code>XXListener</code> .....	152
Figura 4.5: Paneles de opciones predefinidos en Swing .....	156
Figura 4.6: Interfaz de usuario del Visor de Imágenes 1.0.....	158
Figura 4.7: Aplicación ejemplo de diálogo .....	162
Figura 4.8: Clases que permiten gestionar los menús en Swing .....	164
Figura 4.9: Aplicación de ejemplo con un menú.....	165
Figura 4.10: Clases que permiten gestionar los menús en Swing .....	165
Figura 4.11: Aplicación de ejemplo con menús <code>JPopupMenu</code> . .....	171
Figura 6.1: Relaciones de los objetos de la implementación interna de un componente .....	187
Figura 6.2: Estructura de los componentes con modelo de datos y delegado.....	188
Figura 6.3: Estructura de componentes con varios modelos de datos y delegado.....	189
Figura 6.4: Estructura de componentes con varios modelos de datos y delegado.....	191
Figura 6.5: Separación de datos e interfaz (MVC) .....	201
Figura 6.6: Generación de eventos de las clases de datos cuando su estado cambia .....	204
Figura 6.7: Misma información en dos visualizaciones diferentes.....	204
Figura 7.1: Ejecución de una aplicación con interfaz gráfica de usuario .....	211
Figura 8.1: Página Web con Applet incorporado .....	241





## Prólogo

Una interfaz de usuario es la parte del sistema que interactúa con el usuario. Dentro de éstas las más comunes son las interfaces gráficas, las cuales hacen uso de un entorno gráfico para interactuar con él. Por tanto, la interfaz gráfica de usuario es la parte visible de las aplicaciones, es lo que se percibe del sistema y, por ello, tiene mucha importancia. A través de ella y, de forma transparente, se debe proporcionar, entre otras cosas: seguridad, fiabilidad, sencillez y precisión.

El objetivo del libro es, por un lado, presentar los conceptos básicos que se aplican en la construcción de cualquier tipo de interfaz gráfica de usuario y, por otro lado, su implementación en el lenguaje Java. No se pretende describir de forma detallada todas y cada una de las características de la librería estándar utilizada en Java para la construcción de interfaces gráficas. En su lugar, se trata de ofrecer una visión global y de alto nivel al lector sobre la tecnología y sus posibilidades. Además, se hace especial hincapié en el hecho de que las aplicaciones construidas sean modulares, comprensibles, reutilizables y mantenibles.

Los contenidos que se exponen, así como el orden y el modo de presentación de los mismos, son fruto de la experiencia de los autores tanto en el ámbito docente como en el de desarrollo de aplicaciones con interfaz gráfica de usuario en Java.

Por otro lado, este libro está especialmente enfocado para ser usado como guía de estudio en la asignatura de Interfaces de Usuario, del tercer curso de Ingeniería Informática de la Universidad Rey Juan Carlos. Considerándose también útil como libro de apoyo en la asignatura Programación Orientada a Objetos, de segundo curso de la misma titulación.

Por último, se trata de un libro que puede ser de mucha utilidad para cualquier persona que conozca el lenguaje Java y que quiera aprender a construir aplicaciones con interfaz gráfica de usuario mediante dicho lenguaje de programación.

Los contenidos teóricos y prácticos del libro han sido revisados y ampliados para ajustarse a la versión Java 2 Standard Edition (J2SE) 5.0.

Los Autores

## Recomendaciones de uso del libro

Este libro presenta los conceptos básicos necesarios para la construcción de interfaces gráficas de usuario. Para ello, se emplea la librería Swing de Java.

Los autores han pretendido ofrecer una visión global y de alto nivel, exponiendo unos contenidos acorde con ello, y en un orden y cuantía adecuados para la comprensión y el estudio de las interfaces gráficas de usuario en Java. Con esto, se quiere hacer hincapié en que la mejor forma de leer este libro es siguiendo el orden en el que se van desarrollando los diferentes temas. Sólo para aquellos con conocimientos más avanzados, puede ser recomendable una lectura de temas o apartados concretos, sin necesidad de seguir un orden secuencial. Decidir la estructura del libro no ha resultado una tarea fácil. Separar unos contenidos de otros para que se puedan estudiar de uno en uno es complicado, por la propia naturaleza de la librería Swing. Todo está muy relacionado, por ello, en ocasiones, desde un capítulo determinado se hace referencia a otro capítulo para ampliar información, para hacer uso de algún ejemplo ya expuesto, etc.

A lo largo del libro, formando parte de algunos apartados, aparece un título del estilo “Aprendiendo la API...”. Se trata de apuntes sobre la librería, que quizá no tienen entidad suficiente para dedicarles un apartado completo, pero sí son importantes y están relacionados con lo que se trata en el apartado en el que se encuentran.

Algo destacable son los diferentes ejemplos repartidos por todos los capítulos del libro. Estos ejemplos intentan reflejar usos de los conceptos expuestos. Y lo más importante es que, en la medida de lo posible, se ha desarrollado un ejemplo conductor que se ha ido ampliando a lo largo de los diferentes temas. De esta manera, el estudiante puede unificar mejor los conceptos estudiados. Todos los ejemplos del libro están disponibles en formato adecuado en la dirección web <http://www.escet.urjc.es/~smontalvo/interfacesgraficasenjava/>.

Además, los ejemplos se acompañan de diagramas de clases y objetos que facilitan la comprensión de los mismos, permitiendo que la estructura de la aplicación gráfica que se presenta sea más clara. Los diagramas de

clases y objetos se representan siguiendo el Lenguaje Unificado de Modelado (UML, *Unified Modeling Language*) [Booch99], exceptuando lo referente a las relaciones de gestión de eventos y de pintado (cuya notación se indica en el libro). UML es un lenguaje estándar, basado en una notación gráfica, que se utiliza para modelar software.

# Capítulo 1

## Introducción

Este capítulo pretende dar una visión general de las interfaces gráficas de usuario. Para ello, entre otras cosas, describe de forma breve qué se entiende por *interfaz de usuario* e *interfaz gráfica de usuario*. Estudia los conceptos generales sobre elementos gráficos y eventos, e indica los pasos básicos a seguir en la construcción de una aplicación con interfaz gráfica.

### 1.1. Interfaces de usuario

Si un ser humano quiere interactuar con una máquina necesita un medio para poder hacerlo. Ese medio es, precisamente, lo que se conoce como **interfaz de usuario; la parte del sistema que interactúa con el usuario**. La interfaz es un lugar de encuentro entre los bits y las personas [Neg94].

La interfaz es la parte visible de las aplicaciones, siendo lo que se percibe de las mismas; por ello, cada vez se les está dando una importancia mayor y se está poniendo más cuidado en su desarrollo. La creación de interfaces de usuario es un área, dentro del desarrollo de software, que ha evolucionado mucho en los últimos años y lo sigue haciendo a día de hoy.

Se podría decir que la interfaz de usuario es lo primero que se juzga de una aplicación, y si no tiene la calidad adecuada puede producir rechazo por parte del usuario. Una interfaz puede incluso poner limitaciones en la

comunicación de la máquina con el usuario. Todo aquello que no se pueda expresar a través de la interfaz se perderá, por ello, tiene gran importancia dentro del desarrollo de una aplicación el diseño de su interfaz.

Un buen programa con una interfaz pobre y no adaptada a las tareas a realizar, no sirve de mucho, ya que la interfaz será más una barrera que un medio que facilite la interacción. Hay que tener siempre presente que la interfaz de usuario determina la usabilidad de la aplicación.

La usabilidad es una medida de uso de un producto por determinados usuarios, con el fin de alcanzar ciertos objetivos de eficiencia, efectividad y satisfacción, todo ello dentro de un contexto de uso concreto [Abascal01]. Por lo tanto, la interfaz de usuario debe ser usable, lo que implicará que el usuario consiga realizar las tareas por medio de ella de forma rápida y sencilla, y se encuentre cómodo con la aplicación.

Realmente las interfaces son el medio en sí. Por ello, las energías del usuario no deben concentrarse en el uso de la interfaz, sino en su propio trabajo [Nor88]. La interfaz de usuario se encarga de adaptar las complejidades del sistema con las capacidades humanas.

En definitiva, se puede decir que una interfaz de usuario debe ser sencilla, de manera que su uso resulte sencillo y el aprendizaje de la misma sea rápido. Por lo tanto, debe ser intuitiva y directa. También debe ser consistente, manteniendo una uniformidad a lo largo de toda su exposición. En definitiva, el usuario debe sentirse cómodo al manejarla y, también, satisfecho, pudiendo realizar las tareas que desee de manera efectiva.

Una interfaz gráfica de usuario (GUI) es una interfaz de usuario en la que se hace uso de un entorno gráfico. Es decir, permite la interacción del usuario con el ordenador mediante la utilización de imágenes, objetos pictóricos (ventanas, iconos, botones, etcétera),..., además de texto. GUI es un acrónimo del vocablo inglés *Graphical User Interface*.

En la actualidad, la interfaz gráfica de usuario más conocida es el sistema de ventanas.

Para comprender mejor la naturaleza de una aplicación con interfaz gráfica se va a comparar ésta con una aplicación de consola o textual. En una aplicación de consola, donde no existe interfaz gráfica, la interacción con el usuario se realiza de forma textual. Mientras que en una aplicación con interfaz gráfica, la interacción se realiza mediante los elementos gráficos que forman parte de la interfaz. Éstos muestran el estado de la aplicación y permiten su control.

### *1.1.1. Aplicación de consola vs. Aplicación con Interfaz Gráfica de Usuario*

Los elementos que componen la interfaz gráfica son **elementos gráficos**, y a través de ellos el usuario puede interactuar con la aplicación. En esta interacción el usuario introduce datos que el programa necesita para llevar a cabo su funcionalidad y obtiene los resultados de procesar dichos datos. Por ejemplo, las ventanas, los botones, las imágenes, etc. son elementos gráficos.

Una diferencia clara entre una aplicación de consola y una aplicación con interfaz gráfica de usuario, es que la primera no tiene ningún elemento gráfico, mientras que en la segunda éstos si existen.

Por otra parte, un **evento** es la notificación que hace un elemento gráfico cuando el usuario interactúa con él. Por lo tanto, si se realiza alguna acción sobre algún elemento de la interfaz, se dice que se ha generado un evento en dicho elemento.

Otra diferencia destacable entre una aplicación de consola y una con interfaz gráfica de usuario está relacionada con los eventos y su gestión, y afecta notablemente a la hora de programar la aplicación. En una aplicación de consola el programa decide cuándo necesita datos del usuario, y es en ese momento cuando los lee de la cadena de entrada. Sin embargo, una aplicación con interfaz gráfica siempre se encuentra a la espera de una entrada de datos por parte del usuario. Éste, en cualquier momento, puede realizar alguna acción sobre algún elemento de la interfaz (por ejemplo, pulsar con el ratón sobre un botón, introducir un carácter por teclado, etcétera).

Para poder atender las acciones realizadas sobre los elementos de la interfaz gráfica de usuario, es necesario asociar código a los eventos que se puedan generar como consecuencia de dichas acciones. De esta manera, si se asocia código a un evento concreto, éste se ejecutará cuando se realice la acción que genera el evento sobre un elemento de la interfaz. Al código asociado a los eventos se le denomina **código de gestión de eventos**.

A la hora de programar una aplicación, dependiendo si ésta es de consola o con interfaz gráfica, dadas las diferencias existentes entre ellas (existencia o no de elementos gráficos y tratamiento de eventos), los pasos a seguir serán bastante distintos.

### **1.1.2. Elementos gráficos**

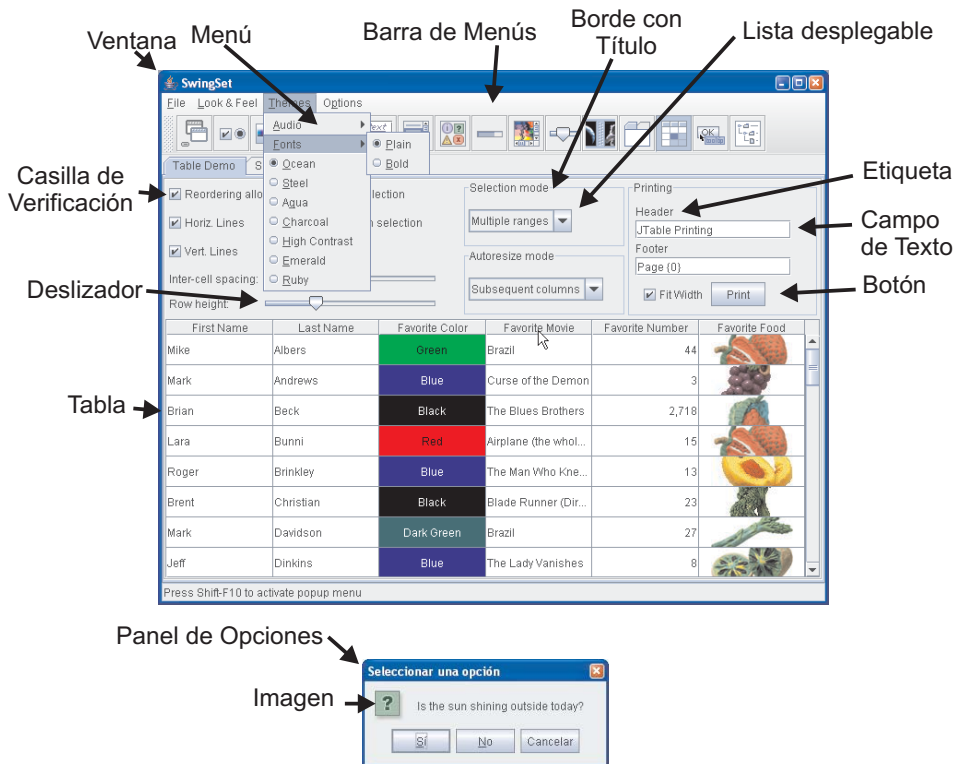
En las aplicaciones de consola, básicamente lo que se le puede mostrar al usuario son cadenas de caracteres, pudiéndose definir, en algunos casos, la posición donde se desea que éstas aparezcan. Sin embargo, en las aplicaciones con interfaz gráfica de usuario, se puede hacer uso de un conjunto de elementos gráficos que permiten una mejor interacción del usuario con la aplicación.

El JDK (*Java Development Kit*) de *Sun Microsystems*<sup>1</sup> incluye una aplicación de demostración, llamada **SwingSet2**, en la que se pueden ver distintos elementos gráficos. Muestra diversas configuraciones de cada uno de ellos junto con el código fuente necesario para construirlos. Si la instalación del JDK se realiza para incluir las demostraciones, la aplicación se encontrará en el directorio <JDK\_DIR>\demo\jfc\SwingSet2. Para ejecutarla basta con hacer doble clic en el fichero `SwingSet2.jar` o invocar al comando `java -jar SwingSet2.jar`. En la figura 1.1 se presenta un ejemplo de dicha aplicación, indicando cuáles son algunos de los elementos que a continuación se definen.

---

<sup>1</sup> <http://java.sun.com/j2se/1.5.0/>





**Figura 1.1:** Aplicación de ejemplo con interfaz gráfica de usuario

Actualmente, las interfaces gráficas están formadas por ventanas de diferentes tipos que se pueden solapar, mover, cerrar, etc. Dentro de estas ventanas se encuentran otros elementos (botones, etiquetas, campos de texto, imágenes, etc.) que permiten introducir datos y mostrar el estado de la aplicación. El ratón y el teclado permiten manejar los elementos que forman parte de la interfaz. A continuación se describen, de forma general, los distintos elementos que puede tener una interfaz gráfica de usuario.

## VENTANAS

Las ventanas son elementos encargados de albergar a otros y que, generalmente, se pueden mover libremente por la pantalla. Existen diferentes tipos en base a su uso y características.

### Ventanas de aplicación

Las ventanas de aplicación son aquellas que contienen a todos los elementos de una aplicación.

### Cuadros de diálogo

Los cuadros de diálogo son ventanas que, normalmente, se muestran un breve periodo de tiempo en la pantalla. Se suelen utilizar para informar al usuario de alguna situación o pedirle datos en un momento determinado.

De forma habitual, todos los cuadros de diálogo están ligados a una ventana de aplicación, de manera que, cuando la ventana de aplicación se minimiza, éstos también lo hacen.

Al mostrarse en pantalla un cuadro de diálogo, es habitual que se bloqueen todos los elementos de la ventana de aplicación. De esta forma, el usuario sólo puede manejar los elementos del cuadro de diálogo. Por ejemplo, cuando se abre un cuadro de diálogo para seleccionar un fichero de disco, es habitual que no se pueda interactuar con otros elementos de la aplicación hasta que el fichero se haya seleccionado. Si un cuadro de diálogo presenta este funcionamiento se dice que es **modal** (característica que se puede deshabilitar si se considera oportuno).

Es posible que un cuadro de diálogo supere los límites de la ventana de aplicación a la que pertenece; es decir, que si el cuadro de diálogo se mueve o es arrastrado por la pantalla (con el ratón por ejemplo), puede sobrepasar los límites de la ventana.

### **Ventanas internas**

Las ventanas internas son un tipo de ventanas específico. Se suelen utilizar para albergar documentos dentro de la ventana de aplicación o para mostrar cajas de herramientas. Por ejemplo, habitualmente en las aplicaciones de retoque fotográfico, cada una de las imágenes abiertas se muestra dentro de una ventana interna, pudiendo ésta maximizarse o minimizarse.

Una restricción que va unida a este tipo de ventanas es que, a diferencia de los cuadros de diálogo, éstas no pueden sobrepasar los límites de la ventana de aplicación.

### **Ventanas sin marco**

Las ventanas sin marco, como su propio nombre indica, son ventanas que carecen de marco y, por tanto, no tienen ni título ni botones para maximizar, minimizar, etc.


Se suelen mostrar al comienzo de la ejecución de una aplicación cuyo tiempo de inicio sea perceptible. De esta forma, el usuario sabe que la aplicación se está cargando aunque aún no pueda usarla. En algunas ocasiones, en la ventana sin marco se muestra el estado del proceso de inicialización, para ofrecer al usuario una estimación del tiempo restante de dicho proceso.

La funcionalidad aquí descrita para las ventanas es la habitual. Sin embargo, dado que la forma de construir interfaces gráficas se encuentra en constante evolución, pueden producirse cambios al respecto. Un ejemplo de una funcionalidad alternativa son aquellas aplicaciones en las cuales se crea una ventana de aplicación por cada documento abierto, de forma que, mientras no se cierren todos los documentos no se finaliza la ejecución de la aplicación. Otro ejemplo son aplicaciones en las cuales los cuadros de diálogo que muestran herramientas, se ajustan a alguna parte de la ventana de aplicación, dejando de ser cuadros de diálogo.

## COMPONENTES

Todos aquellos elementos de una interfaz gráfica con entidad propia y una funcionalidad asociada son componentes. Por ejemplo: botones, barras de desplazamiento, etiquetas, imágenes, listas desplegables, tablas, árboles, etc. No son componentes, por ejemplo, los colores, las líneas, las letras, los píxeles, etc.

### Controles

Los controles son aquellos componentes que pueden recibir información del usuario cuando éste interactúa con la aplicación mediante el ratón o el teclado. Los más comunes son: botones, barras de desplazamiento, cuadros de texto, etc. 

### Contenedores

Un contenedor es un componente que puede mostrar en su interior otros componentes. A los componentes que no son contenedores se les conoce como **componentes atómicos**. Por ejemplo, hay contenedores que muestran todos los componentes a la vez, otros contenedores muestran algunos de los componentes que contienen dependiendo de la pestaña activa, otros muestran los componentes con barras de desplazamiento, etc. A los contenedores se les suele llamar **paneles**.

## MENÚS

Los menús son elementos que contienen botones distribuidos verticalmente. La pulsación de uno de estos botones abrirá un nuevo menú o bien iniciará alguna acción de la aplicación. Los menús pueden aparecer al pulsar el botón secundario del ratón sobre algunos elementos de la interfaz. Si el contenido del menú depende del elemento pulsado, se denomina **menú contextual**.

## BARRAS DE MENÚS

Las barras de menú suelen aparecer en la parte superior de las ventanas. Se componen de una barra horizontal con botones, que al ser pulsados despliegan verticalmente un menú.

## TOOLTIP

Un tooltip es un mensaje que presenta la descripción de un componente o, simplemente, una ayuda acerca de su utilidad. Aparece cuando el ratón permanece inmóvil sobre dicho componente.

### 1.1.3. Gestión de los elementos gráficos

Una vez vistos algunos de los elementos que pueden formar parte de una interfaz gráfica de usuario, a continuación, se pasan a ver aspectos relacionados con la gestión de los mismos.

## JERARQUÍA DE COMPONENTES

La jerarquía de componentes de una interfaz gráfica muestra las relaciones entre los contenedores y los componentes que contienen. Los contenedores son componentes, por lo tanto, se puede crear una jerarquía de varios niveles. En esta jerarquía también se muestra la relación de la ventana con los componentes que contiene.

## DISTRIBUCIÓN DE COMPONENTES



Los componentes dentro de un contenedor han de distribuirse de alguna forma. El desarrollador puede indicar la posición y tamaño de cada componente en píxeles o puede usar algún tipo de lógica o algoritmo. Por ejemplo, puede organizarlos por filas, por columnas, usando una rejilla, con posición fija o variable, dependiendo del tamaño, etc.

## FOCO

Se dice que el componente que recibe las pulsaciones del teclado tiene el foco. Por ejemplo, si hay varios campos de texto, sólo el campo que tenga el foco, mostrará lo que el usuario está escribiendo. Normalmente, el foco puede cambiar de componente usando el ratón o mediante la tecla de tabulación del teclado.

#### **1.1.4. Gestión de Eventos**

La gestión de los eventos será diferente dependiendo del tipo de aplicación que se desarrolle.

##### **Aplicación de Consola**

Una aplicación de consola consiste en una secuencia de instrucciones que se encuentran en un subprograma o método principal, desde donde se hacen llamadas a otros métodos o subprogramas. Al ejecutar una aplicación de este tipo, se van ejecutando cada una de las instrucciones que la componen; algunas de ellas generarán una salida por pantalla y otras pedirán al usuario que introduzca datos. Las instrucciones que piden datos bloquean la ejecución del programa hasta que el usuario los introduzca. Una vez que se obtienen los datos del usuario, se ejecutan las siguientes instrucciones hasta la última, en la que el programa finaliza su ejecución. En la figura 1.2 se muestra gráficamente cómo es una ejecución de una aplicación de consola.

##### **Aplicación con Interfaz Gráfica de Usuario**

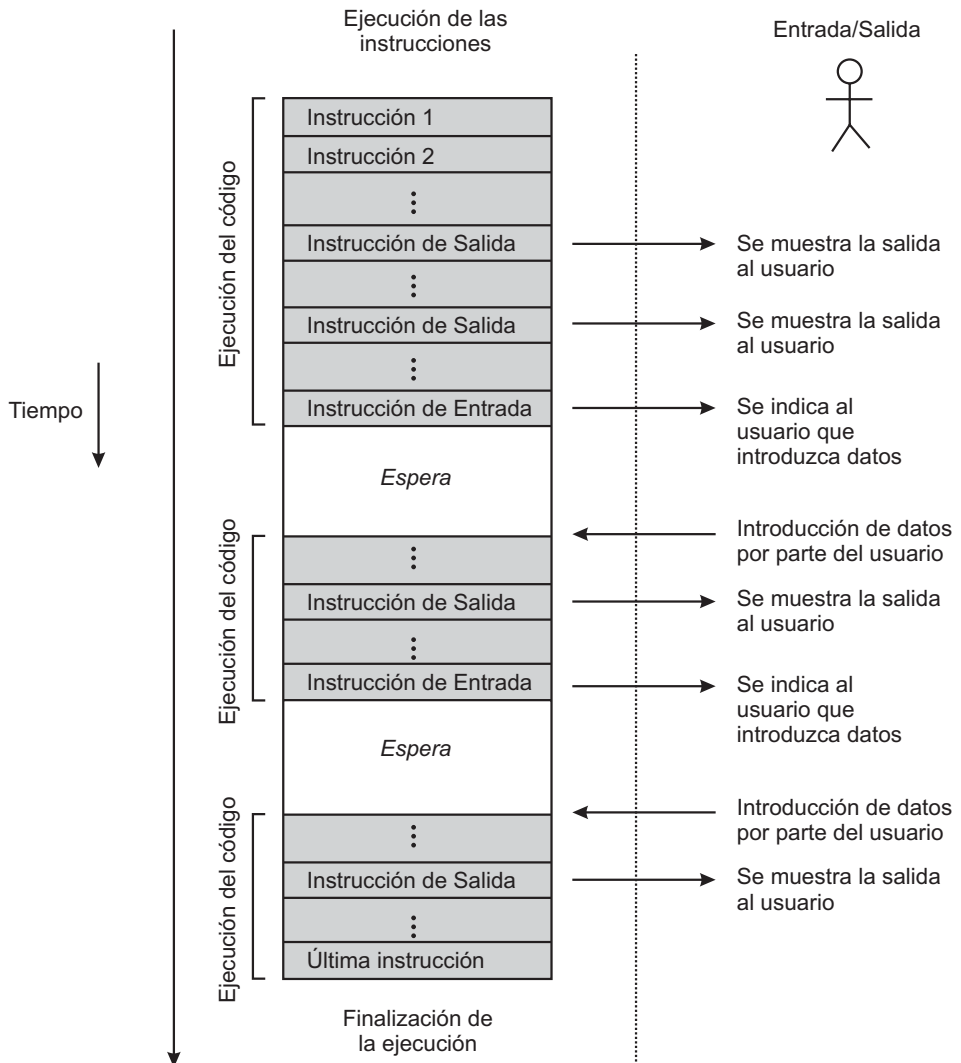
En este caso, el comportamiento en ejecución es muy diferente con respecto a una aplicación de consola. Al ejecutar una aplicación con interfaz gráfica se pueden distinguir, de forma clara, dos partes:

###### **Construcción de la interfaz inicial**

La primera parte se encarga de construir la interfaz gráfica inicial. Se determinan los elementos gráficos que formarán parte de dicha interfaz, su contenido, su organización, etc. Por ejemplo, se indicarán los diferentes botones que se necesiten, la configuración de la ventana de aplicación, los menús, etc.

También se debe establecer la asociación del código de gestión de eventos a los componentes que los puedan producir. De esta manera, se indica, por ejemplo, qué código será ejecutado cuando el usuario pulse sobre un determinado botón, seleccione una determinada opción de un menú, etc. Es decir, se decide qué código se ejecuta para cada evento que se pueda producir.

## Aplicación de consola



**Figura 1.2:** Ejecución de una aplicación de consola

Hasta aquí, la aplicación no muestra nada en pantalla. No se presenta ninguna interfaz visible y, por lo tanto, el usuario no podrá interactuar con la aplicación. Cuando la interfaz gráfica de usuario inicial está totalmente construida, se hace visible ejecutando una instrucción especial.

### **Ejecución del código asociado a los eventos**

Una vez que se muestra la interfaz gráfica, la aplicación se queda a la espera de que el usuario interactúe con ella. Cada vez que éste pulse una tecla, mueva el ratón, etc., se generará un evento en el componente correspondiente. En ese momento, se ejecuta el código asociado a dicho evento (asociación realizada en la etapa anterior). La ejecución de este código suele modificar la interfaz gráfica de usuario para reflejar los resultados de la acción realizada.

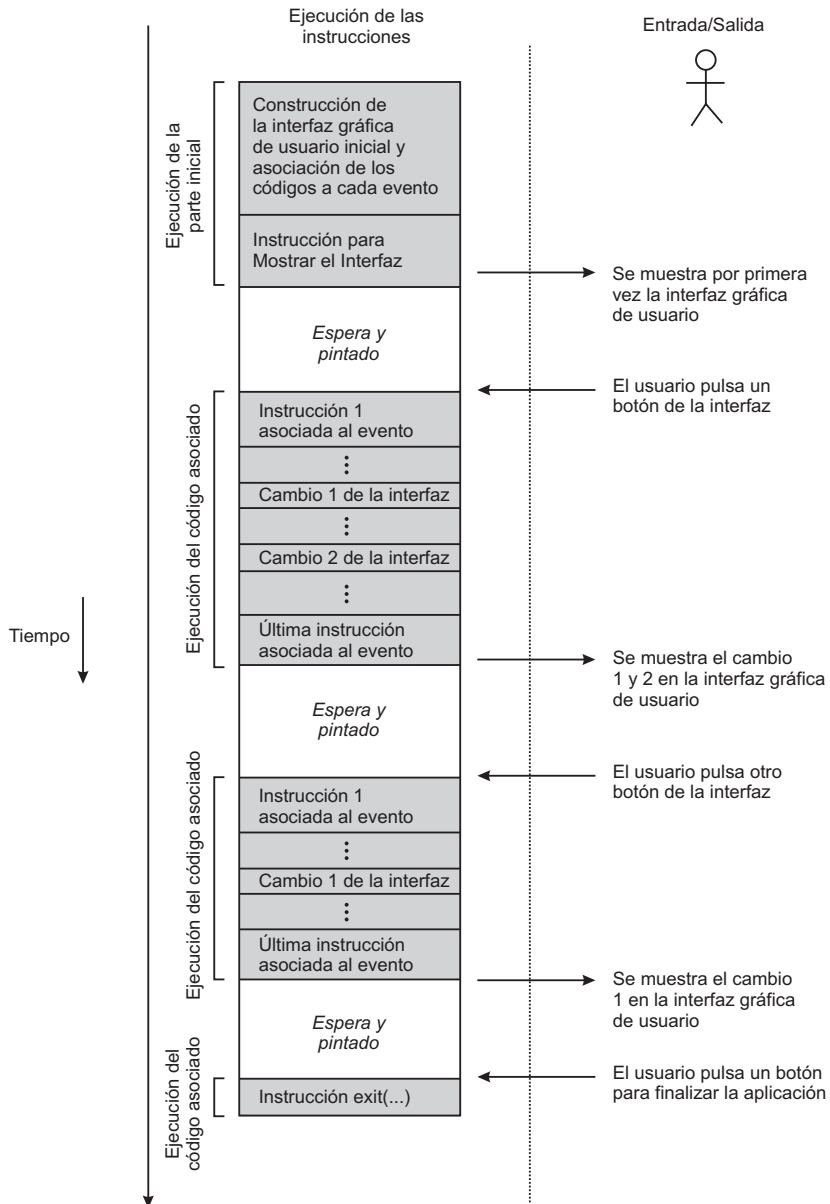
La aplicación queda a la espera de entradas de usuario y modifica la interfaz cuando se haya finalizado la ejecución del código asociado al evento concreto. Esto implica que todo el código asociado a eventos se ejecuta de forma **secuencial**, uno detrás de otro. Por tanto, un código asociado a un evento, deja bloqueada la interfaz gráfica de usuario hasta que termina su ejecución. Durante este tiempo, la interfaz aparecerá “congelada” y la aplicación tendrá aspecto de haberse “colgado”. Más adelante se verán técnicas para evitar esta situación en caso de que sea necesario ejecutar código que tarde un tiempo perceptible en ejecutarse.

Para que finalice la ejecución de una aplicación con interfaz gráfica de usuario es necesario llamar de forma explícita a la instrucción de finalización del programa (normalmente llamada `exit`). También es posible configurar la ventana de aplicación para que, al cerrarse, se finalice la ejecución de la aplicación, algo muy común en este tipo de aplicaciones.

En la figura 1.3 se muestra de forma gráfica la ejecución de una aplicación con interfaz gráfica de usuario.



### Aplicación con interfaz gráfica de usuario



**Figura 1.3:** Ejecución de una aplicación con interfaz gráfica de usuario

## 1.2. Interfaces Gráficas de Usuario en Java

La construcción de interfaces gráficas de usuario ha evolucionado mucho desde la primera versión de Java (Java 1.0). En las primeras versiones se utilizaba la librería **AWT** (*Abstract Windows Toolkit*), la cual no era muy potente. Posteriormente, se creó una librería estándar mucho más completa, llamada **Swing**. Por motivos de compatibilidad y reutilización, la librería Swing sigue utilizando, para su funcionamiento, parte de la librería AWT; por lo tanto, será necesario conocer la librería Swing y parte de la librería AWT.

Como se ha mencionado en el prólogo de este libro, lo que se expone aquí no pretende ser una referencia completa de todos y cada uno de los detalles de la librería Swing. Se ha preferido dar una visión general sobre la construcción de aplicaciones con interfaces gráficas de usuario en Java, los conceptos fundamentales y los primeros pasos. Esta decisión se ha tomado teniendo en cuenta que el grado de detalle en el que un programador necesita conocer un componente concreto, dependerá del uso que vaya a hacer de él. También hay que señalar que la construcción de aplicaciones con interfaz gráfica involucra el conocimiento de muchos y variados conceptos, que no se quieren ocultar entre detalles concretos. Para completar la información presentada en este libro se muestra (tabla 1.1) una relación de direcciones de Internet donde se puede encontrar todo tipo de información sobre Swing.

URL	Descripción
<a href="http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/package-summary.html">http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/package-summary.html</a>	Documentación en formato JavaDoc en línea del paquete javax.swing
<a href="http://java.sun.com/j2se/1.5.0/docs/api/java/awt/package-summary.html">http://java.sun.com/j2se/1.5.0/docs/api/java/awt/package-summary.html</a>	Documentación en formato JavaDoc en línea del paquete java.awt
<a href="http://www.javadesktop.org">http://www.javadesktop.org</a>	Sitio principal donde poder encontrar información sobre Swing: noticias, APIs, artículos, etc...
<a href="http://java.sun.com/products/jfc/tsc/articles/">http://java.sun.com/products/jfc/tsc/articles/</a>	Artículos oficiales sobre Swing y tecnologías relacionadas

<a href="http://java.sun.com/products/jfc/tsc/sightings/">http://java.sun.com/products/jfc/tsc/sightings/</a>	Relación de aplicaciones Java construidas con Swing destacadas por su interfaz gráfica de calidad
<a href="http://java.sun.com/docs/books/tutorial/uiswing/">http://java.sun.com/docs/books/tutorial/uiswing/</a>	Tutorial oficial de Swing y tecnologías relacionadas
<a href="http://www.programacion.com/java/tutorial/swing/">http://www.programacion.com/java/tutorial/swing/</a>	Tutorial oficial de Swing traducido al castellano
<a href="http://www.manning.com/sbe/">http://www.manning.com/sbe/</a>	Libro sobre Swing de la editorial Manning

**Tabla 1.1:** Direcciones de Internet donde encontrar información sobre Swing

En esta sección se verán, de forma somera, algunas de las posibles aplicaciones con interfaz gráfica de usuario que se pueden crear. Para profundizar más sobre ello se remite al lector al capítulo 8.

### ***1.2.1. Tipos de aplicaciones con interfaz gráfica de usuario***

La tecnología Java está muy extendida a día de hoy y permite crear muchos tipos de aplicaciones, con y sin interfaz gráfica. Por ejemplo, aplicaciones de consola, Servlets, Applets, etc.

Las aplicaciones de consola, como ya se ha visto, utilizan la entrada y salida estándar como medio para interactuar con el usuario mediante texto.

Los Servlets son aplicaciones que se ejecutan en un Servidor Web e interactúan con el usuario mediante tecnologías Web (JavaScript, HTML, HTTP, etcétera). Este tipo de aplicaciones no hacen uso de la librería Swing, por ello, no se verán en este libro.

Las aplicaciones con interfaz gráfica de usuario se pueden construir tanto para ordenadores personales como para otro tipo de dispositivos, por ejemplo para los teléfonos móviles, aunque este libro no trata la programación de interfaces gráficas para éstos.

A continuación se presentan, brevemente, los diferentes tipos de aplicaciones que se pueden construir en Java usando Swing: Aplicaciones Autónomas, Applets y Aplicaciones Java Web Start.

### **Aplicaciones autónomas**

Una aplicación autónoma es aquella cuyo funcionamiento es igual a las aplicaciones típicas que se ejecutan en cualquier sistema operativo. Su ejecución se inicia tras pulsar sobre un icono o invocar un comando en la línea de comandos. Habitualmente, cuando se está ejecutando una aplicación de este tipo, aparece en la barra de tareas.

### **Aplicaciones Java Web Start**

Las aplicaciones Java Web Start son aplicaciones muy similares a las aplicaciones autónomas, en lo que se refiere a su funcionamiento durante la ejecución. La diferencia principal es que se pueden cargar desde un servidor Web e instalarse de forma muy cómoda simplemente pulsando un enlace en una página Web.

### **Applets**

Un Applet es una pequeña aplicación Java que se ejecuta dentro de una página Web que está siendo visualizada en un navegador de Internet.

-oOo-

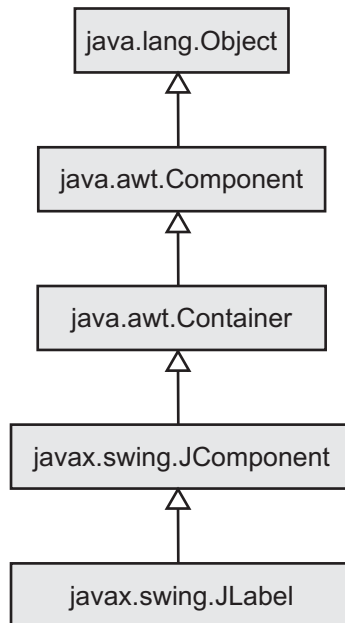
En un primer momento se va a considerar, únicamente, la opción de construir interfaces gráficas de usuario en aplicaciones autónomas, ello se debe a su mayor facilidad de desarrollo y pruebas.

## **1.3. Pasos básicos para la creación de una interfaz gráfica de usuario**

En este apartado se muestran los pasos a seguir en la construcción de una aplicación con interfaz gráfica de usuario. Cada uno de los elementos de una interfaz gráfica de usuario en Java se representa por una instancia de una clase determinada. Por ejemplo, si se quiere que la interfaz de la aplicación tenga un botón, será necesario instanciar un objeto de la clase de ese componente, en este caso: `javax.swing.JButton`.

Para configurar los aspectos gráficos de los componentes se utilizarán los métodos de la clase concreta. Por ejemplo, se podrá cambiar el texto del botón usando el método `void setText(String texto)` de la clase `JButton`.

Todos los componentes de Swing heredan de la clase `javax.swing.JComponent`, la cual hereda de `java.awt.Container` y ésta, a su vez, de `java.awt.Component`. En el capítulo 4 se verán con mayor detalle las relaciones entre las clases de la librería Swing. Por ahora, saber cuáles son las clases padre de todos los componentes permitirá ir conociendo algunas de sus características comunes. En la figura 1.4 se muestra la jerarquía de herencia del componente `JLabel`, que representa una etiqueta. Como puede verse, tanto el paquete `java.awt` como el paquete `javax.swing` contienen clases relativas a la interfaz de usuario, lo que hace patente la relación entre Swing y AWT.

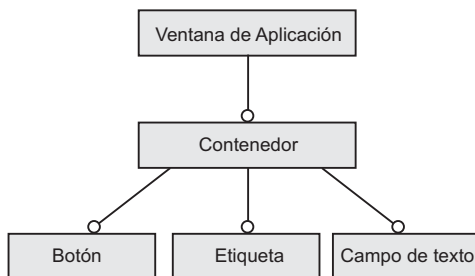


**Figura 1.4:** Jerarquía de herencia del componente `JLabel`

La interfaz gráfica que se va a construir estará formada por una ventana; dentro de ésta aparecerán: un botón, una etiqueta y un cuadro de texto. Los pasos a seguir son los siguientes:

- Crear una ventana de aplicación.
- Crear los componentes que se van a mostrar en dicha ventana.
- Crear un contenedor.
- Asociar los componentes al contenedor para que, al hacerse visible, muestre en su interior dichos componentes.
- Asociar el contenedor a la ventana de aplicación para que, al hacerse visible, muestre en su interior el contenedor y, por tanto, los componentes asociados.
- Hacer visible la ventana.

La jerarquía de componentes de esta sencilla interfaz gráfica se puede ver en la figura 1.5. En ella se indica que un componente va a ser pintado dentro de un determinado contenedor con una línea terminada en círculo. También se muestra igualmente el contenedor que se pinta dentro de la ventana.



**Figura 1.5:** Jerarquía de componentes

Para la creación de la interfaz gráfica anterior será necesario introducir algunos elementos gráficos y eventos concretos, en el capítulo 4 se amplía toda la información al respecto.

### 1.3.1. Creación de una ventana de aplicación

Para crear una ventana de aplicación hay que instanciar un objeto de la clase `javax.swing.JFrame`. Algunos métodos de esta clase relacionados con el aspecto gráfico de la ventana son los siguientes:

- `public JFrame()` - Construye una ventana inicialmente invisible.
- `public JFrame(String title)` - Construye una ventana inicialmente invisible con el título indicado.
- `public void setTitle(String title)` - Establece el título de la ventana.
- `public void setSize(int width, int height)` - Establece el tamaño en píxeles de la ventana.
- `public void setDefaultCloseOperation(int operation)` - Establece la operación que se ha de realizar cuando el usuario cierra la ventana. Los valores permitidos vienen determinados por las siguientes constantes:

`javax.swing.JFrame.EXIT_ON_CLOSE` - Finaliza la ejecución de la aplicación.

`javax.swing.WindowConstants.DO_NOTHING_ON_CLOSE` - No hace nada.

`javax.swing.WindowConstants.HIDE_ON_CLOSE` - Oculta la ventana (por defecto).

`javax.swing.WindowConstants.DISPOSE_ON_CLOSE` - Libera los recursos de la ventana, pero no finaliza la ejecución del programa.

- `public void setResizable(boolean resizable)` - Establece si el usuario puede cambiar el tamaño de la ventana. Por defecto es `true`.
- `public void setExtendedState(int state)` - Establece el estado de la ventana. Puede no funcionar en algunas plataformas. Este método ha de invocarse cuando la ventana sea visible. Los valores permitidos vienen dados por las constantes:

`java.awt.Frame.NORMAL` - No se encuentra ni minimizada ni maximizada.

`java.awt.Frame.ICONIFIED` - Minimizada.

`java.awt.Frame.MAXIMIZED_BOTH` - Maximizada.

`java.awt.Frame.MAXIMIZED_HORIZ` - Maximizada horizontalmente.

`java.awt.Frame.MAXIMIZED_VERT` – Maximizada verticalmente.

- `public void setLocation(int x, int y)` – Establece la posición de la esquina superior izquierda de la ventana.
- `public void setVisible(boolean b)` – Muestra u oculta la ventana.

Cuando ya se ha creado la ventana de aplicación, se procede a crear los componentes que va a contener.

### ***1.3.2. Creación de los componentes de una ventana de aplicación***

Para crear un componente basta con crear una instancia de la clase determinada que represente a dicho componente y configurar ese objeto para que se adapte a las necesidades requeridas para la aplicación particular. Algunos de los métodos relacionados con los aspectos gráficos de algunos componentes muy usados (botones, etiquetas y campos de texto) son los siguientes:

- **JButton**  
`public JButton(String text)` - Crea un botón con el texto indicado.
- **JLabel**  
`public JLabel(String text)` - Crea una etiqueta con el texto indicado.  
`public String getText()` - Devuelve el texto de la etiqueta.  
`public void setText(String text)` - Establece el texto indicado en la etiqueta.
- **JTextField**  
`public JTextField(int columns)` - Crea un campo de texto sencillo con el número de columnas especificado.  
`public String getText()` - Devuelve el texto del campo de texto.

A lo largo del libro se presentarán los componentes más importantes y algunos métodos que permiten configurarlos.

El siguiente paso para crear la interfaz propuesta, una vez se tiene la ventana de aplicación y los componentes que va a contener es crear un contenedor.



### 1.3.3. Creación de un contenedor

En Swing existen muchos tipos de contenedores; sus diferencias radican en la forma en la que manejan los componentes que tienen dentro. Por ejemplo, el `javax.swing.JTabbedPane` es un contenedor con pestañas donde cada una está asociada a un componente. También existe otro contenedor dividido en dos partes, para dos componentes, donde la separación puede cambiar de posición, es el `javax.swing.JSplitPane`.

En este caso, para construir la interfaz propuesta se va a hacer uso del contenedor de propósito general `javax.swing.JPanel`, que es el más sencillo de todos. Puede contener cualquier número de componentes en su interior, los cuales serán mostrados a la vez. La posición y tamaño de los componentes es configurable.

El método constructor de la clase `JPanel` es:

`public JPanel()` – Crea un contenedor de propósito general.

El siguiente paso consiste en asociar los componentes al contenedor creado.

### 1.3.4. Asociación de componentes a un contenedor

Para asociar componentes a un contenedor, de forma que se muestren dentro cuando éste sea visible, se usa el siguiente método de la clase `JPanel` (que hereda de `Container`):

`public void add(Component comp)` – Asocia el componente al contenedor, de forma que se muestre el componente al mostrarse el contenedor. Al usar Swing, como parámetro al método `add(...)` se le pasará un objeto de una clase que herede de `JComponent`.

En este punto, se han creado todos los componentes y contenedores necesarios, con las asociaciones pertinentes, lo único que queda es el último paso: asociar el contenedor a la ventana de aplicación.

### 1.3.5. Asociación de un contenedor a la ventana de aplicación

Para asociar un contenedor a la ventana de aplicación, de forma que se muestre dentro cuando ésta sea visible, se usa el siguiente método de la clase `JFrame`:

`public void setContentPane(Container contentPane)` – Establece el componente pasado como parámetro como contenido de la ventana. Al usar Swing, como parámetro al método `setContentPane(...)` se le pasará un objeto de una clase que herede de `JComponent`.

### 1.3.6. Hacer visible la ventana

Por último, lo único que hace falta es hacer visible la ventana para que la interfaz completa se muestre en la pantalla. Para ello, se usa el método de la clase `JFrame`:

`public void setVisible(boolean b)` – Muestra u oculta la ventana dependiendo del parámetro `visible`.

Para ver cómo se relacionan entre sí todos estos pasos en el ejemplo 1.1 se muestra una aplicación que construye una interfaz de usuario simple.

#### Ejemplo 1.1 – Interfaz de usuario simple

Se pretende construir una aplicación que construya un interfaz de usuario formado por una etiqueta, un campo de texto y un botón. Todo ello en una ventana de 300 x 200 píxeles. La interfaz gráfica de la aplicación se muestra en la figura 1.8.

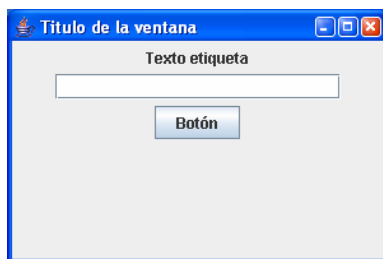


Figura 1.6: Interfaz de usuario simple

El código fuente de la aplicación anterior se muestra a continuación:

**libro/ejemplos/ejemplo1.1/InterfazSimple.java**

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class InterfazSimple {

    public static void main(String[] args) {

        // Crear la ventana de la aplicacion
        JFrame ventana =
            new JFrame("Titulo de la ventana");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

        // Crear los componentes
        JLabel etiq1 = new JLabel("Texto etiqueta");
        JTextField campoDeTexto = new JTextField(20);
        JButton boton = new JButton("Botón");

        // Crear un contenedor
        JPanel panelDeContenido = new JPanel();

        // Asociar los componentes al contenedor para
        // que los muestre en su interior
        panelDeContenido.add(etiq1);
        panelDeContenido.add(campoDeTexto);
        panelDeContenido.add(boton);

        // Asociar el contenedor a la ventana para
        // que le muestre en su interior
        ventana.setContentPane(panelDeContenido);

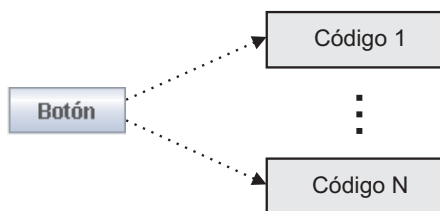
        //Hacer visible la ventana
        ventana.setVisible(true);
    }
}
```

## 1.4. Eventos

Cuando se construye una aplicación con interfaz gráfica de usuario la forma de programar varía con respecto a una aplicación de consola. En una aplicación con interfaz gráfica se asocia código a diferentes eventos que puede producir el usuario cuando interactúa con la aplicación. De manera que, cuando se genera un evento (por ejemplo, pulsar un botón, pulsar una tecla del teclado, pasar el ratón sobre una imagen, etcétera), se ejecuta el código asociado y, habitualmente, se modifica la interfaz gráfica de usuario para mostrar el resultado de esa ejecución. Cuando finaliza la ejecución de ese código asociado, la aplicación espera nuevos eventos por parte del usuario.

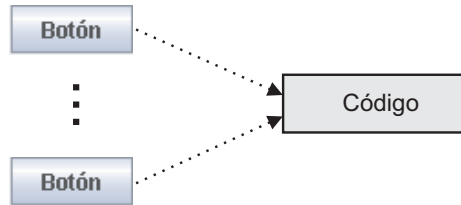
Cada componente de la interfaz gráfica de usuario puede generar varios tipos de eventos. Por ejemplo, un componente de árbol genera eventos de un tipo cuando se selecciona alguno de sus nodos, eventos de otro tipo diferente cuando entra o sale el ratón en él, etc.

Se puede asociar un código a cada uno de los eventos que genera un componente. Es importante tener en cuenta que Swing permite asociar **varios códigos distintos al mismo tipo de evento del mismo componente**. En la figura 1.7 se muestra un esquema indicando esta posibilidad; para ello, se representa mediante una flecha punteada la relación entre un componente que genera eventos y el código asociado.



**Figura 1.7:** Varios códigos asociados a un mismo tipo de evento de un mismo componente

Por otra parte, también está permitido asociar un **mismo código al mismo tipo de evento de varios componentes distintos** (figura 1.8), de forma que, un mismo código será ejecutado cuando se genere ese tipo de evento en cualquiera de los componentes a los que está asociado.



**Figura 1.8:** Un único código de eventos para varios componentes

Por cada tipo de evento que puede ser generado en cualquier componente, existe una interfaz denominada `XXListener`, siendo `xx` el tipo de evento. Por ejemplo, cuando se interactúa de alguna forma con una ventana, se genera un evento de tipo `Window`. Por tanto, existe una interfaz llamada `java.awt.event.WindowListener`. Todas estas interfaces heredan de la interfaz padre `java.util.EventListener`.

Estas interfaces tienen un método por cada una de las acciones concretas que provocan la generación del evento. Por ejemplo, la interfaz `WindowListener` tiene los métodos `windowActivated(...)`, `windowClosed(...)`, `windowClosing(...)`, `windowDeactivated(...)`, `windowDeiconified(...)`, `windowIconified(...)` y `windowOpened(...)`. Cada uno de estos métodos será ejecutado cuando se produzca dicha acción. Por ejemplo, cuando la ventana se abra, se ejecutará el método `windowOpened(...)`.

Cada uno de los métodos de una interfaz `XXListener` tiene un único parámetro llamado `event` de la clase `XXEvent`. De esta forma, los métodos de la interfaz `WindowListener` tienen un único parámetro de la clase `java.awt.event.WindowEvent`. Todas las clases `XXEvent` disponen de métodos que ofrecen información acerca del evento producido. Por ejemplo, la clase `WindowEvent`, tiene el método `int getNewState()` que informa del estado en el que ha quedado la ventana después de producirse el evento. Las clases `XXEvent` heredan de `java.util.EventObject`. Esta clase tiene el método `Object getSource()` que devuelve una referencia al objeto donde se generó el evento.

Por último, cada componente que genere eventos dispone del método `void addXXListener(XXListener listener)`. Este método se utiliza para asociar el código de gestión de eventos del objeto `listener` al compo-

nente. Cuando en el componente se genere un evento de ese tipo, se ejecutará el método correspondiente en el `listener`. Para saber los eventos que se pueden generar en un componente determinado, basta con ver los métodos de la forma `addXXListener(...)` que tiene, tanto en la propia clase como en las clases padre.

Para crear un código de gestión de eventos y asociar dicho código a los eventos que interesen de un componente determinado se deben seguir los pasos siguientes:

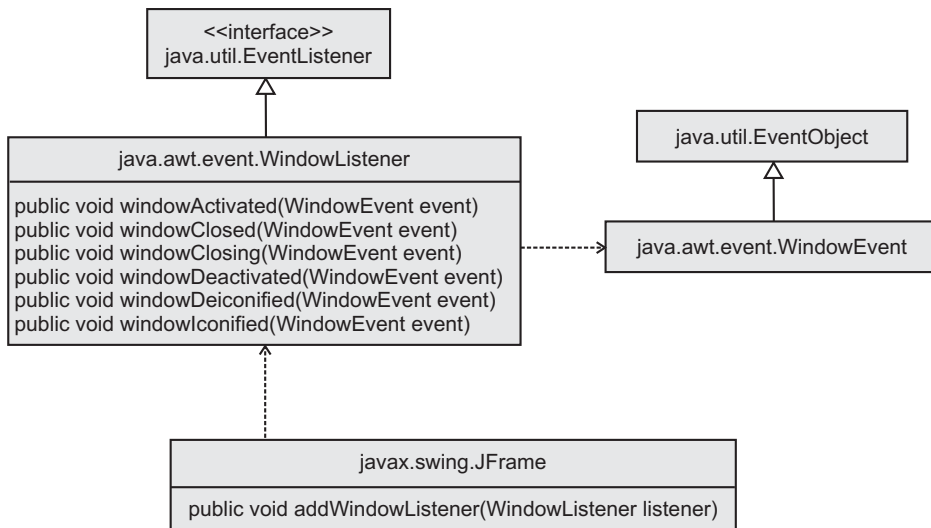
- Crear una clase que implemente una interfaz `XXListener`. Como es lógico, es necesario implementar todos los métodos de la interfaz, no obstante, si no se desea ejecutar nada cuando ocurran determinadas acciones concretas de ese evento, bastará con dejar el cuerpo del método vacío.

Asociar el código anterior al componente o componentes que corresponda. El método void `addXXListener(XXListener listener)` es el que se utiliza para ello.

Por ejemplo, para asociar un código al evento de tipo `Window`, es necesario hacer que una clase implemente la interfaz `WindowListener`, y por tanto implemente sus métodos. Un objeto de esa clase, deberá asociarse a la ventana (objeto de la clase `JFrame`), para ello deberá invocarse el método `addWindowListener(...)`, pasando como parámetro el objeto de la clase que implemente la interfaz `WindowListener`.

En la figura 1.9 se muestran las clases e interfaces que permiten implementar el evento de tipo `Window`.

Los diferentes tipos de eventos se verán en capítulos posteriores. No obstante, se adelanta un tipo de evento muy utilizado: el evento de tipo `Action`. Este evento se genera cuando se pulsa un botón. Al igual que ocurre con el evento de tipo `Window`, visto anteriormente, existe la interfaz `ActionListener`, que tiene el método `actionPerformed` y cuyo único parámetro se llama `event` y es de tipo `ActionEvent`. Además, la clase `JButton` dispone del método `addActionListener(ActionListener listener)` para asociarle los gestores de eventos.



**Figura 1.9:** Clases relacionadas con el evento de tipo Window

### 1.5. La primera aplicación con interfaz gráfica de usuario

En este punto ya se conocen los mecanismos necesarios para construir una interfaz gráfica de usuario. A continuación, se presenta un primer ejemplo de un programa con interfaz gráfica.

Los pasos básicos para implementar un programa con interfaz gráfica de usuario son:

- **Construir la interfaz:**
  - Construir los elementos gráficos.
  - Asociar el código de manejo de eventos a los componentes.
  - Hacer visible la ventana de la aplicación.
- **Código de manejo de eventos:**

Cuando se produce un evento del usuario, se ejecuta el código de manejo de eventos asociado. En este código, habitualmente, se obtienen datos de la interfaz, se procesan esos datos y se modifica la interfaz en consecuencia.

Existen muchas formas de construir el código para cumplir estas directivas generales. Para aplicaciones pequeñas, la forma en que se construya el código de la interfaz gráfica no será relevante, pero en

aplicaciones medias y grandes, la organización de este código será de vital importancia para que sea más fácil de manejar, mantener, reutilizar, etc. Por este motivo, a medida que se vaya avanzando en este libro, no sólo se verá la librería Swing, sino que se prestará especial atención al diseño de la aplicación y no sólo a su funcionalidad.

Para construir una interfaz gráfica de usuario, el código más sencillo se puede conseguir siguiendo los pasos siguientes:

1. Crear una clase.
2. En el método `main` de esa clase se instancia un objeto de dicha clase.
3. Se ponen como atributos cada uno de los componentes de la interfaz gráfica de usuario.
4. El constructor de la clase se encarga de:
  - Construir los componentes.
  - Asociar el código de manejo de eventos a los mismos.
  - Hacer visible la aplicación.
5. La clase implementa la interfaz `XXListener` e implementa sus métodos. En el cuerpo de esos métodos se obtienen los datos de la interfaz gráfica de usuario: a través del objeto evento, a través de la fuente del evento o directamente de otros componentes de la interfaz gráfica. Posteriormente, se procesan esos datos y se modifica la interfaz para mostrar los resultados.

### **Ejemplo 1.2 - Interfaz de un “Conversor de Euros a Pesetas”**

Se pretende construir una interfaz gráfica de usuario que esté formada por una ventana de 300 x 400 píxeles y cuyo título sea “*Conversor de Euros a Pesetas*”. Dentro de la ventana de aplicación se tiene que mostrar una etiqueta con el texto “*Importe en Euros*” y un cuadro de texto de 20 columnas de longitud. También debe aparecer un botón con el texto “*Convertir*” y una etiqueta con el texto “*Pulse para obtener el importe en pesetas*”.

La aplicación debe presentar la siguiente funcionalidad:

- Al cerrar la ventana de aplicación se finalizará la ejecución del programa.



- Cuando se pulse sobre el botón, se debe obtener el valor del campo de texto, hacer la conversión de euros a pesetas y modificar la interfaz gráfica de usuario con el valor de la conversión.

La interfaz gráfica de la aplicación se muestra en la figura 1.10.



**Figura 1.10:** Interfaz gráfica de la aplicación de conversión

El código fuente de la aplicación anterior se muestra a continuación.

**libro/ejemplos/ejemplo1.2/ConversorEuros.java**

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

public class ConversorEuros implements ActionListener {
    private JLabel etiqueta1;
    private JTextField campoDeTexto;
    private JButton boton;
    private JLabel etiqueta2;

    public ConversorEuros() {
        //*****
        //  CREACIÓN DEL INTERFAZ GRÁFICO
        //*****

        // Crear la ventana de la aplicación
        JFrame ventana = new JFrame(
            "Conversor de Euros a pesetas");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

        // Crear los componentes
        etiqueta1 = new JLabel("Importe en Euros");
```

```

campoDeTexto = new JTextField(20);
boton = new JButton("Convertir");
etiqueta2 = new JLabel("Pulse para obtener el "
    + "importe en pesetas");

// Crear un contenedor
JPanel panelDeContenido = new JPanel();

// Configurar el contenedor para mostrar los
// componentes cuando se muestre.
panelDeContenido.add(etiqueta1);
panelDeContenido.add(campoDeTexto);
panelDeContenido.add(boton);
panelDeContenido.add(etiqueta2);

// Configurar la ventana para mostrar el panel
// cuando se muestre
ventana.setContentPane(panelDeContenido);

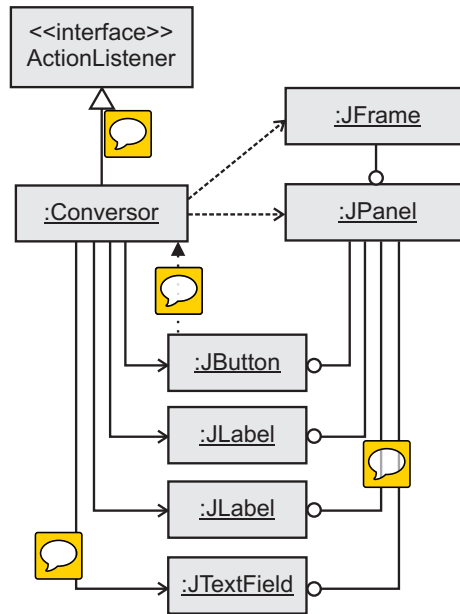
//*****
//  ASOCIACIÓN DEL CÓDIGO DE MANEJO DE EVENTOS
//*****
boton.addActionListener(this);
//*****
//  HACER VISIBLE LA VENTANA
//*****
ventana.setVisible(true);
}

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

public void actionPerformed(ActionEvent e) {
    //*****
    //  CÓDIGO DE MANEJO DE EVENTOS
    //*****
    try {
        double euros = Double.parseDouble(
            campoDeTexto.getText());
        double pesetas = euros * 166.386;
        etiqueta2.setText("Equivale a " + pesetas
            + " pesetas");
    } catch (NumberFormatException e2) {
        etiqueta2.setText(
            "En el campo de texto no hay un número");
    }
}
}

```

De acuerdo a los tipos de relaciones que se pueden dar entre clases y objetos, el esquema de la aplicación del ejemplo 1.2 sería el que se muestra en la figura 1.11.



**Figura 1.11:** Diagrama de clases y objetos de la aplicación de conversión de euros a pesetas (ejemplo 1.2)



## **Capítulo 2**

### **Organización interna de una aplicación con interfaz gráfica**

En el presente capítulo se pretende construir una aplicación con interfaz gráfica de usuario. Esta aplicación se irá construyendo poco a poco; es decir, se van a ir añadiendo de forma paulatina componentes y eventos, con la finalidad de poder experimentar y ver de forma detallada la implementación de estos últimos.

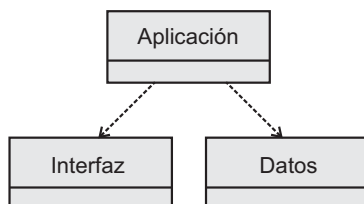
#### **2.1. Introducción**

A la hora de construir cualquier aplicación informática es bueno tener siempre presente la buena modularización y organización del código, lo que redundará en una aplicación fácilmente mantenible, con un código más legible, etc. Esto, además, es especialmente recomendable en aplicaciones con determinadas características, como es el caso de aplicaciones con interfaz gráfica de usuario. Debido, principalmente, a que en estas aplicaciones hay que tener especial cuidado en no acoplar el código de la interfaz gráfica con el resto del código de la aplicación.

El resto del capítulo se estructura en diferentes apartados en los cuales se presenta, en un primer momento, la estructura general de una aplicación con interfaz gráfica. Más adelante se ve en detalle cómo organizar el código de gestión de eventos, así como diferentes partes de la API Swing.

## 2.2. Estructura general. Interfaz y Datos

A la hora de construir una aplicación con un entorno gráfico es muy habitual dividir la estructura principal de la misma en tres clases bien diferenciadas: Aplicación, Interfaz y Datos (no se trata de los nombres de las clases reales de cada aplicación, sino una vista de un diseño conceptual), como se aprecia en la figura 2.1.



**Figura 2.1:** Clases básicas de una aplicación con interfaz gráfica de usuario

La **clase Aplicación** tiene el método estático `main`, encargado de arrancar la aplicación accediendo a los datos necesarios para ello y realizando diversas tareas de inicialización. También se encarga de instanciar un objeto de la clase que representa la interfaz gráfica de usuario, y de establecer la asociación de ésta con el objeto de la clase que alberga los datos o la lógica del programa. En general, en esta clase se suelen hacer todas las tareas de inicialización. Si el tiempo consumido por estas tareas es perceptible por el usuario, es muy recomendable implementar un código para visualizar en pantalla una ventana sin marco que muestre el proceso de carga de la aplicación. En el momento en el que la aplicación esté inicializada y cargada, se muestra la interfaz gráfica de usuario inicial.

La **clase principal de Datos** es una clase que se encarga de todo lo relacionado con la gestión de los datos o la lógica de la aplicación. Normalmente, en aplicaciones sencillas suele ser una única clase la encargada de coordinar toda la lógica de la aplicación, por ejemplo: una clase que gestione la lógica de una calculadora, una agenda sencilla, etc. Como se puede apreciar, esta clase de datos será muy diferente dependiendo de la aplicación concreta y define la funcionalidad de la aplicación con independencia de la interfaz gráfica de usuario. Por otro lado, si

se trata de aplicaciones más complejas, son varias las clases dedicadas a la gestión de los datos o la lógica de la aplicación.

**La clase principal de Interfaz** es la clase encargada de la creación de la interfaz gráfica de usuario y de cómo ésta se relaciona con los datos. Al igual que ocurre con la clase de los datos, si la aplicación es más compleja, serán varias las clases dedicadas a la parte de la creación de la interfaz y su gestión.

Se quiere dejar constancia de que en todo programa con interfaz gráfica debe haber una separación clara entre las clases encargadas de los datos o de la lógica de la aplicación, y las clases encargadas de la gestión de la interfaz. Las clases de la interfaz dependen de las clases de datos, pero las clases de datos deben ser totalmente independientes de la interfaz. Esta es la base que sustenta el patrón de diseño MVC (Model-View-Controller) que se verá en el capítulo 7.

### **Ejemplo 2.1** – Aplicación de traducción 1.0

Se pretende crear una aplicación de traducción entre los idiomas español e inglés. La interfaz gráfica de usuario debe estar formada por: una etiqueta con el texto *“Pulse el botón para traducir”*; un campo de texto de 20 caracteres, que inicialmente estará vacío y un botón con el texto *“Traducir”*. La ventana de aplicación tendrá un tamaño de 300 x 200 píxeles y el título de la misma será: *“Traductor de Español a Inglés”*.

La aplicación debe presentar la siguiente funcionalidad:

- Al cerrar la ventana de aplicación se finalizará la ejecución del programa.
- Cuando se pulse el botón *“Traducir”*, el contenido del campo de texto deberá ser traducido de español a inglés, y el resultado de la traducción se mostrará en la etiqueta de texto (sustituyendo, por tanto, el texto inicial que tenía dicha etiqueta).
- Cuando se intente traducir una palabra que no se encuentre en el diccionario del traductor, el resultado de la traducción será la cadena de caracteres *“X”*.

El diccionario de palabras que utilizará la aplicación se muestra en la tabla 2.1.

Español	Inglés	Francés
Largo	Long	Longue
Barato	Cheap	Réduit
Pan	Bread	Pain
Petardo	Bomb	Bombe
Moto	Motorbike	Moto

**Tabla 2.1:** Diccionario de la aplicación de traducción

Para este ejemplo de aplicación se dispone de una clase de datos llamada `Traductor`. Como la aplicación actual es un traductor, los datos a gestionar son, por ejemplo: los idiomas de origen y destino de la traducción, el diccionario de palabras, el proceso de traducción, etc. A continuación se comentan algunos detalles de esta clase de datos:

- Cada idioma vendrá representado por una constante:
  - `Traductor.ESPAÑOL`
  - `Traductor.INGLES`
  - `Traductor.FRANCES`
- Al crearse el traductor se configura con algún idioma origen y destino por defecto. Existen métodos para actualizar y gestionar estos idiomas:
  - `public void setIdiomaOrigen(int idiomaOrigen)`
  - `public void setIdiomaDestino(int idiomaDestino)`
  - `public int getIdiomaOrigen()`
  - `public int getIdiomaDestino()`
  - `public void invierteIdioma()`
- Para traducir se utilizan los siguientes métodos:
  - `public String traducePalabra(String palabra, int idiomaOrigen, int idiomaDestino)`
  - `public String traducePalabra(String palabra)`
- Se puede obtener una representación textual de una constante del idioma con el método estático siguiente:
  - `public static String getCadenaIdioma(int idioma)`



El código fuente de la clase Traductor es:

**libro/ejemplos/ejemplo2.1/Traductor.java**

```
import java.util.StringTokenizer;

public class Traductor {

    public static final int FRANCES = 0;
    public static final int ESPAÑOL = 1;
    public static final int INGLES = 2;

    public static final int NUM_IDIOMAS = 3;

    private int idiomaOrigen;
    private int idiomaDestino;

    private String[][] diccionario = {
        { "Longue", "Rédiut", "Pain", "Bombe", "Moto" },
        { "Largo", "Barato", "Pan", "Petardo", "Moto" },
        { "Long", "Cheap", "Bread", "Bomb", "Motorbike" } };

    private static String[] cadena = { "Francés", "Español",
                                        "Inglés" };

    public Traductor() {
        this.idiomaOrigen = ESPAÑOL;
        this.idiomaDestino = INGLES;
    }

    public static String getCadenaIdioma(int codIdioma) {
        return cadena[codIdioma];
    }

    public void setIdiomaOrigen(int idiomaOrigen) {
        this.idiomaOrigen = idiomaOrigen;
    }

    public void setIdiomaDestino(int idiomaDestino) {
        this.idiomaDestino = idiomaDestino;
    }

    public int getIdiomaOrigen() {
        return idiomaOrigen;
    }

    public void invierteIdioma() {
        int aux;
        aux = idiomaOrigen;
        idiomaOrigen = idiomaDestino;
```

```
        idiomaDestino = aux;
    }

    public int getIdiomaDestino() {
        return idiomaDestino;
    }

    public String[] getPalabras(int codIdioma) {
        return diccionario[codIdioma];
    }

    public String traducePalabra(String palabra) {
        return this.traducePalabra(palabra, this.idiomaOrigen,
            this.idiomaDestino);
    }

    public String traducePalabra(String palabra,
        int codOrigen, int codDestino) {
        int i = 0;
        boolean encontrado = false;
        while ((i < diccionario[ESPAÑOL].length) &&
            !encontrado)
        {
            if (palabra.equalsIgnoreCase(
                diccionario[codOrigen][i])) {
                encontrado = true;
            } else {
                i++;
            }
        }
        if (encontrado) {
            return diccionario[codDestino][i];
        } else {
            return "X";
        }
    }

    public String traduceTexto(String texto) {
        return this.traduceTexto(texto, this.idiomaOrigen,
            this.idiomaDestino);
    }

    public String traduceTexto(String texto, int codOrigen,
        int codDestino) {
        StringBuffer traduccion = new StringBuffer();
        StringTokenizer st = new StringTokenizer(texto,
            " \n");

        while (st.hasMoreTokens()) {
            String palabra = st.nextToken();
            traduccion.append(this.traducePalabra(palabra,
```

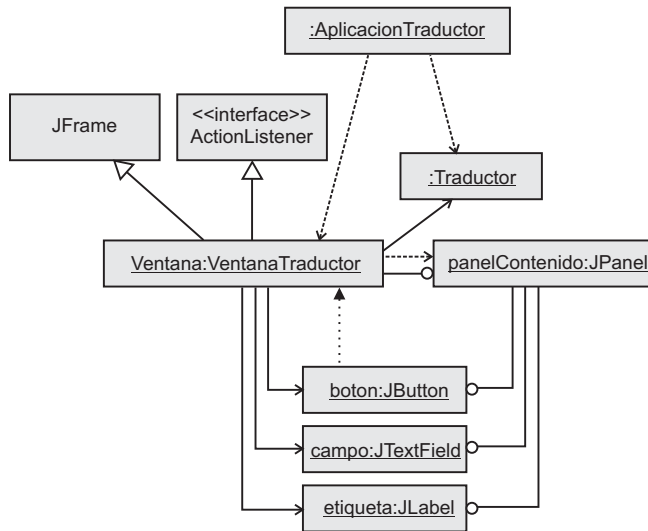
```

        codOrigen, codDestino));
        traduccion.append(" ");
    }

    return traduccion.toString();
}

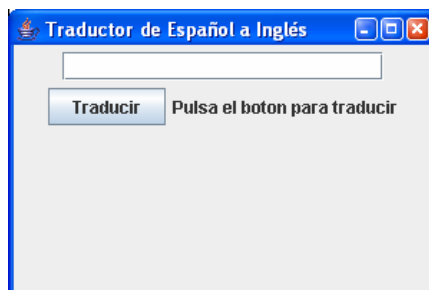
```

El diagrama de clases y objetos de la aplicación se muestra en la figura 2.2.



**Figura 2.2:** Diagrama de clases y objetos de la aplicación de traducción 1.0

La interfaz de la aplicación se presenta en la figura 2.3.



**Figura 2.3:** Interfaz de la aplicación de traducción

El resto del código fuente de la aplicación se muestra a continuación.

**libro/ejemplos/ejemplo2.1/AplicacionTraductor.java**

```
public class AplicacionTraductor {

    public AplicacionTraductor() {
        new VentanaTraductor(new Traductor());
    }

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

**libro/ejemplos/ejemplo2.1/VentanaTraductor.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

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

public class VentanaTraductor extends JFrame implements
    ActionListener {

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;

    public VentanaTraductor(Traductor traductor) {

        this.traductor = traductor;

        this.setTitle("Traductor de Español a Inglés");

        JPanel panelContenido = new JPanel();
        this.setContentPane(panelContenido);

        boton = new JButton("Traducir");
        etiqueta = new JLabel("Pulsa el boton para traducir");
        campo = new JTextField(20);

        boton.addActionListener(this);
    }
}
```

```
        panelContenido.add(campo);
        panelContenido.add(boton);
        panelContenido.add(etiqueta);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(300, 200);
        this.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        etiqueta.setText(traductor.traducePalabra(campo
            .getText()));
    }
}
```

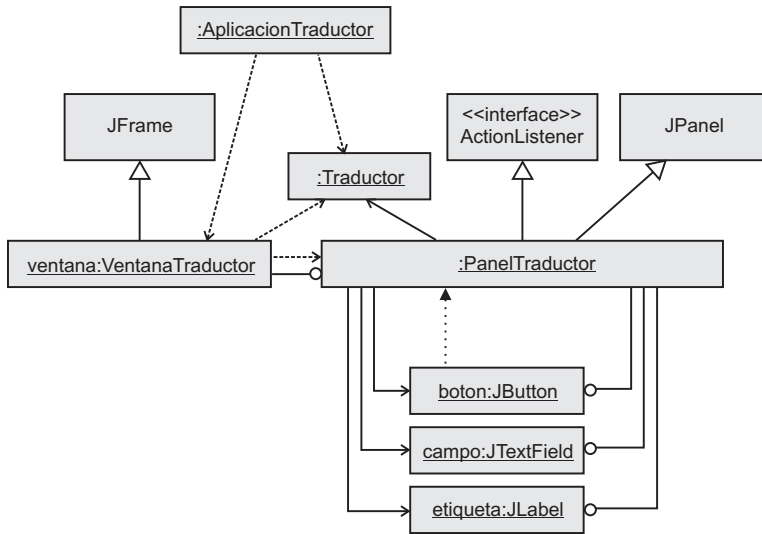
### 2.3. Organización del código de elementos gráficos

Cuando se programa, independientemente del paradigma de programación, del lenguaje y de la aplicación a implementar, algo que siempre se debe tener presente es que **el código ha de ser fácil de mantener y de reutilizar**. Para conseguirlo, una buena idea es separar el código dependiendo de su función; es decir, de forma general, es recomendable separar las sentencias de configuración de la ventana, de las instrucciones de creación y gestión del panel de contenido. Por este motivo, **para el contenido de la ventana se creará una nueva clase**. Esta separación del contenido de la ventana en una clase diferente a la de la propia ventana, permite que la funcionalidad de la aplicación pueda ser incorporada en cualquier parte de la interfaz gráfica de usuario de otra aplicación. En el ejemplo de la aplicación de traducción, al crear una clase para el contenido de la ventana se permite que el traductor se pueda incorporar, por ejemplo, en una aplicación de proceso de textos.

#### **Ejemplo 2.2 – Aplicación de traducción 2.0**

Se pretende organizar mejor el código de la versión 1.0 de la aplicación de traducción (ejemplo 2.1); por ello, se va a modificar dicho código para que la creación y gestión de los componentes de la ventana se realice en una clase distinta a la encargada de la gestión de la propia ventana.

El diagrama de clases y objetos de esta nueva aplicación se muestra en la figura 2.4.



**Figura 2.4:** Diagrama de clases y objetos de la aplicación de traducción 2.0

La interfaz gráfica no cambia con respecto a la de la versión 1.0.

La clase `AplicacionTraductor` queda igual que en la versión del traductor anterior. Sin embargo, la clase `VentanaTraductor` cambia y se crea una nueva, llamada `PanelTraductor`. El nuevo código fuente se muestra a continuación.

#### libro/ejemplos/ejemplo2.2/VentanaTraductor.java

```

import javax.swing.JFrame;

public class VentanaTraductor extends JFrame {
    public VentanaTraductor(Traductor traductor) {

        this.setContentPane(new PanelTraductor(traductor));
        this.setTitle("Traductor de Español a Inglés");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(300, 200);
        this.setVisible(true);

    }
}

```

**libro/ejemplos/ejemplo2.2/PanelTraductor.java**

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class PanelTraductor extends JPanel implements ActionListener
{

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;

    public PanelTraductor(Traductor traductor) {

        this.traductor = traductor;

        boton = new JButton("Traducir");
        etiqueta = new JLabel("Pulsa el boton para traducir");
        campo = new JTextField(20);

        boton.addActionListener(this);

        this.add(campo);
        this.add(boton);
        this.add(etiqueta);

    }

    public void actionPerformed(ActionEvent e) {
        etiqueta.setText(traductor.traducePalabra(campo
            .getText()));
    }

}

```

En los ejemplos mostrados siempre se construye un panel de contenido instanciando un objeto de la clase `JPanel`. No obstante, cuando se instancia un objeto de la clase `JFrame`, se está instanciando de forma automática un panel de contenido, el cual se puede obtener mediante el método `public Container getContentPane()`. Este método devuelve un objeto de la clase `Container`, pero el panel construido automáticamente es una instancia de la clase `JPanel`; por tanto, puede hacerse el *casting* sin

problemas. Además, desde la versión de Java 5.0, se puede usar el método `add(...)` de la clase `JFrame` para asociar los componentes directamente al panel de contenido de la ventana.

## 2.4. Organización del código de eventos

Se ha determinado la estructura general de la aplicación y se ha separado la configuración de la ventana con respecto a la gestión de los componentes que componen su panel de contenido; en definitiva, se ha visto cómo organizar el código de los elementos gráficos. Recordando los pasos básicos de la creación de una aplicación con interfaz gráfica de usuario, vistos en el capítulo anterior, la primera parte se refería a la creación de la interfaz y la segunda a la gestión de eventos. Por lo tanto, una vez organizado el código relativo a la creación de la interfaz, es necesario organizar el código de gestión de eventos.

Una de las características que diferencia a la API Swing de Java de otras APIs de construcción de GUIs de otros lenguajes o sistemas, es la potencia de configuración y adaptación del código de gestión de eventos. No obstante, para aprovechar esta potencia, es necesario organizar cuidadosamente la gestión de eventos. En este apartado se van a establecer las pautas a seguir cuando se implementa código de gestión de eventos, para que éste sea mantenible, reutilizable, eficiente, etc.

### 2.4.1. *Varios componentes generan eventos para la misma acción*

Un mismo código de manejo de eventos se puede ejecutar cuando se produce un evento en diferentes componentes; para ello, simplemente hay que asociar el gestor de eventos a todos los componentes que producen eventos. Sobre la aplicación de traducción 2.0, se va a añadir nueva funcionalidad para mostrar la posibilidad de asociar un mismo código de manejo de eventos a varios componentes.

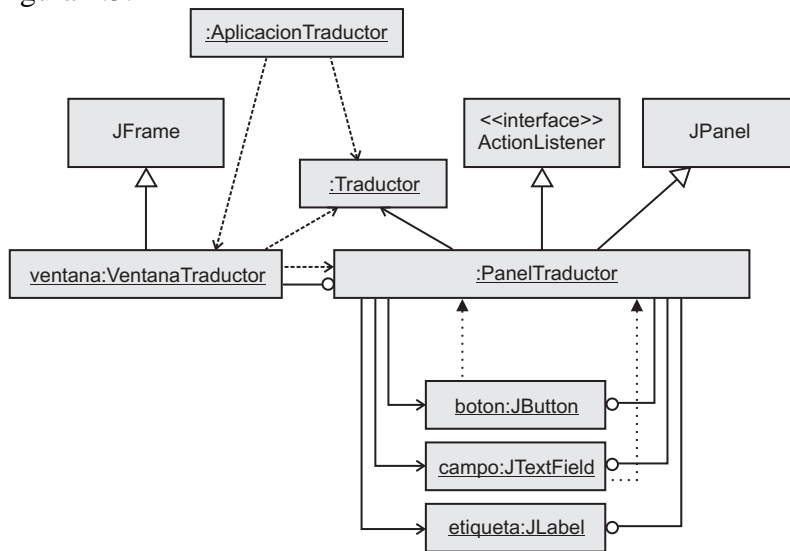
#### **Ejemplo 2.3** – Aplicación de traducción 3.0

Se pretende ampliar la aplicación de traducción 2.0 para que traduzca cuando en el campo de texto se pulse la tecla `ENTER` y el foco se encuentre en él. Hay que recordar que el componente de campo de texto genera un



evento de tipo `Action` cuando esto ocurre. Entonces, lo único que hay que hacer es asociar el código de gestión de eventos, además de al botón, al componente de texto.

El diagrama de clases y objetos de esta nueva aplicación se presenta en la figura 2.5.



**Figura 2.5:** Diagrama de clases y objetos de la aplicación de traducción 3.0

La interfaz gráfica de la aplicación no cambia con respecto a la de la versión 2.0.

Recapitulando, en esta nueva aplicación se tiene, relacionado con la gestión de eventos, lo siguiente:

- Dos componentes que generan eventos: un botón y un campo de texto.
- Un tipo de evento: `Action`.
- Una acción asociada a los eventos: traducción.
- Un objeto encargado de la gestión de eventos: `PanelTraductor`.

El código fuente de la aplicación es muy parecido al código de la versión anterior, tan sólo hay que añadir una sentencia (se muestra en

negrita) para asociar el código de eventos al componente de texto. A continuación se muestra el nuevo código.

**libro/ejemplos/ejemplo2.3/PanelTraductor.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class PanelTraductor extends JPanel implements ActionListener
{
    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;

    public PanelTraductor(Traductor traductor) {

        this.traductor = traductor;

        boton = new JButton("Traducir");
        etiqueta = new JLabel("Pulsa el boton para traducir");
        campo = new JTextField(20);

        boton.addActionListener(this);

        //Campo de texto asociado al gestor de eventos
        //del boton
        campo.addActionListener(this);

        this.add(campo);
        this.add(boton);
        this.add(etiqueta);
    }

    public void actionPerformed(ActionEvent e) {
        etiqueta.setText(traductor.traducePalabra(campo
            .getText()));
    }
}
```

### **2.4.2. Varios componentes generan eventos del mismo tipo para acciones distintas**

En el ejemplo anterior se ha visto que un código de gestión de eventos será ejecutado cuando se produzca un evento en un botón o en un campo de texto. No obstante, independientemente del componente que genere el evento, la acción es la misma: traducir. Sin embargo, lo más habitual es que dependiendo del componente que genera el evento, se realice una acción u otra. Por este motivo, desde el código de gestión de eventos es necesario determinar cuál ha sido el componente que generó el evento. Para ello se utiliza el método `public Object getSource()` declarado en la clase `java.util.EventObject`, clase padre de todos los eventos.

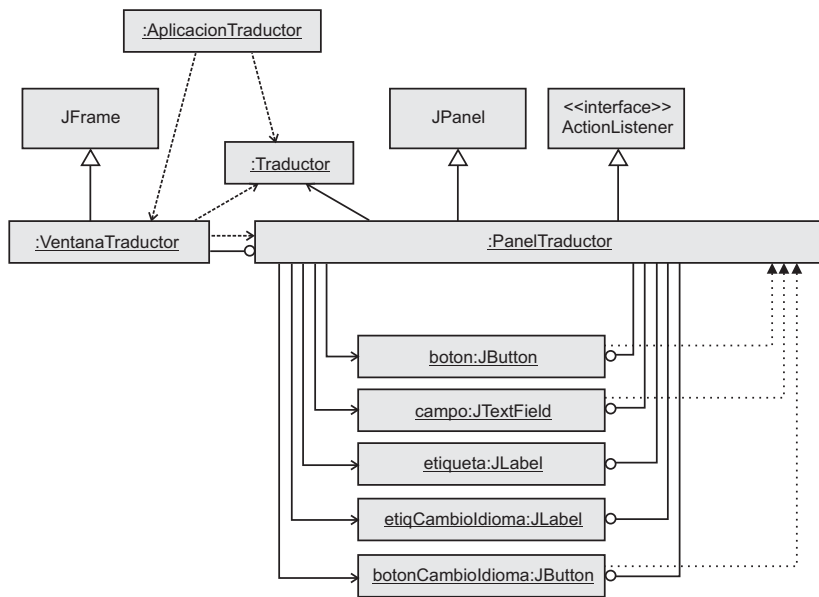
Para mostrar cómo gestionar acciones diferentes se va a ampliar la funcionalidad de la aplicación de traducción añadiendo más componentes y nuevas acciones.

#### **Ejemplo 2.4 – Aplicación de traducción 4.0**

Sobre la aplicación 3.0 del traductor, se va a añadir un botón que permita cambiar el sentido de la traducción y una etiqueta que muestre el sentido actual. Con “sentido de la traducción” se quiere expresar cuál es el idioma origen y destino. Por ejemplo, si el idioma origen es “español” y el idioma destino es “inglés”, al cambiar el sentido de la traducción, entonces “inglés” pasará a ser el idioma origen y “español” el destino.

La aplicación debe usar el idioma origen y destino establecidos en el objeto `Traductor` recibido como parámetro. La clase `Traductor` dispone de métodos para consultar estos valores. Además, todas las etiquetas que muestren el sentido de la traducción, deberán generarse dinámicamente usando los métodos de la clase `Traductor`, para obtener la cadena de caracteres correspondiente a cada idioma. Esto permite que la interfaz gráfica de usuario sea totalmente funcional independientemente de los idiomas del traductor; por tanto, la clase `VentanaTraductor` y la clase `PanelTraductor` serán reutilizables para cualquier idioma origen y destino, sin necesidad de adaptarlas para cada caso.

El diagrama de clases y objetos de esta la aplicación de traducción 4.0 se presenta en la figura 2.6.



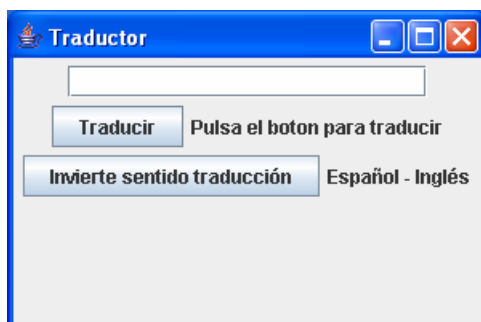
**Figura 2.6:** Diagrama de clases y objetos de la aplicación de traducción 4.0

Recapitulando, en esta aplicación, con respecto a la gestión de eventos, se tiene lo siguiente:

- Tres componentes que generan eventos: dos botones y un campo de texto.
- Un tipo de eventos: `Action`.
- Dos acciones asociadas a los eventos: traducción y cambio de idioma.
- Un objeto encargado de la gestión de eventos: `PanelTraductor`.

La interfaz gráfica de usuario de la aplicación se presenta en la figura 2.7.

Las clases que cambian respecto a la versión anterior del traductor son: `VentanaTraductor`, que ya no muestra el sentido de la traducción y `PanelTraductor`.



**Figura 2.7:** Interfaz gráfica de la aplicación de traducción 4.0

**libro/ejemplos/ejemplo2.4/VentanaTraductor.java**

```
import javax.swing.JFrame;

public class VentanaTraductor extends JFrame {

    public VentanaTraductor(Traductor traductor) {

        this.setContentPane(new PanelTraductor(traductor));

        this.setTitle("Traductor");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(300, 200);
        this.setVisible(true);

    }
}
```

**libro/ejemplos/ejemplo2.4/PanelTraductor.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PanelTraductor extends JPanel implements
    ActionListener {

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;
```

```
private JButton botonCambioIdioma;
private JLabel etiquetaIdioma;

public PanelTraductor(Traductor traductor) {

    this.traductor = traductor;

    etiqueta = new JLabel("Pulsa el boton para traducir");
    campo = new JTextField(20);
    boton = new JButton("Traducir");
    botonCambioIdioma = new JButton(
        "Invierte sentido traducción");
    etiquetaIdioma = new JLabel();

    this.muestraSentidoTraduccion();

    boton.addActionListener(this);
    campo.addActionListener(this);
    botonCambioIdioma.addActionListener(this);

    this.add(campo);
    this.add(boton);
    this.add(etiqueta);
    this.add(botonCambioIdioma);
    this.add(etiquetaIdioma);

}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == botonCambioIdioma) {
        traductor.invierteIdioma();
        this.muestraSentidoTraduccion();
    } else {
        etiqueta.setText(traductor.traducePalabra(
            campo.getText()));
    }
}

private void muestraSentidoTraduccion() {
    etiquetaIdioma.setText(Traductor
        .getCadenaIdioma(traductor.getIdiomaOrigen())
        + " - "
        + Traductor.getCadenaIdioma(
            traductor.getIdiomaDestino()));
}
}
```

Como se puede ver en el código, el método `getSource()` permite conocer el componente que genera el evento y tomar una acción u otra

dependiendo de ello. También es interesante destacar el método `muestraSentidoTraduccion()`, éste es invocado siempre que sea necesario representar el sentido de la traducción en la interfaz gráfica de usuario. El sentido de la traducción queda almacenado en el traductor (clase encargada de la gestión de datos). De nuevo, se aprecia una separación clara de la funcionalidad de cada una de las partes de la aplicación: la clase encargada de los datos mantiene los datos y las clases encargadas de la interfaz gráfica sólo se encargan de mostrarlos.

### ***2.4.3. Varios componentes generan eventos de distinto tipo***

En los apartados anteriores se ha visto cómo varios componentes pueden generar eventos tanto para una misma acción, como para acciones diferentes. Pero siempre se trataba del mismo tipo de evento, el evento `Action`. También es posible que el mismo gestor de eventos atienda varios tipos de eventos. Para ello, es necesario implementar cada una de las interfaces correspondientes a los diferentes tipos de eventos que se quiera gestionar.

A continuación, se dan algunos detalles de parte de la API Swing y AWT que pueden resultar muy útiles, sobre todo en los primeros pasos en el aprendizaje de la construcción de interfaces gráficas de usuario. La intención es ir adentrándose poco a poco e incitar al lector a que busque en la documentación de dichas APIs, lo cual es prácticamente imprescindible para poder construir interfaces gráficas de usuario en Java.

### **Aprendiendo la API. Colores en interfaces gráficas de usuario**

Para representar un color en Java ha de usarse la clase `java.awt.Color`, donde algunas de sus características son:

- Los colores se representan por la cantidad de rojo, azul y verde que los componen en una mezcla aditiva. Este tipo de mezcla implica que si todos están a su valor máximo, el color resultante es el blanco y si todos están a su valor mínimo, el color es el negro.
- Se puede especificar transparencia en un color: el valor máximo es opaco y el mínimo es totalmente transparente.

- Las cantidades de los tres colores y la transparencia se pueden establecer con valores de tipo `int` (de 0 a 255) o con valores de tipo `float` (de 0.0 a 1.0).
- Al poder representarse los colores con valores de diferentes tipos primitivos, es lógico que existan constructores para crear colores con valores de esos tipos; por tanto, la clase `Color` tiene algunos constructores para usar `float` y otros para usar `int`. También existen constructores para especificar la transparencia.
- Existen constantes en la clase que representan los colores más habituales: `Color.BLACK`, `Color.BLUE`, `Color.CYAN`, `Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE`, `Color.YELLOW`.

Todos los componentes pueden cambiar tanto su color principal como su color de fondo. Cada componente utiliza el color principal y el color de fondo de una forma particular; por ejemplo, el color principal de un botón (`JButton`) es el color del texto y el color de fondo es el color del propio botón; las etiquetas (`JLabel`) usan el color principal para el texto pero ignoran el color de fondo; etc. En la clase `Component`, clase padre de todos los componentes, existen los siguientes métodos para gestionar estos colores:

- `public void setBackground(Color c)`
- `public Color getBackground()`
- `public void setForeground(Color c)`
- `public Color getForeground()`

## Aprendiendo la API. Clase `ComponentEvent`

Algunos eventos heredan de la clase `java.awt.ComponentEvent`. Esta clase tiene el método `Component getComponent()`, que devuelve el mismo objeto que el método `Object getSource()`. Entonces, cabe preguntarse ¿por qué existen dos métodos que realmente devuelven el mismo objeto pero están declarados para devolver tipos diferentes? Esta práctica es muy habitual en programación orientada a objetos cuando se hace uso de la herencia. Como se verá más adelante, los eventos pueden ser generados por objetos que no sean componentes de la interfaz gráfica; por eso, la clase padre de todos los eventos, `EventObject`, dispone del método `Object getSource()`. No obstante, los eventos que heredan de `ComponentEvent`



`tEvent` sólo pueden ser generados por componentes, por este motivo, dicha clase dispone del método `Component GetComponent()`, el cual es usado para que los usuarios de la API conozcan este hecho y no se use el *casting*, haciendo las aplicaciones más seguras.

## Aprendiendo la API. Evento de tipo Mouse

Hasta ahora hemos estado usando el evento de tipo `Action`. Como veremos en el capítulo 4, existen muchos tipos de eventos en la librería `Swing` que son generados ante diversas situaciones en los distintos componentes. No obstante, vamos a introducir aquí el evento que permite gestionar el ratón.

El evento de tipo `Mouse` es un evento que se genera cuando el ratón es manejado. Más concretamente, se genera un evento de este tipo cuando el ratón entra en un componente, cuando sale del mismo, cuando se presiona algún botón o cuando se suelta. Además, cuando se presiona y suelta el botón dentro del mismo componente, también se genera para indicar que se ha hecho clic dentro del componente. Como ocurre con los demás eventos, existe la interfaz `java.awt.event.MouseListener` que dispone de un método por cada una de las acciones que pueden generarlo:

- `public void mouseClicked(MouseEvent e);`
- `public void mouseReleased(MouseEvent e);`
- `public void mousePressed(MouseEvent e);`
- `public void mouseExited(MouseEvent e);`
- `public void mouseEntered(MouseEvent e);`

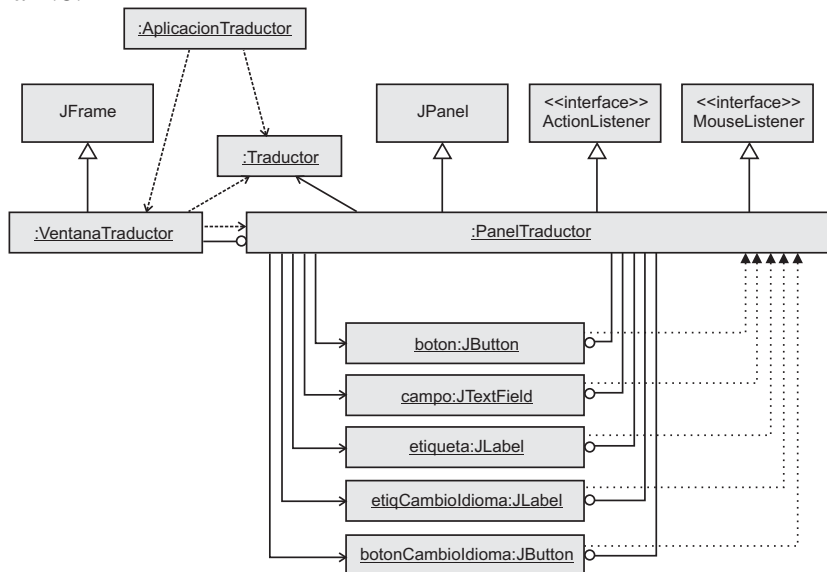
También existe la clase `java.awt.event.MouseEvent`, como se puede ver, la cual contiene información sobre el evento generado. Los eventos de tipo `Mouse` son generados por cualquier componente, por ello, la clase `java.awt.Component`, padre de todos los componentes, dispone del método `void addMouseListener(MouseListener listener)` para asociar gestores del evento de tipo `Mouse` a cualquier componente.

Para mostrar cómo se gestionan los eventos de distintos tipos y poder poner en práctica los nuevos conocimientos sobre las interfaces, se va a continuar trabajando sobre la aplicación de traducción.

### Ejemplo 2.5 – Aplicación de traducción 5.0

Se pretende ampliar la aplicación de traducción 4.0 para que los componentes sean capaces de cambiar su color principal cuando el ratón esté situado sobre ellos, y deben recuperar su color original cuando deje de estarlo. Esta característica, el resaltado de un componente cuando el ratón pasa sobre él, se denomina **rollover**.

El diagrama de clases y objetos para esta aplicación se muestra en la figura 2.8.



**Figura 2.8:** Diagrama de clases y objetos de la aplicación de traducción 5.0

Recapitulando, en esta aplicación, con respecto a la gestión de eventos, se tiene lo siguiente:

- Cinco componentes que generan eventos: dos botones, un campo de texto y dos etiquetas.
- Dos tipos de eventos: `Action` y `Mouse`.

- Tres acciones asociadas a los eventos: traducción, cambio de idioma y rollover.
- Un objeto encargado de la gestión de eventos: `PanelTraductor`.

La interfaz gráfica de usuario no ha variado con respecto a la aplicación de traducción de las versiones anteriores.

La clase que cambia respecto a la versión anterior es la clase `PanelTraductor`, a continuación se muestra su implementación resaltando en negrita los cambios con respecto a la versión anterior.

**libro/ejemplos/ejemplo2.5/PanelTraductor.java**

```
import java.awt.Color;
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PanelTraductor extends JPanel
    implements ActionListener, MouseListener {

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;
    private JButton botonCambioIdioma;
    private JLabel etiquetaIdioma;

    private Color colorAnterior;

    public PanelTraductor(Traductor traductor) {
```

```
this.traductor = traductor;

etiqueta = new JLabel(
    "Pulsa el boton para traducir");
campo = new JTextField(20);
boton = new JButton("Traducir");
botonCambioIdioma =
    new JButton("Invierte sentido traducción");
etiquetaIdioma = new JLabel();
this.muestraSentidoTraduccion();

boton.addActionListener(this);
campo.addActionListener(this);
botonCambioIdioma.addActionListener(this);

etiqueta.addMouseListener(this);
campo.addMouseListener(this);
boton.addMouseListener(this);
botonCambioIdioma.addMouseListener(this);
etiquetaIdioma.addMouseListener(this);

this.add(campo);
this.add(boton);
this.add(etiqueta);
this.add(botonCambioIdioma);
this.add(etiquetaIdioma);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == botonCambioIdioma) {
        traductor.invierteIdioma();
        this.muestraSentidoTraduccion();
    } else {
        etiqueta.setText(traductor
            .traducePalabra(campo.getText()));
    }
}

private void muestraSentidoTraduccion() {
    etiquetaIdioma.setText(
        Traductor.getCadenaIdioma(
            Traductor.getIdiomaOrigen())
        + " - "
        + Traductor.getCadenaIdioma(traductor
            .getIdiomaDestino()));
}

public void mouseClicked(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}

public void mouseExited(MouseEvent e) {
```

```
        Component component = e.getComponent();  
        component.setForeground(colorAnterior);  
    }  
  
    public void mouseEntered(MouseEvent e) {  
        Component component = e.getComponent();  
        colorAnterior = component.getForeground();  
        component.setForeground(java.awt.Color.BLUE);  
    }  
}
```

Observando el código se pueden ver varios métodos vacíos. Se trata de eventos que no interesan para la funcionalidad de la aplicación, por ello sus métodos no se implementan.

#### 2.4.4. Gestores de eventos reutilizables

Si la aplicación que se desarrolla es muy sencilla, la interfaz gráfica de usuario no tendrá muchos componentes y serán pocos los eventos que se generarán. Pero, para aplicaciones más complejas, posiblemente se tendrán que gestionar eventos de muchos componentes. Por lo tanto, es muy posible que en algunas ocasiones se necesite reutilizar el código de gestión de eventos.

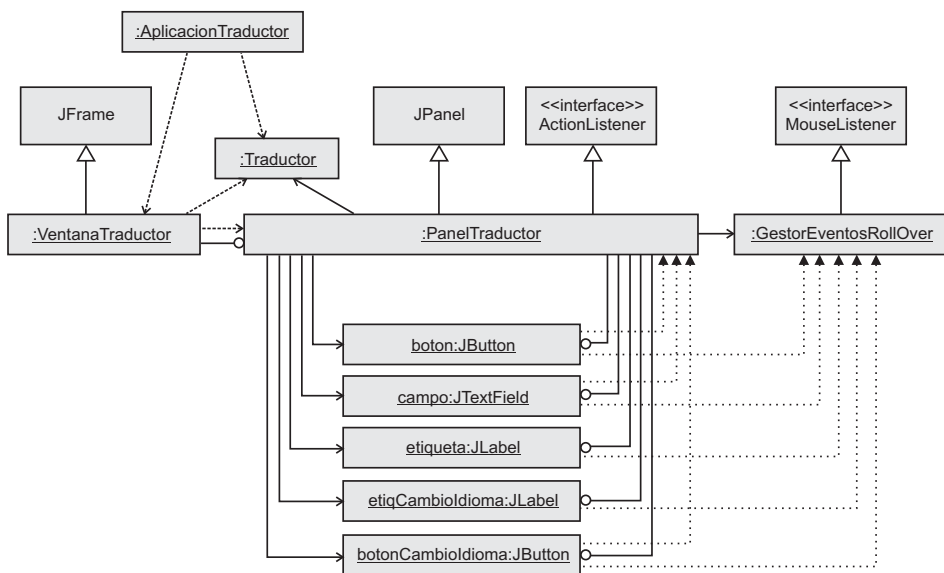
En la aplicación de traducción 5.0 se ha asociado código de eventos para la acción de rollover sobre los componentes. Podría ser interesante disponer de esa funcionalidad en diferentes partes de la aplicación o en diferentes aplicaciones; para ello, se crearía una clase independiente de gestor de eventos de rollover; de forma que podría ser usada en cualquier aplicación. Además, si la lógica de gestión de eventos se hace más compleja, es preferible que aparezca en una única clase desacoplada de otras funcionalidades.

Para mostrar cómo modularizar el código de gestión de eventos de forma que pueda ser reutilizado, se modificará la aplicación de traducción.

### Ejemplo 2.6 – Aplicación de traducción 6.0

En este ejemplo se va a modificar la aplicación de traducción 5.0 de forma que sean varias clases las encargadas de la gestión de eventos. En concreto, se creará una clase encargada de la gestión de eventos de rollover.

El diagrama de clases y objetos para esta aplicación se muestra en la figura 2.9.



**Figura 2.9:** Diagrama de clases y objetos de la aplicación de traducción 6.0

Recapitulando, en esta aplicación, con respecto a la gestión de eventos, se tiene lo siguiente:

- Cinco componentes que generan eventos: dos botones, un campo de texto y dos etiquetas.
- Dos tipos de eventos: `Action` y `Mouse`.
- Tres acciones asociadas a los eventos: traducción, cambio de idioma y rollover.

- Dos objetos encargados de la gestión de eventos: `PanelTraductor` y `GestorEventosRollOver`.

La interfaz gráfica de usuario no ha variado con respecto a la aplicación de traducción de la versión anterior.

En el código fuente se modifica la clase `PanelTraductor` y se crea una nueva clase llamada `GestorEventosRollOver`.

#### **libro/ejemplos/ejemplo2.6/PanelTraductor.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PanelTraductor extends JPanel implements
    ActionListener {

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;
    private JButton botonCambioIdioma;
    private JLabel etiquetaIdioma;

    public PanelTraductor(Traductor traductor) {

        this.traductor = traductor;

        etiqueta = new JLabel("Pulsa el boton para traducir");
        campo = new JTextField(20);
        boton = new JButton("Traducir");
        botonCambioIdioma = new JButton(
            "Invierte sentido traducción");
        etiquetaIdioma = new JLabel();

        this.muestraSentidoTraduccion();

        boton.addActionListener(this);
        campo.addActionListener(this);
        botonCambioIdioma.addActionListener(this);

        GestorEventosRollOver gestor =
            new GestorEventosRollOver();
```

```
        etiqueta.addMouseListener(gestor);
        campo.addMouseListener(gestor);
        boton.addMouseListener(gestor);
        botonCambioIdioma.addMouseListener(gestor);
        etiquetaIdioma.addMouseListener(gestor);

        this.add(campo);
        this.add(boton);
        this.add(etiqueta);
        this.add(botonCambioIdioma);
        this.add(etiquetaIdioma);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == botonCambioIdioma) {
            traductor.invierteIdioma();
            this.muestraSentidoTraduccion();
        } else {
            etiqueta.setText(traductor.traducePalabra(campo
                .getText()));
        }
    }

    private void muestraSentidoTraduccion() {
        etiquetaIdioma.setText(Traductor
            .getCadenaIdioma(traductor.getIdiomaOrigen())
            + " - "
            + Traductor.getCadenaIdioma(
                traductor.getIdiomaDestino()));
    }
}
```

**libro/ejemplos/ejemplo2.6/GestorEventosRollOver.java**

```
import java.awt.Color;
import java.awt.Component;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class GestorEventosRollOver implements MouseListener {

    private Color colorAnterior;

    public void mouseClicked(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}

    public void mouseExited(MouseEvent e) {
        Component comp = e.getComponent();
        comp.setForeground(colorAnterior);
    }
}
```



```
public void mouseEntered(MouseEvent e) {  
    Component comp = e.getComponent();  
    colorAnterior = comp.getForeground();  
    comp.setForeground( java.awt.Color.BLUE );  
}
```

#### 2.4.5. Clases anónimas en la gestión de eventos

En Java existen unas clases especiales que se usan habitualmente en la construcción de interfaces gráficas de usuario, aunque pueden usarse en cualquier parte de la aplicación. Son clases **internas** y **anónimas** y tienen varias peculiaridades que se detallan a continuación.

Las **clases internas** se declaran con la sintaxis habitual de las clases de Java, pero con la particularidad de que su declaración se hace dentro del cuerpo de otra clase, llamada **clase contenedora**, o bien, dentro del cuerpo de sus métodos. Los objetos de las clases internas tienen un atributo oculto, declarado automáticamente, que mantiene una referencia al objeto de la clase contenedora que los instancia. Este atributo oculto permite que desde los métodos de la clase interna se puedan invocar métodos y acceder a los atributos de la clase contenedora de forma directa, sin anteponer la referencia a ningún objeto.

Por otra parte, las **clases anónimas** son clases que, además de ser internas, se declaran con una sintaxis especial. Se llaman anónimas porque no se les da nombre al declararlas y, puesto que no tienen nombre, no se pueden declarar variables ni atributos de estas clases. Por último, su característica más llamativa es que se declaran en la misma sentencia en la que se instancia un objeto de las mismas.

En el siguiente fragmento de código se puede ver la creación de una clase anónima, que implementa la interfaz `ActionListener` y que implementa el método `actionPerformed(...)`. En la misma sentencia se instancia un objeto de dicha clase y se asocia a una variable.


```
...
ActionListener gestorEventos = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Método de la clase anónima");
    }
};
...
```

La sentencia que permite declarar la clase anónima e instanciar un objeto suyo tiene la siguiente sintaxis:

`"new" <RefClaseInterfaz> <ParametrosActuales> <CuerpoClase>`

Cada uno de los elementos de esta definición sintáctica se describe a continuación:

- Mediante `<RefClaseInterfaz>` se indica la clase padre o la interfaz que implementa la clase anónima.
- Con `<ParametrosActuales>` se indican los parámetros que se van a pasar al constructor de la clase anónima. El constructor de la clase no se puede declarar de forma explícita por el programador en el cuerpo de la clase anónima, sino que se construye automáticamente por el compilador siguiendo los siguientes criterios:
  - Si en `<RefClaseInterfaz>` se utiliza una referencia a una interfaz, el constructor que se crea automáticamente será un constructor sin parámetros y cuyo cuerpo estará vacío. Por tanto, en este caso, `<ParametrosActuales>` será una lista vacía de parámetros y la sentencia de creación de la clase anónima será `"new" <RefInterfaz> "(" ")" <CuerpoClase>`.
  - Si en `<RefClaseInterfaz>` se utiliza una referencia a una clase, el constructor que se crea automáticamente será un constructor en cuyo cuerpo sólo habrá una sentencia de llamada a un constructor de la clase padre. El constructor de la clase padre al que se llama, dependerá de los parámetros que se indiquen en `<ParametrosActuales>`; puesto que dichos parámetros son los que se van a pasar al constructor de la clase anónima y, por consiguiente, al constructor de la clase padre. Si la clase padre indicada tiene un constructor sin parámetros, se puede indicar la lista vacía de parámetros. Por lo tanto, la sentencia será: `"new" <RefClase> <ParámetrosActuales> <CuerpoClase>`.

- `<CuerpoClase>` representa el cuerpo de la clase anónima, es como el cuerpo de una clase normal, excepto que **no se pueden declarar constructores** (puesto que éstos son creados de forma automática por el compilador). Al ser una clase interna, dentro del cuerpo de la clase anónima se puede:
  - Invocar métodos de la clase donde aparece esta sentencia; es decir, **invocar métodos de la clase contenedora**. Para ello, basta con poner el nombre del método y los parámetros, sin hacer referencia a ningún objeto.
  - Hacer referencia a atributos de la clase donde aparece esta sentencia; es decir, a los **atributos de la clase contenedora**. Al igual que ocurre con los métodos, para acceder a un atributo basta con poner el nombre del atributo, sin hacer referencia a ningún objeto.
  - Hacer referencia a las variables locales y parámetros del método donde aparece esta sentencia, que estén dentro de su ámbito y declaradas como `final`. Las variables o parámetros `final` se declaran poniendo `final` antes del tipo. La única característica de este tipo de variables o parámetros es que no pueden cambiar de valor.
-  Hacer referencia explícita al objeto de la clase contenedora, por ejemplo si se necesita pasarle como parámetro. Para ello se deberá poner:

```
<RefClaseContenedora> "." "this"
```

A continuación, se muestra el código del ejemplo del conversor de euros a pesetas (presentado en el capítulo 1) implementado mediante una clase anónima, la cual se encarga de invocar el método encargado de realizar la conversión. La clase anónima está resaltada en negrita.

#### libro/ejemplos/ejemplo2.6.1/ConversorEuros

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

// La clase ya no implementa la interfaz ActionListener
public class Conversor {

    private JLabel etiqueta1;
    private JTextField campoDeTexto;
    private JButton boton;
```

```
private JLabel etiqueta2;

public Conversor() {

    //*****
    //  CREACIÓN DEL INTERFAZ GRÁFICO
    //*****

    // Crear la ventana de la aplicacion
    JFrame ventana =
        new JFrame("Conversor de Euros a pesetas");
    ventana.setSize(300, 200);
    ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Crear los componentes
    etiquetal = new JLabel("Importe en Euros");
    campoDeTexto = new JTextField(20);
    boton = new JButton("Convertir");
    etiqueta2 = new JLabel("Pulse para obtener el" +
        "importe en pesetas");

    // Crear un contenedor
    JPanel panelDeContenido = new JPanel();

    // Configurar el contenedor para mostrar los componentes
    // cuando se muestre.
    panelDeContenido.add(etiquetal);
    panelDeContenido.add(campoDeTexto);

    panelDeContenido.add(boton);
    panelDeContenido.add(etiqueta2);

    // Configurar la ventana para mostrar el panel cuando se
    // muestre
    ventana.setContentPane(panelDeContenido);

    //*****
    //  ASOCIACIÓN DEL CÓDIGO DE MANEJO DE EVENTOS
    //*****
    ActionListener gestorEventos = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            convierte();
        }
    };

    boton.addActionListener(gestorEventos);
    ventana.setVisible(true);
}

public void convierte(){
    try {
        double euros =
```

```

        Double.parseDouble(campoDeTexto.getText());
        double pesetas = euros * 166.386;
        etiqueta2.setText(
            "Equivale a " + pesetas + " pesetas");
    }
    catch (NumberFormatException e2) {
        etiqueta2.setText(
            "En el campo de texto no hay un número");
    }
}

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

```

Las clases anónimas no tienen nombre, pero cuando el compilador del JDK de *Sun Microsystems* compila un código Java con clases anónimas, les asigna automáticamente un nombre. Se utiliza el nombre de la clase contenedora seguido del carácter '\$' y un número comenzando en 1. En el ejemplo anterior, la clase anónima se llamará `Conversor$1`.

Las clases anónimas tienen un atributo oculto que mantiene una referencia al objeto de la clase que las contiene (la clase contenedora). En los diagramas de clases y objetos se mostrará la relación entre el objeto de la clase contenedora y el objeto de la clase anónima con el símbolo que se presenta en la figura 2.10.

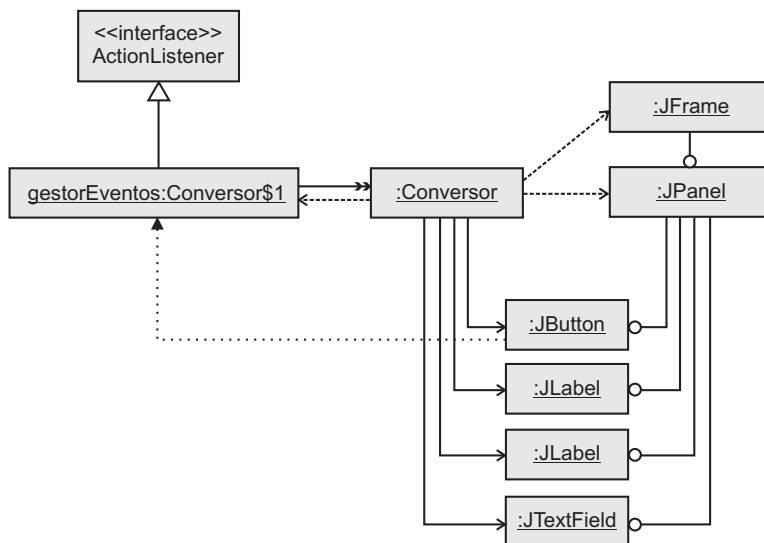


**Figura 2.10:** Relación entre los objetos de la clase anónima y contenedora

Atendiendo a la nueva relación establecida, el diagrama de clases y objetos de la aplicación del conversor de euros a pesetas que hace uso de clases anónimas se puede ver en la figura 2.11.

Las clases anónimas son muy usadas para la gestión de eventos. Esto es debido a que permiten gestionar eventos del mismo tipo generados en componentes distintos, sin necesidad de preguntar el componente que originó el evento y, sin necesidad de que la clase que contiene los componentes tenga que implementar las interfaces de gestión de eventos.

Se suele declarar una clase anónima por cada evento en cada componente. En el cuerpo del método de la clase anónima es conveniente llamar a un método de la clase contenedora en el cual se realice la gestión del evento propiamente dicha.



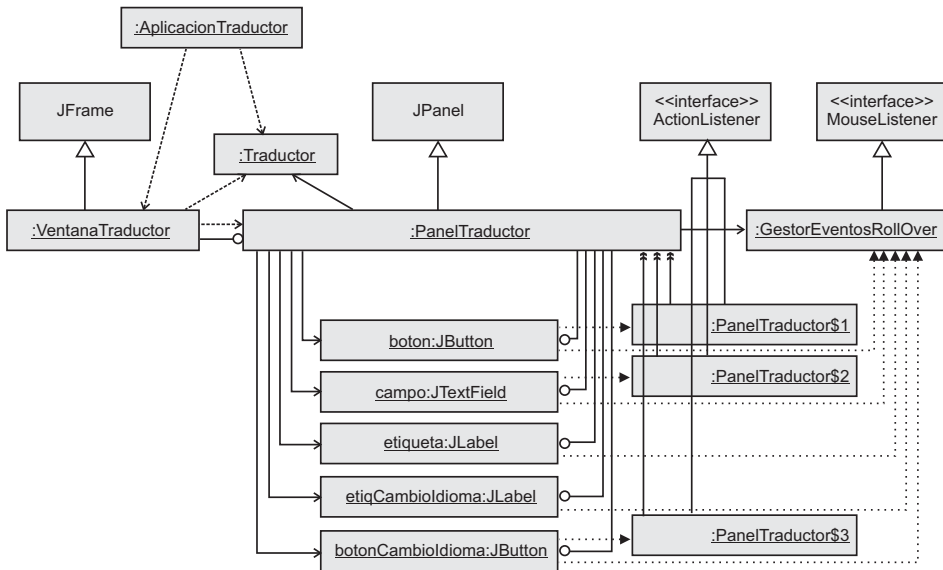
**Figura 2.11:** Diagrama de clases y objetos de la aplicación de conversión de euros a pesetas con clases anónimas

### Ejemplo 2.7 – Aplicación de traducción 7.0

En este ejemplo se modificará la última versión de la aplicación de traducción (6.0), de forma que la gestión de eventos se realice con clases anónimas, excepto la gestión de rollover, que seguirá en una clase diferente.

El diagrama de clases y objetos para esta aplicación se muestra en la figura 2.12.

La interfaz gráfica de usuario no ha variado con respecto a la aplicación de traducción de una versión anterior.



**Figura 2.12:** Diagrama de clases y objetos de la aplicación de traducción 7.0

En el código fuente se modifica la clase `PanelTraductor` que se muestra a continuación:

**libro/ejemplos/ejemplo2.7/PanelTraductor.java**

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PanelTraductor extends JPanel {

    private Traductor traductor;

    private JLabel etiqueta;
    private JTextField campo;
    private JButton boton;
    private JButton botonCambioIdioma;
    private JLabel etiquetaIdioma;

    public PanelTraductor(Traductor traductor) {

        this.traductor = traductor;
    }
  
```

```
etiqueta = new JLabel("Pulsa el boton para traducir");
campo = new JTextField(20);
boton = new JButton("Traducir");
botonCambioIdioma = new JButton(
    "Invierte sentido traducción");
etiquetaIdioma = new JLabel();

this.muestraSentidoTraduccion();

boton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        traduce();
    }
});
campo.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        traduce();
    }
});
botonCambioIdioma
    .addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            invierteSentidoTraduccion();
        }
    });
GestorEventosRollOver gestor =
    new GestorEventosRollOver();

etiqueta.addMouseListener(gestor);
campo.addMouseListener(gestor);
boton.addMouseListener(gestor);
botonCambioIdioma.addMouseListener(gestor);
etiquetaIdioma.addMouseListener(gestor);

this.add(campo);
this.add(boton);
this.add(etiqueta);
this.add(botonCambioIdioma);
this.add(etiquetaIdioma);

}

protected void invierteSentidoTraduccion() {
    traductor.invierteIdioma();
    this.muestraSentidoTraduccion();
}

protected void traduce() {
    etiqueta.setText(traductor.traducePalabra(campo
        .getText()));
}
```



```
private void muestraSentidoTraduccion() {  
    etiquetaIdioma.setText(Traductor  
        .getCadenaIdioma(traductor.getIdiomaOrigen())  
        + " - "  
        + Traductor.getCadenaIdioma(traductor  
            .getIdiomaDestino()));  
}  
}
```

En el código anterior se aprecia el uso habitual de las clases anónimas en la gestión de eventos: se declaran e instancian en la propia llamada al método que asocia el código de gestión de eventos al componente. Otro detalle a tener en cuenta es que está totalmente desaconsejado incorporar métodos con mucha complejidad en las clases anónimas; lo ideal es invocar un método de la clase contenedora para que se realice en él la gestión de eventos propiamente dicha.

#### ***2.4.6. Gestión de eventos cuando el número de componentes es dinámico***

Hasta ahora se ha visto cómo gestionar eventos cuando el número de componentes no varía, siempre son los mismos componentes. Sin embargo, cuando se construyen interfaces gráficas de usuario es habitual que exista un número de componentes dinámico, el cual dependerá de muchos factores; por ejemplo: puede depender del contenido de un array, de una lista de objetos, etc.

Al igual que cuando los componentes son siempre los mismos, al gestionar los eventos producidos en un número variable de componentes es necesario conocer cuál es el componente concreto que genera el evento, para actuar en consecuencia. La gestión de eventos para un número dinámico de componentes se puede implementar de muchas formas, en este apartado se verá una de ellas con clases anónimas.

### Ejemplo 2.8 – Aplicación con número dinámico de componentes

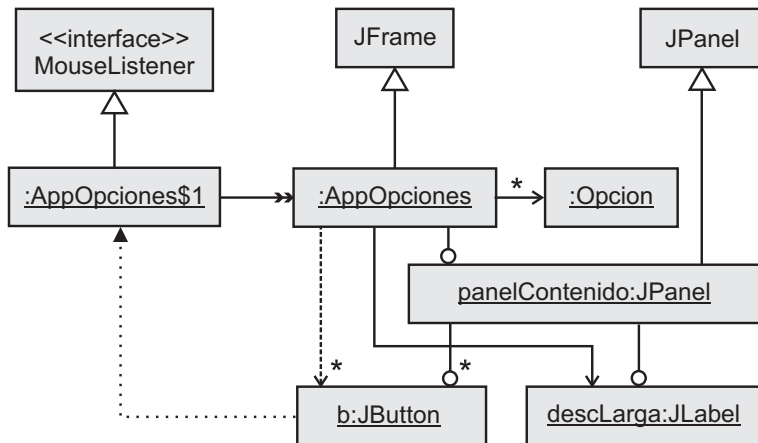
Se pretende construir una interfaz gráfica de usuario que tenga una etiqueta y tantos botones como objetos de la clase `Opcion` existan en una lista.

La clase `Opcion` tiene dos atributos, uno que indica la descripción corta de la opción y otro que indica la descripción larga.

La aplicación debe presentar la siguiente funcionalidad:

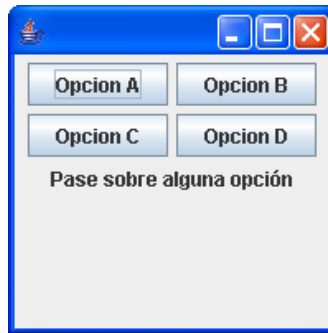
- El texto de los botones deberá mostrar la descripción corta de la opción.
- Cuando se pase con el ratón sobre algún botón, la descripción larga de la opción deberá ser mostrada en una etiqueta.

El diagrama de clases y objetos para esta aplicación se muestra en la figura 2.13.



**Figura 2.13:** Diagrama de clases y objetos de la aplicación con número dinámico de componentes

La interfaz gráfica de usuario de la aplicación se presenta en la figura 2.14.



**Figura 2.14:** Interfaz gráfica de usuario de la aplicación con número dinámico de componentes

Para construir esta interfaz gráfica de usuario, lo lógico es recorrer la lista de opciones e ir creando instancias de objetos de la clase `JButton`. Estos botones se añadirán al panel y se les asociará el código de manejo de eventos.

En el método encargado de establecer la descripción larga en la etiqueta es necesario conocer cuál es la opción sobre la que se pasa para obtener su descripción larga. Para ello, se usa la característica de las clases anónimas que permite acceder a variables locales que estén en el ámbito de la clase anónima, siempre que sean `final`.

El código fuente de la aplicación, incluida la gestión de eventos, se muestra a continuación.

**libro/ejemplos/ejemplo2.8/Opcion.java**

```
public class Opcion {  
  
    private String descCorta;  
    private String descLarga;  
  
    public Opcion(String descCorta, String descLarga) {  
        this.descCorta = descCorta;  
        this.descLarga = descLarga;  
    }  
  
    public String getDescCorta() {  
        return descCorta;  
    }  
}
```

```
public void setDescCorta(String descCorta) {
    this.descCorta = descCorta;
}

public String getDescLarga() {
    return descLarga;
}

public void setDescLarga(String descLarga) {
    this.descLarga = descLarga;
}
}
```

**libro/ejemplos/ejemplo2.8/AppOpciones.java**

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import java.util.List;

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

public class AppOpciones {

    private static final String mensaje =
        "Pase sobre alguna opción";

    private JLabel descLarga;
    private List opciones;

    public AppOpciones() {

        // Datos de prueba
        opciones = new ArrayList();
        opciones.add(
            new Opcion("Opcion A", "Desc Larga Opcion A"));
        opciones.add(
            new Opcion("Opcion B", "Desc Larga Opcion B"));
        opciones.add(
            new Opcion("Opcion C", "Desc Larga Opcion C"));
        opciones.add(
            new Opcion("Opcion D", "Desc Larga Opcion D"));

        // Construcción del interfaz gráfico
        JFrame ventana = new JFrame();
        ventana.
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setSize(200, 200);
    }
}
```

```
JPanel panelContenido = new JPanel();
ventana.setContentPane(panelContenido);

for (int i = 0; i < opciones.size(); i++) {
    Opcion opcion = (Opcion) opciones.get(i);
    JButton b = new JButton(opcion.getDescCorta());
    panelContenido.add(b);

    // Declaración de variable debido a que desde
    // las clases anónimas sólo se pueda acceder a
    // variables locales o parámetros final.
    final int numOpcion = i;

    b.addMouseListener(new MouseListener() {
        public void mouseEntered(MouseEvent e) {
            setDescLarga(numOpcion);
        }

        public void mouseExited(MouseEvent e) {
            resetMensaje();
        }

        public void mouseClicked(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
    });
}

descLarga = new JLabel(mensaje);

panelContenido.add(descLarga);

ventana.setVisible(true);
}

protected void resetMensaje() {
    this.descLarga.setText(mensaje);
}

protected void setDescLarga(int numOpcion) {
    Opcion opcion = (Opcion) opciones.get(numOpcion);
    this.descLarga.setText(opcion.getDescLarga());
}

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

#### ***2.4.7. Conclusiones en la organización del código de gestión de eventos***

Entre las diferentes soluciones que se pueden adoptar a la hora de enfrentarse a la implementación de un código de gestión de eventos, unas presentan ventajas y/o desventajas con respecto a otras. Se intentará en este apartado extraer unas breves conclusiones que ayuden a decidir cuál es la solución más adecuada para cada tipo de problema.

En general, la utilización de clases anónimas es una buena opción siempre que la gestión de eventos esté muy relacionada con la interfaz gráfica y, siempre que la funcionalidad sea sencilla. Sin embargo, cuando la gestión de eventos puede ser reutilizada, como en el caso del rollover, es interesante la creación de una clase independiente y poco acoplada con el resto de la interfaz gráfica. En otras ocasiones, en las que la funcionalidad de los eventos tenga suficiente complejidad, también puede ser conveniente, por modularidad, la creación de una clase diferente, a pesar de que no sea fácilmente reutilizable.

## Capítulo 3

### Organización de los componentes: administradores de distribución y bordes

Una vez que se han visto las diferentes formas de organizar el código de gestión de eventos (capítulo 2), se verán, en el presente capítulo, cómo organizar los elementos gráficos de una interfaz gráfica de usuario.

#### 3.1. Introducción

Una interfaz gráfica de usuario se compone de elementos gráficos, los cuales no se colocan de forma arbitraria en la ventana de aplicación, sino que se organizan de una manera determinada en función del tipo y características de la aplicación concreta. Para gestionar dicha organización de componentes existen los llamados **administradores de distribución**.

En lo que resta del capítulo, a lo largo de varios apartados, se estudian los diferentes administradores de distribución que ofrece Swing, viendo cómo se implementan y sus características más relevantes. También se ven algunas partes de la API Swing y en los dos últimos apartados, se muestran otras alternativas para la organización de los elementos de la interfaz gráfica sin hacer uso de ningún administrador de distribución.

#### 3.2. Administradores de distribución

Los administradores de distribución son objetos que se asocian a los contenedores. Se encargan de decidir el tamaño y posición de los componentes que se pintan en dichos contenedores y también si un

componente debe estar visible o no. Son objetos de clases que implementan la interfaz `java.awt.LayoutManager`.

La mayoría de los contenedores que proporciona Swing tienen un administrador predeterminado, el cual distribuye los componentes de una forma específica para la funcionalidad de ese contenedor y no debe ser cambiado. Por ejemplo, el contenedor `JTabbedPane` coloca cada componente en una pestaña y muestra el componente de la pestaña seleccionada, la funcionalidad es muy concreta y su administrador de distribución predeterminado se ajusta perfectamente a dicha funcionalidad. Sin embargo, con el contenedor `JPanel` el programador puede elegir el administrador de distribución que más se ajuste a sus necesidades para organizar los componentes. Existen varias clases para los administradores de distribución; las instancias de dichas clases se encargan de organizar los componentes de una determinada forma y de asignar tamaños concretos siguiendo unos determinados criterios.

Todos los administradores implementan la interfaz `LayoutManager` o su subinterfaz `LayoutManager2`. Los administradores de distribución que ofrece Swing son los siguientes:

- `java.awt.FlowLayout` (implementa `java.awt.LayoutManager`)
- `java.awt.GridLayout` (implementa `java.awt.LayoutManager`)
- `java.awt.CardLayout` (implementa `java.awt.LayoutManager2`)
- `java.awt.BorderLayout` (implementa `java.awt.LayoutManager2`)
- `javax.swing.BoxLayout` (implementa `java.awt.LayoutManager2`)
- `javax.swing.GridBagLayout` (implementa `java.awt.LayoutManager2`)
- `javax.swing.SpringLayout` (implementa `java.awt.LayoutManager2`)

Está permitido asociar a un contenedor `JPanel` el administrador de distribución nulo (`null`); en ese caso, la distribución de los componentes se denomina **distribución estática**.

La forma en la que un administrador de distribución decide el tamaño y posición de los componentes que se añaden al `JPanel` se basa en los siguientes criterios:

- **Naturaleza del administrador** – Cada administrador de distribución se ha diseñado para distribuir los componentes de manera distinta, más adelante se verán, de forma detallada, los más representativos.



- **Configuración del administrador** – Un administrador de distribución se puede configurar con diversas opciones.
- **Configuración por componente** – Cada componente puede tener asociada una configuración para que el administrador decida dónde colocarlo y cual será su tamaño. Esta configuración se puede establecer de dos formas:
  - **Configuración en el propio componente** – Todos los componentes guardan cierta información que es usada para su distribución. Esta información se crea por defecto al instanciar el componente. Cada administrador de distribución puede usar esta información o puede ignorarla a la hora de posicionar y dimensionar los componentes. Pese a que esta información se crea automáticamente al crear el componente, es posible configurarla mediante métodos de la clase `Component`:

◦ `public void setLocation(int x, int y)` – Establece la posición deseada del componente dentro del contenedor. Es relativa a la esquina superior izquierda.

Y de la clase `JComponent`:

- `public void setMaximumSize(Dimension maximumSize)` – Establece el tamaño máximo deseable.
- `public void setMinimumSize(Dimension minimumSize)` – Establece el tamaño mínimo deseable.
- `public void setPreferredSize(Dimension preferredSize)` – Establece el tamaño preferido. Por tamaño preferido de un componente se entiende aquél que hace que el componente se muestre correctamente. Por ejemplo, el tamaño preferido de un botón será aquél que haga que el texto del botón se vea completo.
- `public void setAlignmentX(float alignmentX)` – Alineación horizontal deseable. En este libro se considera que el valor para este atributo es el valor por defecto.
- `public void setAlignmentY(float alignmentY)` – Alineación vertical deseable. En este libro se considera que el valor para este atributo es el valor por defecto.

- o **Configuración que se establece al asociarle al panel** – También se puede configurar un componente al asociarle al panel con información específica para el administrador que se esté usando. Las opciones que se pueden configurar dependen del administrador de distribución.

Como ya se sabe, un contenedor es un componente; por lo tanto, los contenedores también tienen toda la información mencionada. Por ejemplo, el tamaño preferido de los contenedores lo determina el administrador de distribución y los componentes que tenga en su interior en ese momento.

Si se quiere que la ventana de aplicación muestre a su panel de contenido con **el tamaño preferido**, se usará el método de la clase `Window` (clase abuela de `JFrame`):

```
public void pack()
```

Es importante que este método sea ejecutado después de añadir todos los componentes a la ventana. Lo ideal es que se ejecute justo antes de invocar al método `setVisible(true)`. La razón es porque el tamaño preferido de cada componente sólo puede ser determinado una vez que se hayan establecido todas las propiedades o características gráficas de los mismos. Por ejemplo, el tamaño preferido de un botón depende del texto que tenga dicho botón y el tamaño de ese texto lo decide el usuario al configurarlo.

### 3.2.1. **Implementación de los administradores de distribución**

Para asociar un administrador de distribución a un contenedor `JPanel` se dispone del siguiente método de la clase `Container`:

```
public void setLayout(LayoutManager mgr)
```

A este método se le pasa como parámetro una **nueva** instancia de algún administrador de distribución. Algunos administradores que requieren configuración se configuran en su constructor. La información de configuración asociada al componente se proporciona cuando éste se añade al contenedor, para ello se usa el método de la clase `Container`:

```
void add(Component comp, Object constraints)
```

Cada administrador de distribución utiliza el parámetro `constraints` para recibir la información que considere oportuna. En las siguientes secciones se verá la información de configuración que requiere cada administrador de distribución.

Es importante saber que la distribución de componentes no se realiza de forma automática a medida que se van añadiendo componentes al contenedor, ya que esto sería ineficiente. La distribución de componentes se realiza cuando se va a hacer visible la interfaz gráfica; es decir, cuando se llama al método de la ventana de aplicación `setVisible(true)`. Por consiguiente, si el contenedor ya es visible y se eliminan o añaden componentes, entonces, será necesario indicar al contenedor de forma explícita que redistribuya los componentes de nuevo; por ello, en el contenedor donde se hayan realizado los cambios habrá que invocar el método de la clase `Container`:

```
public void validate()
```

### 3.2.2. Administrador de distribución *FlowLayout*

El administrador de distribución `FlowLayout` es el administrador por defecto del contenedor `JPanel` y, además, es el más sencillo de todos. Lo que hace es ir colocando todos los componentes por filas, de forma que cuando no caben más componentes en la fila actual, crea una nueva fila para ir colocándolos, y así sucesivamente. Creará tantas filas como sean necesarias para que quepan todos los componentes.

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

- **Configuración asociada al componente al añadirle al panel** – No tiene; es decir, se usará el método `add()` sin parámetro de configuración para añadir componentes a un contenedor que tenga este administrador de distribución.

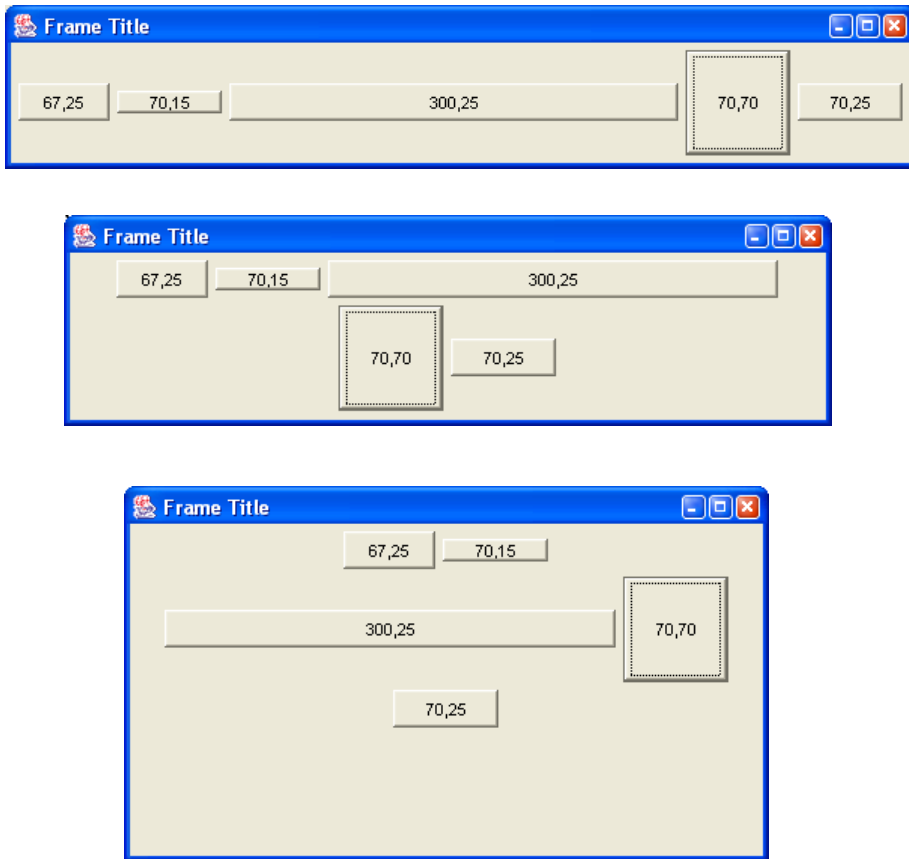
- **Configuración del administrador** – Existe la posibilidad de indicar el espacio entre componentes, por defecto es de 5 píxeles. También se puede indicar la alineación de los componentes en una línea; para ello, se usa alguna de las constantes siguientes: `FlowLayout.CENTER`, `FlowLayout.RIGHT`, o `FlowLayout.LEFT`. Para configurar el administrador de distribución se tienen los siguientes constructores:

```
o public FlowLayout()  
o public FlowLayout(int align)  
o public FlowLayout(int align, int hgap, int vgap)
```

Donde `align` es la alineación, `hgap` es el espacio horizontal y `vgap` es el espacio vertical.

- **Configuración en el propio componente** – La única información del componente que se usa es el tamaño preferido.
- **Tamaño preferido del contenedor** – El tamaño preferido del contenedor es aquel que hace que todos los componentes quepan en una única línea. El contenedor tendrá como altura el tamaño del componente más alto y como anchura la suma de las anchuras preferidas de todos los componentes, además del espacio de separación entre ellos.

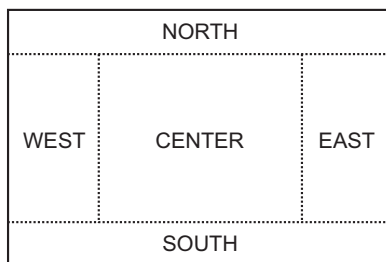
En la figura 3.1 se presentan una misma interfaz gráfica con diferentes tamaños de contenedor para poder observar cómo se distribuyen los componentes en su interior dependiendo de dicho tamaño.



**Figura 3.1:** Interfaz gráfica de ejemplo con administrador de distribución `FlowLayout`

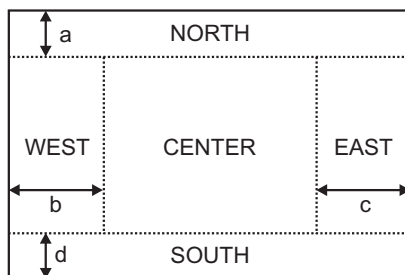
### 3.2.3. *Administrador de distribución `BorderLayout`*

El administrador de distribución `BorderLayout` divide al contenedor `JPanel` en cinco zonas, una para cada componente, como se puede ver en la figura 3.2.



**Figura 3.2:** División de zonas del contenedor por el administrador  
`BorderLayout`

Es posible que alguna de las zonas no tenga componente, pero como máximo este administrador admite cinco componentes. La forma en la que determina el tamaño de cada una de las zonas o regiones se muestra en la figura 3.3.



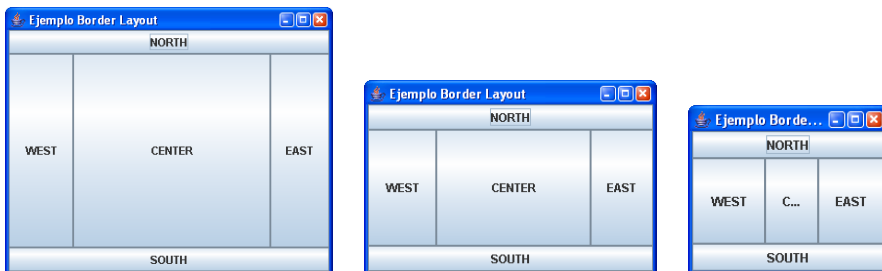
**Figura 3.3:** Asignación de tamaño por regiones en el administrador  
`BorderLayout`

Para las regiones NORTH y SOUTH la altura es la altura preferida del componente en esa región y la anchura es la del contenedor `JPanel`. Para las regiones WEST y EAST la anchura es la anchura preferida del componente en cada región y la altura la del contenedor `JPanel`, menos la de los componentes de NORTH y SOUTH. El tamaño de la región CENTER es el espacio sobrante tanto en horizontal como en vertical.

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

- **Configuración asociada al componente al añadirle al panel** - Para decidir en qué zona del contenedor `JPanel` va el componente, el valor del segundo parámetro del método `add(...)` será alguna de las constantes siguientes:
  - o `BorderLayout.CENTER`
  - o `BorderLayout.WEST`
  - o `BorderLayout.EAST`
  - o `BorderLayout.SOUTH`
  - o `BorderLayout.NORTH`
- **Configuración del administrador** - Se puede especificar el espacio en píxeles entre los componentes. Los constructores de `BorderLayout` son:
  - o `public BorderLayout(int hgap, int vgap)`
  - o `public BorderLayout()`
- **Configuración en el propio componente** – La única información usada es el tamaño preferido de cada componente.
- **Tamaño preferido del contenedor** – Aquel que, además de asignar el espacio a cada zona mencionado anteriormente, deja al componente de la zona `CENTER` con su tamaño preferido.

En la figura 3.4 se presenta la misma interfaz gráfica con diferentes tamaños para mostrar cómo el tamaño de los componentes se ajusta dependiendo del tamaño disponible para el contenedor.



**Figura 3.4:** Interfaz gráfica de ejemplo con administrador de distribución `BorderLayout`

**Ejemplo 3.1**

Como primer ejemplo de este capítulo, se presenta el código fuente correspondiente a la construcción de la interfaz gráfica de la figura 3.4.

**libro/ejemplos/ejemplo3.1/EjemploBorderLayout.java**

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class EjemploBorderLayout {

    private JButton botonNorte;
    private JButton botonSur;
    private JButton botonEste;
    private JButton botonOeste;
    private JButton botonCentro;

    public EjemploBorderLayout() {

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

        this.botonNorte = new JButton("NORTH");
        this.botonSur = new JButton("SOUTH");
        this.botonEste = new JButton("EAST");
        this.botonOeste = new JButton("WEST");
        this.botonCentro = new JButton("CENTER");

        JPanel panelDeContenido = new JPanel();
        panelDeContenido.setLayout(new BorderLayout());
        panelDeContenido.add(
            this.botonNorte, BorderLayout.NORTH);
        panelDeContenido.add(
            this.botonSur, BorderLayout.SOUTH);
        panelDeContenido.add(
            this.botonEste, BorderLayout.EAST);
        panelDeContenido.add(
            this.botonOeste, BorderLayout.WEST);
        panelDeContenido.add(
            this.botonCentro, BorderLayout.CENTER);

        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }
}
```



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

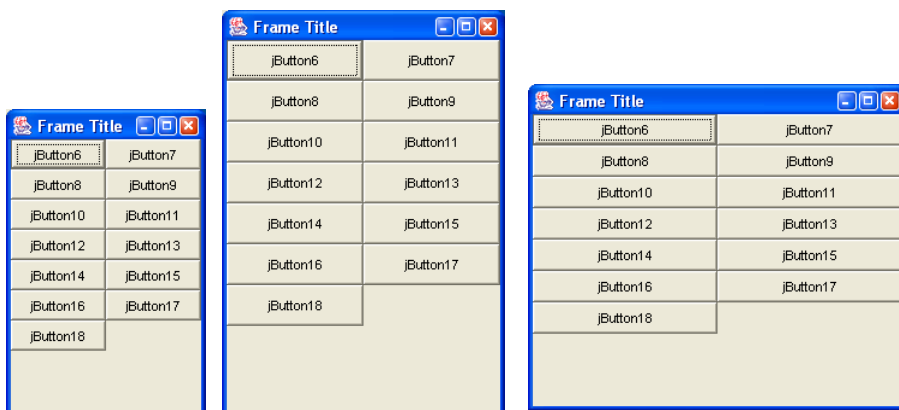
### 3.2.4. *Administrador de distribución GridLayout*

El administrador de distribución `GridLayout` divide el contenedor `JPanel` formando una rejilla de celdas iguales. A medida que se van añadiendo los componentes, se van colocando en cada una de las celdas. El componente ocupa todo el tamaño de la celda, tanto a lo alto como a lo ancho. Las celdas sin componente se mostrarán vacías y, por tanto, se verá el panel de fondo.

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

- **Configuración asociada al componente al añadirle al panel** – No tiene.
- **Configuración del administrador** – Es necesario especificar el número de filas y de columnas de la rejilla. También se puede especificar el espacio en píxeles entre los componentes. Se usarán los siguientes constructores:
  - `public GridLayout(int rows, int cols)`
  - `public GridLayout(int rows, int cols, int hgap, int vgap)`
- **Configuración en el propio componente** – No se usa.
- **Tamaño preferido del contenedor** – Aquel tamaño que considera el tamaño de cada celda como el tamaño preferido más grande de entre todos los componentes.

En la figura 3.5 se presenta la misma interfaz gráfica con diferentes tamaños.



**Figura 3.5:** Interfaz gráfica de ejemplo con administrador de distribución GridLayout

### 3.2.5. *Administrador de distribución GridBagLayout*

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

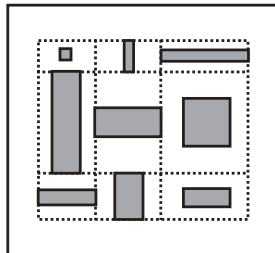
Las características más importantes de este administrador se resumen a continuación:

- **Configuración asociada al componente al añadirle al panel** – Se utiliza para especificar la posición de la celda en la que incluir el componente, el número de celdas del área de visualización, la alineación, etc. Toda esta configuración se establece mediante objetos de la clase `java.awt.GridBagConstraints`.
- **Configuración del administrador** – No tiene. Se usa el constructor sin parámetros.
- **Configuración en el propio componente** – El tamaño mínimo y el tamaño preferido.

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

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

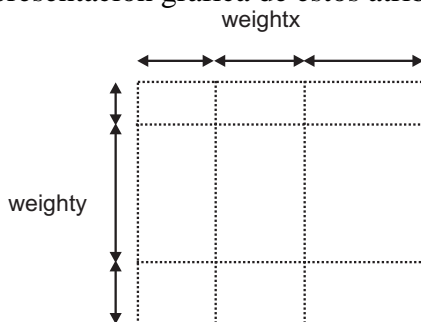
- **Número de filas y columnas de la tabla** – No es necesario indicar las filas y columnas de la rejilla al construir el administrador como ocurre con `GridLayout`. Al añadir cada componente al `JPanel` se indica la fila y la columna de la celda en las que debe estar y el número de celdas que ocupa (tanto de ancho como de alto). Con esta información se determina el número de filas y columnas de la rejilla.
- **Tamaño de las celdas y de la rejilla** – Existen unos atributos públicos de la clase `GridBagConstraints` que controlan el tamaño de cada una de las celdas de la rejilla. Con sus valores por defecto la anchura de cada columna es la anchura preferida más grande de los componentes de esa columna. La altura de cada fila es la altura preferida más grande de los componentes de esa fila. Y la rejilla aparecerá centrada dentro del contenedor como se puede apreciar en la figura 3.6. Cuando el contenedor es más pequeño de su tamaño preferido, se utilizan la altura y anchura mínimas, en lugar de las preferidas.



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

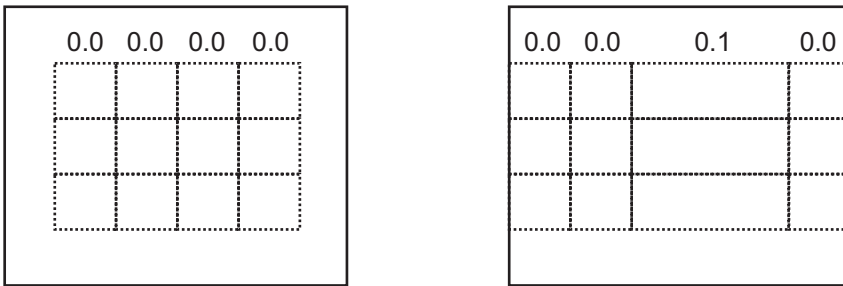
A continuación se presentan y describen los diferentes atributos que se pueden configurar en una instancia de la clase `GridBagConstraints`, y cómo se deben usar para configurar la posición y tamaño de los componentes.

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



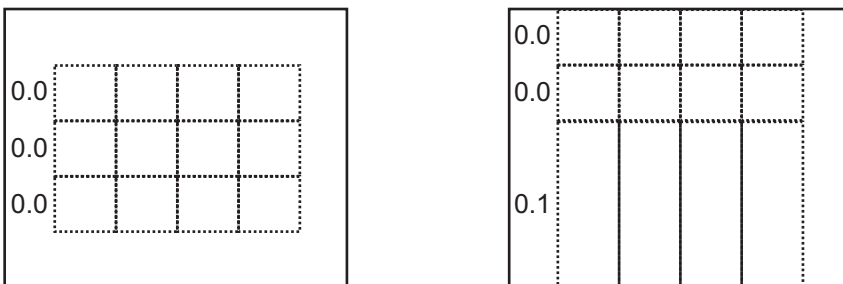
**Figura 3.7:** Atributos de pesos que influyen en el tamaño de las filas y columnas de la rejilla en un `JPanel` con administrador de distribución `GridBagLayout`

Estos atributos pueden tomar valores entre 0.0 y 1.0. Cuando su valor es 0.0 (valor por defecto) la rejilla aparecerá centrada en el `JPanel`. Si el valor del atributo `weightx` de alguna columna es mayor que 0.0, la rejilla tendrá la anchura del `JPanel`. Todas las columnas tendrán el tamaño preferido de sus componentes excepto la columna con el valor mayor que 0.0, que ocupará todo el espacio **sobrante**. Esto se refleja de forma gráfica en la figura 3.8.



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

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



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

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

una columna es diferente que el tamaño preferido de otra, aunque tengan el mismo peso, nunca serán iguales.

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

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

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

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

`GridBagConstraints.REMAINDER` - indica que el área de visualización abarca hasta el final de la fila (si al atributo `gridwidth` se le asigna como valor esta constante) o hasta el final de la columna (si al atributo `gridheight` se le asigna como valor esta constante).

`GridBagConstraints.RELATIVE` - indica que el área de visualización abarca hasta el siguiente componente de la fila (si al atributo `gridwidth` se le asigna como valor esta constante) o de la columna (si al atributo `gridheight` se le asigna como valor esta constante).

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

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

`GridBagConstraints.NORTH`

`GridBagConstraints.SOUTH`

`GridBagConstraints.WEST`

`GridBagConstraints.EAST`

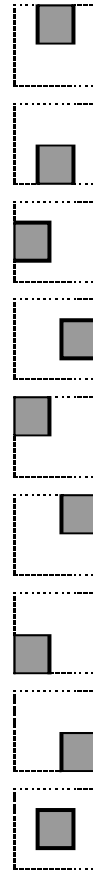
`GridBagConstraints.NORTHWEST`

`GridBagConstraints.NORTHEAST`

`GridBagConstraints.SOUTHWEST`

`GridBagConstraints.SOUTHEAST`

`GridBagConstraints.CENTER`



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

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

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

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



○ `GridBagConstraints.HORIZONTAL` - indica que la anchura será como la anchura del área de visualización y su altura será la preferida.



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



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



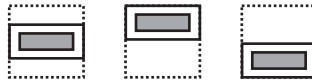
El valor por defecto del atributo `fill` es `GridBagConstraints.NONE`.

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



ño del espacio en cada lado y el constructor `public Insets(int top, int left, int bottom, int right)`.

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



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

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

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

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

```
...
JPanel panelContenido = new JPanel();
GridBagLayout administrador = new GridBagLayout();
panelContenido.setLayout(administrador);

GridBagConstraints config = new GridBagConstraints();

config.gridx = ...;
config.gridy = ...;
config. ...

panelContenido.add(componente, config);
```

```
GridBagConstraints config2 = new GridBagConstraints();  
  
config2.gridx = ...;  
config2.gridy = ...;  
config2. ...  
  
panelContenido.add(componente2, config2);  
...
```

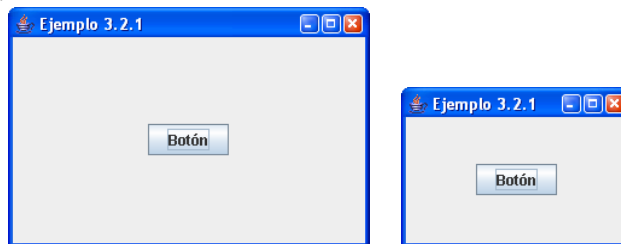
A continuación, se presentan varios ejemplos con el fin de mostrar cómo construir interfaces gráficas de usuario con las opciones estudiadas para posicionar y dimensionar un componente dentro de su área de visualización.

### **Ejemplo 3.2**

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

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

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



**Figura 3.11:** Interfaz gráfica de usuario del ejemplo 3.2.1

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.2.1/Ejemplo321.java**

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ejemplo321 {

    public Ejemplo321() {

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

        JButton boton = new JButton("Botón");
        JPanel panelDeContenido = new JPanel();

        panelDeContenido.setLayout(new GridBagLayout());
        GridBagConstraints config = new GridBagConstraints();
        config.weightx = 1.0;
        config.weighty = 1.0;
        panelDeContenido.add(boton, config);

        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

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

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

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

public class Ejemplo321 {

    public Ejemplo321() {

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

```

        JButton boton = new JButton("Botón");

        JPanel panelDeContenido = new JPanel();

        panelDeContenido.setLayout(new GridBagLayout());

        GridBagConstraints config =
            new GridBagConstraints();
        config.weightx = 1.0;
        config.weighty = 1.0;
        panelDeContenido.add(boton, config);

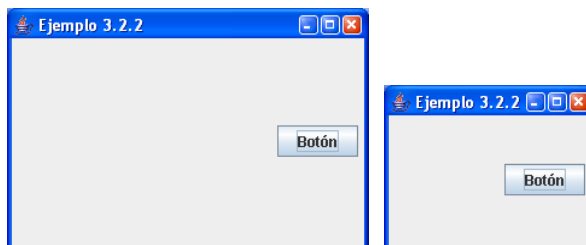
        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

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

```

**Ejemplo 3.2.2: Aplicación con el botón alineado a la derecha, con su tamaño preferido y con un espacio alrededor de 5 píxeles.**

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



**Figura 3.12:** Interfaz gráfica de usuario del ejemplo 3.2.2

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.2.2/Ejemplo322.java**

```

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

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

```

```

public class Ejemplo322 {

    public Ejemplo322() {

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

        JButton boton = new JButton("Botón");
        JPanel panelDeContenido = new JPanel();
        panelDeContenido.setLayout(new GridBagLayout());
        GridBagConstraints config = new GridBagConstraints();

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

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

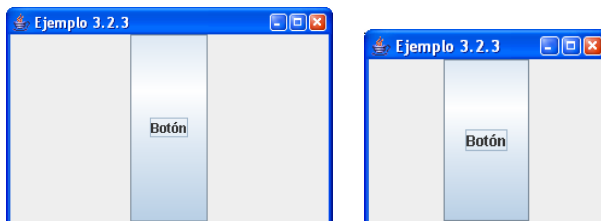
        panelDeContenido.add(boton, config);
        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

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

```

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

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



**Figura 3.13:** Interfaz gráfica de usuario del ejemplo 3.2.3

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.2.3/Ejemplo323.java**

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

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

public class Ejemplo323 {

    public Ejemplo323() {
        JFrame ventana = new JFrame("Ejemplo 3.2.3");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

        JButton boton = new JButton("Botón");
        JPanel panelDeContenido = new JPanel();
        panelDeContenido.setLayout(new GridBagLayout());
        GridBagConstraints config = new GridBagConstraints();

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

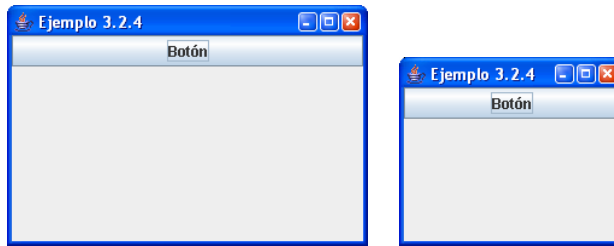
        config.fill = GridBagConstraints.VERTICAL;
        config.ipadx = 4;

        panelDeContenido.add(boton, config);
        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

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

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

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



**Figura 3.14:** Interfaz gráfica de usuario del ejemplo 3.2.4

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.2.4/Ejemplo324.java**

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ejemplo324 {

    public Ejemplo324() {
        JFrame ventana = new JFrame("Ejemplo 3.2.4");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

        JButton boton = new JButton("Botón");
        JPanel panelDeContenido = new JPanel();
        panelDeContenido.setLayout(new GridBagLayout());
        GridBagConstraints config = new GridBagConstraints();

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

        config.fill = GridBagConstraints.HORIZONTAL;
        config.anchor = GridBagConstraints.NORTH;

        panelDeContenido.add(boton, config);
        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

    public static void main(String[] args) {
        new Ejemplo324();
    }
}
```

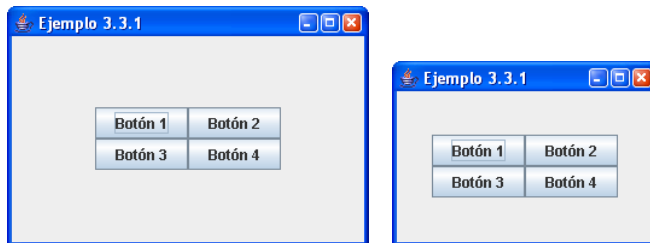
En los ejemplos anteriores se ha mostrado cómo colocar componentes dentro de su área de visualización. Ahora, a través del siguiente ejemplo, se pretende ver cómo configurar el tamaño de las celdas cuando hay más espacio del preferido.

### **Ejemplo 3.3**

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

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

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



**Figura 3.15:** Interfaz gráfica de usuario del ejemplo 3.3.1

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.3.1/Ejemplo331.java**

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

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

public class Ejemplo331 {

    public Ejemplo331() {

        JFrame ventana = new JFrame("Ejemplo 3.3.1");
```



```
ventana.setSize(300, 200);
ventana.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE);

JButton boton1 = new JButton("Botón 1");
JButton boton2 = new JButton("Botón 2");
JButton boton3 = new JButton("Botón 3");
JButton boton4 = new JButton("Botón 4");

JPanel panelDeContenido = new JPanel();

panelDeContenido.setLayout(new GridBagLayout());

GridBagConstraints config1 = new GridBagConstraints();
config1.gridx = 0;
config1.gridy = 0;
panelDeContenido.add(boton1, config1);

GridBagConstraints config2 = new GridBagConstraints();
config2.gridx = 1;
config2.gridy = 0;
panelDeContenido.add(boton2, config2);

GridBagConstraints config3 = new GridBagConstraints();
config3.gridx = 0;
config3.gridy = 1;
panelDeContenido.add(boton3, config3);

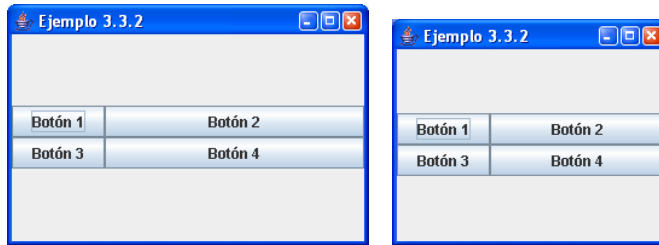
GridBagConstraints config4 = new GridBagConstraints();
config4.gridx = 1;
config4.gridy = 1;
panelDeContenido.add(boton4, config4);

ventana.setContentPane(panelDeContenido);
ventana.setVisible(true);
}

public static void main(String[] args) {
    new Ejemplo331();
}
}
```

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

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



**Figura 3.16:** Interfaz gráfica de usuario del ejemplo 3.3.2

El código fuente de la aplicación es el siguiente:

**libro/ejemplos/ejemplo3.3.2/Ejemplo332.java**

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

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

public class Ejemplo432 {

    public Ejemplo432() {

        JFrame ventana = new JFrame("Ejemplo 3.3.2");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);

        JButton boton1 = new JButton("Botón 1");
        JButton boton2 = new JButton("Botón 2");
        JButton boton3 = new JButton("Botón 3");
        JButton boton4 = new JButton("Botón 4");

        JPanel panelDeContenido = new JPanel();
        panelDeContenido.setLayout(new GridBagLayout());

        GridBagConstraints config1 = new GridBagConstraints();
        config1.gridx = 0;
        config1.gridy = 0;

        config1.weightx = 0.0;

        panelDeContenido.add(boton1, config1);
        GridBagConstraints config2 = new GridBagConstraints();
        config2.gridx = 1;
```

```
        config2.gridy = 0;

        config2.weightx = 1.0;
        config2.fill = GridBagConstraints.HORIZONTAL;

        panelDeContenido.add(boton2, config2);

        GridBagConstraints config3 = new GridBagConstraints();
        config3.gridx = 0;
        config3.gridy = 1;

        config3.weightx = 0.0;

        panelDeContenido.add(boton3, config3);

        GridBagConstraints config4 = new GridBagConstraints();
        config4.gridx = 1;
        config4.gridy = 1;

        config4.weightx = 1.0;
        config4.fill = GridBagConstraints.HORIZONTAL;

        panelDeContenido.add(boton4, config4);
        ventana.setContentPane(panelDeContenido);
        ventana.setVisible(true);
    }

    public static void main(String[] args) {
        new Ejemplo332();
    }
}
```

A continuación se dan algunos detalles de parte de la API Swing que pueden resultar muy útiles, al igual que se ha hecho en otros capítulos del presente libro.

### Aprendiendo la API. **Barras de desplazamiento**

Cuando el tamaño preferido de un componente es muy grande, pero se desea que se muestre completamente, se puede hacer uso de las barras de desplazamiento. Para que un componente se muestre con barras de desplazamiento si su tamaño preferido es mayor que el tamaño disponible, se utiliza el contenedor `JScrollPane`. A continuación se presenta un esquema de uso.

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

## Aprendiendo la API. Texto multilinea

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

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

Se suele usar este componente dentro de un contenedor `JScrollPane`.

De este modo cuando el número de líneas haga que el componente sea más grande que el espacio asignado por el administrador de distribución aparecerán las barras de desplazamiento.

## Aprendiendo la API. Botones de radio

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

- `public boolean isSelected()` - determina si el botón está o no seleccionado.
- `public void setSelected(boolean b)` - se usa para establecer la selección o no del botón.

Para que un conjunto de botones de radio se agrupen de forma que sólo uno de ellos pueda estar seleccionado, es necesario usar un objeto de la clase `ButtonGroup`. De manera que, para agrupar los botones de radio, simplemente habrá que invocar el siguiente método de la clase `ButtonGroup`:

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

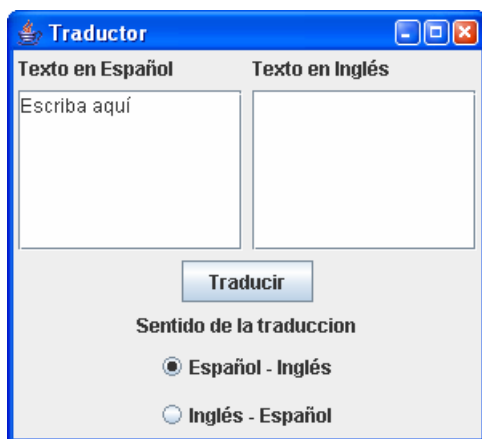
Es importante destacar que la clase `ButtonGroup` no ofrece muchas facilidades para determinar el botón de radio que se encuentra seleccionado en cada momento. Si se quiere disponer de esta funcionalidad de forma cómoda y reutilizable, será necesario programarla. Además, `ButtonGroup` no es un componente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

Por otra parte, si se desea ejecutar código cuando el usuario seleccione un nuevo botón de radio; es decir, cuando cambie el botón seleccionado, habrá que asociar código al evento de tipo `Action`. Pero si se desea seleccionar un botón de radio desde el código del programa, no desde la interfaz gráfica, no se generará este evento. Por este motivo, los botones de radio generan eventos de tipo `Item` cuando su estado pasa de seleccionado a no seleccionado o viceversa.

### **Ejemplo 3.4 – Aplicación de traducción 8.0**

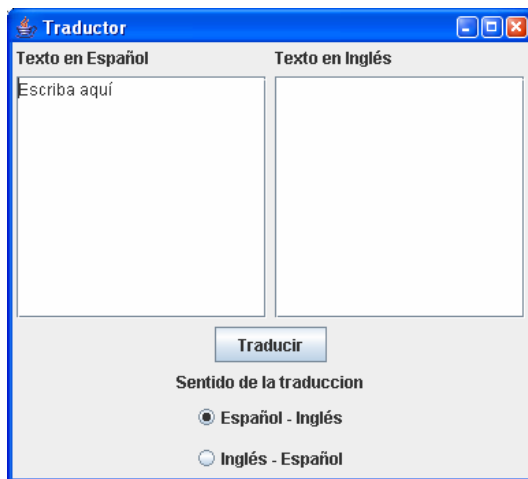
Se pretende que el traductor sea capaz de traducir textos formados por varias palabras. Para ello, se añadirá una ventana que contenga un área de texto para poder introducir el texto que se desea traducir, y otra área de texto para mostrar el texto resultante de la traducción. Además, se cambiará el botón y la etiqueta, que marcaban el sentido de la traducción en versiones anteriores de la aplicación, por dos botones de radio que se encargarán ahora de esa funcionalidad. Por último, la función de rollover será suprimida de la aplicación.

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



**Figura 3.17:** Interfaz gráfica de usuario de la aplicación de traducción 8.0

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



**Figura 3.18:** Interfaz gráfica de usuario redimensionada de la aplicación de traducción 8.0

Para implementar este ejemplo se han modificado diversas clases. La clase `VentanaTraductor` se ha modificado para no establecer el tamaño de forma fija, sino que se ha calculado para que el panel de contenido tenga su tamaño preferido. `PanelTraductor` es la otra clase que ha variado.

#### **libro/ejemplos/ejemplo3.4/VentanaTraductor.java**

```
import javax.swing.JFrame;

public class VentanaTraductor extends JFrame {

    public VentanaTraductor(Traductor traductor) {

        this.setContentPane(new PanelTraductor(traductor));
        this.setTitle("Traductor");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);

    }
}
```

#### **libro/ejemplos/ejemplo3.4/PanelTraductor.java**

```
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

class PanelTraductor extends JPanel {

    private Traductor traductor;

    private JButton botonTraduccion;
    private JTextArea areaTextoOrigen;
    private JTextArea areaTextoDestino;
    private JLabel etiIdiomaOrigen;
    private JLabel etiIdiomaDestino;
```

```
private JRadioButton rbSentidoOriginal;
private JRadioButton rbSentidoInvertido;

private final int idiomaOrigenOriginal;
private final int idiomaDestinoOriginal;

public PanelTraductor(Traductor traductor) {

    this.traductor = traductor;
    this.idiomaOrigenOriginal=traductor.getIdiomaOrigen();
    this.idiomaDestinoOriginal=
        traductor.getIdiomaDestino();

    this.botonTraduccion = new JButton("Traducir");
    this.areaTextoOrigen = new JTextArea("Escriba aquí");
    this.areaTextoDestino = new JTextArea();
    this.areaTextoDestino.setEditable(false);

    JScrollPane scrollTextoOrigen =
        new JScrollPane(areaTextoOrigen);
    JScrollPane scrollTextoDestino =
        new JScrollPane(areaTextoDestino);

    scrollTextoOrigen.setPreferredSize(
        new Dimension(140,100));
    scrollTextoDestino.setPreferredSize(
        new Dimension(140,100));

    this.etiqIdiomaOrigen = new JLabel();
    this.etiqIdiomaDestino = new JLabel();
    muestraIdiomasEnTextos();

    this.rbSentidoOriginal = new JRadioButton(Traductor
        .getCadenaIdioma(traductor.getIdiomaOrigen())
        + " - "
        + Traductor.getCadenaIdioma(
            traductor.getIdiomaDestino()));

    this.rbSentidoInvertido = new JRadioButton(Traductor
        .getCadenaIdioma(traductor.getIdiomaDestino())
        + " - "
        + Traductor.getCadenaIdioma(
            traductor.getIdiomaOrigen()));

    ButtonGroup grupoBotones = new ButtonGroup();
    grupoBotones.add(rbSentidoOriginal);
    grupoBotones.add(rbSentidoInvertido);
    rbSentidoOriginal.setSelected(true);

    rbSentidoOriginal.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
```



```
        if(rbSentidoOriginal.isSelected()){
            invierteIdioma();
        }
    });

rbSentidoInvertido.addItemListener(new ItemListener()
{
    public void itemStateChanged(ItemEvent e) {
        if(rbSentidoInvertido.isSelected()){
            invierteIdioma();
        }
    }
});

botonTraduccion.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e) {
        traducir();
    }
});

this.setLayout(new GridBagLayout());

GridBagConstraints config1 = new GridBagConstraints();
config1.insets = new Insets(3, 3, 3, 3);
config1.anchor = GridBagConstraints.SOUTHWEST;
config1.gridx = 0;
config1.gridy = 0;
config1.weightx = 1.0;
this.add(etiqIdiomaOrigen, config1);

GridBagConstraints config2 = new GridBagConstraints();
config2.insets = new Insets(3, 3, 3, 3);
config2.anchor = GridBagConstraints.SOUTHWEST;
config2.gridx = 1;
config2.gridy = 0;
config2.weightx = 1.0;
this.add(etiqIdiomaDestino, config2);

GridBagConstraints config3 = new GridBagConstraints();
config3.insets = new Insets(3, 3, 3, 3);
config3.anchor = GridBagConstraints.SOUTHWEST;
config3.gridx = 0;
config3.gridy = 1;
config3.weightx = 1.0;
config3.weighty = 1.0;
config3.fill = GridBagConstraints.BOTH;
this.add(scrollTextoOrigen, config3);

GridBagConstraints config4 = new GridBagConstraints();
```

```
config4.insets = new Insets(3, 3, 3, 3);
config4.anchor = GridBagConstraints.SOUTHWEST;
config4.gridx = 1;
config4.gridy = 1;
config4.weightx = 1.0;
config4.weighty = 1.0;
config4.fill = GridBagConstraints.BOTH;
this.add(scrollTextoDestino, config4);

GridBagConstraints config5 = new GridBagConstraints();
config5.insets = new Insets(3, 3, 3, 3);
config5.gridx = 0;
config5.gridy = 2;
config5.weightx = 1.0;
config5.weighty = 0;
config5.gridwidth = 2;
this.add(botonTraduccion, config5);

GridBagConstraints config6 = new GridBagConstraints();
config6.insets = new Insets(3, 3, 3, 3);
config6.gridx = 0;
config6.gridy = 3;
config6.weightx = 1.0;
config6.weighty = 0;
config6.gridwidth = 2;
this.add(new JLabel("Sentido de la traduccion"),
        config6);

GridBagConstraints config7 = new GridBagConstraints();
config7.insets = new Insets(3, 3, 3, 3);
config7.gridx = 0;
config7.gridy = 4;
config7.weightx = 1.0;
config7.weighty = 0;
config7.gridwidth = 2;
this.add(rbSentidoOriginal, config7);

GridBagConstraints config8 = new GridBagConstraints();
config8.insets = new Insets(3, 3, 3, 3);
config8.gridx = 0;
config8.gridy = 6;
config8.weightx = 1.0;
config8.weighty = 0;
config8.gridwidth = 2;
this.add(rbSentidoInvertido, config8);

}

protected void invierteIdioma() {
    this.traductor.invierteIdioma();
    this.muestraIdiomasEnTextos();
}
```

```
}

private void traducir() {
    areaTextoDestino.setText(traductor
        .traduceTexto(areaTextoOrigen.getText()));
}

private void muestraIdiomasEnTextos() {
    this.etiqIdiomaOrigen.setText("Texto en "
        + Traductor.getCadenaIdioma(traductor
            .getIdiomaOrigen()));

    this.etiqIdiomaDestino.setText("Texto en "
        + Traductor.getCadenaIdioma(traductor
            .getIdiomaDestino()));
}
}
```

### **3.2.6. Otros administradores de distribución: *BoxLayout*, *SpringLayout* y *CardLayout***

No se van a estudiar los administradores de distribución `BoxLayout`, `SpringLayout` y `CardLayout`. Como ya se ha comentado anteriormente, este libro no pretende aportar un estudio exhaustivo y detallado de todos los componentes, y con respecto a los administradores de distribución, se ha preferido estudiar los más usados y potentes para la construcción de interfaces gráficas de usuario. Por ello, se recomienda al lector consultar la documentación para más información sobre estos administradores.

Se podría decir que el administrador de distribución `BoxLayout` se encuentra, en cuanto a potencia, entre los administradores `FlowLayout` y `GridBagLayout`.

Con respecto al administrador de distribución `SpringLayout`, comentar que se basa en las relaciones que tienen los componentes entre sí y es usado, sobre todo, en entornos de desarrollo integrados.

Por último, el administrador de distribución `CardLayout`, de todos los componentes que tiene el contenedor, sólo muestra uno a la vez, pudiéndose elegir el componente que se va a mostrar mediante un método del administrador.

### 3.3. Administrador de distribución nulo. Distribución estática

Existe la posibilidad de usar un administrador de distribución nulo. Este es el administrador que se obtiene cuando se invoca, en un contenedor, el método `public void setLayout(LayoutManager lm)` pasando como valor para el parámetro `lm` un valor nulo (`null`). Con este administrador de distribución nulo, los componentes se dimensionarán según el tamaño establecido a través del método `public void setSize(int width, int height)` de la clase `Component`. Si no se especifica un tamaño para los componentes, este será de 0 píxel de alto por 0 píxel de ancho. Si se quiere que tengan su tamaño preferido, habrá que usar la sentencia `comp.setSize(comp.getPreferredSize())`.

La posición del componente vendrá determinada por la indicada en el método `public void setLocation(int x, int y)` y será relativa a la esquina superior izquierda del contenedor.

No se aconseja este modo de distribuir componentes en un contenedor ya que, si el contenedor cambia de tamaño los componentes no se distribuirán de manera uniforme. Otro motivo por el que no se aconseja usar esta forma de distribuir componentes, es que Java utiliza un sistema que permite cambiar el aspecto de la aplicación, llamado **Pluggable Look & Feel**. Dependiendo del sistema, podrá ejecutarse la aplicación con un look & feel u otro. Los componentes gráficos pueden ser de distinto tamaño en distintos look & feel; por lo tanto, con unos se podrá obtener un buen resultado y con otros no. En este libro no se va a tratar el Pluggable Look & Feel de Java, por tanto, se recomienda al lector que utilice las referencias indicadas en el capítulo 1 para obtener más información.

### 3.4. Administración por anidamiento de contenedores

Como se ha podido comprobar, la administración de componentes no es una tarea sencilla. En interfaces gráficas de usuario complejas se suele usar el anidamiento de contenedores; es decir, tratar un contenedor `JPanel` como si fuera un componente más. En ese contenedor `JPanel` irán ciertos componentes con cierto administrador de distribución y ese panel irá también dentro de otro contenedor. De esta forma, se combinan

administradores de distribución eligiendo el más efectivo en cada momento.

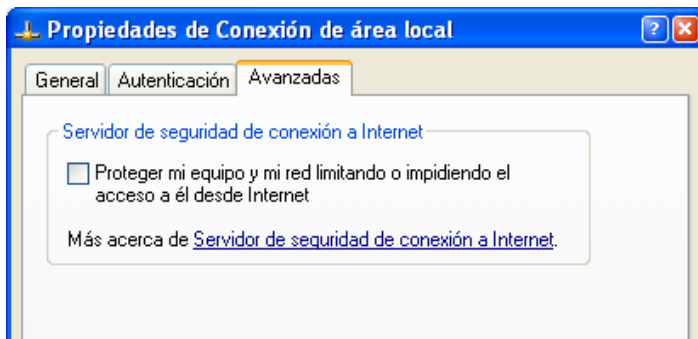
### 3.5. Bordes

Para construir algunos componentes de una interfaz gráfica de usuario se utilizan bordes. Los bordes aparecen alrededor de muchos componentes. Por ejemplo, alrededor de un botón aparece un borde que le da esa apariencia 3D en algunas plataformas. Al pulsar un botón el borde suele cambiar para que el usuario perciba que el botón ha sido pulsado. Otros componentes también pueden usar otros tipos de bordes.

Los bordes predefinidos de los componentes no suelen manejarse directamente por el desarrollador. En cambio, al construir una interfaz gráfica se suele usar un borde con título para agrupar componentes relacionados. Este título aparecerá en la interfaz gráfica como se muestra en la figura 3.19.

Para poner un borde de texto alrededor de un panel se usa la siguiente estructura básica:

```
...
panel.setBorder(
    javax.swing.BorderFactory.
        createTitledBorder(" Título del panel "));
...
```



**Figura 3.19:** Ejemplo de interfaz gráfica de usuario con bordes

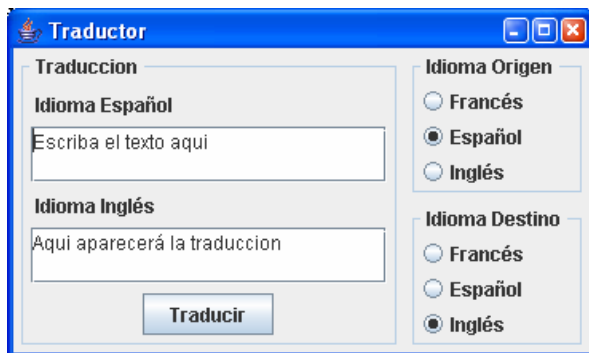
Existen otros tipos de bordes, por ejemplo, el creado con `BorderFactory.createEmptyBorder(int top, int left, int bottom, int right)` crea un borde vacío, sin ninguna línea ni color, que añade un espacio alrededor del componente.

A continuación, se presenta un ejemplo para usar bordes y anidamiento de contenedores.

### Ejemplo 3.5 - Aplicación de traducción 9.0

El panel de contenido de la aplicación de traducción 9.0 ha de administrarse con un administrador de distribución `GridBagLayout`. Se utilizará otro panel que haga uso de `GridBagLayout` para los campos de texto, las etiquetas y el panel de traducción. Por último, se utilizará un panel con el administrador `GridLayout` en el que aparezcan botones de radio que permitan elegir el idioma origen y destino de la traducción.

La interfaz gráfica de usuario de la aplicación se presenta en la figura 3.20.



**Figura 3.20:** Interfaz de usuario de la aplicación de traducción 9.0

El nuevo código fuente resultante de la ampliación de la aplicación se muestra a continuación:

**libro/ejemplos/ejemplo3.5/PanelTraductor.java**

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

```
import java.awt.Insets;
import javax.swing.JPanel;

public class PanelTraductor extends JPanel {
    private PanelTextos panelTextos;
    private PanelSeleccionIdioma panelIdiomaOrigen;
    private PanelSeleccionIdioma panelIdiomaDestino;

    public PanelTraductor(Traductor traductor) {

        panelTextos = new PanelTextos(traductor);
        panelIdiomaOrigen = new PanelSeleccionIdioma(
            PanelSeleccionIdioma.IDIOMA_ORIGEN,
            traductor,
            this);
        panelIdiomaDestino = new PanelSeleccionIdioma(
            PanelSeleccionIdioma.IDIOMA_DESTINO,
            traductor,
            this);

        this.setLayout(new GridBagLayout());

        GridBagConstraints config1 = new GridBagConstraints();
        config1.insets = new Insets(3, 3, 3, 3);
        config1.weighty = 1.0;
        config1.weightx = 1.0;
        config1.fill = GridBagConstraints.BOTH;
        config1.gridheight = 2;
        this.add(panelTextos, config1);

        GridBagConstraints config2 = new GridBagConstraints();
        config2.insets = new Insets(3, 3, 3, 3);
        config2.gridx = 1;
        config2.weighty = 1.0;
        config2.fill = GridBagConstraints.BOTH;
        this.add(panelIdiomaOrigen, config2);

        GridBagConstraints config3 = new GridBagConstraints();
        config3.insets = new Insets(3, 3, 3, 3);
        config3.gridx = 1;
        config3.gridy = 1;
        config3.weighty = 1.0;
        config3.fill = GridBagConstraints.BOTH;
        this.add(panelIdiomaDestino, config3);
    }

    public void muestraIdiomas() {
        this.panelTextos.muestraIdiomas();
    }
}
```

**libro/ejemplos/ejemplo3.5/PanelTextos.java**

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

class PanelTextos extends JPanel {

    private Traductor traductor;

    private JLabel etiquetaOrigen;
    private JLabel etiquetaDestino;
    private JTextArea areaOrigen;
    private JTextArea areaDestino;
    private JButton botonTraduccion;

    public PanelTextos(Traductor traductor) {

        this.traductor = traductor;
        this.etiquetaOrigen = new JLabel();
        this.etiquetaDestino = new JLabel();
        this.muestraIdiomas();

        this.areaOrigen = new JTextArea(
            "Escriba el texto aqui", 2, 20);
        this.areaDestino = new JTextArea(
            "Aqui aparecerá la traduccion", 2, 20);
        this.areaDestino.setEditable(false);

        botonTraduccion = new JButton("Traducir");
        botonTraduccion.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e) {
                traducir();
            }
        });
        this.setBorder(BorderFactory
            .createTitledBorder(" Traduccion "));
        JScrollPane spAreaOrigen =
            new JScrollPane(areaOrigen);
        JScrollPane spAreaDestino =
            new JScrollPane(areaDestino);
        this.setLayout(new GridBagLayout());
    }
}
```



```

GridBagConstraints config1 = new GridBagConstraints();
config1.insets = new Insets(3, 3, 3, 3);
config1.anchor = GridBagConstraints.WEST;
this.add(etiquetaOrigen, config1);

GridBagConstraints config2 = new GridBagConstraints();
config2.insets = new Insets(3, 3, 3, 3);
config2.anchor = GridBagConstraints.WEST;
config2.gridy = 2;
this.add(etiquetaDestino, config2);

GridBagConstraints config3 = new GridBagConstraints();
config3.insets = new Insets(3, 3, 3, 3);
config3.gridy = 4;
this.add(botonTraduccion, config3);

GridBagConstraints config4 = new GridBagConstraints();
config4.insets = new Insets(3, 3, 3, 3);
config4.weighty = 1.0;
config4.weightx = 1.0;
config4.fill = GridBagConstraints.BOTH;
config4.gridy = 1;
this.add(spAreaOrigen, config4);

GridBagConstraints config5 = new GridBagConstraints();
config5.insets = new Insets(3, 3, 3, 3);
config5.weighty = 1.0;
config5.weightx = 1.0;
config5.fill = GridBagConstraints.BOTH;
config5.gridy = 3;
this.add(spAreaDestino, config5);
}
public void traducir() {
    areaDestino.setText(
        traductor.traduceTexto(areaOrigen.getText()));
}
public void muestraIdiomas() {
    this.etiquetaOrigen.setText(" Idioma "
        + Traductor.getCadenaIdioma(traductor
            .getIdiomaOrigen()));
    this.etiquetaDestino.setText(" Idioma "
        + Traductor.getCadenaIdioma(traductor
            .getIdiomaDestino()));
}
}

```

**libro/ejemplos/ejemplo3.5/PanelSeleccionIdioma.java**

```
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JRadioButton;

public class PanelSeleccionIdioma extends JPanel {

    public static final int IDIOMA_ORIGEN = 0;
    public static final int IDIOMA_DESTINO = 1;

    private Traductor traductor;

    private int tipoIdioma;
    private PanelTraductor panelTraductor;

    public PanelSeleccionIdioma(int tipoIdioma,
        Traductor traductor,
        PanelTraductor panelTraductor) {

        this.panelTraductor = panelTraductor;
        this.tipoIdioma = tipoIdioma;
        this.traductor = traductor;
        String cadenaTipo;

        if (tipoIdioma == IDIOMA_ORIGEN) {
            cadenaTipo = " Idioma Origen ";
        } else {
            cadenaTipo = " Idioma Destino ";
        }

        this.setBorder(BorderFactory
            .createTitledBorder(cadenaTipo));
        this.setMinimumSize(new Dimension(110, 50));
        this.setPreferredSize(new Dimension(110, 50));

        ButtonGroup grupoBotones = new ButtonGroup();
        this.setLayout(new GridLayout(
            Traductor.NUM_IDIOMAS, 1, 3, 3));

        for (int i = 0; i < Traductor.NUM_IDIOMAS; i++) {

            JRadioButton boton =
                new JRadioButton(
                    Traductor.getCadenaIdioma(i));
```

```
this.add(boton);
final int numBoton = i;

boton.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() ==
            ItemEvent.SELECTED)
        {
            botonRadioSeleccioado(numBoton);
        }
    }
});

grupoBotones.add(boton);
if (tipoIdioma == IDIOMA_ORIGEN
    && i == traductor.getIdiomaOrigen()) {
    boton.setSelected(true);
} else if (tipoIdioma == IDIOMA_DESTINO
    && i == traductor.getIdiomaDestino()) {
    boton.setSelected(true);
}
}

public void botonRadioSeleccioado(int numBoton){

    if (this.tipoIdioma == IDIOMA_ORIGEN) {
        traductor.setIdiomaOrigen(numBoton);
    } else if (this.tipoIdioma == IDIOMA_DESTINO){
        traductor.setIdiomaDestino(numBoton);
    } else {
        throw new Error();
    }

    this.panelTraductor.muestraIdiomas();
}
}
```



## **Capítulo 4**

### **Visión general de la librería Swing**

En el presente capítulo se verán los diferentes elementos gráficos y eventos de la librería Swing. También se verá en detalle algunos de los componentes más comunes en el desarrollo de aplicaciones con interfaces gráficas de usuario.

#### **4.1. Introducción**

Con lo visto en capítulos anteriores, se tiene una visión general sobre construcción de interfaces gráficas de usuario en Java. En el capítulo 1 se introdujo la librería Swing, viendo aquellos elementos básicos para la construcción de una primera aplicación con interfaz gráfica. En el capítulo 2 se vio cómo gestionar y organizar el código de eventos de Swing. En el siguiente capítulo, en el tercero, se vieron diferentes formas de organización de los componentes de la interfaz gráfica de la aplicación. Además, a lo largo de estos capítulos previos, se han ido presentando algunos componentes necesarios para los diferentes ejemplos de aplicaciones mostrados.

Se considera, por lo tanto, que en este punto se tienen suficientes conocimientos sobre cuáles son los pasos principales para la creación de cualquier interfaz gráfica de usuario. Por ello, aquí se presentan todos los componentes y eventos de la API. Se entra en más detalle en ciertos componentes como menús, imágenes, diálogos y paneles de opciones. Estos componentes, junto con los vistos en capítulos anteriores, son clave en la construcción de aplicaciones con interfaz gráfica.

## 4.2. Los componentes de Swing

En este apartado se presentan los diferentes elementos gráficos que tiene la librería Swing. Se presentan en diferentes tablas (4.1 - 4.7), agrupados por tipos. Se indica, en cada tabla, la clase que representa cada elemento acompañada de una breve descripción. Todas las clases de los diferentes elementos se encuentran en el paquete `javax.swing`.

Clase	Descripción
<code>JLabel</code>	Etiqueta. Muestra imágenes y texto. El texto se puede formatear usando HTML
<code>JButton</code>	Botón. Puede mostrar imágenes y texto
<code>JCheckBox</code>	Casilla de verificación
<code>JRadioButton</code>	Botón de radio. Se usa para seleccionar una opción entre varias
<code>JToggleButton</code>	Botón que se queda presionado al ser pulsado
<code>JComboBox</code>	Lista desplegable
<code>JScrollBar</code>	Barra de desplazamiento. Se usa en los contenedores que permiten que su contenido sea más grande que ellos. El desarrollador lo usará indirectamente cuando utilice el panel <code>JScrollPane</code>
<code>JSeparator</code>	Se usa en los menús y barras de herramientas para separar opciones
<code>JSlider</code>	Deslizador
<code>JSpinner</code>	Campo de texto con botones para elegir el elemento siguiente o anterior. Se puede usar para números, fechas o elementos propios
<code>JProgressBar</code>	Barra de progreso
<code>JList</code>	Lista de elementos

**Tabla 4.1:** Componentes simples

Clase	Descripción
<code>JTable</code>	Tabla
<code>JTree</code>	Árbol
<code>JFileChooser</code>	Selector de ficheros
<code>JColorChooser</code>	Selector de color
<code>JOptionPane</code>	Cuadro de diálogo personalizable

**Tabla 4.2:** Componentes complejos

Clase	Descripción
JTextField	Campo de texto simple. Sólo puede contener una línea de texto
JFormattedTextField	Campo de texto que acepta textos que siguen un patrón
JPasswordField	Campo de texto para contraseñas
JTextArea	Área de texto simple. Puede contener varias líneas de texto
JEditorPane	Área de texto que puede mostrar estilos, tipos de letra, imágenes, etc. Muestra documentos en formato RTF y HTML 3.2
JTextPane	Área de texto que puede mostrar estilos, tipos de letra, imágenes, etc. Permite un mayor control sobre los estilos de texto que JEditorPane

**Tabla 4.3:** Componentes de texto

Clase	Descripción
JPanel	Contenedor de propósito general
JScrollPane	Contenedor con barras de desplazamiento
JSplitPane	Contenedor dividido en dos partes
JTabbedPane	Contenedor con pestañas
JDesktopPane	Contenedor para incluir ventanas dentro
JToolBar	Barra de herramientas

**Tabla 4.4:** Contenedores

Clase	Descripción
JFrame	Ventana de aplicación
JDialog	Cuadro de diálogo
JWindow	Ventana sin marco
JInternalFrame	Ventana interna

**Tabla 4.5:** Ventanas

Clase	Descripción
JMenu	Un menú
JCheckBoxMenuItem	Elemento de menú como casilla de verificación
JRadioButtonMenuItem	Elemento de menú como botón de radio
JMenuItem	Un botón que se encuentra en un menú
JMenuBar	Barra de menús

**Tabla 4.6:** Menús

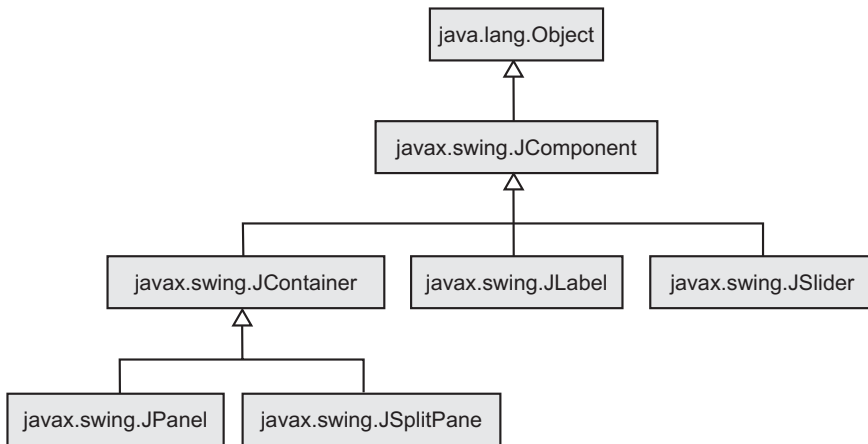
Clase	Descripción
JToolTip	Tooltip

**Tabla 4.7:** Otro componente

En este libro no se presentan todos los componentes en detalle. Algunos de ellos se verán con cierta profundidad para mostrar de forma general como se construyen interfaces gráficas escalables, modulares, reutilizables, etc. Se pretende que el lector tenga la capacidad de consultar la documentación técnica en formato JavaDoc y de otras fuentes, y pueda comprender lo que se dice en ella sin ninguna dificultad.

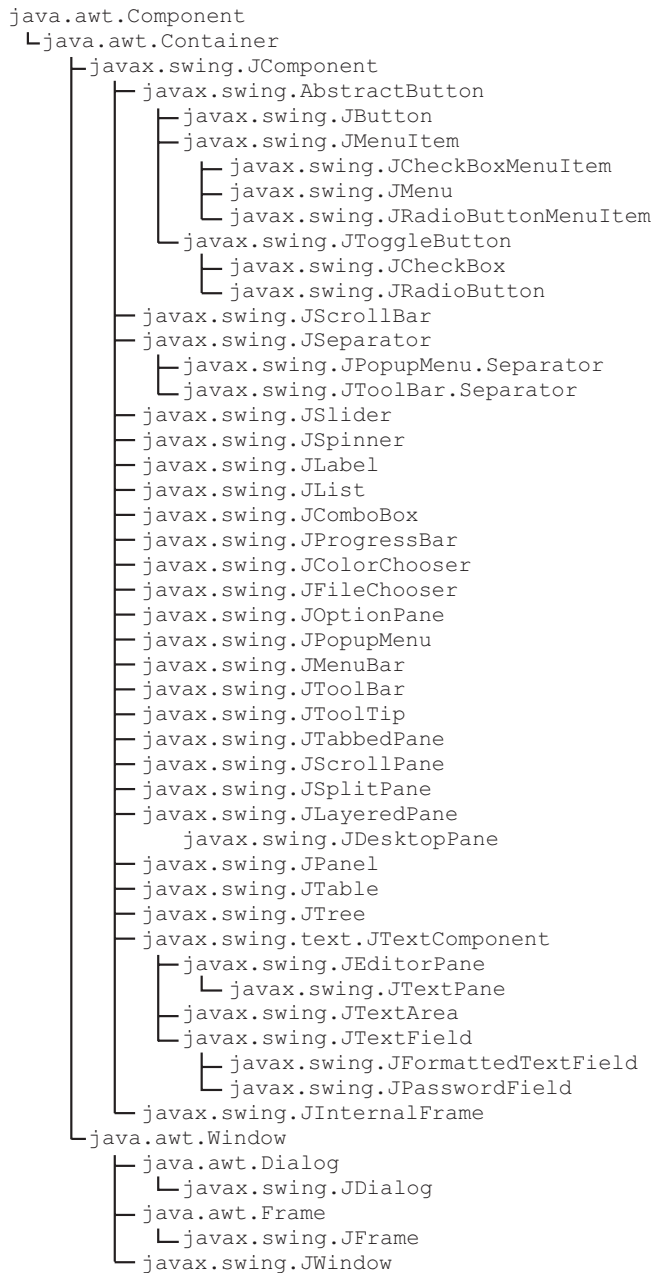
Debido a la evolución que han sufrido las interfaces gráficas de usuario en Java, la librería Swing no es intuitiva en algunos aspectos. Considerando que existen componentes atómicos y contenedores, en una jerarquía de herencia razonable todos los componentes heredarían de la clase `javax.swing.JComponent`. Todos aquellos que además sean contenedores, heredarían de `javax.swing.JContainer`. Una vista parcial de esta jerarquía de herencia razonable sería como la mostrada en la figura 4.1





**Figura 4.1:** Vista de una jerarquía de herencia razonable para los componentes Swing

Por motivos de compatibilidad, la verdadera jerarquía no es tan razonable. La jerarquía de clases **real**, que representa la mayoría de los componentes de la interfaz gráfica de usuario en Swing, se muestra en la figura 4.2. Aparece de forma compacta para mejorar su legibilidad.

**Figura 4.2:** Jerarquía de clases real de los componentes Swing

### 4.3. Los eventos de Swing

En este apartado se verán los tipos de eventos que pueden ser generados en cada uno de los componentes de la librería Swing. Estas tablas han sido construidas con los métodos que tiene cada clase que representa a un componente (`addXXListener(XXListener listener)`). Hay que tener en cuenta la herencia de componentes, de forma que los eventos generados en objetos de la clase padre serán generados también en objetos de las clases hijas.

Los diferentes tipos de eventos se muestran en las tablas 4.8 - 4.25. Cada tipo de evento se acompaña de las acciones concretas que lo generan. Por ejemplo, un evento de tipo `Mouse`, se acompaña de las acciones: tecla del ratón presionada, tecla del ratón soltada, ratón entrando en un componente, etc.; Entre paréntesis aparece el tipo del componente y el nombre de los métodos que se corresponden con cada una de las acciones.

Tipos de eventos que genera	Evento concreto
Eventos del Foco ( <code>Focus</code> )	Foco obtenido ( <code>focusGained</code> ) Foco perdido ( <code>focusLost</code> )
Eventos de entrada de teclado ( <code>Key</code> )	Tecla presionada ( <code>keyPressed</code> ) Tecla soltada ( <code>keyReleased</code> ) Tecla presionada y soltada ( <code>keyTyped</code> )
Eventos de ratón ( <code>Mouse</code> )	Tecla del ratón presionada y soltada ( <code>mouseClicked</code> ) El ratón entra en un componente ( <code>mouseEntered</code> ) El ratón sale de un componente ( <code>mouseExited</code> ) Tecla del ratón presionada ( <code>mousePressed</code> ) Tecla del ratón soltada ( <code>mouseReleased</code> )
Eventos de movimiento del ratón ( <code>MouseMotion</code> )	El ratón se ha movido mientras una tecla se encuentra pulsada. El componente está siendo arrastrado ( <code>mouseDragged</code> ) El ratón se ha movido y no se está pulsando ninguna tecla ( <code>mouseMoved</code> )

Eventos de la rueda del ratón (MouseWheel)	La rueda ha rotado. (mouseWheelMoved)
--	---------------------------------------

**Tabla 4.8:** Eventos para la clase Component

Tipos de eventos que genera	Evento concreto
Eventos de acción (Action)	Se ha hecho clic en el botón (actionPerformed)
Eventos de cambio (Change)	El estado del botón ha cambiado (stateChanged)
Eventos de selección de un item (Item)	Ha cambiado el estado de selección del botón (itemStateChanged)

**Tabla 4.9:** Eventos para la clase AbstractButton

Tipos de eventos que genera	Evento concreto
Eventos de acción (Action)	Se ha seleccionado un elemento. Si se puede escribir en el cuadro de texto se genera este evento cuando se pulsa la tecla ENTER (actionPerformed)
Eventos de selección de un item (Item)	Ha cambiado la selección de la lista desplegable (itemStateChanged)

**Tabla 4.10:** Eventos para la clase JComboBox

Tipos de eventos que genera	Evento concreto
Eventos de foco en la ventana (WindowFocus)	Foco obtenido por la ventana (windowGainedFocus) Foco perdido por la ventana (windowLostFocus)
Eventos del ciclo de vida de la ventana (Window)	Ventana activada (windowActivated) La ventana ha sido cerrada como resultado de llamar al método dispose (windowClosed)

	<p>El usuario ha pulsado el botón de cierre de la ventana (<code>windowClosing</code>)</p> <p>La ventana ya no es la ventana activa (<code>windowDeactivated</code>)</p> <p>La ventana ha pasado de minimizada a normal (<code>windowDeiconified</code>)</p> <p>La ventana se ha minimizado (<code>windowIconified</code>)</p> <p>La ventana se ha hecho visible por primera vez (<code>windowOpened</code>)</p>
Eventos de estado de la ventana ( <code>WindowState</code> )	La ventana ha cambiado de estado en cuanto a minimizada, maximizada, restaurada, etc. ( <code>WindowStateChange</code> )

**Tabla 4.11:** Eventos para la clase `Window`

Tipos de eventos que genera	Evento concreto
Eventos de entrada de texto ( <code>InputMethod</code> )	<p>El cursor ha cambiado de posición (<code>caretPositionChanged</code>)</p> <p>El texto introducido por teclado ha cambiado (<code>inputMethodTextChanged</code>)</p>
Eventos de cursor ( <code>Caret</code> )	El cursor ha cambiado de posición ( <code>caretUpdate</code> )

**Tabla 4.12:** Eventos para la clase `JTextComponent`

Tipos de eventos que genera	Evento concreto
Eventos de hiperenlaces ( <code>Hyperlink</code> )	El link ha sido pulsado ( <code>hyperlinkUpdate</code> )

**Tabla 4.13:** Eventos para la clase `JEditorPane`

Tipos de eventos que genera	Evento concreto
Eventos de acción (Action)	El usuario ha pulsado la tecla ENTER en el cuadro de texto (actionPerformed)

**Tabla 4.14:** Eventos para la clase `JTextField`

Tipos de eventos que genera	Evento concreto
Eventos de ciclo de vida de la ventana interna (InternalFrame)	Ventana activada (internalFrameActivated) La ventana ha sido cerrada como resultado de llamar al método dispose (internalFrameClosed) El usuario ha pulsado el botón de cierre de la ventana (internalFrameClosing) La ventana ya no es la ventana activa (internalFrameDeactivated) La ventana ha pasado de minimizada a normal (internalFrameDeiconified) La ventana se ha minimizado (internalFrameIconified) La ventana se ha hecho visible por primera vez (internalFrameOpened)

**Tabla 4.15:** Eventos para la clase `JInternalFrame`

Tipos de eventos que genera	Evento concreto
Eventos de selección (ListSelection)	La selección ha cambiado (valueChanged)

**Tabla 4.16:** Eventos para la clase `JList`

Tipos de eventos que genera	Evento concreto
Eventos del menú (Menu)	El menú ha sido cancelado (menuCanceled) El menú ha sido deseleccionado (menuDeselected)

	El menú ha sido seleccionado (menuSelected)
--	---

**Tabla 4.17:** Eventos para la clase JMenu

<b>Tipos de eventos que genera</b>	<b>Evento concreto</b>
Eventos de arrastrar con el ratón en un ítem de menú (MenuDragMouse)	<p>En el ítem se está arrastrando el ratón (menuDragMouseDragged)</p> <p>En el ítem ha entrado el ratón con una tecla pulsada, ha entrado arrastrando (menuDragMouseEntered)</p> <p>Ha salido del ítem un ratón con una tecla pulsada, ha salido arrastrando (menuDragMouseExited)</p> <p>Un ratón que se estaba moviendo con una tecla pulsada, ha soltado la tecla dentro del ítem (menuDragMouseReleased)</p>
Eventos de entrada de teclado en los menús (menuKey)	<p>Tecla presionada (menuKeyPressed)</p> <p>Tecla soltada (menuKeyReleased)</p> <p>Tecla presionada y soltada (menuKeyTyped)</p>

**Tabla 4.18:** Eventos para la clase JMenuItem

<b>Tipos de eventos que genera</b>	<b>Evento concreto</b>
Evento de menú popup (PopupMenu)	<p>Menú cancelado (popupMenuCanceled)</p> <p>Menú que se va a hacer invisible (popupMenuWillBecomeInvisible)</p> <p>Menú que se va a hacer visible (popupMenuWillBecomeVisible)</p>

**Tabla 4.19:** Eventos para la clase JPopupMenu

Tipos de eventos que genera	Evento concreto
Eventos de cambio (Change)	El estado de la barra de progreso ha cambiado (stateChanged)

**Tabla 4.20:** Eventos para la clase `JProgressBar`

Tipos de eventos que genera	Evento concreto
Eventos de cambio (Change)	El estado de la barra de deslizamiento ha cambiado (stateChanged)

**Tabla 4.21:** Eventos para la clase `JSlider`

Tipos de eventos que genera	Evento concreto
Eventos de cambio (Change)	El estado del selector ha cambiado (stateChanged)

**Tabla 4.22:** Eventos para la clase `JSpinner`

Tipos de eventos que genera	Evento concreto
Eventos de cambio (Change)	El estado del panel con pestañas ha cambiado (stateChanged)

**Tabla 4.23:** Eventos para la clase `JTabbedPane`

Tipos de eventos que genera	Evento concreto
Eventos de expansión en el árbol (TreeExpansion)	Un ítem del árbol ha sido colapsado (treeCollapsed) Un ítem del árbol ha sido expandido (treeExpanded)
Eventos de futuras expansiones en el árbol (TreeWillExpand)	Un ítem del árbol será colapsado (treeWillCollapse) Un ítem del árbol será expandido (treeWillExpand)
Eventos de selección de los	La selección en el árbol ha cambiado



items del árbol (TreeSelection)	(valueChanged)
---------------------------------	----------------

Tabla 4.24: Eventos para la clase JTree

Tipos de eventos que genera	Evento concreto
Debido al uso del patrón MVC no genera eventos directamente. En el capítulo 6 se explica en detalle dicho patrón.	

Tabla 4.25: Eventos para la clase JTable

En el capítulo 1 se vio cómo implementar y asociar el código de eventos, y que para cada tipo de eventos existía una interfaz denominada `XXListener`, siendo `xx` el tipo de evento. Por ello, en la figura 4.3 se presentan los distintos interfaces `XXListener` correspondientes a cada uno de los tipos de eventos.

```
java.util.EventListener
├── java.awt.event.ActionListener
├── java.awt.event.FocusListener
├── java.awt.event.InputMethodListener
├── java.awt.event.ItemListener
├── java.awt.event.KeyListener
├── java.awt.event.MouseListener
├── java.awt.event.MouseMotionListener
├── java.awt.event.MouseWheelListener
├── java.awt.event.TextListener
├── java.awt.event.WindowFocusListener
├── java.awt.event.WindowListener
├── java.awt.event.WindowStateListener
├── javax.swing.event.CaretListener
├── javax.swing.event.ChangeListener
├── javax.swing.event.HyperlinkListener
├── javax.swing.event.InternalFrameListener
├── javax.swing.event.ListSelectionListener
├── javax.swing.event.MenuDragMouseListener
├── javax.swing.event.MenuKeyListener
├── javax.swing.event.MenuListener
├── javax.swing.event.PopupMenuListener
├── javax.swing.event.TreeExpansionListener
├── javax.swing.event.TreeSelectionListener
└── javax.swing.event.TreeWillExpandListener
```

Figura 4.3: Clases `XXListener` correspondientes a cada uno de los eventos

En la figura 4.4 se muestran todas las clases de `XXEvent` que existen para los componentes de la interfaz gráfica.

```

java.util.EventObject
├── javax.swing.event.CaretEvent
├── javax.swing.event.ChangeEvent
├── javax.swing.event.HyperlinkEvent
├── javax.swing.event.ListSelectionEvent
├── javax.swing.event.MenuEvent
├── javax.swing.event.PopupMenuEvent
├── javax.swing.event.TreeExpansionEvent
├── javax.swing.event.TreeSelectionEvent
├── java.awt.AWTEvent
│   ├── javax.swing.event.InternalFrameEvent
│   ├── java.awt.event.ActionEvent
│   ├── java.awt.event.InputMethodEvent
│   ├── java.awt.event.ItemEvent
│   ├── java.awt.event.TextEvent
│   └── java.awt.event.ComponentEvent
│       ├── java.awt.event.WindowEvent*
│       ├── java.awt.event.FocusEvent
│       └── java.awt.event.InputEvent
│           ├── java.awt.event.KeyEvent
│           │   └── javax.swing.event.MenuKeyEvent
│           └── java.awt.event.MouseEvent**
│               ├── java.awt.event.MouseWheelEvent
│               └── javax.swing.event.MenuDragMouseEvent

```

\* Clase de evento usada por el tipo de evento `WindowStateListener` y `WindowFocusListener`

\*\* Clase de evento usada por el tipo de evento `MouseMotionListener`

**Figura 4.4:** Clases `XXEvent` pasadas como parámetro en cada uno de los métodos de `XXListener`

Los eventos se pueden clasificar según distintos criterios dependiendo de las acciones que los generan y de qué información asociada llevan al generarse. Dependiendo de las acciones que los generan, existen dos clases de eventos:

- **Eventos de bajo nivel:** Representan de forma directa las acciones que realiza el usuario, es decir, la pulsación de teclas y el movimiento del ratón. Los eventos de este tipo en Swing están representados por eventos de tipo `Input`. Puesto que los eventos de bajo nivel se generan en los objetos de la clase `java.awt.Component`, clase padre de todos los componentes, todos los componentes generan eventos de bajo nivel.

- **Eventos de alto nivel o semánticos:** Representan acciones más complejas. Por ejemplo, seleccionar un elemento en una lista desplegable o arrastrar un elemento o pulsar un hiperenlace.

Dependiendo de la información que contienen los eventos, existen dos clases de eventos:

- **Eventos ligeros** – Sólo albergan la referencia al componente que generó el evento. Este tipo de eventos se usan cuando se pueden conocer detalles sobre el evento producido invocando métodos en el componente que originó dicho evento. Este tipo de eventos en Swing están representados por los eventos de tipo `Change`, que únicamente indican que algo ha cambiado. Como ya se ha comentado, para obtener una referencia al objeto que generó el evento, se utilizará el método `public Object getSource()` de la clase `EventObject`.
- **Eventos informativos** – Además de guardar una referencia al componente que generó el evento, tienen información sobre las características del evento que se ha producido. Con eventos de este tipo no suele ser necesario consultar al componente que originó el evento.

Por último, cabe mencionar que existen clases abstractas, llamadas `XXAdapter`, que implementan una o varias interfaces `XXListener`, pero dejan todos sus métodos vacíos, sin implementación. Son usadas para construir clases de gestión de eventos sin necesidad de implementar la interfaz `XXListener` y dejar vacíos los métodos que no se usen. En vez de esto, se hereda de la clase `XXAdapter` y únicamente se redefinen los métodos que interesen.

## 4.4. Imágenes

En este apartado se dan algunos detalles de Swing relativos a la gestión de imágenes que pueden ser de mucha utilidad.

### 4.4.1. Iconos

Un icono se representa en Swing como instancias de la interfaz `javax.swing.Icon`. Los iconos se pueden construir dinámicamente con código Java haciendo uso de la librería `Java2D`<sup>2</sup>, la cual no se estudia en

---

<sup>2</sup> <http://java.sun.com/products/java-media/2D/>

este libro. Por otro lado, los iconos también se pueden construir desde ficheros de imágenes.

En Swing una imagen se representa como una instancia de la clase `javax.swing.ImageIcon`, que implementa la interfaz `Icon`.

Si se quiere crear un icono partiendo de una imagen cargada desde una URL, se debe utilizar el siguiente esquema básico:

```
...
try {
    ...
    java.net.URL imageSource = ...
    Icon icono =
        new javax.swing.ImageIcon(
            javax.imageio.ImageIO.read(imageSource);
    ...
} catch (IOException e){}
...
```

Si la imagen se carga desde un fichero el esquema será:

```
...
try {
    ...
    java.io.File imageSource = ...
    Icon icono =
        new javax.swing.ImageIcon(
            javax.imageio.ImageIO.read(imageSource);
    ...
} catch (IOException e){}
...
```

Por último, si la imagen se carga desde un flujo de entrada se utilizará lo siguiente:

```
...
try {
    ...
    java.io.InputStream imageSource = ...
    Icon icono =
        new javax.swing.ImageIcon(
            javax.imageio.ImageIO.read(imageSource);
    ...
} catch (IOException e){}
...
```

El método estático `read(...)` de la clase `javax.imageio.ImageIO` es el encargado de leer los bytes de la imagen y convertirlos en una matriz de píxeles en memoria. El método se bloquea hasta que no ha terminado la carga completa de la imagen.

Si se necesita conocer el progreso de carga de la imagen, por ejemplo para mostrarlo en la interfaz gráfica, se debe usar el siguiente esquema (algo más complejo):

```
...
try {

    ImageInputStream iis =
        ImageIO.createImageInputStream(urlImagen.openStream());

    ImageReader ir = (ImageReader)
        ImageIO.getImageReaders(iis).next();

    ir.addIIIOReadProgressListener(
        new IIIOReadProgressListener() {

            public void sequenceStarted(
                ImageReader source, int minIndex) {}
            public void sequenceComplete(ImageReader source) {}
            public void imageStarted(ImageReader source,
                int imageIndex)
            {
                //Invocado cuando el progreso es 0 %
            }

            public void imageComplete(ImageReader source) {
                //Invocado cuando el progreso es 100 %
            }

            public void imageProgress(ImageReader source,
                float percentageDone) {
                //Invocado cuando el progreso es percentageDone %
            }

            public void thumbnailStarted(
                ImageReader source,
                int imageIndex,int thumbnailIndex) {}
            public void thumbnailProgress(
                ImageReader source, float percentageDone) {}
            public void thumbnailComplete(ImageReader source) {}
            public void readAborted(ImageReader source) {}
        });
}
```

```
ir.setInput(iis);  
Icon icon = new ImageIcon(ir.read(0));  
...  
} catch(IOException e){}  
...
```

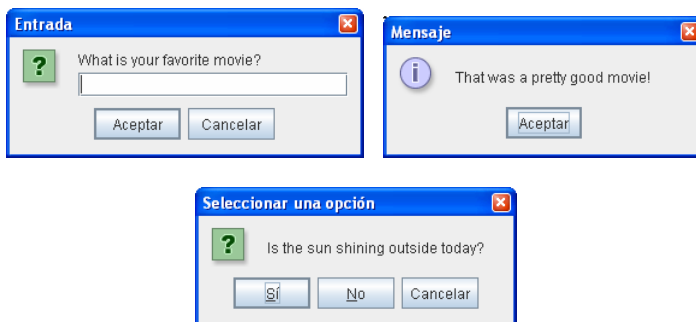
Para mostrar una imagen en la interfaz se usa una etiqueta (`JLabel`). Una etiqueta se puede configurar para que muestre solo un texto, una imagen o ambos. Para ello, dispone de diversos métodos, a continuación se muestran algunos:

- `public JLabel(Icon image)` - Construye una etiqueta que muestra la imagen pasada como parámetro.
- `public setIcon(Icon image)` - Muestra la imagen en la etiqueta.

#### 4.5. Paneles de opciones

Un panel de opciones es un cuadro de diálogo estándar que pide al usuario cierta información o bien informa de algo, solicitando aceptación o no por parte del usuario.

Con la librería Swing es posible crear paneles de opciones con las acciones más habituales (mostrar un mensaje, hacer una pregunta cuya respuesta es “Si” o “No”, etcétera) de una forma muy sencilla. Ejemplos de paneles de opciones se muestran en la figura 4.5.



**Figura 4.5:** Paneles de opciones predefinidos en Swing

Para mostrar un panel de opciones en una interfaz gráfica de usuario se usa la clase `javax.swing.JOptionPane`. Esta clase dispone de un buen

número de métodos, campos, etc., que pueden hacer pensar, a primera vista, que su uso sea complejo. Sin embargo, aunque son muchas las posibilidades de configuración, la mayoría de las veces, su uso se reduce a invocar alguno de los siguientes métodos estáticos (`showXxxDialog`):

- `int showConfirmDialog(...)` - pregunta y espera una respuesta Si/No/Cancel.
- `int showInputDialog(...)` - pide alguna entrada.
- `int showMessageDialog(...)` - informa al usuario sobre algo.
- `int showOptionDialog(...)` - muestra un diálogo genérico configurable con los parámetros del método.

Para mostrar un panel de opciones con un mensaje de error, se utiliza el siguiente esquema básico:

```
...
JOptionPane.showMessageDialog(parentComponent,
                               "message", "title",
                               JOptionPane.ERROR_MESSAGE);
...
```

El parámetro `parentComponent` es cualquier componente de la interfaz gráfica de usuario. Ese componente se usa para saber la ventana (`JFrame`) que se debe bloquear si el panel de opciones es modal. Hay que recordar que los paneles modales son aquellos que bloquean la ventana de aplicación para que no pueda ser usada hasta que se cierra el panel de opciones.

#### **Ejemplo 4.1 – Visor de Imágenes 1.0**

Para mostrar como usar imágenes y paneles de opciones se va a implementar una aplicación muy sencilla que permite cargar una imagen desde una URL. La aplicación está formada por los siguientes componentes:

- **Una barra de direcciones:** Siguiendo la idea de la barra de los navegadores. Servirá para que el usuario introduzca la URL de la imagen que desea cargar.
- **Un botón para iniciar la carga:** El botón aparecerá a la derecha de la barra de direcciones.

- **Una etiqueta:** Donde se mostrará la imagen cargada.
- **Otros componentes auxiliares:** Por ejemplo, el panel de desplazamiento (`JScrollPane`) para que muestre las barras si la imagen es más grande que el espacio que tiene disponible.

La interfaz gráfica de usuario de la aplicación se presenta en la figura 4.6, en este caso, visualizando una imagen ya cargada.



**Figura 4.6:** Interfaz de usuario del Visor de Imágenes 1.0

El código del visor de imágenes se muestra a continuación:

**libro/ejemplos/ejemplo4.1/AplicacionVisorImagenes.java**

```
public class AplicacionVisorImagenes {
    public AplicacionVisorImagenes() {
        new VentanaVisorImagenes();
    }
    public static void main(String args[]) {
        new AplicacionVisorImagenes();
    }
}
```



**libro/ejemplos/ejemplo4.1/VentanaVisorImagenes.java**

```
import javax.swing.JFrame;

public class VentanaVisorImagenes extends JFrame {

    public VentanaVisorImagenes() {

        this.setTitle("Visor de imagenes");

        this.setContentPane(new MalVisorImagenes());

        this.setSize(500,500);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);

    }
}
```

**libro/ejemplos/ejemplo4.1/MalVisorImagenes.java**

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

import javax.imageio.ImageIO;
import javax.swing.*;

public class MalVisorImagenes extends JPanel {

    private JLabel etqImagen;

    private JLabel etqURLImagen;
    private JTextField campoURLImagen;
    private JButton botonCarga;

    public MalVisorImagenes() {

        this.etqURLImagen = new JLabel("URL imagen: ");
        this.campoURLImagen = new JTextField(50);
        this.campoURLImagen.setText(
            "http://www.escet.urjc.es/~smontalvo/"+
            "interfacesgraficasenjava/imagen3.jpg");

        this.botonCarga = new JButton("Cargar imagen");
        this.botonCarga.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent arg0) {
                    iniciaCargaImagen();
                }
            }
        );
    }
}
```

```
});

this.setLayout(new BorderLayout());

JPanel panelControles = new JPanel();
panelControles.setLayout(new BorderLayout(10, 10));
panelControles.add(this.etqURLImagen,
    BorderLayout.WEST);
panelControles.add(this.campoURLImagen,
    BorderLayout.CENTER);
panelControles.add(this.botonCarga,
    BorderLayout.EAST);

panelControles.setBorder(BorderFactory
    .createEmptyBorder(10, 10, 10, 10));

this.add(panelControles, BorderLayout.NORTH);

this.etqImagen = new JLabel(
    "<HTML><div align='center'>" +
    "No hay imagen cargada<BR><BR>" +
    "Seleccione una URL y pulsa cargar</div>");

this.etqImagen.setHorizontalAlignment(
    SwingConstants.CENTER);

JScrollPane sp = new JScrollPane(this.etqImagen);
this.add(sp, BorderLayout.CENTER);
}

protected void iniciaCargaImagen() {

    final URL url;
    try {
        url = new URL(this.campoURLImagen.getText());
    } catch (MalformedURLException e) {
        JOptionPane.showMessageDialog(this,
            "URL inválida.", "Error",
            JOptionPane.ERROR_MESSAGE);

        return;
    }

    this.botonCarga.setEnabled(false);
    this.etqImagen.setText("Cargando imagen");
    cargaImagen(url);
}

protected void cargaImagen(URL urlImagen) {

    final Icon icono;
```

```

        try {
            icono = new ImageIcon(ImageIO.read(urlImagen));
        } catch (IOException e) {
            errorCargaImagen();
            return;
        }

        etqImagen.setText("");
        etqImagen.setIcon(icono);

        botonCarga.setEnabled(true);
    }

    private void errorCargaImagen() {

        JOptionPane.showMessageDialog(MalVisorImagenes.this,
            "Error al cargar la imagen.", "Error",
            JOptionPane.ERROR_MESSAGE);
        botonCarga.setEnabled(true);
    }
}

```

Para realizar pruebas se pueden usar las siguientes URLs:

- <http://www.escet.urjc.es/~smontalvo/interfacesgraficasenjava/imagen1.jpg>
- <http://www.escet.urjc.es/~smontalvo/interfacesgraficasenjava/imagen2.jpg>
- <http://www.escet.urjc.es/~smontalvo/interfacesgraficasenjava/imagen3.jpg>

Donde `imagen1.jpg` es una imagen relativamente pequeña y que tarda poco tiempo en descargarse, e `imagen3.jpg` una imagen relativamente grande, que tarda un poco más.

Al ejecutar la aplicación se observa que con imágenes grandes, que tardan mucho en descargarse, la interfaz gráfica de usuario no se redibuja correctamente y los controles no funcionan. Para evitarlo, han de usarse una serie de técnicas que permiten descargar la imagen en segundo plano. Estas técnicas se verán en el capítulo 7, por lo que se remite al lector a dicho capítulo para poder ver la solución del ejemplo anterior haciendo uso de dichas técnicas.

## 4.6. Diálogos

Los diálogos se representan en Swing como instancias de la clase `JDialog`. Son un tipo de ventanas que se asocian a un `JFrame` o a otro

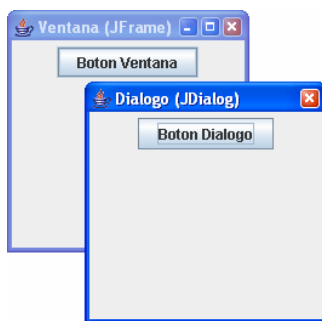
`JDialog`. Únicamente dispone de un botón para cerrarlo, pero no para minimizarlo ni maximizarlo. Puede configurarse como modal, lo cual hace que los componentes del `JFrame` o el `JDialog` asociado estén bloqueados mientras el `JDialog` sea visible. Para configurar un `JDialog` se utilizan la mayoría de los métodos usados en un `JFrame`, por ejemplo, `setLocation(...)`, `setVisible(...)`, `setSize(...)`, `setTitle(...)`, etc.

Algunos métodos representativos de `JDialog` son los siguientes:

- **`public JDialog(JFrame owner, String title)`** - Construye un `JDialog` asociado al `JFrame` pasado y con el título especificado.
- **`public JDialog(JDialog owner, String title)`** - Construye un `JDialog` asociado al `JDialog` pasado y con el título especificado.
- **`public void setModal(boolean modal)`** - Establece si el `JDialog` es modal.
- **`public void setVisible(boolean b)`** - Establece la visibilidad del cuadro de diálogo.

#### **Ejemplo 4.1 – Ejemplo de Dialogo**

Para mostrar el uso de diálogos, este ejemplo dispone de una ventana con un botón que al ser pulsado muestra un diálogo. En la figura 4.7 aparece la aplicación en el momento en que se muestra el diálogo.



**Figura 4.7:** Aplicación ejemplo de diálogo

El diálogo es modal y dispone de un botón que lo oculta. El sencillo código se muestra a continuación:

**libro/ejemplos/ejemplo4.2/EjemploDialogo.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

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

public class EjemploDialogo {

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

    public EjemploDialogo() {

        JFrame ventana = new JFrame("Ventana (JFrame)");
        ventana.setSize(200, 200);
        ventana.
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        final JDialog dialogo = createJDialog(ventana);
        JButton boton = new JButton("Boton Ventana");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dialogo.setVisible(true);
            }
        });

        JPanel panelContenido = new JPanel();
        panelContenido.add(boton);

        ventana.setContentPane(panelContenido);
        ventana.setVisible(true);
    }

    public JDialog createJDialog(JFrame ventana) {
        final JDialog dialogo =
            new JDialog(ventana, "Dialogo (JDialog)");
        dialogo.setModal(true);
        dialogo.setSize(200, 200);
        dialogo.setLocation(50, 50);

        JButton boton = new JButton("Boton Dialogo");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dialogo.setVisible(false);
            }
        });
    }
}
```

```
JPanel panelContenido = new JPanel();
panelContenido.add(boton);

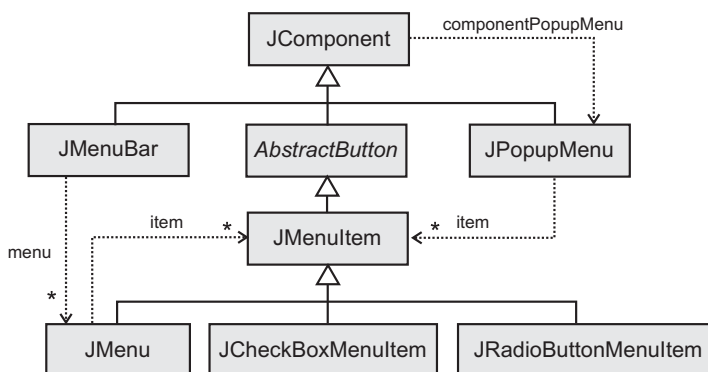
dialogo.setContentPane(panelContenido);
return dialogo;
}
```

Como puede verse en el código, la forma de manejar un diálogo es bastante similar a la forma de manejar un `JFrame`.

## 4.7. Menús

Un menú es un botón que al ser pulsado muestra un panel que contiene botones distribuidos de forma vertical y sin borde. Habitualmente aparecen en la barra de menús, una zona rectangular de la interfaz de usuario que suele aparecer en la parte superior de la ventana de aplicación. También pueden permanecer ocultos y mostrarse al pulsar el botón secundario del ratón, como ocurre con los menús contextuales. Los menús pueden contener botones, botones de chequeo, botones de radio y otros menús.

En el diagrama de clases de la figura 4.8 se muestran las principales clases que permiten gestionar menús en Swing.



**Figura 4.8:** Clases que permiten gestionar los menús en Swing

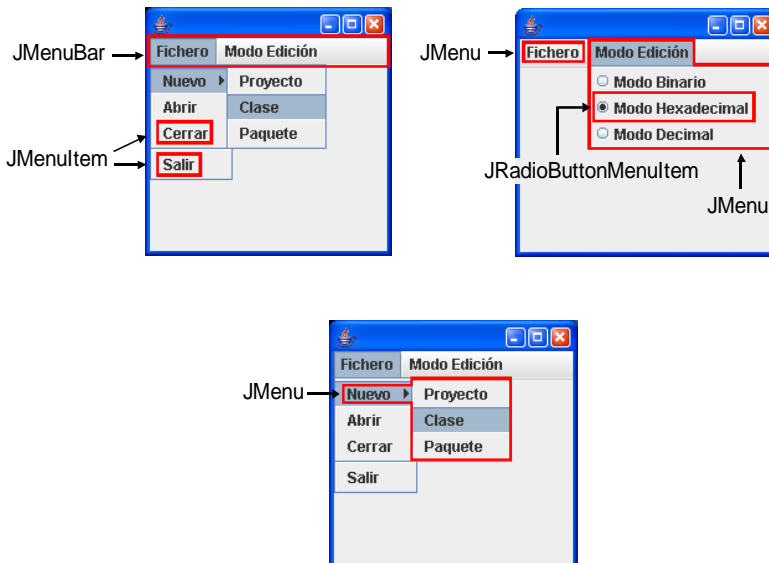
#### 4.7.1. Menús en barras de menús

Por ejemplo, la aplicación mostrada en la figura 4.9 muestra un menú de ejemplo con algunos de los elementos más relevantes.



**Figura 4.9:** Aplicación de ejemplo con un menú

En la figura 4.10 están marcadas las clases cuyos objetos representan las partes del menú.



**Figura 4.10:** Clases que permiten gestionar los menús en Swing

Las clases `JFrame`, `JDialog` y `JApplet` disponen del método `setJMenuBar(JMenuBar menuBar)` para establecer una barra de menús. La clase `JMenuBar` dispone del método `add(JMenu c)` para añadirle los `JMenu`. La clase `JMenu` dispone del método `add(JMenuItem menuItem)` para añadirle los botones o bien otros submenús. Además, dispone del método `addSeparator()` para insertar un separador.

### **Ejemplo 4.3 – Ejemplo de Menú 1.0**

La aplicación que muestra el menú de la figura 4.9 se muestra a continuación:

#### **libro/ejemplos/ejemplo4.3/EjemploMenu1.java**

```
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JRadioButtonMenuItem;

public class EjemploMenu1 extends JFrame {

    public static void main(String[] args) {
        new EjemploMenu1();
    }

    public EjemploMenu1() {
        JMenu menuA = creaMenuFichero();
        JMenu menuB = creaMenuModoEdicion();

        JMenuBar barraMenus = new JMenuBar();
        barraMenus.add(menuA);
        barraMenus.add(menuB);

        this.setJMenuBar(barraMenus);
        this.setSize(200,200);
        this.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    private JMenu creaMenuModoEdicion() {

        JRadioButtonMenuItem itemA =
            new JRadioButtonMenuItem("Modo Binario");
        JRadioButtonMenuItem itemB =
```



```

        new JRadioButtonMenuItem("Modo Hexadecimal");
        JRadioButtonMenuItem itemC =
            new JRadioButtonMenuItem("Modo Decimal");

        itemA.setSelected(true);

        ButtonGroup bg = new ButtonGroup();
        bg.add(itemA);
        bg.add(itemB);
        bg.add(itemC);

        JMenu menu = new JMenu("Modo Edición");
        menu.add(itemA);
        menu.add(itemB);
        menu.add(itemC);

        return menu;
    }

    private JMenu creaMenuFichero() {

        JMenuItem itemAA = new JMenuItem("Proyecto");
        JMenuItem itemAB = new JMenuItem("Clase");
        JMenuItem itemAC = new JMenuItem("Paquete");

        JMenu menuAA = new JMenu("Nuevo");
        menuAA.add(itemAA);
        menuAA.add(itemAB);
        menuAA.add(itemAC);

        JMenuItem itemB = new JMenuItem("Abrir");
        JMenuItem itemC = new JMenuItem("Cerrar");

        JMenuItem itemAD = new JMenuItem("Salir");

        JMenu menu = new JMenu("Fichero");
        menu.add(menuAA);
        menu.add(itemB);
        menu.add(itemC);
        menu.addSeparator();
        menu.add(itemAD);

        return menu;
    }
}

```

Como puede verse, los `JRadioButtonMenuItem` deben ser agrupados en un `ButtonGroup` al igual que ocurre con los `JRadioButton`.

La gestión de eventos en los botones del menú es igual que en cualquier otro botón, los `JMenuItem` generan eventos de tipo `Action` al ser pulsados. Y los botones `JRadioButtonMenuItem` y `JCheckBoxMenuItem` generan eventos de tipo `Item`.

#### **Ejemplo 4.4 – Ejemplo de Menú 2.0**

Para mostrar la gestión de eventos en los elementos del menú, en este ejemplo aparece un cuadro de diálogo cada vez que se pulsa una opción del mismo. Para facilitar la lectura, se han puesto en negrita aquellos cambios introducidos respecto al ejemplo anterior.

**libro/ejemplos/ejemplo4.4/EjemploMenu2.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JRadioButtonMenuItem;

public class EjemploMenu2 extends JFrame implements
ActionListener, ItemListener {
    public static void main(String[] args) {
        new EjemploMenu2();
    }

    public EjemploMenu2() {

        JMenu menuA = creaMenuFichero();
        JMenu menuB = creaMenuModoEdicion();

        JMenuBar barraMenus = new JMenuBar();
        barraMenus.add(menuA);
        barraMenus.add(menuB);

        this.setJMenuBar(barraMenus);
        this.setSize(200,200);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}
```

```
private JMenu creaMenuModoEdicion() {

    JRadioButtonMenuItem itemA =
        new JRadioButtonMenuItem("Modo Binario");
    JRadioButtonMenuItem itemB =
        new JRadioButtonMenuItem("Modo Hexadecimal");
    JRadioButtonMenuItem itemC =
        new JRadioButtonMenuItem("Modo Decimal");

    itemA.setSelected(true);

    ButtonGroup bg = new ButtonGroup();
    bg.add(itemA);
    bg.add(itemB);
    bg.add(itemC);

    itemA.addItemListener(this);
    itemB.addItemListener(this);
    itemC.addItemListener(this);

    JMenu menu = new JMenu("Modo Edición");
    menu.add(itemA);
    menu.add(itemB);
    menu.add(itemC);

    return menu;
}

private JMenu creaMenuFichero() {

    JMenuItem itemAA = new JMenuItem("Proyecto");
    JMenuItem itemAB = new JMenuItem("Clase");
    JMenuItem itemAC = new JMenuItem("Paquete");

    itemAA.addActionListener(this);
    itemAB.addActionListener(this);
    itemAC.addActionListener(this);

    JMenu menuAA = new JMenu("Nuevo");
    menuAA.add(itemAA);
    menuAA.add(itemAB);
    menuAA.add(itemAC);

    JMenuItem itemB = new JMenuItem("Abrir");
    JMenuItem itemC = new JMenuItem("Cerrar");

    itemB.addActionListener(this);
    itemC.addActionListener(this);

    JMenuItem itemAD = new JMenuItem("Salir");
```

```
        itemAD.addActionListener(this);

        JMenu menu = new JMenu("Fichero");
        menu.add(menuAA);
        menu.add(itemB);
        menu.add(itemC);
        menu.addSeparator();
        menu.add(itemAD);

        return menu;
    }

    public void actionPerformed(ActionEvent e) {
        JMenuItem item = (JMenuItem) e.getSource();
        JOptionPane.showMessageDialog(this,
                                     "Acción "+item.getText());
    }

    public void itemStateChanged(ItemEvent e) {
        if(e.getStateChange() == ItemEvent.SELECTED) {
            JMenuItem item = (JMenuItem) e.getSource();
            JOptionPane.showMessageDialog(this,
                                         "Seleccionado "+item.getText());
        }
    }
}
```

#### 4.7.2. Menús independientes

Como hemos visto en la sección anterior, los menús pueden estar incluidos en una barra de herramientas. También pueden ser independientes y mostrarse ante determinadas circunstancias.

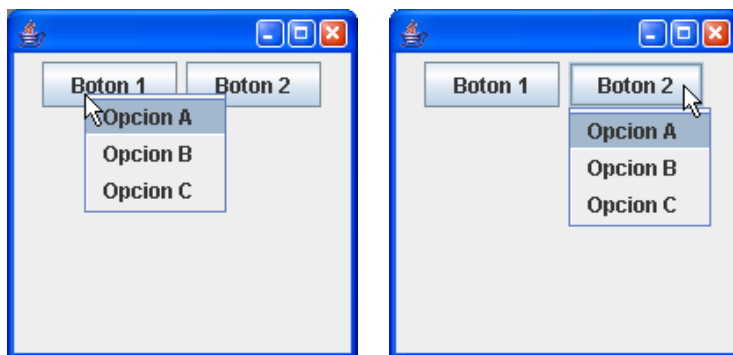
La clase que representa a los menús independientes es `JPopupMenu`. Los menús de esta clase pueden ser usados de dos formas distintas:

- **Menú contextual:** Aparece habitualmente al pulsar el botón secundario del ratón sobre el componente al que está asociado. Si este componente no dispone de menú contextual, por defecto mostrará el menú contextual asociado a su padre en la jerarquía de componentes.
- **Menú mostrado explícitamente:** Este menú puede aparecer en cualquier posición y ante cualquier situación que considere el programador. No está asociado a ningún componente.

Para asociar un menú contextual a un `JComponent`, se usa el método de la clase `JComponent` `setComponentPopupMenu(JPopupMenu jPopupMenu)`. De esta forma, cuando se pulse el botón secundario del ratón sobre el componente se mostrará el menú. En cambio, cuando se quiere mostrar un `JPopupMenu` explícitamente, se usa el método de la clase `JPopupMenu` `show(Component invoker, int x, int y)`. Las coordenadas `x` e `y` son relativas al sistema de coordenadas del componente `invoker`.

#### Ejemplo 4.5 – Ejemplo de Menú 3.0

En este ejemplo se muestra una aplicación con dos botones, el Botón 1 tiene asociado un `JPopupMenu` actuando como menú contextual, en cambio, el Botón 2 muestra explícitamente el `JPopupMenu` debajo del botón cuando es pulsado. En la figura 4.11 se muestra esta aplicación al mostrar cada uno de los menús.



**Figura 4.11:** Aplicación de ejemplo con menús `JPopupMenu`.

A continuación se muestra el código del ejemplo. En negrita aparecen resaltados los métodos que usan el `JPopupMenu` de las dos formas posibles:

**libro/ejemplos/ejemplo4.5/EjemploMenu3.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
```

```
import javax.swing.JPanel;
import javax.swing.JPopupMenu;

public class EjemploMenu3 extends JFrame {

    public static void main(String[] args) {
        new EjemploMenu3();
    }

    public EjemploMenu3() {

        //JPopupMenu como menú contextual de un componente
        JPopupMenu jPopupMenu1 = createJPopupMenuA();
        JButton botonMenuContextual = new JButton("Boton 1");
botonMenuContextual.
setComponentPopupMenu(jPopupMenu1);

        //JPopupMenu mostrado explícitamente al
        //pulsar el botón
        final JButton botonMostradorMenu = new JButton(
                                                    "Boton 2");
        final JPopupMenu jPopupMenu2 = createJPopupMenuA();
        botonMostradorMenu.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e) {
jPopupMenu2.show(
    botonMostradorMenu,
    0,botonMostradorMenu.getHeight());
                }
            });
        JPanel contentPane = new JPanel();
        contentPane.add(botonMenuContextual);
        contentPane.add(botonMostradorMenu);

        this.setContentPane(contentPane);
        this.setSize(200,200);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);

    }

    private JPopupMenu createJPopupMenuA() {

        JMenuItem itemA = new JMenuItem("Opcion A");
        JMenuItem itemB = new JMenuItem("Opcion B");
        JMenuItem itemC = new JMenuItem("Opcion C");

        JPopupMenu jPopupMenu = new JPopupMenu();

        jPopupMenu.add(itemA);
```

```
jPopupMenu.add(itemB);
jPopupMenu.add(itemC);

return jPopupMenu;
}

private JPopupMenu createJPopupMenuB() {

    JMenuItem itemA = new JMenuItem("Opcion 1");
    JMenuItem itemB = new JMenuItem("Opcion 2");
    JMenuItem itemC = new JMenuItem("Opcion 3");

    JPopupMenu jPopupMenu = new JPopupMenu();

    jPopupMenu.add(itemA);
    jPopupMenu.add(itemB);
    jPopupMenu.add(itemC);

    return jPopupMenu;
}
}
```





## Capítulo 5

### Creación de componentes personalizados

Una vez que se sabe cómo organizar el código de gestión de eventos, estudiado en el capítulo segundo y se sabe cómo organizar los diferentes elementos gráficos que forman parte de la interfaz, estudiado en el capítulo tercero, en este capítulo se pretende estudiar la forma de crear componentes propios, totalmente equivalentes a los componentes estándar de la API Swing y capaces de generar eventos de alto nivel según la funcionalidad de la aplicación concreta.

#### 5.1. Introducción

En este momento y después de lo estudiado en el capítulo anterior, se puede intuir lo interesante que puede resultar crear componentes propios; es decir, no limitarse única y exclusivamente a la utilización de los componentes existentes o predefinidos de la librería Swing.

Normalmente, cuando se construyen aplicaciones con interfaces gráficas de usuario es necesario tener ciertas funcionalidades que no proporcionan los componentes predefinidos de Swing. Por ejemplo, se podría necesitar una calculadora o un calendario a través de una interfaz gráfica y, para obtener esa funcionalidad conjunta lo lógico sería crear un componente propio que representara dicha calculadora o calendario. Para crear un componente propio, lo que se suele hacer es utilizar un contenedor `JPanel`, configurarlo y añadir los componentes que se necesiten para lograr esa funcionalidad conjunta que se persigue. Por ejemplo, en el caso de la calculadora, habría que añadir una serie de botones, un campo

de texto, etc. Pero, para que un componente propio sea igual que cualquier componente predefinido de Swing, es necesario que sea capaz de generar eventos.

El hecho de construir componentes propios que sean capaces de generar eventos, va a permitir la reutilización de código y, también, tener nuevos componentes que serán como cualquier otro componente ya existente, pero con una funcionalidad más adaptada a las necesidades de la aplicación concreta.

El resto del capítulo se estructura como sigue: un primer apartado donde se presenta un recordatorio de eventos. Y un segundo y último apartado en el que se ve la creación de eventos propios.

## 5.2. Recordatorio de eventos

Antes de continuar, se considera conveniente recordar cuáles son las clases, interfaces y métodos que entran en juego cuando se trabaja con eventos en la construcción de interfaces gráficas de usuario con Java.

Por un lado están las interfaces `XXListener`, siendo `xx` el tipo de evento. Estas interfaces serán implementadas por los gestores de eventos. Cada uno de los métodos de estas interfaces indicará el evento concreto que se puede producir. Cualquier interfaz `XXListener` siempre hereda de la interfaz `java.util.EventListener`.

Por otro lado se tienen las clases `XXEvent`. Se trata de clases utilizadas como único parámetro de los métodos de las interfaces `XXListener`. Cualquier clase `XXEvent` siempre hereda de la clase `java.util.EventObject`. Podría tratarse de un evento ligero si sólo se guarda la fuente del evento o bien, podría ser un evento informativo si, además, tiene información adicional sobre el evento que se ha producido.

Por último, conviene recordar los métodos `addXXListener(XXListener listener)` y `removeXXListener(XXListener listener)`. Estos métodos aparecen en aquellos componentes que generan los eventos de tipo `xx`. Sirven para asociar el código de gestión de eventos o bien, para eliminar dicha asociación.

### 5.3. Creación de eventos propios

Si se crea un componente propio y se pretende que genere eventos, hay una primera decisión a la que enfrentarse: ¿los eventos existentes en la API se ajustan a las necesidades del evento particular que se tiene que generar? Por ejemplo, el evento de tipo `ChangeEvent` del paquete `javax.swing.event` es un evento ligero que se genera cuando el objeto cambia, pero no ofrece información adicional. En el caso de que, para la aplicación que se esté desarrollando sirva con esa funcionalidad, entonces se podrían reutilizar las clases `ChangeEvent` y `ChangeListener`. Si, por el contrario, lo que se quiere es crear un evento informativo; es decir, que aporte más información además de la fuente del evento, entonces la única posibilidad consiste en crear un evento propio.

Se va a suponer la creación de una interfaz gráfica de usuario que represente un tablero de fichas, para ello se crea una clase `TableroFichas` y se quiere que genere un evento cada vez que se mueva una ficha. Lo primero que hay que hacer es darle un nombre al evento concreto, por ejemplo `Move`. Entonces, para crear eventos del tipo `Move` se creará la clase `MoveEvent` y la interfaz `MoveListener`:

- La clase `MoveEvent` heredaré de la clase `EventObject` y tendrá un constructor con todos los parámetros que sean requeridos. El objeto fuente del evento es siempre obligatorio. También se pasarán la fila y columna origen y la fila y columna destino del movimiento. Y se crearán métodos de acceso para acceder a esta información.

```
o public MoveEvent(Object fuente, int filaOrigen,
                    int columnaOrigen, int filaDestino,
                    int columnaDestino)
o public int getFilaOrigen()
o public int getFilaDestino()
o public int getColumnaOrigen()
o public int getColumnaDestino()
```

- La interfaz `MoveListener` tendrá tantos métodos como eventos concretos se quieran generar. Se va a suponer que sólo existe un evento concreto, llamado `movementPerformed`. De esta forma, esta interfaz tendrá el siguiente método:

```
o public void movementPerformed(MoveEvent e)
```

En este momento ya se han creado las clases e interfaces necesarias para representar los eventos de tipo `Move`. Ahora sólo queda conseguir

que el componente propio creado, de la clase `TableroFichas` y que hereda de `JPanel`, pueda asociar código a esos eventos y generarlos cuando sea necesario.

Si se piensa en el funcionamiento de los eventos, es fácil darse cuenta de que en la implementación del método `addXXListener(XXListener listener)`, se guarda en alguna estructura de datos una referencia al `listener` pasado como parámetro. La implementación de método `removeXXListener(XXListener listener)` implica lo contrario, borrar de esa estructura de datos el `listener` pasado.

Como ya se sabe, todos los componentes de Swing son capaces de generar eventos; por tanto, todos los componentes tienen una estructura de datos para guardar referencias a los gestores de eventos que se asocian a ellos. Esta estructura de datos es un atributo de la clase `EventListenerList` del paquete `javax.swing.event`, llamado `listenerList`.

Cuando se necesita generar un evento, se recorre la estructura de datos buscando aquellos `listener` interesados por el tipo de evento actual y, en cada uno de los interesados, se invoca el método del evento concreto. Por ejemplo, si se quisiera elevar un evento de tipo `Move`, lo único que habría que hacer sería invocar el método `movementPerformed(...)` en todos los `listener` que estén interesados por los eventos de tipo `Move`. El código que recorre la estructura de datos e invoca el método correspondiente se suele introducir en un método llamado `fireYY(XXEvent e)`, donde `YY` es el nombre del evento en concreto; esto se hace, entre otras cosas, para organizar el código. En el ejemplo que se está tratando, el método se llamaría `fireMovementPerformed(MoveEvent event)`.

La forma de manejar esta estructura de datos se observa en la implementación parcial de la clase `TableroFichas` que se muestra a continuación:

**libro/ejemplos/ejemplo5.0/TableroFichas.java**

```
import javax.swing.*;

public class TableroFichas extends JPanel {

    //Algunos atributos...
```

```

public TableroFichas() {
    //Cuerpo del constructor...
}

public void addMoveListener(MoveListener l) {
    listenerList.add(MoveListener.class, l);
}

public void removeChangeListener(MoveListener l) {
    listenerList.remove(MoveListener.class, l);
}

//Método invocado cuando se quiere generar un evento
protected void fireMovementPerformed(MoveEvent event) {

    Object[] listeners = listenerList.getListenerList();

    for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==MoveListener.class) {
            ((MoveListener)listeners[i+1])
                .movementPerformed(event);
        }
    }
}

//Otros métodos ....
}

```

La implementación de la clase `MoveEvent` sería la siguiente:

**libro/ejemplos/ejemplo5.0/MoveEvent.java**

```

import java.util.EventObject;

public class MoveEvent extends EventObject {

    private int filaOrigen;
    private int columnaOrigen;

    private int filaDestino;
    private int columnaDestino;

    public MoveEvent(Object fuente, int filaOrigen,
                     int columnaOrigen, int filaDestino,
                     int columnaDestino) {
        super(fuente);

        this.filaDestino = filaDestino;
        this.filaOrigen = filaOrigen;
        this.columnaDestino = columnaDestino;
    }
}

```

```
        this.columnaOrigen = columnaOrigen;
    }
    public int getFilaOrigen() {
        return filaOrigen;
    }
    public int getFilaDestino() {
        return filaDestino;
    }
    public int getColumnaOrigen() {
        return columnaOrigen;
    }
    public int getColumnaDestino() {
        return columnaDestino;
    }
}
```

Por último, la implementación de la interfaz `MoveListener` sería:

**libro/ejemplos/ejemplo5.0/MoveListener.java**

```
import java.util.EventListener;

public interface MoveListener extends EventListener {

    public void movementPerformed(MoveEvent e);

}
```

Siguiendo esta estructura básica se pueden crear eventos del tipo y subtipos que se quiera. Para que el componente personalizado genere un evento bastaría invocar el método `fireYY(...)` correspondiente para ejecutar el código de gestión de eventos. De forma habitual, en los componentes propios los eventos son generados a causa de otros eventos producidos en los componentes contenidos en el componente personalizado. Se puede decir que se van **encadenando los eventos**, de eventos más sencillos y de más bajo nivel a eventos más elaborados y de alto nivel.

Se pretende modificar la aplicación de traducción 9.0 para que el panel que muestra los botones de radio sea un componente que genera eventos. Este componente será reutilizable y simple de utilizar.

El componente se construirá indicando el número de filas y columnas en las que debe distribuir los botones de radio. Posteriormente, se irá invocando un método para crear cada una de las opciones. Se utilizará un administrador de distribución `GridLayout` para la distribución de los componentes. Cuando el botón seleccionado cambie, este componente debe generar eventos.

No se quiere reutilizar ningún evento existente, por lo que se creará un evento propio. Este evento se llamará `Radio`. El evento concreto se llamará `radioSelectionChanged`. Se tratará de un evento informativo e indicará el número de radio pulsado, comenzando por 0.

La interfaz gráfica de usuario no ha cambiado con respecto a la creada en la aplicación de traducción 9.0.

Las nuevas clases son: `RadioEvent` y `PanelRadioButtons`; la nuevo interfaz es: `RadioListener`. Además, se ha modificado la clase `PanelSeleccionIdioma` para que use el componente `PanelRadioButtons`.

#### **libro/ejemplos/ejemplo5.1/RadioEvent.java**

```
import java.util.EventObject;

public class RadioEvent extends EventObject {

    private int numOpcion;

    public RadioEvent(Object source, int numOpcion) {
        super(source);
        this.numOpcion = numOpcion;
    }

    public int getNumOpcion() {
        return numOpcion;
    }
}
```

#### **libro/ejemplos/ejemplo5.1/RadioListener.java**

```
import java.util.EventListener;
```

```
public interface RadioListener extends EventListener {  
    void radioSelectionChanged(RadioEvent e);  
}
```

**libro/ejemplos/ejemplo5.1/PanelRadioButtons.java**

```
import javax.swing.*.*;  
import javax.swing.event.*;  
import java.awt.*.*;  
import java.awt.event.*;  
  
public class PanelRadioButtons extends JPanel {  
  
    private ButtonGroup grupo = new ButtonGroup();  
    private int numBotones = 0;  
  
    public PanelRadioButtons(int numFilas, int numColumnas) {  
        this.setLayout(new GridLayout(numFilas, numColumnas));  
    }  
  
    public void addRadioButton(String texto,  
                               boolean seleccionado) {  
  
        JRadioButton boton = new JRadioButton(texto);  
        grupo.add(boton);  
        boton.setSelected(seleccionado);  
        this.add(boton);  
  
        final int numBoton = numBotones;  
  
        boton.addItemListener(new ItemListener() {  
            public void itemStateChanged(ItemEvent e) {  
                if (e.getStateChange() == ItemEvent.SELECTED) {  
                    botonRadioSeleccionado(numBoton);  
                }  
            }  
        });  
  
        this.numBotones++;  
    }  
  
    protected void botonRadioSeleccionado(int numBoton) {  
        this.fireRadioSelectionChanged(new RadioEvent(this,  
                                                         numBoton));  
    }  
  
    public void addRadioListener(RadioListener l) {  
        listenerList.add(RadioListener.class, l);  
    }  
}
```



```

    public void removeRadioListener(RadioListener l) {
        listenerList.remove(RadioListener.class, l);
    }

    protected void fireRadioSelectionChanged
        (RadioEvent event) {

        Object[] listeners = listenerList.getListenerList();

        //Método invocado cuando se quiere generar un evento
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == RadioListener.class) {
                ((RadioListener) listeners[i + 1])
                    .radioSelectionChanged(event);
            }
        }
    }
}

```

**libro/ejemplos/ejemplo5.1/PanelSeleccionIdioma.java**

```

import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.JPanel;

public class PanelSeleccionIdioma extends JPanel {

    public static final int IDIOMA_ORIGEN = 0;
    public static final int IDIOMA_DESTINO = 1;

    private Traductor traductor;

    private int tipoIdioma;
    private PanelTraductor panelTraductor;
    private PanelRadioButtons panelRadioButtons;

    public PanelSeleccionIdioma(int tipoIdioma,
        Traductor traductor,
        PanelTraductor panelTraductor) {

        this.panelTraductor = panelTraductor;
        this.tipoIdioma = tipoIdioma;
        this.traductor = traductor;

        String cadenaTipo;

        if (tipoIdioma == IDIOMA_ORIGEN) {
            cadenaTipo = " Idioma Origen ";

```

```
    } else {
        cadenaTipo = " Idioma Destino ";
    }
    this.setBorder(BorderFactory
        .createTitledBorder(cadenaTipo));
    this.setMinimumSize(new Dimension(110, 110));
    this.setPreferredSize(new Dimension(110, 110));

    panelRadioButtons =
        new PanelRadioButtons(Traductor.NUM_IDIOMAS, 1);

    this.add(panelRadioButtons);

    for (int i = 0; i < Traductor.NUM_IDIOMAS; i++) {

        boolean seleccionado =
            (tipoIdioma == IDIOMA_ORIGEN &&
             i == traductor.getIdiomaOrigen())
            || (tipoIdioma == IDIOMA_DESTINO &&
              i == traductor.getIdiomaDestino());

        panelRadioButtons.
            addRadioButton(Traductor.getCadenaIdioma(i),
                           seleccionado);

    }
    panelRadioButtons.addRadioListener(
        new RadioListener() {
            public void radioSelectionChanged(RadioEvent e) {
                botonRadioSeleccioado(e.getNumOpcion());
            }
        });
}

public void botonRadioSeleccioado(int numBoton) {

    if (this.tipoIdioma == IDIOMA_ORIGEN) {
        traductor.setIdiomaOrigen(numBoton);
    } else if (this.tipoIdioma == IDIOMA_DESTINO) {
        traductor.setIdiomaDestino(numBoton);
    } else {
        throw new Error();
    }

    this.panelTraductor.muestraIdiomas();

}
}
```

## Capítulo 6

### Diseño de aplicaciones con interfaz gráfica: separación de datos e interfaz

Los componentes y los eventos son los pilares básicos en la construcción de una interfaz gráfica de usuario en Swing. En este capítulo se verá cómo están contruidos internamente los componentes, lo que permitirá añadir más funcionalidad a los programas que se desarrollen. Además, se mostrará cómo el principio básico de construcción de los componentes se puede aplicar en cualquier aplicación con interfaz gráfica a cualquier escala.

#### 6.1. Introducción

Los componentes de Swing basan su construcción en una arquitectura denominada model-view-controller (MVC) que proviene de SmallTalk. Esta arquitectura establece que en la programación de aplicaciones con interfaz gráfica de usuario deben aparecer tres partes fundamentales:

- **Modelo (model)** – Parte que albergará los datos y cuya construcción será totalmente ajena a la forma en que se representen esos datos en la interfaz gráfica.
- **Vista (view)** – Parte encargada de la representación gráfica de los datos.
- **Controlador (controller)** – Parte que se ocupará de la interpretación de las entradas del usuario, las cuales permiten modificar el modelo (datos) de forma adecuada.

La arquitectura MVC puede aplicarse a muchos niveles en la programación de una aplicación. Se puede aplicar, por ejemplo, para la creación de un simple botón de la interfaz gráfica de usuario y se puede aplicar también, por ejemplo, en la estructuración de una aplicación distribuida en Internet, en la cual se observan las tres partes fundamentales.

En este libro se va a estudiar es la arquitectura MVC aplicada en dos niveles:

- **Construcción interna de componentes** - Internamente los componentes de Swing utilizan la arquitectura MVC. Se verá con más detalle cómo se realiza esta implementación y qué ventajas ofrece.
- **Construcción de una aplicación con interfaz gráfica** – Se guiará el desarrollo de las aplicaciones con interfaz gráfica de usuario aplicando la arquitectura MVC.

El resto del capítulo presenta los siguientes apartados: el primero describe cómo se han construido internamente los componentes de Swing. El siguiente y último apartado se centra en la construcción de la aplicación completa usando MVC.

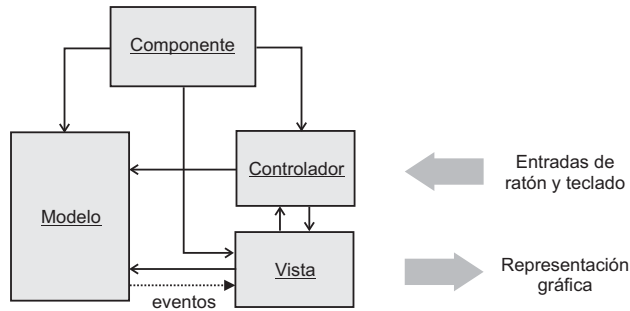
## 6.2. Construcción interna de componentes

En este apartado se verá cómo implementar los conceptos generales de la arquitectura MVC para construir internamente los componentes de la interfaz gráfica de usuario, y las ventajas que esto conlleva.

Hasta ahora, se ha visto que los componentes de la interfaz gráfica de usuario son los únicos objetos capaces de generar o elevar eventos. Se puede asociar código a estos eventos, el cual se ejecutará cuando el evento se produzca. Pero, la generación de eventos no es exclusiva de los componentes de la interfaz gráfica de usuario. Por ejemplo, un temporizador puede configurarse para generar un evento cada 20 minutos. En realidad, cualquier objeto puede generar eventos cuando su estado cambia si se implementa su clase para que tenga esta capacidad.

En una primera aproximación, la implementación interna de cada componente se realiza delegando la funcionalidad en tres objetos: un objeto será el encargado de albergar los datos; otro de representar

gráficamente esos datos y el tercero de interpretar las acciones del usuario. Estos objetos se relacionan entre sí de la forma que se presenta en la figura 6.1.



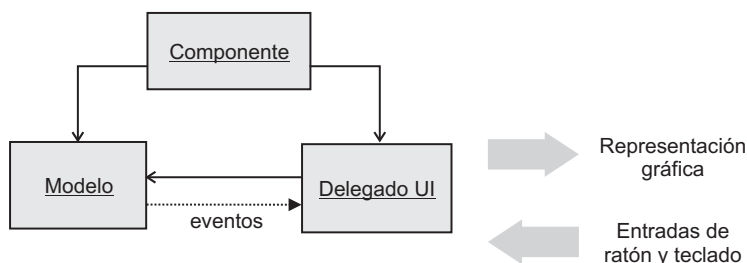
**Figura 6.1:** Relaciones de los objetos de la implementación interna de un componente

La responsabilidad de cada uno de los objetos anteriores es la siguiente:

- **Modelo** – Objeto encargado de almacenar los datos. Como puede observarse en la figura, este objeto eleva eventos cuando su estado cambia. Por ejemplo, si el componente es un `JCheckbox`, el modelo es el objeto que guarda la información sobre si el botón de chequeo está seleccionado o no. Y cada vez que su estado cambie el objeto elevará un evento.
- **Vista** – Objeto encargado de representar visualmente los datos que están almacenados en el modelo. Este objeto cambia la representación visual cada vez que el modelo eleva un evento de cambio. Si el evento es ligero, la vista debe preguntar al modelo sobre su estado para poder visualizarle; si el evento es informativo, con la información que viene en el propio evento es suficiente para poder representar los datos.
- **Controlador** – Es el responsable de interpretar las entradas del usuario. En el ejemplo del botón de chequeo este objeto interpreta las teclas que pulsa el usuario, y si es la tecla espacio cambia el estado del botón. En los componentes más complejos, la vista y el controlador tienen que intercambiar mucha información.

Los desarrolladores de Swing se basaron en esta idea para construir los componentes, pero realizaron algunos cambios importantes: fusionaron en un solo objeto la responsabilidad de visualización y control, debido al fuerte acoplamiento que tienen y a la complejidad de la interacción entre ambos. A esta nueva clase se la denomina **delegado UI** (UI delegate). De esta forma, los componentes de Swing tienen lo que se conoce como **arquitectura modelo-delegado** o **arquitectura de modelo separable**.

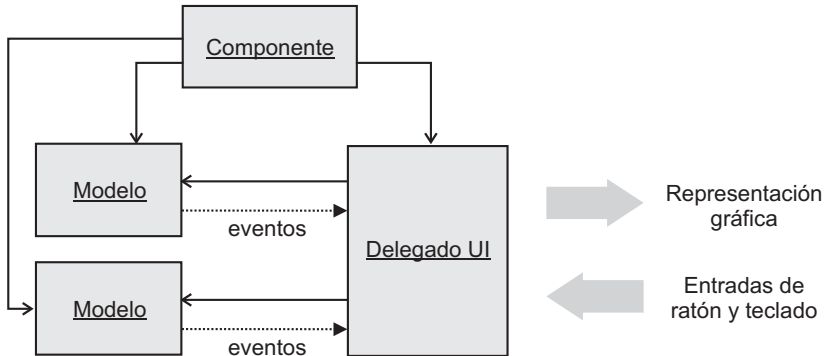
Los componentes tienen principalmente dos atributos: el modelo y el delegado. Estos atributos están enlazados como se muestra en la figura 6.2.



**Figura 6.2:** Estructura de los componentes con modelo de datos y delegado

Esta es la estructura general de la mayoría de los componentes en Swing, pero existen algunas excepciones a esta regla. Existen algunos componentes que tienen dos modelos para albergar datos de distinta naturaleza en cada uno de ellos. Por ejemplo, una lista de elementos guarda en un modelo los elementos y en otro modelo guarda cuáles de ellos están seleccionados. Con dos modelos, la arquitectura quedaría como se muestra en la figura 6.3.

También existen otros componentes que no tienen ningún modelo y los datos se representan como atributos convencionales. Estos atributos no elevan eventos y, por tanto, no pueden considerarse como modelo. La visualización se actualizará cuando se cambie el valor de estos atributos desde los métodos del componente. Por ejemplo, una etiqueta no tiene modelo, el delegado cambia la representación textual cuando se cambia el atributo de tipo `String` invocando el método `setText()` de la clase `javax.swing.JLabel`.



**Figura 6.3:** Estructura de componentes con varios modelos de datos y delegado

### 6.2.1. *Ventajas de la arquitectura modelo – delegado*

Lo que se hace siguiendo la arquitectura modelo – delegado es separar las funciones de pintado y gestión de la entrada de usuario, de las funciones de gestión de los datos en los componentes. Esto presenta una serie de ventajas:

- **Delegado** – Al estar muy poco acoplados los datos y la forma en la que se visualizan, es sencillo crear varias formas de representar los datos. Esto es especialmente útil cuando se quiere que la aplicación que se desarrolla tenga “aspectos” o “skin” diferentes.

En Java, como ya se ha comentado, a cada uno de estos aspectos se les denomina **look & feel**. La versión estándar de Java dispone de varios: un look & feel propio, llamado Metal; el de Windows; el de Solaris y el de MacOS, en equipos de Apple. No obstante, existen numerosos look & feel que se pueden utilizar y están disponibles en la red.

- **Modelo** – Varias ventajas aparecen cuando el modelo, el encargado de almacenar y gestionar los datos, se encuentra en un objeto distinto al componente. Las más importantes son las siguientes:

### ○ **Varios componentes muestran el mismo estado**

Al comunicarse el modelo con el delegado mediante eventos, se pueden tener varios componentes asociados al mismo modelo. Esto permite que el mismo dato se esté visualizando en dos sitios a la vez y sus datos siempre sean consistentes. Si en alguno de los componentes se modifica el estado del modelo, se elevará un evento que hará que se actualice la visualización en todos los componentes.

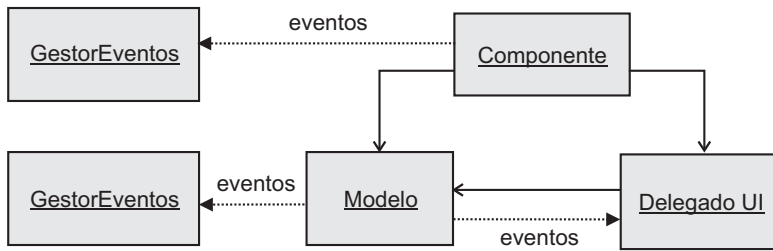
### ○ **Separación de responsabilidades entre el componente y el modelo**

Al poder trabajar directamente con los modelos es posible modificar los datos con métodos adecuados, sin necesidad de conocer la forma en que se visualizarán esos datos. Esto hace que los métodos sobre los aspectos visuales de un componente, estén en el componente y los métodos de la gestión de los propios datos, se encuentren en el modelo. Cuando cambie de estado el modelo, la representación visual cambiará.

### ○ **Eventos en el modelo**

Además de asociar código a los eventos que generan los componentes, se puede asociar código a los eventos que generan los modelos. En algunas ocasiones esto es inevitable, ya que existen eventos que se generan solo en el modelo y no en el propio componente. Los componentes de texto como `JTextField`, `JTextArea`, `JEditorPane`, etc., son algunos de los componentes en los que cobran mayor importancia sus modelos, al permitir gestionar características del texto en sí, no del componente en el que se representan. En la figura 6.4 se muestra como los gestores de eventos pueden asociarse tanto al componente como a su modelo.





**Figura 6.4:** Estructura de componentes con varios modelos de datos y delegado

### 6.2.2. Implementación de la arquitectura modelo – delegado en Swing

Para cada uno de los componentes de la interfaz gráfica de usuario existe una interfaz para su modelo y una clase para su delegado UI. Algunos componentes comparten la misma interfaz para definir su modelo. En la tabla 6.1 se muestran las interfaces para cada uno de los modelos de los componentes de la interfaz gráfica de usuario.

Componente	Modelo
<code>javax.swing.AbstractButton</code>	<code>ButtonModel</code>
<code>javax.swing.JButton</code>	<code>ButtonModel</code>
<code>javax.swing.JMenuItem</code>	<code>ButtonModel</code>
<code>javax.swing.JCheckBoxMenuItem</code>	<code>ButtonModel</code>
<code>javax.swing.JMenu</code>	<code>ButtonModel</code>
<code>javax.swing.JRadioButtonMenuItem</code>	<code>ButtonModel</code>
<code>javax.swing.JToggleButton</code>	<code>ButtonModel</code>
<code>javax.swing.JCheckBox</code>	<code>ButtonModel</code>
<code>javax.swing.JRadioButton</code>	<code>ButtonModel</code>
<code>javax.swing.JScrollBar</code>	<code>BoundedRangeModel</code>
<code>javax.swing.JSeparator</code>	No tiene modelo
<code>javax.swing.JPopupMenu.Separator</code>	No tiene modelo
<code>javax.swing.JToolBar.Separator</code>	No tiene modelo
<code>javax.swing.JSlider</code>	<code>BoundedRangeModel</code>
<code>javax.swing.JSpinner</code>	<code>SpinnerModel</code>
<code>javax.swing.JLabel</code>	No tiene modelo
<code>javax.swing.JList</code>	<code>ListModel</code> , <code>ListSelectionModel</code>
<code>javax.swing.JComboBox</code>	<code>ComboBoxModel</code>
<code>javax.swing.JProgressBar</code>	<code>BoundedRangeModel</code>
<code>javax.swing.JColorChooser</code>	No tiene modelo
<code>javax.swing.JFileChooser</code>	No tiene modelo
<code>javax.swing.JOptionPane</code>	No tiene modelo

javax.swing.JPopupMenu	No tiene modelo
javax.swing.JMenuBar	No tiene modelo
javax.swing.JToolBar	No tiene modelo
javax.swing.JToolTip	No tiene modelo
javax.swing.JTabbedPane	SingleSelectionModel
javax.swing.JScrollPane	No tiene modelo
javax.swing.JSplitPane	No tiene modelo
javax.swing.JLayeredPane	No tiene modelo
javax.swing.JDesktopPane	No tiene modelo
javax.swing.JPanel	No tiene modelo
javax.swing.JTable	TableModel, TableColumnModel
javax.swing.JTree	TreeModel, TreeSelectionModel
javax.swing.text.JTextComponent	Document
javax.swing.JEditorPane	Document
javax.swing.JTextPane	Document
javax.swing.JTextArea	Document
javax.swing.JTextField	Document
javax.swing.JFormattedTextField	Document
javax.swing.JPasswordField	Document
javax.swing.JInternalFrame	No tiene modelo
java.awt.Panel	No tiene modelo
java.awt.Window	No tiene modelo
java.awt.Dialog	No tiene modelo
javax.swing.JDialog	No tiene modelo
java.awt.Frame	No tiene modelo
javax.swing.JFrame	No tiene modelo
javax.swing.JWindow	No tiene modelo

**Tabla 6.1:** Interfaces de los modelos de los componentes

De forma paralela, existe una clase por cada delegado, aunque, por otro lado, existen componentes que no tienen delegado. Los componentes que no tienen delegado deben interpretar directamente la entrada del usuario, o bien lo realiza el sistema operativo, como en el caso de las ventanas. Además, deben elegir otras formas de pintarse; por ejemplo, indicándoselo a otro componente, o haciéndolo directamente. En

cualquier caso, no es necesario conocer realmente cómo se realiza para la construcción de la mayoría de las aplicaciones.

Una lista de los delegados existentes se puede ver en la tabla 6.2.

Componente	Delegado UI
<code>javax.swing.AbstractButton</code>	No tiene delegado
<code>javax.swing.JButton</code>	<code>BasicButtonUI</code>
<code>javax.swing.JMenuItem</code>	<code>BasicMenuItemUI</code>
<code>javax.swing.JCheckBoxMenuItem</code>	<code>BasicCheckBoxMenuItemUI</code>
<code>javax.swing.JMenu</code>	<code>BasicMenuUI</code>
<code>javax.swing.JRadioButtonMenuItem</code>	<code>BasicRadioButtonMenuItemUI</code>
<code>javax.swing.JToggleButton</code>	<code>BasicToggleButtonUI</code>
<code>javax.swing.JCheckBox</code>	<code>BasicCheckBoxUI</code>
<code>javax.swing.JRadioButton</code>	<code>BasicRadioButtonUI</code>
<code>javax.swing.JScrollBar</code>	<code>BasicScrollBarUI</code>
<code>javax.swing.JSeparator</code>	<code>BasicSeparatorUI</code>
<code>javax.swing.JPopupMenu.Separator</code>	<code>BasicSeparatorUI</code>
<code>javax.swing.JToolBar.Separator</code>	<code>BasicSeparatorUI</code>
<code>javax.swing.JSlider</code>	<code>BasicSliderUI</code>
<code>javax.swing.JSpinner</code>	<code>BasicSpinnerUI</code>
<code>javax.swing.JLabel</code>	<code>BasicLabelUI</code>
<code>javax.swing.JList</code>	<code>BasicListUI</code>
<code>javax.swing.JComboBox</code>	<code>BasicComboBoxUI</code>
<code>javax.swing.JProgressBar</code>	<code>BasicProgressBarUI</code>
<code>javax.swing.JColorChooser</code>	<code>BasicColorChooserUI</code>
<code>javax.swing.JFileChooser</code>	<code>BasicFileChooserUI</code>
<code>javax.swing.JOptionPane</code>	<code>BasicOptionPaneUI</code>
<code>javax.swing.JPopupMenu</code>	<code>BasicPopupMenuUI</code>
<code>javax.swing.JMenuBar</code>	<code>BasicMenuBarUI</code>
<code>javax.swing.JToolBar</code>	<code>BasicToolBarUI</code>
<code>javax.swing.JToolTip</code>	<code>BasicToolTipUI</code>
<code>javax.swing.JTabbedPane</code>	<code>BasicTabbedPaneUI</code>
<code>javax.swing.JScrollPane</code>	<code>BasicScrollPaneUI</code>
<code>javax.swing.JSplitPane</code>	<code>BasicSplitPaneUI</code>
<code>javax.swing.JLayeredPane</code>	No tiene delegado
<code>javax.swing.JDesktopPane</code>	<code>BasicDesktopPaneUI</code>
<code>javax.swing.JPanel</code>	<code>BasicPanelUI</code>
<code>javax.swing.JTable</code>	<code>BasicTableHeaderUI</code> , <code>BasicTableUI</code>

<code>javax.swing.JTree</code>	<code>BasicTreeUI</code>
<code>javax.swing.text.JTextComponent</code>	<code>BasicTextUI</code>
<code>javax.swing.JEditorPane</code>	<code>BasicEditorPaneUI</code>
<code>javax.swing.JTextPane</code>	<code>BasicEditorPaneUI</code>
<code>javax.swing.JTextArea</code>	<code>BasicTextAreaUI</code>
<code>javax.swing.JTextField</code>	<code>BasicTextFieldUI</code>
<code>javax.swing.JFormattedTextField</code>	<code>BasicFormattedTextFieldUI</code>
<code>javax.swing.JPasswordField</code>	<code>BasicPasswordFieldUI</code>
<code>javax.swing.JInternalFrame</code>	<code>InternalFrameUI</code>
<code>java.awt.Panel</code>	No tiene delegado
<code>java.awt.Window</code>	No tiene delegado
<code>java.awt.Dialog</code>	No tiene delegado
<code>javax.swing.JDialog</code>	No tiene delegado
<code>java.awt.Frame</code>	No tiene delegado
<code>javax.swing.JFrame</code>	No tiene delegado
<code>javax.swing.JWindow</code>	No tiene delegado

**Tabla 6.2:** Listado de delegados existentes

La mayoría de los componentes tienen un modelo y un delegado como atributo, y los asocian de la forma vista anteriormente. Este apartado se va a centrar en los métodos y las clases relacionadas con los modelos.

Todos los componentes que definen modelos tienen las siguientes características:

- Métodos de acceso en la clase de cada uno de los componentes para el modelo (siendo `XXModel` la interfaz del modelo que use cada clase):
  - o `public XXModel getModel()`
  - o `public void setModel(XXModel modelo)`

En los componentes que tienen el modelo `Document`, estos métodos se sustituyen por `public Document getDocument()` y `public void setDocument(Document d)`.

En aquellos componentes que tienen más de un modelo, los métodos `getModel()` y `setModel(...)`, son usados para acceder al modelo de datos principal. Para los otros modelos existen los métodos siguientes:

- o `public XXModel getXXModel()`
- o `public void setXXModel(XXModel modelo)`

- La mayoría de los componentes tienen un constructor al que se le puede pasar el modelo o los modelos del componente (siendo `JXX` la clase del componente):

```
o public JXX(XXModel modelo, ...)
```

Si se usa un constructor en el que no se especifica ningún modelo, se instancia un modelo por defecto de la clase `DefaultXXModel` (siendo `XXModel` la interfaz del modelo).

### 6.2.3. *Varios componentes representan los mismos datos*

Para poder implementar el hecho de que varios componentes representen los mismos datos, basta con instanciar un componente de una clase determinada con su constructor sin parámetros; internamente, se creará un modelo para albergar los datos de ese componente. El modelo por defecto se obtiene con el método `getModel()`. Por último, se instancia un nuevo componente al que se le pasa como parámetro en su constructor, el modelo del otro componente. A partir de ese momento, los dos componentes tendrán el mismo estado y por tanto la interfaz gráfica será consistente. Por ejemplo, si se quiere que dos componentes `JSlider` tengan el mismo estado, el código sería el siguiente:

```
...
JSlider barral = new JSlider();
BoundedRangeModel modelo = barral.getModel();
JSlider barra2 = new JSlider(modelo);
...
```

Como puede observarse, se puede compartir de esta forma el modelo entre componentes de la misma clase. Además, todos aquellos componentes que sean distintos, pero que tengan la misma interfaz para el modelo, también pueden tener el mismo estado.

### 6.2.4. *Separación de responsabilidades entre el componente y el modelo*

Atendiendo a la división que se está realizando entre datos y visualización, existen dos formas para interactuar con un componente:

- **Gestión de datos**

La inserción de texto en un componente campo de texto es un ejemplo de gestión de datos de ese componente. El determinar si un botón de radio se encuentra pulsado o no, también es un ejemplo de gestión de datos de ese componente, etc.

- **Gestión de visualización**

El hecho de establecer el color de fondo de un componente o su tamaño preferido, por ejemplo, son formas de gestión de visualización del componente.

Cuando se maneja un componente de la interfaz gráfica de usuario, realmente existen tres objetos: el modelo, el delegado y el componente. Los métodos de cada objeto permiten lo siguiente:

- En las interfaces de los modelos, `XXModel`, tan solo existen métodos para gestionar los datos, no existen métodos para determinar aspectos visuales.
- Con respecto a las clases de los delegados, `BasicXXUI`, sería razonable pensar que en ellas se encuentran los métodos para la gestión de la visualización de los componentes. Debido a la implementación, hay motivos que impiden que esto sea así y los detalles que ayudan a comprender esto no se van a tratar en este libro, ya que su desconocimiento no afecta al objetivo del mismo; entre otras cosas porque un delegado nunca es usado por un desarrollador para crear la mayoría de las aplicaciones con interfaz gráfica de usuario.
- Por último, de la parte del componente, existen métodos en las clases `JXX` para que un desarrollador configure los aspectos visuales del componente, como por ejemplo: el color de fondo, el tamaño, el tipo de borde, etc. Además, también estas clases suelen tener algunos métodos para gestionar los datos más utilizados y estos métodos, internamente, llaman al modelo. Se usan en componentes sencillos. Por ejemplo, en un `JTextField` se puede acceder al texto con el método `getText()` sin necesidad de acceder al modelo.

Como puede observarse, cuando se esté usando un componente a nivel de código, se invocarán métodos en el propio componente y también en su modelo. Un esquema básico sería el siguiente:

```
// Construimos el componente
String[] valores = new String[]{
    "Casa", "Coche", "Perro", "Ordenador" };
JList lista = new JList(valores);
...

// Configuramos el aspecto gráfico en el componente
lista.setSelectedIndex(3);
lista.setBorder(BorderFactory.createLineBorder(Color.RED));
...

// Gestionamos los datos desde el componente
Object valorSeleccionado = lista.getSelectedValue();
...

// Gestionamos los datos desde el modelo
ListModel modelo = lista.getModel();
int tamaño = modelo.getSize();
Object elemento = modelo.elementAt(3);
...
```

La interfaz `ListModel`, la cual define los métodos de acceso al modelo de un componente, tiene los métodos estrictamente necesarios para el correcto funcionamiento del mismo. En el caso del componente `JList` visto en el esquema anterior, los métodos existentes en el `ListModel` son sólo de lectura de los datos, y no permiten la modificación de los mismos.

Cuando se construye un componente y no se especifica el modelo, el propio componente construye uno de la clase `DefaultListModel`. Además de los métodos de la interfaz `ListModel`, la clase `DefaultListModel` incorpora muchos más métodos de acceso a los datos. En el caso del componente `JList`, el `DefaultListModel` incluye métodos que permiten la modificación de los datos.

Cuando no se especifica un modelo propio en un componente, entonces se puede estar seguro de que el modelo será de la clase `DefaultListModel` y, por tanto, se puede hacer *casting* a esa clase para poder manejar el modelo con más métodos. Un esquema de esta idea sería el siguiente:

```
...
// Gestionamos los datos desde el modelo por defecto
DefaultListModel modeloDefecto =
(DefaultListModel)lista.getModel();
modeloDefecto.addElement("Bicicleta");
modeloDefecto.remove(4);
...
```

### 6.2.5. *Eventos en el modelo*

Los cambios en los datos de un componente que suceden en respuesta a una acción del usuario no generan eventos en el propio componente. Esto es debido a que estos eventos se generan en el modelo. Por tanto, cuando se esté trabajando con un componente, se tendrá que ver cuáles son los eventos generados en el propio componente y cuáles los generados en el modelo o modelos que usa el componente.

Existen algunos tipos de eventos que se generan tanto en el componente como en el modelo. Son totalmente equivalentes, la única diferencia será que el objeto fuente del evento será distinto. Se puede asociar código de gestión de eventos a cualquiera de ellos con el mismo resultado. Por ejemplo, para poder ejecutar código cuando se cambia el texto de un componente campo de texto, hay que asociar código al evento de cambio que se genera en el modelo. Se seguiría la estructura siguiente:

```
...
JTextField campoTexto = new JTextField();
Document d = campoTexto.getDocument();
d.addDocumentListener(new DocumentListener {

    public void insertUpdate(DocumentEvent e) {
        //Se han insertado caracteres en el texto
    }
    public void removeUpdate(DocumentEvent e) {
        //Se han borrado caracteres del texto
    }
    public void changedUpdate(DocumentEvent e) {
        //Ha cambiado algo en el texto
    }
});
...
```



**Ejemplo 6.1 – Aplicación de traducción 11.0**

Para mostrar cómo usar los eventos en el modelo se va a incorporar una nueva característica a la aplicación de traducción 10.0. En la nueva versión la traducción se realizará en tiempo real, a medida que el usuario edite el campo de texto. Puesto que la traducción se realiza automáticamente, se eliminará el botón de traducir.

La única clase que cambia en esta nueva versión es la clase `PanelTextos`, la cual se muestra a continuación:

**libro/ejemplos/ejemplo6.1/PanelTextos.java**

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

import javax.swing.*;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

class PanelTextos extends JPanel {

    private Traductor traductor;

    private JLabel etiquetaOrigen;
    private JLabel etiquetaDestino;
    private JTextArea areaOrigen;
    private JTextArea areaDestino;

    public PanelTextos(Traductor traductor) {
        this.traductor = traductor;
        this.etiquetaOrigen = new JLabel();
        this.etiquetaDestino = new JLabel();
        this.muestraIdiomas();

        this.areaOrigen = new JTextArea(
            "Escriba el texto aqui", 2, 20);

        this.areaOrigen.getDocument().
            addDocumentListener(new DocumentListener() {
                public void insertUpdate(DocumentEvent e) {
                    traducir();
                }
                public void removeUpdate(DocumentEvent e) {
                    traducir();
                }
                public void changedUpdate(DocumentEvent e) {
```

```
        traducir();
    }
});

this.areaDestino = new JTextArea(
    "Aquí aparecerá la traducción", 2, 20);
this.areaDestino.setEditable(false);

this.setBorder(
    BorderFactory.createTitledBorder(" Traducción "));

JScrollPane spAreaOrigen =
    new JScrollPane(areaOrigen);
JScrollPane spAreaDestino =
    new JScrollPane(areaDestino);

this.setLayout(new GridBagLayout());

GridBagConstraints config1 = new GridBagConstraints();
config1.insets = new Insets(3, 3, 3, 3);
config1.anchor = GridBagConstraints.WEST;
this.add(etiquetaOrigen, config1);

GridBagConstraints config2 = new GridBagConstraints();
config2.insets = new Insets(3, 3, 3, 3);
config2.anchor = GridBagConstraints.WEST;
config2.gridy = 2;
this.add(etiquetaDestino, config2);

GridBagConstraints config4 = new GridBagConstraints();
config4.insets = new Insets(3, 3, 3, 3);
config4.weighty = 1.0;
config4.weightx = 1.0;
config4.fill = GridBagConstraints.BOTH;
config4.gridy = 1;
this.add(spAreaOrigen, config4);

GridBagConstraints config5 = new GridBagConstraints();
config5.insets = new Insets(3, 3, 3, 3);
config5.weighty = 1.0;
config5.weightx = 1.0;
config5.fill = GridBagConstraints.BOTH;
config5.gridy = 3;
this.add(spAreaDestino, config5);
}

public void traducir() {
    areaDestino.setText(
        traductor.traduceTexto(areaOrigen.getText()));
}
```

```

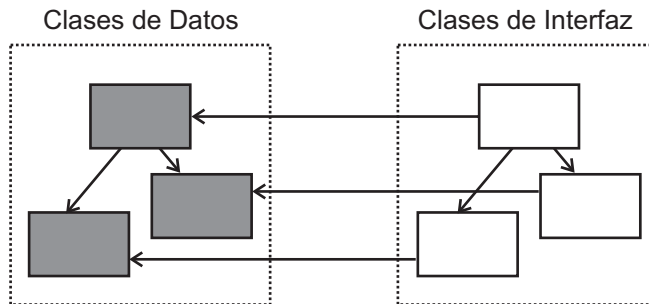
public void muestraIdiomas() {
    this.etiquetaOrigen.setText(" Idioma "
        + Traductor.getCadenaIdioma(
            traductor.getIdiomaOrigen()));

    this.etiquetaDestino.setText(" Idioma "
        + Traductor.getCadenaIdioma(
            traductor.getIdiomaDestino()));
}
}

```

### 6.3. Construcción de una aplicación con interfaz gráfica de usuario

La idea general de MVC es la separación de los datos y de la interfaz gráfica en dos entidades distintas con el mínimo acoplamiento entre ellas. El desarrollo del código de gestión de datos no debe estar influenciado por la forma en que estos datos se van a visualizar y manejar en la interfaz gráfica de usuario. Como consecuencia de esta situación, el código de gestión de datos no debe hacer ninguna referencia al código de la interfaz gráfica. La estructura general se puede ver en la figura 6.5.



**Figura 6.5:** Separación de datos e interfaz (MVC)

Con clases de datos se hace referencia a todas aquellas clases que representan los datos de la aplicación y la gestión que se hace de ellos, sin ninguna consideración sobre la interfaz gráfica. Las clases de interfaz serán aquellas que configuran aspectos como botones, ventanas, colores, etc.

Como puede observarse en la figura, las clases de interfaz se asocian a las clases de datos, tanto para poder representarlas gráficamente cuando sea necesario, como para cambiar su estado en respuesta a las interaccio-

nes con la interfaz gráfica por parte del usuario. Las clases de interfaz se construyen basadas en las clases de datos.

Este enfoque de separación presenta una serie de ventajas:

- **División del trabajo**

Al desarrollar una aplicación, la separación entre datos e interfaz permite dividir el trabajo entre distintos desarrolladores. Los desarrolladores más experimentados en la construcción de interfaces gráficas podrían construir la interfaz y los desarrolladores que conocen más a fondo el funcionamiento interno de la aplicación construirían las clases de datos.

En los ejemplos que se han ido viendo a lo largo de los diferentes capítulos del libro, se ha usado la clase `Traductor` como clase de datos, y la clase `PanelTraduccion` y los diversos gestores de eventos, como clases de interfaz. La clase `Traductor` ya estaba implementada y se ha construido una interfaz gráfica de forma autónoma al desarrollador de la clase de datos, simplemente haciendo uso de los métodos de la clase.

- **Reducción de errores**

Al ser las clases de gestión de datos independientes de la interfaz, se pueden construir y probar sin necesidad de que la interfaz esté construida. Esto permite realizar pruebas altamente automatizadas y de forma más rápida, las cuales no serían posibles si hubiese un alto acoplamiento entre interfaz y datos. Además, es más fácil la localización de errores cuando se separan datos de interfaz.

- **Construcción evolutiva de la interfaz**

La construcción de interfaces gráficas de usuario es una tarea muy costosa. Separando los datos de la interfaz se potencia el hecho de poder construir aplicaciones con una interfaz sencilla de forma rápida (que permita ir validando el resto de código de la aplicación) y, de forma paralela, poder ir construyendo otras clases de interfaz que pueden llevar más tiempo. El código de datos será reutilizado por todas las visualizaciones que se construyan. Un código de datos, que funcione con una interfaz sencilla, seguirá funcionando con una interfaz compleja; es decir, la localización de errores se hace más sencilla.

- **Posibilidad de elegir entre varias interfaces**

Al ser la construcción de la interfaz independiente de los datos, es posible que se construyan varias interfaces diferentes y, en cada parte de la aplicación, se utilice cada una de ellas.

- **Diferentes tecnologías de interfaces usan las mismas clases de datos**

Si se afronta el desarrollo de aplicaciones separando los datos y la gestión de los mismos, de la forma en que éstos se van a representar, se está propiciando que se puedan reutilizar las mismas clases de datos en aplicaciones aparentemente muy diferentes. Por ejemplo, si se construye la misma aplicación con interfaz gráfica en Swing y vía Web, al separar los datos, se podrían reutilizar esas clases en las dos versiones de la aplicación.

En la mayoría de las aplicaciones, se requieren algunas funcionalidades adicionales que no se consiguen únicamente separando los datos de la interfaz de usuario. Estas funcionalidades se enumeran a continuación:

- **Dos partes de la interfaz mostrando los mismos datos**

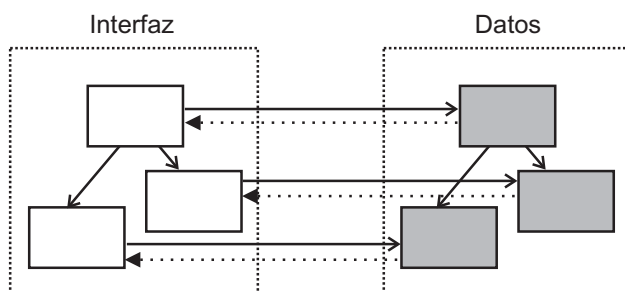
Es habitual que en cualquier aplicación aparezcan en varios lugares de la interfaz gráfica los mismos datos. Por ejemplo, al seleccionar una fila en una tabla, se muestra en otra parte una descripción detallada de esa fila. Y, frecuentemente, en cada una de estas partes se pueden modificar esos datos y es deseable que los datos sean consistentes en las dos visualizaciones.

- **Modificación de datos sin intervención del usuario en la interfaz**

Los datos que se están representando no siempre cambian por eventos generados en la interfaz, de hecho es habitual que no sea así. Por ejemplo, los datos llegan de la línea de comunicaciones, o mediante un temporizador, o bien mediante los resultados de un algoritmo. Cada vez que los datos cambian, hay que actualizar la interfaz para reflejar dichos cambios.

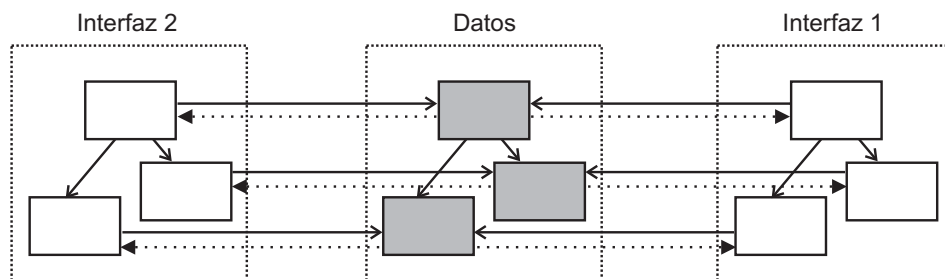
La nueva funcionalidad, como podrá intuir el lector, no se obtiene simplemente separando los datos de la interfaz como ya se ha mostrado. Para permitir que la interfaz gráfica de usuario cambie cuando los datos cambian; bien porque se han modificado en otro punto de la interfaz o

bien de cualquier otro modo, se necesita que las clases de datos generen eventos cada vez que su estado cambia. El esquema básico se presenta en la figura 6.6.



**Figura 6.6:** Generación de eventos de las clases de datos cuando su estado cambia

Y para tener dos partes de la interfaz en las que se visualice la misma información se presenta el esquema de la figura 6.7.



**Figura 6.7:** Misma información en dos visualizaciones diferentes

Las clases que se construyan deberán elevar eventos al cambiar de estado si se pretende tener dos visualizaciones consistentes, o bien, si se quiere que ante un cambio en los datos que no provenga de la interfaz gráfica, la visualización cambie.

### **Ejemplo 6.2 – Aplicación de traducción 12.0**

Para mostrar una aplicación que haga que los datos eleven eventos ante el cambio, se va a ampliar la aplicación de traducción 11.0. El título

de la ventana, los paneles de texto y los botones de radio mostrarán, tanto el idioma origen como el idioma destino de la traducción. La sincronización se realizará gestionando eventos en el traductor, no como hasta ahora, que era necesario actualizar explícitamente las partes de la interfaz gráfica cuando se cambiaba el idioma. Ahora todos los componentes que muestren los idiomas origen o destino de la traducción, deberán escuchar los eventos del traductor y cuando cambie el sentido, actualizar la interfaz.

Las clases que han cambiado con respecto a la versión anterior son `VentanaTraductor`, `PanelTraductor` y `PanelSeleccionIdioma`.

**libro/ejemplos/ejemplo6.2/VentanaTraductor.java**

```
import javax.swing.JFrame;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class VentanaTraductor extends JFrame {

    private Traductor traductor;

    public VentanaTraductor(Traductor traductor) {
        this.traductor = traductor;
        this.setContentPane(new PanelTraductor(traductor));

        traductor.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                muestraIdiomas();
            }
        });

        this.muestraIdiomas();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    protected void muestraIdiomas() {
        this.setTitle("Traductor de "
            + Traductor.getCadenaIdioma(traductor
                .getIdiomaOrigen())
            + " a "
            + Traductor.getCadenaIdioma(traductor
                .getIdiomaDestino()));
    }
}
```

**libro/ejemplos/ejemplo6.2/PanelTraductor.java**

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JPanel;

public class PanelTraductor extends JPanel {

    private PanelTextos panelTextos;
    private PanelSeleccionIdioma panelIdiomaOrigen;
    private PanelSeleccionIdioma panelIdiomaDestino;

    public PanelTraductor(Traductor traductor) {

        panelTextos = new PanelTextos(traductor);
        panelIdiomaOrigen = new PanelSeleccionIdioma(
            PanelSeleccionIdioma.IDIOMA_ORIGEN,
            traductor);
        panelIdiomaDestino = new PanelSeleccionIdioma(
            PanelSeleccionIdioma.IDIOMA_DESTINO,
            traductor);

        this.setLayout(new GridBagLayout());

        GridBagConstraints config1 = new GridBagConstraints();
        config1.insets = new Insets(3, 3, 3, 3);
        config1.weighty = 1.0;
        config1.weightx = 1.0;
        config1.fill = GridBagConstraints.BOTH;
        config1.gridheight = 2;
        this.add(panelTextos, config1);

        GridBagConstraints config2 = new GridBagConstraints();
        config2.insets = new Insets(3, 3, 3, 3);
        config2.gridx = 1;
        config2.weighty = 1.0;
        config2.fill = GridBagConstraints.BOTH;
        this.add(panelIdiomaOrigen, config2);

        GridBagConstraints config3 = new GridBagConstraints();
        config3.insets = new Insets(3, 3, 3, 3);
        config3.gridx = 1;
        config3.gridy = 1;
        config3.weighty = 1.0;
        config3.fill = GridBagConstraints.BOTH;
        this.add(panelIdiomaDestino, config3);
    }
}
```



**libro/ejemplos/ejemplo6.2/PanelSeleccionIdioma.java**

```
import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.JPanel;

public class PanelSeleccionIdioma extends JPanel {

    public static final int IDIOMA_ORIGEN = 0;
    public static final int IDIOMA_DESTINO = 1;

    private Traductor traductor;

    private int tipoIdioma;
    private PanelRadioButtons panelRadioButtons;

    public PanelSeleccionIdioma(int tipoIdioma, Traductor traductor) {

        this.tipoIdioma = tipoIdioma;
        this.traductor = traductor;
        String cadenaTipo;

        if (tipoIdioma == IDIOMA_ORIGEN) {
            cadenaTipo = " Idioma Origen ";
        } else {
            cadenaTipo = " Idioma Destino ";
        }

        this.setBorder(
            BorderFactory.createTitledBorder(cadenaTipo));
        this.setMinimumSize(new Dimension(110, 110));
        this.setPreferredSize(new Dimension(110, 110));

        panelRadioButtons = new
            PanelRadioButtons(Traductor.NUM_IDIOMAS, 1);

        this.add(panelRadioButtons);

        for (int i = 0; i < Traductor.NUM_IDIOMAS; i++) {

            boolean seleccionado =
                (tipoIdioma == IDIOMA_ORIGEN &&
                 i == traductor
                    .getIdiomaOrigen())
                || (tipoIdioma == IDIOMA_DESTINO
                    && i == traductor
                        .getIdiomaDestino());

            panelRadioButtons.addRadioButton(
                Traductor.getCadenaIdioma(i),
```

```
seleccionado);

panelRadioButtons.addRadioListener(
    new RadioListener() {
        public void radioSelectionChanged(RadioEvent e) {
            botonRadioSeleccioado(e.getNumOpcion());
        }
    });
}

public void botonRadioSeleccioado(int numBoton) {
    if (this.tipoIdioma == IDIOMA_ORIGEN) {
        traductor.setIdiomaOrigen(numBoton);
    } else if (this.tipoIdioma == IDIOMA_DESTINO) {
        traductor.setIdiomaDestino(numBoton);
    } else {
        throw new Error();
    }
}
}
```

## **Capítulo 7**

### **Técnicas para evitar el bloqueo de la interfaz gráfica**

En las aplicaciones con interfaz gráfica de usuario el código encargado de la gestión de eventos se ejecuta de forma secuencial. Durante la ejecución de este código la interfaz gráfica no responde, por lo que es recomendable que este código se ejecute de forma rápida. En este capítulo se estudian las técnicas que proporciona Swing para iniciar la ejecución de tareas que tardan un tiempo sustancial en ejecutarse (más de 250 milisegundos) y no bloquear la interfaz gráfica de usuario.

#### **7.1. Introducción**

Como ya se estudió en el primer capítulo, la ejecución de una aplicación con interfaz gráfica de usuario consiste, principalmente, en la ejecución de código de gestión de eventos. El usuario interactúa con la aplicación a través de la interfaz gráfica y diferentes acciones sobre los componentes de la interfaz generan uno o varios eventos. Cuando se producen eventos, si hay código de gestión de eventos, se ejecuta de forma secuencial, uno detrás de otro. Durante esta ejecución la interfaz gráfica no responde, no se actualiza, tampoco se atienden las entradas del usuario; entonces, es imprescindible que dicho código de gestión se ejecute lo más rápidamente posible. No obstante, hay procesos o tareas que necesitan un tiempo sustancial en ejecutarse, por ejemplo, descargar una imagen desde Internet, realizar un cálculo complejo, ejecutar una sentencia en una base de datos, etc. Por ello, se va a estudiar cómo

ejecutar tareas que tarden en ejecutarse un tiempo perceptible por el usuario sin que la interfaz gráfica de usuario se bloquee.

El resto del capítulo se encuentra estructurado en diferentes apartados. El primero de ellos describe el hilo de ejecución de los eventos en Swing. En el segundo apartado se presenta el modo de crear un nuevo hilo de ejecución para tareas en segundo plano. Un tercer y último apartado muestra cómo actualizar la interfaz gráfica de usuario cuando se ejecutan tareas en segundo plano.

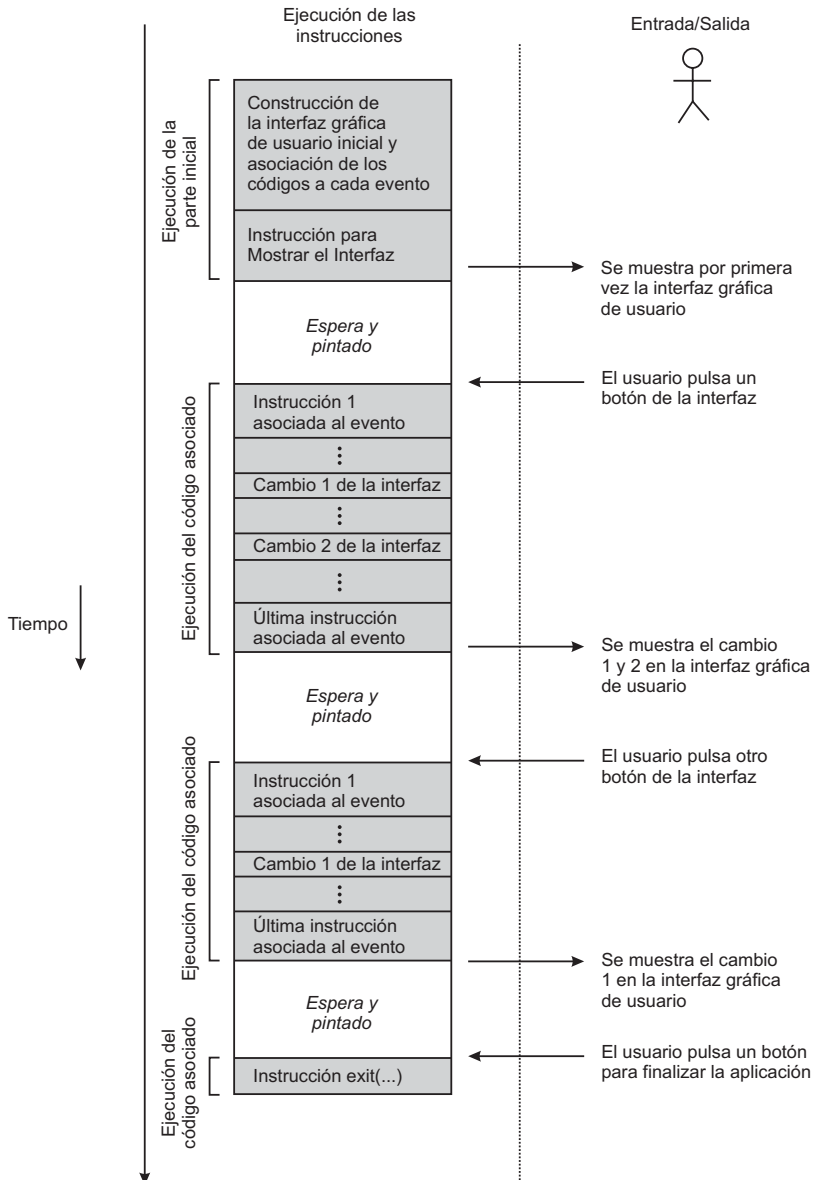
## 7.2. Hilo de despacho de eventos de Swing

Cuando no se está ejecutando ningún código de gestión de eventos, Swing se encarga de actualizar la interfaz gráfica y atender las entradas del usuario. La figura 1.2 del primer capítulo muestra, de forma esquemática, la ejecución de una aplicación con interfaz gráfica. En este punto se presenta de nuevo dicha figura (figura 7.1) para ver la secuencia de ejecución de una aplicación con interfaz gráfica de usuario.

Tanto los códigos de gestión de eventos como los procesos de pintado y control de las entradas del usuario se ejecutan en un único hilo de ejecución o thread. Este thread es muy importante y es denominado **hilo de despacho de eventos** o **event dispatch thread**. Por lo tanto, todo el código de gestión de eventos será ejecutado en este hilo. La clase `javax.swing.SwingUtilities` dispone del método `public static boolean isEventDispatchThread()`. Este método devuelve `true` si es invocado por un código que se está ejecutando desde el hilo de despacho de eventos, o `false` en caso contrario.

Es muy importante conocer que la **interfaz gráfica de usuario solamente puede ser modificada desde el hilo de despacho de eventos**. Si se modifica la interfaz gráfica desde otro hilo de ejecución los resultados son imprevisibles, por ejemplo, podrían aparecer, entre otras cosas, componentes parcialmente dibujados.

## Aplicación con interfaz gráfica de usuario



**Figura 7.1:** Ejecución de una aplicación con interfaz gráfica de usuario

### 7.3. Ejecución de tareas en segundo plano (background)

Puesto que todo el código de manejo de eventos se ejecuta en el hilo de despacho de eventos, ha de ejecutarse rápidamente para no bloquear la interfaz gráfica. Cuando sea necesario realizar tareas que tardan un tiempo sustancial en ejecutarse, éstas deberán ejecutarse en un nuevo hilo de ejecución.

En Java la creación de un nuevo hilo de ejecución es sencilla. Crear e iniciar un nuevo hilo, utilizando clases anónimas, se puede hacer siguiendo el siguiente esquema básico:

```
...
public void codigoGestionEventos() {

    //Código ejecutado en el hilo de despacho de eventos

    new Thread(){
        public void run() {
            // Código ejecutado en un nuevo hilo
        }
    }.start();

    //Código ejecutado en el hilo de despacho de eventos
}
...
```

Desde el punto de vista de la interfaz gráfica, se puede decir que el código que se ejecuta en este nuevo hilo está siendo ejecutado en **segundo plano** o **background**. Este código no bloquea la interfaz y puede estar ejecutándose todo el tiempo que sea necesario. Sin embargo, la creación de nuevos hilos de ejecución es una técnica relativamente costosa, por lo que no hay que abusar creando cientos de hilos de ejecución a la vez.

### 7.4. Actualización de la interfaz gráfica desde una tarea en segundo plano

Resumiendo, si se necesita realizar una tarea que tarda mucho tiempo en ejecutarse, se crea un nuevo hilo de ejecución. Con frecuencia, es interesante actualizar la interfaz gráfica de usuario con el progreso de la

tarea o bien con el resultado de la misma. Por ejemplo, si una tarea se encarga de descargar un fichero de Internet, es habitual que una barra de progreso muestre precisamente eso, el progreso de descarga.

Como ya se ha dicho antes, la **interfaz gráfica de usuario solamente puede ser modificada desde el hilo de despacho de eventos**. Por tanto, se necesita algún mecanismo que permita ejecutar código en el hilo de despacho de eventos a medida que se ejecuta la tarea de segundo plano o, al final, cuando se obtenga el resultado.

Si se quiere modificar la interfaz gráfica de usuario desde una tarea en segundo plano, se pueden usar los siguientes métodos de la clase `javax.swing.SwingUtilities`:

- `public static void invokeLater(Runnable doRun)`
- `public static void invokeAndWait(Runnable doRun)`  
`throws InterruptedException,`  
`InvocationTargetException`

Utilizando clases anónimas, se puede usar el método `invokeLater(...)` con el siguiente esquema:

```
...
public void tareaSegundoPlano(){

//Código ejecutado en segundo plano.
//No se puede modificar la interfaz.

SwingUtilities.invokeLater(new Runnable(){
    public void run() {
        //Código ejecutado en el
        //hilo de despacho de eventos.
        //Se puede modificar la interfaz
    }
});

//Código ejecutado en segundo plano
//No se puede modificar la interfaz.
}
...
```

El método `invokeAndWait(...)` tiene un esquema algo diferente, ya que es necesario capturar las excepciones:

```
public void tareaSegundoPlano(){

//Código ejecutado en segundo plano.
//No se puede modificar la interfaz.

try {

    SwingUtilities.invokeLater(new Runnable(){
        public void run() {
            //Código ejecutado en el
            //hilo de despacho de eventos.
            //Se puede modificar la interfaz
        }
    });

} catch (InterruptedException e) {
} catch (InvocationTargetException e) {}

//Código ejecutado en segundo plano
//No se puede modificar la interfaz.
}
```

Cuando se usa el método `invokeLater(...)` lo que se hace es indicar al hilo de eventos que tiene que ejecutar el código pasado. La tarea en segundo plano que utiliza el método `invokeLater(...)` no se queda bloqueada esperando a que dicho código sea ejecutado, sino que continua la ejecución. En cambio, cuando se utiliza el método `invokeAndWait(...)`, la tarea de segundo plano se queda bloqueada hasta que el código que ha de ejecutarse en el hilo de despacho de eventos sea ejecutado.

Dependiendo del proceso que esté realizando la tarea de segundo plano se usará un método u otro. Por ejemplo, si se quiere mostrar el progreso de descarga de un fichero, basta con usar el método `invokeLater(...)`, porque no es necesario esperar a que se haya actualizado la interfaz gráfica para continuar con el proceso de descarga.

### **Ejemplo 7.1 – Ejemplo de tarea larga 1.0**

Este ejemplo se crea para mostrar los efectos del bloqueo que se produce en la interfaz de usuario cuando el código de gestión de eventos tarda mucho tiempo en ejecutarse. La tarea que tarda mucho tiempo en ejecutarse se simula aquí llamando al método `static void sleep(long`



milis) de la clase `java.lang.Thread`, que detiene la ejecución durante el tiempo indicado.

La aplicación consta de un botón y una etiqueta. El botón inicia la tarea y en la etiqueta se muestra el momento en que comienza y finaliza su ejecución. El código de la aplicación se muestra a continuación:

**libro/ejemplos/ejemplo7.1/EjemploTareaLarga1.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

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

public class EjemploTareaLarga1 {

    JLabel etiqueta = new JLabel("No ha comenzado la tarea");

    public static void main(String[] args) {
        new EjemploTareaLarga1();
    }

    public EjemploTareaLarga1() {

        JFrame ventana = new JFrame("Ejemplo Tarea Larga 1");
        ventana.setSize(300, 200);
        ventana.
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton boton = new JButton("Botón");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                botonPulsado();
            }
        });

        JPanel panelDeContenido = new JPanel();
        ventana.setContentPane(panelDeContenido);

        panelDeContenido.add(boton);
        panelDeContenido.add(etiqueta);
        ventana.setVisible(true);
    }

    protected void botonPulsado() {
        etiqueta.setText("Ha comenzado la tarea");
    }
}
```

```
        ejecutaTareaLarga();
        etiqueta.setText("Ha finalizado la tarea");
    }

    private void ejecutaTareaLarga() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {}
    }
}
```

Al ejecutar la aplicación pueden observarse dos efectos no deseados. El primero de ellos es que durante la ejecución de la tarea la aplicación no se redibuja correctamente. Por ejemplo, al pasar el ratón sobre el botón, no se comporta como lo hace normalmente. Del mismo modo, si ponemos otra ventana encima de la aplicación y la retiramos, no se redibuja el interfaz hasta que no haya finalizado la tarea. El otro efecto no deseado, es que la etiqueta no muestra "Ha comenzado la tarea". Esto es debido a que se redibuja el interfaz cuando acaba la tarea, y en ese momento la etiqueta ya tiene el valor "Ha finalizado la tarea".

### **Ejemplo 7.2 – Ejemplo de tarea larga 2.0**

En este ejemplo se van a solventar los efectos no deseados que aparecen en el ejemplo anterior. Para ello, se va a crear un hilo de ejecución para ejecutar la tarea larga. Además, cuando haya finalizado dicha tarea, se actualizará la interfaz empleando el método `invokeLater(...)`. El código de la aplicación se muestra a continuación:

**libro/ejemplos/ejemplo7.2/EjemploTareaLarga2.java**

```
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class EjemploTareaLarga2 {

    JLabel etiqueta = new JLabel("No ha comenzado la tarea");

    public static void main(String[] args) {
        new EjemploTareaLarga2();
    }
}
```

```

public EjemploTareaLarga2() {

    JFrame ventana = new JFrame("Ejemplo Tarea Larga 1");
    ventana.setSize(300, 200);
    ventana.
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton boton = new JButton("Botón");
    boton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            botonPulsado();
        }
    });

    JPanel panelDeContenido = new JPanel();
    ventana.setContentPane(panelDeContenido);

    panelDeContenido.add(boton);
    panelDeContenido.add(etiqueta);

    ventana.setVisible(true);
}

protected void botonPulsado() {
    etiqueta.setText("Ha comenzado la tarea");

    new Thread() {
        public void run() {
            ejecutaTareaLarga();
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    etiqueta.setText(
                        "Ha finalizado la tarea");
                }
            });
        }
    }.start();
}

private void ejecutaTareaLarga() {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {}
}
}

```

En el código se puede observar como la llamada al método `ejecutaTareaLarga()` se realiza desde el nuevo hilo y por tanto en segundo plano.

Cuando finaliza su ejecución, la forma de modificar la interfaz para notificarlo es usando el método `invokeLater(...)`.

### **Ejemplo 7.3 – Visor de Imágenes 2.0**

En el capítulo 4, en el ejemplo 4.1, se implementó un Visor de Imágenes donde se cargaba una imagen desde una URL. Si la carga de la imagen era muy lenta, la interfaz se bloqueaba. Por ello, se propone crear una nueva versión del Visor de Imágenes, haciendo que la imagen se cargue en segundo plano.

En el código que se presenta a continuación las sentencias que se ejecutan en segundo plano se muestran en **negrita**.

#### **libro/ejemplos/ejemplo7.3/VisorImagenes.java**

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import javax.imageio.ImageIO;
import javax.swing.*;

public class VisorImagenes extends JPanel {

    private JLabel etqImagen;

    private JLabel etqURLImagen;
    private JTextField campoURLImagen;
    private JButton botonCarga;

    public VisorImagenes() {

        this.etqURLImagen = new JLabel("URL imagen: ");
        this.campoURLImagen = new JTextField(50);
        this.campoURLImagen.setText("http://www.escet.urjc.es/"+
            "~/smontalvo/interfacesgraficasenjava/imagen3.jpg");

        this.botonCarga = new JButton("Cargar imagen");
        this.botonCarga.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                iniciaCargaImagen();
            }
        });
    }
}
```

```
this.setLayout(new BorderLayout());

JPanel panelControles = new JPanel();
panelControles.setLayout(new BorderLayout(10,10));
panelControles.add(this.etqURLImagen, BorderLayout.WEST);
panelControles.add(this.campoURLImagen,
                    BorderLayout.CENTER);
panelControles.add(this.botonCarga, BorderLayout.EAST);

panelControles.setBorder(BorderFactory
    .createEmptyBorder(10, 10, 10, 10));

this.add(panelControles, BorderLayout.NORTH);

this.etqImagen = new JLabel(
    "<HTML><div align='center'>"+
    "No hay imagen cargada<BR><BR>"+
    "Seleccione una URL y pulsa cargar</div>");

this.etqImagen.
    setHorizontalAlignment(SwingConstants.CENTER);

JScrollPane sp = new JScrollPane(this.etqImagen);

this.add(sp, BorderLayout.CENTER);
}

protected void iniciaCargaImagen() {

    final URL url;
    try {
        url = new URL(this.campoURLImagen.getText());
    } catch (MalformedURLException e) {
        JOptionPane.showMessageDialog(this,
            "URL inválida.", "Error",
            JOptionPane.ERROR_MESSAGE);

        return;
    }

    this.botonCarga.setEnabled(false);
    this.etqImagen.setText("Cargando imagen");

    new Thread() {
        public void run() {
            cargaImagen(url);
        }
    }.start();
}
```

```
protected void cargaImagen(URL urlImagen) {

    final Icon icono;

    try {
        icono = new ImageIcon(ImageIO.read(urlImagen));
    } catch (IOException e) {
        errorCargaImagen();
        return;
    }

    SwingUtilities.invokeLater(new Runnable(){
        public void run() {
            etqImagen.setText("");
            etqImagen.setIcon(icono);

            botonCarga.setEnabled(true);
        }
    });
}

private void errorCargaImagen() {

    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(
                VisorImagenes.this,
                "Error al cargar la imagen.", "Error",
                JOptionPane.ERROR_MESSAGE);
            botonCarga.setEnabled(true);
        }
    });
}
}
```

Se puede observar que el proceso de carga se realiza en segundo plano. Cuando éste ha finalizado, desde el hilo de despacho de eventos se muestra la imagen en la interfaz gráfica de usuario. En caso de que ocurra un error durante la carga, también desde el hilo de despacho de eventos, se muestra un panel de opciones informando de ello.

#### **Ejemplo 7.4 – Visor de Imágenes 3.0**

En este ejemplo se va a ampliar el visor de imágenes 2.0 para que muestre el proceso de carga de la imagen en la interfaz gráfica de

usuario. Para ello, solo hay que recordar que toda modificación de la interfaz ha de hacerse desde el hilo de despacho de eventos.

Nuevamente, se destacan en **negrita** en el código aquellas sentencias que se ejecutan en segundo plano.

**libro/ejemplos/ejemplo7.4/VisorImagenes.java**

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.event.IIOReadProgressListener;
import javax.imageio.stream.ImageInputStream;
import javax.swing.*;

public class VisorImagenes extends JPanel {

    private JLabel etqImagen;

    private JLabel etqURLImagen;
    private JTextField campoURLImagen;
    private JButton botonCarga;

    public VisorImagenes() {

        this.etqURLImagen = new JLabel("URL imagen: ");
        this.campoURLImagen = new JTextField(50);
        this.campoURLImagen.setText(
            "http://vido.escet.urjc.es/"+
            "interfacesgraficasenjava/imagen3.jpg");

        this.botonCarga = new JButton("Cargar imagen");
        this.botonCarga.addActionListener(new ActionListener() {
            {
                public void actionPerformed(ActionEvent arg0) {
                    iniciaCargaImagen();
                }
            }
        });

        this.setLayout(new BorderLayout());

        JPanel panelControles = new JPanel();
        panelControles.setLayout(new BorderLayout(10, 10));
        panelControles.add(this.etqURLImagen,
```

```
        BorderLayout.WEST);
panelControles.add(this.campoURLImagen,
                    BorderLayout.CENTER);
panelControles.add(this.botonCarga,
                    BorderLayout.EAST);

panelControles.setBorder(BorderFactory
    .createEmptyBorder(10, 10, 10, 10));

this.add(panelControles, BorderLayout.NORTH);
this.etqImagen = new JLabel(
    "<HTML><div align='center'>"+
    "No hay imagen cargada<BR><BR>"+
    "Seleccione una URL y pulsa cargar</div>");

this.etqImagen.setHorizontalAlignment(
    SwingConstants.CENTER);

JScrollPane sp = new JScrollPane(this.etqImagen);

this.add(sp, BorderLayout.CENTER);
}

protected void iniciaCargaImagen() {

    final URL url;
    try {
        url = new URL(this.campoURLImagen.getText());
    } catch (MalformedURLException e) {
        JOptionPane.showMessageDialog(this,
            "URL inválida.", "Error",
            JOptionPane.ERROR_MESSAGE);
        return;
    }

    this.botonCarga.setEnabled(false);

    this.etqImagen.setIcon(null);
    this.etqImagen.setText("Cargando imagen");

    new Thread() {
        public void run() {
            cargaImagen(url);
        }
    }.start();
}

protected void cargaImagen(URL urlImagen) {

    final Icon icono;
```



```
try {
    icono = monitorizaCargaImagen(urlImagen);

} catch (IOException e) {
    errorCargaImagen();
    return;
}

SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        etqImagen.setText("");
        etqImagen.setIcon(icono);
        botonCarga.setEnabled(true);
    }
});
}

private Icon monitorizaCargaImagen(URL urlImagen)
    throws IOException {

    ImageInputStream iis = ImageIO.createImageInputStream(
        urlImagen.openStream());
    ImageReader ir = (ImageReader)
        ImageIO.getImageReaders(iis).next();

    ir.addIIIOReadProgressListener(
        new IIIOReadProgressListener() {

        public void sequenceStarted(
            ImageReader source, int minIndex) {}
        public void sequenceComplete(ImageReader source) {}

        public void imageStarted(
            ImageReader source, int imageIndex) {
            actualizaProgresoCarga(0f);
        }
        public void imageComplete(ImageReader source) {
            actualizaProgresoCarga(100f);
        }
        public void imageProgress(
            ImageReader source, float percentageDone) {
            actualizaProgresoCarga(percentageDone);
        }

        public void thumbnailStarted(ImageReader source,
            int imageIndex, int thumbnailIndex) {}
        public void thumbnailProgress(ImageReader source,
            float percentageDone) {}
        public void thumbnailComplete(ImageReader source) {}
        public void readAborted(ImageReader source) {}
    });
}
```

```
        ir.setInput(iis);
        return new ImageIcon(ir.read(0));
    }

    protected void actualizaProgresoCarga(
        final float percentageDone){

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                etqImagen.setText("Cargando imagen ("
                    + (int) percentageDone +
                    " % completado)");
            }
        });
    }

    private void errorCargaImagen() {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JOptionPane
                    .showMessageDialog(VisorImagenes.this,
                        "Error al cargar la imagen.", "Error",
                        JOptionPane.ERROR_MESSAGE);
                botonCarga.setEnabled(true);
            }
        });
    }
}
```

## Capítulo 8

### Tipos de aplicaciones con interfaz gráfica de usuario y su distribución

Se pretende, en este último capítulo, dar una visión general de los diferentes tipos de aplicaciones con interfaz gráfica de usuario, donde la principal diferencia entre ellas, aparte de sus características y posibilidades de interfaz gráfica, es la forma en que son distribuidas para que el usuario pueda ejecutarlas.

#### 8.1. Introducción

Se entiende por **distribución de la aplicación** al proceso o a la preparación de todos los recursos software necesarios para que un usuario final pueda ejecutar la aplicación.

En general, existen dos formas de distribuir aplicaciones Java: como varios ficheros `.class` o bien como un único fichero `.jar`. Dependiendo del tipo de aplicación hay que tener en cuenta algunos detalles para usar una forma de distribución u otra.

Como ya se adelantó en el primer capítulo, se pueden construir diferentes tipos de aplicaciones con interfaz gráfica de usuario: aplicaciones autónomas, aplicaciones Java Web Start y Applets. Por ello, cada uno de los tres apartados siguientes del capítulo hace referencia a un tipo de aplicación, mientras que en el último apartado se resumen las diferentes posibilidades de distribución de una aplicación Java.

## 8.2. Aplicación autónoma

Las aplicaciones autónomas son aquellas que tienen las mismas características que cualquier aplicación que se ejecuta en un sistema operativo; en cada sistema operativo se inicia la ejecución de una forma determinada.

Las características fundamentales de una aplicación autónoma escrita en Java son las siguientes:

- Su ejecución no depende de otras aplicaciones auxiliares, simplemente necesita la máquina virtual de Java (*JVM, Java Virtual Machine*) para ejecutarse.
- De forma habitual se encuentra almacenada en el disco duro del usuario.
- Su ejecución se inicia, o bien ejecutando un comando en la línea de comandos (por ejemplo: `java NombreClaseMain`), o bien haciendo clic en algún icono.
- La interfaz gráfica de una aplicación autónoma está formada por una o varias ventanas de aplicación.
- Normalmente, finaliza su ejecución cuando el usuario cierra la ventana o ventanas de aplicación.
- No tiene ninguna restricción de seguridad. Puede acceder al disco duro y establecer conexiones de red.

### Distribución de una aplicación autónoma en ficheros `.class`

Una aplicación autónoma puede distribuirse como un conjunto de ficheros `.class`, obtenidos como resultado de la compilación de los ficheros fuente. Obviamente, para ejecutar esta aplicación se usa el siguiente comando:

```
java nombreClaseMain
```

Si la aplicación autónoma necesita acceder a un fichero asociado a la aplicación (por ejemplo un icono, un texto de ayuda, etcétera), dicho fichero debe ir en la distribución junto con los ficheros `.class`. Para acceder a este fichero, simplemente se abre el directorio donde se

encuentra (habitualmente el directorio de trabajo) y se leen los datos usando el siguiente esquema básico:

```
...
String directorioTrabajo = System.getProperty("user.dir");
File fichero = new File(directorioTrabajo +
    File.separator + "nombreFichero.txt");
InputStream input = new FileInputStream(fichero);

//... leemos los datos del InputStream

input.close();
...
```

También se puede acceder a los ficheros que se encuentran en el CLASSPATH de Java; es decir, en el lugar donde se encuentran los ficheros .class de la forma siguiente:

```
...
InputStream input = this.getClass().
    getResourceAsStream("nombreFichero.txt");

//... leemos los datos del InputStream

input.close();
...
```

De manera habitual, cuando se distribuyen las aplicaciones como un conjunto de ficheros .class y demás ficheros asociados, se utiliza una aplicación de instalación, que se encarga de copiar todos estos ficheros al disco duro del usuario en un lugar adecuado. Existen muchas aplicaciones de instalación, tanto comerciales como de código abierto. Una de código abierto se puede encontrar en la dirección de Internet <http://www.izforge.com/izpack/>.

## **Distribución de una aplicación autónoma empaquetada en un fichero .jar**

Un fichero .jar almacena tanto las clases de la aplicación, como aquellos ficheros necesarios para su ejecución. Para empaquetar todos los ficheros de la aplicación se ejecuta la siguiente sentencia en el directorio donde se encuentren los ficheros .class y los demás ficheros:

```
jar cmvf mainClass.txt aplicacion.jar *
```

donde `mainClass.txt` es un fichero de texto con el nombre de la clase que contiene el método `main`. La estructura de este fichero es como sigue:

**mainClass.txt**

```
Main-Class: paquete.nombreClaseMain.L
```

Es obligatorio poner un salto de línea al final del nombre de la clase que tiene el método `main`. Además, la clase debe ir cualificada con el nombre del paquete.

Para ejecutar la aplicación empaquetada hay que usar el comando `java -jar aplicacion.jar`. En algunas plataformas, por ejemplo Windows, la aplicación se ejecutará al pulsar dos veces sobre el fichero en el explorador. El fichero `.jar` realmente es un fichero `.zip`, salvo que tiene una extensión diferente.

Puesto que el contenido del fichero `.jar` se incluye automáticamente en el `CLASSPATH`, al ejecutar la aplicación, para acceder a los ficheros que se encuentran empaquetados en un fichero `.jar` desde la aplicación Java, es necesario usar el siguiente esquema:

```
...
InputStream input = this.getClass().
    getResourceAsStream("nombreFichero.txt");

//... leemos los datos del InputStream

input.close();
...
```

Por supuesto, la aplicación podrá acceder a cualquier fichero del disco a través de la clase `File`, pero no podrá acceder de esta forma a los ficheros contenidos en el `.jar`.

**Ejemplo 8.1**

Para mostrar cómo se empaqueta una aplicación concreta, se va a empaquetar la última versión de la aplicación de traducción. Para ello,

simplemente hay que ejecutar el comando siguiente en el directorio donde se tengan los ficheros `.class` (hay que recordar que todo el contenido de ese directorio se incluirá en el fichero `.jar`):

```
jar cmvf mainClass.txt AplicacionTraductor.jar *
```

Siendo el fichero `mainClass.txt` el siguiente:

**mainClass.txt**

Main-Class: AplicacionTraductor.J
-----------------------------------

Y se ejecutará usando el comando:

```
java -jar AplicacionTrauctor.jar
```

### 8.3. Java Web Start

Se podría decir que las aplicaciones Java Web Start son una conjunción de las posibilidades de una aplicación autónoma y un Applet. La idea principal es que la aplicación se descarga de un servidor Web como si fuera un Applet. Sin embargo, una vez descargada no tiene que ejecutarse en el navegador Web, sino que puede tener ventanas de aplicación propias.

Hay dos modos de ejecución para este tipo de aplicaciones: con acceso total a disco y red, como las aplicaciones autónomas, o bien con acceso restringido, como los Applets. Si la aplicación necesita acceso total a la red y al disco, se pide al usuario que dé su permiso mediante un cuadro de diálogo.

Las características fundamentales de las aplicaciones Java Web Start son:

- Además de la máquina virtual de Java, se necesita un gestor de aplicaciones asociado al navegador para descargarse de la Web las aplicaciones. Este gestor viene incluido en la distribución del JRE (*Java Runtime Environment*) de *Sun Microsystems*<sup>3</sup>.

---

<sup>3</sup> <http://www.java.com>

- Usualmente, antes de ejecutarse se cargan desde un servidor Web. Una vez descargadas, se almacenan en el disco duro para poder ejecutarse más rápidamente en el futuro y sin necesidad de conexión a la red.
- Se ejecutan, o bien cuando se pulsa un enlace en una página Web, o una vez que han sido descargadas utilizando un gestor de aplicaciones o usando un acceso directo en el escritorio o en los menús de aplicaciones.
- Su interfaz gráfica está formada por una o varias ventanas de aplicación.
- Normalmente finaliza la aplicación cuando el usuario cierra la ventana o ventanas de aplicación.
- Se pueden establecer restricciones de seguridad, dependiendo de la decisión del usuario para cada aplicación individual.

### Seguridad en las aplicaciones Java Web Start

Existen dos tipos de aplicaciones Java Web Start dependiendo de los recursos del sistema a los que quieran acceder:

- **Aplicaciones con todos los permisos** – Tendrán acceso total y sin restricciones a la máquina donde se ejecutan.
- **Aplicaciones con permisos limitados** – Tendrán el acceso a la máquina con restricciones, no podrán acceder directamente al disco duro ni establecer conexiones de red a servidores de Internet distintos al servidor de donde se descarga la aplicación Java Web Start. No obstante, estas aplicaciones pueden usar una librería especial que les permite acceder a ficheros, al sistema de portapapeles, a la impresora, etc., todo ello de forma limitada y, además, el usuario será informado siempre de ello.

En el caso de que el desarrollador construya una aplicación con todos los permisos, el usuario será informado de esta situación al ejecutar la aplicación, para dar su consentimiento si así lo desea o impedir la ejecución de la misma.

Si la aplicación tiene permisos limitados y ejecuta alguna sentencia no permitida por las restricciones de seguridad, se elevará una excepción del tipo `java.lang.SecurityException`. Esta excepción podrá ser capturada



en la propia aplicación para actuar en consecuencia, por ejemplo, informar al usuario de ello.

## La aplicación Java Web Start en la Web. La página HTML y el fichero .jnlp

Para que el usuario pueda ejecutar una aplicación Java Web Start desde una página Web, el desarrollador debe empaquetar la aplicación en un fichero .jar y crear un fichero con extensión .jnlp, el cual contendrá toda la información necesaria para la carga y ejecución de la aplicación. Una vez hecho esto, basta con incluir un enlace a dicho fichero en cualquier documento HTML para que al ser pulsado se cargue la aplicación.

A continuación se muestra un ejemplo de fichero .jnlp.

### nombreDelFichero.jnlp

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
  spec="1.0+"
  codebase="http://www.miweb.com/aplicacion/"
  href="nombreDelFichero.jnlp">
  <information>
    <title>Titulo</title>
    <vendor>Empresa</vendor>
    <description>Aplicacion de prueba</description>
    <description kind="short">Prueba</description>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4"/>
    <jar href="Aplicacion.jar"/>
  </resources>
  <application-desc main-class="NombreClaseMain"/>
</jnlp>
```

Se ha considerado que el ejemplo anterior es lo suficiente representativo para comprender las opciones básicas, para una información más detallada se recomienda consultar la documentación en la dirección <http://java.sun.com/products/javawebstart/>. En el ejemplo se puede ver

que existe una etiqueta `<security>` en la que se si se quiere que la aplicación tenga todos los permisos se especifica `<all-permissions/>`. En caso contrario, se omitirá la etiqueta `<security>`.

Un fichero HTML de ejemplo, con un enlace para ejecutar la aplicación Java Web Start, podría ser el siguiente:

**web.html**

```
<HTML>
<HEAD><TITLE>Página Web Start</TITLE></HEAD>
<BODY>
  <a href="nombreDelFichero.jnlp">
    Ejecuta la aplicación Java Web Start
  </a>
</BODY>
</HTML>
```

## Implementación de una aplicación Java Web Start

La estructura de implementación de una aplicación Java Web Start es similar a la de una aplicación autónoma; es decir, puede usar ventanas (objetos `JFrame`) y tiene método `main`.

Este tipo de aplicaciones pueden utilizar una librería, `javaws.jnlp`, que el resto de aplicaciones no pueden. Esta librería viene incluida en el fichero `<JDK_DIR>\jre\lib\javaws.jar` con el JDK 5.0 de *Sun Microsystems*. Si se desea usar la librería, deberá incluirse en el `CLASSPATH` al compilar la aplicación. Con ella, la aplicación puede, entre otras cosas, cargar páginas Web en el navegador, conocer el servidor desde donde se descargó, etc.

Se trata de una librería especialmente útil en aplicaciones con permisos limitados, porque al utilizarla se puede acceder, desde estas aplicaciones, a ficheros del disco, al portapapeles, a las impresoras, etc. La forma de acceder a estos recursos es siempre segura, debido a que el usuario es, en todo momento, avisado de los peligros de seguridad que ello conlleva.

En este libro no se muestra esta librería, sino que simplemente es utilizada para saber el servidor del que se descarga la aplicación Java Web Start y acceder a los recursos que en él se encuentran.

## **Firmar un fichero .jar**

Firmar un fichero .jar es una técnica que permite asegurar que el fichero no ha sido alterado por alguien distinto al creador. Cuando la aplicación Java Web Start está configurada en el fichero .jnlp con todos los permisos, es obligatorio firmar el fichero .jar en el que se encuentra la aplicación.

Las connotaciones de seguridad no se van a tratar en este libro; por tanto, se van a firmar los ficheros .jar como un trámite necesario para que la aplicación Java Web Start funcione. Para una mayor información sobre seguridad se recomienda consultar la documentación.

Para poder firmar ficheros .jar es necesario crear un almacén de claves, este paso sólo se realiza una vez, ejecutando el comando que se encuentra en el directorio <JDK\_DIR>\bin:

```
keytool -genkey -keystore myKeystore -alias mySelf
```

Al ejecutar el comando anterior, el sistema pide una contraseña, un nombre, etcétera (se puede poner cualquier cosa, no se trata de una información relevante). Creará un fichero llamado myKeystore, el cual será usado para firmar los ficheros .jar. A continuación se ejecutará el siguiente comando:

```
keytool -selfcert -alias mySelf -keystore myKeystore
```

En la ejecución del comando anterior se pregunta por la contraseña que se había introducido previamente. Una vez hecho esto, por cada fichero .jar que se quiera firmar, hay que ejecutar el comando siguiente:

```
jarsigner -keystore myKeystore fichero.jar mySelf
```

Si el fichero `myKeystore` se encuentra en un lugar diferente al lugar donde se encuentra el `.jar`, habrá que indicar la ruta donde se encuentra precediendo a su nombre.

### Acceso a los ficheros desde una aplicación Java Web Start

Cuando se descarga una aplicación Java Web Start todos los ficheros empaquetados en el `.jar` se descargan antes de comenzar la ejecución. Para acceder a estos ficheros, puesto que están en el `CLASSPATH`, se sigue el esquema visto anteriormente, el cual se repite de nuevo:

```
...
InputStream input = this.getClass().
    getResourceAsStream("nombreFichero.txt");

//... leemos los datos del InputStream

input.close();
...
```

También hay que decir que, pese a que la aplicación Java Web Start esté empaquetada, se puede acceder a los ficheros del servidor Web desde donde se descargó dicha aplicación usando la clase estándar de Java `java.net.URL`. El esquema básico sería el siguiente:

```
try {

    BasicService bs = (BasicService)ServiceManager
        .lookup("javax.jnlp.BasicService");

    URL url = new URL(bs.getCodeBase() + "/"
        + "nombreFichero.txt");

    InputStream input = url.openStream();

    //... leemos los datos del InputStream

    input.close();

}
catch(UnavailableServiceException e) {}
catch(IOException e) {}
```

Como se puede ver en el esquema mostrado, se utiliza la interfaz `BasicService` y la clase `ServiceManager` para conocer la URL del servidor desde donde se descarga la aplicación Java Web Start. `BasicService` y `ServiceManager` se encuentran en la librería `javax.jnlp` mencionada anteriormente.

## 8.4. Applet

Un Applet es una pequeña aplicación Java que se ejecuta dentro de una página Web que está siendo visualizada en un navegador de Internet. Los Applets permitieron una gran expansión de Java en sus inicios, aunque ahora no se utilizan tanto. Sus características fundamentales son:

- Además de la máquina virtual de Java, necesitan un navegador Web o un visor de Applets para poder ejecutarse.
- Usualmente, se descargan desde un servidor Web antes de ejecutarse.
- Comienza su ejecución cuando se carga la página que contiene el Applet en un navegador Web.
- Su interfaz gráfica está formada por una región rectangular dentro de una página HTML. El tamaño y posición de la región se define en el código HTML de la página.
- Normalmente finaliza la aplicación cuando el usuario abandona la página.
- Tienen restricciones de seguridad. Dependiendo de determinados aspectos, no pueden acceder a disco y sólo pueden establecer conexiones de red con el servidor del que se descargaron.

### Seguridad en los Applets

Al igual que con las aplicaciones Java Web Start, existen dos tipos de Applets dependiendo de los recursos del sistema a los que quieran acceder:

- **Applets con todos los permisos** – Tendrán acceso total y sin restricciones a la máquina donde se ejecutan.

- **Applets con permisos limitados** – Tendrán acceso a la máquina con restricciones. No podrán acceder directamente al disco duro, ni establecer conexiones de red a servidores de Internet distintos al servidor desde donde se descargaron. A diferencia de las aplicaciones Java Web Start, no pueden usar una librería especial para acceder al disco duro del usuario.

Al igual que en una aplicación Java Web Start, siempre que un Applet intente hacer alguna operación no permitida, se elevará una excepción del tipo `java.lang.securityException`, que podrá ser capturada para reaccionar de forma adecuada.

Un Applet que esté en el disco duro del usuario y se ejecute con un fichero HTML del disco (no se ha descargado de un servidor Web) tiene todos los permisos. Si el Applet se descarga de Internet, por defecto, tiene permisos limitados.

Para conseguir que un Applet descargado de Internet tenga todos los permisos, debe estar empaquetado en un fichero `.jar` y éste debe estar firmado. En el momento en que el Applet se ejecute por primera vez por el usuario, se le pedirá la autorización pertinente al igual que se hace con las aplicaciones Java Web Start.

## El Applet en el servidor Web. La página HTML

Un Applet se puede distribuir como un conjunto de ficheros `.class` o como un único fichero `.jar` empaquetado. Estos ficheros se colocan en un directorio público del servidor Web, junto con el resto de los ficheros HTML del sitio.

Un Applet se visualiza dentro de un documento HTML. En el mismo documento están todos los datos necesarios para la carga y ejecución del Applet, para lo que se usa una etiqueta especial. A continuación se muestra un ejemplo de esta etiqueta.

```
<APPLET
  CODEBASE = "."
  ARCHIVE = "applet.jar"
  CODE = "paquete.ClaseApplet.class"
```

```
NAME = "Nombre para el navegador"
WIDTH = 400
HEIGHT = 300
VSPACE = 0
HSPACE = 0
>
<PARAM NAME = "param1" VALUE = "valor1">
<PARAM NAME = "param2" VALUE = "valor2">

Este texto se muestra si el navegador no soporta java

</APPLET>
```

El atributo `CODEBASE` permite especificar la ruta donde se encuentra el Applet, relativa a la página HTML que lo contiene. Para indicar que la aplicación está en el mismo directorio que el documento HTML, el valor de este atributo debe ser un punto (“.”).

Si la aplicación está empaquetada como un fichero `.jar`, este atributo es obligatorio; sin embargo, si la aplicación está formada por ficheros `.class`, el atributo es optativo y el valor por defecto es el directorio del fichero HTML.

El atributo `ARCHIVE` se utiliza para indicar el nombre del fichero `.jar` si la aplicación está empaquetada en un fichero de este tipo.

El atributo `CODE` es obligatorio. Especifica el nombre completo (con el paquete) de la clase principal del Applet. Un Applet no tiene método `main`, más adelante, en esta sección, se verá como construir la clase principal del Applet.

El atributo `NAME` es optativo. Consiste en una cadena de caracteres que simplemente describe al Applet. Esta cadena se leerá en la barra de estado del navegador cuando se haga referencia al mismo.

Los atributos `WIDTH` / `HEIGHT` son obligatorios. Se utilizan para definir el tamaño en píxeles asignado al Applet en la página. Este tamaño no puede cambiarse en el código del Applet.

Los atributos `HSPACE` / `VSPACE` son optativos. Representan el espacio horizontal o vertical en píxeles alrededor del Applet.

El elemento `PARAM` es optativo y puede repetirse tantas veces como sea necesario. Permite especificar parámetros que, posteriormente, se van a poder leer en la ejecución del Applet. Son similares a la lista de argumentos utilizada por el método `main` en las aplicaciones Java autónomas. Los parámetros se componen de dos atributos: `NAME` y `VALUE`, ambas cadenas de caracteres. Cuando se desea obtener el valor de un parámetro se indica el `NAME` para obtener el `VALUE`.

## Implementación de un Applet. Clase `JApplet`

La estructura básica de construcción de un Applet difiere sustancialmente de la de una aplicación autónoma de Java: no existe método `main` y la clase principal de la aplicación (aquella en la que se inicia la ejecución) debe heredar de la clase `java.swing.JApplet`, la cual, a su vez, hereda de la clase `java.applet.Applet`.

La ejecución de una aplicación autónoma comienza en el método estático `main` de la clase principal. En el caso de un Applet, se inicia la ejecución instanciando un objeto de la clase principal con el constructor sin parámetros. Además, la clase `Applet` (clase *abuela* de la clase principal del Applet que se esté construyendo) define una serie de métodos que serán invocados por el navegador ante determinadas circunstancias. El desarrollador podrá redefinirlos para ejecutar el código deseado cuando sean invocados. Estos métodos se describen a continuación:

```
public void init()
```

Este método se invocará siempre después de invocar el constructor y antes de que el applet se muestre por pantalla. Debe redefinirse para construir la interfaz gráfica de usuario. Para ello, la clase `JApplet` dispone del método `setContentPane(...)`, cuya función es similar al homónimo de la clase `JFrame`. El tamaño y título del Applet no se especifican desde el código, esto se indica en el documento HTML. Por último, hay que destacar que pese a que la clase `JApplet` dispone del método `setVisible(...)`, no debe llamarse, puesto que será el navegador el que haga visible al Applet cuando sea necesario.

```
public void stop()
```



El método `stop()` será invocado por el navegador cuando el usuario abandone la página que contiene al Applet o cierre el navegador. En este método se pueden parar animaciones o música, por ejemplo.

```
public void start()
```

Este método se ejecuta después del método `init()` y cada vez que se muestra el Applet por pantalla. Se puede usar, por ejemplo, para reiniciar la música, reiniciar una animación, etc.

```
public void destroy()
```

Se usa cuando el Applet va a ser destruido. Se utiliza para cerrar las conexiones con una base de datos, por ejemplo.

Desde un Applet es posible, por ejemplo, conocer la URL de la página que lo contiene, abrir páginas Web en el navegador actual, etc. Se accede a esta funcionalidad con los métodos de la clase `Applet`.

Para obtener los valores de los parámetros que se especifican en la página HTML dentro de la etiqueta `<APPLET>`, se utiliza el método `public String getParameter(String name)` de la clase `java.applet.Applet`.

## **Empaquetado de un Applet en un fichero .jar**

Para poder empaquetar todos los ficheros del Applet en un fichero `.jar` se usa un comando ligeramente diferente a cuando se empaqueta una aplicación autónoma o Java Web Start, ya que no hay que indicar la clase que tiene el método `main`. En el directorio donde se encuentren los ficheros `.class` y el resto de ficheros de la aplicación se debe ejecutar el siguiente comando

```
jar cvf applet.jar *
```

## **Acceso a los ficheros desde un Applet**

Si el Applet se descarga desde un servidor Web con ficheros `.class`, los ficheros de la aplicación no son descargados hasta que no se indique explícitamente mediante código. Si los ficheros se encuentran alojados en el mismo lugar que los ficheros `.class`, se van a encontrar en el

CLASSPATH de la aplicación y, por tanto, se podrá acceder a ellos como se ha visto anteriormente, usando el siguiente esquema:

```
...
InputStream input = this.getClass().
    getResourceAsStream("nombreFichero.txt");

//... leemos los datos del InputStream

input.close();
...
```

Si estos ficheros se encuentran en un lugar diferente de los ficheros `.class` del Applet, se pueden descargar igualmente del servidor usando la clase `URL`, pero esta vez habrá que usar el siguiente esquema básico:

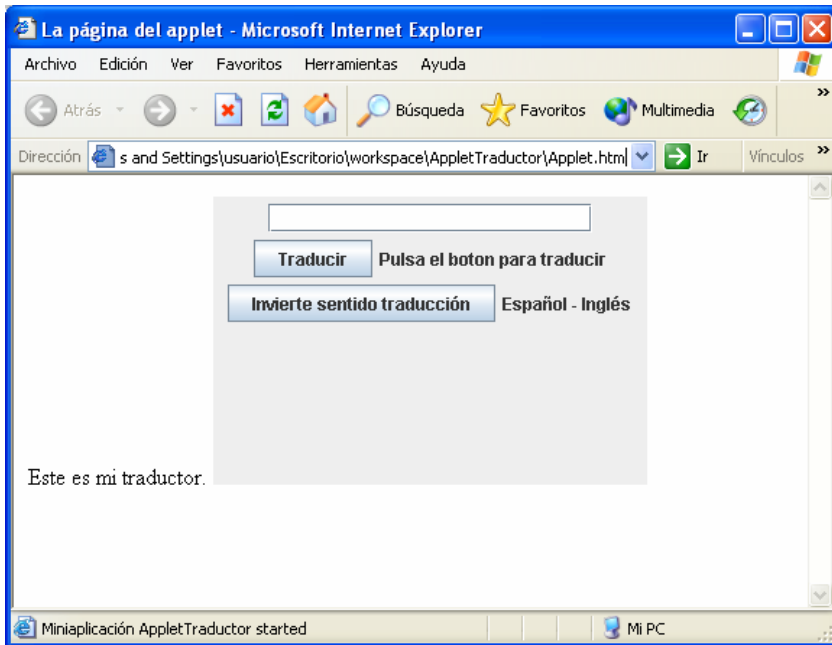
```
try {
    URL codeBase = this.getCodeBase();
    URL url = new URL(codeBase.getProtocol(), codeBase
        .getHost(), codeBase.getPort(),
        "nombreFichero.txt");
    InputStream input = url.openStream();

    //... leemos los datos del InputStream

    input.close();
} catch (MalformedURLException e) {
} catch (IOException e) {}
```

### **Ejemplo 8.2**

Para mostrar los pasos básicos de construcción de un Applet, se va a transformar la aplicación de traducción de una versión cualquiera (la que se prefiera) en un Applet. Para ello, es necesario construir una página Web, en la figura 8.1 se muestra como se visualizará en el navegador.



**Figura 8.1:** Página Web con Applet incorporado

El código fuente de la página Web es el siguiente:

**libro/ejemplos/ejemplo8.2/Applet.htm**

```
<HTML>
<HEAD>
<TITLE> La página del applet </TITLE>
</HEAD>
<BODY>
Este es mi traductor.
<APPLET CODE =
    "AppletTraductor.class" WIDTH = 300 HEIGHT = 200>
</APPLET>
</BODY>
</HTML>
```

Con respecto a la aplicación de traducción (en este caso particular se ha tomado la versión 5.0), lo único que hay que hacer es sustituir la clase `VentanaTraductor` y `AplicacionTraductor` por la siguiente clase `AppletTraductor`:

**libro/ejemplos/ejemplo8.2/AppletTraductor.java**

```
import javax.swing.*;

public class AppletTraductor extends JApplet {

    public void init(){
        Traductor traductor = new Traductor();
        this.setContentPane(new PanelTraductor(traductor));
    }
}
```

**8.5. Sumario de la distribución de aplicaciones Java**

Dadas las distintas posibilidades a la hora de distribuir una aplicación, dependiendo de su tipo o de si requieren todos los permisos, se presenta a modo de resumen la tabla 8.1 con las posibilidades más representativas.

Tipo	Formato	Preparación para distribuir	Ejecución
Aplicación autónoma	Múltiples ficheros .class	Es la salida del compilador Se dejan todos los ficheros en un directorio	Línea de comandos: java paquetes.ClaseMain  Se crea un fichero .BAT con este comando y se hace doble clic en él
	Un único fichero .jar	Se empaquetan los .class y todos los ficheros necesarios con: jar cmvf mainClass.txt aplicacion.jar *  mainClass.txt <div>Main-Class: ClaseMain</div>	Línea de comandos: java -jar aplicacion.jar  Doble clic en el fichero
Aplicación Java Web Start	Un único fichero .jar	Se empaquetan los .class y todos los ficheros necesarios con: jar cmvf mainClass.txt aplicacion.jar *  mainClass.txt <div>Main-Class: ClaseMain</div>  Después hay que firmar el .jar si se quieren todos los permisos.	Al pulsar un hiperenlace a un fichero .jnlp que contiene una referencia a la aplicación.

Applet	Múltiples ficheros .class	Es la salida del compilador. Se publican en un servidor web o se dejan en un directorio local	Al visitar la página que referencia al applet
	Un único fichero .jar	Se empaquetan los .class y todos los ficheros necesarios con: <code>jar cvf applet.jar *</code>  Se publican en un servidor web o se dejan en un directorio local  Si se quieren todos los permisos hay que signar el .jar	

**Tabla 8.1:** Tipos de aplicaciones con interfaz gráfica y sus características



## Referencias

- [Abascal01] J. Abascal, J. J. Cañas, M. Gea, A. B. Gil, J. Lorés, A. B. Martínez Prieto, M. Ortega, P. Valero, M. Vélez. *La interacción persona-ordenador*. Jesús Lorés, editor. Lleida, España, 2001.
- [Neg94] N. Negroponte. *Being Digital*. Vintage books, Nueva York, 1994.
- [Nor88] D. Norman. *The design of everyday things*. Currency/Doubleday, Nueva York, 1988.
- [Booch99] G. Booch, J. Rumbaugh, I. Jacobson; “*El Lenguaje Unificado de Modelado*”. Addison Wesley Iberoamericana, Madrid, 1999.

Para la escritura de este libro se han consultado direcciones de Internet oficiales de *Sun Microsystems*, así como otras direcciones de documentación técnica de Swing.

Página principal de Sun sobre Java:

<http://java.sun.com>

JavaDoc en línea del paquete `javax.swing`:

<http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/package-summary.html>

JavaDoc en línea del paquete `java.awt`:

<http://java.sun.com/j2se/1.5.0/docs/api/java/awt/package-summary.html>

Artículos oficiales sobre Swing y tecnologías relacionadas:

<http://java.sun.com/products/jfc/tsc/articles/>

Tutorial oficial de Swing y tecnologías relacionadas:

*<http://java.sun.com/docs/books/tutorial/uiswing/>*

Relación de aplicaciones Java construidas con Swing, destacadas por su interfaz gráfica de calidad:

*<http://java.sun.com/products/jfc/tsc/sightings/>*

Libro sobre Swing de la editorial Manning:

*<http://www.manning.com/sbe/>*

Sitio principal donde poder encontrar información sobre Swing (noticias, APIs, artículos, etcétera):

*<http://www.javadesktop.org>*

Tutorial de Swing en castellano:

*<http://www.programacion.com/java/tutorial/swing/>*