

UNIDAD DIDÁCTICA 3

CREACIÓN DE COMPONENTES VISUALES

MÓDULO PROFESIONAL:
DESARROLLO DE INTERFACES



CESUR
Tu Centro Oficial de FP

Índice

| | |
|---|----|
| RESUMEN INTRODUCTORIO..... | 2 |
| INTRODUCCIÓN | 2 |
| CASO INTRODUCTORIO | 2 |
| 1. HERRAMIENTAS PARA DESARROLLO DE COMPONENTES VISUALES. SCENE BUILDER 3 | |
| 1.1 Desarrollo de software basado en componentes..... | 5 |
| 1.2 Concepto de componente y sus características | 8 |
| 1.2.1 Propiedades y atributos | 10 |
| 1.2.2 Propiedades simples e indexadas | 13 |
| 1.2.3 Ámbito de las propiedades | 15 |
| 1.2.4 Atributos para los miembros de un componente o control | 16 |
| 1.2.5 Atributos que afectan en tiempo de diseño y en tiempo de ejecución | 17 |
| 1.2.6 Métodos para la creación y manipulación de componentes..... | 19 |
| 1.3 Eventos..... | 22 |
| 1.3.1 Asociación de acciones a eventos | 23 |
| 1.3.2 Generalizar el componente mediante la creación de eventos | 25 |
| 1.3.3 Comunicación del componente con la aplicación que lo usa, parámetros por valor y por referencia | 27 |
| 2. REUTILIZACIÓN DEL SOFTWARE | 31 |
| 2.1 Extender la apariencia y el comportamiento de los controles en modo de diseño | 32 |
| 2.2 Persistencia del componente..... | 33 |
| 2.3 Integrar controles existentes en nuestros componentes..... | 36 |
| 2.4 Prueba de los componentes | 37 |
| 2.5 Empaquetado de componentes..... | 42 |
| RESUMEN FINAL | 46 |

RESUMEN INTRODUCTORIO

En esta unidad, uniremos todos los conceptos sobre Eclipse, OpenJFX, FXML y Scene Builder para la creación de interfaces de usuario.

Avanzaremos sobre las propiedades de los componentes desarrollados con Scene Builder y OpenJFX. También introduciremos los conceptos base de eventos y manejadores, tan importantes cuando trabajamos con componentes e interfaces.

Por último, introduciremos los conceptos sobre reutilización de software y cómo la arquitectura MVC y la programación orientada a los módulos nos ayuda a crear aplicaciones más mantenibles y reutilizables.

INTRODUCCIÓN

Las herramientas para el desarrollo del software han evolucionado enormemente, proporcionando un entorno cómodo, adaptable, visual y de control de errores.

Esta evolución nos permite desarrollar interfaces visuales de una forma mucho más rápida, ya que previsualizamos el resultado de una forma directa, pero también de una forma más correcta, pues reducimos el número de errores en el desarrollo.

La herramienta perfecta, tanto si nuestro IDE es Eclipse como NetBeans, es Scene Builder, puesto que tiene todas las características para desarrollar sobre OpenJFX y, además, manejar ficheros FXML dentro de los estándares de OpenJFX.

CASO INTRODUCTORIO

Te contratan en una empresa de alquiler y venta de patinetes eléctricos como desarrollador informático. La empresa comienza a crecer debido al incremento en el uso de patinetes por parte de los usuarios. Por ese motivo, se plantea centralizar la información de alquiler de patinetes.

Para comenzar, se pretende realizar una pequeña interfaz que permita almacenar los datos básicos del alquiler de los patinetes eléctricos, y para ello, se plantea Java y OpenJFX como tecnología para dicho desarrollo.

Al final de esta unidad, tendrás los conocimientos para poder desarrollar aplicaciones completas basadas en estas tecnologías.

1. HERRAMIENTAS PARA DESARROLLO DE COMPONENTES VISUALES. SCENE BUILDER

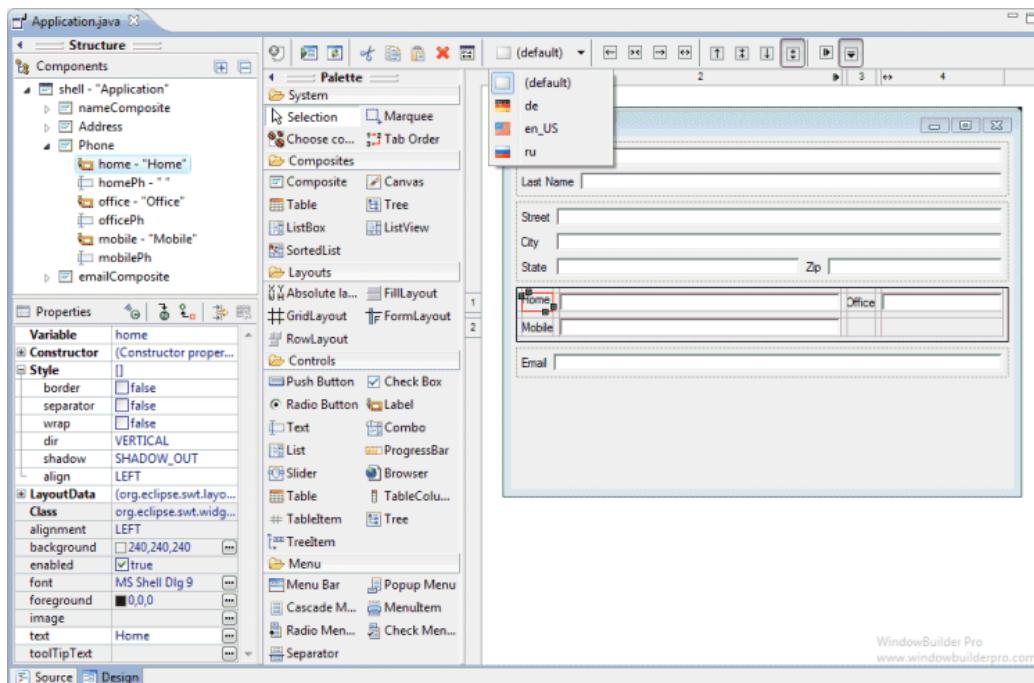
En el desarrollo de cualquier aplicación, es necesario transformar las especificaciones a interfaces de usuario.

Junto al gerente de la empresa, os planteáis que la pantalla para la captura de datos para el alquiler de patinetes debe ser lo más sencilla posible y, además, lo más rápida de cara al empleado de la tienda. Harás la interfaz con OpenJFX, poniendo los componentes adecuados (botones, etiquetas, campos de texto, entre otros). Tras esto, establecerás sus propiedades y métodos para hacer la interfaz lo más sencilla y eficaz posible.

El desarrollo de aplicaciones, y en concreto el desarrollo de componentes visuales o interfaces, ha evolucionado enormemente. Hemos introducido en unidades anteriores el proceso de desarrollo de aplicaciones basándonos en el paquete de librerías OpenJFX:

- Elección de un IDE de desarrollo.
- Selección e instalación de los plugins necesarios para las tecnologías de desarrollo.
- Incorporación de las librerías en nuestros proyectos.

Una herramienta gráfica será de vital importancia en el momento del desarrollo de interfaces de usuario. Dentro de Eclipse y el desarrollo de interfaces con Swing y AWT, históricamente hemos usado herramientas integradas como WindowBuilder.



Captura de WindowBuilder

Fuente: <https://www.eclipse.org/windowbuilder/>

En el caso de OpenJFX/JavaFX, la herramienta apropiada que se integra con el IDE, tanto Eclipse como NetBeans, se denomina Scene Builder, con la que ya hemos trabajado en unidades anteriores y donde vamos a hacer un uso intensivo en esta.

Una herramienta de desarrollo de componentes visuales como Scene Builder nos proporciona las siguientes ventajas en el trabajo de desarrollo:

- Trabajo de forma visual de toda la interfaz de usuario.
- Corrección del código de lenguaje de marcas, evitando errores de cierre de marcas de elementos y atributos.
- Tener diferentes vistas y paneles que nos permiten disponer de todos los contenedores, controles y sus propiedades.
- Una actualización con el estándar de definición FXML.

Por último, cabe destacar que es importante conocer el estándar sobre el cual se definen los interfaces de usuario mediante FXML, ya que, mediante el conocimiento de este estándar, somos capaces de retocar características que puede que las herramientas gráficas no sean capaces de manejar o bien podemos reordenar el código para que sea más legible.



ENLACE DE INTERÉS

No son muchos los artículos en castellano que encontramos sobre OpenJFX y JavaFX, pero los existentes nos ayudan a comprender las ventajas y la evolución de esta tecnología:



**PARA SABER MÁS**

Aquí tienes otra web que trata sobre JavaFX:



1.1 Desarrollo de software basado en componentes

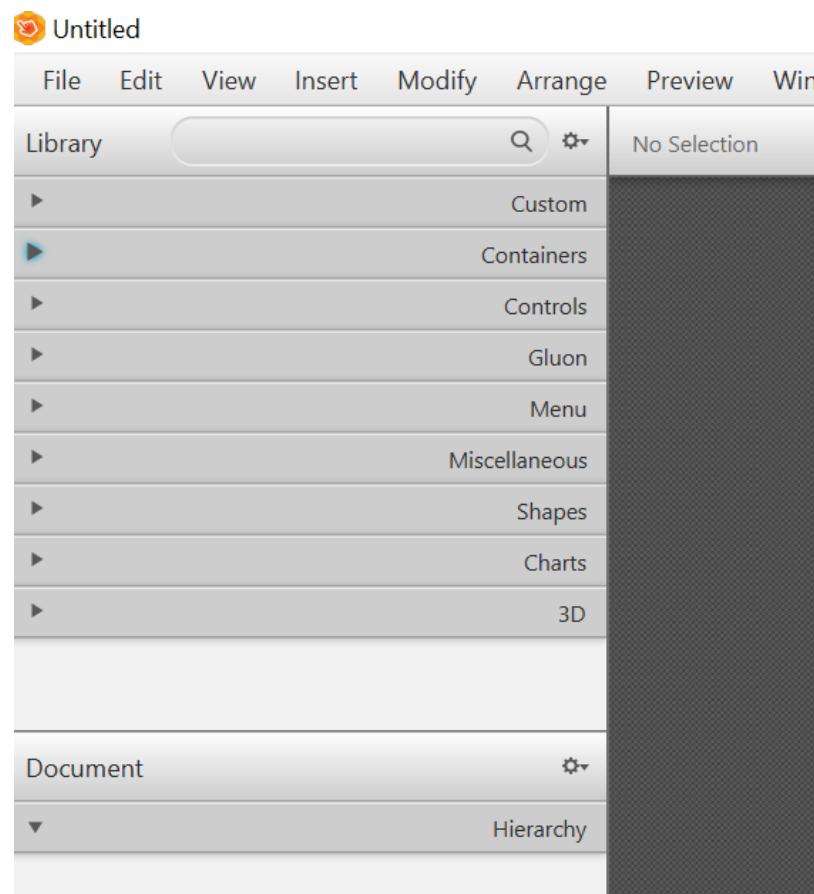
Basaremos nuestros ejemplos y actividades en aplicaciones de interacción con el usuario basadas en formularios. En este caso, nos encontramos con dos tipos de componentes principales para el desarrollo de aplicaciones que ya hemos visto en anteriores unidades:

- **Contenedores:** nos permiten almacenar y administrar los controles.
- **Controles:** son los componentes básicos que permiten mostrar y recibir información por parte del usuario.

**RECUERDA**

En anteriores unidades, definimos y mostramos las librerías de componentes que disponemos en OpenJFX, como los shapes, los controls y los containers.

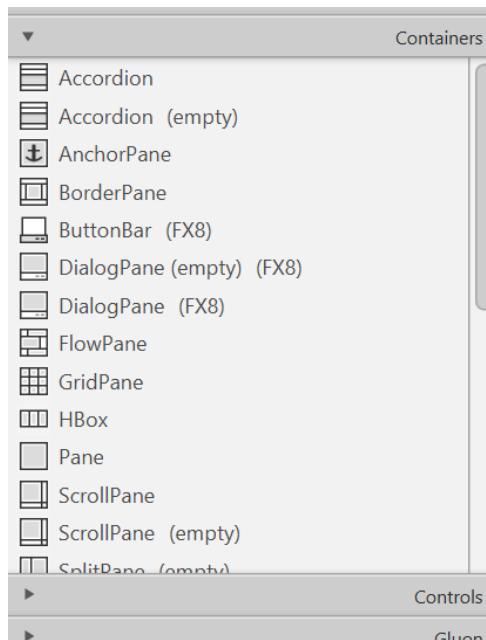
Dentro de Scene Builder, tenemos estos componentes definidos y preparados para usar dentro del apartado librerías, tal y como vemos en la imagen.



Librerías de componentes dentro de Scene Builder

Fuente: elaboración propia

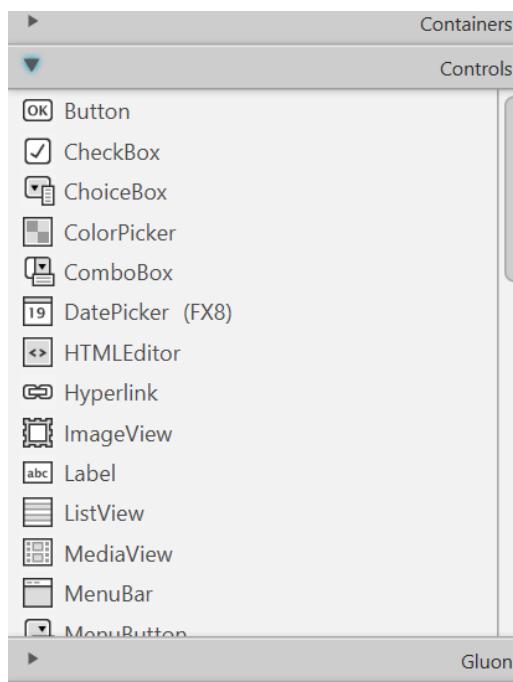
Si desplegamos la persiana de Containers, nos encontramos con el listado de containers usables.



Containers dentro de Scene Builder

Fuente: elaboración propia

De la misma forma, para poder usar cualquier control, deberemos desplegar la persiana inmediatamente debajo de Containers denominada Controls.

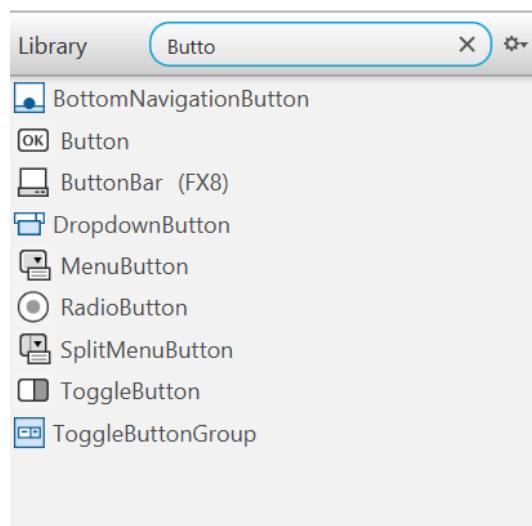


Controls dentro de Scene Builder

Fuente: elaboración propia

Tanto los containers como los controls coinciden con los paquetes de definición dentro de OpenJFX, siendo el paquete por defecto usado en su versión 21, ya que es la LTS marcada por el fabricante.

Por último, tenemos un buscador rápido que nos permite realizar la búsqueda de cualquier componente.



Búsqueda de los componentes

Fuente: elaboración propia

**ENLACE DE INTERÉS**

Dentro de la documentación oficial de Scene Builder, encontraremos más información y detalle sobre el software:



1.2 Concepto de componente y sus características

Podemos encontrar muchas definiciones de componentes, una de ellas puede ser que es una funcionalidad independiente del todo o del sistema. Un componente realiza funciones que requieren entradas y requieren salidas. Además, representa una o más tareas lógicas. Imaginemos el componente coche: se puede considerar que tiene diferentes entradas o funcionalidades, como la de transportar a gente, que tiene determinadas propiedades, como la velocidad máxima o el color, y que, además, puede estar formado por subcomponentes como el motor.

A nivel formal, un componente se define como una unidad modular y reutilizable que encapsula una funcionalidad específica dentro de una aplicación. Los componentes visuales representan elementos de la interfaz de usuario, como botones, etiquetas, campos de texto y contenedores, que son usados para construir interfaces.

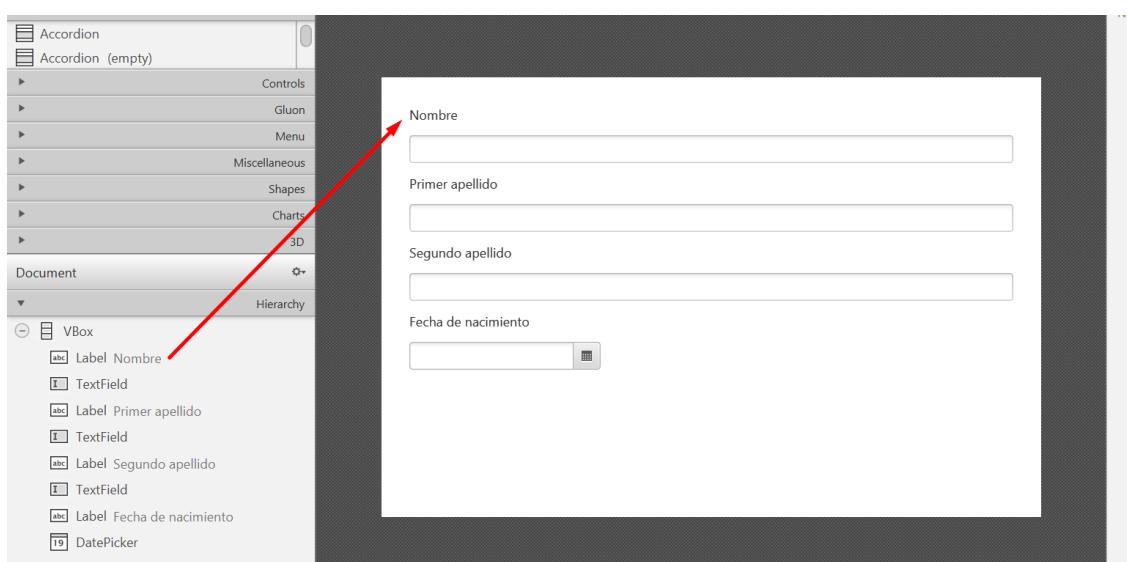
Las características principales de un componente son:

- **Encapsulación:** permite ocultar la implementación interna y exponer solo las interfaces necesarias.
- **Reusabilidad:** permite el uso del componente en diferentes partes de la aplicación o en proyectos futuros sin necesidad de reescribir el código.
- **Composición:** posibilita la creación de estructuras más elaboradas mediante la combinación de componentes más simples.
- **Propiedades y eventos:** permiten la personalización del componente y la interacción con el usuario.

El proceso para definir una interfaz con Scene Builder seguirá los pasos siguientes:

1. Creamos un fichero FXML dentro de nuestro proyecto con Eclipse.
2. Abriremos el fichero con Scene Builder.
3. Introduciremos el container principal. Cualquier interfaz de usuario (UI) necesita de un container raíz a partir del cual incluiremos el resto de controles o más containers.
4. Mediante Scene Builder, arrastraremos un nuevo componente y podremos ver sus características reflejadas en el fichero FXML.

Un ejemplo de este proceso lo vemos en la siguiente imagen donde, la inclusión de un nuevo control de tipo Label tiene su reflejo en el fichero FXML.



Control Label dentro de un container tipo VBox

Fuente: elaboración propia

```

<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity>
    <Label maxWidth="Infinity" text="Nombre" />
    <TextField />
    <Label maxWidth="Infinity" text="Primer apellido" />
    <TextField />

```

Código escrito en el fichero FXML

Fuente: elaboración propia

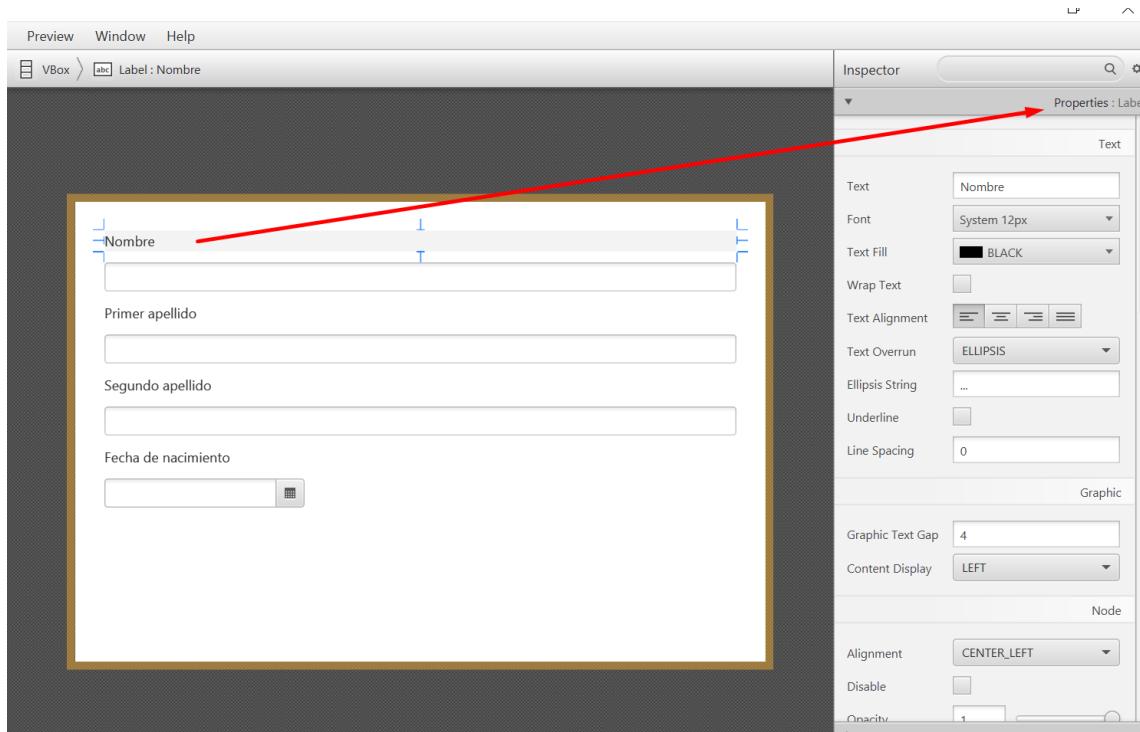
**ENLACE DE INTERÉS**

Aquí encontrarás un repaso a los componentes de JavaFX:



1.2.1 Propiedades y atributos

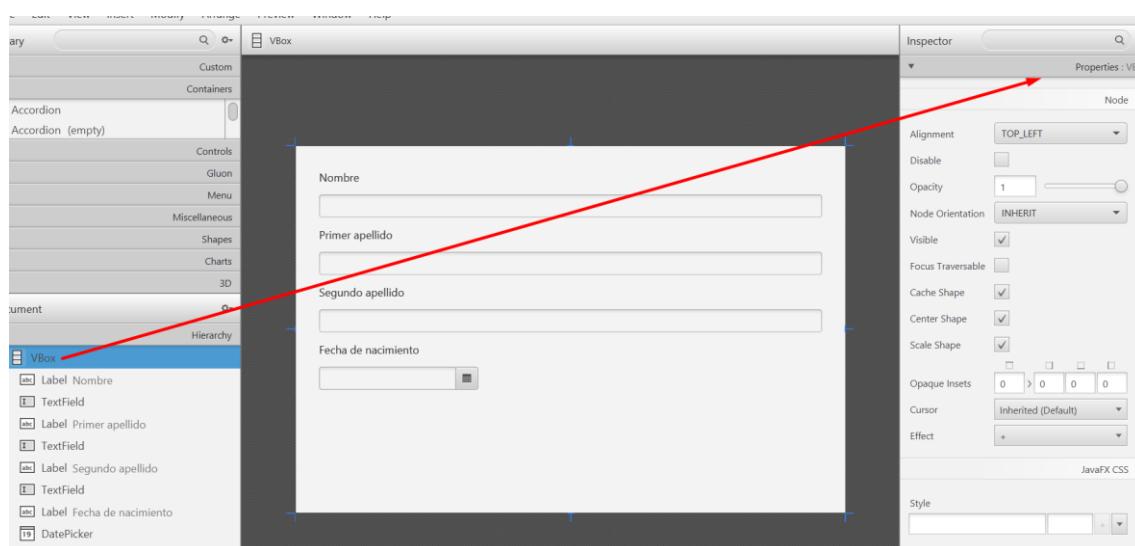
Una vez en nuestro diseño, seleccionando el componente, podremos visualizar todos los atributos y propiedades de dicho componente.



Selección y visualización de las propiedades

Fuente: elaboración propia

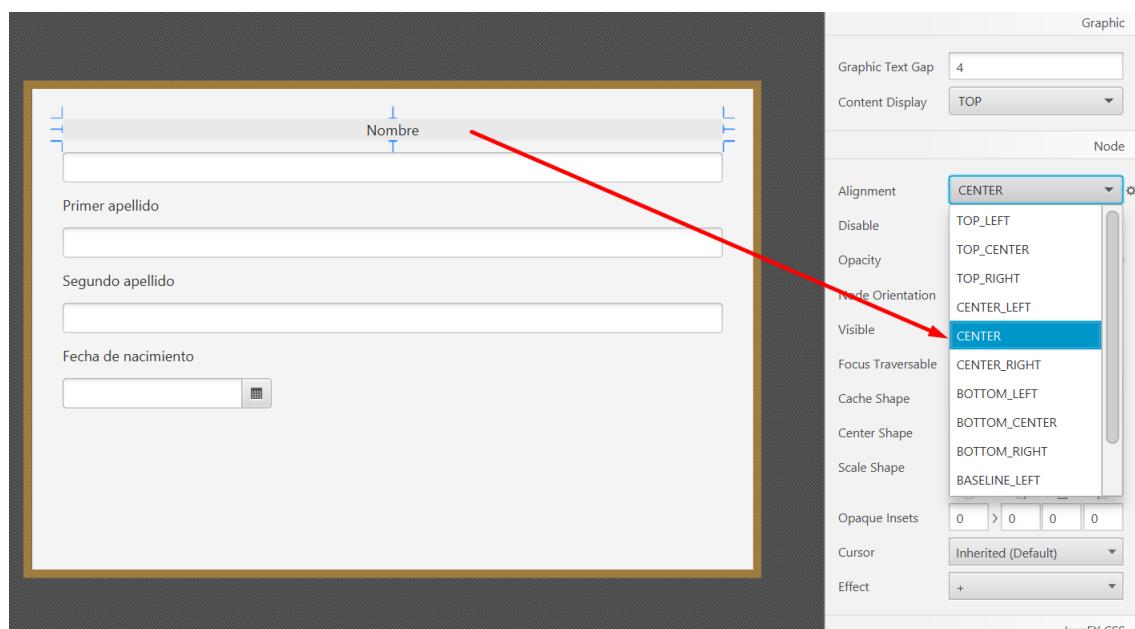
El panel de propiedades se adaptará al componente seleccionado y, aunque hay propiedades y atributos comunes, cada componente tendrá unas características diferentes.



Selección y visualización de las propiedades

Fuente: elaboración propia

Una vez que se selecciona un componente, podremos recorrer y buscar una propiedad y, por supuesto, modificar la propiedad o el atributo. Por ejemplo, una vez seleccionado el componente Label, podemos modificar cómo el texto es mostrado dentro del componente a través de la propiedad Alignment dentro del apartado Node.



Cambio de las propiedades

Fuente: elaboración propia

Una vez que cambiemos el diseño a través de Scene Builder, se reflejará dentro del documento FXML, que es realmente el fichero usado por la aplicación para manejar y cargar las interfaces.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.geometry.Insets?>
4 <?import javafx.scene.control.DatePicker?>
5 <?import javafx.scene.control.Label?>
6 <?import javafx.scene.control.TextField?>
7 <?import javafx.scene.layout.VBox?>
8
9 <VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" 
10 <children>
11     <Label alignment="CENTER" contentDisplay="TOP" maxWidth="Infinity" text="Nombre" />
12     <TextField />
13     <Label maxWidth="Infinity" text="Primer apellido" />
14     <TextField />
15     <Label maxWidth="Infinity" text="Segundo apellido" />
16     <TextField />
17     <Label maxWidth="Infinity" text="Fecha de nacimiento" />
18     <DatePicker />
19 </children>
20 <padding>
21     <Insets bottom="25.0" left="25.0" right="25.0" top="25.0" />

```

Cambio en el FXML
Fuente: elaboración propia



EJEMPLO PRÁCTICO

En la empresa donde trabajas, se pretende crear una interfaz sencilla para recoger los datos básicos y necesarios para el alquiler de patinetes. El primer paso es siempre crear un proyecto base con Eclipse, Scene Builder y su repositorio en GitHub. ¿Cuáles serían los pasos necesarios para tener esa plantilla?

El primer paso es crear un nuevo proyecto con Eclipse de tipo Java FX Project.

Generar 3 paquetes: Vista, Controlador y Modelo.

Generar la clase Main de entrada a la aplicación.

```

package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.fxml.FXMLLoader;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            BorderPane root =
(FXMLLoader)FXMLLoader.load(getClass().getResource("/Vista/alquiler.fxml"));
            Scene scene = new Scene(root,400,400);

            scene.getStylesheets().add(getClass().getResource("/Vista/application.css").toExternalForm());
            primaryStage.setScene(scene);
            primaryStage.setTitle("Alquiler de Patinetes");
            primaryStage.show();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

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

1. Crear una nueva vista denominada alquiler.fxml dentro del paquete Vista.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2. Crear un nuevo repositorio local y remoto.

```
>git init  
>git add -A  
>git commit -m "Nuevo proyecto Alquiler patinetes"  
>git remote add origin https://github.com/pacogomezarnal/AlquilerPatinetes  
>git push origin master
```



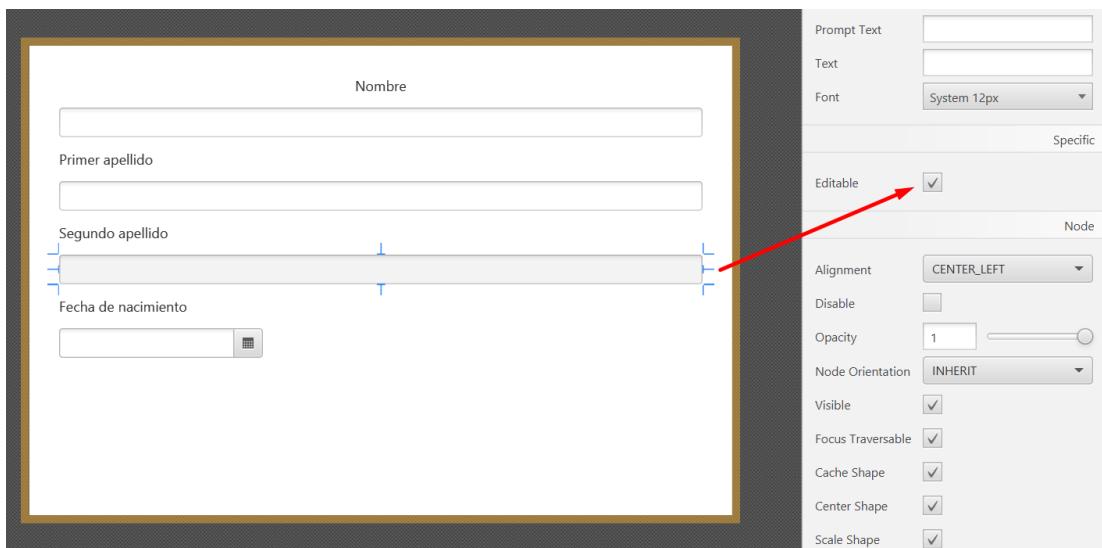
VÍDEO DE INTERÉS

Visualiza, paso a paso, la generación y diseño de un documento con Scene Builder:



1.2.2 Propiedades simples e indexadas

Dentro de un componente, nos encontramos diferentes tipos de propiedades con diferentes tipos de valores. Entre estas clasificaciones, nos encontramos con las propiedades simples y las indexadas. En el primer caso, la propiedad tiene asignado un valor único, numérico, textual, booleano o incluso complejo como otro objeto. Ya hemos visto ejemplos y, en la siguiente imagen, vemos cómo un control de tipo TextField puede ser editable o no mediante la propiedad simple Editable.



Propiedad simple de tipo booleano

Fuente: elaboración propia

Las propiedades indexadas requieren que el tipo que las defina permita una colección tipo lista, array, colección u otra dentro de Java que permita secuenciar de alguna manera los elementos que incluye dentro de la propiedad.

Es el caso del contenedor GridPane, donde los elementos se colocan de una forma ordenada en la posición indicada. Veamos un ejemplo.

```
GridPane gridpane = new GridPane();

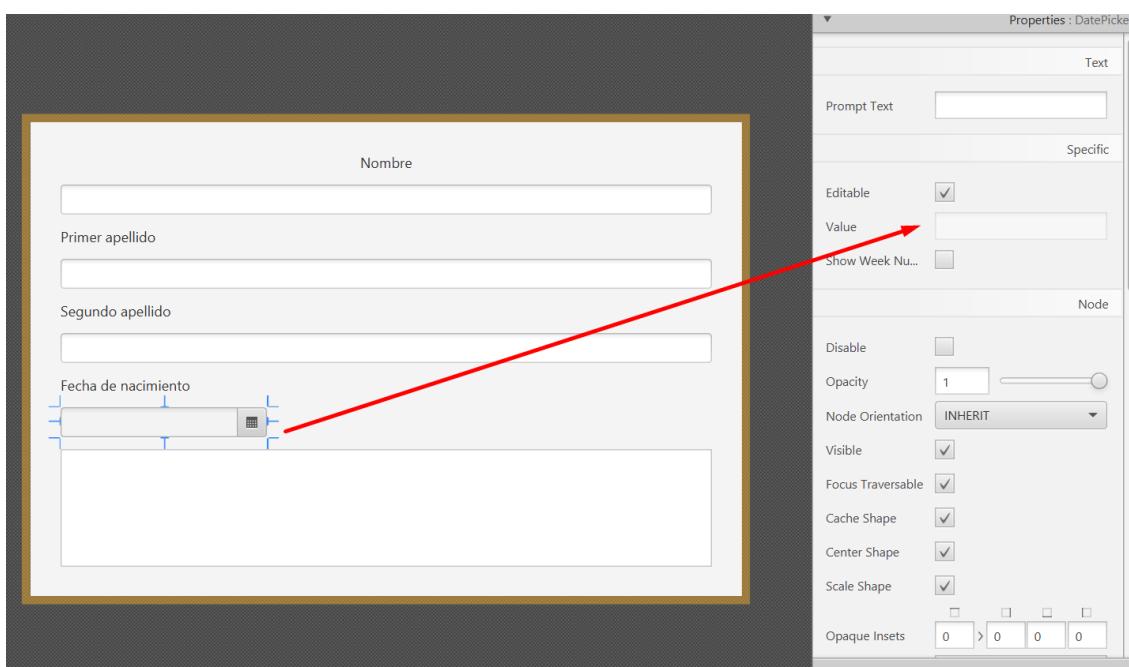
// Set one constraint at a time...
// Places the button at the first row and second column
Button button = new Button();
GridPane.setRowIndex(button, 0);
GridPane.setColumnIndex(button, 1);
```

En el anterior ejemplo se coloca el botón dentro de la posición 0,1.



1.2.3 Ámbito de las propiedades

El ámbito de las propiedades de los componentes se define como el espacio desde donde una aplicación puede acceder a estas propiedades. Por regla general, el ámbito de las propiedades, como cualquier otro objeto dentro de Java, va a seguir las mismas reglas de ámbito y, además, para poder acceder para modificar o leer las mismas, deberemos pasar por el componente.

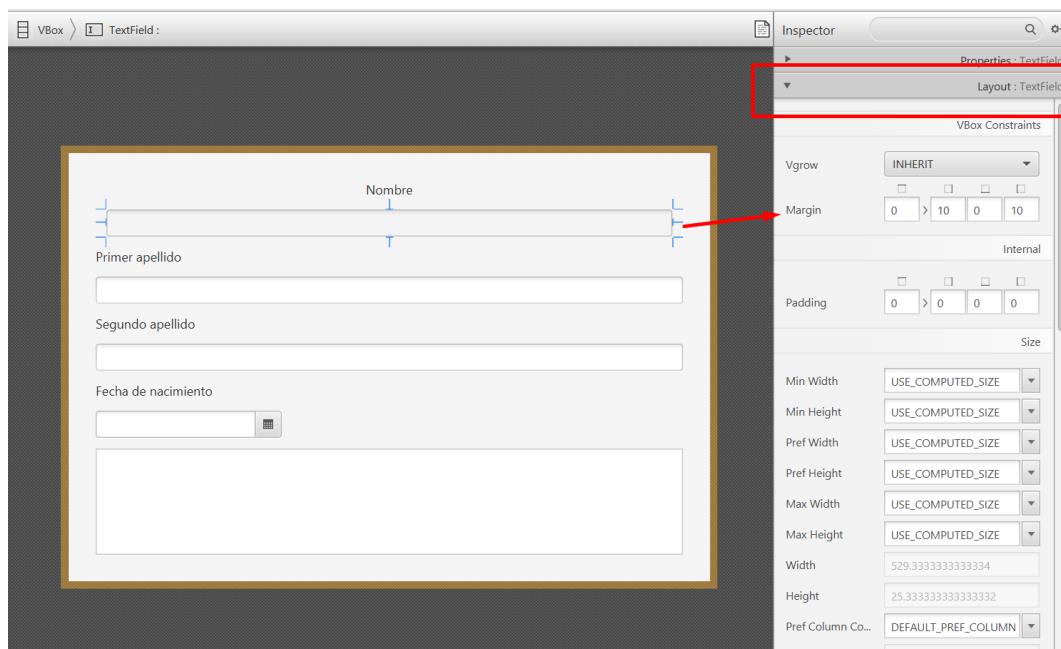


Propiedad Value de un DatePicker

Fuente: elaboración propia

Como vemos en la anterior imagen, la propiedad Value es accesible a través de la instancia del objeto.

En el caso de los contenedores, las propiedades que se refieren a posicionamiento, tamaño y alineación se encuentran delegadas a los controles que los contienen bajo las características del contenedor. En el ejemplo que estamos usando, el contenedor base es un VBox, que apila los elementos por filas en una única columna. Tenemos dentro de cada control un apartado layout que contiene propiedades referidas al control, pero con relación a su contenedor padre. En la imagen siguiente, vemos cómo podemos modificar el margen izquierdo y derecho de un control dentro de un contenedor Vbox.



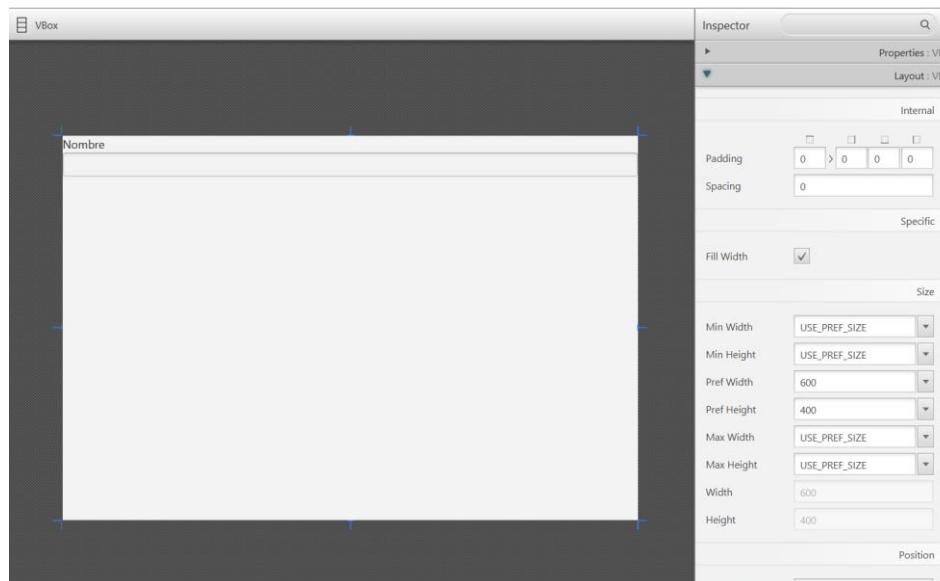
Propiedades de control referidas al layout

Fuente: elaboración propia

1.2.4 Atributos para los miembros de un componente o control

Cuando hablamos de miembros de un componente, estamos hablando, por regla general, de los nodos que contiene un contenedor. Tal y como vimos, un contenedor permite organizar en posición y en alineación los elementos que contiene.

Pongamos, por ejemplo, el container que ya hemos usado, VBox. En este container, al igual que en otros, existen propiedades y atributos que modificarán las características de los atributos miembros.

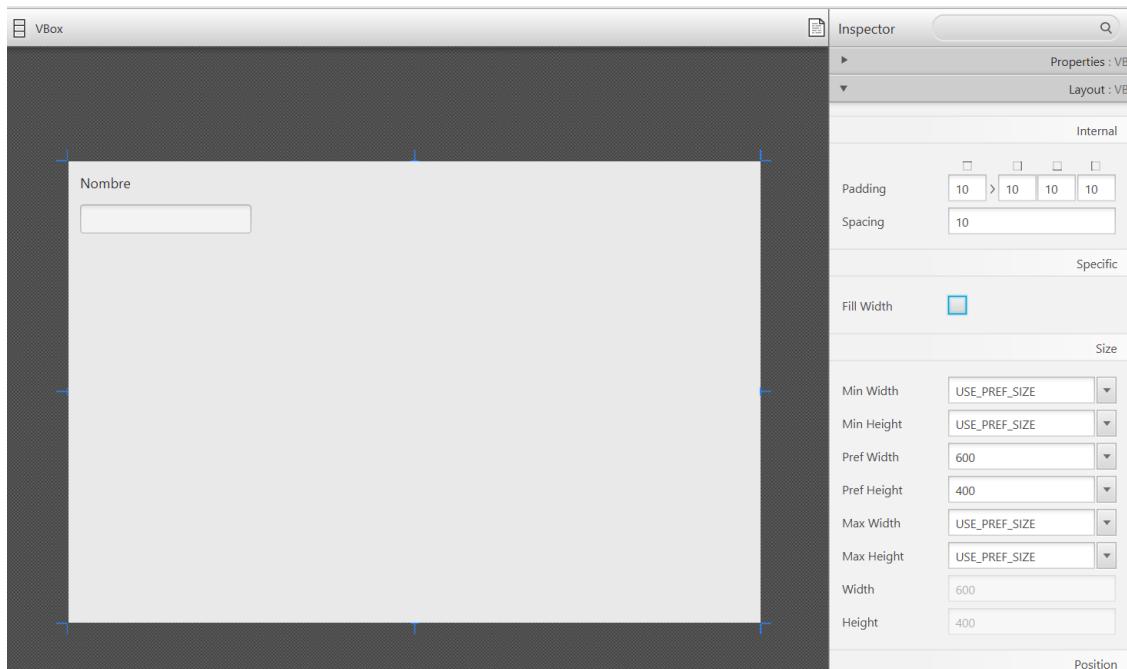


Propiedades de layout de VBox

Fuente: elaboración propia

En la anterior imagen vemos una serie de propiedades que, modificadas, aplicarán cambios a sus miembros, ya que modificarán sus tamaños o su posicionamiento:

- **Padding:** introduce un espaciado interno y, por lo tanto, el tamaño de los componentes internos se reduce en la cantidad introducida.
- **Spacing:** introduce un espacio entre componentes y, por ello, afecta al posicionamiento de los componentes internos.
- **Fill width:** afecta al comportamiento del tamaño en anchura de un miembro. Si los componentes internos no tienen un tamaño por defecto, los componentes internos ocuparán el tamaño completo.



Cambio de las propiedades de layout de VBox

Fuente: elaboración propia

Al realizar el cambio de las propiedades evaluadas, se producen cambios en las propiedades internas.

1.2.5 Atributos que afectan en tiempo de diseño y en tiempo de ejecución

Cuando diseñamos una aplicación que contiene interfaces de usuario, esperamos que, justamente, el usuario interactúe con esta. Veremos en el siguiente apartado los eventos como el recurso más importante para poder interaccionar con el usuario. En este proceso de diseño y uso de la aplicación, por lo tanto, tenemos dos escenarios como desarrolladores y diseñadores de una aplicación:

1. **Diseño de la aplicación mediante diferentes layouts y vistas a través de FXML y Scene Builder.** Es el escenario principal de diseño en el que trasladamos nuestro mockup a una visión de componentes, containers y controles.
2. **Ejecución de la aplicación, en la que podemos seguir programando el diseño de la aplicación, ya que podemos interactuar con la misma.** En este caso, cambia la forma en la que interactuamos con los diferentes componentes de la aplicación y deberemos usar el API que nos proporciona OpenJFX para poder seguir interactuando con los diferentes atributos de nuestros componentes.

El ejemplo lo podemos tener con una etiqueta Label introducida en tiempo de diseño a través del fichero FXML. En tiempo de diseño, definiríamos la propiedad del contenido de su texto mediante el atributo text, tal y como vemos en el ejemplo.

```
<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
xmlns="http://javafx.com/javafx/11.0.1"
xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <Label text="Nombre" />
        <TextField />
    </children>
</VBox>
```

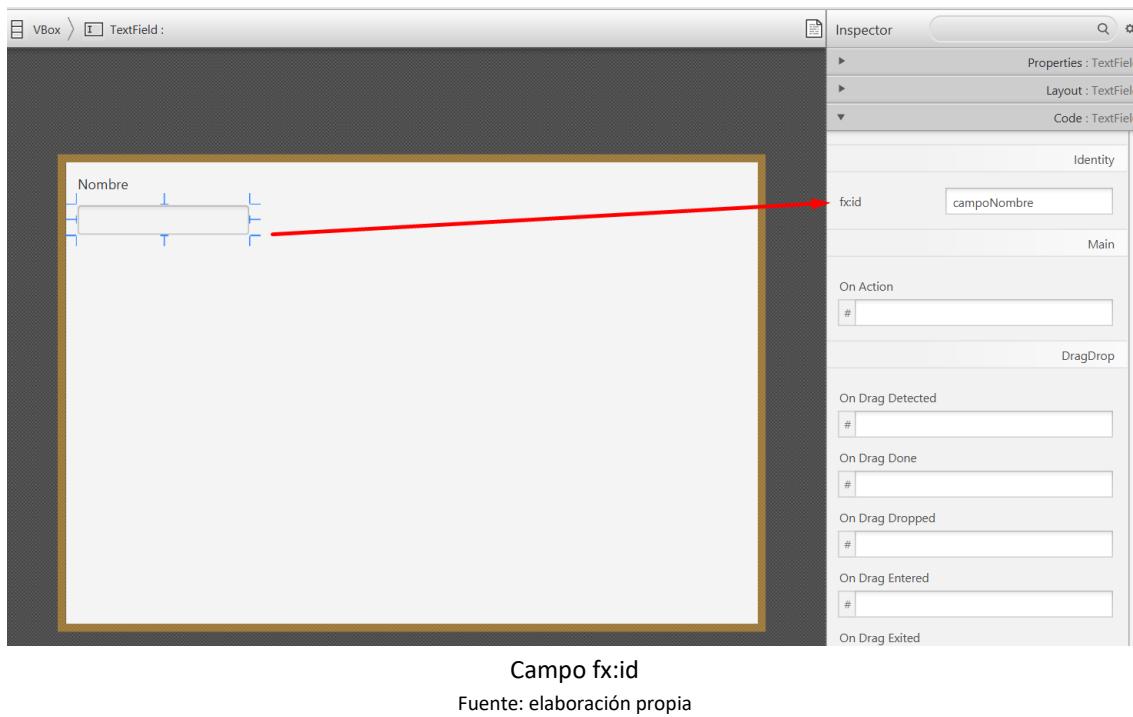
En tiempo de ejecución, podemos seguir modificando el componente, pero no podemos/debemos modificar el fichero FXML para realizar esos cambios. En este caso, utilizaremos los métodos definidos en el API:

```
public final void setText(String value)
Sets the value of the property text.
Property description:
The text to display in the label. The text may be null.
```

Una forma de referirse a una instancia en tiempo de ejecución es a través del nombre de variable de esa instancia. FXML nos permite asignar un id a una determinada instancia:

```
<fx:define>
    <ToggleGroup fx:id="myToggleGroup"/>
</fx:define>
...
<RadioButton text="A" toggleGroup="$myToggleGroup"/>
<RadioButton text="B" toggleGroup="$myToggleGroup"/>
<RadioButton text="C" toggleGroup="$myToggleGroup"/>
```

Como vemos en el anterior ejemplo, el id se puede definir y usar posteriormente. También lo podemos definir con Scene Builder a través de la ventana Code.



1.2.6 Métodos para la creación y manipulación de componentes

Los métodos son funciones o procedimientos que permiten interactuar con los componentes visuales en JavaFX en tiempo de ejecución. Dentro de estos métodos, tenemos métodos para la creación de componentes, modificación de propiedades, manejar eventos y para actualizar dinámicamente la interfaz.

Los métodos de creación de componentes nos permiten añadir componentes en tiempo de ejecución, como botones, etiquetas, campos de texto, etc. Cada componente tiene su clase y, por tanto, su método correspondiente. Por ejemplo:

```
Button button = new Button("Pulse aquí");
Label label = new Label("Etiqueta");
TextField textField = new TextField();
```

Una vez que se ha creado un componente, sus propiedades pueden modificarse mediante los siguientes métodos:

- **setText(String text)**: establece el texto que se mostrará en un componente, como un botón, etiqueta o campo de texto.
- **setStyle(String style)**: permite aplicar estilos CSS en línea a un componente.

- **setDisable(boolean value):** habilita o deshabilita un componente.
- **setVisible(boolean value):** controla la visibilidad de un componente.

Como vimos en la primera unidad, los componentes en JavaFX pueden interactuar con el usuario a través de eventos, como clics de ratón, entradas de teclado, etc. Los métodos se pueden utilizar para registrar manejadores de eventos:

- **setOnAction(EventHandler<ActionEvent> event):** asigna un manejador de eventos que se ejecuta cuando se produce una acción, como un clic en un botón.
- **setOnMouseEntered(EventHandler<MouseEvent> event):** se activa cuando el ratón entra en el área del componente.

Por último, durante la ejecución de la aplicación, existen métodos que nos permiten cambiar dinámicamente la interfaz de usuario. Por ejemplo, añadir o eliminar componentes en tiempo de ejecución:

```
VBox vbox = new VBox();
vbox.getChildren().add(button); // Añadir un botón al VBox
vbox.getChildren().remove(label); // Eliminar una etiqueta del VBox
```



EJEMPLO PRÁCTICO

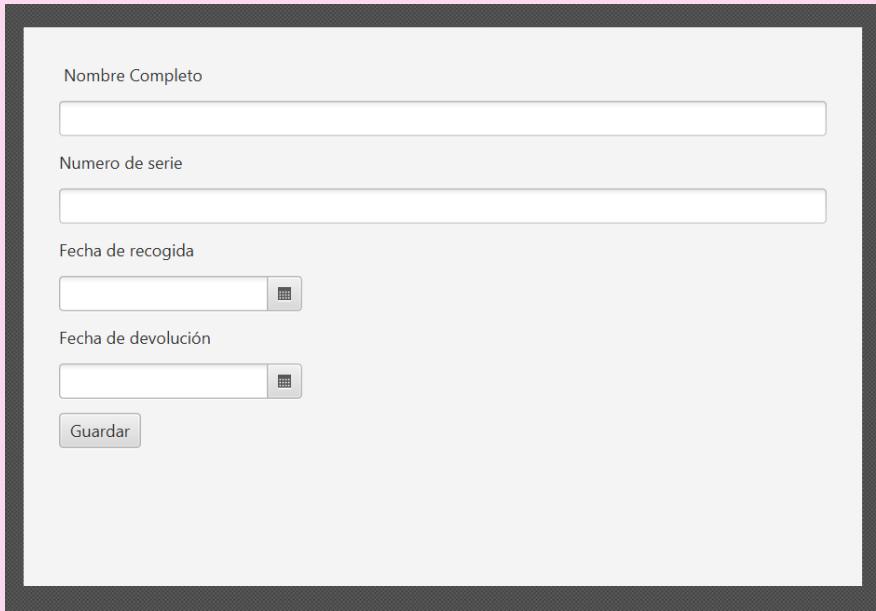
A partir de un proyecto base que tenemos como plantilla para el desarrollo sobre Eclipse, Java y OpenJFX en GitHub, queremos desarrollar nuestra primera interfaz (<https://github.com/pacogomezarnal/AlquilerPatinetes>). Queremos realizar una interfaz básica que cumpla con las especificaciones y datos que queremos recoger para el alquiler de patinetes dentro de nuestra empresa:

- Nombre completo del cliente.
- Número de serie del patinete.
- Fecha de recogida.
- Fecha de devolución.
- Observaciones.

¿Cuáles serían los pasos necesarios para tener esa interfaz?

1. El primer paso es clonar, si no lo tenemos ya, la plantilla base.
2. Abrimos el fichero alquiler.fxml mediante Scene Builder y añadimos los siguientes elementos:
 - a. VBox como container.
 - b. Añadimos un Label y un TextField para el nombre completo.

- c. Añadimos un Label y un TextField para el número de serie.
- d. Añadimos un Label y un DatePicker para la fecha de recogida y la de devolución.
- e. Añadimos el botón de guardado.



Formulario de alquiler de patinetes

Fuente: elaboración propia

3. El fichero FXML quedaría de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.DatePicker?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.VBox?>

<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" spacing="10.0"
xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="Controlador.AlquilerController">
    <children>
        <Label contentDisplay="TOP" maxWidth="Infinity" text=" Nombre
Completo" />
        <TextField />
        <Label maxWidth="Infinity" text="Número de serie" />
        <TextField />
        <Label maxWidth="Infinity" text="Fecha de recogida" />
        <DatePicker />
        <Label maxWidth="Infinity" text="Fecha de devolución" />
        <DatePicker />
        <Button mnemonicParsing="false" onAction="#handleButtonAction"
text="Guardar" />
    </children>
    <padding>
        <Insets bottom="25.0" left="25.0" right="25.0" top="25.0" />
    </padding>
</VBox>
```

4. El fichero Main se debe actualizar, ya que hemos cambiado el nodo raíz a un VBox:

```
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox; // Cambiado de BorderPane a VBox
import javafx.fxml.FXMLLoader;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            VBox root =
(VBox) FXMLLoader.load(getClass().getResource("/Vista/alquiler.fxml")); // Cambiado de BorderPane a VBox
            Scene scene = new Scene(root, 400, 400);

            scene.getStylesheets().add(getClass().getResource("/Vista/application.css").toExternalForm());
            primaryStage.setScene(scene);
            primaryStage.setTitle("Alquiler de Patinetes");
            primaryStage.show();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

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

5. Verificamos que el fichero module-info.java queda así para que no dé errores.

```
module AlquilerPatinetes {
    requires javafx.controls;
    requires javafx.fxml;

    opens Controlador to javafx.fxml;
    opens application to javafx.graphics, javafx.fxml;
}
```

6. Actualizamos el repositorio remoto.

```
>git add -A
>git commit -m "Modificado el interfaz de alquiler"
>git push origin master
```

1.3 Eventos

Como vimos en anteriores unidades, los eventos son acciones realizadas por el usuario, el sistema u otros componentes que permiten que una aplicación reaccione a estas interacciones. En JavaFX, los eventos son manejados a través de escuchadores, que son objetos encargados de 'escuchar' cuando ocurre un evento y ejecutar el código

correspondiente para manejarlo. Este sistema de manejo de eventos sirve para crear aplicaciones interactivas, ya que permite asociar acciones específicas a eventos definidos, como hacer clic en un botón o mover el ratón. La programación de eventos es una práctica muy habitual en el desarrollo de interfaces gráficas.

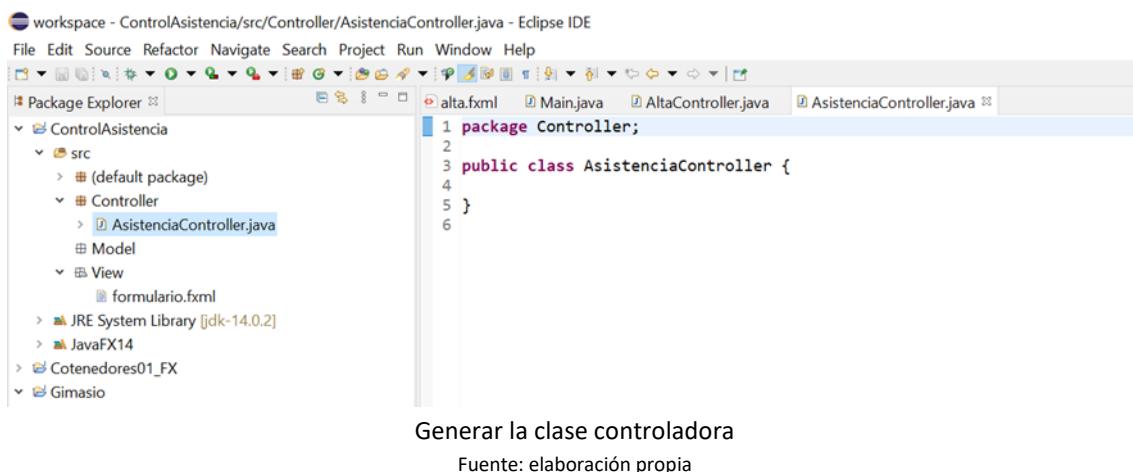
Como vimos, algunos de los eventos más comunes con el ratón son:

- **OnMouseClicked:** evento que se produce cuando se aprieta el botón.
- **OnMouseEntered:** evento que se produce cuando el ratón pasa por el botón.
- **OnMousePressed:** evento que se produce cuando el ratón aprieta, pero antes de soltar el botón.
- **OnMouseExited:** evento que se produce cuando el ratón sale de la zona de acción del botón.

1.3.1 Asociación de acciones a eventos

Con Scene Builder, la asociación de acciones a eventos es muy sencilla, ya que automáticamente nos permite elegir sobre los controles y métodos que tengamos asociados. El paso a paso lo podemos ver en el siguiente ejemplo:

1. Generamos el controlador dentro del paquete Controller:



2. Generamos un método para poder asociar a un evento:

```

package Controller;

import javafx.event.ActionEvent;

public class AsistenciaController {

    public void handleButtonAction(ActionEvent event) {

```

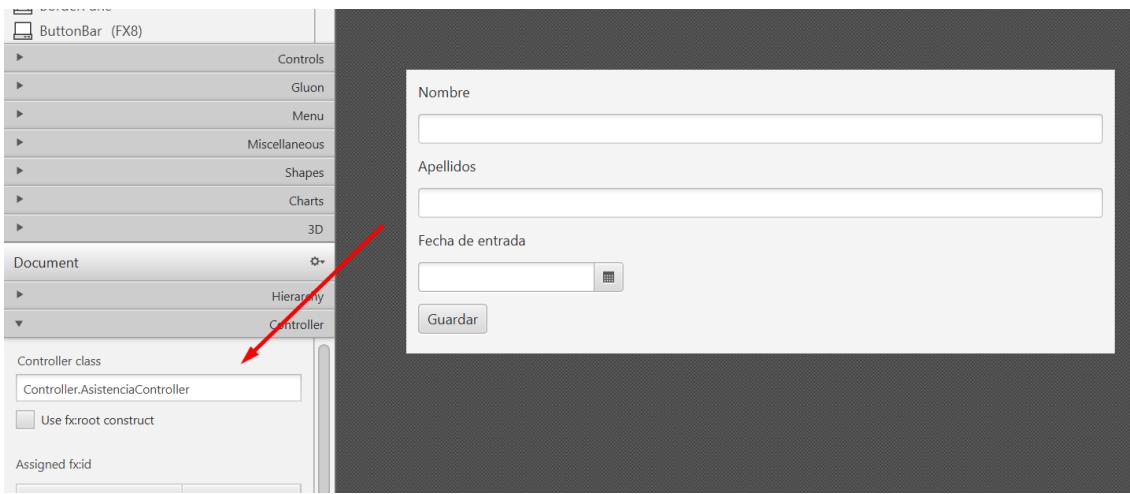
```

        System.out.println("Asistencia registrada");
    }

}

```

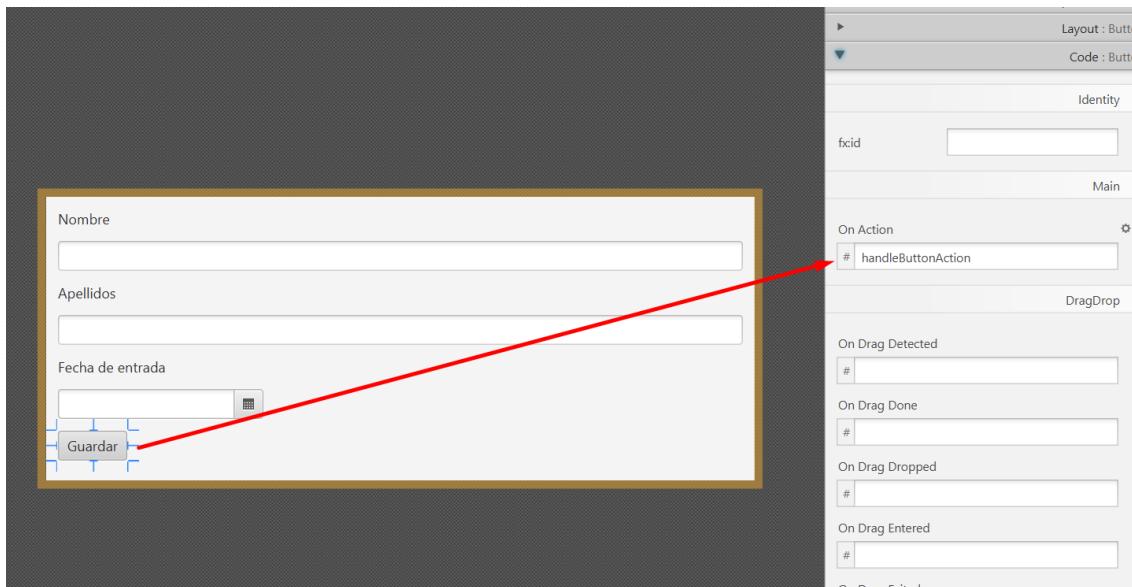
3. Con Scene Builder, asociamos el controlador al container:



Asociar el Controller al container

Fuente: elaboración propia

4. Por último, asociamos el manejador del evento al control o componente:



Asociar el manejador al componente

Fuente: elaboración propia

Al salvar la interfaz con Scene Builder, nuestro FXML quedará correctamente definido y asociado.

**VÍDEO DE INTERÉS**

Comprueba el paso a paso de la inclusión de un evento mediante Scene Builder:



1.3.2 Generalizar el componente mediante la creación de eventos

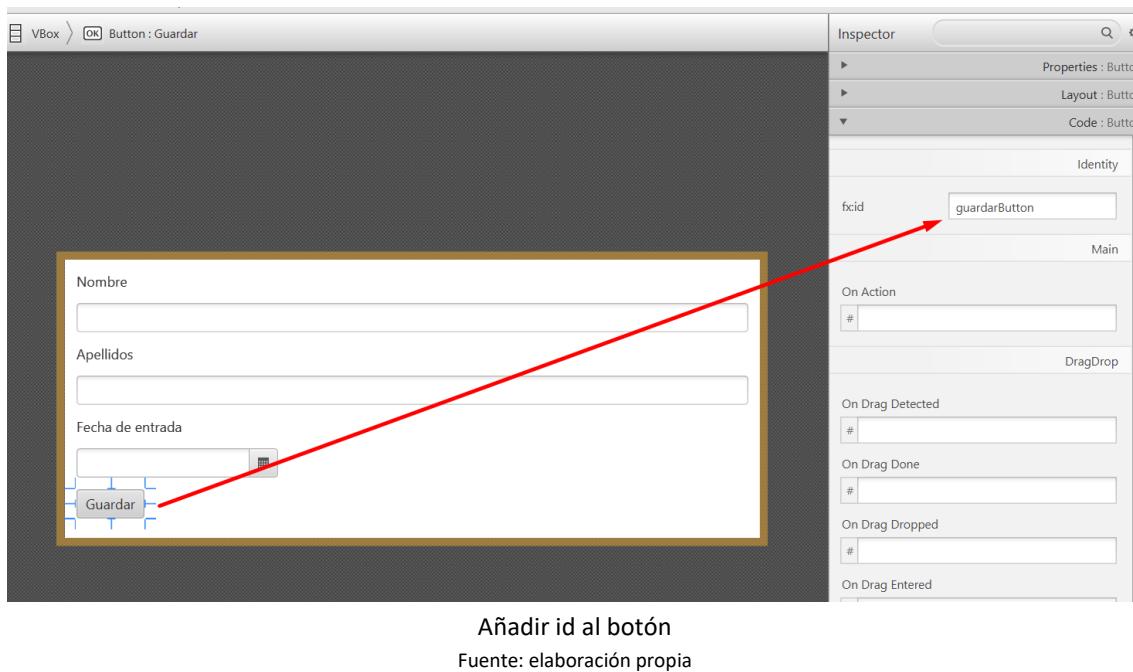
Acabamos de estudiar cómo podemos, de una forma muy visual, asociar un evento a un control. Sin embargo, no es la única forma de realizar esa asociación, como veremos a continuación.

Otra forma de inicializar un evento en un determinado componente es realizarlo mediante código en la iniciación del Controller. Para ello, seguiremos los siguientes pasos:

1. Creamos una clase de tipo Controller que asociaremos al container, vía Scene Builder o bien a través del fichero FXML.

```
<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="240.0" prefWidth="600.0"
spacing="10.0" xmlns="http://javafx.com/javafx/11.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="Controller.AsistenciaController">
    <children>
        <Label text="Nombre" />
        <TextField />
        <Label text="Apellidos" />
        <TextField />
        <Label text="Fecha de entrada" />
        <DatePicker />
        <Button fx:id="guardarButton" mnemonicParsing="false"
text="Guardar" />
    </children>
    <padding>
        <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
    </padding>
</VBox>
```

2. Damos un id único al botón dentro del diseño:



3. En el controlador, podemos asociar directamente el botón con una nueva propiedad:

```
@FXML
private Button guardarButton;
```

4. La forma más recomendada para asociar un evento al botón definido es mediante el método init, pero para ello debemos implementar Initializable en el Controller. El código final quedaría de la siguiente forma:

```
package Controller;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;

public class AsistenciaController implements Initializable {

    @FXML
    private Button guardarButton;

    public void handleButtonAction(ActionEvent event) {
        System.out.println("Asistencia registrada");
    }
}
```

```
@Override  
public void initialize(URL arg0, ResourceBundle arg1) {  
    System.out.println("Inicializo el controller");  
    guardarButton.setOnAction(this::handleButtonAction);  
}  
  
}
```



VÍDEO DE INTERÉS

Aquí visualizarás el paso a paso en la definición de eventos a través de la inicialización dentro de un Controller:



1.3.3 Comunicación del componente con la aplicación que lo usa, parámetros por valor y por referencia

Cuando se crea un método o una función, tenemos dos formas de pasar los parámetros:

- **Parámetros por valor:** se realiza una copia de la variable que se pasa al método y, por lo tanto, dentro de este método tendremos el valor, pero no la variable.
- **Parámetro por referencia:** donde se está pasando la variable original y, por lo tanto, tenemos su valor y la variable.

En otros lenguajes puede que tenga sentido hablar de paso por referencia o por valor, ya que existen esas opciones. Sin embargo, en Java, es la máquina virtual de Java la encargada de la gestión de la memoria, la creación y destrucción. Por lo tanto, no existe un paso por referencia como tal, como existe en otros lenguajes de programación:

- Si la variable es primitiva (un int, boolean, ...) se realiza una copia de la variable y se pasa al método.
- Si la variable no es primitiva, se crea un nuevo apuntador y se pasa al método.

Si probamos el siguiente código:

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

    int x = 1;
    int y = 2;
    System.out.print("Values of x & y before primitive
modification: ");
    System.out.println(" x = " + x + " ; y = " + y );
    modifyPrimitiveTypes(x,y);
    System.out.print("Values of x & y after primitive
modification: ");
    System.out.println(" x = " + x + " ; y = " + y );
}

private static void modifyPrimitiveTypes(int x, int y)
{
    x = 5;
    y = 10;
}
```

Vamos a obtener el siguiente resultado:

```
Values of x & y before primitive modification: x = 1 ; y = 2
Values of x & y after primitive modification: x = 1 ; y = 2
```

En el caso de los objetos, Java crea también una copia, pero en este caso se trata de una copia de la referencia, lo que significa que estará apuntando a la instancia del objeto. Si tenemos la siguiente clase y el siguiente método:

```
public class MiClase {
    public int valor;
}

public static void metodo_referencia(MiClase m) {
    m.valor = 3;
}
```

Cuando pasamos el objeto por el método, estamos modificando el parámetro valor:

```
MiClase m1 = new MiClase();
m1.valor = 2;
System.out.println(m1.valor); // Devuelve 2
metodo_referencia(m1);
System.out.println(m1.valor); // Devuelve 3
```

Como hemos visto, generar un evento para un determinado componente significa realizar dos pasos:

- Asignar el evento a un método o manejador del evento.
- Generar el código dentro del manejador.

El manejador puede ser creado a través del diseño o en la inicialización del controlador, y su forma es muy parecida en todos los casos. Retomemos el código anterior:

```

@Override
public void initialize(URL arg0, ResourceBundle arg1) {
    System.out.println("Inicializo el controller");
    guardarButton.setOnAction(this::handleButtonAction);
}

}

```



EJEMPLO PRÁCTICO

Tenemos desarrollada para nuestra empresa una interfaz básica que recoge los datos de un cliente que necesita alquilar un patinete. Necesitamos preparar esta interfaz introduciendo un evento a partir del botón y su comprobación de datos.

1. El primer paso es identificar los textfields y los datapickers con un identificador. El fichero FXML quedará de la siguiente forma:

```

<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.DatePicker?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.VBox?>

<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" spacing="10.0"
xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="Controlador.AlquilerController">
    <children>
        <Label contentDisplay="TOP" maxWidth="Infinity" text="Nombre
Completo" />
        <TextField fx:id="nombreCompleto" />
        <Label maxWidth="Infinity" text="Número de serie" />
        <TextField fx:id="numeroSerie" />
        <Label maxWidth="Infinity" text="Fecha de recogida" />
        <DatePicker fx:id="fechaRecogida" />
        <Label maxWidth="Infinity" text="Fecha de devolución" />
        <DatePicker fx:id="fechaDevolucion" />
        <Button fx:id="botonAlquiler" mnemonicParsing="false"
onAction="#alquilerAction" text="Guardar" />
    </children>
    <padding>

```

```

<Insets bottom="25.0" left="25.0" right="25.0" top="25.0" />
</padding>
</VBox>
```

2. Creamos un controlador que asignamos a nuestro container, además de implementar Initializable y asignárselo al botón.
3. En el evento del botón comprobamos que todos los campos del formulario no están vacíos. El controlador quedará, por lo tanto, de la siguiente forma:

```

package Controlador;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import javafx.scene.control.TextField;

public class AlquilerController implements Initializable {

    @FXML
    private Button botonAlquiler;
    @FXML
    private TextField nombreCompleto,numeroSerie;
    @FXML
    private DatePicker fechaRecogida,fechaDevolucion;

    public void alquilerAction(ActionEvent event) {
        System.out.println("Nuevo alquiler");
        //Comprobamos el campo nombreCompleto
        if(nombreCompleto.getText().isEmpty()) {
            nombreCompleto.setStyle("-fx-control-inner-background: #ef9a9a");
        } else if(numeroSerie.getText().isEmpty()) {
            numeroSerie.setStyle("-fx-control-inner-background: #ef9a9a");
        } else if(fechaRecogida.getValue() == null) {
            fechaRecogida.setStyle("-fx-control-inner-background: #ef9a9a");
        } else if(fechaDevolucion.getValue() == null) {
            fechaDevolucion.setStyle("-fx-control-inner-background: #ef9a9a");
        } else {
            System.out.println("Todos los campos tienen un valor");
        }
    }

    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
        System.out.println("Inicializo Eventos y otros elementos");
        botonAlquiler.setOnAction(this::alquilerAction);
    }
}
```

4. Actualizamos el repositorio remoto:

```

>git add -A
>git commit -m "Modificado el interfaz de alquiler"
>git push origin master
```

2. REUTILIZACIÓN DEL SOFTWARE

El desarrollo de cualquier aplicación pasa por diferentes estados, desde la escritura de las especificaciones del software hasta las pruebas y despliegue. También, en cualquier desarrollo, llega el momento de hacer persistente la información manejada.

Por lo tanto, te planteas cómo realizar el almacenamiento, cómo plantear la estructura de la aplicación y, por último, cómo desplegar la misma. Almacenas la aplicación en un gestor de bases de datos MariaDB, conectando la aplicación Java a través del conector JDBC. Haces pruebas unitarias y de integración sobre la interfaz y, por último, lo empaquetas en un JAR para su correcta distribución.

El software no es algo estático, paradigma que cada vez se hace más evidente en todos los ámbitos del desarrollo, tanto en la confección de los equipos de trabajo como en el desarrollo del software, el mantenimiento y la puesta en marcha. Una vez puesta en producción una determinada aplicación (despliegue y entrega al cliente), entramos en lo que es una de las fases más largas del software: el **mantenimiento**.

Para que ese mantenimiento se convierta en un proceso claro y sencillo, durante el proceso de desarrollo debemos tener en consideración el paradigma de la programación modular. Esta consiste en dividir un programa en módulos o subprogramas. Y, para ello, es necesario aplicarlo a diferentes niveles: a nivel de control, de clase, de paquetes y de aplicación.

Es importante, cuando se desarrolla un software, no pensar en los detalles específicos de la implementación, sino que el primer paso sea pensar a alto nivel y cómo descomponer el problema y, en última instancia, cómo será el desarrollo a nivel técnico.

Tener subdividido el problema en subproblemas y estas subpartes desarrolladas como módulos permite que esos módulos se puedan utilizar en otras situaciones. Nos permite, por lo tanto, la **reutilización de software**, implementado en módulos, para la resolución de problemas idénticos. Este paradigma ahorra tiempo de desarrollo y mejora el mantenimiento del software.

Los tres niveles de los que estamos hablando se pueden reflejar en las siguientes normas de buenas prácticas en el desarrollo para un buen planteamiento de la reutilización del software:

- **Programación orientada a objetos.** Java es una tecnología orientada principalmente a objetos, por lo que la utilización de esta ayuda mucho a tener esta metodología de programación. Debemos evitar el desarrollo excesivo en

una única clase, dividiendo el código, en primer lugar, en funciones o métodos y, en segundo lugar, en clases, apoyándonos de la tecnología Java para su realización.

- **Arquitectura MVC.** La separación de la lógica, la visualización y los datos permite tener, a su vez, dividido el desarrollo en diferentes clases que fácilmente se pueden extender y mantener en el futuro. Además, consiste en una metodología ampliamente estudiada y usada, por lo que es fácil encontrar paquetes y librerías que sigan esta arquitectura.
- **Documentación y pruebas.** Permiten que lo que en un principio parece una organización más compleja sea una organización perfecta para la organización en tareas e hitos dentro de un equipo de trabajo.
- **Implantación de herramientas de control de versiones y despliegue de aplicaciones.** Estas nos permiten tener el control del desarrollo y de su historia y sus partes.

2.1 Extender la apariencia y el comportamiento de los controles en modo de diseño

Hemos explicado la importancia de separar nuestro código en diferentes módulos, en concreto, en paquetes. Un paquete es una colección de clases con la posibilidad de crear nuevos subpaquetes. Esta agrupación nos permite definir, a su vez, lo que se denomina namespace (“espacio de nombres” sería la traducción directa, pero la traducción literal no define tan correctamente como el término en inglés, namespace).

Para especificar un paquete, usamos en Java la directiva package:

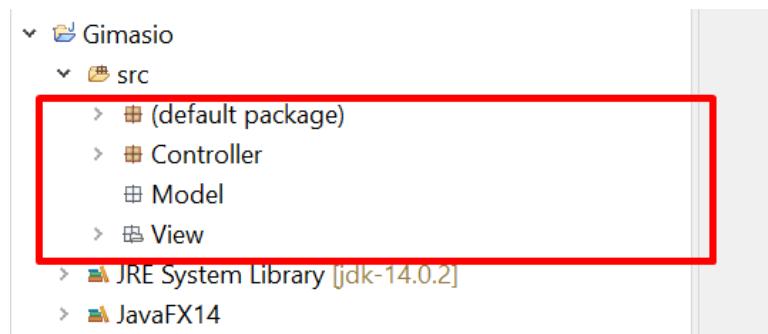
```
package com.dam.di;
```

En el anterior ejemplo definiríamos el paquete con el namespace com.dam.di, como ejemplo del conjunto de clases que pertenecerían a esa aplicación. En nuestros ejemplos y código ya han aparecido estas nomenclaturas:

- **default package:** cuando comenzamos un nuevo proyecto con Eclipse, se define un paquete por defecto que no tiene nombre y sobre el cual nuestras clases y, sobre todo, la clase de inicio de la aplicación, están definidas.
- **javafx.*:** librerías y clases importadas, una vez incorporadas a nuestra aplicación, mediante la directiva import, son utilizadas por nuestro código con el nombre

abreviado Button, por ejemplo, o a través de su namespace completo, javafx.scene.control.Button.

A partir de aquí, para extender y tener un código mejor ordenado y estructurado, introducimos el concepto de arquitectura MVC, sobre la cual podemos definir paquetes que permitirán ordenar nuestro código y hacerlo más reutilizable.



Paquetes arquitectura MVC

Fuente: elaboración propia

Como vemos en la imagen, además del paquete por defecto referido al classpath de la aplicación, aparecen nuevos paquetes que permitirán incluir código para:

- La lógica del programa a través del paquete Controller.
- La interacción con las BBDD a través del paquete Model.
- Las vistas e interfaces a través del paquete View.



ENLACE DE INTERÉS

Aquí encontrarás una introducción muy visual sobre la arquitectura MVC:



2.2 Persistencia del componente

Las ventanas de propiedades de las herramientas de diseño visual se usan para modificar las propiedades de los componentes a medida de las aplicaciones específicas. Las propiedades modificadas quedan almacenadas de tal forma que permanecen junto al

componente desde el diseño hasta la ejecución. La capacidad de almacenar los cambios realizados en las propiedades de un componente se conoce como persistencia. La persistencia permite personalizar los componentes para usarlos posteriormente.

Java Beans permite la persistencia a través de la serialización de objetos (capacidad de escribir un objeto Java en un flujo, de tal manera que se conservan la definición y el estado actual del objeto). Cuando se lee un objeto serializado en un flujo, el objeto se inicializa exactamente en el mismo estado en el que estaba cuando fue escrito en el flujo.

Poco a poco, estos sistemas de comunicación servidor-cliente mediante SOAP se han ido sustituyendo por arquitecturas más estándares como, por ejemplo, el de la comunicación mediante HTTP y JSON, sistemas ApiRestFull, en vez de sistemas SOAP.

Por otro lado, tenemos la persistencia de los componentes respecto a la información y datos tratados. No vamos a entrar en profundidad en la persistencia de los datos a través de los componentes, ya que otros módulos se encargan de esta formación. El diseño de interfaces tiene como fin máximo interactuar con el usuario, recogiendo y mostrando información para el correcto funcionamiento de la aplicación.

Dentro de las metodologías que podemos usar para la interacción con información y datos, tenemos:

- **Datos en memoria:** es rápido de usar e implementar. Sin embargo, no se produce persistencia alguna de información, ya que, una vez finalizada la aplicación, los datos se pierden.
- **Datos en ficheros:** en este caso, es lento su uso y complicada su implementación.
- **Datos en bases de datos:** es el ideal de uso de una aplicación para producir una persistencia y, además, separar las funciones dentro de la arquitectura MVC.

El ideal es utilizar la última propuesta, pues:

- Permite separar funciones.
- Permite una reutilización del código en diferentes partes de la aplicación.
- Permite un mejor mantenimiento del software.

Sin llegar a profundizar en las instalaciones de los diferentes softwares que necesitaríamos, para implementar persistencia de datos a través de los componentes en nuestra aplicación, deberíamos:

- Crear una base de datos, por ejemplo, dentro del gestor y servidor MariaDB.
- Asociar las librerías JDBC para la conexión contra la base de datos.
- Crear una clase DB.java para la gestión de las conexiones.
- Crear un modelo que se relacionase con una de las tablas de la base de datos.
- Importar el modelo dentro del controlador correspondiente.
- Usar el modelo dentro del controlador y las diferentes acciones.



ENLACE DE INTERÉS

Accediendo a este enlace, encontrarás una de las bases de datos relacionales open source más usadas para el desarrollo individual y profesional de proyectos, MariaDB:



EJEMPLO PRÁCTICO

En la empresa que trabajamos, tenemos instalada MariaDB en el servidor. Pretendemos usar esta base de datos centralizada para almacenar los diferentes pedidos de alquiler de patinetes a partir de la interfaz de usuario que hemos desarrollado y que tenemos almacenada en un repositorio remoto.

¿Cómo distribuimos las clases? ¿Cómo hacemos persistentes los datos?

1. El primer paso será crear una base de datos y una tabla acordes con la información que vamos a almacenar:
 - a. Creamos la base de datos Empresa.
 - b. Creamos la tabla alquiler con la siguiente estructura:

```
CREATE TABLE `empresa`.`alquiler` ( `id` INT NOT NULL AUTO_INCREMENT ,  
 `nombreCompleto` VARCHAR(256) NOT NULL , `numeroSerie` VARCHAR(128) NOT NULL  
 , `fechaRecogida` DATE NOT NULL , `fechaDevolucion` DATE NOT NULL  
 , `observaciones` TEXT NOT NULL , PRIMARY KEY (`id`)) ENGINE = InnoDB;
```

2. En nuestra aplicación, añadimos a nuestro classpath las librerías JDBC-MySQL.
3. En el paquete Modelo, añadimos una nueva clase denominada DB.java., cuya misión será conectar con la base de datos:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

```
Connection con = null;
String sURL = "jdbc:mysql://localhost:3306/empresa";
con = DriverManager.getConnection(sURL,"root","");

```

4. Creamos una nueva clase Modelo que se denomine AlquilerModel y que será la encargada de realizar las consultas contra la base de datos.

```
stmt = con.prepareStatement("INSERT INTO alquiler VALUES (?,?,?,?,?,?)");

String nCompleto = "Paco Gomez";
String nSerie = "23rfw4324";
Data fAlquiler= new Data();
Data fDevolucion = new Data();
String Observaciones = 3;

stmt.setString(1, nCompleto);
stmt.setString(2, nSerie);
stmt.setDate(3, fAlquiler);
stmt.setDate(4, fDevolucion);
stmt.setString(5, Observaciones);
```

5. Por último, importaremos el Modelo dentro de nuestro controlador y añadiremos la inserción en el evento del botón.

2.3 Integrar controles existentes en nuestros componentes

Una de las opciones que nos ofrece FXML es la de importar otros ficheros FXML dentro un fichero y, por lo tanto, poder reutilizar componentes y containers ya creados.

```
<fx:include source="filename"/>
```

La etiqueta fx:include crea justamente un objeto definido en otro fichero FXML. Un ejemplo de uso lo tenemos en la propia documentación:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns:fx="http://javafx.com/fxml">
    <children>
        <fx:include source="my_button.fxml"/>
    </children>
</VBox>
```

Donde el botón tendrá la siguiente forma:

```
<?import javafx.scene.control.*?>
<Button text="My Button"/>
```

Las opciones que nos permite esta etiqueta son múltiples:

- Definir controladores y controles reutilizables.

- Definir un sistema de layouts con plantillas. Estas pueden tener el esqueleto principal y, posteriormente, mediante los controles concretos, modificar la funcionalidad de las interfaces.

Por lo tanto, será de crucial importancia que los controladores estén definidos de acuerdo a estas definiciones y asociaciones.

2.4 Prueba de los componentes

Una vez que los componentes han sido desarrollados, debemos asegurarnos de que funcionen correctamente antes de integrarlos en la aplicación final o empaquetarlos para su distribución. La prueba de componentes en JavaFX consiste en verificar que cada parte de la interfaz gráfica de usuario (GUI) opere según lo esperado, tanto individualmente como en conjunto con otros componentes.

Dentro de las pruebas a componentes, tenemos distintas técnicas de prueba:

- **Pruebas unitarias en JavaFX:** se utilizan herramientas como JUnit para probar funciones individuales de los componentes JavaFX. Por ejemplo, se pueden crear pruebas para verificar que un botón realiza la acción correcta cuando se hace clic en él, o que un campo de texto maneja la entrada de datos adecuadamente.
- **Pruebas de integración en JavaFX:** estas pruebas se centran en comprobar cómo interactúan los diferentes componentes entre sí dentro de la aplicación. Por ejemplo, se podría probar si la selección de un elemento en un ComboBox actualiza correctamente un Label o si la activación de un evento en un Button cambia el estado de un CheckBox.
- **Automatización de pruebas con TestFX:** TestFX nos permite la automatización de pruebas de interfaces gráficas en JavaFX. Con esta herramienta podemos simular interacciones del usuario, como clics y entradas de teclado, y verificar que la interfaz responde como se espera.

En esta unidad nos centraremos en las pruebas unitarias. JUnit funciona por el sistema de notaciones que estructura las pruebas en JavaFX.

Por ejemplo, @Test marca un método como una prueba unitaria que se ejecutará automáticamente cuando se ejecuten las pruebas. @BeforeEach y @AfterEach se utilizan para configurar y limpiar el entorno antes y después de cada prueba, garantizando que cada prueba se ejecute en un estado controlado.

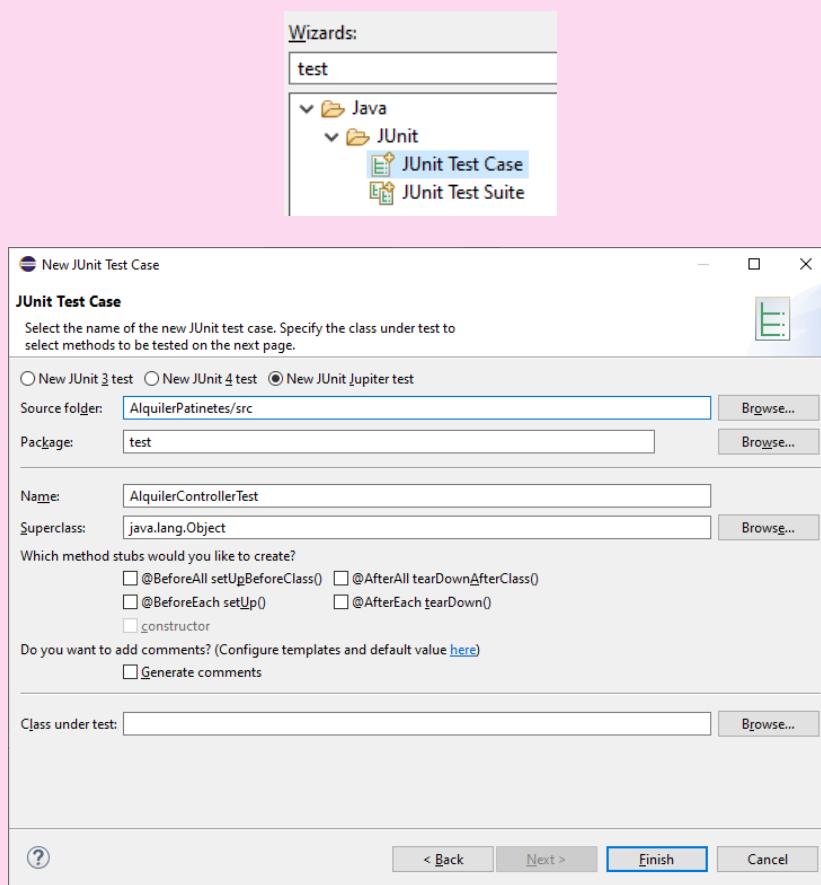
Para inicializar recursos compartidos o realizar configuraciones globales, se emplean `@BeforeAll` y `@AfterAll`, que se ejecutan una vez, antes y después de todas las pruebas. Estas anotaciones facilitan la creación de pruebas repetibles y confiables, asegurando que los componentes de JavaFX se comporten como se espera en distintos escenarios de prueba.



EJEMPLO PRÁCTICO

Nuestro jefe nos solicita diseñar una prueba unitaria para la interfaz de alquiler de patinetes desarrollada previamente y así garantizar su correcto funcionamiento. ¿Cómo lo hacemos?

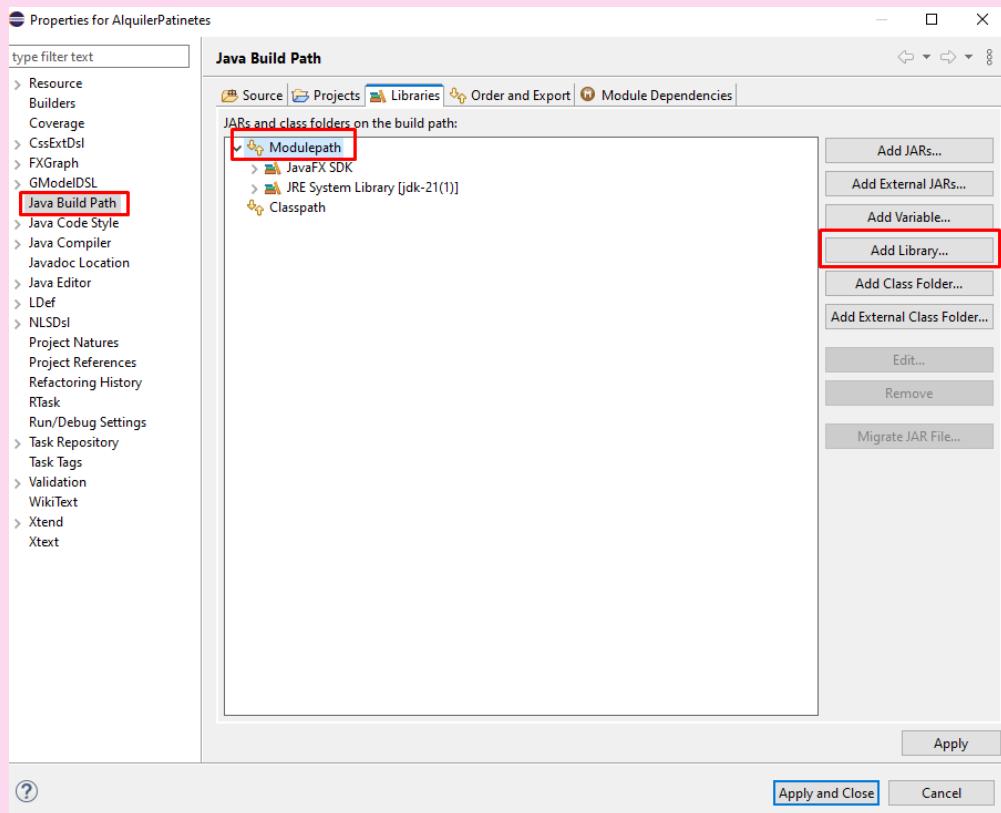
1. Diseñaremos 3 pruebas. Vamos a comprobar si se cumple que el marco se marca en rojo si no rellenamos el dato (una para nombre, otra para núm. de serie y otra para las fechas) y pulsamos en el botón Enviar.
2. El primer paso consiste en crear la clase para las pruebas. Pulsamos en el proyecto New -> Other y elegimos JUnit Test Case. En Package ponemos: test (ya que es mejor ponerlo aparte del código) y en Name: AlquilerControllerTest. Cuando aceptemos, el sistema nos preguntará si agregar las dependencias de JUnit, marcamos OK.



Configuración del Proyecto de Pruebas

Fuente: elaboración propia

- Si por cualquier razón la importación de JUnit no se hace bien, también se puede importar manualmente. Pulsamos botón derecho en el proyecto y le damos a Properties. Pulsamos en Modulepath y Add Library.



Agregando JUnit

Fuente: elaboración propia

- Marcamos JUnit y la versión 5, luego pulsamos Apply and Close.
- Ahora, una vez creado, tenemos que permitir poder acceder a las propiedades del controlador de alquiler. Como no tenemos mecanismo para ello, crearemos en la clase de alquiler getters y setters. El código quedará así:

```
package Controlador;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import javafx.scene.control.TextField;

public class AlquilerController implements Initializable{
```

```
    @FXML
    private Button botonAlquiler;
    @FXML
    private TextField nombreCompleto,numeroSerie;
    @FXML
    private DatePicker fechaRecogida,fechaDevolucion;
```

```

public void alquilerAction(ActionEvent event) {
    System.out.println("Nuevo alquiler");
    //Comprobamos el campo nombreCompleto
    if(nombreCompleto.getText().isEmpty()) {
        nombreCompleto.setStyle("-fx-control-inner-background:
#ef9a9a");
    }else if(numeroSerie.getText().isEmpty()) {
        numeroSerie.setStyle("-fx-control-inner-background: #ef9a9a");
    }else if(fechaRecogida.getValue() == null) {
        fechaRecogida.setStyle("-fx-control-inner-background: #ef9a9a");
    }else if(fechaDevolucion.getValue() == null) {
        fechaDevolucion.setStyle("-fx-control-inner-background:
#ef9a9a");
    }else {
        System.out.println("Todos los campos tienen un valor");
    }
}

public Button getBotonAlquiler() {
    return botonAlquiler;
}

public void setBotonAlquiler(Button botonAlquiler) {
    this.botonAlquiler = botonAlquiler;
}

public TextField getNombreCompleto() {
    return nombreCompleto;
}

public void setNombreCompleto(TextField nombreCompleto) {
    this.nombreCompleto = nombreCompleto;
}

public TextField getNumeroSerie() {
    return numeroSerie;
}

public void setNumeroSerie(TextField numeroSerie) {
    this.numeroSerie = numeroSerie;
}

public Datepicker getFechaRecogida() {
    return fechaRecogida;
}

public void setFechaRecogida(Datepicker fechaRecogida) {
    this.fechaRecogida = fechaRecogida;
}

public Datepicker getFechaDevolucion() {
    return fechaDevolucion;
}

public void setFechaDevolucion(Datepicker fechaDevolucion) {
    this.fechaDevolucion = fechaDevolucion;
}

@Override
public void initialize(URL arg0, ResourceBundle arg1) {
    System.out.println("Inicializo Eventos y otros elementos");
    botonAlquiler.setOnAction(this::alquilerAction);
}
}

```

6. Una vez ya podemos acceder a las propiedades, podemos definir bien nuestro fichero de pruebas Unitarias:

```

package test;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import Controlador.AlquilerController;
import javafx.scene.control.TextField;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Button;
import javafx.embed.swing.JFXPanel;
import javafx.event.ActionEvent;
import java.time.LocalDate;

public class AlquilerControllerTest {

    private AlquilerController controller;
    private TextField nombreCompleto;
    private TextField numeroSerie;
    private DatePicker fechaRecogida;
    private DatePicker fechaDevolucion;
    private Button botonAlquiler;

    @BeforeAll
    public static void initJFX() {
        // Inicializa el entorno de JavaFX
        new JFXPanel(); // Esto inicializa el toolkit de JavaFX
    }

    @BeforeEach
    public void setUp() {
        controller = new AlquilerController();
        nombreCompleto = new TextField();
        numeroSerie = new TextField();
        fechaRecogida = new DatePicker();
        fechaDevolucion = new DatePicker();
        botonAlquiler = new Button();

        controller.setNombreCompleto(nombreCompleto);
        controller.setNumeroSerie(numeroSerie);
        controller.setFechaRecogida(fechaRecogida);
        controller.setFechaDevolucion(fechaDevolucion);
        controller.setBotonAlquiler(botonAlquiler);
    }

    @Test
    public void testAlquilerActionNombreCompletoVacio() {
        nombreCompleto.setText("");
        numeroSerie.setText("123456");
        fechaRecogida.setValue(LocalDate.now());
        fechaDevolucion.setValue(LocalDate.now().plusDays(1));

        controller.alquilerAction(new ActionEvent());

        assertEquals("-fx-control-inner-background: #ef9a9a",
        controller.getNombreCompleto().getStyle());
    }
}

```

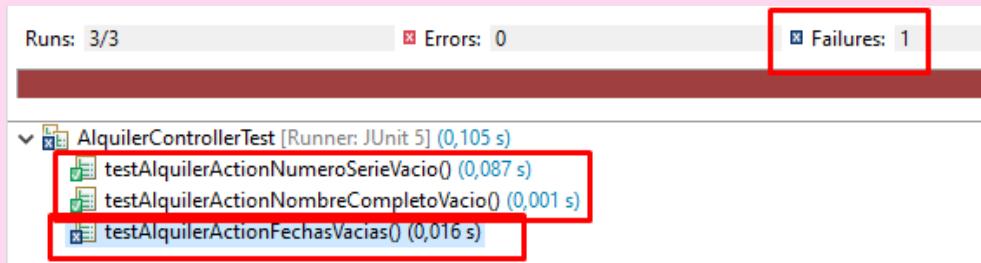
```

    @Test
    public void testAlquilerActionNumeroSerieVacio() {
        controller.getNombreCompleto().setText("Juan Perez");
        controller.getNumeroSerie().setText("");
        controller.getFechaRecogida().setValue(LocalDate.now());
        controller.getFechaDevolucion().setValue(LocalDate.now().plusDays(1));
        controller.alquilerAction(new ActionEvent());
        assertEquals("-fx-control-inner-background: #ef9a9a",
        controller.getNumeroSerie().getStyle());
    }

    @Test
    public void testAlquilerActionFechasVacias() {
        controller.getNombreCompleto().setText("Juan Perez");
        controller.getNumeroSerie().setText("123456");
        controller.getFechaRecogida().setValue(null);
        controller.getFechaDevolucion().setValue(null);
        controller.alquilerAction(new ActionEvent());
        assertEquals("-fx-control-inner-background: #ef9a9a",
        controller.getFechaRecogida().getStyle());
        assertEquals("-fx-control-inner-background: #ef9a9a",
        controller.getFechaDevolucion().getStyle());
    }
}

```

- Una vez hemos creado las pruebas, ya solo queda ejecutarlas. Debemos pulsar en Run -> Run As -> JUnit Test y podremos ver el resultado de las pruebas.



Resultado de las pruebas
Fuente: elaboración propia

- Como podemos observar, las primeras dos pruebas han pasado correctamente, mientras que la tercera prueba no la ha pasado. Ahora deberíamos pasarle estos resultados al programador del componente para que analice y corrija su comportamiento.

2.5 Empaquetado de componentes

El empaquetado de componentes consiste en proporcionar los componentes en forma de paquetes. Estos paquetes están formados por los componentes, así como por todas las bibliotecas de las que depende y otros tipos de ficheros, de forma que se proporcionan como un conjunto.

El usuario percibe el paquete como un conjunto que representa al componente en sí, cuando en realidad incluye varios ficheros.

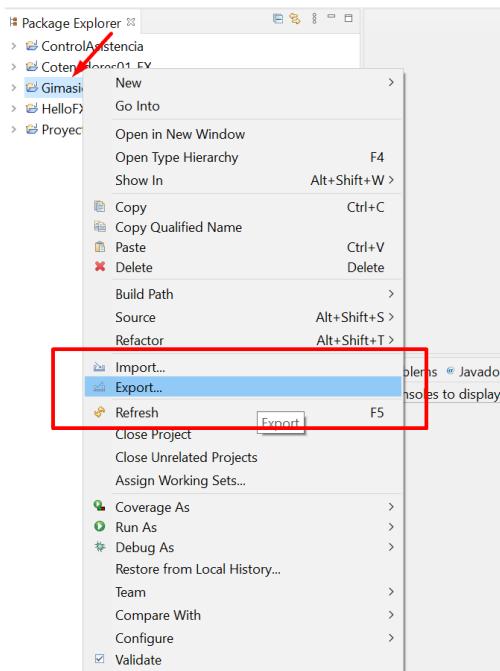
El empaquetado de componentes permite evitar los problemas de las dependencias tanto a la hora de instalar el componente como a la hora de usarlo.

Su principal ventaja es precisamente que se evita la problemática de las dependencias y que el componente se puede trasladar de un computador a otro sin necesidad de reinstalarlo, ya que el paquete contiene todos los ficheros necesarios para ejecutarlo. Sin embargo, como desventaja, se presenta que estos paquetes ocupan mucho más espacio en el disco, especialmente si el paquete incluye bibliotecas.

Dentro de Java, una forma de realizar un empaquetado listo para ser usado en cualquier otro desarrollo o bien como aplicación independiente es realizar un fichero .jar.

Veamos el paso a paso para la creación de este empaquetado con todas las librerías necesarias:

1. Sobre el proyecto que queremos generar el programa, hacemos click derecho y pulsamos sobre el menú “Exportar”.

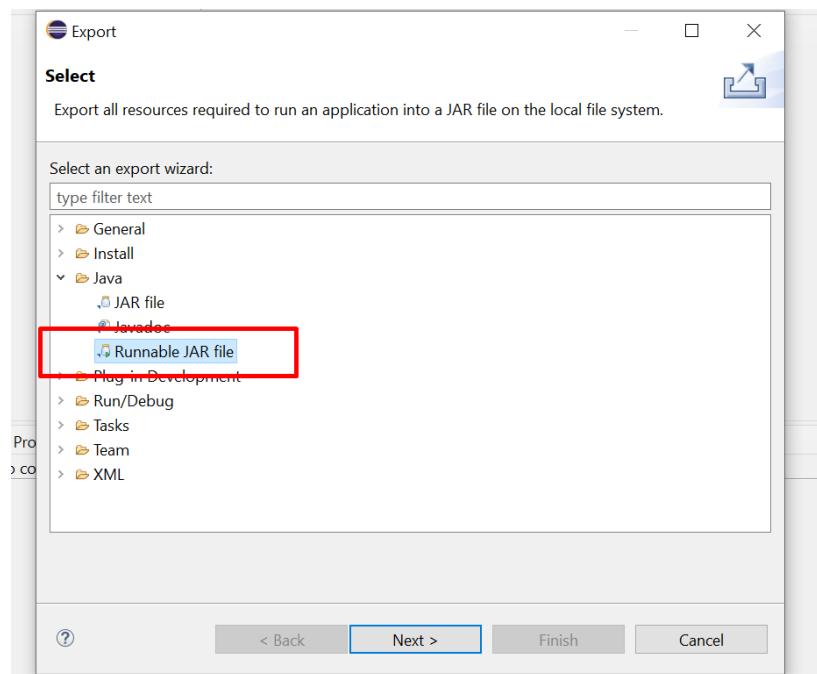


Paso 1. Creación paquete ejecutable

Fuente: elaboración propia

2. Dentro del menú Java, nos encontramos las dos opciones para crear un paquete JAR. La primera crea un empaquetado únicamente de nuestra aplicación, por lo que para poder usar el software deberemos tener en cuenta que necesitaremos

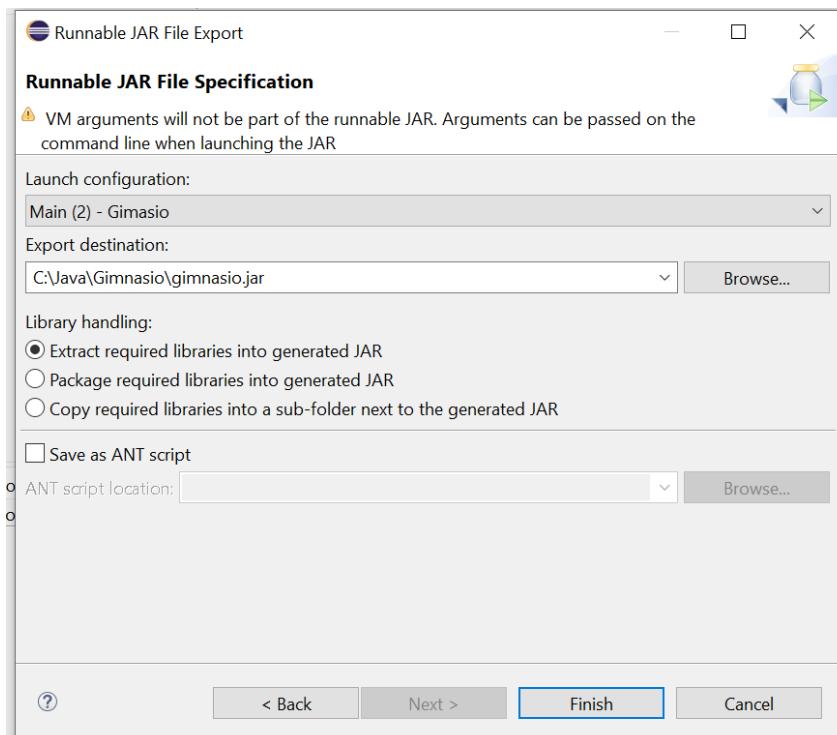
las librerías usadas. En la segunda opción, la que usaremos tal y como muestra la imagen, incluiremos todo lo necesario para que nuestro paquete se ejecute de forma autónoma.



Paso 2. Creación paquete ejecutable

Fuente: elaboración propia

3. En el último wizard, nos encontramos con las últimas configuraciones necesarias.



Paso 3. Creación paquete ejecutable

Fuente: elaboración propia



VÍDEO DE INTERÉS

Comprueba cómo empaquetar una aplicación Java en un JAR:



RESUMEN FINAL

En el desarrollo de software, las interfaces de desarrollo se han convertido en pieza clave para el éxito de una aplicación, ya que es el mecanismo de comunicación del usuario con los datos y la presentación de los mismos.

Con JavaFX, podemos desarrollar las interfaces de usuario y, para ello, debemos tener en cuenta tres aspectos: los componentes, sus propiedades y diseño y, por último, su interacción con la lógica.

Mediante Scene Builder, tenemos la herramienta perfecta para poder realizar las dos tareas más importantes en la creación de los interfaces: podemos diseñar de una forma gráfica, pero también podemos interactuar con las propiedades de una forma detallada y profunda a través de los diferentes paneles.

Una vez que los diseños están finalizados, es importante definir su relación con la lógica, y este aspecto se realiza principalmente a través de los eventos y definiendo los manejadores de estos, proceso que se combinará entre el IDE Eclipse y la herramienta Scene Builder.

El desarrollo realizado de esta forma nos permite la reutilización de software, implementado en módulos, para la resolución de problemas idénticos. Este paradigma ahorra el tiempo de desarrollo y mejora el mantenimiento del software, ya que tenemos subdividido el problema en paquetes, en controladores y lógica, y al final en vistas e interfaces. Una vez terminado, es necesario probar los componentes para garantizar su funcionamiento. Dentro de las pruebas, tenemos unitarias, de integración (que evalúa los componentes incorporados) y automatizadas. Las pruebas unitarias las realizamos con JUnit.

En el último paso, el empaquetado de componentes consiste en proporcionar los componentes en forma de paquetes. Generar un empaquetado correcto, de acuerdo con las necesidades de instalación, es clave para la última fase de desarrollo de nuestra aplicación.