UVa

Introducción a JSwing

Prácticas de Interacción Persona Computadora

```
jLabel2.setText( text: "Fecha:");
jPanel2.add( comp: jLabel2);

diaSpinner.setModel(new javax.swing.SpinnerNumberModel( value: 1,  minimum: 1,  maximum: 31,  stepSize: 1));
diaSpinner.addChangeListener(new javax.swing.event.ChangeListener() {
    public void stateChanged(javax.swing.event.ChangeEvent evt) {
        diaSpinnerStateChanged(evt);
    }
});
jPanel2.add( comp: diaSpinner);

mesSpinner.setModel(new javax.swing.SpinnerNumberModel( value: 1,  minimum: 1,  maximum: 12,  stepSize: 1));
mesSpinner.addChangeListener(new javax.swing.event.ChangeListener() {
    public void stateChanged(javax.swing.event.ChangeEvent evt) {
        mesSpinnerStateChanged(evt);
    }
});
jPanel2.add( comp: mesSpinner);

anoSpinner.setModel(new javax.swing.SpinnerNumberModel( value: 2000,  minimum: 2000,  maximum: 3000,  stepSize: 1));
jPanel2.add( comp: anoSpinner);

jLabel3.setText( text: "Cantidad:");
jPanel2.add( comp: jLabel3);
jPanel2.add( comp: cantidadTextField);
jPanel2.add( comp: mensajeLabel);
jPanel2.add( comp: jLabel6);

jLabel7.setText( text: "Tipo");
jPanel2.add( comp: jLabel7);
```

UVa

Universidad de Valladolid

Grado en Ingeniería Informática. Grado en Estadística, Universidad de Valladolid

Interacción Persona Computadora

Autores Mario Corrales Astorgano [mario.corrales@uva.es] Alejandra Martínez Monés [amartinez@infor.uva.es], v1.0 Fecha 05-2023
Revisado por: Alejandra Martínez Monés y David Escudero Mancebo

JAVA SWING

Índice general

1 Introducción . . . 7

1.1 Contextualización y Justificación . . . 7

1.2 Contenidos . . . 7

1.3 Motivación . . . 8

2 JSwing . . . 9

2.1 Objetivos de aprendizaje . . . 9

2.2 Requisitos previos . . . 9

2.3 Introducción . . . 9

2.4 Contenedores . . . 10

2.5 Componentes . . . 10

2.6 Layout Manager . . . 12

2.7 Netbeans . . . 14

2.8 Gestión de eventos . . . 15

2.9 Ejemplo de aplicación con Netbeans 16
2.9.1 Paso 1: crear un proyecto 16
2.9.2 Paso 2: crear la interfaz 16
2.9.3 Paso 3: Cambiar el texto y el nombre de las variables 19
2.9.4 Paso 4: añadir funcionalidad 21
2.9.5 Paso 5: tratamiento de errores 24
2.10 Ejercicio propuesto 27
2.11 Enlaces a proyectos de ejemplo 28

3 Modelo-Vista-Controlador con JSwing 29
3.1 Objetivos de aprendizaje 29
3.2 Requisitos previos 29
3.3 Modelo Vista Controlador 29
3.4 MVC activo vs MCV pasivo 30
3.5 MVC en JSwing 31
3.5.1 Vista 31
3.5.2 Controlador 32
3.5.3 Modelo 32
3.6 Guías de aplicación del patrón MVC 33
3.7 Ejemplo 33
3.7.1 Paso 1: definir el Modelo 34
3.7.2 Paso 2: definir el Controlador 37
3.7.3 Paso 3: definir la Vista 40
3.8 Ejercicios propuestos 41
3.8.1 Ejercicio 1 41
3.8.2 Ejercicio 2 41
3.9 Enlaces a proyectos de ejemplo 41

4 Gestión de múltiples vistas 43

4.1 Objetivos de aprendizaje 43

4.2 Requisitos previos 43

4.3 Introducción 43

4.4 Máquinas de estados 44

4.5 Máquinas de estado en el diseño de interfaces de usuario 45

4.6 Máquinas de estados en JSwing 45

4.7 Ejemplo 47

4.7.1 Paso 1: creación del gestor de vistas 49

4.7.2 Paso 2: creación de las clases del modelo 49

4.7.3 Paso 3: creación de la clase Main 52

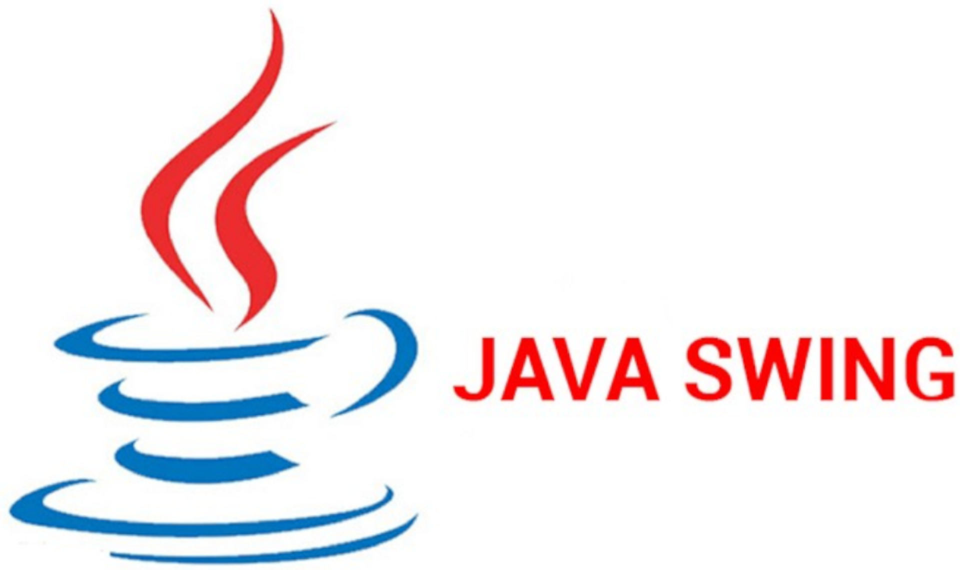
4.7.4 Paso 4: creación de las vistas y controladores 52

4.8 Ejercicio propuesto 55

4.8.1 Ejercicio 1 55

4.9 Enlaces a proyectos de ejemplo 55

UVa



1. Introducción

1.1 Contextualización y Justificación

El desarrollo de aplicaciones con Interfaz Gráfica de Usuario (GUI, en sus siglas en Inglés) es una partes más importantes englobadas dentro del desarrollo de software, ya que cualquier aplicación orientada a ser utilizada por un ser humano debe tener algún tipo de interfaz. Existen multitud de tecnologías orientadas a crear este tipo de aplicaciones, ya sean aplicaciones webs (Angular, Node), aplicaciones de escritorio (JSwing, .NET) o móviles (Android, iOs). Los contenidos desarrollados en la asignatura Interacción Persona Computadora (IPC) del Grado de Ingeniería Informática de la Universidad de Valladolid se enfocan en conceptos generales sobre como los seres humanos interactúan con las aplicaciones informáticas, independientemente de la tecnología utilizada. Aunque sí existen matices concretos dependiendo del tipo de interacción, tamaño de la pantalla, controles disponibles, etc, el aprendizaje de una tecnología puede servir de base para la adaptación a otra.

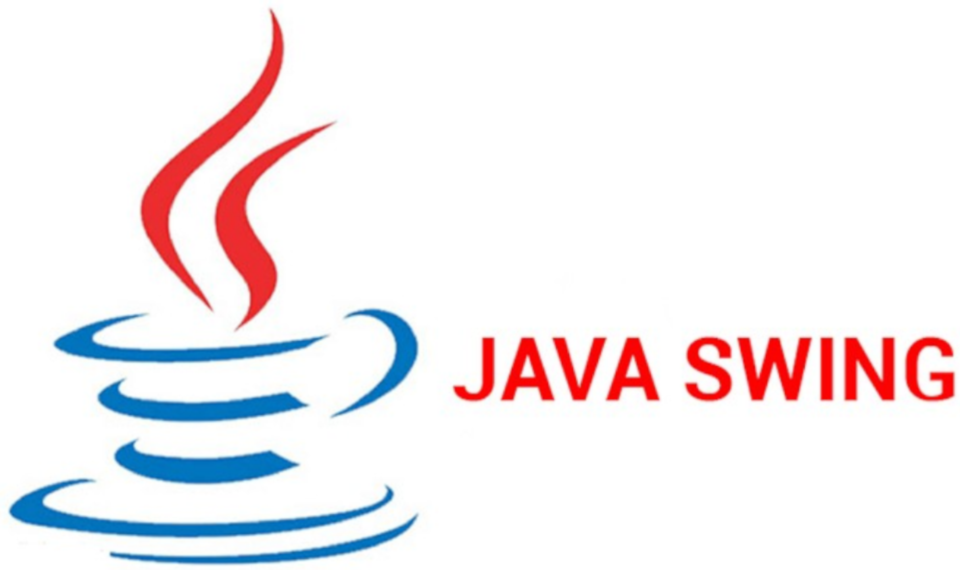
1.2 Contenidos

Este guión de la parte práctica de la asignatura IPC tiene como objetivo ofrecer a los y las estudiantes una introducción a la librería JSwing[cita] integrada dentro del entorno de desarrollo Netbeans[cita], así como la explicación y aplicación del patrón de diseño Modelo-Vista-Controlador (MVC) y la gestión de múltiples vistas utilizando máquinas de estado.

- Capítulo 1: Introducción a JSwing.
- Capítulo 2: Modelo-Vista-Controlador con JSwing
- Capítulo 3: Gestión de múltiples vistas

1.3 Motivación

Las actividades formativas desarrolladas en una asignatura incluida en los planes de estudio adaptados al EEES deben seguir unos procesos de enseñanza-aprendizaje orientados a la adquisición de competencias, ya sean generales o específicas de una materia. Dentro de las diferentes actividades realizadas en estos procesos, están las prácticas de laboratorio, orientadas a poner en práctica los conocimientos desarrollados en las clases teóricas. Según la guía docente de la asignatura IPC del Grado en Ingeniería Informática de la Universidad de Valladolid, la competencia específica más importante es la *Capacidad para diseñar y evaluar interfaces persona computador que garanticen la accesibilidad y usabilidad a los sistemas, servicios y aplicaciones informáticas*. La guía docente incluye a su vez competencias generales a todas las asignaturas (expresión oral, trabajo en equipo), que también se trabajan en las actividades de las prácticas de laboratorio. El cumplimiento de los objetivos definidos para alcanzar esta competencia se relaciona tanto con la docencia teórica desarrollada en las clases magistrales como con la docencia y actividades incluidas en las prácticas de laboratorio.



2. JSwing

2.1 Objetivos de aprendizaje

El objetivo general de este capítulo que el o la estudiante aprendan los conceptos básicos de la librería de Java JSwing, incluyendo los componentes gráficos que proporciona la librería y la gestión de eventos:

- Objetivo específico 1: Crear una interfaz gráfica de usuario utilizando los componentes de JSwing, mediante el uso del IDE Netbeans.
- Objetivo específico 2: Implementar funcionalidad asociada a los diferentes eventos producidos por el usuario.

Tiempo estimado: Dos horas

2.2 Requisitos previos

Para la realización de este tutorial es necesario tener conocimientos básicos de programación orientada a objetos en lenguaje Java.

2.3 Introducción

Java Swing (JSwing) [cita] es una biblioteca para el desarrollo de aplicaciones con interfaz gráfica de usuario (GUI) en el lenguaje de programación Java. Esta biblioteca se basa en AWT, un kit de herramientas más antiguo. Está totalmente escrita en Java y permite crear interfaces adaptadas a cada sistema operativo sin cambiar el código. Incluye clases que permiten crear interfaces con los componentes básicos que se pueden encontrar en cualquier GUI (campos de texto, checkboxes, botones, desplegables,

etc), permite distribuir estos elementos en contenedores y implementa mecanismos para capturar los eventos que produce el usuario sobre los componentes.

2.4 Contenedores

Las clases contenedor son clases que pueden incluir componentes visuales, como botones, etiquetas, checkboxes, etc. En todas las aplicaciones construidas con JSwing se debe tener al menos un contenedor. Existen dos tipos de contenedores:

- Contenedores superiores (Figura 2.1). Son los contenedores principales de la aplicación, y cada aplicación tendrá un contenedor superior. Existen tres tipos principales: JFrame (ventana principal, la más utilizada), JDialog (pop up) y JApplet (para aplicaciones de navegador).
- Contenedores intermedios: Son contenedores que están dentro de uno superior y permiten contener otros componentes. Los principales contenedores intermedios son JPanel, JScrollPane o JSplitPane, pero existen muchos más.

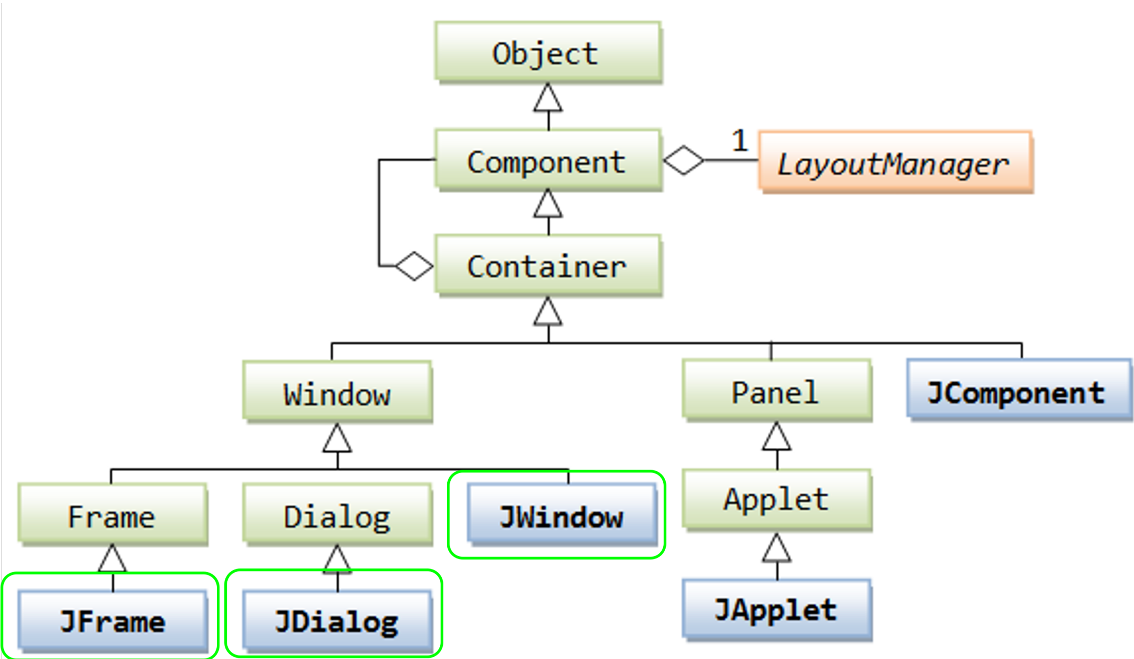


Figura 2.1: Jerarquía de clases de los contenedores jSwing. Extraída de `https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI_2.html`

2.5 Componentes

Los componentes son los diferentes elementos gráficos que se pueden añadir a la interfaz para interactuar con el usuario (ver Figura 2.3). Existen multitud de componentes en JSwing [Gar23; Ora23a], en este tutorial destacaremos los más utilizados. Todos

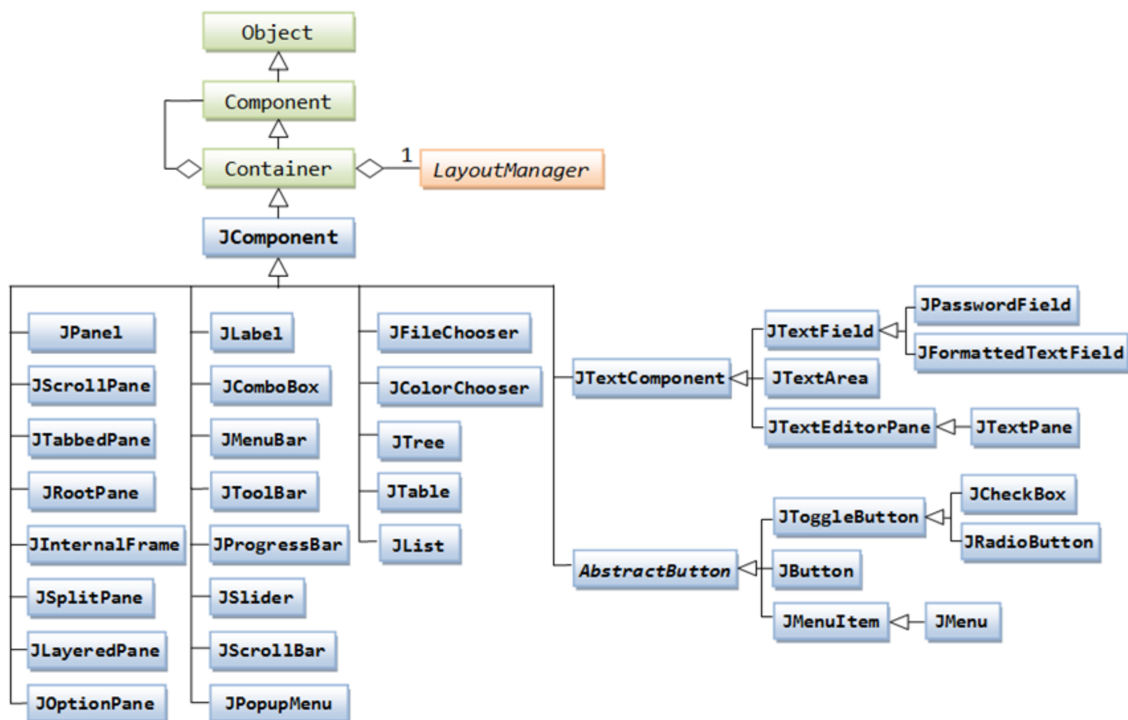


Figura 2.2: Jerarquía de clases de los componentes jSwing. Extraída de `https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI_2.html`

los componentes heredan de la clase JComponent, por lo que tienen características comunes que se pueden modificar en todos los componentes, como color, tamaño, etc (Figura 2.2). A continuación se detallan los principales componentes:

- JButton: Son botones que se pueden pulsar con el ratón. El uso más común de este componente es responder a clicks del usuario para desencadenar acciones, aunque se puede utilizar para otros usos. Existe una variante denominada ToggleButton que permite tener botones en dos estados: pulsado o no pulsado.
- JLabel: Son etiquetas de texto. Muy útiles para mostrar información al usuario.
- JCheckbox: Son elementos que pueden estar en dos estados: marcado o no marcado. Son útiles para selecciones binarias.
- JRadioButton o ButtonGroup: Son botones circulares que permiten que el usuario elija entre un conjunto limitado de opciones. Normalmente se utilizan cuando sólo se requiere elegir una opción.
- JTextField: Son elementos pensados para que el usuario introduzca información breve por teclado, aunque también se pueden utilizar para mostrar información al usuario.
- JTextArea: Similar a los JTextField pero con un tamaño mayor. Pensados para que el usuario introduzca mucha información a través del teclado.
- JList: Son elementos que permiten mostrar información al usuario y que esa información pueda ser seleccionada. Permiten además seleccionar múltiples elementos. Son útiles cuando necesitamos que el usuario puede seleccionar información

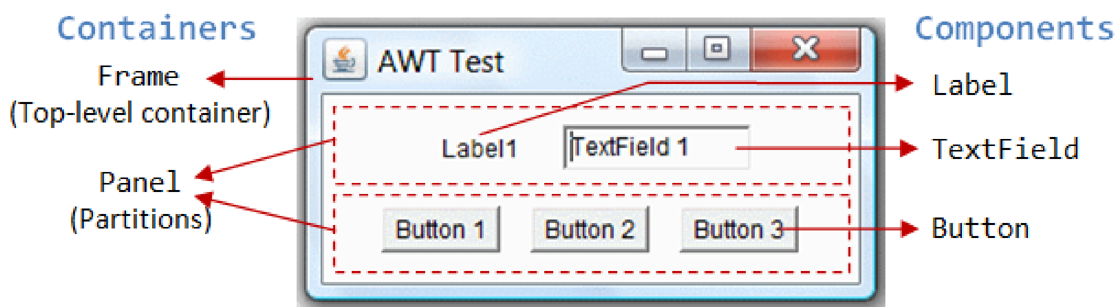


Figura 2.3: Componentes y contenedores de JSwing. Extraído de `https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html`

en un rango limitado de opciones.

- JComboBox: Similartes a los JList pero en formato desplegable. Útiles para la selección de información cuando el espacio que puede ocupar la interfaz el limitado.
- Menús: JSwing también permite crear diferentes tipos de menús, de forma similar a los que podemos encontrar en cualquier aplicación. Estos menús son útiles para agrupar funcionalidades y dar un acceso rápido a las mismas.

2.6 Layout Manager

Todos los contenedores superiores e intermedios tienen un LayourManager, que determina como se distribuirán los componentes en el contenedor. Existen multitud de ellos, pero los principales son:

- AbsoluteLayout: Permite disponer los elementos en coordenadas x e y específicas. No es recomendable su uso ya que al redimensionar la ventana se pueden descolocar los elementos.
- BorderLayout: Este layout divide el contenedor donde se aplica en cinco secciones: norte, sur, este, oeste y centro. El componente añadido ocupará todo el espacio disponible en la parte donde se ha introducido (Figura 2.4).
- BoxLayout: Este layout dispone los componentes añadidos en una columna o fila. Respecta el tamaño máximo definido para los componentes y permite alinearlos (Figura 2.5).
- FlowLayout: Este layout distribuye los componentes en un fila si hay espacio suficiente. Si no lo hay, crea nuevas filas en las que distribuye los componentes de forma uniforme según el espacio que necesita cada componente (Figura 2.6).
- GridLayout: Este layout distribuye los componentes uniformemente en forma de tabla. Se pueden indicar el número de filas y columnas de esta distribución, y cada elemento ocupará el espacio completo de la celda correspondiente (Figura 2.7).

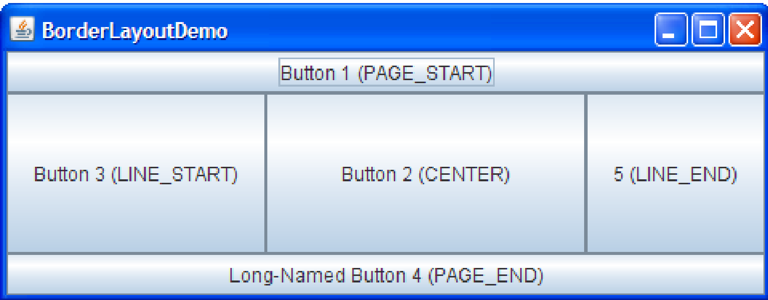


Figura 2.4: BorderLayout

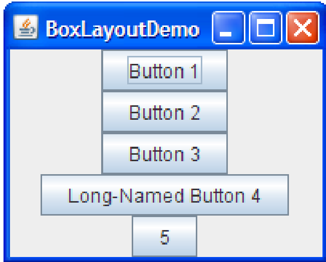


Figura 2.5: BoxLayout

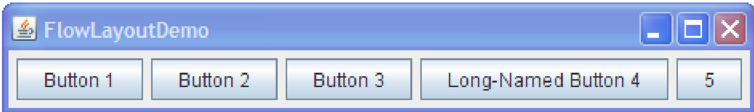


Figura 2.6: FlowLayout

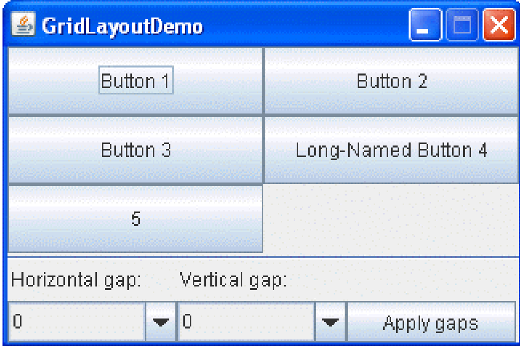


Figura 2.7: GridLayout

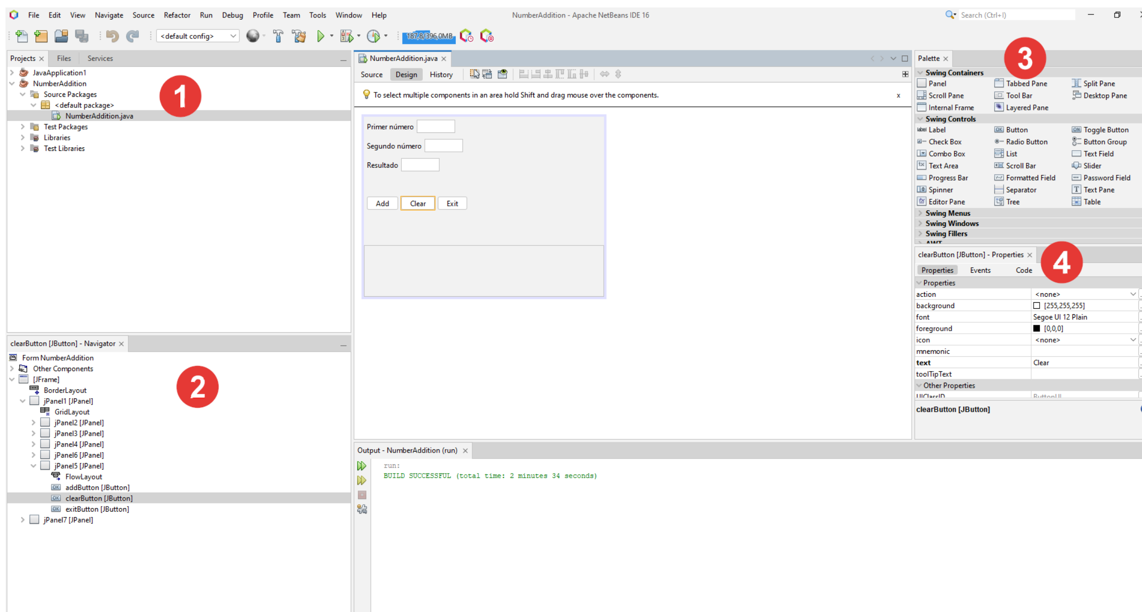


Figura 2.8: Espacio de trabajo en Netbeans

2.7 Netbeans

Netbeans [Apa23] es un entorno de desarrollo integrado (IDE) que permite desarrollar aplicaciones en diferentes lenguajes de programación. En relación con JSwing y Java, la ventaja de Netbeans frente a cualquier otro IDE es que integra un plugin específico para desarrollar aplicaciones gráficas con JSwing, utilizando un editor que permite desarrollar la interfaz mediante "drag and dropçon interfaz visual, además de auto generar mucho código, sobre todo el relacionado con la colocación de los elementos en la pantalla.

Para descargar Netbeans hay que acudir a su página oficial [Apa23] y descargar la última versión (en este tutorial, se utiliza la versión 16). Además, se necesita un jdk de java (JDK 11 en este tutorial). Por último, hay que indicar a Netbeans que utilice el JDK descargado. Esto se puede realizar modificando la variable de entorno JAVA_HOME o la hora de instalarlo. El plugin para utilizar JSwing de forma visual ya viene instalado por defecto. En al Figura 2.8 podemos ver las principales vistas que ofrece Netbeans para trabajar con JSwing. A continuación, se describen las principales vistas de Netbeans:

- Ventana Projects (1): Esta ventana muestra los proyectos abiertos que están en el workspace de Netbeans.
- Ventana Navigator (2): Muestra los diferentes contenedores y componentes de la aplicación que estamos desarrollando.
- Ventana Palette (3): Muestra los diferentes contenedores y componentes disponibles en JSwing.
- Ventana Properties (4): Muestra las propiedades del componente seleccionado. En esta ventana se puede cambiar el color, los eventos asociados, el tamaño, etc.

2.8 Gestión de eventos

Todos lo componentes de JSwing tiene asociados una serie de eventos que se disparan por acciones del usuario o de la propia aplicación. Algunos eventos son generales a todos los componentes de JSwing, y otros son específicos de cada componente. No todos los eventos necesitan ser tenidos en cuenta por la aplicación y, generalmente, solo se implementan algunos. Para responder a un evento, hay que seguir estos pasos (figura 2.9 y listado 2.1):

1. Añadir el *escuchador* del evento al componente concreto, pasando como parámetro un objeto de clase interna anónima que maneja los eventos de los componentes.
2. Implementar el código que responde al lanzamiento de ese evento sobre el componente elegido.

La clase Listener concreta será diferente para cada tipo de evento. Existe un evento general denominado ActionListener que implementa un evento por defecto dependiendo del tipo de componente. Por ejemplo, si añadimos un ActionListener a un botón, se implementará el evento de click izquierdo. En la tabla 2.1 se resumen los principales eventos específicos existentes. Se puede consultar la lista de todos los posibles eventos en la documentación oficial de JSwing [Ora23c]. Además, podemos obtener información del evento a través de los métodos de la clase *ActionListener*, como si se ha pulsado alguna tecla modificadora (Ctrl, Alt) al producirse el evento.

Figura 2.9: Ejemplo de ActionListener

Evento	Listener type
El usuario clica un botón, presiona Enter, escribe un texto	ActionListener
El usuario cierra un frame	WindowListener
El usuario pulsa un botón del ratón mientras el cursor está en un componente	MouseListener
El usuario mueve el ratón sobre un componente	MouseMotionListener
Un componente se hace visible	ComponentListener
Cambia la selección de una lista	ListSelectionListener
Cambia el contenido de un elemento	ChangeListener

Tabla 2.1: Resumen de los principales eventos y su clase Listener asociada

```
private void sumarBotonActionPerformed
                        (java.awt.event.ActionEvent evt)
{
      float num1, num2, result;
      num1 = Float.parseFloat(primerNumero.getText());
      num2 = Float.parseFloat(segundoNumero.getText());
      result = num1 + num2;
      resultado.setText(String.valueOf(result));
 }
```

Listado 2.1: Código de respuesta al evento de click

2.9 Ejemplo de aplicación con Netbeans

En esta sección se explica como desarrollar una aplicación sencilla con JSwing utilizando el IDE Netbeans. La aplicación a desarrollar consiste en sumar dos números introducidos en dos TextField y mostrar el resultado en otro. Además, se almacena en un historial todas las sumas realizadas y se muestran en un JTextArea.

2.9.1 Paso 1: crear un proyecto

El primer paso es crear un proyecto para desarrollar la aplicación. Para crear el proyecto, hay que seguir los siguientes pasos (Figura 2.10 y Figura 2.11):

1. Selecciona File >New Project.
2. Selecciona Java with Ant >Java Application.
3. Escribe NumberAddition como nombre del proyecto.
4. Deselecciona el checkbox *Create Main Class*, ya que el propio JFrame ya tiene un método main por defecto.
5. Pulsa *Finish*

2.9.2 Paso 2: crear la interfaz

Para crear una interfaz utilizando JSwing, vamos a utilizar el contenedor JFrame, que es el más habitual para construir aplicaciones de escritorio. También vamos a añadir los

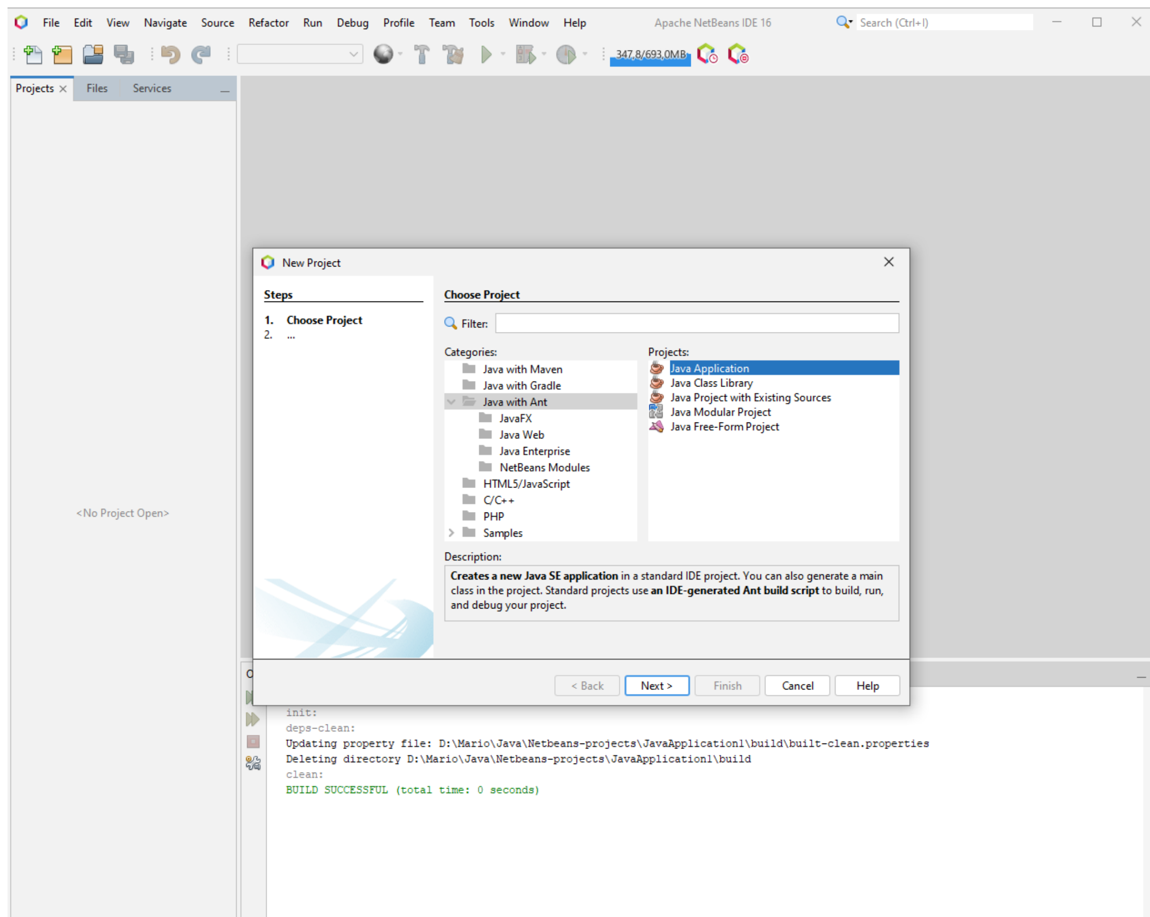


Figura 2.10: Creación de un proyecto (paso 1)

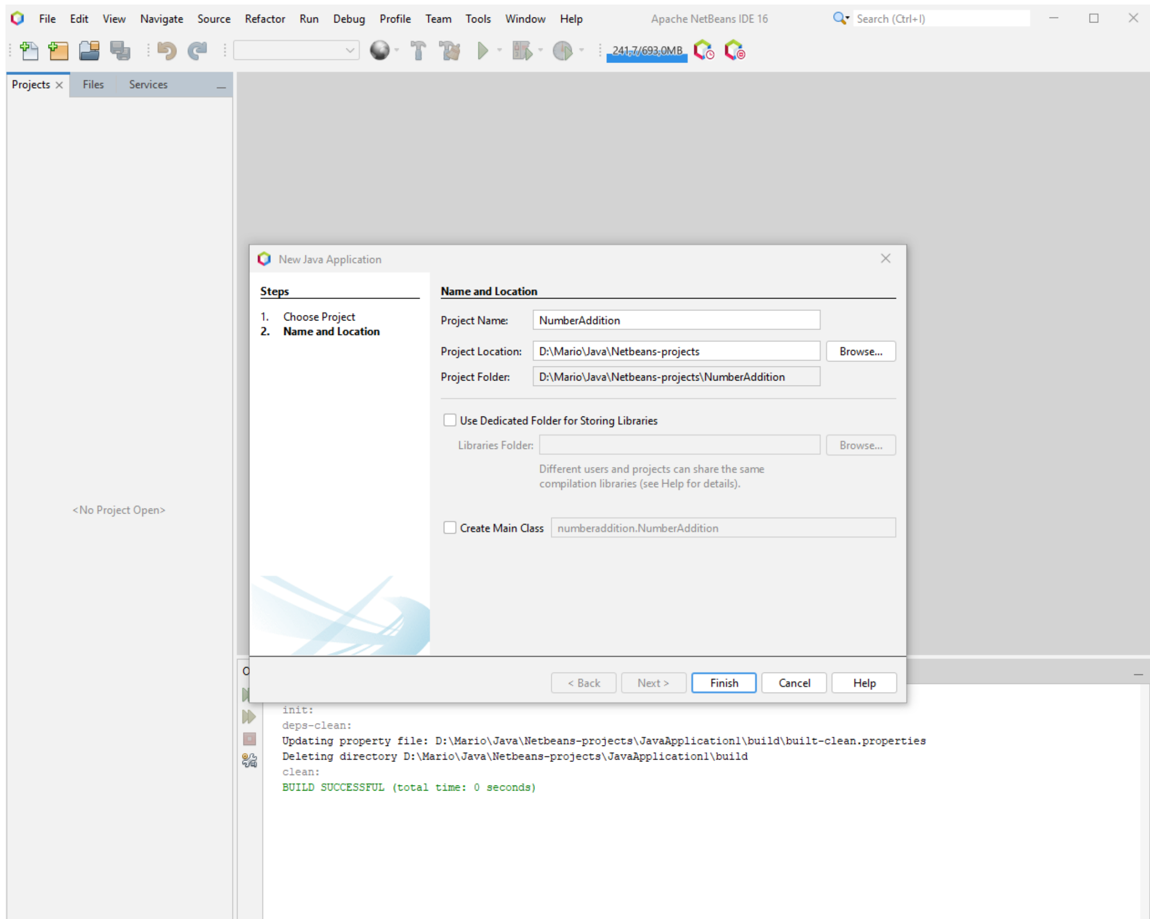


Figura 2.11: Creación de un proyecto (paso 2)

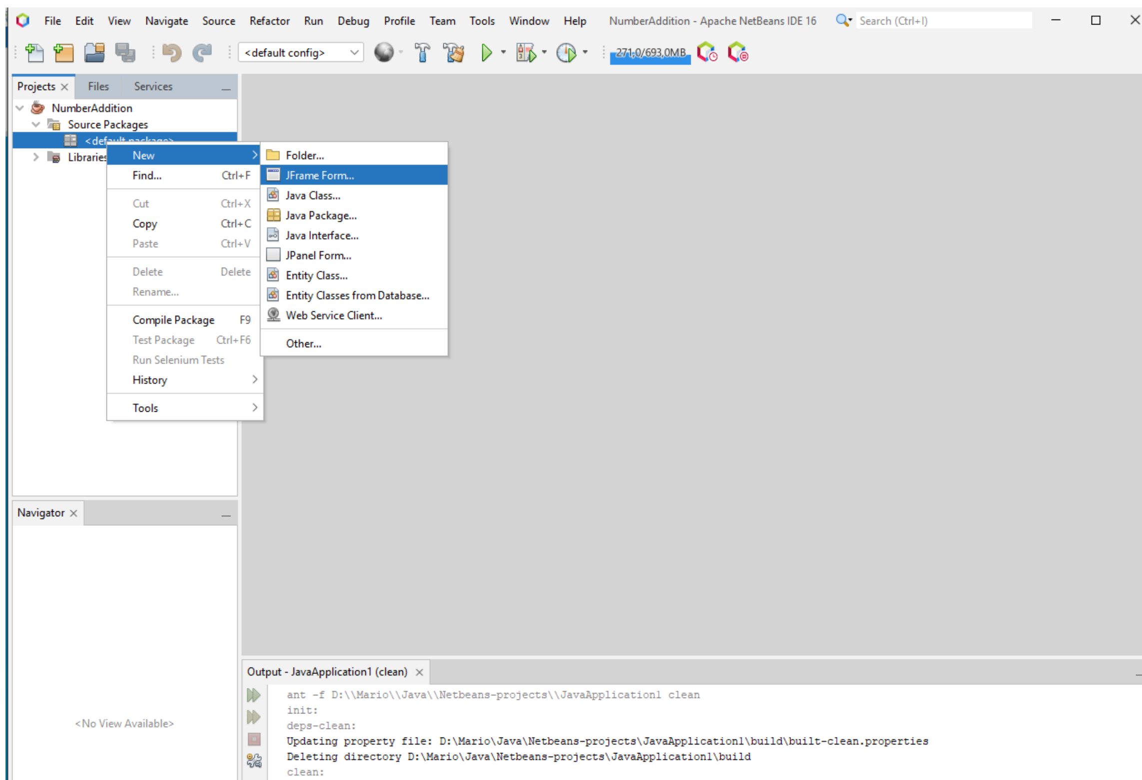


Figura 2.12: Creación de una clase JFrame

diferentes componentes que forman parte de nuestra aplicación (Figura 2.12).

- En la ventana de proyectos, hacemos click derecho en el paquete donde vamos a crear la clase y pulsamos New >JFrame Form
- Introduce NumberAddition como nombre de la clase.
- Pulsa *Finish*.

Ahora vamos a seleccionar un layout manager para nuestro JFrame. Pulsamos botón derecho sobre el JFrame en la vista de Navigator >Set Layout >BorderLayout. Este layout divide la ventana principal en 5 zonas. Arrastramos desde el panel de Pallete un Panel en la parte norte, y un Panel en la parte sur. Al Panel superior le asignamos un GridLayout de 5 filas, y añadimos 5 paneles. Estos paneles contendrán los diferentes elementos gráficos que se pueden ver el la Figura 2.13. Estos paneles tienen asignado un FlowLayout. En el Panel de abajo añadimos un TextArea que utilizaremos para mostrar un historial de las sumas realizadas.

2.9.3 Paso 3: Cambiar el texto y el nombre de las variables

Ahora vamos a cambiar el texto que se muestra en los Labels y en los TextField pulsando botón derecho sobre el componente >Edit Text o cambiando la propiedad text en la vista de Properties. Además, Netbeans por defecto crea las variables asociadas a los componentes como *JNombreDelComponenteIndice*, lo que no es una buena prác-

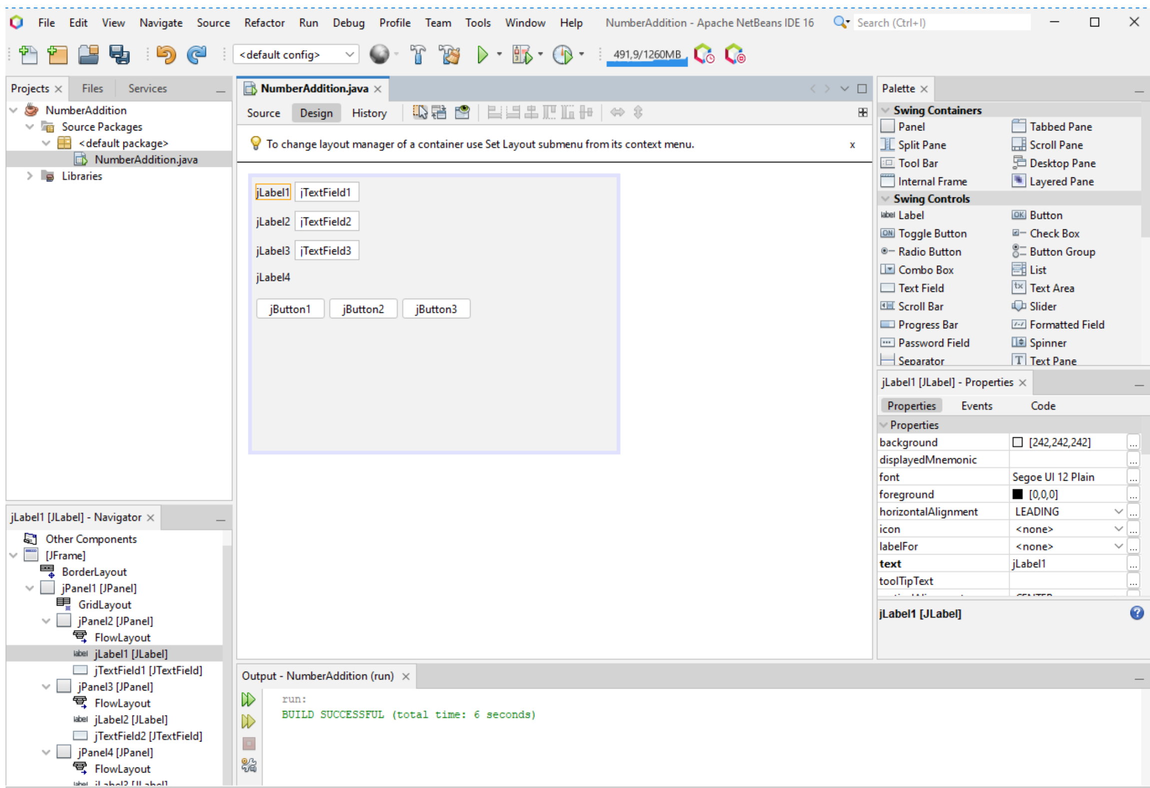


Figura 2.13: Aplicación con los diferentes componentes añadidos

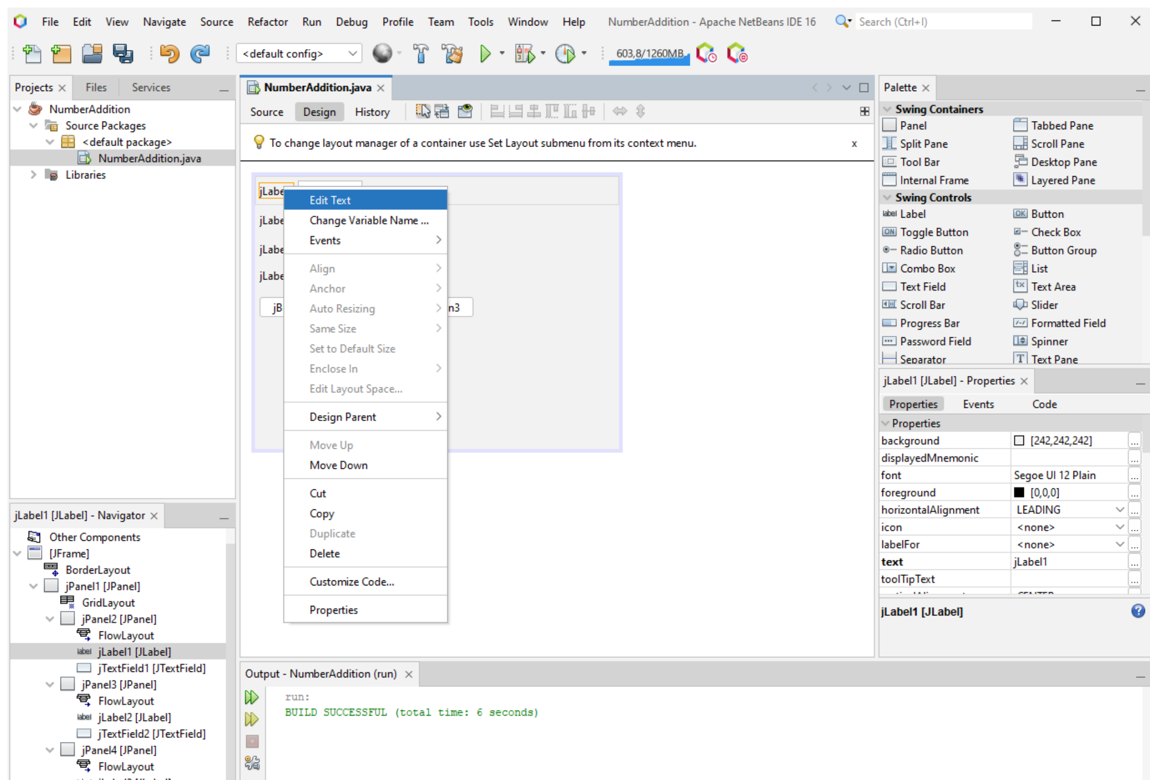


Figura 2.14: Editar texto de los componentes añadidos

tica, ya que, al crecer el número de componentes, puede ser difícil identificar que componente es el que está relacionado con dicha variable. Para cambiar el nombre de un componente, hacemos click derecho sobre el componente >Change Variable Name o en la vista de Properties (ver Figuras 2.14 y 2.15) Los nombres de las variables deben seguir las convenciones de nombres de Java [Ora23b]. En este tutorial nombraremos las variables siguiendo la siguiente estructura: nombre explicativo del componente+clase del componente, por ejemplo, *salirButton*, *primerNumeroTextField*, etc.

2.9.4 Paso 4: añadir funcionalidad

En este paso, vamos a añadir la funcionalidad a cada botón de la interfaz. Comenzamos por el botón *Clear*, que simplemente borra el contenido de los TextField. Para ello, hacemos click derecho sobre el botón >Events >Action >actionPerformed (ver Figura 2.16). Como se ha comentado anteriormente, para los botones el evento *Action* general se refiere al click en el botón izquierdo del ratón. Netbeans automáticamente genera el código para añadir el listener al botón y crea un método para introducir el código que responde al evento. Por último, añadimos el siguiente código a nuestra aplicación (listado 2.2):

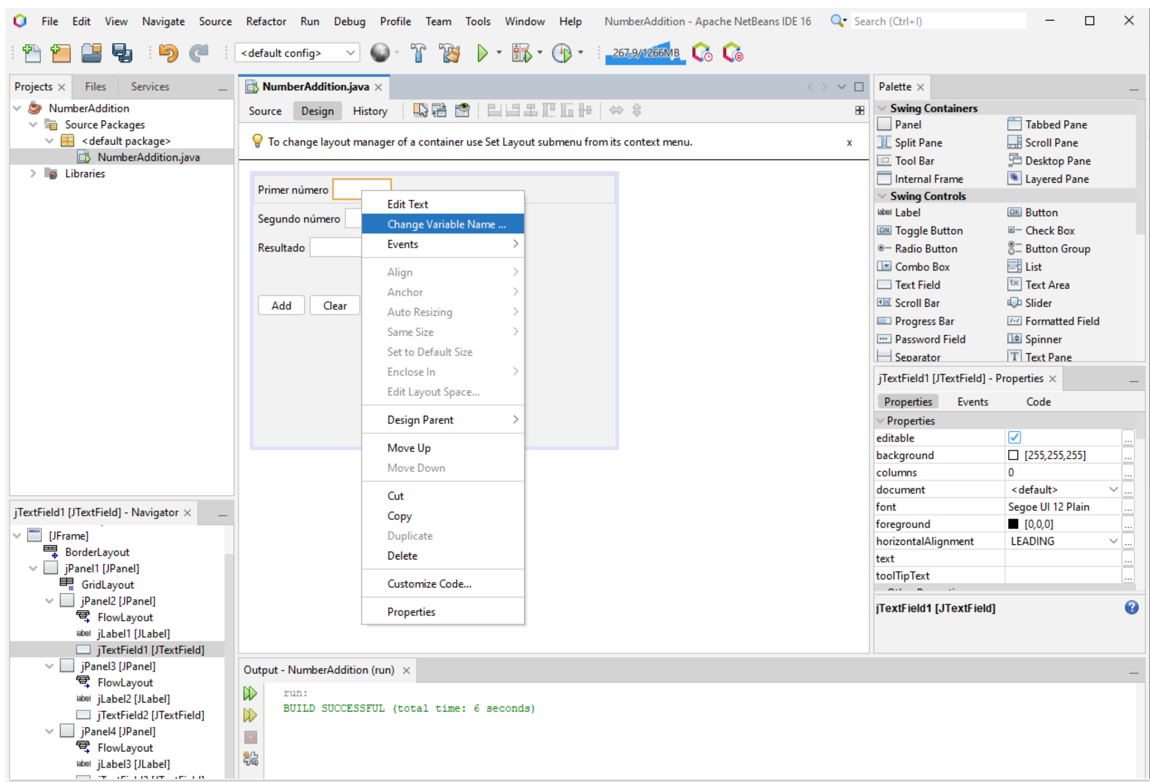


Figura 2.15: Editar nombre de las variables

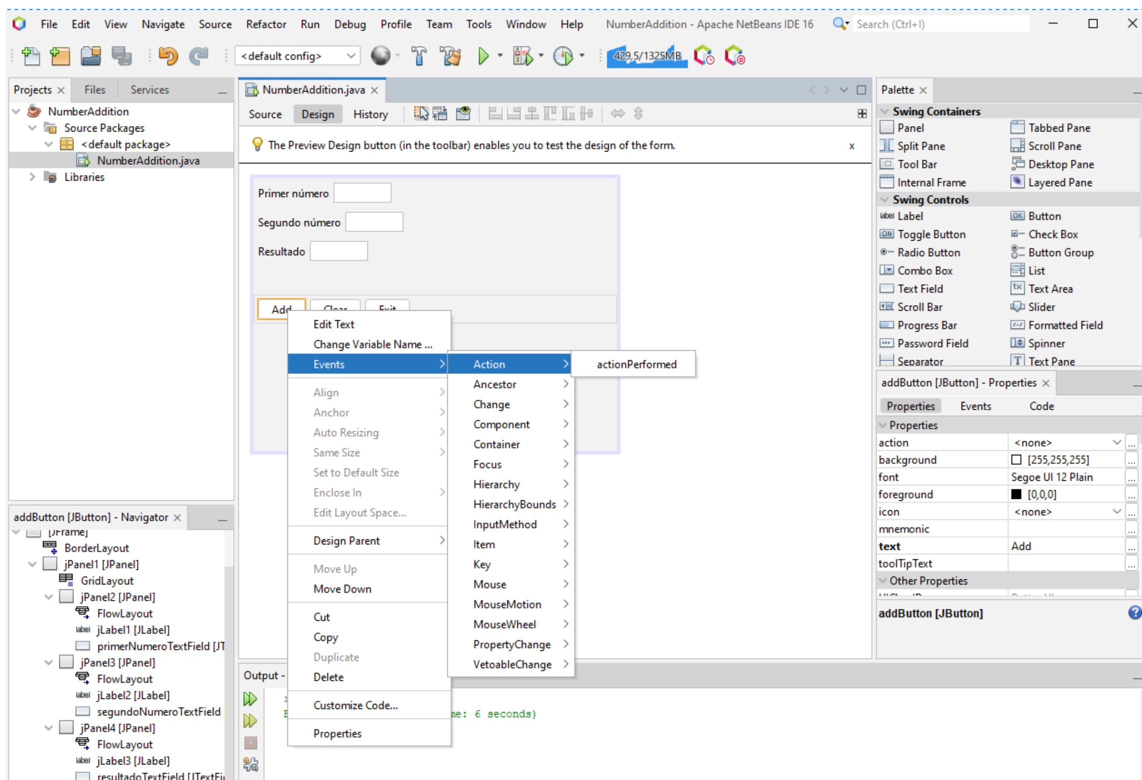


Figura 2.16: Añadir eventos a los componentes

```
private void clearButtonActionPerformed
(java.awt.event.ActionEvent evt) {
        primerNumeroTextField.setText("");
        segundoNumeroTextField.setText("");
        resultadoTextField.setText("");
    }
```

Listado 2.2: Código de respuesta al evento de click sobre el botón clear

Hacemos lo mismo para el botón *Exit*, añadiendo el siguiente código que cerrará la aplicación (listado 2.3):

```
private void exitButtonActionPerformed
(java.awt.event.ActionEvent evt) {
        System.exit(0);
    }
```

Listado 2.3: Código de respuesta al evento de click sobre el botón Exit

El botón *Add* es el principal de nuestra aplicación de ejemplo y responderá al click sumando los valores de los TextField y mostrándolos en el TextField de resultado. Para ello, tenemos que obtener los valores de los TextField y convertirlos a double, para poder sumarlos y mostrar el resultado. Por último, añadiremos la suma realizada a la variable que almacena el historial y actualizamos el JTextArea donde se muestra dicho historial (listado 2.4):

```
       private void addButtonActionPerformed
       (java.awt.event.ActionEvent evt) {

        double num1, num2, result;

        num1 = Float.parseFloat(primerNumeroTextField.getText());
        num2 = Float.parseFloat(segundoNumeroTextField.getText());
        result = num1 + num2;
        resultadoTextField.setText(String.valueOf(result));
        historial.add(num1+"+"+num2);
        actualizarHistorial();
    }
```

Listado 2.4: Código de respuesta al evento de click sobre el botón Add

Para ejecutar nuestra aplicación, sólo es necesario pulsar sobre el botón *Play*.

2.9.5 Paso 5: tratamiento de errores

Tenemos una aplicación con una funcionalidad definida y que funciona. Pero, ¿qué pasa si lo que introducimos en los TextFields no es un número?. Con la implementación actual, nos salta una excepción *NumberFormatException* al intentar convertir un String que no es un número a double (ver Figura 2.17). Por ello, hay que añadir algún tipo de validación y tratamiento de errores, no solo para evitar que se produzcan errores

en el código que hagan que nuestra aplicación no funcione correctamente, si no para informar al usuario del error que se ha producido y que pueda corregirlo. Para ello añadiremos la gestión de excepciones propias de Java a través de try catch y mostraremos un mensaje de error al usuario (ver Figura 2.18). El código actualizado es el siguiente (listado 2.5):

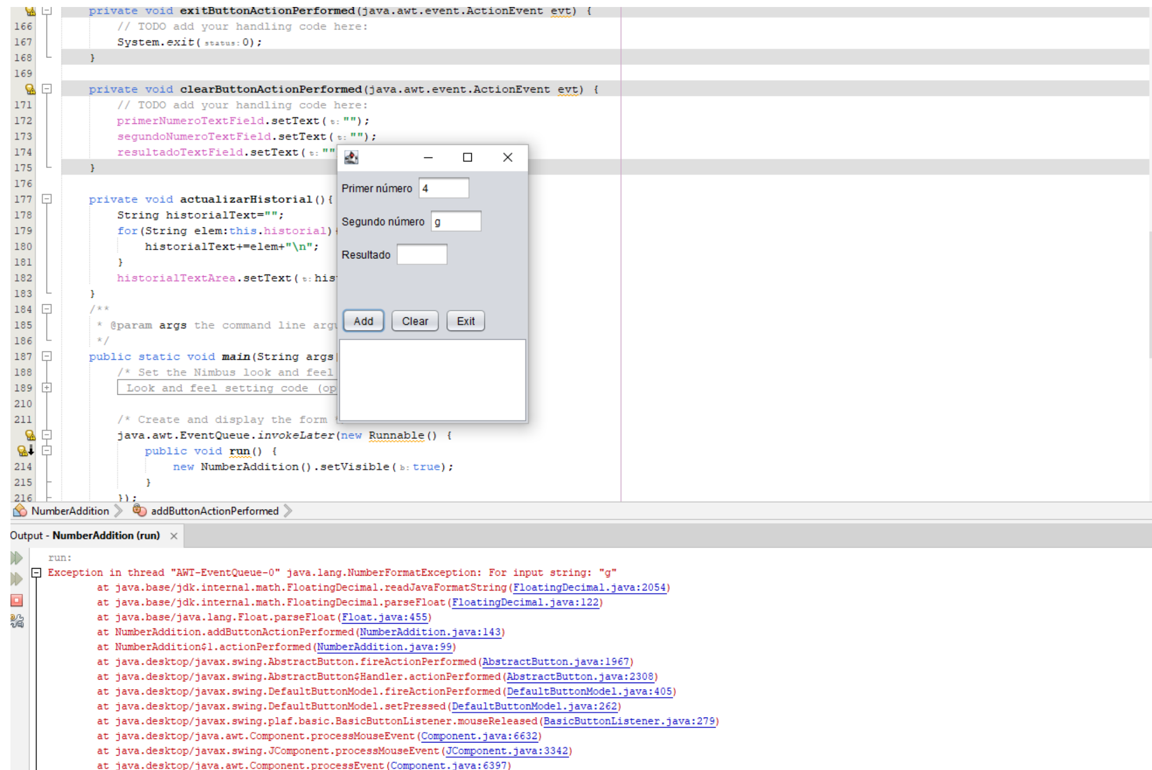


Figura 2.17: Excepción producida al intentar convertir un string no numérico a double

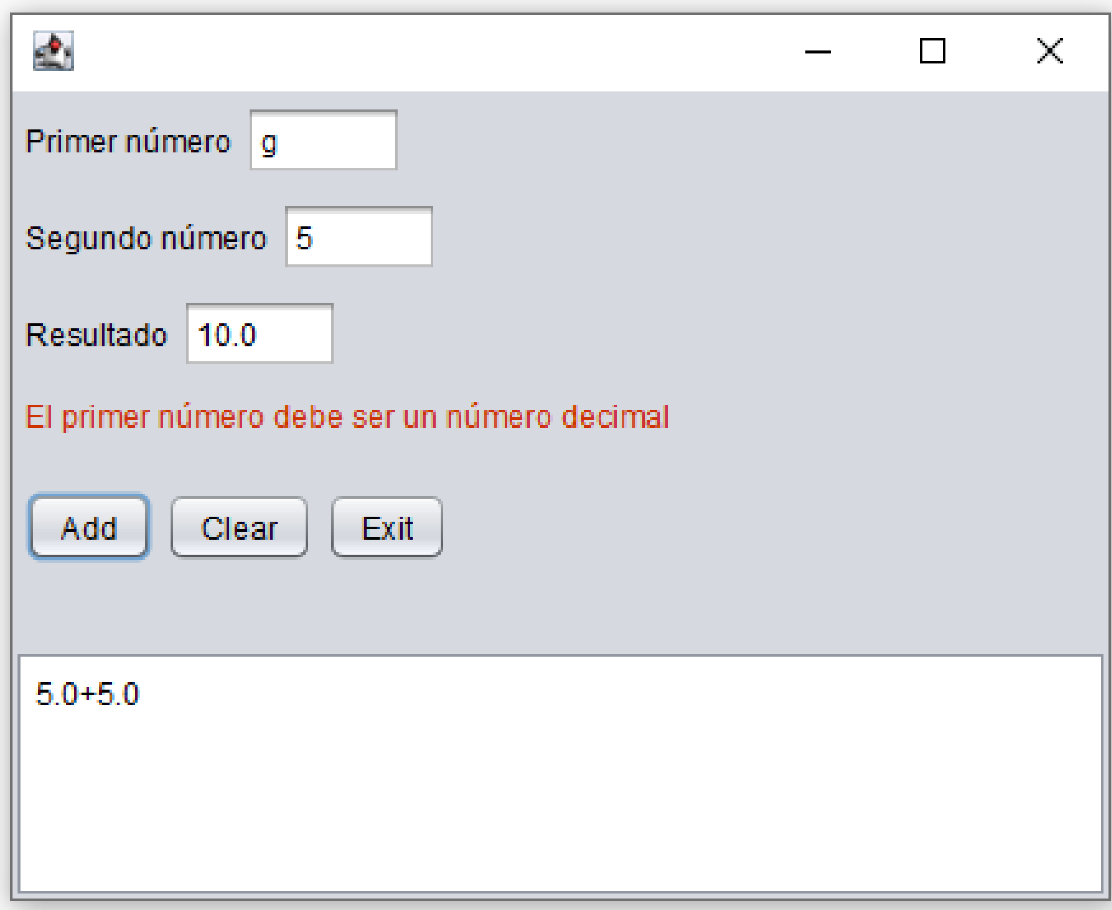


Figura 2.18: Tratamiento de errores añadido a la interfaz

```
private void addButtonActionPerformed
(java.awt.event.ActionEvent evt) {
        double num1, num2, result;

        try{
            num1=Double.parseDouble(
            primerNumeroTextField.getText());
        }
        catch(NumberFormatException e){
            errorLabel.setText(
            "El primer numero debe ser un numero decimal");
            return;
        }
        try{
            num2=Double.parseDouble(
            segundoNumeroTextField.getText());
        }
        catch(NumberFormatException e){
            errorLabel.setText(
            "El segundo numero debe ser un numero decimal");
```

```
            return;
        }
        result = num1 + num2;
        resultadoTextField.setText(String.valueOf(result));
        historial.add(num1+"+"+num2);
        actualizarHistorial();
    }
```

Listado 2.5: Código de respuesta al evento de click sobre el botón Add con captura de excepciones

2.10 Ejercicio propuesto

Para practicar con todos los elementos que se han mencionado, se propone crear una aplicación que consiste en un gestor sencillo de facturas, cuya interfaz puede verse en la Figura 2.19. A continuación se describen los componentes de la interfaz:

- Asunto (1): TextField para introducir el asunto de la factura. El asunto debe tener entre 1 y 10 caracteres.
- Fecha (2): Spinner para introducir día, mes y año de la expedición de la factura. La fecha no puede ser inferior al 1 de Enero de 2000 y debe cumplir los criterios de validez estándar de las fechas (no puede añadirse un factura con fecha 40 de Enero, por ejemplo).
- Cantidad (3): TextField para introducir la cantidad de la factura. Debe ser un número decimal mayor o igual que 0.
- Tipo (4): JComboBox para seleccionar el tipo de factura. Puede ser Empresas o Particulares.
- Lista de facturas (5): JList para mostrar la información de las facturas. Debe mostrarse toda la información separada por :.
- Mensajes (6): JLabel para mostrar información al usuario, como errores o mensajes de confirmación.
- Botón Añadir/Actualizar (7): JButton para añadir o actualizar un factura.
- Botón Editar (8): JButton para editar una factura.
- Botón Eliminar (9): JButton para eliminar una factura.

Las funcionalidades requeridas son:

- Añadir facturas: Se debe poder añadir una factura con la información definida anteriormente. No se puede añadir una factura si ya existe una con el mismo asunto. Al añadir una factura, se debe mostrar todos sus datos en la JList (5). Para añadir una factura, se debe pulsar sobre el botón *Añadir/Actualizar* (7). Si algún campo no es correcto, se debe informar al usuario del error y no introducir la factura en el sistema.
- Editar facturas: Se debe poder editar la información de las facturas ya añadidas. Si se pulsa el botón *Editar* (8) y existe una factura seleccionada en el JList, se deben cargar los datos actuales en los campos de arriba. Si no hay ninguna factura seleccionada, se debe informar al usuario del error.
- Eliminar facturas: Se debe poder eliminar facturas del sistema. Si se pulsa el botón

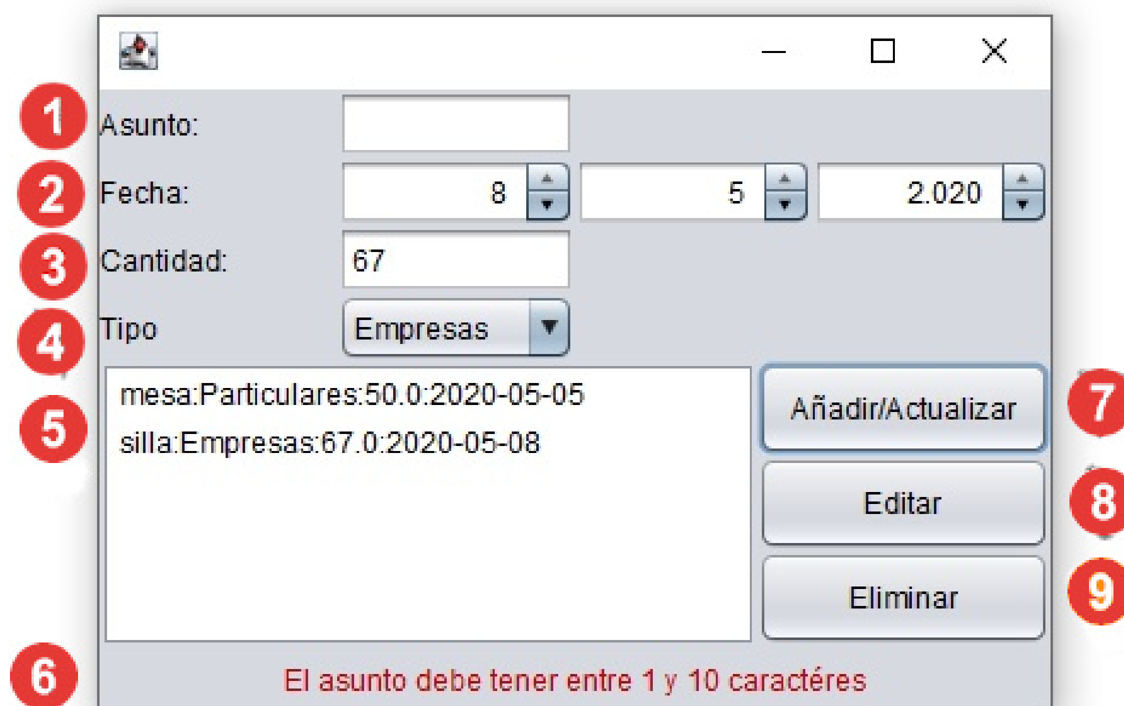


Figura 2.19: Interfaz propuesta

Eliminar (9) y existe una factura seleccionada en el JList, se debe eliminar dicha factura y desaparecer del JList. Si no hay ninguna factura seleccionada, se debe informar al usuario del error.

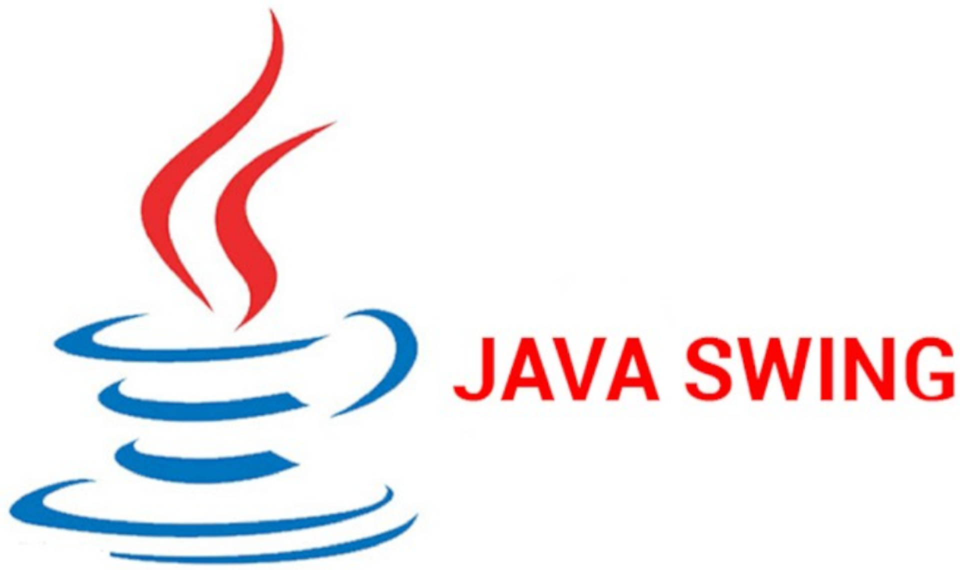
- Actualizar factura: Se debe poder almacenar la información de la factura en edición. Si se pulsa el botón *Añadir/Actualizar* (7) y existe una factura seleccionada en el JList, se debe actualizar la información de esa factura y actualizar la información mostrada en el JList. Si algún campo no es correcto, se debe informar al usuario del error y no se actualiza la información de la factura en el sistema.

Para realizar el ejercicio propuesto, se recomienda seguir los siguientes pasos:

1. Crear un proyecto en Netbeans, llamado por ejemplo *GestorFacturas*.
2. Crear el contenedor (JFrame).
3. Añadir los paneles con sus respectivos layouts y los componentes necesarios, para obtener una interfaz similar a la de la Figura 2.19.
4. Dar la funcionalidad descrita previamente a los diferentes componentes de la interfaz.

2.11 Enlaces a proyectos de ejemplo

- Código de NumberAddition: `https://github.com/IPC-UVa/NumberAddition`
- Código de GestorFacturas: `https://github.com/IPC-UVa/Facturas`



3. Modelo-Vista-Controlador con JSwing

3.1 Objetivos de aprendizaje

El objetivo general de este capítulo que el o la estudiante aprendan los conceptos básicos del patrón arquitectónico *Modelo-Vista-Controlador* y su implementación utilizando la librería JSwing:

- Objetivo específico 1: Entender qué es y cómo se aplica el patrón *Modelo-Vista-Controlador* al desarrollo de aplicaciones con interfaz gráfica de usuario.
- Objetivo específico 2: Crear una interfaz gráfica de usuario utilizando los componentes de JSwing, aplicando el patrón *Modelo-Vista-Controlador* para organizar el código generado.

Tiempo estimado: Dos horas

3.2 Requisitos previos

Para la realización de este tutorial es necesario tener conocimientos básicos de programación orientada a objetos en lenguaje Java. También es necesario tener instalado el IDE Netbeans.

3.3 Modelo Vista Controlador

El modelo vista controlador (MVC) es un patrón arquitectónico de software orientado al diseño de aplicaciones con interfaz gráfica de usuario [Fow06; Jav23]. La idea de este patrón es dividir el código de nuestra aplicación en tres módulos distintos:

- Modelo: Se encarga de trabajar con los datos de la aplicación, normalmente ac-

cediendo a una fuente de datos persistente, como una base de datos, un fichero, etc
- Vista: Es la parte encargada de interactuar con el usuario, presentando la información y recibiendo los diferentes eventos (clicks, redimensionamiento de ventanas, pulsación en el teclado).
- Controlador: Es la parte encargada de hacer de intermediario entre la vista y el modelo, recibiendo y procesando los eventos de la vista y accediendo al modelo para modificarlo o acceder a su información. También se comunica con la vista para que actualice la información si es necesario.

La idea de este patrón es separar responsabilidades en nuestro código, siendo cada módulo responsable de una parte de la aplicación de manera que los objetos del Modelo (dominio) no tengan conocimiento directo de los objetos de la Vista (presentación). La principal ventaja de la aplicación del MVC es que los modelos y las vistas pueden cambiar de forma independiente. Además, un modelo podría ser utilizado por diferentes vistas y una vista se podría reutilizar para otros modelos. Por ejemplo, si tenemos una aplicación de venta de libros, podemos tener la misma fuente de datos en el modelo y utilizar diferentes vistas para acceder a la información, como una web, una aplicación móvil, un smartwatch, etc. Utilizando el mismo ejemplo, podríamos cambiar la fuente de datos desde una base de datos relacional (mysql) a una no relacional (MongoDB) sin tener que modificar la vista.

3.4 MVC activo vs MCV pasivo

Existen dos versiones ligeramente diferentes del patrón MVC [Bra19]. La diferencia se encuentra en la comunicación entre el modelo y la vista:

- MVC activo: En esta versión, el modelo notifica a la vista cuando se produce un cambio en su información asociada. De esta manera, la vista siempre estará actualizada con los últimos cambios del modelo, sin necesidad de la intervención del controlador. Para evitar un acoplamiento entre el modelo y la vista que va en contra de la filosofía de este patrón, existen variantes del MVC, como por ejemplo, la introducción del patrón Observador para realizar las notificaciones.
- MVC pasivo: Esta es la versión que desarrollaremos en este tutorial. En esta versión, el modelo no notifica los cambios ni a la vista ni al controlador, ya que no tiene acceso directo a ellos. Por lo tanto, es el controlador el encargado de gestionar la lógica de la aplicación para mantener siempre actualizada la vista. La principal desventaja de esta versión es que si el modelo sufre un cambio de estado, puede no reflejarse automáticamente en la interfaz.

El patrón MVC pasivo define un flujo de control en nuestro código que se refleja en los siguientes pasos (Figura 3.1):

1. El usuario realiza una acción en la vista que genera un evento (por ejemplo, hace click en un botón).
2. La vista notifica al controlador que se ha producido ese evento. Para ello, la vista tiene que tener una referencia al controlador. El controlador procesa el evento e

implementa la lógica asociada.

3. El controlador se comunica con el modelo para obtener o para actualizar la información. Para ello, puede comunicarse con la vista para pedir información de algún elemento gráfico.
4. (Opcional) El controlador se comunica con la vista para que se actualice. Si la vista necesita información para actualizarse, realiza una consulta al modelo. El modelo no tiene conocimiento directo de la vista y la vista sólo se comunica con el modelo para obtener información, nunca para modificarla.
5. La interfaz de usuario espera otra interacción del usuario, por lo que se repite el flujo.

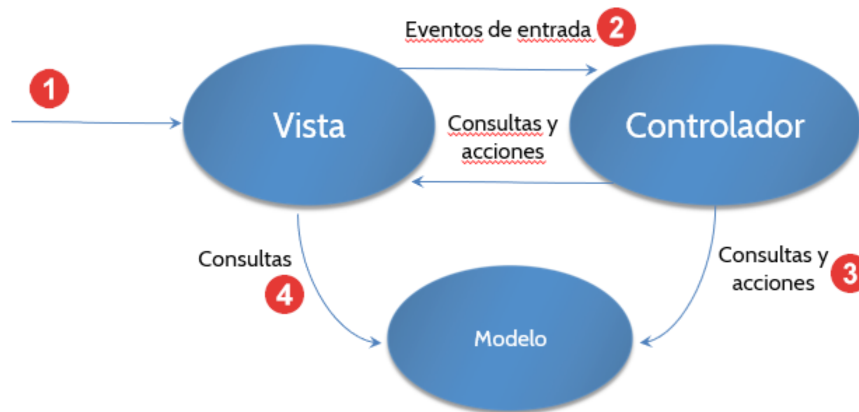


Figura 3.1: Diagrama de MVC pasivo

3.5 MVC en JSwing

En esta sección vamos a ver cómo aplicar el patrón MVC utilizando JSwing. Las vistas se desarrollan utilizando las diferentes clases que incluye JSwing para crear elementos gráficos y para el manejo de eventos que se producen sobre estos. Los controladores son clases Java que capturan y propagan los efectos de los eventos a las vistas y al modelo. Cada vista tiene asociado un controlador. El modelo puede acceder a la información a través de diferentes fuentes de datos (en memoria, ficheros, servicios web, bases de datos). En esta sección se explican los diferentes componentes de este patrón, y en la sección 3.7 se muestra un ejemplo de aplicación.

3.5.1 Vista

La vista es un contenedor superior en JSwing, normalmente un JFrame. La vista necesita tener referencia de su controlador para notificarle los eventos. La vista puede necesitar información del modelo, pero es el controlador el que le proporcionará refe-

rencia al modelo cuando lo necesite. Por lo tanto, la vista crea el controlador y se pasa a sí misma como parámetro (listado 3.1).

```
public class VistaNA extends javax.swing.JFrame {
    private ControladorNA miControl;

    public VistaNA() {
        initComponents();
        this.miControl=new ControladorNA(this);
    }
}
```

Listado 3.1: Código de la clase VistaNA. Se omite por simplicidad el código referido a la creación y disposición de los componentes gráficos

3.5.2 Controlador

El controlador es una clase Java que necesita tener referencia a la vista, para consultar la información de los elementos de la vista y para requerir actualizaciones, y al modelo, para obtener/actualizar la información. En aplicaciones más complejas, el acceso al modelo se haría desde otra clase distinta, pero por simplificar construiremos el modelo desde el controlador. El controlador puede mandar información a la vista para que la muestre. Esta información puede ser un tipo básico (int, String, double) o un objeto del modelo (listado 3.2).

```
public class ControladorNA {
    private VistaNA miVista;
    private ModeloNA miModelo;

    public ControladorNA (VistaNA v) {
        miVista = v;
        miModelo = new ModeloNA();
    }
}
```

Listado 3.2: Código de la clase ControladorNA

3.5.3 Modelo

El modelo no necesita conocer ni a la vista ni al controlador, ya que al usar MVC pasivo, es totalmente independiente. El modelo puede tener una o más clases, dependiendo de la cantidad y estructura de la información de la aplicación (listado 3.3).

```java
public class ModeloNA {

    public ModeloNA() {

    }
}
```

Listado 3.3: Código de la clase ModeloNA

3.6 Guías de aplicación del patrón MVC

En esta sección se detallan algunas guías de aplicación así como puntos clave en los que fijarse cuando estamos construyendo una aplicación usando MVC:

- Para construir el modelo, es útil pensar en qué información tendría que almacenar la aplicación en algún medio persistente (bases de datos, ficheros), aunque la aplicación no lo requiera directamente.
- La vista NUNCA tiene que actualizar el modelo. La vista solo accede al modelo para consultar información.
- La vista no contiene lógica de la aplicación.
- La lógica de negocio va en el modelo (comprobación de valores en un rango, valores repetidos, etc).
- Ni el modelo ni el controlador deben conocer los detalles de implementación de la vista, es decir, no saben cómo se muestra/introduce la información. Cambiar componentes de la vista no debe suponer cambios en el modelo ni en el controlador. Para esto es útil revisar los imports que realizan nuestros modelos y nuestros controladores, ya que no deben importar clases específicas de la vista (en nuestro caso, clases de JSwing).
- La vista no debe pasar información al controlador a través de los métodos de procesamiento de los eventos, salvo excepciones que ayuden a reducir el código generado. El controlador debe preguntar a la vista sobre información concreta cuando lo necesite.

3.7 Ejemplo

En este ejemplo, haremos la conversión de la aplicación NumberAddition realizada previamente (ver sección 2.9) para seguir la arquitectura MVC. En esa aplicación, todo el código generado está en una única clase, que hereda de JFrame (Figura 3.2). De esta manera, todas las responsabilidades son asumidas por la misma clase. Para realizar la conversión, lo más importante es definir qué información tiene que almacenar el modelo, para separar ese código de la vista.

Lo primero que vamos a hacer es crear tres paquetes en nuestro proyecto, uno para separar la entrada al programa a través del método main, otro para las clases del

modelo y otro para la vista y el controlador. Además, vamos a crear las clases correspondientes. Por un lado, creamos la clase *Main*, que contiene el método main y define la entrada a la aplicación. También creamos la clase del modelo, que llamaremos *ModeloNA*. Por último, creamos la clase controlador *ControladorNA* y copiamos la clase *NumberAdditionUI*, que renombraremos como *VistaNA* (Figura 3.3). Ahora copiamos el método main de la clase *VistaNA* a la nueva clase *Main*, y eliminamos el método main de *VistaNA* (listado 3.4).

```
public class Main {
    public static void main(String args[]) {

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new VistaNA().setVisible(true);
            }
        });
    }
}
```

Listado 3.4: Código de la clase Main

3.7.1 Paso 1: definir el Modelo

Vamos a definir el modelo, es decir, qué información maneja nuestra aplicación. Nuestra aplicación de ejemplo permite introducir dos números para ser sumados, muestra el resultado de la suma y almacena y muestra todas las sumas realizadas. También tenemos una serie de mensajes que se le muestran al usuario cuando se produce algún error.

El modelo contiene la lógica de negocio y los datos de la aplicación. La clase *ModeloNA* contiene un método para realizar la suma, otro para devolver el resultado, uno para añadir una suma al historial y otro para devolver este historial (listado 3.5). Para estructurar mejor la información, se crea la clase (*EntradaHistorial*) para almacenar los sumandos, con sus respectivos *setters* y *getters* (listado 3.6). En este ejemplo, no utilizaremos ningún medio persistente para almacenar la información, la almacenaremos en memoria como objetos, por lo que la información se perderá al cerrarse la aplicación. Con respecto a los mensajes de error, la mejor solución es que estuvieran también en el modelo, ya que nos permitiría cambiar esos textos sin modificar la vista o tener la aplicación en varios idiomas y elegir uno con algún parámetro de configuración. Para simplificar, en este ejemplo los mensajes se guardan en el controlador y este se los pasa la vista para que los muestre.

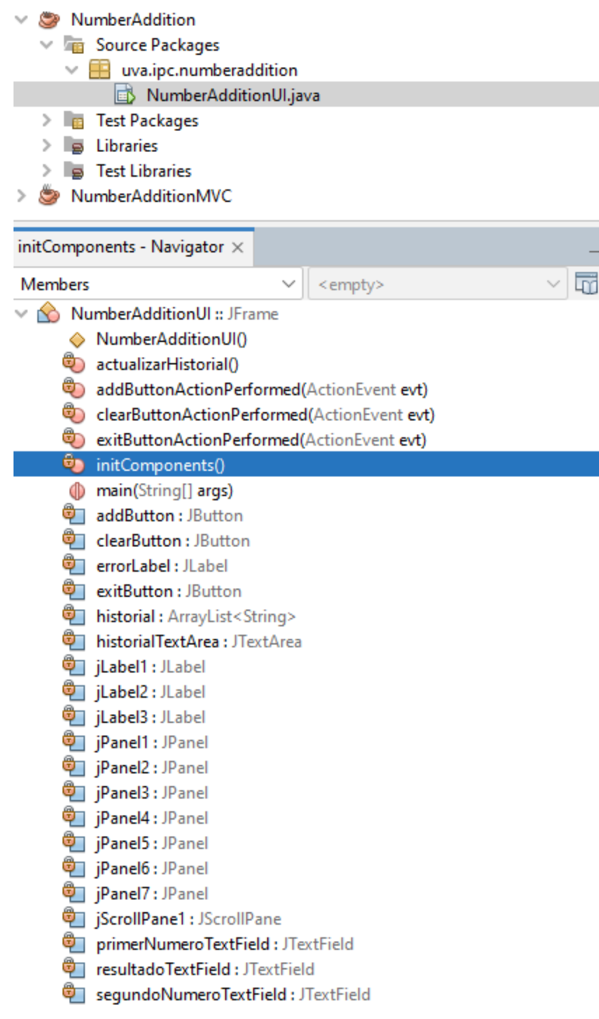


Figura 3.2: Código original

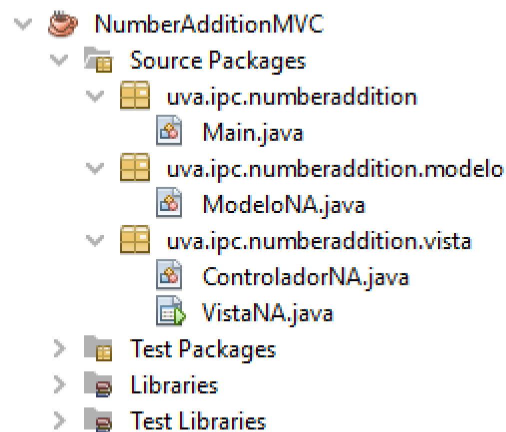


Figura 3.3: Paquetes y clases siguiendo el patrón MVC

```java
public class ModeloNA {

    private ArrayList<EntradaHistorial> historial;
    private double resultado;

    public double getResultado() {
        return resultado;
    }

    public void suma(double sumando1, double sumando2) {
        this.resultado = sumando1 + sumando2;
    }

    public ModeloNA() {
        historial = new ArrayList<>();
    }

    public ArrayList<EntradaHistorial> getHistorial() {
        return this.historial;
    }

    public void addEntradaHistorial(EntradaHistorial entrada) {
        this.historial.add(entrada);
    }

}
```

Listado 3.5: Código de la clase ModeloNA

```java
public class EntradaHistorial {

    private double primerSumando;
    private double segundoSumando;

    public EntradaHistorial(double primerSumando,
    double segundoSumando){
        this.primerSumando=primerSumando;
        this.segundoSumando=segundoSumando;
    }

    public double getPrimerSumando() {
        return primerSumando;
    }

    public void setPrimerSumando(double primerSumando) {
        this.primerSumando = primerSumando;
    }

    public double getSegundoSumando() {
        return segundoSumando;
    }

    public void setSegundoSumando(double segundoSumando) {
        this.segundoSumando = segundoSumando;
    }

}
```

Listado 3.6: Código de la clase EntradaHistorial

3.7.2 Paso 2: definir el Controlador

En el controlador añadimos dos atributos para almacenar la referencia a la vista y al modelo. En el constructor del controlador, asignamos la vista y creamos el modelo. Además, en el controlador hay que definir métodos para manejar los eventos que estaban ya definidos en la única clase existente (métodos lanzados por los manejadores definidos en Java). También copiamos el código existente en esos métodos a los nuevos métodos del controlador (listado 3.7).

```java
public class ControladorNA {

    private VistaNA miVista;
    private ModeloNA miModelo;

    public ControladorNA(VistaNA vista) {
        this.miVista = vista;
        this.miModelo = new ModeloNA();
    }

    public void procesaEventoSumar(){

    }

    public void procesaEventoLimpiar(){

    }

    public void procesaEventoSalir(){

    }
}
```

Listado 3.7: Código de la clase ControladorNA

El método que procesa el evento del botón de salir es el mismo que en la vista, ya que lo único que hace es cerrar la aplicación a través de la clase *System* de Java (listado 3.8).

```java
public void procesaEventoSalir() {
        System.exit(0);
}
```

Listado 3.8: Código del método procesaEventoSalir

Para el método limpiar los TextField, como no tenemos (ni debemos tener) acceso directo a los componentes gráficos de la vista, tenemos que crear en la vista un método para realizar esta tarea, y desde el controlador llamaremos a ese método. También llamamos a la vista para que limpie la etiqueta de los mensajes de error (listado 3.9).

```java
public void procesaEventoLimpiar() {
        this.miVista.limpiarCamposTexto();
        this.miVista.mostrarMensajeError("");
}
```

Listado 3.9: Código del método procesaEventoLimpiar

Por último, el método para realizar la suma, mostrar el resultado y actualizar el historial es un poco más complejo, ya que necesitamos hacer consultas a la vista así como realizar modificaciones en el modelo. Para acceder a los números introducidos por el usuario, utilizamos dos métodos de la vista que nos devuelven esa información. Aquí

empezamos a ver algunas ventajas del MVC, ya que el controlador no conoce los componentes específicos que se utilizan, sólo necesita la información. Por lo tanto, podríamos cambiar de componentes sin tener que modificar el controlador (por ejemplo, utilizando *JSpinner* en vez de *JTextField*). Después, procesamos esa información y, si no es correcta (no es un número), utilizamos otros métodos de la vista para que muestre los mensajes de error donde tenga definido (de nuevo, el controlador no sabe como se muestra la información). Después, el controlador llama al modelo para realizar la suma y utilizamos otro método de la vista para que lo muestre al usuario. Por último, creamos un objeto *EntradaHistorial* y lo añadimos al modelo, notificando después a la vista pasándole el modelo para que actualice la información. La información pasada a la vista pueden ser tipos básicos o clases del modelo, pero, como regla general, hay que pasar a la vista solo la información que necesite para mostrar la información, con el objetivo de reducir el acoplamiento entre la vista y el modelo (listado 3.10).

```
public void procesaEventoSumar() {
        double num1, num2;

        try {
            num1 = Double.parseDouble(this.miVista
            .getPrimerNumero());
        } catch (NumberFormatException e) {
            this.miVista.mostrarMensajeError("El primer numero
            debe ser un numero decimal");
            return;
        }
        try {
            num2 = Double.parseDouble(this.miVista
            .getSegundoNumero());
        } catch (NumberFormatException e) {
            this.miVista.mostrarMensajeError("El segundo numero
            debe ser un numero decimal");
            return;
        }
        this.miModelo.suma(num1, num2);
        this.miVista.mostrarResultado(this.miModelo
        .getResultado());
        EntradaHistorial entradaHistorial =
        new EntradaHistorial(num1, num2);
        this.miModelo.addEntradaHistorial(entradaHistorial);
        this.miVista.actualizarHistorial(this.miModelo
        .getHistorial());
        this.miVista.mostrarMensajeError("");
}
```

Listado 3.10: Código del método procesaEventoSumar

3.7.3 Paso 3: definir la Vista

En la vista solo nos falta hacer dos cosas: sustituir el código original de respuesta a los eventos por llamadas al controlador y añadir los nuevos métodos necesarios por el controlador. La primera parte es sencilla, ya que simplemente hay que llamar a los métodos del controlador que procesan los eventos (listado 3.11).

```
private void addButtonActionPerformed(java.awt.event
.ActionEvent evt) {
        miControlador.procesaEventoSumar();
    }

    private void exitButtonActionPerformed(java.awt.event
    .ActionEvent evt) {
        miControlador.procesaEventoSalir();
    }

    private void clearButtonActionPerformed(java.awt.event
    .ActionEvent evt) {
        miControlador.procesaEventoLimpiar();
    }
```

Listado 3.11: Código del método de respuesta al evento Sumar

En la segunda parte, los métodos *getPrimerNumero* y *getSegundoNumero* simplemente devuelven el contenido de los dos TextFields donde el usuario introduce la información. El resto de métodos actualizan la interfaz, bien a través de tipos básicos (*mostrarMensajeError* y *mostrarResultado*) o bien accediendo a los objetos del modelo (*actualizarHistorial*) (listado 3.12).

```
    public void limpiarCamposTexto() {
        primerNumeroTextField.setText("");
        segundoNumeroTextField.setText("");
        resultadoTextField.setText("");
    }

    public String getPrimerNumero() {
        return primerNumeroTextField.getText();
    }

    public String getSegundoNumero() {
        return segundoNumeroTextField.getText();
    }

    public void mostrarMensajeError(String mensaje) {
        errorLabel.setText(mensaje);
    }

    public void mostrarResultado(double resultado) {
        resultadoTextField.setText(String.valueOf(resultado));
    }
```

```java
    public void actualizarHistorial(
    ArrayList<EntradaHistorial> historial) {
        String historialText = "";
        for (EntradaHistorial elem : historial) {
            historialText += elem.getPrimerSumando() + "+"+
            elem.getSegundoSumando()+"\n";
        }
        historialTextArea.setText(historialText);
    }
```

Listado 3.12: Código de la Vista para obtener los valores de los componentes y actualizar el historial

3.8 Ejercicios propuestos

3.8.1 Ejercicio 1

- Crea una nueva vista de la aplicación *NumberAddition* para que los números a sumar se introduzcan a través de JSpinners en vez de TextFields e intercámbiala por la actual. ¿Qué cambios has tenido que hacer en la aplicación?
- Crea un nuevo modelo de la aplicación *NumberAddition* para que la información del modelo se lea/escriba en un fichero. ¿Qué cambios has tenido que hacer en la aplicación?

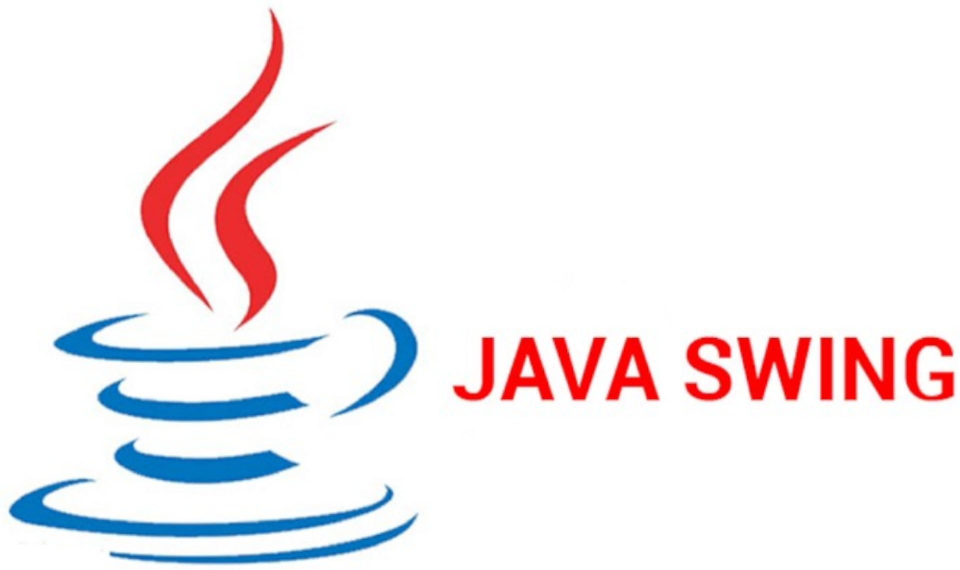
3.8.2 Ejercicio 2

Modifica el ejercicio propuesto (sección 2.10) del gestor de facturas para seguir el patrón arquitectónico MVC.

3.9 Enlaces a proyectos de ejemplo

- Código de NumberAdditionMVC: `https://github.com/IPC-UVa/NumberAdditionMVC`
- Código de GestorFacturasMVC: `https://github.com/IPC-UVa/FacturasMVC`

UVa



4. Gestión de múltiples vistas

4.1 Objetivos de aprendizaje

El objetivo general de este capítulo que el o la estudiante aprenda los conceptos básicos de la gestión de múltiples vistas utilizando JSwing.

- Objetivo específico 1: Entender qué es una máquina de estados y cómo se pueden aplicar para la gestión de múltiples vistas
- Objetivo específico 2: Crear una aplicación con varias ventanas utilizando los componentes de JSwing, utilizando una implementación de la máquina de estados para gestionar las diferentes vistas.
- Aplicar el patrón MVC a cada vista desarrollada.

Tiempo estimado: Dos horas

4.2 Requisitos previos

Para la realización de este tutorial es necesario tener conocimientos básicos de programación orientada a objetos en lenguaje Java. También es necesario tener instalado el IDE Netbeans.

4.3 Introducción

Aunque existen aplicaciones que se ejecutan en una única ventana, la mayoría reparten la funcionalidad entre múltiples ventanas. Esto puede deberse a que existe mucha funcionalidad para tenerla toda aglutinada en una misma vista, o esta división puede deberse a querer agrupar funcionalidad similar sin distraer al usuario con funciona-

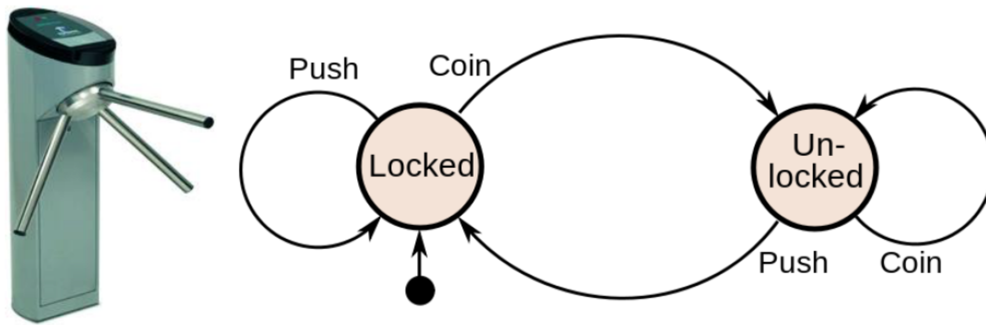


Figura 4.1: Ejemplo de aplicación de una máquina de estados a un torniquete de acceso. Extraída de `https://commons.wikimedia.org`

lidades que no necesita. Un ejemplo son las páginas webs, que, mediante hiperenlaces, permiten navegar entre múltiples interfaces. Actualmente, es muy común que la interfaz se muestre en un único frame, recargando solo las partes necesarias para evitar conexiones innecesarias y mejorar la experiencia de usuario. Independientemente de como se muestre la información, es necesario algún mecanismo que permita gestionar las relaciones entre las diferentes vistas o interfaces existentes, sobre todo cuando la aplicación crece y esta gestión se hace más complicada. En este punto es donde entran en juego las llamadas máquinas de estados.

4.4 Máquinas de estados

Aunque no es el objetivo de este tutorial entrar en profundidad en el concepto de máquina de estados y sus posibles usos (existe un patrón de diseño específico que se basa en este concepto), podemos definir como máquina de estado un modelo computacional que consta principalmente de los siguientes elementos:

- Estados: se definen como un conjunto de valores o comportamientos por lo que puede pasar un objeto. Siempre tenemos que definir un estado inicial que será el primero por el que pasa el objeto.
- Eventos: Son acciones, ya sean internas o externas que hacen que el objeto cambie de estado.
- Transiciones: Son los posibles caminos por los que el objeto cambia de un estado a otro.

Para poner un ejemplo de lo que es una máquina de estados en un objeto cotidiano, podemos ver la Figura 4.1, que representa un torniquete de acceso. El torniquete puede estar en dos estados diferentes, *Abierto* o *Cerrado*. Tenemos dos posibles eventos que disparan transiciones, empujar el torniquete o introducir una moneda. Por último, tenemos cuatro posibles transiciones. Si el torniquete está cerrado, al empujarlo no cambiará de estado, pero si se introduce una moneda, se produce la transición al estado abierto. Si el torniquete está abierto, no se produce ningún cambio de estado al introducir una moneda, pero sí al empujar el torniquete.

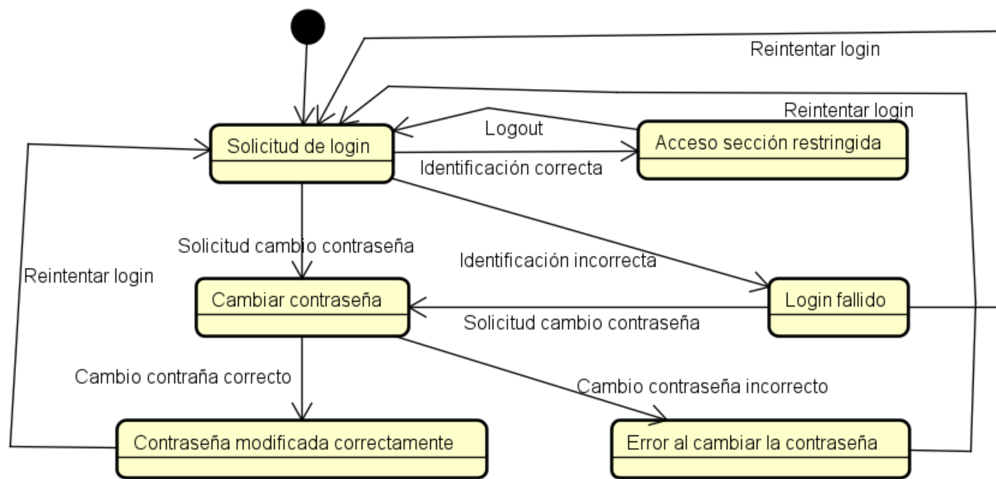


Figura 4.2: Diagrama de estados de un proceso de login

4.5 Máquinas de estado en el diseño de interfaces de usuario

Las máquinas de estados se pueden utilizar para modelar la lógica interna de una aplicación y definir los diferentes comportamientos según las entradas que se produzcan en el sistema. Por ejemplo, si queremos modelar el proceso de identificación de un usuario en un sistema, podemos crear un diagrama de estados como el que se puede ver en la Figura 4.2. Según este diagrama, primero el usuario tiene que identificarse en el sistema (*Solicitud de login*). Si la identificación es correcta, el sistema pasa a estar en un estado de *Acceso sección restringida*. Si no es correcta, se pasa al estado *Login fallido*, y en ese punto, podemos hacer una transición al estado de *Solicitud de login* para que el usuario vuelva a intentarlo o permitir que cambie la contraseña si la ha olvidado (*Cambiar contraseña*). Este estado también es accesible directamente desde *Solicitud de login*. Una vez en el estado *Cambiar contraseña*, se procede al cambio de la misma, y esta transición puede llevar a el estado de *Contraseña modificada correctamente* o, si se produce algún error, ir al estado *Error al cambiar la contraseña*. Desde estos dos estados se hace la transición a *Solicitud de login* para volver a intentar realizar el login. Si estamos en una aplicación con interfaz de usuario, algunos estados tendrán asociada una ventana para realizar las acciones del usuario y otros estados pueden mostrarse en la misma ventana. En el ejemplo que hemos visto en la Figura 4.2, *Solicitud de login*, *Cambiar contraseña* y *Acceso sección restringida* se pueden mostrar en tres ventanas diferentes, mientras que los demás estados se muestran mediante mensajes al usuario.

4.6 Máquinas de estados en JSwing

Para aplicar una máquina de estados en una aplicación que utilice la librería JSwing, vamos a seguir utilizando el patrón Modelo Vista Controlador (MVC), donde cada vista será un JFrame diferente con su propio controlador asociado. Sin embargo, las clases

del modelo serán las mismas para todas las vistas, ya que queremos que la aplicación conserve los cambios realizados en una vista cuando nos cambiemos a otra. La principal novedad es la introducción de una clase (o clases) que implementan la máquina de estados para gestionar las diferentes vistas de la aplicación. Para ello, vamos a utilizar una clase Java similar a la que se puede ver en el código del listado 4.1.

```
public class GestorVistas {

    private JFrame estadoActual;

    public GestorVistas(){
        mostrarVista1();
    }

    public void mostrarVista1(){
        if(estadoActual!=null){
            estadoActual.setVisible(false);
            estadoActual.dispose();
        }
        estadoActual=new Vista1();
        estadoActual.setVisible(true);
    }

    public void mostrarVista2(){
        if(estadoActual!=null){
            estadoActual.setVisible(false);
            estadoActual.dispose();
        }
        estadoActual=new Vista2();
        estadoActual.setVisible(true);
    }

}
```

Listado 4.1: Código de la clase de ejemplo de un gestor de vistas

Suponiendo que tenemos dos vistas (*Vista1* y *Vista2*), la clase *GestorVistas* tiene un atributo denominado *estadoActual* de tipo JFrame, que se encarga de almacenar la vista (estado) actual. Además, contiene un método para ocultar y destruir la vista actual (para liberar memoria) y para crear y mostrar la nueva vista. Estos dos métodos serían las transiciones de la máquina. Por otro lado, la clase *Main* (código del listado 4.2) ahora contendrá un atributo para almacenar la máquina de estados. El acceso al gestor de vistas (máquina de estados) se realiza a través de un método de clase estático, con el objetivo de que la máquina de estados se cree una única vez para poder ser utilizada desde los diferentes controladores de la aplicación. Para ello, definimos el atributo como privado y estático y creamos un método estático de clase para crear el gestor la primera vez que se le llame y devolver la instancia a quién la necesite. Existen técnicas para el acceso a los objetos del modelo, pero se salen del objetivo de este tutorial, por lo que seguiremos la misma estrategia que con el gestor de vistas y lo crearemos como un atributo privado y estático con su respectivo método de acceso. Si el número

de vistas es muy grande, es recomendable añadir más gestores específicos de vistas concretas y aplicar la misma estrategia de acceso que se ha detallado previamente.

```
public class Main {

    private static GestorVistas gestor;
    private static EjemploModelo modelo;

    public static GestorVistas getGestorVistas() {
        return gestor;
    }

    public static EjemploModelo getEjemploModelo() {
        return modelo;
    }

    public static void main(String args[]) {
        gestor = new GestorVistas();
        modelo = new EjemploModelo();
        gestor.mostrarVista1();
    }

}
```

Listado 4.2: Código de una clase Main genérica que tiene como parámetros el gestor de vistas y el modelo

4.7 Ejemplo

Para mostrar un ejemplo de la utilización de máquinas de estados en la gestión de múltiples vistas en JSwing, vamos a implementar una aplicación cuyo flujo de ejecución hemos visto en la Figura 4.2. En esta aplicación tendremos 3 ventanas, una para realizar el login, otra para cambiar la contraseña de un usuario y otra para ver la zona restringida del usuario una vez se ha logueado (ver Figuras 4.3, 4.4, 4.5). Siguiendo el patrón MVC, cada vista es una clase JFrame y tiene asociada un controlador propio. El modelo es único y compartido por todos los controladores de la aplicación.

Figura 4.3: Vista para realizar el login

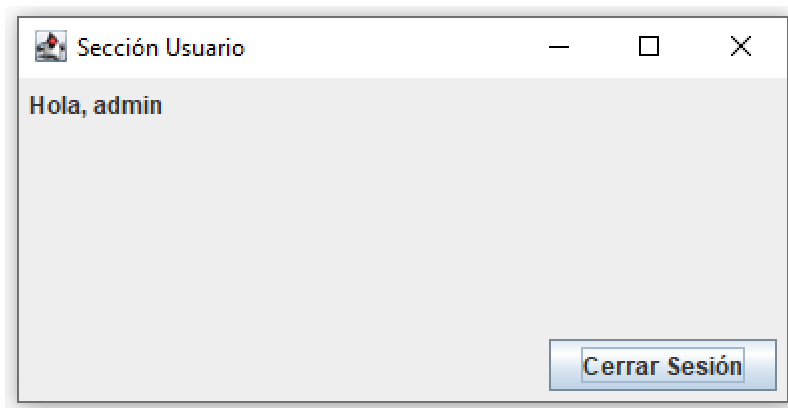


Figura 4.4: Vista de la zona restringida

Figura 4.5: Vista para cambiar la contraseña

```java
public class GestorVistas {

    private JFrame vistaActual;

    public void mostrarVistaLogin() {
        if (vistaActual != null) {
            vistaActual.setVisible(false);
            vistaActual.dispose();
        }
        vistaActual = new VistaLogin();
        vistaActual.setVisible(true);
    }

    public void mostrarVistaCambiarDatos() {
        if (vistaActual != null) {
            vistaActual.setVisible(false);
            vistaActual.dispose();
        }
        vistaActual = new VistaCambiarDatos();
        vistaActual.setVisible(true);
    }

    public void mostrarVistaSeccionRestringida() {
        if (vistaActual != null) {
            vistaActual.setVisible(false);
            vistaActual.dispose();
        }
        vistaActual = new VistaSeccionRestringida();
        vistaActual.setVisible(true);
    }

}
```

Listado 4.3: Código de la clase GestorVistas

4.7.1 Paso 1: creación del gestor de vistas

La clase *GestorVistas* (listado 4.3) implementa la máquina de estados para gestionar las tres vistas de la aplicación. La estructura de paquetes del proyecto puede verse en la Figura 4.6.

4.7.2 Paso 2: creación de las clases del modelo

Tenemos dos clases en el modelo, *RegistroUsuarios* (listado 4.5) y *Usuario* (listado 4.4). La clase *Usuario* almacena el usuario y contraseña y tiene getters y setters de esos atributos. La clase *RegistroUsuarios* tiene un atributo para almacenar los diferentes usuarios y otro para almacenar el usuario logueado. Además, tiene un método

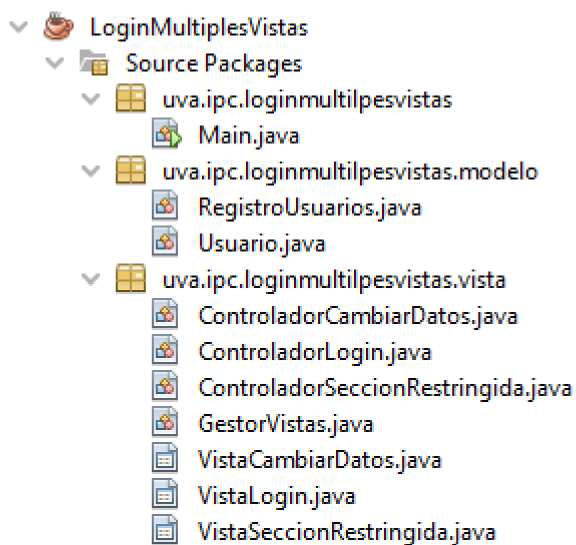


Figura 4.6: Estructura del proyecto de ejemplo

para comprobar las credenciales introducidas por el usuario en el login, un método para obtener un usuario por el nombre y dos métodos para devolver y asignar el usuario logueado. Por último, se ha añadido un método privado para poblar la lista de usuarios con dos usuarios de prueba para poder ver el funcionamiento de nuestra aplicación.

```
public class Usuario {

    private String nombre;
    private String password;

    public Usuario(String nombre, String password) {
        this.nombre = nombre;
        this.password = password;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
```

```
}
```

Listado 4.4: Código de la clase Usuario

```
public class RegistroUsuarios {

    private ArrayList<Usuario> listaUsuarios;
    private Usuario usuarioIdentificado;

    public RegistroUsuarios() {
        listaUsuarios = new ArrayList<>();
        rellenarUsuarios();
        usuarioIdentificado = null;
    }

    public boolean comprobarCredenciales(String usuario, String password) {
        for (Usuario u : listaUsuarios) {
            if (u.getNombre().equals(usuario)
            && u.getPassword().equals(password)) {
                return true;
            }
        }
        return false;
    }

    public Usuario getUsuarioByNombre(String nombreUsuario) {
        for (Usuario u : listaUsuarios) {
            if (u.getNombre().equals(nombreUsuario)) {
                return u;
            }
        }
        return null;
    }

    public Usuario getUsuarioIdentificado() {
        return usuarioIdentificado;
    }

    public void setUsuarioIdentificado(Usuario usuarioIdentificado) {
        this.usuarioIdentificado = usuarioIdentificado;
    }

    private void rellenarUsuarios() {
        Usuario usuario1 = new Usuario("admin", "admin");
        Usuario usuario2 = new Usuario("mario", "12345");
        listaUsuarios.add(usuario1);
        listaUsuarios.add(usuario2);
    }

}
```

Listado 4.5: Código de la clase RegistroUsuarios

4.7.3 Paso 3: creación de la clase Main

La clase *Main* (listado 4.6) es similar a la que ya hemos visto en otros ejemplos, añadiendo los atributos privados y estáticos para almacenar la referencia al gestor de vistas y al registro de usuarios y dos métodos privados y estáticos para obtener las instancias. La creación de los objetos y la llamada al método para mostrar la primera ventana (login) se realizá en el método main.

```
public class Main {

    private static GestorVistas gestor;
    private static RegistroUsuarios registro;

    public static void main(String[] args) {
        gestor = new GestorVistas();
        registro = new RegistroUsuarios();
        gestor.mostrarVistaLogin();
    }

    public static GestorVistas getGestorVistas() {
        return gestor;
    }

    public static RegistroUsuarios getRegistro() {
        return registro;
    }

}
```

Listado 4.6: Código de la clase Main

4.7.4 Paso 4: creación de las vistas y controladores

Las clases de las vistas no tienen nada especial más allá de notificar al controlador correspondiente los eventos que se producen y tener métodos para devolver los valores de los componentes de la interfaz. La clase *ControladorLogin* (listado 4.7) es el controlador de *VistaLogin*, y se encarga de comprobar si las credenciales introducidas por el usuario se corresponden con algún usuario del registro. Si es así, asigna el usuario como usuario logueado y abre la vista *VistaSeccionRestringida* utilizando el gestor de vistas al que accedemos a través del método estático de la clase *Main*. Si no existe el usuario, se muestra un mensaje de error. Además, tiene otro método para abrir la vista *VistaCambiarDatos*.

```
public class ControladorLogin {

    private VistaLogin vista;
    private RegistroUsuarios registro;

    public ControladorLogin(VistaLogin vista) {
        this.vista = vista;
        registro = Main.getRegistro();
    }

    public void procesaEventoAcceder() {
        if (registro.comprobarCredenciales(vista.getUsuario(),
        vista.getPassword())) {
            registro.setUsuarioIdentificado(registro
            .getUsuarioByNombre(vista.getUsuario()));
            Main.getGestorVistas().mostrarVistaSeccionRestringida();
            return;
        }
        this.vista.mostrarMensaje("Usuario o contrasena incorrectos");
    }

    public void procesaEventoCambiarDatos() {
        Main.getGestorVistas().mostrarVistaCambiarDatos();
    }

}
```

Listado 4.7: Código de la clase ControladorLogin

La clase *ControladorVistaCambiarDatos* (listado 4.8) comprueba si las contraseñas introducidas son iguales y si existe el usuario asociado al nombre introducido. Si es así, se muestra un mensaje de confirmación al usuario y se actualiza la contraseña. Si no, se muestra un mensaje de error al usuario. Además, tiene otro método para abrir la vista *VistaLogin*.

```
public class ControladorCambiarDatos {

    private VistaCambiarDatos vista;
    private RegistroUsuarios registro;

    public ControladorCambiarDatos(VistaCambiarDatos vista) {
        this.vista = vista;
        registro = Main.getRegistro();
    }

    public void procesaEventoGuardar() {
        String nombreUsuario = vista.getUsuario();
        String password = vista.getPassword();
        String passwordRepetida = vista.getPasswordRepetida();
        if (!password.equals(passwordRepetida)) {
            vista.mostrarMensaje("Las contrasenas no coinciden");
```

```
            return;
        }
        Usuario usuarioActual = registro
        .getUsuarioByNombre(nombreUsuario);
        if (usuarioActual == null) {
            vista.mostrarMensaje("El nombre de usuario
            no existe");
            return;
        }

        usuarioActual.setPassword(password);
        vista.mostrarMensaje("Contrasena actualizada con exito");
    }

    public void procesaEventoVolverALogin() {
        Main.getGestorVistas().mostrarVistaLogin();

    }

}
```

Listado 4.8: Código de la clase ControladorCambiarDatos

Por último, la clase *ControladorSeccionRestringida* (listado 4.9) muestra el nombre del usuario logueado (si existe) procesa el evento para cerrar la sesión, asignando null al usuario logueado en registro y mostrando la vista *VistaLogin*.

```
public class ControladorSeccionRestringida {

    private VistaSeccionRestringida vista;
    private RegistroUsuarios registro;

    public ControladorSeccionRestringida(
    VistaSeccionRestringida vista) {
        this.vista = vista;
        registro = Main.getRegistro();
        Usuario usuarioIdentificado = registro.getUsuarioIdentificado();
        if (usuarioIdentificado != null) {
            vista.mostrarNombreUsuario("Hola, " + usuarioIdentificado.getNombre());
        }
    }

    public void procesaEventoCerrarSesion() {
        registro.setUsuarioIdentificado(null);
        Main.getGestorVistas().mostrarVistaLogin();
    }

}
```

Listado 4.9: Código de la clase ControladorSeccionRestringida

4.8 Ejercicio propuesto

4.8.1 Ejercicio 1

Crea una nueva vista de la aplicación *LoginMultiplesVistas* para poder añadir usuarios nuevos al registro. Añade esta nueva ventana a la aplicación habilitando su acceso mediante la máquina de estados y modificando el modelo para que sea accesible a las otras vistas. También introduce elementos de navegación en las vistas para poder acceder a esta nueva ventana.

4.9 Enlaces a proyectos de ejemplo

- Código de LoginMultiplesVistas: `https://github.com/IPC-UVa/LoginMultiplesVistas`

UVa

Bibliografía

[Apa23] Apache. *Página principal de Apache Netbeans*. https://netbeans.apache.org [Accessed: 2023-06-01]. 2023 (véase página 14).

[Bra19] Paolo Brandi. *Architecture Patterns: Model-View-Controller*. https://medium.com/android-news/architecture-patterns-model-view-controller-de312417b4bd [Accessed: 2023-06-01]. 2019 (véase página 30).

[Fow06] Martin Fowler. *GUI Architectures*. https://martinfowler.com/eaaDev/uiArchs.html [Accessed: 2023-06-01]. 2006 (véase página 29).

[Gar23] Jairo García. *Aspectos generales de usabilidad y JSwing*. https://www.jairogarciarincon.com/clase/interfaces-de-usuario-con-java-swing/ui-ux-y-usabilidad [Accessed: 2023-06-01]. 2023 (véase página 10).

[Jav23] Javatpoint. *MVC Architecture in Java*. https://www.javatpoint.com/mvc-architecture-in-java [Accessed: 2023-06-01]. 2023 (véase página 29).

[Ora23a] Oracle. *Como usar los diferentes componentes JSwing*. https://docs.oracle.com/javase/tutorial/uiswing/components/index.html [Accessed: 2023-06-01]. 2023 (véase página 10).

[Ora23b] Oracle. *Convenciones de nombres para el lenguaje Java*. https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html [Accessed: 2023-06-01]. 2023 (véase página 21).

[Ora23c] Oracle. *Eventos que proporciona JSwing para cada componente*. https://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html [Accessed: 2023-06-01]. 2023 (véase página 15).