Programación Orientada a Objetos

AWT/SWING

junio, 2003

AWT & Swing.

Dos vías para la construcción de Interfaces Gráficas de Usuario en Java

Luis Bazo García

Raúl García Neila

Departamento de Informática y Automática

Universidad de Salamanca

Información de los autores:

Luis Bazo García
Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 - Salamanca
luisbazo2000@yahoo.es

Raúl García Neila
Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 - Salamanca
wickermorgan@hotmail.com

Resumen

Una breve introducción hacia el conocimiento de *AWT* y *Swing* dos bibliotecas de clases orientadas a la elaboración de Interfaces Graficas de Usuario en *Java*.

En este informe, se comenta brevemente el funcionamiento de los modelos de componentes y eventos que incorporan las citadas bibliotecas. Además de reseñar las principales novedades de *Swing* respecto a *AWT*.

Abstract

Brief introduction towards the knowledge of AWT and Swing two libraries of classes oriented to the elaboration of GUI in Java.

In this report, one briefly comments the operation of the models of components and events that incorporate the mentioned libraries. It is commented too the main new features of Swing with respect to AWT.

Tabla de Contenido

1. Introducción

Al programar en *Java* no se parte de cero. Cualquier aplicación que se desarrolle se apoya en un gran número de clases existentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el *API* o *Application Programming Interface* de *Java*). *Java* incorpora muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de *software* o fabricantes de ordenadores (*threads*, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.).

El concepto de *API* muestra la importancia de una buena construcción en las bibliotecas de clases, poniendo de manifiesto que un desarrollo orientado a la reutilización de las clases puede dar lugar a herramientas cada vez más potentes que facilitan el diseño de conceptos y paradigmas, que partiendo de cero serían prácticamente imposibles.

Uno de los aspectos que ha cobrado mayor relevancia en la informática actual en los últimos años es el desarrollo de aplicaciones con interfaces más amigables, asimilables e intuitivas para el usuario. Por lo tanto, el desarrollo de unas bibliotecas para la creación de *GUI's (Guide User Interfaces)* potentes y fáciles de utilizar ha supuesto un gran avance en el campo del desarrollo de aplicaciones.

El gran apogeo de *Java* y la gran cantidad de facilidades que da su extensa documentación, unido a la especial relevancia de la construcción de interfaces gráficas en la actualidad, hacen de *AWT* y *Swing* dos claros ejemplos de la importancia de conceptos como la reutilización de *software* y de las facilidades que ofrece el desarrollo orientado a objetos.

2. *AWT*

El *Abstract Windows Toolkit* es una colección de clases orientadas a la construcción de interfaces gráficas de usuario (GUI) en *Java*.

2.1 Un poco de historia

El *AWT* ha estado presente desde la versión Java 1.0.

Java 1.1 modificó ampliamente *AWT* sobretodo en lo que respecta al modelo de eventos que mas tarde veremos.

El *AWT* que se estudia en este trabajo es el *AWT* de *Java* 1.1.

2.2 Introducción a los componentes y eventos del *AWT*

Los componentes son aquella serie de objetos que pueden formar parte de nuestra interfaz como botones, menús, barras de desplazamiento, cajas, áreas de texto...

Los componentes tienen que estar situados obligatoriamente en un contenedor de componentes (*container*).

Los eventos son una forma de comunicar al programa todo lo que el usuario, mediante el ratón y el teclado, esta realizando sobre los componentes.

2.2.1 **Por qué se genera un evento**

Cuando hacemos algo con algún componente se produce un determinado tipo de evento que el sistema operativo transmite al *AWT*. Este reacciona creando un objeto de una determinada clase de evento, derivada de `AWTEvent`.

2.2.2 Cómo se gestiona un evento

El evento tendrá que ser gestionado por algún método y esto se consigue gracias a que el modelo de eventos de *Java* se basa en que los objetos sobre los que se producen los eventos (*event sources*) "avisan" a los objetos que gestionan los eventos (*event listeners*) para que actúen en consecuencia. Los objetos gestores de eventos deben de disponer de los métodos adecuados para saber responder. *Java* obliga a los event listeners a implementar los métodos de las interfaces `Listener` que *Java* proporciona para cada tipo de evento. Esto es que *Java* dispone de una interfaz `Listener` para cada evento con unos determinados métodos (cabeceras) que el usuario tendrá que implementar en sus objetos *event listeners* dentro de su aplicación interactiva.

2.3 Componentes y eventos soportados por el *AWT*

2.3.1 Componentes y jerarquía

Las características más importantes de los componentes son:

- Todos los componentes excepto los contenedores de más alto nivel (`Window` y sus descendientes) deben de estar dentro de un contenedor.
- Un contenedor se puede poner dentro de otro contenedor.
- Un componente solo puede estar dentro de un contenedor.

Todos los componentes del *AWT* de *Java* son objetos que pertenecen a una jerarquía de clases según se muestra en la figura.

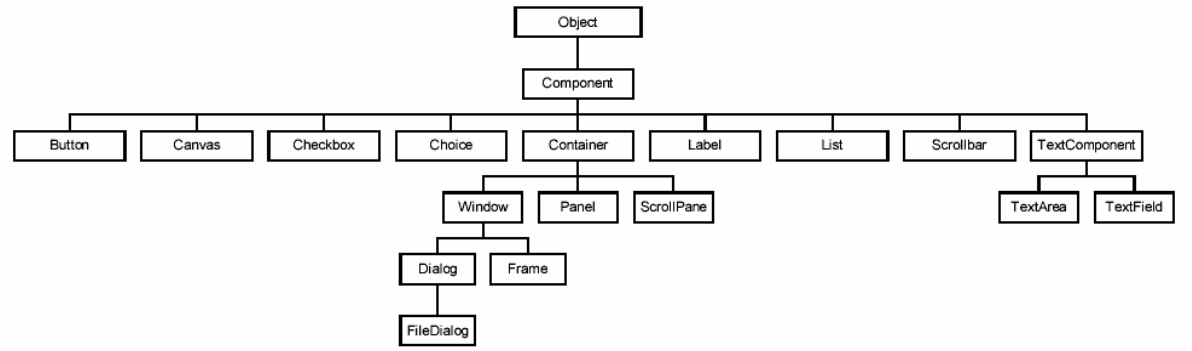

Figura 1. Jerarquía de componentes de *AWT*.

Para añadir un componente a un contenedor se utiliza la el método `add ()` de la clase `Container`.

```
containerName.add (componentName);
```

2.3.2 Eventos y jerarquía

Todos los eventos de *AWT* son objetos de clases que pertenecen a una jerarquía como la de la figura.

Las clases de la jerarquía se encuentran se encuentran definidas en el *package* `java.awt.event`

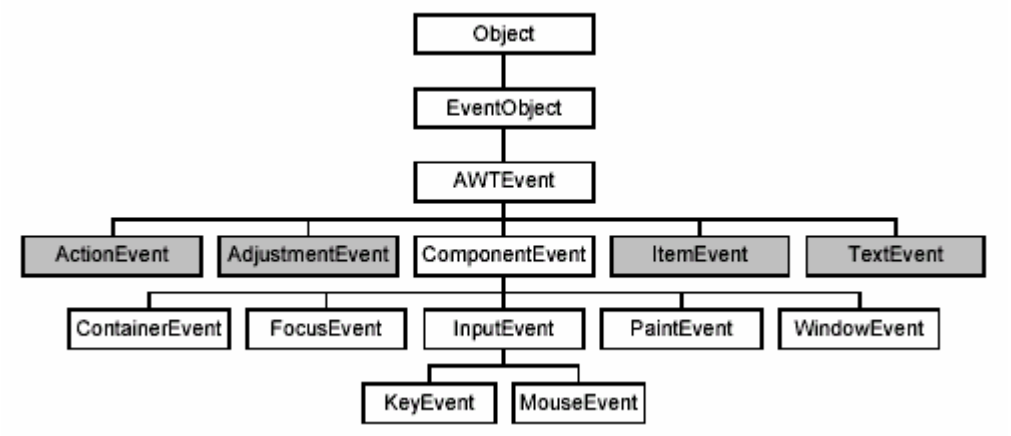

Figura 2. Jerarquía de eventos en *AWT*.

En *AWT* existen dos tipos de eventos, los eventos de alto nivel y los eventos de bajo nivel. Los eventos de alto nivel suelen implicar muchos de bajo nivel.

Los de alto nivel son:

- `ActionEvent`, clicar sobre un botón o elegir un elemento del menú.
- `AdjustmentEvent`, mover las barras de desplazamiento.
- `ItemEvent`, elegir valores.
- `TextEvent`, cambiar texto.

Los de bajo nivel son:

- `ComponentEvent`, eventos elementales relacionados con componentes.
- `ContainerEvent`, eventos elementales relacionados con contenedores.
- `KeyEvent`, eventos relacionados con las pulsaciones sobre el teclado.
- `MouseEvent`, eventos relacionados con las pulsaciones del ratón.
- `FocusEvent`, eventos relacionados con el focus.
- `WindowEvent`, eventos elementales relacionados con ventanas.

2.3.3 Relación entre componentes y eventos

Es muy importante conocer cual es la relación que existe entre eventos y componentes ya que a la hora de programar una interfaz gráfica de usuario necesitamos esta información. Por eso a continuación mostramos una tabla en donde se relaciona cada componente con sus correspondientes eventos más una pequeña explicación.

Component	Eventos Generados	Significado
`Button`	`ActionEvent`	Clicar en el botón
`Checkbox`	`ItemEvent`	Seleccionar o deseleccionar un item
`CheckboxMenuItem`	`ItemEvent`	Seleccionar o deseleccionar un item
`Choice`	`ItemEvent`	Seleccionar o deseleccionar un item

`Component`	`ComponentEvent`	Mover, cambiar tamaño, mostrar u ocultar un componente
	`FocusEvent`	Obtener o perder el focus
	`KeyEvent`	Pulsar o soltar una tecla
	`MouseEvent`	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón
`Container`	`ContainerEvent`	Añadir o eliminar un componente de un container
`List`	`ActionEvent`	Hacer doble click sobre un item de la lista
	`ItemEvent`	Seleccionar o deseleccionar un item de la lista
`MenuItem`	`ActionEvent`	Seleccionar un item de un menú
`Scrollbar`	`AdjustementEvent`	Cambiar el valor de la scrollbar
`TextComponent`	`TextEvent`	Cambiar el texto
`TextField`	`ActionEvent`	Terminar de editar un texto pulsando Intro
`Window`	`WindowEvent`	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre

Figura 3. Tabla de relación entre componentes y eventos.

Hay que considerar que porque un componente no se relacione con ningún evento esto no quiere decir que no los pueda recibir. Una clase que recibe un evento puede transmitírselo a sus sub-clases.

2.4 Pasos para crear una interfaz gráfica con *AWT*

Unos pasos sencillos para la elaboración de una primitiva interfaz con *AWT* serían los siguientes:

- Crear una clase con el método `main ()` que es donde empezará a correr nuestro programa.
- Crear una clase derivada de `Frame` que responda al evento `WindowClosing ()`.
- Añadir al contenedor todos los componentes que queremos que tenga nuestra interfaz gráfica.
  ```
  containerName.add (componentName);
  ```
- Implementar los objetos gestores (*event listeners*) que responden a nuestros eventos.

Un posible ejemplo sería:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class CerrarVentana extends WindowAdapter
4. {
5.    public void windowClosing(WindowEvent e)
6.    {
```

```
7.        System.exit(0);
8.    }
9. }

10. class Ventana extends Frame
11. {
12.     public Ventana()
13.     {
14.         CerrarVentana cv = new CerrarVentana();
15.         addWindowListener(cv);
16.         Button Boton = new Button();
17.       this.add(Boton);//se puede omitir this
18.       Boton.setLabel("Boton");
19.         setSize(400, 400);
20.         setTitle("Ventana");
21.         setVisible(true);
22.     }

23.     public static void main(String args[])
24.     {
25.         Ventana mainFrame = new Ventana() ;
26.     }
27. }
```

Este ejemplo es el que se va a tomar para explicar los aspectos más importantes de *AWT* modificándolo en el caso que sea necesario.

En las líneas 1 y 2 se importa a la aplicación los *packages* de clases `java.awt` y `java.awt.event` para incluir las clases de componentes y eventos respectivamente.

De la línea 3 a la 9 es donde se declara la clase gestora de eventos. Esta clase es una derivada de `WindowAdapter` osea que es una clase `Adapter` que se comentará más adelante.

De la 10 a la 22 es donde se define el *container*. Dentro de él en las líneas 14 y 15 se registran los eventos que nos interesan para la aplicación (de momento la aplicación solo responde al evento `windowClosing()`). También dentro del *container* se ha añadido un componente `Button` en la línea 17 y se han utilizado unos métodos heredaros de la jerarquía de clases de componentes y eventos en las líneas 19, 20, 21

De la 23 a la 26 se ha declarado el método `main ()` que es donde comenzará a ejecutarse la aplicación. Es en este método donde se instancia la clase Ventana (contenedor de la aplicación).

El resultado es un botón que no desencadena ninguna acción, redimensionable y que ocupa toda la ventana de programa.

Figura 4. Un ejemplo sencillo con utilizando *Abstract Windowing Toolkit* (*AWT*)

2.5 Interfaces `Listener`

La forma de gestionar eventos por *AWT* es que cada objeto que recibe un evento (*event source*) registra un objeto que lo gestione (*event listener*).

Para esto se utiliza el método `addEventObject` de la siguiente manera, `eventSourceObject.addEventListener(eventListenerObject);`

`EventSourceObject` es el objeto que recibe el evento y registra con el método `addEventListener` al objeto `eventListenerObject` para que gestione el evento cuando se produzca.

Las interfaces `Listener` son una forma de llevar esto a cabo.

El `eventListenerObject` debe de implementar la interfaz `Listener` para el evento que se desee gestionar.

Evento	`Interface Listener`	Métodos de `Listener`
`ActionEvent`	`ActionListener`	`actionPerformed()`
`AdjustementEvent`	`AdjustementListener`	`adjustementValueChanged()`
`ComponentEvent`	`ComponentListener`	`componentHidden(), componentMoved(), componentResized(), componentShown()`
`ContainerEvent`	`ContainerListener`	`componentAdded(), componentRemoved()`
`FocusEvent`	`FocusListener`	`focusGained(), focusLost()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`
`KeyEvent`	`KeyListener`	`keyPressed(), keyReleased(), keyTyped()`
`MouseEvent`	`MouseListener`	`mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()`
	`MouseMotionListener`	`mouseDragged(), mouseMoved()`

TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

Figura 5. Relación de eventos con su `Listener`

Cada interfaz `Listener` se corresponde con un tipo de evento. Así pues, si por ejemplo tenemos que gestionar un evento `ContainerEvent` existe una interfaz con los métodos relacionados con el `ContainerEvent` que son `componentAdded()` y `componentRemoved()`. Estos son los métodos que tiene que implementar el `eventListenerObject`. Cada vez que se añada o borre un componente de un *container* se generará el evento `ContainerEvent` y se ejecutará el método correspondiente cuya implementación estará en el `eventListenerObject`.

Es muy importante resaltar que el `eventListenerObject` debe de implementar todos y cada uno de los métodos incluidos en la interfaz `Listener`. Pero claro, si la aplicación del usuario no va a utilizar muchos de ellos, ¿para qué implementarlos? Habrá muchos métodos vacíos y para evitar esto se utilizan las llamadas clases `Adapter`.

En el ejemplo de arriba tendríamos que implementar todos los métodos correspondientes a la interfaz `WindowListener` y no solo `windowClosing()` como está hecho en el ejemplo(el ejemplo esta hecho con clases `Adapter`).

2.6 Clases `Adapter`

Las clases `Adapter` permiten implementar únicamente aquellos métodos que van a hacer falta en la interfaz de usuario.

Hay 7 clases `Adapter`: `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MouseMotionAdapter` y `WindowAdapter`.

Las clases `Adapter` derivan de `Object` y son clases predefinidas con implementaciones vacías para todos los métodos.

Lo que tiene que hacer el programador es hacer que sus `eventListenerObject` deriven directamente de alguna de estas clases según el evento y después implementar únicamente aquellos métodos que vaya a utilizar en su aplicación como se hace en el ejemplo de arriba.

2.7 Clases Anónimas

Son muy útiles cuando solo se necesita un objeto `eventListenerObject` que contenga los métodos de `Listener` para gestionar el evento adecuado.

Un ejemplo podría ser el siguiente:

```
1. import java.awt.*;
2. import java.awt.event.*;
```

```
3. class Ventana extends Frame
4. {
5.     public Ventana()
6.     {
7.         addWindowListener(new WindowAdapter()
8.         {
9.             public void windowClosing(WindowEvent e)
10.            {
11.           dispose();
12.           System.exit(0);
13.        }
14.      });
15.   }
16.   public static void main(String args[])
17.   {
18.        System.out.println("Starting Ventana...");
19.        Ventana mainFrame = new Ventana();
20.        mainFrame.setSize(400, 400);
21.        mainFrame.setTitle("Ventana");
22.        mainFrame.setVisible(true);
23.   }
24. }
```

Se puede ver como en la línea 7 se instancia la clase `WindowAdapter` en un objeto sin nombre dentro del método `addWindowListener` además de, aprovechándonos de la sintaxis del lenguaje, implementamos también ahí los métodos que nos hagan falta.

2.8 Clases componentes y eventos

A continuación vamos a ver brevemente las clases componentes y eventos de *AWT*.

Hay que saber que cuando se produce un evento el *AWT* de *Java* genera automáticamente un objeto de ese evento que tendrá métodos que el usuario podrá utilizar.

- Clase `Component` :
  Una clase abstracta de la que derivan todas las clases del modelo de componentes de *AWT*.
- Clase `EventObject` y `AWTEvent` :
  Todos los métodos de las interfaces `Listener` relacionadas con el *AWT* tienen como argumento único una clase que desciende de la clase `java.awt.AWTEvent`. La clase `AWTEvent` no define ningún método pero hereda de `EventObject` el método `getSource` que devuelve una referencia al objeto que produjo el evento.
- Clase `ComponentEvent` :
  Se genera cuando un `Component` de cualquier tipo se muestra, se oculta o cambia de posición o tamaño.
- Clases `InputEvent, MouseEvent, MouseMotionEvent`:
  De la clase `InputEvent` descienden los eventos del ratón y teclado detectando si los botones del ratón o las teclas especiales han sido pulsadas. Se produce un `MouseEvent` cada vez que el ratón entra o sale de un componente visible en la pantalla, al clicar o cuando se pulsa o suelta un botón del ratón. Se produce un `MouseMotionEvent` cuando se mueve el ratón donde quiera que sea.
- Clase `FocusEvent`:
  Se produce cuando un componente pierde o gana el *focus*.

- Clase `Container`:
  Una clase muy general. Nunca se crea un objeto de esta clase pero heredan sus métodos las clases `Frame` y `Panel`.
- Clase ContainerEvent:
  Se genera cada vez que un Component se añade o se retira de un Container. Este evento solo tiene papel de aviso no es necesario gestionarlo.
- Clase `Window`:
  Los objetos de la clase `Window` son ventanas de máximo nivel sin bordes y sin barras de menú. Son más interesantes las clases que derivan de ella `Frame` y `Dialog`.
- Clase `WindowEvent`:
  Se produce cada vez que se abre, cierra, iconiza, restaura, activa o desactiva una ventana.
- Clase `Frame`:
  Es una ventana con un borde y que puede tener una barra de menús. Si una ventana depende de otra ventana es mejor utilizar una `Window` que un `Frame`.
- Clase `Dialog`:
  Es una ventana que depende de otra ventana (una `Frame`) si una `Frame` se cierra se cierra también los `Dialogs` que depende de ella. Si se minimiza sus `Dialogs` desparecen; si se restablece sus `Dialogs` aparecen de nuevo. Por defecto los `Dialogs` son no modales es decir, no requieren atención inmediata del usuario.
- Clase `FileDialog`:
  Muestra una ventana de dialogo en la cual se puede seleccionar un fichero. Las constantes enteras `LOAD` y `SAVE` definen el modo de apertura del fichero.
- Clase `Panel`:
  Un panel es un Container de propósito general se puede utilizar tal cual para contener otras componentes y para crear sub-clases de finalidad mas especificas. No tiene métodos propios y suele utilizar los heredados de Component y Container. Un Panel puede contener otros Panel una gran ventaja respecto a los otros tipos de containers.
- Clase `Button`:
  Lo más importante es que al clicar sobre él se genera un evento de la clase `ActionEvent`. El aspecto de un `Button` depende de la plataforma pero la funcionalidad es la misma. Se pueden cambiar el texto y el color del `Button` si se desea.
- Clase `ActionEvent`:
  Estos eventos se producen al clicar con el ratón en un botón (`Button`), al elegir un comando de un menú (`MenuItem`), al hacer doble clic en un elemento de una lista (`List`), y al pulsar intro para introducir texto en una caja de texto (`TextField`).
- Clase `Canvas`:
  Una `Canvas` es una zona rectangular de pantalla en la que se puede dibujar y en la que se pueden generar eventos. Las `Canvas` permiten realizar realizar dibujos, mostrar imágenes y crear componentes a medida de modo que muestren un aspecto similar en todas las plataformas.
- Component `Checkbox` y Clase `CheckboxGroup`:
  Los objetos de la clase `Checkbox` son botones de opción o de selección con dos posibles valores *on* y *off*. La clase `CheckboxGroup` permite la opción de agrupar varios `Checkbox` de modo que uno y solo uno este en *on*. Al cambiar la selección de un `Checkbox` se produce un `ItemEvent`.
- Clase `ItemEvent`:
  Se produce un `ItemEvent` cuando ciertos componentes cambian de estado (*on/off*).
- Clase `Choice`:

Permite elegir un item de una lista desplegable los objetos `Choice` ocupan menos espacio en pantalla que los `Checkbox`. Al elegir un item se genera un `ItemEvent`.

- Clase `Label`:
  La clase `Label` introduce en un *container* un texto no editable y no seleccionable.
- Clase List:
  Viene definida por una zona de pantalla con varias líneas de las que se muestran solo algunas y entre las que se puede hacer una selección simple y múltiple. Las `List` generan eventos de la clase `ActionEvent` e `ItemEvents`.
- Clase `Scrollbar`:
  Es una barra de desplazamiento con un cursor que permite introducir y modificar valores, con pequeños y grandes incrementos. Al cambiar el valor de la `Scrollbar` se produce un `AdjustementEvent`.
- Clase `AdjustementEvent`:
  Hay cinco tipos de `AdjustementEvent`: *track* (se arrastra el cursor de la `Scrollbar`), *unit increment y unit decrement* (se clica en las flechas de la `Scrollbar`), *block increment y block decrement* (se clica encima o debajo del cursor).
- Clase `ScrollPane`:
  Es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos `Scrollbars` que son visibles solo si son necesarias por defecto.
- Clase `TextArea` y `TextField`:
  Ambas componentes se heredan de la clase `TextComponent` y muestran texto seleccionable y editable. `TextArea` ofrece posibilidades de edición de texto seleccionables además puede estar compuesta de varias líneas. No se pueden crear objetos de la clase `TextComponent` porque su constructor no es `public`. Reciben eventos `TextEvent` y todos los de sus super-clase.
- Clase `TextEvent`:
  Se produce un `TextEvent` cada vez que cambia algo en un `TextComponent`. Se puede desear evitar ciertos caracteres y para eso es necesario gestionar los eventos correspondientes.
- Clase `KeyEvent`:
  Se produce un `KeyEvent` al pulsar sobre el teclado. Hay dos tipos *key-typed* que representa la introducción de un carácter *Unicode*; y *key-pressed y key-released* que representan pulsar o soltar una tecla.

Menús: Los menús de *Java* no descienden de `Component` sino de `MenuComponent` pero tienen un comportamiento similar pues aceptan `Events`.

Para crear un menú se debe crear primero una `MenuBar`; después se crean los `Menus` y los `MenuItem`. Los `MenuItems` se añaden al `Menu` correspondiente; los `Menus` se añaden a la `MenuBar` y la `MenuBar` se añade a un `Frame`. También puede añadirse un `Menu` a otro `Menu` para crear un sub-menú, del modo que es habitual en *Windows*. La clase `Menu` es sub-clase de `MenuItem`. Esto es así precisamente para permitir que un `Menu` sea añadido a otro `Menu`.

- Clase `MenuShortcut`:
  Derivada de `Object`, representa las teclas aceleradoras que pueden utilizarse para activar los menús desde teclado, sin ayuda del ratón.
- Clase `Menubar`:
  Es un contenedor de objetos `Menu`.
- Clase `Menu`:

El objeto `Menu` define las opciones que aparecen al seleccionar uno de los menús de la barra de menús. En un `Menu` se pueden introducir objetos `MenuItem`, otros `Menu`, objetos `CheckboxMenuItem` y separadores.

- Clase `MenuItem`:

  Representa las distintas opciones de un menú. Al seleccionar un objeto `MenuItem` se generan eventos del tipo `ActionEvent`.

- Clase `CheckboxMenuItem`:

  Son items de un `Menu` que pueden estar activados o no. No generan `ActionEvent`, generan `ItemEvent` de modo similar a la clase `CheckBox`.

- Menus *Pop-Up*:

  Son menús que aparecen en cualquier parte de la pantalla al cliclar con el botón derecho del ratón (*pop-up trigger*) sobre un componente determinado.

2.9 *Layout Managers*

Los diferentes tipos de `LayoutManager` son clases que nos ayudan a distribuir los componentes en los contenedores de una forma u otra.

Cada *container* tiene un `LayoutManager` por defecto que se crea en el constructor, si se desea utilizar otro `LayoutManager` habrá que decírselo al `container` utilizando el método `SetLayout`, de la forma:

`miContenedor.SetLayout (new xxxLayout( ))` *Donde xxx es un tipo de `LayoutManager`

Los diferentes tipos de `LayoutManager` incluidos en *AWT* son:

Figura 6. `FlowLayout` y `BorderLayout`

Figura 7. `GridLayout` y `GridBagLayout`

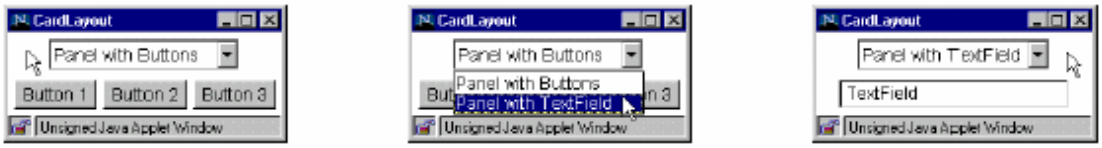

Figura 8. Varias pantallas de `CardLayout`

2.10 Gráficos y animaciones con *AWT*

La clase `Component` tiene tres métodos muy importantes `paint (),repaint()` y `update()`.

- Método `paint(Graphics g)`

El método `paint` que esta definido en la clase `Component` no hace nada por defecto y hay que redefinirlo en alguna de sus clases derivadas. Este método se llama automáticamente cuando se pinta la ventana por primera vez y cada vez que *AWT* entiende que debe ser redibujada.

- Método `update(Graphics g)`

Cuando se llama al método update se redibuja la ventana con el color de fondo y luego se llama al método `paint`. Este método puede ser llamado por el programador explícitamente.

- método `repaint()`

Es el método mas utilizado por el programador. Se puede utilizar de las cuatro formas siguientes:

- ✓ `repaint()` .- Se llama al método `update`.
- ✓ `repaint(long time)` .- Se especifica el numero de milisegundos transcurridos los cuales se llama al método `update`.
- ✓ `repaint(int x, int y, int w, int h)` .- Se especifica la zona sobre la cual hay que aplicar `update`.
- ✓ `repaint(long time, int x, int y, int w, int h)` .- Combinación de las dos modalidades anteriores

2.10.1 Clase `Graphics`

La clase `Graphics` que define un objeto que se pasa como único argumento a los métodos `update` y `paint`, nos permite realizar dos cosas : dibujo de primitivas gráficas y dibujos en *gif* y *jpeg*.

Las primitivas gráficas son unos métodos definidos en la clase `Graphics` que nos permiten dibujar líneas, polígonos simples, texto, ... utilizando como coordenadas *pixels* como se muestra en la figura.

La clase `Graphics` también permite incorporar imágenes en formato *gif* y *jpeg*. Para conseguir esto hay que seguir los siguientes pasos.

- Crear un objeto `Image` y llamar al método getImage de la siguiente forma:

```
Image miImagen=getImage("Imagen.gif")
```

- Una vez cargada la imagen hay que representarla. Se redefine el método `paint` y se utiliza el método `DrawImage` de la clase `Graphics`. Este método es polimorfo, pero casi siempre hay que incluir en sus parámetros el nombre del objeto Image creado, las dimensiones de dicha imagen y un objeto `ImageObserver`. `ImageObserver` es un interfaz que declara métodos para observar el estado de la carga y visualización de la imagen.

2.10.2 **Animaciones con *AWT***

Con *AWT* se pueden hacer animaciones de una forma muy sencilla: se define el método `paint` repetidas veces haciendo que dibuje una imagen diferente a la anterior. Una forma de hacer esto declarando una clase `Thread` con un bucle infinito que duerma "x" milisegundos entre dibujo y dibujo. El problema es que se suele producir un parpadeo o *flipper*, aunque existen varias técnicas de evitar este problema como redefinir el método *update* cambiando solo aquello que hay que repintar o utilizando la técnica de doble *buffer*.

3 *Swing*

Swing es un paquete *Java* para la generación de la GUI en aplicaciones reales de gran tamaño, viene a complementar y ampliar al modelo de componentes y eventos de *AWT*, basándose en este. Es una de las *API'*s del *JFC* (*Java Foundation Classes*).

3.1 Un poco de historia

Swing era el nombre clave del proyecto que desarrolló los nuevos componentes que vienen a sustituir o complementar a los de *AWT*. Aunque no es un nombre oficial, frecuentemente se usa para referirse a los nuevos componentes y al *API* relacionado. Está inmortalizado en los nombres de paquete del *API Swing*, que empiezan con `javax.swing.` Esta versión de las *JFC* fue publicada como *JFC* 1.1, que algunas veces es llamada 'Versión Swing'. El *API* del *JFC* 1.1 es conocido como el *API Swing*.

Swing constituye la característica más importante que se ha añadido a la plataforma 1.2, como Sun ha preferido llamarla, *Java2*. Con esta última incorporación por fin se completa totalmente la parte gráfica de la programación en *Java*, ofreciendo al programador acceso a todas las características existentes en un entorno gráfico actual, así como un conjunto de componentes gráficos completo y fácilmente ampliable con el que construir la interfaz gráfica de usuario, o *GUI*, de nuestras aplicaciones y *applets*, de una forma totalmente transparente e independiente de la plataforma en la que ejecutemos nuestro código.

3.2 Eventos Swing

El modelo de eventos que utiliza *Swing* es el mismo que *AWT*, el de *Java* 1.1, añadiendo algunos nuevos eventos para los nuevos componentes. Utilizando igualmente las interfaces *Listener*, las clases *Adapter* o las clases anónimas para registrar los objetos que se encargaran de gestionar los eventos.

Algunos de los nuevos eventos son:

a.) Eventos de bajo nivel

- `MenuKeyEvent`
- `MenuDragMouseEvent`

b.) Eventos de alto nivel

- `AncestorEvent`: Antecesor añadido desplazado o eliminado.
- `CaretEvent`: El signo de intercalación del texto ha cambiado.
- `ChangeEvent`: Un componente ha sufrido un cambio de estado.
- `DocumentEvent`: Un documento ha sufrido un cambio de estado.
- `HyperlinkEvent`: Algo relacionado con un vínculo hipermedia ha cambiado.
- `InternalFrameEvent`: Un `AWTEvent` que añade soporte para objetos `JInternalFrame`.

- `ListDataEvent`: El contenido de una lista ha cambiado o se ha añadido o eliminado un intervalo.
- `ListSelectionEvent`: La selección de una lista ha cambiado.
- `MenuEvent`: Un elemento de menú ha sido seleccionado o mostrado o bien no seleccionado o cancelado.
- `PopupMenuEvent`: Algo ha cambiado en `JPopupMenu`.
- `TableColumnModelEvent`: El modelo para una columna de tabla ha cambiando.
- `TableModelEvent`: El modelo de una tabla ha cambiado.
- `TreeExpansionEvent`: El nodo de un árbol se ha extendido o se ha colapsado.
- `TreeModelEvent`: El modelo de un árbol ha cambiado.
- `TreeSelectionEvent`: La selección de un árbol ha cambiado de estado.
- `UndoableEditEvent`: Ha ocurrido una operación que no se puede realizar.

3.3 Modelo de componentes *Swing*

En principio implementa de nuevo todos los componentes gráficos existente en el *AWT*, pero en este caso con implementaciones ligeras, o *lighweight*, con todas las ventajas que esto implica. Además añade nuevas y útiles funcionalidades a estos componentes, tales como la posibilidad de presentar imágenes o animaciones en botones, etiquetas, listas o casi cualquier elemento gráfico.

Este paquete nuevo está enteramente basado en *AWT* y más específicamente en el soporte para interfaz de usuario ligero. Debido a ello y a ser puro *Java*, es posible hacer aplicaciones basadas en *Swing* desde la plataforma 1.1.5 y que funcione sin ningún problema con la *JVM* de dicha plataforma, así como con la incluida junto con los navegadores más actuales, lo que asegura que un *applet* realizado usando estos nuevos componentes funcionará sin problemas en dichos navegadores.

Entre los componentes que se incorporan en *Swing* está la reimplementación de todos los componentes gráficos existentes en *AWT* y que, para no confundir con los antiguos, ahora empiezan todos por J. Así en vez de `Button`, tenemos `JButton`.

La mayor diferencia entre los componentes *AWT* y los componentes *Swing* es que éstos últimos están implementados sin nada de código nativo. Esto significa que los componentes *Swing* pueden tener más funcionalidad que los componentes *AWT*, porque no están restringidos al denominador común, es decir las características presentes en cada plataforma. El no tener código nativo también permite que los componentes *Swing* sean vendidos como añadidos al *JDK* 1.1, en lugar de sólo formar parte del *JDK* 1.2.

Incluso el más sencillo de los componentes *Swing* tiene capacidades que van más allá de lo que ofrecen los componentes *AWT*. Por ejemplo:

- Los botones y las etiquetas *Swing* pueden mostrar imágenes en lugar de o además del texto.
- Se pueden añadir o modificar fácilmente los bordes dibujados alrededor de casi cualquier componente *Swing*. Por ejemplo, es fácil poner una caja alrededor de un contenedor o una etiqueta.
- Se puede modificar fácilmente el comportamiento o la apariencia de un componente *Swing* llamando a métodos o creando una subclase.
- Los componentes *Swing* no tienen porque ser rectangulares. Por ejemplo, los botones pueden ser redondos.
- Bordes complejos: Los componentes pueden presentar nuevos tipos de bordes. Además el usuario puede crear tipos de bordes personalizados.

Otro mejora importante es que ahora todos los componentes pueden presentar una pequeña leyenda de texto con una breve explicación, que es conocida como *tooltip*.

3.4 Un primer programa con *Swing*

Figura 9. Un programa sencillo con *Swing*

Este es el código de un sencillo programa hecho utilizando *Swing* a modo de "Hola Mundo".

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. public class UnJBoton  {
5. public static void main(String args[]) {
6. Frame f = new Frame();
7. JButton b = new JButton("Pulsame");
8. f.add(b);
9. f.pack();
10.  f.setVisible(true);
11.  b.addActionListener(new ActionListener() {
12.  public void actionPerformed(ActionEvent e) {
                System.out.println("HOLA");
13.  }
14.  });
15.  }
16.  }
```

En las líneas 1,2,3 se importan los packages de *AWT* y *Swing* necesarios para la aplicación.

Desde la línea 4 hasta el final del programa se implementa la clase `UnJBoton` en donde se instancia un objeto de tipo `Frame` (contenedor), se instancia también un `Jbutton` y se añade al contenedor `Frame` con el método `add`. En la línea 9 de dicha clase se utiliza el método pack que establece un tamaño por defecto al `Frame` que hemos creado. `SetVisible`, una línea más abajo muestra el `Frame` por pantalla y a continuación mediante una clase anónima registramos el objeto que gestionara el evento `ActionEvent`. La gestión de este evento es que cuando el usuario pulse el botón "púlsame" se escribe por la salida estándar "HOLA".

3.5 Descripción de algunos componentes *Swing*.

Se da una descripción breve de cada componente por la diversidad de métodos que contienen.

- `Jframe`: se trata de la implementación de la clase `Frame` añadiendo nuevas funcionalidades y una nueva filosofía, ya que ahora un `Frame` contiene varios tipos de paneles.
- `Jpanel`: es el nuevo contenedor básico de componentes gráficos.
- `JscrollPane`: esta nueva implementación de un panel con barra de desplazamiento no se limita sólo a proporcionar barras de desplazamiento en caso de que los componentes que contengan no entren en él área de visualización, sino que también se

encarga de proporcionar barras de desplazamiento a todo el resto de componentes gráficos existente.

- `Japplet`: Se trata de la reimplementación de la clase applet para que ésta pueda aprovechar todas las nuevas características existentes.
- `Jbutton`: botón que además de texto puede contener imágenes en cualquier posición en relación con el texto.
- `JtoggleButton`: se trata de una clase que no tiene equivalentes en *AWT*. Representa un botón que se mantiene presionado aunque dejemos de presionar con nuestro ratón sobre él, pero visualmente no se difiere de un `Jbutton` cuando no está activado.
- `Jlabel`: etiqueta de texto que puede contener imágenes en cualquier posición en relación con el texto.
- `JtextField`: Componente que sirve para conseguir la entrada de texto por parte del usuario.
- `JtextPane`: se trata también de un panel para visualizar texto, con la salvedad de que tiene la capacidad de mutar en relación al tipo de texto que se desee mostrar para poder visualizarlo correctamente.
- `Jlist`: presenta una lista de elementos de entre los que se puede elegir uno o varios simultáneamente.
- `JdesktopPane`: se trata de un panel base para el desarrollo de aplicaciones *MDI*.
- `JinternalFrame`: este panel no es una ventana en sí, sino que simula una ventana interior a un `JdesktopPane`. Como ventana interior puede ser movida, cerrada, o iconizada.
- `Jtable`: este componente presenta una rejilla en la que se puede colocar cualquier componente *Swing*.
- `JcheckBox`: es similar al componente checkbox que encontramos en cualquier lenguaje y que permite mostrar al usuario si la opción está seleccionada o no.
- `JpasswordField`: se encuentra justo por debajo de `Jtextfield` en la jerarquía de componentes, y permite la edición de texto sin realizar un eco de los caracteres tecleados en pantalla, que son sustituidos por un carácter "*".
- `JtextArea`: hereda de `JtextComponent`, con la diferencia es que un área de texto permite la edición de múltiples líneas. Hay que tener en cuenta que si queremos tener la capacidad de desplazarnos a lo largo de un texto que no cabe en el componente tendremos que colocar el `JtextArea` en un `JscrollPane`.
- `JprogressBar`: las barras de progreso permiten ver de forma gráfica la evolución de una tarea. Normalmente el avance se mide en porcentaje y la barra se acompaña de la visualización en texto del valor de dicho porcentaje. `Java` incorpora otra posibilidad para esta tarea: un monitor de progreso, que permite ver el avance de un proceso de forma más sencilla que las barras. Además al ser mostradas en un cuadro de dialogo, el usuario puede cerrarlas en cualquier momento e incluso llegar a cancelar el proceso.
- `JcomboBox`: es similar al `Choice` de *AWT*. Dispone de una lista desplegable de posibles valores, pero si no encontramos entre ellos el valor que buscamos, podemos teclearlo(si hemos habilitado la opción con `setEditable(true)`). Este control de *Swing* presenta una facilidad añadida: cuando pulsamos la inicial de la opción que buscamos, nos lleva directamente al primer término de la lista que empiece por dicha letra.

3.6 Nuevos *Layouts*

Swing incorpora nuevos gestores de impresión, ampliando los cinco que *AWT* incorporaba. Entre ellos conviene destacar los siguientes:

- `BoxLayout`: Es similar al `FlowLayout` de *AWT*, con la diferencia de que con él se pueden especificar los ejes (x o y). Viene incorporada en el componente `Box`, pero está disponible como una opción en otros componentes.
- `OverlayLayout`: Todos los componentes se añaden encima de cada componente previo.
- `SpringLayout`: El espacio se asigna en función de una serie de restricciones asociadas con cada componente.
- `ScrollPaneLayout`.
- `ViewportLayout`.

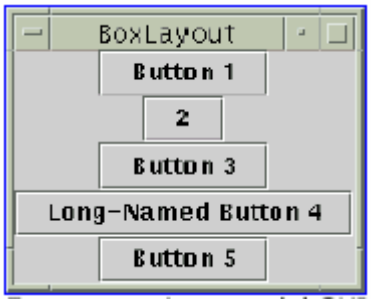

Figura 10. Un *Layout* de *Swing*, `BoxLayout`

3.7 *Look & Feel*

Otra característica que introduce *Swing* es que se puede especificar el Aspecto y Comportamiento (*Look & Feel* o *L&F*) que utilice el *GUI* de nuestro programa. Por el contrario, los componentes *AWT* siempre tienen el aspecto y comportamiento de la plataforma nativa.

Desde el código de la aplicación o *applet Swing* se puede exportar un *Look & Feel* diferente al nativo de la plataforma con el método `UIManager.setLookAndFeel`.

En las siguientes figuras podremos observar tres de los *L&F* de *Swing* en la misma aplicación.

Figura 11. *Java/Metal Look & Feel*

Figura 12. *CDE/Motif Look & Feel*

Figura 13. ***Windows Look & Feel***

3.8 Otras Nuevas Características

3.8.1 `Action`

Con objetos `Action`, el *API Swing* proporciona un soporte especial para compartir datos y estados entre dos o más componentes que pueden generar eventos `Action`. Por ejemplo, si tenemos un botón y un ítem de menú que realizan la misma función, podríamos considerar la utilización de un objeto Action para coordinar el texto, el icono y el estado de activado de los dos componentes.

3.8.2 Modelos de datos y estados separados

La mayoría de los componentes *Swing* no-contenedores tienen modelos. Por ejemplo, un botón (`JButton`) tiene un modelo (`ButtonModel`) que almacena el estado del botón -- cuál es su mnemónico de teclado, si está activado, seleccionado o pulsado, etc. Algunos componentes tienen múltiples modelos. Por ejemplo, una lista (`JList`) usa un `ListModel` que almacena los contenidos de la lista y un `ListSelectionModel` que sigue la pista de la selección actual de la lista.

Normalmente no se necesita conocer los modelos que usa un componente. Por ejemplo, casi todos los programas que usan botones tratan directamente con el objeto `JButton`, y no lo hacen en absoluto con el objeto `ButtonModel`.

Entonces ¿Por qué existen modelos separados? Porque ofrecen la posibilidad de trabajar con componentes más eficientemente y para compartir fácilmente datos y estados entre componentes. Un caso común es cuando un componente, como una lista o una tabla, contiene muchos datos. Puede ser mucho más rápido manejar los datos trabajando directamente con un modelo de datos que tener que esperar a cada petición de datos al modelo. Se puede usar el modelo por defecto del componente o implementar uno propio.

Las clases `Model` son una importante innovación que facilita la programación siguiendo la arquitectura MVC (modelo-vista-controlador) ya que separa para cada componente una clase de modelo de datos y otra de "interfaz", aunque con la funcionalidad de la "interfaz" se podría controlar todo totalmente sin ayuda del modelo de datos de ese componente.

3.8.3 Soporte para tecnologías asistivas

Las tecnologías asistivas para minusválidos como los lectores de pantallas pueden usar el *API* de accesibilidad para obtener información sobre los componentes *Swing*. Incluso sin hacer nada, un programa *Swing* probablemente funcionará correctamente con tecnologías asistivas, ya que el *API* de accesibilidad está construido internamente en los componentes *Swing*. Sin embargo, con un pequeño esfuerzo extra, se puede hacer que nuestro programa funcione todavía mejor con tecnologías asistivas, lo que podría expandir el mercado de nuestro programa.

4. Conclusiones

A lo largo de este informe se han comprobado las similitudes y diferencias entre ambos *API's* para la creación de interfaces gráficas. Básicamente, *Swing* viene a ser una ampliación y revisión de *AWT*. Ambas bibliotecas de clases comparten aspectos como el manejo de eventos, jerarquías similares de componentes, la representación de gráficas básicas e imágenes... Además, *Swing* añade nuevas funcionalidades a los antiguos componentes, añade nuevos componentes, introduce el concepto de *Look & Feel*, da soporte a el paradigma *MVC*; en resumen, aprovecha el camino recorrido por *AWT* respondiendo a sus deficiencias y a las nuevas necesidades.

Esto pone de manifiesto que las capacidades y respuestas que ofrece *Swing* son fruto de una suma de trabajos anteriores a él, lo que indica la importancia en la construcción de software de las buenas prácticas de la ingeniería del *software*, que facilitan conceptos tan útiles como la reutilización, la portabilidad entre plataformas, la escalabilidad... en resumen al *software* de calidad.

En resumen, *AWT* y *Swing* son un gran ejemplo de la buena puesta en marcha de las prácticas de la ingeniería del *software* y de la orientación a objeto.

5. Referencias

[1] "Java Swing" **http://jungla.dit.upm.es/~santiago/docencia/apuntes/Swing/**

[2] Sun "The Swing Tutorial" **http://java.sun.com/docs/books/tutorial/uiswing**

[3] Sun, Traductor : Juan Antonio Palos "Swing y JFC" **http://programacion.com/java/tutorial/swing/**

[4] Universidad de Navarra "Aprenda Java como si estuviera en primero" San Sebastián, Marzo 1999

[5] Universidad de Las Palmas de Gran Canaria "Tutorial de Java – AWT" **http://www.ulpgc.es/otros/tutoriales/java/Cap4/awt.html**