Soporte de Java para el patrón Command

(Swing)

Java proporciona soporte para el patrón Command mediante su sistema de delegación de eventos basado en AWT (Abstract Window Toolkit). En este *framework*:

- Los objetos comando implementan la interfaz ActionListener, en lugar de la interfaz Command.
- El método para ejecutar operaciones es actionPerformed(), en lugar de execute().
- Para asociar un objeto comando con un componente gráfico (widget), tan sólo tenemos que utilizar el método addXXXListener del widget, proporcionando como argumento una instancia de un objeto que implemente la interfaz ActionListener.

Por ejemplo, para el caso de un botón tendríamos algo del estilo:

```
javax.swing.JButton miBoton = new javax.swing.JButton();
miBoton.addActionListener(new java.awt.event.ActionListener() {
      @Override
      public void actionPerformed(java.awt.event.ActionEvent e){
            // Tarea cuando se pulsa el botón
      }
});
```

Notad que en este ejemplo:

- El botón es el Invoker.
- La clase anónima que implementa la interfaz ActionListener es el objeto Command.
- No se especifica quién es el Receiver.

Veamos a continuación una serie de ejemplos que nos ayudarán a profundizar en el soporte de Java para el patrón Command.

Posibilidades de diseño

Supongamos una aplicación con una GUI con botones en una barra de herramientas que permiten al usuario realizar diversas acciones, así como un menú que también permite llevar a cabo las mismas acciones que los botones.

Tenemos varias opciones para implementar esto:

- Un único método actionPerformed() para todos los widgets. Podemos hacer que la clase que contiene la GUI implemente la interfaz ActionListener, con lo que tendremos que proporcionar código para un único método actionPerformed() -el manejador de eventos. En este método controlaremos los eventos producidos por los botones y por los elementos del menú. Sin embargo, esto atenta contra el principio de diseño abierto/cerrado. Recordemos el código del documento de los "functors":

```
// Manejar las acciones GUI. Esta es la función de callback
@Override public void actionPerformed(ActionEvent event) {
        Object src = event.getSource();
        if (src == btnTirarDado)
                tirarDado();
        else if (src == btnSalir)
                System.exit(0);
}
```

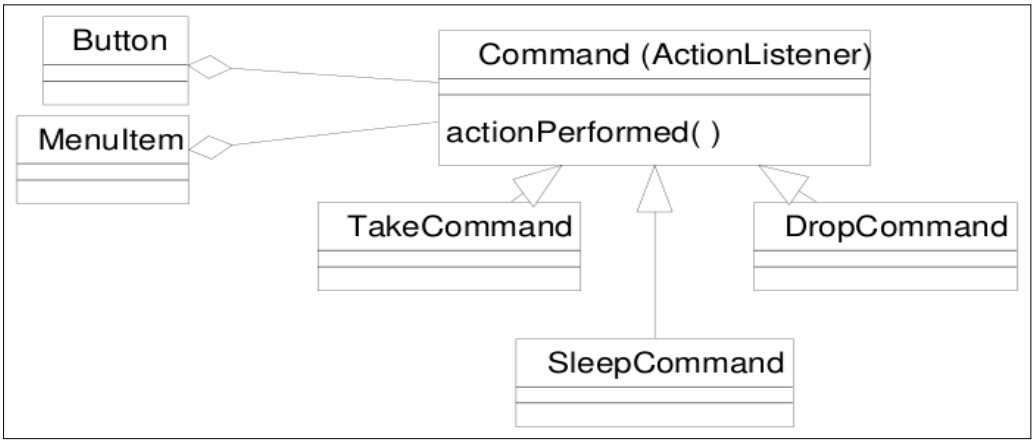
- Un método actionPerformed() por cada widget. Otra opción, que también conocemos, es que cada botón y cada elemento del menú tengan su propio método actionPerformed() para realizar su tarea, por lo que la clase que contiene la GUI no tendrá que implementar la interfaz ActionListener. En el documento de los "functors" vimos que mediante una clase anónima podíamos encapsular el comportamiento asociado a una acción con el widget que permite ejecutar esa acción:

```
// Se utilizan instancias de clases anónimas como objetos
```

```
// callback para el manejo de los eventos de los botones
btnTirarDado.addActionListener(new ActionListener() {  // clase anónima
        @Override public void actionPerformed(ActionEvent e) {
                tirarDado();
        }
});
btnSalir.addActionListener(new ActionListener() {  // clase anónima
        @Override
        public void actionPerformed(ActionEvent e) {
                System.exit(0);
        }
});
```

Ahora bien, dado que tenemos una relación uno a uno entre los elementos del menú y los botones de la barra de tareas, con esta aproximación estaremos repitiendo código, es decir, tendremos clases anónimas iguales para cada par ítem de menú/botón. Para evitar esto, veamos la tercera opción de diseño.

- Métodos actionPerformed() compartidos por widgets que realicen la misma tarea. Este sería el caso de cada pareja botón/elemento de menú. Para implementar esta aproximación, que se ajusta en gran medida al concepto del patrón Command, no podemos utilizar clases anónimas, sino clases “normales” (con nombre), ya que en otro caso no sería posible referenciarlas desde los widgets. Estas clases implementarán la interfaz Command, por lo que deben proporcionar código para el método actionPerformed() (execute(), según el patrón Command). La figura siguiente muestra la idea de esta tercera opción:

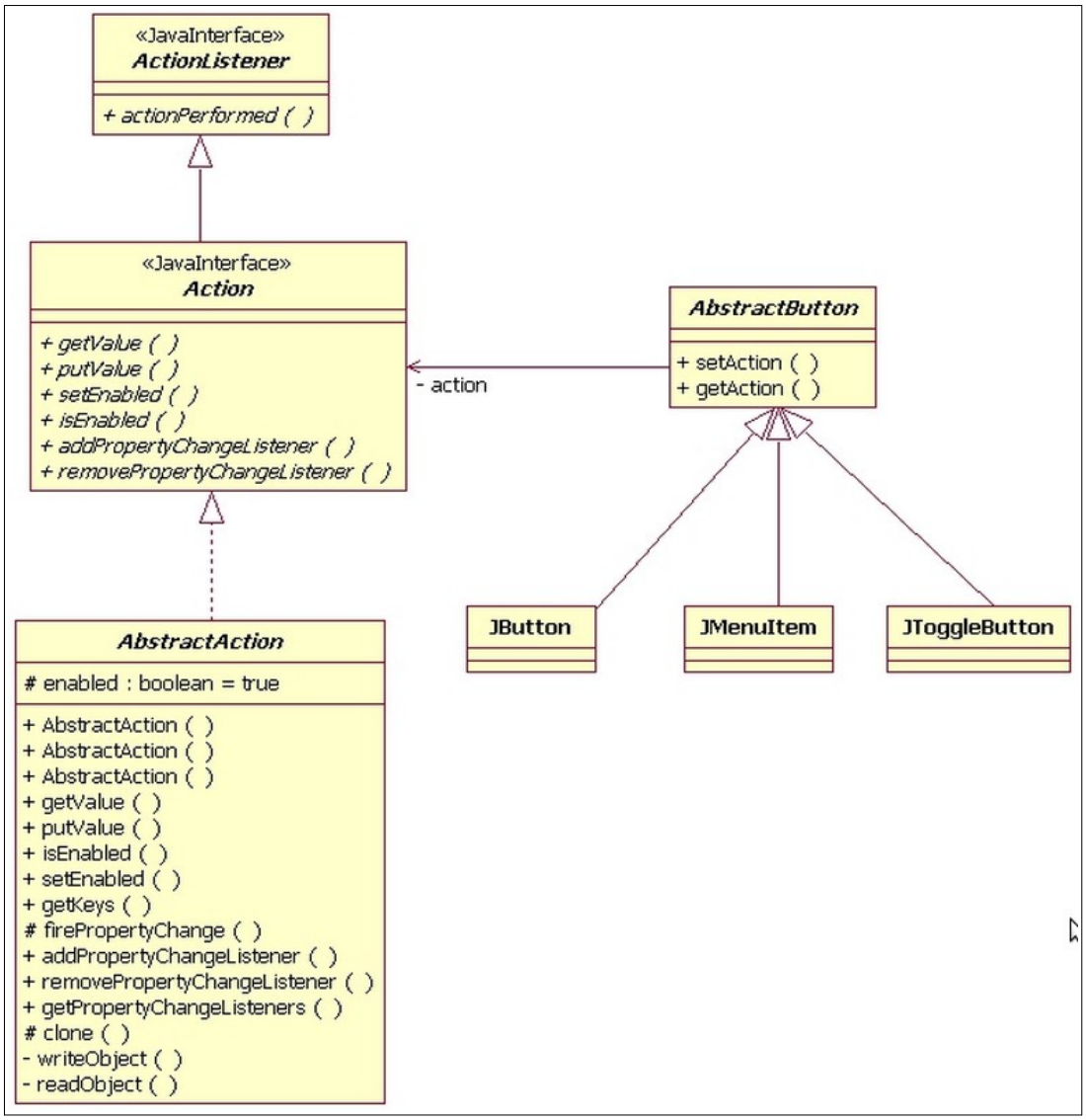


Según la figura, un botón y un elemento de menú pueden compartir un mismo objeto TakeCommand, por ejemplo. La clase TakeCommand, igual que el resto de subclases de Command, contendrá el mismo código que habríamos puesto en una clase anónima si hubiéramos seguido la segunda aproximación.

Swing no utiliza la nomenclatura Command. En su lugar proporciona la interfaz Action y la subclase AbstractAction, mediante las cuales podemos llevar a cabo lo descrito en la tercera aproximación, vista anteriormente.

La figura siguiente muestra una versión simplificada de las clases que utiliza Swing en su implementación del patrón Command. Podemos ver dos jerarquías:

- La jerarquía de Action, equivalente a la jerarquía de Command.
- La jerarquía de AbstractButton, equivalente a la clase Invoker.



AbstractAction es una clase de conveniencia que nos evita que nuestros comandos tengan que implementar directamente la interfaz Action. Notad que el programador de una aplicación final creará subclases de AbstractAction para definir (sus) comandos que serán llamados por los diferentes widgets (botones, menús, etc.)

JButton, JMenuItem, JToggleButton y otras subclases de AbstractButton que no aparecen aquí por simplicidad, son Invokers, según el diagrama general de patrón Command, esto es, clases que llaman a objetos comando.

La jerarquía de Action son clases comando según el patrón Command. Así para indicar al Invoker cual es su Command basta con hacer lo siguiente:

miButton.setAction(miAction);

Cuando algún widget es pulsado o seleccionado, invoca al método actionPerformed() del objeto Action que tenga asociado. La siguiente clase representa un comando creado por un programador con tal de mostrar un diálogo “AcercaDe”:

```
class MostrarDialogoAction extends AbstractAction {
	public MostrarDialogoAction() {
		super("Mostrar dialogo");
	}
	public void actionPerformed(ActionEvent e) {
		JOptionPane.showMessageDialog(
			(Component) e.getSource(),
			"Gestion v1.0 Licencia GPL");
	}
}
```

La clase anterior puede ser enlazada por un botón y por un elemento de menú, por ejemplo.

Las Action de Swing además de permitirnos implementar el patrón Command ofrecen otras posibilidades:

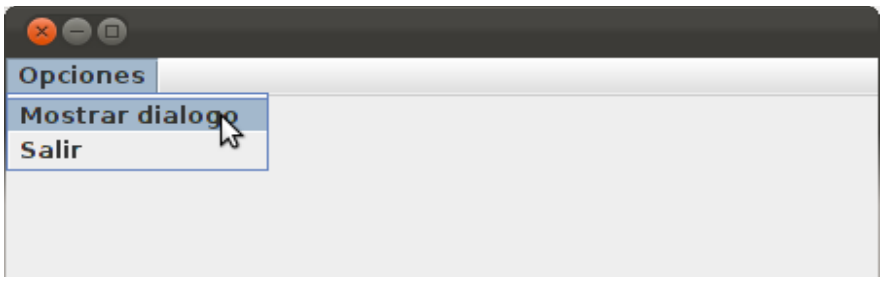
- Nos permiten establecer el texto asociado a la acción (texto que aparecerá en botones y menús).
- Los iconos asociados a los widgets.
- Activación/desactivación de los widgets (útil cuando no tiene sentido que en una determinada situación un widget sea seleccionado por el usuario).

A continuación veremos algunos ejemplos que utilizan la interfaz Action de Swing.

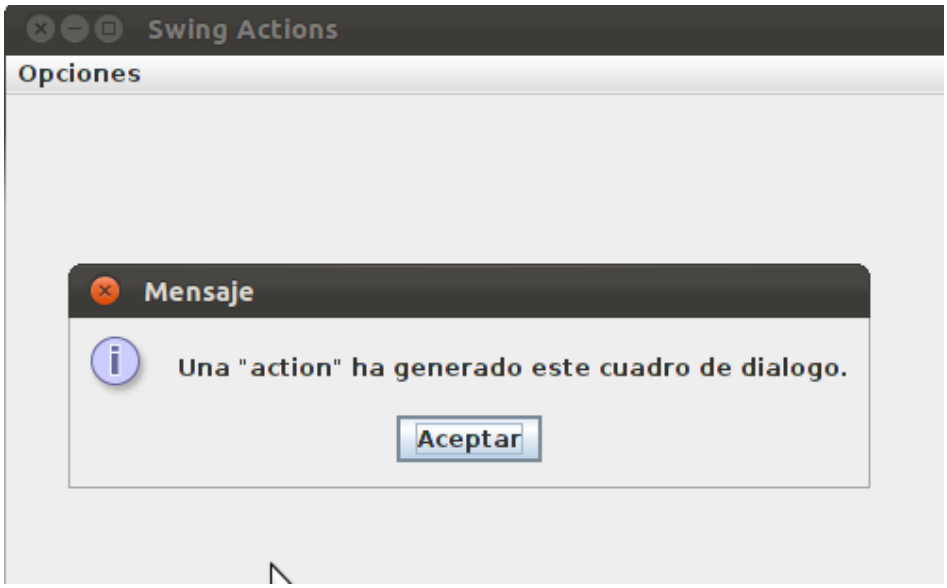
Cada uno hace énfasis en algún aspecto de la interfaz Action. Ninguno de los ejemplos utiliza Receivers (objetos receptores que realizan el trabajo asociado a un petición). Más adelante veremos ejemplos con Receivers.

Ejemplo 1 - Uso de clases Action

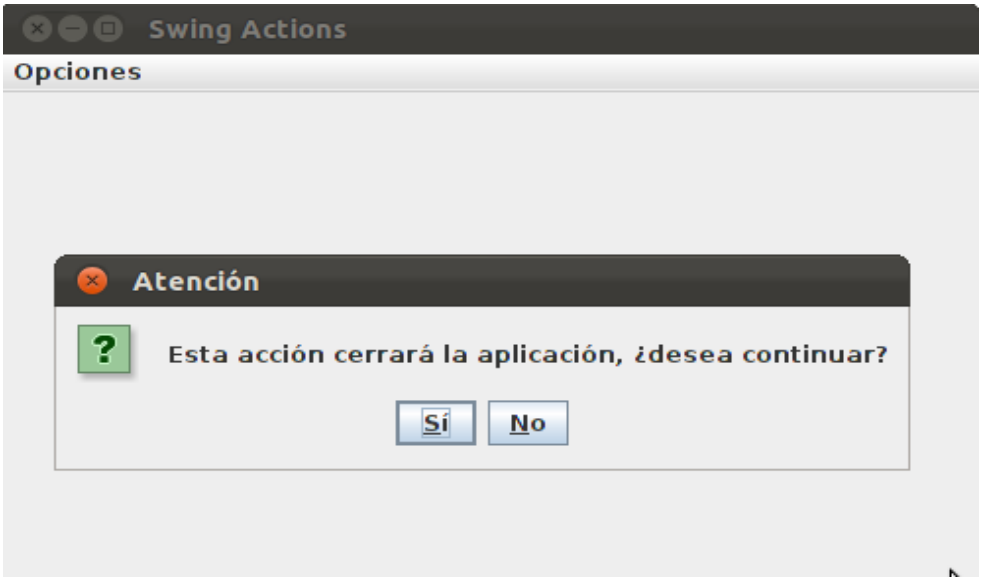
En este primer ejemplo crearemos una ventana principal con un menú formado por dos opciones de menú:



El primer elemento de menú mostrará un cuadro de diálogo indicando quién lo ha invocado:



El segundo elemento de menú nos preguntará si queremos finalizar la aplicación:



Para cada una de estas opciones de menú crearemos una clase Action:

MostrarDialogoAction.java

```
package comportamiento.command.swingActions.ejemplo1_1;
import java.awt.Component;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.JOptionPane;

class MostrarDialogoAction extends AbstractAction {
	private static final long serialVersionUID = 1L;
	public MostrarDialogoAction() {
		super("Mostrar dialogo");
	}
	public void actionPerformed(ActionEvent e) {
		JOptionPane.showMessageDialog(
			(Component) e.getSource(),
			"Una \"action\" ha generado este cuadro de
dialogo.");
	}
```

```
}
```

SalirAction.java

```
package comportamiento.command.swingActions.ejemplo1_1;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.JOptionPane;

class SalirAction extends AbstractAction {
	private static final long serialVersionUID = 1L;
	public SalirAction() {
		super("Salir");
	}
	public void actionPerformed(ActionEvent e) {
		int respuesta = JOptionPane.showConfirmDialog(
			null,
			"Esta acción cerrará la aplicación, ¿desea
continuar?",
			"Atención",
			JOptionPane.YES_NO_OPTION);

		if (respuesta == JOptionPane.YES_OPTION) {
			System.exit(0);
		}
	}
}
```

Notad que las dos clases anteriores extienden a la clase AbstractAction de Swing. Ambas clases implementan el método actionPerformed() en el que se encapsula una determinada funcionalidad requerida por el usuario del framework.

Tanto MostrarDialogoAction como SalirAction son clases totalmente desconocidas por el framework y sin embargo consigue invocarlas gracias a que implementan la interfaz Action (vía herencia de AbstractAction).

Y finalmente, vemos el código para clase de la ventana principal:

TestSwingActions.java

```
package comportamiento.command.swingActions.ejemplo1_1;
import java.awt.EventQueue;
import javax.swing.*;

public class TestSwingActions extends JFrame {
	private static final long serialVersionUID = 1L;
	public static void main(String args[]) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					TestSwingActions instancia = new
TestSwingActions();
					instancia.setTitle("Swing Actions");
					instancia.setSize(500, 400);
					instancia.setLocationRelativeTo(null);

					instancia.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	// Constructor
	public TestSwingActions() {
```

```
            // Creamos una barra de menús
            JMenuBar barraMenu = new JMenuBar();
            // Creamos un menú
            JMenu menuOpciones = new JMenu("Opciones");

            /*
             * Creamos dos elementos de menú. Al proporcionar un
             * objeto Action al método add() de JMenu, esta action ya
             * queda vinculada al nuevo elemento de menú que se crea.
             */
            menuOpciones.add(new MostrarDialogoAction());
            menuOpciones.add(new SalirAction());
            // Añadimos el menu a la barra
            barraMenu.add(menuOpciones);
            // Añadimos la barra al JFrame (ventana principal)
            setJMenuBar(barraMenu);
        }
}
```

Comentarios sobre la clase principal:

TestSwingActions crea un menú llamado menuOpciones, que referencia a una instancia del widget JMenu, y mediante su método add() le pasa una instancia de MostrarDialogoAction y de SalirAction. El método add(), implícitamente, crea una instancia de widget JMenuItem y le asocia el objeto Action que recibe como argumento. Esto último lo consigue, también implícitamente, invocando al método setAction() de JMenuItem (este método se hereda de AbstractAction). Para que esto se vea más claro, veamos cómo podríamos haber conseguido lo mismo, pero de manera explícita, para el elemento de menú Mostrar dialogo;

```
        // Version corta
        //menuOpciones.add(new MostrarDialogoAction());

        // Version larga
```

```
JMenuItem menuItemMostrarDialogo = new JMenuItem();
menuItemMostrarDialogo.setAction(new MostrarDialogoAction());
menuOpciones.add(menuItemMostrarDialogo);
```

La cuestión es que los elementos de menú y las acciones ya quedan enlazados, por lo que cuando un usuario seleccione una opción de menú, se invocará al método actionPerformed() de la acción correspondiente.

ActionPerformed() recibe como parámetro un ActionEvent, el cual proporciona información sobre el evento, incluyendo el componente que lo ha generado. La clase MostrarDialogoAction aprovecha esa información para generar un texto, mientras que SalirAction no, ya que no la necesita.

Ejemplo 2 – Compartir una Action entre dos widgets

En este ejemplo veremos que cuando dos controles gráficos realizan el mismo comando es interesante que utilicen el mismo objeto Action, ya que así, además de reducir código innecesario, tenemos un único sitio donde mantener el código.

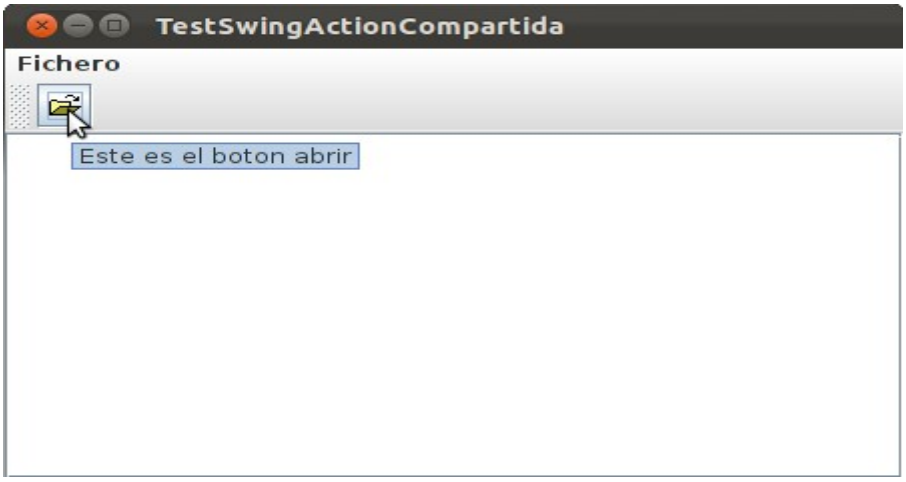
Ya que la misma Action puede compartirse entre otros widgets, necesitamos menos clases manejadoras de eventos. Para ello, tenemos que definir una Action de Swing y compartirla con los dos widgets, que en el ejemplo que sigue se trata de un botón de una barra de herramientas y de un elemento de menú.

A continuación, un extracto del código como anticipo:

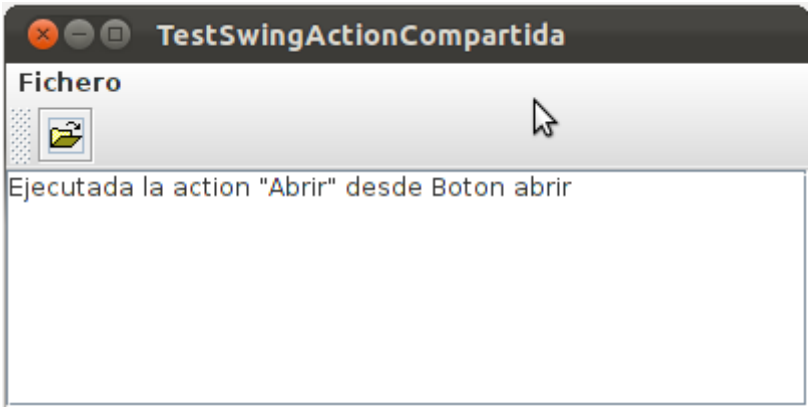
```
// Crear una action como clase anónima para el comando "Abrir"
Action abrirAction = new AbstractAction("Abrir") {
      @Override public void actionPerformed(ActionEvent e) {
            cajaTexto.append("Ejecutada la action \"Abrir\" desde " +
                              e.getActionCommand() + "\n");
            }
};

/* Añadimos el botón "Abrir" a la barra de herramientas e
```

```
* indicamos la action que se debe ejecutar al pulsar el botón */

botonAbrir = barraHerramientas.add(abrirAction);

/* Usamos la misma action para añadir un elemento de menú* al menú
'Fichero' */

elementoMenuAbrir = menuFichero.add(abrirAction);
```

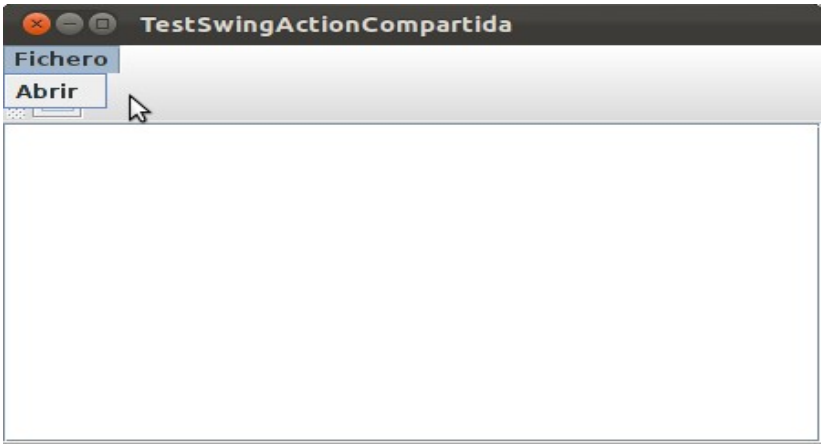
Al iniciar la aplicación veremos un menú denominado “Fichero” y un botón en la barra de herramientas con un icono para abrir algo (supongamos un documento):



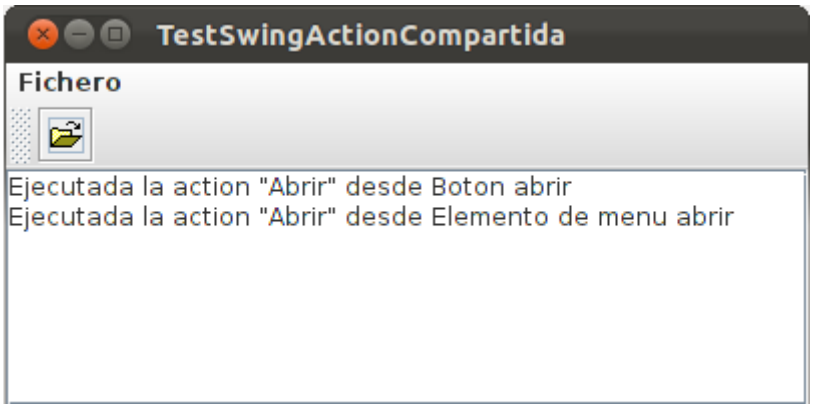
Al pulsar sobre el botón veremos en la parte central de la ventana (que se trata de una caja de texto) el mensaje generado por la Action. Notad cómo especifica quien la llamó:



También podemos desplegar el menú y seleccionar (la única) opción 'Abrir':



Igual que en el caso anterior, veremos en la parte central de la ventana el mensaje generado por la Action:



Pasemos a ver el código, que en este ejemplo se trata de una única clase, ya que la Action se implementa como clase anónima en la propia clase principal.

TestSwingActionCompartida.java

```
package comportamiento.command.swingActions.ejemplo1_2;


import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
```

```
import javax.swing.*
public class TestSwingActionCompartida extends JFrame {

	private static final long serialVersionUID = 1L;
	private JMenuBar barraMenu;
	private JMenu menuFichero;
	private JMenuItem elementoMenuAbrir;
	private JToolBar barraHerramientas;
	private JButton botonAbrir;
	private JTextArea cajaTexto;
	private Action abrirAction;

	public TestSwingActionCompartida() {
		super("TestSwingActionCompartida");
		crearGUI();
	}

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					TestSwingActionCompartida instancia = new
TestSwingActionCompartida();
					instancia.pack();
					instancia.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
```

```
                }
            });
        }

        private void crearGUI() {
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            // Crear la barra de menú y el menú Fichero
            barraMenu = new JMenuBar();
            menuFichero = new JMenu("Fichero");
            // Crear la barra de herramientas
            barraHerramientas = new JToolBar();

            // Crear la caja de texto para mostrar la salida.
            cajaTexto = new JTextArea(5, 30);
            // Añadir barras de desplazamiento a la caja de texto
            JScrollPane scrollPane = new JScrollPane(cajaTexto);
            // Creamos un panel para organizar en él los controles
            JPanel panel = new JPanel();
            panel.setLayout(new BorderLayout());
            panel.setPreferredSize(new Dimension(400, 150));
            panel.add(barraHerramientas, BorderLayout.NORTH);
            panel.add(scrollPane, BorderLayout.CENTER);
            setContentPane(panel);

            // Configurar la barra de menú y añadirla a la ventana
(JFrame)
            barraMenu.add(menuFichero);
            setJMenuBar(barraMenu);
            /*
```

```
             * Crear una action como clase anónima para el comando
"Abrir"
             */
            ImageIcon iconoAbrir = new ImageIcon("abrir.gif");
            abrirAction = new AbstractAction("Abrir", iconoAbrir)
            {
                  private static final long serialVersionUID = 1L;

                  @Override
                  public void actionPerformed(ActionEvent e) {
                        cajaTexto.append("Ejecutada la action \"Abrir\"
desde " +
                                    e.getActionCommand() + "\n");
                  }
            };

            /*
             * Añadimos el botón "Abrir" a la barra de herramientas e
             * indicamos la action que se debe ejecutar al pulsar el
botón
             */
            botonAbrir = barraHerramientas.add(abrirAction);
            botonAbrir.setText("");
            botonAbrir.setActionCommand("Boton abrir");
            botonAbrir.setToolTipText("Este es el boton abrir");

            /*
             * Usamos la misma action para añadir un elemento de menú
             * al menú 'Fichero'
             */
```

```
            elementoMenuAbrir = menuFichero.add(abrirAction);
            elementoMenuAbrir.setIcon(iconoAbrir);
            elementoMenuAbrir.setActionCommand("Elemento de menu
abrir");
      }
}
```

Ejemplo 3 – Activar/desactivar una Action

En este ejemplo vamos a destacar el hecho de que una acción (o comando) es algo más que sólo código que realiza una operación, es decir, los comandos tienen estado. Por ejemplo, si en un procesador de textos un usuario no ha seleccionado texto alguno entonces la opción “Cortar”, tanto del elemento del menú como del botón de la barra de herramientas, debe estar desactivada.

Un control GUI desactivado normalmente presenta una apariencia diferente que cuando está activado. Por tanto, sería de ayuda que el comando "Cortar" pudiera “recordar” en todo momento si se encuentra activado o desactivado. De esta manera, vemos que un comando tiene tanto comportamiento (implementado en los métodos) como estado (almacenado en los atributos).

Hemos visto en los ejemplos anteriores que la interfaz Action de Swing nos permite implementar comandos que pueden almacenar información sobre los iconos y los textos que deben aparecer en menús, elementos de menú, botones, etc (que Swing obtiene a partir del nombre de la acción).

Establecemos estas propiedades mediante el método putValue(), utilizando una serie de constantes predefinidas en la propia interfaz Action. Por ejemplo:

miAccion.putValue(Action.NAME, "Subir");

miAccion.putValue(Action.SMALL_ICON.new ImageIcon("subir.png"));

Como ya hemos visto anteriormente, una vez que tenemos un objeto Action es muy fácil añadirlo a un menú y a la barra de herramientas:

JMenu menu = new Jmenu("Simulador");

...

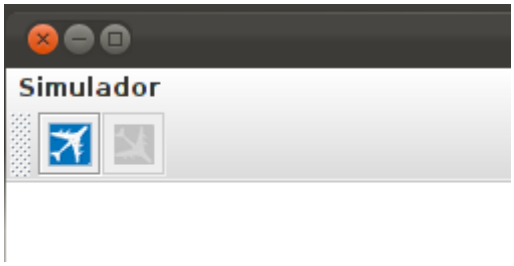
menu.add(miAccion);

toolbar.add(miAccion);

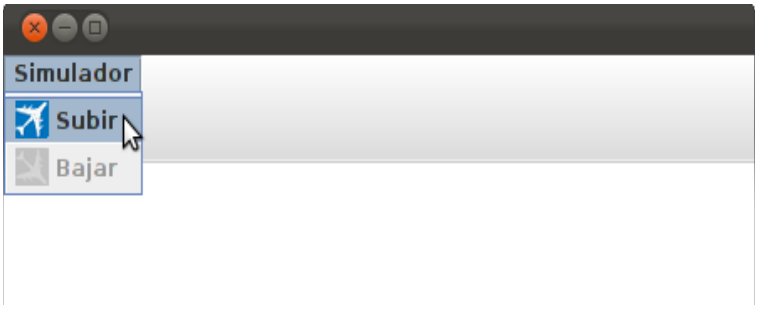
Tanto el menú como la barra de herramientas recuperan el nombre de la acción y su icono correspondiente y los muestran.

Por tanto, no es de extrañar que también sea posible activar o desactivar los widgets (comandos). Para ello utilizamos, respectivamente, los métodos setEnabled(true) y setEnabled(false).

En la siguiente imagen podemos ver cómo colaboran dos botones opuestos -el que permite ascender el avión está activado por lo que el avión en estos momentos está descendiendo y lo único que permite hacer el simulador es ascenderlo:



Del mismo modo, al desplegar el menú 'Simulador' podemos ver los elementos de menú 'Subir" y 'Bajar', también de modo excluyentes, como en la barra de herramientas:



Veamos el código. Por un lado tenemos la clase Action:

SimulacionVueloAction.java

```
package comportamiento.command.swingActions.ejemplo1_3;
import java.awt.event.*;
import javax.swing.*;
```

```java
/*
 Esta Action escribe un mensaje en una caja de texto y a continuación
 se desactiva para poner como activa a su Action opuesta.
 */
public class SimulacionVueloAction extends AbstractAction {
	private static final long serialVersionUID = 1L;
	private String sentidoVuelo;
	private JTextArea textArea;
	private Action accionOpuesta;

	public SimulacionVueloAction(String sentidoVuelo, JTextArea
textArea) {
		this.sentidoVuelo = sentidoVuelo;
		this.textArea = textArea;
	}

	// Establecer la Action opuesta
	public void setOpuesto(Action action) {
		accionOpuesta = action;
	}

	@Override
	public void actionPerformed(ActionEvent event) {
		textArea.append(sentidoVuelo);
		textArea.append("\n");
		if (accionOpuesta != null) {
			setEnabled(false); // Se autidesactiva
			accionOpuesta.setEnabled(true); // activa la opuesta
		}
	}
}
```

Y por otro lado, tenemos la clase con el main:

TestSwingActionEnabledDisabled.java

```
package comportamiento.command.swingActions.ejemplo1_3;
import java.awt.*;
import javax.swing.*;
/*
 * Creamos dos instancias de la clase SimulacionVueloAction.
 * Estas acciones se ejecutan mediante un menú o un botón
 * de la barra de herramientas.

 */
public class TestSwingActionEnabledDisabled extends JFrame {

	private static final long serialVersionUID = 1L;

	public static void main(String[] args) {

		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					TestSwingActionEnabledDisabled instancia
=
						new
TestSwingActionEnabledDisabled();
					instancia.pack();
					instancia.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
```

```
        });
    }

    public TestSwingActionEnabledDisabled() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Creamos la barra de menú y el menú
        JMenuBar barraMenu = new JMenuBar();
        JMenu menu = new JMenu("Simulador");
        barraMenu.add(menu);

        // Añadimos lo anterior al JFrame
        setJMenuBar(barraMenu);

        // Creamos una barra de herramientas
        JToolBar toolBar = new JToolBar();
        JTextArea textArea = new JTextArea(10, 40);

        // Añadimos lo anterior al JFrame
        add(toolBar, BorderLayout.NORTH);
        add(textArea, BorderLayout.CENTER);

        /*
         * Creamos las dos instancias de la Action
         */
        SimulacionVueloAction subirAction =
                new SimulacionVueloAction("Subiendo...",
textArea);
        subirAction.putValue(Action.NAME, "Subir");
        subirAction.putValue(Action.SMALL_ICON,
                new ImageIcon("subir.png"));
```

```
            SimulacionVueloAction bajarAction =
                        new SimulacionVueloAction("Bajando...",
textArea);
            bajarAction.putValue(Action.NAME, "Bajar");
            bajarAction.putValue(Action.SMALL_ICON,
                        new ImageIcon("bajar.png"));

            // Establecemos el opuesta de cada una
            subirAction.setOpuesto(bajarAction);
            bajarAction.setOpuesto(subirAction);

            // Decidimos que la Action activa sea la subir
            bajarAction.setEnabled(false);

            /*
             * Añadimos las Action al menú, lo que implícitamente crea
             * dos objetos JMenuItem
             */
            menu.add(subirAction);
            menu.add(bajarAction);

            /*
             * Añadimos las Action a la barra de herramientas, lo que
             * implícitamente crea dos objetos JButton
             */
            toolBar.add(subirAction);
            toolBar.add(bajarAction);
      }
}
```

Conclusión sobre las Action de Swing

Después de todo lo visto debe quedar claro que las Action de Swing proporcionan una manera de manejar eventos más orientada a objetos que si nos limitáramos a no usarlas, como veíamos en los diseños iniciales de esta sección. Por ejemplo, mediante las Action nos resultaría fácil cambiar el comportamiento de un widget, ya que tan sólo tendríamos que enlazarle un objeto Action diferente.

Ahora bien, el diseño visto hasta ahora de las Action presenta un inconveniente importante: la dificultad de su reutilización en otras aplicaciones. Esta dificultad obedece al hecho de que las clases que implementan la interfaz Action o, más habitual, extienden a AbstractAction, contienen lógica especifica de una aplicación en el método actionPerformed(). La solución a esto pasa por separar conceptos.

Separación de conceptos

Es frecuente que utilicemos en nuestros programa acciones del tipo 'Cortar', 'Copiar' y 'Pegar', por lo que sería interesante poder separar las propiedades genéricas de estas acciones (sus nombres, sus iconos, sus teclas de acceso rápido, etc) de la lógica especifica de la aplicación, es decir, del comportamiento de la acción (si actúa sobre un objeto JTextArea o sobre un JTree, si la acción delega en un Receiver o no, etc).

Entonces, ¿Cómo separamos los conceptos? Pues una manera es crear una clase CommandAction

(PENDIENTE DESARROLLAR ESTA SECCIÓN)