# Notas sobre la Construcción de Interficies Gráficas en Java

Material para la asignatura: Projectes de Programació

Enrique Romero, Alicia Ageno, Horacio Rodríguez
Departament de Computació
Universitat Politècnica de Catalunya

# Contents

# 1 Nociones Generales del AWT (Abstract Window Toolkit)

El AWT (Abstract Window Toolkit) es la parte de Java que se encarga de gestionar las interficies gráficas. El resto de gestores de interficies se basan o están muy relacionados con AWT.

Conceptos básicos del AWT:

- Componentes
  - Son los objetos de la interficie y sus contenedores
- Eventos
  - Cada vez que el usuario realiza una determinada acción, se produce el evento correspondiente, que el sistema operativo transmite al AWT
- Layout Managers
  - Establecen la distribución de los componentes en los contenedores

La mayoría de la información de esta sección se ha obtenido de [1], donde se puede encontrar una descripción mucho más detallada.

## 1.1 Componentes

Tipos de componentes (ver figura 1):

- Componentes primarios (widgets)
  - Texto no editable (Label)
  - Texto editable (TextArea, TextField)
  - Botones (Button)
  - Selección de una o varias opciones (Choice, Checkbox, List)
  - Barras de desplazamiento (Scrollbar)
  - Zonas de imágenes (Canvas)
- Componentes contenedores (también son componentes)
  - Contenedores principales: **Frame**, Dialog (subclases de Window)
    **El contenedor principal más importante es el Frame**
    Los Dialog se suelen usar para llamar la atención temporalmente aparte de la ventana principal (por ejemplo, mostrar un mensaje de error o de aviso, cosas del estilo “Ok”, “Yes/No”, “Yes/No/Cancel”,...), y pueden bloquear el resto de ventanas (Dialog modal, opción por defecto) o no

Un Dialog depende de un Frame: al crear un Dialog, hay que decir de qué Frame depende, y al cerrar/minimizar el Frame también se cierran/desaparecen los Dialog asociados

– Contenedores secundarios: **Panel**, ScrollPane (necesitan estar dentro de otro contenedor)
**El contenedor secundario más importante es el Panel**
Un **Applet** es una subclase de Panel

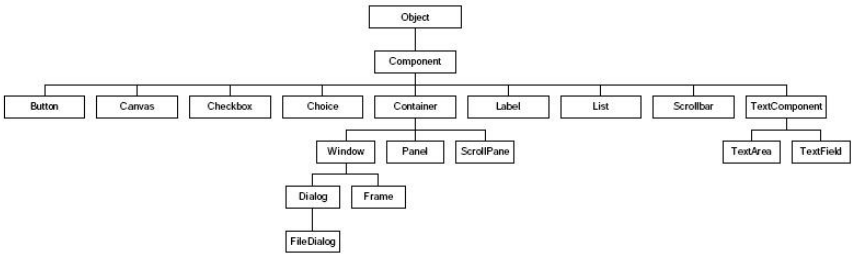


Figure 1: AWT: Componentes

Reglas básicas de los contenedores:

- Un contenedor principal no puede estar dentro de otro contenedor, ni principal ni secundario
- Un contenedor secundario puede contener a otro/s contenedor/es secundario/s
- Un componente sólo puede estar en un contenedor: si se añade a un segundo contenedor, deja de estar en el primero

Reglas básicas de funcionamiento:

- Para tener una interficie funcional, hay que tener un contenedor principal, y hacerlo visible: *miContainer.setVisible(true)*.
- Un contenedor principal siempre es funcional
- Para ser funcionales, cada componente (incluidos los contenedores secundarios, no los contenedores principales) tiene que estar en un contenedor funcional
- Para añadir un componente a un contenedor: *miContainer.add(miComponent)*
- Para eliminar un componente de un contenedor: *miContainer.remove(miComponent)*

**En resumen, hay que formar un árbol n-ario, en cuya raíz hay un Frame (los Applet van aparte), en los nodos intermedios hay contenedores secundarios y en las hojas hay componentes primarios (aunque también podría haber contenedores secundarios, pero no tendría ningún efecto).**

## 1.2 Eventos

### 1.2.1 Funcionamiento General y Maneras de Implementarlo

Después de que al AWT recibe el evento transmitido desde el sistema operativo, el AWT crea un objeto de una subclase de AWTEvent. Este objeto tiene que ser transmitido a un determinado método para que lo gestione. Este método está definido en una interface Listener, que hay que asociar al componente.

Dicho de otro modo:

- Hay que asociar al componente C un objeto L que implementa una interface Listener (dependiendo del evento, será de una interface u otra)
  **Es posible asociar varios Listener al mismo componente para el mismo evento**
- El objeto L tiene que definir un método M que recibe como parámetro un evento E, que es un objeto subclase de AWTEvent (dependiendo del evento, será de una subclase u otra)
- Dentro del método M es donde hay que poner el código a ejecutar cuando se produce el evento

**Ejemplo:** Supongamos que tenemos un botón tal que al clickar en él queremos salir del programa. El evento asociado al click de un botón es un objeto de la clase ActionEvent, el listener donde hay que tratarlo es un objeto de la interface ActionListener, y el método que lo gestiona se llama actionPerformed (y tiene que ser declarado como *public*). Para asociar el objeto de la clase ActionListener al botón se usa el método addActionListener. De manera que tendremos que tener algo así:

```
public class MiClase() {
  // En alguna parte de la clase (al crearla, por ejemplo)
  public MiClase() {
    ...
    // Crea el listener del evento
    ListenerBoton miListenerBoton = new ListenerBoton();
    // Asocia el listener al boton
    miBoton.addActionListener(miListenerBoton);
    ...
  }
  ...
  // La clase ListenerBoton es privada
  private class ListenerBoton implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      System.exit(0); // CODIGO ASOCIADO AL EVENTO
    }
  }
}
```

Relacionándolo con lo anterior:

- C = miBoton
- L = miListenerBoton, que es un objeto de la interface ActionListener
- M = actionPerformed
- E = event, que es un objeto de la clase ActionEvent

Una manera equivalente de hacer lo mismo sin definir clases privadas es usar clases anónimas :

```
public class MiClase() {
  // En alguna parte de la clase (al crearla, por ejemplo)
  public MiClase() {
    ...
    // Crea y asocia el listener al boton
    miBoton.addActionListener
      (new ActionListener() {
         public void actionPerformed (ActionEvent event) {
           System.exit(0); // CODIGO ASOCIADO AL EVENTO
         }
       });
    ...
  }
  ...
}
```

Y si la clase principal ya implementa la interface ActionListener, entonces basta con definir el método *actionPerformed* en la clase: :

```
public class MiClase() implements ActionListener {
  // En alguna parte de la clase (al crearla, por ejemplo)
  public MiClase() {
    ...
    // Asocia el listener al boton
    miBoton.addActionListener(this);
    ...
  }
  ...
  public void actionPerformed (ActionEvent event) {
    System.exit(0); // CODIGO ASOCIADO AL EVENTO
  }
  ...
}
```

### 1.2.2 Tipos de Eventos

Hay dos tipos de eventos:

- De bajo nivel: son los que se producen con las operaciones elementales con el ratón, teclado, contenedores y ventanas, como por ejemplo:
  - Pulsar teclas
  - Mover, pulsar, arrastrar y soltar el ratón
  - Recuperar o perder el foco
  - Operaciones con ventanas: cerrar, maximizar, minimizar,...
  - ...
- De alto nivel: son los que tienen un significado en sí mismo (en el contexto gráfico):
  - Clickar en un botón
  - Elegir un valor entre varios
  - Cambiar un texto
  - Cambiar las barras de desplazamiento

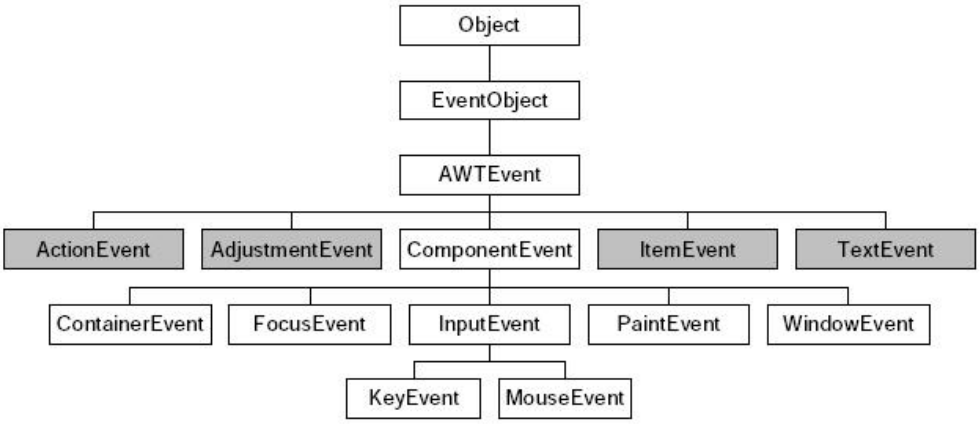


Figure 2: AWT: Eventos

La jerarquía de los eventos de AWT está en la figura 2 (los eventos de alto nivel están sombreados en gris).

### 1.2.3 Relación entre Componentes y Eventos

En la figura 3 se muestra la relación entre los componentes del AWT y los eventos **específicos** que se pueden capturar, junto con una breve explicación. En la figura 4 se muestra la relación de **todos** los eventos que pueden ser generados por los diferentes componentes, teniendo en cuenta las superclases.

| 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 (tener en cuenta que este evento tiene dos Listener) |
| 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 |
| MunuItem | 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 |

Figure 3: AWT: Relación entre componentes y eventos específicos

| AWT Components | Eventos que se pueden generar | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | AtionEvent | AdjustementEvent | ComponentEvent | ContainerEvent | FocusEvent | ItemEvent | KeyEvent | MouseEvent | TextEvent | WindowEvent |
| Button | ✓ | | ✓ | | ✓ | | ✓ | ✓ | | |
| Canvas | | | ✓ | | ✓ | | ✓ | ✓ | | |
| Checkbox | | | ✓ | | ✓ | ✓ | ✓ | ✓ | | |
| Checkbox-MenuItem | | | | | | ✓ | | | | |
| Choice | | | ✓ | | ✓ | ✓ | ✓ | ✓ | | |
| Component | | | ✓ | | ✓ | | ✓ | ✓ | | |
| Container | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| Dialog | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ |
| Frame | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ |
| Label | | | ✓ | | ✓ | | ✓ | ✓ | | |
| List | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ | | |
| MenuItem | ✓ | | | | | | | | | |
| Panel | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| Scrollbar | | ✓ | ✓ | | ✓ | | ✓ | ✓ | | |
| TextArea | | | ✓ | | ✓ | | ✓ | ✓ | ✓ | |
| TextField | ✓ | | ✓ | | ✓ | | ✓ | ✓ | ✓ | |
| Window | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ |

Figure 4: AWT: Relación entre componentes y eventos

### 1.2.4 Relación entre Eventos y Listeners

Normalmente, cada evento tiene asociado una interface Listener, excepto el ratón que tiene dos interfaces: MouseListener y MouseMotionListener. El motivo es por razones de eficiencia: los eventos de movimiento del ratón se producen con muchísima frecuencia, y son gestionados por una interface especial.

**Nomenclatura:** A un evento de tipo XxxxxxEvent **siempre** le corresponde un listener de nombre XxxxxxListener.

Cada Listener es una interface. Eso quiere decir que cada vez que usa un Listener hay que implementar obligatoriamente los métodos definidos en ella. En la figura 5 se muestra la relación de eventos, interfaces Listener y métodos que hay en cada una de estas interfaces.

| 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() |

Figure 5: AWT: Relación entre eventos, Listeners y métodos de los Listeners

### 1.2.5 Asociar un Listener a un Componente

Para asociar un Listener de la interface XxxxxxListener a un componente (para gestionar un evento de tipo XxxxxxEvent), siempre se hace de la misma manera, con el método *addXxxxxxListener*:

*miComponente.addXxxxxxListener(objetoXxxxxxListener);*

Para desasociar un Listener:
*miComponente.removeXxxxxxListener(objetoXxxxxxListener);*

**IMPORTANTE: Es posible asociar varios Listener al mismo componente para el mismo evento.** En este caso, se ejecutará el código asociado a los dos listeners en el

orden inverso al que se han añadido.

Ejemplos:

- *miBoton.addActionListener(objetoActionListener);*
- *miPanel.addKeyListener(objetoKeyListener);*
- *miTextArea.addTextListener(objetoTextListener);*
- *miFrame.addWindowListener(objetoWindowListener);*

### 1.2.6 Las Clases Adapter

Las clases Adapter son ayudas proporcionadas por el lenguaje para no tener que definir todos los métodos declarados en las interfaces Listener. Hay una clase Adapter para cada una de las interfaces Listener que contienen más de un método, y son clases que contienen definiciones vacías para todos los métodos de la interface. De manera que, en vez de crear una clase que implemente la interface Listener, basta crear una clase que derive de la clase Adapter correspondiente y sólo redefina los métodos necesarios.

## 1.3 Layout Managers

Un layout manager es un objeto que controla cómo se distribuyen los componentes en los contenedores (tanto principales como secundarios).

Se pueden definir layout managers personalizados. Sin embargo, lo normal es usar uno de los 5 layout managers predefinidos:

- FlowLayout: los componentes se van añadiendo de izquierda a derecha y de arriba a abajo
- BorderLayout: los componentes se colocan en 5 zonas: North, South, East, West y Center
- GridLayout: los componentes se pueden colocar en una matriz de celdas del mismo tamaño
- GridBagLayout: similar en esencia al GridLayout, pero hay componentes que puede ocupar más de una celda
- CardLayout: se permite que el mismo espacio sea utilizado en diferentes momentos por elementos diferentes

Todos los contenedores tienen un layout manager por defecto:

- Contenedores principales (Frame y Dialog): BorderLayout
- Contenedores secundarios (Panel y ScrollPane): FlowLayout

Más cosas sobre los layout managers:

- Para cambiar el layout por defecto (a GridLayout, por ejemplo):
  *miContainer.setLayout(new GridLayout());*

## 1.4 Menús

A un contenedor principal se le pueden asociar menús en forma de "barra de menús". Para ello hay tres clases básicas (la jerarquía de clases para las clases asociadas a los menús se puede ver en la figura 6):

- MenuItem: Es la clase cuyos objetos acabarán ejecutando la acción correspondiente
- Menu: Agrupa varios MenuItem u otros Menu
- MenuBar: Es la barra de menús

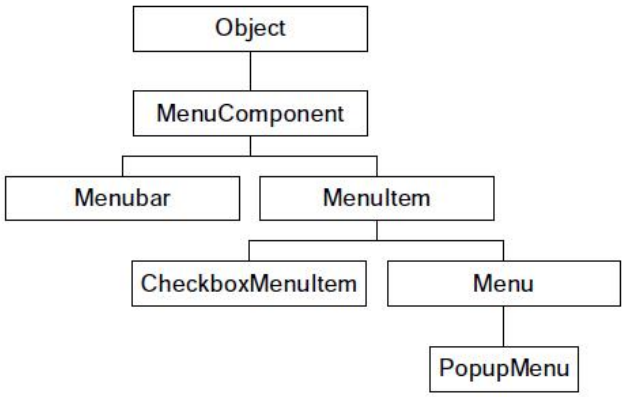


Figure 6: AWT: Menús

Para tener una barra de menús funcional, hay que :

- Crear los MenuItem: *MenuItem miMenuItem = new MenuItem("Menu Option");*
- Crear los Menu: *Menu miMenu = new Menu("Menu Name");*
- Crear la MenuBar: *MenuBar miMenuBar = new MenuBar();*
- Asociar los MenuItem a los Menu: *miMenu.add(miMenuItem);*
- Asociar los Menu a la MenuBar: *miMenuBar.add(miMenu);*

- Asociar la MenuBar al contenedor principal: *miFrame.setMenuBar(miMenuBar);*
- Asociar un ActionListener a cada MenuItem:

```
miMenuItem.addActionListener
   (new ActionListener() {
      public void actionPerformed (ActionEvent event) {
        ... // CODIGO ASOCIADO AL EVENTO
      }
    });
```

La clase CheckboxMenuItem son ítems de un Menu que puedes estar activados o no. No generan un ActionEvent, sino un ItemEvent, de manera similar a la clase Chekbox. Los objetos de la clase PopupMenu son menús que aparecen en cualquier parte de la pantalla al hacer click con el botón derecho del ratón.

## 1.5 Threads y el Event-dispatching Thread

Java funciona con threads. Los threads son flujos secuenciales de ejecución dentro de un proceso. Un único proceso puede tener varios threads ejecutándose "a la vez" (donde "a la vez" significa que la CPU se reparte entre todos los threads activos).

### 1.5.1 Threads

En general, las diferentes partes de un programa se pueden ejecutar en los siguientes tipos de threads:

- El *thread inicial*: es el thread que inicia la ejecución de la aplicación
- El *event-dispatching thread*: es el thread donde, en general, se procesan las tareas gráficas del AWT una vez los componentes están activos
- Los *worker threads* o *background threads*: pensados para encargarse de tareas costosas en segundo plano

Para sincronizar threads se usan los métodos:

- *wait(...)*: bloquea el objeto al que se aplica el método, deteniendo la ejecución del thread en el que está hasta que se llame a *notify()* o *notifyAll()*, o finalice el tiempo indicado en su argumento
- *notify()*: desbloquea el objeto, lanzando una señal que indica al sistema que puede activar uno de los threads que estaban esperando a que se desbloquease el objeto
- *notifyAll()*: desbloquea el objeto, activando todos los threads que estaban esperando

Las llamadas a los métodos *wait()*, *notify()* y *notifyAll()* deben estar incluidas en métodos *synchronized*:

```
public synchronized void metodoEsperar (...) {
  ...
  miObjeto.wait();
  ...
}

public synchronized void metodoContinuar (...) {
  ...
  miObjeto.notify();
  ...
}
```

Para sincronizar a nivel de clase (hacer que desde método se bloqueen todos los objetos creados de una clase), hay que declarar este método como *synchronized static*.

**Desgraciadamente**, los métodos *wait()*, *notify()* y *notifyAll()* no son muy útiles para sincronizar los diferentes componentes gráficos, porque la ejecución del método *wait()* en un objeto gráfico bloquea el *event-dispatching thread* (ver sección 1.5.2), y por tanto bloquea la ejecución del resto de componentes gráficos. El uso típico de *wait()* es para esperar a que se cumpla alguna determinada condición **ajena** al propio thread.

### 1.5.2 El Event-dispatching Thread

En una aplicación con interficie gráfica, el thread inicial no tiene mucho que hacer: su principal tarea es, aparte de las inicializaciones necesarias, construir la vista principal y hacerla visible:

```
public static void main (String args[]) {
  vistaPrincipal = new VistaPrincipal();
  ... // Inicializaciones
  vistaPrincipal.hacerVisible();
  System.out.println
    ("Seguramente no tiene sentido poner nada aqui...");
}
```

En un programa principal como el de arriba, tanto el constructor como *vistaPrincipal.hacerVisible()* se ejecutan en el thread inicial. Aunque se puede hacer, seguramente no tiene sentido poner nada después de *vistaPrincipal.hacerVisible()*.

Después de hacer visible vistaPrincipal, el thread inicial termina. ¿Cómo continua el programa? Después de hacer visible la vista, son los Listeners asociados a los componentes los que toman el control: cada vez que se genere un evento se ejecutará el

Listener asociado, y esa ejecución se hará, por defecto, en el ***event-dispatching thread***. Dentro del *event-dispatching thread* se pueden crear uno o más *worker threads* para hacer tareas que consuman mucho tiempo de CPU sin dejar a la parte gráfica bloqueada esperando a que acabe.

Alternativamente (y mejor, porque así TODAS las tareas gráficas se ejecutan en el *event-dispatching thread*), se puede hacer creando un objeto de la interface Runnable:

```
public static void main (String args[]) {
  java.awt.EventQueue.invokeLater (
    new Runnable() {
      public void run() {
        vistaPrincipal = new VistaPrincipal();
        ... // Inicializaciones
        vistaPrincipal.hacerVisible();
        System.out.println
          ("Seguramente no tiene sentido poner nada aqui...");
      }
    }
  );
}
```

El método *invokeLater()* hace que *run()* se ejecute de manera asíncrona en el *event-dispatching thread*, después de que los eventos pendientes hayan sido procesados.

Para ver si algo se está ejecutando en el *event-dispatching thread* se puede llamar al método *EventQueue.isDispatchThread()*.

## 1.6 Imports Necesarios para AWT

Para que AWT funcione basta con importar *java.awt.** y *java.awt.event.**.

# 2 Swing

Es la extensión más habitual de AWT, presente desde la versión 1.2 de Java. Swing ofrece una mayor variedad de componentes y eventos, nuevas funcionalidades y una serie de ventajas sobre AWT. Además, entornos de desarrollo como Netbeans incorporan editores gráficos basados en Swing (ver sección 5).

Una alternativa a Swing es SWT (Standard Widget Toolkit), originalmente desarrollado por IBM y mantenido por la comunidad de Eclipse. SWT no es una extensión de AWT, sino que ha sido desarrollado desde cero. SWT es más simple que Swing, más limpio en cuanto a su diseño y más rápido (su rendimiento en plataformas diferentes a Windows es mucho peor), pero también es menos potente, y comparte algunos de los problemas de portabilidad de AWT. No está integrado en la instalación standard de Java, y no está disponible en todas las plataformas que soportan Java.

## 2.1 Diferencias Generales entre Swing y AWT

A nivel interno:

- AWT se apoya en los componentes proporcionados por la plataforma (*peers*), llamados comúnmente componentes pesados ("heavyweight" components), que
  - Consume muchos recursos del sistema operativo
  - No funcionan de manera exactamente igual en cada plataforma, lo que da problemas de portabilidad
  - Su *Look & Feel* está ligado a la plataforma (se ve diferente en cada plataforma), y no puede cambiarse

  En cambio, Swing está escrito completamente en Java y no depende tanto de los *peers* (los contenedores principales en Swing son todavía pesados, el resto no), lo que ofrece
  - Consume menos recursos del sistema operativo
  - Mejor y mayor portabilidad
  - El *Look & Feel* puede hacerse independiente de la plataforma (para que se vea igual en todas las plataformas) o no, y se puede seleccionar
- En Swing hay componentes intermedios (de apoyo) que el sistema crea automáticamente, y que oscurecen la interpretación de lo que está pasando (por ejemplo, todos los contenedores que se crean al crear un JFrame: *rootPane*, *contentPane*, *layeredPane*, *glassPane*)

A nivel de desarrollo:

- Implementar AWT en un sistema operativo nuevo es complejo, puesto que hay que conectar los *peers* con los componentes

- La mayoría de desarrolladores de software crean nuevos componentes basados en Swing, no en AWT

A nivel de funcionalidades:

- Había un tiempo en que los applets hechos con AWT funcionaban sin problemas en la mayoría de navegadores pero los de Swing no (algunos navegadores no incluían las clases de Swing), pero esto ya no es así (al menos para los navegadores más habituales)
- Swing ofrece una mucho mayor gama de componentes que AWT:
  - JTabbedPane, un contenedor con pestañas
  - JTree, para representar árboles
  - JComboBox, flexibiliza las "listas seleccionables"
  - JTable, para representar tablas
  - JProgessBar, para mostrar el progreso de la ejecución
  - ...
- Swing amplia los métodos y eventos disponibles en AWT
- Swing ofrece o amplia funcionalidades que AWT no tiene o son poco "usables":
  - Asociar iconos a componentes
  - Tooltips (sugerencias o comentarios)
  - Toolbars (barras con botones)
  - *Look & Feel* parametrizable
  - Key bindings: asociar teclas a acciones de componentes (por ejemplo, hacer la barra espaciadora equivalente a hacer click en el botón)
  - Transferencia de datos entre componentes, via cut, copy, paste, y drag & drop
  - Funcionalidades de undo y redo sobre componentes
  - Campos de password
  - Funcionalidades relacionadas con texto formateado
  - Posibilidad de trabajar en idiomas con miles de caracteres, tales como japonés, chino o coreano
  - Adaptación a las particularidades locales del idioma (texto de derecha a izquierda o de izquierda a derecha,...)
  - Accesibilidad para personas discapacitadas
  - ...

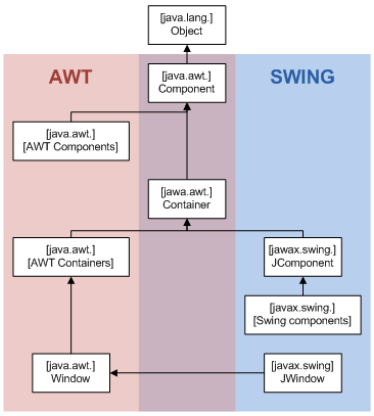


Figure 7: AWT vs Swing: Jerarquías de clases

## 2.2 La Jerarquía de Clases de Swing

Swing se integra/complementa con AWT de la siguiente manera (ver figura 7):

- Los componentes primarios y los contenedores secundarios se agrupan en una clase JComponent, que hereda de Container (AWT), que a su vez hereda de Component. Así pues, JComponent contiene
  - Componentes primarios: JButton, JMenuItem, JComboBox, JFileChooser, JLabel, JList, JMenuBar, JScrollBar, JTextArea, JTextField, etc (no son los mismos que en AWT)
  - Contenedores secundarios: JPanel, JScrollPane, JOptionPane,...
- Los contenedores principales (JFrame y JDialog) heredan directamente de sus equivalentes en AWT:
  - JFrame hereda de Component.Container.Window.Frame (AWT)
  - JDialog hereda de Component.Container.Window.Dialog (AWT)

  La relación entre JFrame y JDialog es la misma que en AWT (ver sección 1.1). Los diálogos más interesantes se pueden construir con
  - JOptionPane (método *createDialog(...)*, por ejemplo), para establecer "diálogos rápidos" (del estilo "Ok", "Yes/No", "Yes/No/Cancel",...)
  - JFileChooser (método *showOpenDialog(...)*, por ejemplo), para seleccionar un fichero
  - JColorChoser (método *showDialog(...)*, por ejemplo), para seleccionar un color

El hecho de que los componentes primarios sean también contenedores parece, en principio, anti-intuitivo, pero flexibiliza el diseño de la interficie: permite, por ejemplo, añadir un icono a un botón .

Hay clases entrelazadas que provocan cierta confusión. Por ejemplo, la manera más sencilla de construir diálogos es usando métodos de otros JComponent (JOptionPane, JFileChooser,..., ver más arriba) en vez de usar directamente la clase JDialog.

## 2.3 Similitudes entre Swing y AWT

Las reglas básicas de los contenedores y su funcionamiento no cambian: **hay que formar un árbol n-ario, en cuya raíz hay un JFrame (los JApplet van aparte), en los nodos intermedios hay contenedores secundarios y en las hojas hay componentes primarios (aunque también podría haber contenedores secundarios, pero no tendría ningún efecto).**

Similitudes a nivel de implementación:

- Por regla general, las clases de Swing "equivalentes" a las de AWT se llaman igual, pero con una J delante: JFrame, JPanel, JButton, JLabel,... teniendo en cuenta que
  **No todos los componentes están extendidos: para algunos hay componentes "equivalentes" (Choice), para otros no (Canvas) y para otros los constructores son diferentes (Checkbox, Scrollbar, List)**

- Muchos constructores en Swing tienen los mismos argumentos que en AWT, **pero no todos**

- La gestión de eventos (ver sección 1.2) es igual, teniendo en cuenta que

  - Hay nuevos eventos (importar *javax.swing.event.**),...
  - Hay eventos que cambian (por ejemplo, los asociados a componentes de texto (JTextArea, JTextfield), que pasan de generar un evento de clase TextListener a un evento de clase DocumentListener)

  Más información en

  - `http://download.oracle.com/javase/tutorial/uiswing/events/index.html`
  - `http://download.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html`

- El significado y funcionamiento de los layout managers (ver sección 1.3) es prácticamente igual, teniendo en cuenta que hay nuevos layout managers:

  - BoxLayout: coloca los componentes en una única fila o columna, respetando los tamaños máximos y permitiendo alinearlos (es como el FlowLayout, pero más completo)
  - GroupLayout: es muy flexible, puesto que define los componentes de cada dimensión de manera independiente
  - SpringLayout: permite añadir condiciones a la colocación de los componentes

  Más información en `http://download.oracle.com/javase/tutorial/uiswing/layout/index.html`

- El significado y funcionamiento de los menús (ver sección 1.4) es igual
- El significado y funcionamiento de los threads (ver sección 1.5.1) es igual
- Con respecto al *event-dispatching thread* (ver sección 1.5.2), hay alguna diferencia (más información en [8]):
  - Todos los métodos de AWT son *thread-safe* (pueden usarse en entornos multi-thread sin ocasionar problemas no deseados, por ejemplo, que la información del componente no aparezca actualizada, aunque la instrucción de actualización se haya lanzado antes que la de pintado), pero la mayoría de los métodos de Swing no los son
  - Para evitar problemas no deseados con Swing, la regla es: **"Una vez que un componente está activo, todo el código que le afecte o dependa de él debería ejecutarse en el *event-dispatching thread*"**. Dicho de otro modo: "El *event-dispatching thread* es el único thread válido para modificar los estados de las componentes gráficas una vez están activas"
  - Normalmente no habrá que preocuparse de cumplir esta regla, porque es lo que ocurre por defecto (por ejemplo, al ejecutar métodos como *paint()* o *actionPerformed()*). De hecho, de lo único que nos tendremos que preocupar es de que los threads que creemos una vez la interficie gráfica está activa no modifiquen ningún componente gráfico
  - Para ver si algo se está ejecutando en el *event-dispatching thread* se puede llamar al método *SwingUtilities.isEventDispatchThread()*

## 2.4 Diferencias Específicas entre Swing y AWT

Diferencias a nivel de implementación:

- Por defecto, al intentar cerrar un Frame (click en la X) en AWT no se hace nada: si se quiere salir hay que asociar un evento. En Swing no hay que asociar un evento al cerrar un JFrame, basta con asignar la operación por defecto:
  *miJFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);*
  Otras opciones disponibles: DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE

- La clase JComponent incorpora nuevos métodos que pueden ser útiles: *setMaximumSize*, *setMinimumSize*, *setPreferredSize*
- Al crear un JFrame, se crean automáticamente una serie de contenedores (más informacion en [5]):
  - *rootPane*, que contiene al resto de contenedores
  - *layeredPane*, que sirve para colocar los contenidos: *contentPane* y la barra de menús
  - *contentPane*, que contiene los componentes visibles (excepto la barra de menús), y que es donde se añaden todos los componentes al hacer un *add(...)*. El *contentPane*:
    - Se obtiene mediante el método *getContentPane*, al cuál habrá que hacer un cast al contenedor especifico (por defecto es de la clase Container): *JPanel contentPane = (JPanel) miJFrame.getContentPane();*
    - Su layout por defecto es BorderLayout (el layout por defecto de los JPanel es FlowLayout)
  - *glassPane*, útil cuando se quieren capturar eventos de áreas que contienen uno o más componentes
- Los componentes primarios no se suelen añadir directamente al JFrame, sino que se añaden al *contentPane*
  **No hay nada que impida asociar los componentes al JFrame directamente y definir un layout para el JFrame, como en AWT, pero realmente se están añadiendo al *contentPane***. Por ejemplo,

```
JPanel contentPane = (JPanel) myJFrame.getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(miBoton);
```

  es equivalente a

```
myFrame.setLayout(new FlowLayout());
myFrame.add(miBoton);
```

## 2.5 Imports Necesarios para Swing

Además de las clases de AWT (*java.awt.*;* y *java.awt.event.*;*) hay que importar:

- *javax.swing.**
- *javax.swing.event.** (si se gestionan eventos exclusivos de Swing)

# 3 Implementación Típica (con Swing)

## 3.1 Ideas Generales

Antes de empezar, es muy importante diseñar muy bien el layout que se quiere usar, y la estructura jerárquica de los componentes.

En la practica, lo habitual es tener un árbol de contenedores y componentes del estilo:

- Un JFrame principal
- Una barra de menús, contenida en el JFrame
- Uno o varios JPanel, contenidos en el JFrame
- Varios componentes primarios u otros JPanel, contenidos en cada JPanel

En el caso de que haya varios JPanel, A nivel de visibilidad podemos querer

- Tener todos los JPanel siempre visibles (algunos quizá desactivados), es decir, ocupando diferentes posiciones de la pantalla. En este caso tendremos que añadir todos los JPanel al *contentPane*
- Tener sólo un subconjunto (típicamente uno) visible cada vez, e ir intercambiando el JPanel visible entre los diferentes JPanels. En este caso tendremos que ir modificando el *contentPane* con el JPanel correspondiente (con el método *miJFrame.setContentPane(miJPanel)*)

En resumen, hay que diseñar layouts a varios niveles:

- A nivel JPanel, para colocar las componentes en el JPanel
- A nivel JFrame, para colocar los JPanel en el JFrame

## 3.2 Una Implementación del Main, el Controlador de Presentación y la Vista Principal

Para iniciar la ejecución:

- El *main()* crea un objeto de la clase CtrlPresentacion, e inicia la presentación, todo en el *event-dispatching thread* (la implementación que hace Netbeans es ligeramente diferente, ver sección 5):

```
public class MiPrograma {
  public static void main (String[] args) {
    javax.swing.SwingUtilities.invokeLater (
```

```
      new Runnable() {
        public void run() {
          CtrlPresentacion ctrlPresentacion =
            new CtrlPresentacion();
          ctrlPresentacion.inicializarPresentacion();
    }});
  }
}
```

Alternativamente, el método *inicializarPresentacion()*, se podría ejecutar en el constructor de *ctrlPresentacion*.

- El controlador de presentación CtrlPresentacion típicamente crea una instancia del controlador de dominio y de la vista principal en su constructor, y hace visible la vista en inicializarPresentacion (quizá haya que pasar cosas por parámetro)

```
public class CtrlPresentacion {

  private CtrlDominio ctrlDominio;
  private VistaPrincipal vistaPrincipal;

  public CtrlPresentacion() {
    ctrlDominio = new CtrlDominio();
    vistaPrincipal = new VistaPrincipal();
  }

  public void inicializarPresentacion() {
    vistaPrincipal.inicializarComponentes();
    vistaPrincipal.hacerVisible();
  }
  ...
}
```

Alternativamente, el método *inicializarComponentes()*, se podría ejecutar en el constructor de *vistaPrincipal* (es lo que implementa Netbeans por defecto, ver sección 5).

- La vista principal VistaPrincipal contiene un atributo privado JFrame y atributos privados para todos los componentes (los atributos de los componentes podrían ser variables locales, pero no es habitual). Una alternativa igualmente válida sería que la vista principal heredase de JFrame (es lo que implementa Netbeans por defecto, ver sección 5), aunque
  - Si no se hace se permite que la vista herede de cualquier otra clase (esto es, en principio, una ventaja)
  - Si se hace, sólo hay que cambiar todas las ocurrencias de *miJFrame.metodo(...)* por *this.metodo(...)*

- El método *inicializarComponentes()* se encarga de:
  - Acabar de definir las características del JFrame (operación por defecto al cerrar, tamaño,...) y de los componentes (tamaños, literales, colores, opciones posibles,...)
  - Definir y asociar los layout managers
  - Añadir los componentes a sus contenedores correspondientes
  - Definir y asociar los Listeners a los componentes
  - Asignar el "primer panel visible" a *contentPane* del JFrame (en caso de que haya varios, si no se pueden asociar los componentes directamente al *contentPane*)
  - (Recomendado) Posicionar el frame en la pantalla: *miJFrame.setLocation(...)* o *miJFrame.setLocationRelativeTo(...)*

  **En principio, el orden en el que se añaden los componentes a los contenedores, se asocian los layout managers a los contenedores y se asocian los Listeners a los componentes es irrelevante.**

- El método *hacerVisible()* se encarga de:
  - (Muy recomendado) Colocar automáticamente los contenidos: *miJFrame.pack()*
  - Hacer visible el JFrame: *miJFrame.setVisible(true)*

**A partir de aquí, son los Listeners los que controlan el resto de la aplicación.**

## 3.3 Cosas Importantes a Tener en Cuenta en la Implementación

- El método *pack()* redimensiona el JFrame al tamaño predeterminado y coloca los componentes según su layout manager. Si no se ejecuta el método *pack()*, los componentes pueden verse en posiciones incorrectas (o incluso no verse)
- A veces, además de ejecutar el método *pack()*, hay que hacer un *repaint()*
- Excepto para un Dialog modal, después de hacer visible un contenedor principal, la ejecución sigue. **Es lo normal**, puesto que es igual que al ejecutar cualquier otro método. Si se quiere sincronización hay que usar los métodos *wait()* y *notify()* (ver sección 1.5.1)
- Dentro de un Listener (tanto para AWT como para Swing) es posible obtener el objeto que ha generado el evento con el método *getSource()*. Por ejemplo, en el caso de un botón tendríamos:

```
private class ListenerBoton implements ActionListener {
  public void actionPerformed (ActionEvent event) {
    JButton miEventBoton = (JButton) event.getSource();
```

```
    if (miEventBoton == miBoton1) {
      ...
    }
    if (miEventBoton.getText().equals("...")) {
      ...
    }
  }
}
```

También es posible asignarle (vía *setActionCommand(...")*) una etiqueta al componente para poder recuperarla en el Listener (vía *getActionCommand*):

```
// Al definir las propiedades del componente...
miBoton.setActionCommand("VPresentacion-Boton-1");
...
ListenerBoton miListenerBoton = new ListenerBoton();
miBoton.addActionListener(miListenerBoton);
...
private class ListenerBoton implements ActionListener {
  public void actionPerformed (ActionEvent event) {
    String actionCommand = event.getActionCommand();
    if (actionCommand.equals("VPresentacion-Boton-1")) {
      ...
    }
  }
}
```

- **HAY COSAS QUE SE PUEDEN HACER DE MUCHAS MANERAS**.
  Aparte de las que ya se han visto (eventos con clases anónimas en al sección 1.2, que la vista principal herede de JFrame o no en la sección 3.2,...):
  - Usar el *contentPane* creado automáticamente por el sistema

    ```
    JPanel contentPane = (JPanel) myJFrame.getContentPane();
    contentPane.add(...);
    ... // equivalente a
    myJFrame.getContentPane().add()...;
    ```

    o usar nuestro propio JPanel como *contentPane*

    ```
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.add(...);
    myJFrame.setContentPane(contentPane);
    ```

  - **Hay muchos más ejemplos...**

# 4 Un Ejemplo “Completo”

Siguiendo las ideas comentadas en la sección 3, en esta sección se describe un ejemplo “completo”, que consta de:

- MainEjemplo: La clase que contiene el *main()*
- CtrlPresentacion: El controlador de presentación
- CtrlDominio: El controlador de dominio
- VistaPrincipal: Una vista principal
- VistaSecundaria: Una vista secundaria
- VistaDialogo: Una vista de diálogo modal

**Este ejemplo se ha creado desde cero, sin usar Netbeans ni ningún otro entorno de desarrollo.**

## 4.1 La Clase MainEjemplo

El *main()* crea una instancia del controlador de presentación e inicializa la presentacion ejecutando el método *inicializarPresentacion()* del controlador de presentación:

```
public class MainEjemplo {
  public static void main (String[] args) {
    javax.swing.SwingUtilities.invokeLater (
      new Runnable() {
        public void run() {
          CtrlPresentacion ctrlPresentacion = new CtrlPresentacion();
          ctrlPresentacion.inicializarPresentacion();
    }});
  }
}
```

## 4.2 La Clase CtrlPresentacion

El controlador de presentación:

- Crea una instancia del controlador de dominio y de la vista principal en su constructor
- Inicializa el controlador de dominio y hace visible la vista principal en *inicializarPresentacion()*
- Contiene los métodos para la sincronización entre las vistas
- Contiene los métodos para llamar al controlador de dominio

```
import java.util.*;

public class CtrlPresentacion {

  private CtrlDominio ctrlDominio;
  private VistaPrincipal vistaPrincipal = null;
  private VistaSecundaria vistaSecundaria = null;

  // Constructor y metodos de inicializacion

  public CtrlPresentacion() {
    ctrlDominio = new CtrlDominio();
    if (vistaPrincipal == null)
      vistaPrincipal = new VistaPrincipal(this);
  }

  public void inicializarPresentacion() {
    ctrlDominio.inicializarCtrlDominio();
    vistaPrincipal.hacerVisible();
  }

  // Metodos de sincronizacion entre vistas

  public void sincronizacionVistaPrincipal_a_Secundaria() {
    vistaPrincipal.desactivar();
    // Solo se crea una vista secundaria (podria crearse una nueva cada vez)
    if (vistaSecundaria == null)
      vistaSecundaria = new VistaSecundaria(this);
    vistaSecundaria.hacerVisible();
  }

  public void sincronizacionVistaSecundaria_a_Principal() {
    // Se hace invisible la vista secundaria (podria anularse)
    vistaSecundaria.hacerInvisible();
    vistaPrincipal.activar();
  }

  // Llamadas al controlador de dominio

  public ArrayList<String> llamadaDominio1 (String selectedItem) {
    return ctrlDominio.llamadaDominio1(selectedItem);
  }

  public ArrayList<String> llamadaDominio2() {
    return ctrlDominio.llamadaDominio2();
  }
}
```

Una alternativa a pasar por parámetro el controlador de presentación a las vistas sería pasar por parámetro los Listeners necesarios, y asociar dos Listeners a cada componente: uno definido en la vista y otro definido en el controlador de presentación, pero es una alternativa innecesariamente complicada.

## 4.3 La Clase VistaPrincipal

La vista principal es la que contiene los componentes gráficos y gestiona los eventos. La vista está dividida en tres zonas **(obviamente, esto es sólo un ejemplo, en cada caso concreto habrá que redefinirlo)**:

- Un menú, que contiene una opción para salir y un submenú con cuatro opciones (análogas a las funciones asociadas a los botones)
- Un panel de botones
- Un panel de información

El árbol de contenedores y componentes del único JFrame de VistaPrincipal es:

```
JPanel panelContenidos (layout BorderLayout)
   |
   +--- JPanel panelBotones (NORTH, layout FlowLayout)
   |       |
   |       +--- JButton buttonLlamadaDominio
   |       |
   |       +--- JButton buttonAbrirJFrame
   |       |
   |       +--- JButton buttonCambiarPanel
   |       |
   |       +--- JButton buttonAbrirDialog
   |
   +--- JPanel panelInformacion (CENTER, layout por defecto)
           |
           +--- JPanel panelInformacionA
```

El *panelInformacionA* es un panel auxiliar que irá cambiando cada vez que se haga click en el botón *buttonCambiarPanel*. Se usa este panel auxiliar para no tener problemas con los layout managers, puesto que para cambiar de panel lo que realmente se hace es borrar y añadir *panelInformacionA* de/a *panelInformacion*: si se usase *panelInformacion* directamente los componentes podrían cambiar de sitio al repintar *panelContenidos*.

Los dos paneles a los que podrá apuntar *panelInformacionA* son:

```
JPanel panelInformacion1 (layout BorderLayout)
   |
   +--- JLabel labelPanelInformacion1 (NORTH)
   |
   +--- JComboBox comboboxInformacion1 (EAST)
   |
   +--- JScrollPane (anonimo, SOUTH)
            |
            +--- JTextArea textareaInformacion1
```

y

```
JPanel panelInformacion2 (layout BorderLayout)
   |
   +--- JLabel labelPanelInformacion2 (NORTH)
   |
   +--- JTextField textfieldInformacion2 (WEST)
   |
   +--- JSpinner spinnerInformacion2 (SOUTH)
   |
   +--- JSlider sliderInformacion2 (CENTER)
```

El código está estructurado de la siguiente manera:

- Hay un atributo privado *iCtrlPresentacion* para guardar el controlador de presentación (que recibe por parámetro en el constructor) y atributos privados para el único JFrame que contiene (*frameVista*) y todos sus componentes
- El constructor se guarda el controlador de presentación y ejecuta el método *inicializarComponentes()*, que:
  - Acaba de definir las características de *frameVista* (operación por defecto al cerrar, tamaño,...)
  - Crea la jerarquía de componentes y contenedores explicada anteriormente (incluyendo los layout managers de los contenedores)
  - Inicializa los componentes (tamaños, literales, opciones posibles,...)
  - Asigna *panelInformacion1* como el "primer panel visible"
  - Asigna los Listeners a los componentes, en forma de clases anónimas, llamando (en esencia) a un método por cada acción a ejecutar (menús y botones podrían compartir Listener, si no fuera por el cast en *event.getSource()*)
- Hay tres métodos públicos más, :
  - *hacerVisible()*, que hace un *pack()* y un *setVisible(true)* de *frameVista*
  - *activar()*, que hace un *setEnabled(true)* de *frameVista*
  - *desactivar()*, que hace un *setEnabled(false)* de *frameVista*

Las funciones asociadas a los botones son autoexplicativas:

- *buttonLlamadaDominio*: Hace una llamada a dominio
  **La llamada a dominio se hace vía el controlador de presentación:**
  *ArrayList<String> resulDominio = iCtrlPresentacion.llamadaDominio1(...);*
  en el método
  *public void actionPerformed_buttonLlamadaDominio (ActionEvent event)*

- *buttonAbrirJFrame*: Abre una nueva vista (VistaSecundaria), bloqueando VistaPrincipal
  **La sincronización entre vistas se hace vía el controlador de presentación:**
  *iCtrlPresentacion.sincronizacionVistaPrincipal_a_Secundaria();*
  en el método
  *public void actionPerformed_buttonAbrirJFrame (ActionEvent event)*

- *buttonCambiarPanel*: Intercambia el apuntador al JPanel *panelInformacionA* entre *panelInformacion1* y *panelInformacion2*.

- *buttonAbrirDialog*: Abre un diálogo modal: VistaDialogo

## 4.4 La Clase VistaSecundaria

La clase VistaSecundaria es similar a la clase VistaPrincipal, aunque mucho más simple en cuanto a estructura y funcionalidades. El panel de información contiene un JScrollPane (anónimo) que a su vez contiene un JTextArea *textareaInformacion*. Las dos únicas funcionalides asociadas a los botones *buttonLlamadaDominio* y *buttonVolver* son:

- *buttonLlamadaDominio*: Hace una llamada a dominio
  **La llamada a dominio se hace vía el controlador de presentación:**
  *ArrayList<String> resulDominio = iCtrlPresentacion.llamadaDominio2();*
  en el método
  *public void actionPerformed_buttonLlamadaDominio (ActionEvent event)*

- *buttonVolver*: Hace "desaparecer" la vista actual, activando VistaPrincipal
  **La sincronización entre vistas se hace vía el controlador de presentación:**
  *iCtrlPresentacion.sincronizacionVistaSecundaria_a_Principal();*
  en el método
  *public void actionPerformed_buttonVolver (ActionEvent event)*

## 4.5 La Clase VistaDialogo

La clase VistaDialogo implementa un diálogo modal parametrizado, pensado para dar información, avisar de errores o hacer preguntas con respuesta cerrada. Al ser un diálogo modal, la sincronización con la vista principal se realiza automáticamente.

## 4.6 La Clase CtrlDominio

En este ejemplo el controlador de dominio no haca prácticamente nada, salvo informar una lista de strings por cada uno de los (dos) métodos implementados. En una aplicación real, tendrá que hacer muchas más cosas.

```
import java.util.*;

public class CtrlDominio {

  // Constructor y metodos de inicializacion

  public CtrlDominio() {
  }

  public void inicializarCtrlDominio() {
  }

  // Llamadas desde el controlador de presentacion

  public ArrayList<String> llamadaDominio1 (String selectedItem) {
    ArrayList<String> lista = new ArrayList<String>();
    for (int i=0; i<4; ++i)
      lista.add(selectedItem + " " + "resulDominio1-" + i);
    return lista;
  }

  public ArrayList<String> llamadaDominio2() {
    ArrayList<String> lista = new ArrayList<String>();
    for (int i=0; i<3; ++i)
      lista.add("resulDominio2-" + i);
    return lista;
  }
}
```

# 5 Implementación con Netbeans

Netbeans ofrece un editor de interficies que, inicialmente, puede servir para generar el esqueleto de la interficie final. Al ser un editor gráfico, es muy fácil y rápido generar una primera versión de la interficie (el código se genera automáticamente desde el editor gráfico). Por contra, puede ser complicado tanto entender las innumerables opciones que ofrece como modificar a mano lo que genera automáticamente.

Al crear una interficie gráfica con Netbeans hay que tener en cuenta que:

- El *main()* se implementa con AWT (ver sección 3.2 para la alternativa equivalente con Swing), con la única instrucción de hacer visible un JFrame:

```
public static void main(String args[]) {
  java.awt.EventQueue.invokeLater(
    new Runnable() {
      public void run() {
        new MiJFrame().setVisible(true);
    }
  });
```

  **Habrá que cambiarlo para incluir el controlador de presentación**

- Por defecto, todas las vistas creadas con el editor gráfico heredan de JFrame (ver sección 3.2 para una alternativa)

- Por defecto, todos los contenedores tienen un método *initComponents()*, que se ejecutan en el constructor, destinado a definir la parte "estática" de la interficie: acabar de definir las características del contenedor, definir los componentes, y sus propiedades, definir y asociar los layout managers, añadir los componentes a los contenedores, asociar los Listeners a los componentes, etc
  **Este método se genera automáticamente** (entre GEN-BEGIN:initComponents y GEN-END:initComponents)

- Para cada componente y evento seleccionados:
  - Se crea un método *NombreComponenteNombreEvento(...)*, sin instrucciones
    **Estos métodos se generan automáticamente** (entre GEN-FIRST:Metodo y GEN-LAST:Metodo)
  - En *initComponents()*, se asocia al componente un Listener del evento correspondiente, cuya única instrucción es una llamada a *NombreComponenteNombreEvento(...)*

- Por defecto, el layout manager que usa Netbeans para todos los contenedores es GroupLayout (el código asociado normalmente es algo difícil de entender)

- La definición de las componentes es automática (entre GEN-BEGIN:variables y GEN-END:variables)

# References


[1] *Aprenda Java como si estuviera en Primero*, Escuela Superior de Ingenieros Industriales de San Sebastián, Universidad de Navarra, 2000.

[2] Wikipedia - Abstract Window Toolkit: `http://en.wikipedia.org/wiki/Abstract_Window_Toolkit`

[3] Wikipedia - Swing: `http://en.wikipedia.org/wiki/Swing_(Java)`

[4] Wikipedia - Standard Widget Toolkit: `http://http://en.wikipedia.org/wiki/Standard_Widget_Toolkit`

[5] Tutorial de Swing: `http://download.oracle.com/javase/tutorial/uiswing/`

[6] Jerarquía de clases de Swing: `http://download.oracle.com/javase/6/docs/api/javax/swing/package-tree.html`

[7] *AWT & Swing*, Departamento de Informática y Automática, Universidad de Salamanca, 2003.

[8] Threads and Swing: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html