DESARROLLO DE INTERFACES
2º DAM

I.E.S. POLITÉCNICO H.LANZ
JOSÉ MARÍA MOLINA

TEMA 5 – COMPONENTES VISUALES

3 COMPONENTES VISUALES

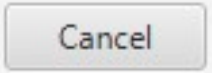
Un Componente Visual es una **parte individual** de una interfaz gráfica, un **elemento gráfico** con una apariencia y función particular. Hemos visto muchos ejempos (Label, TextField, Button…). Pero también se puede considerar como componente una **interfaz más compleja (conjunto de componentes) que cumple una única misión**.

Aquí vamos a **PERSONALIZAR** componentes y a **REUTILIZAR** interfaces ya creadas. Además, veremos otra utilidad asociada a los **Properties**: **asociación** intra/inter nodos mediantes los **BINDINGS**.

- ✓ 1 COMPONENTES VISUALES
- ✓ 2 PERSONALIZACIÓN
- ✓ 3 REUTILIZACIÓN
- ✓ 4 BINDINGS
- ✓ 5 ÚLTIMOS PASOS

1 – COMPONENTES VISUALES

✓ **Componente Visual:** es una **parte individual** de una interfaz gráfica, un **elemento gráfico** con una apariencia y función particular.



✓ **Ventajas**:

– Personalización para **Reutilización** (no confundir con utilización!!): por ejemplo, usar un Button una vez es distinto a personalizar para reutilizar constantemente => si ya hay botones personalizados, será más rápido hacer la interfaz. Ej: Botón Cancelar que SIEMPRE cierre stage actual y saque un Alert.

– **Coherencia** entre aplicaciones similares : las APPs hechas por una misma empresa tendrían que tener el "mismo aspecto" y "funcionalidad similar"→**Usabilidad**

– Mejora de la **calidad/escalabilidad**: los componentes se pueden ir mejorando sucesivamente.

✓ **Desventajas**:

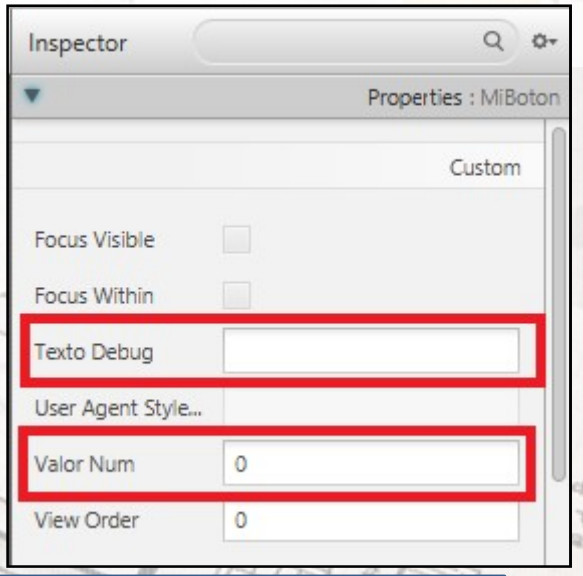
– No siempre es posible encontrar componentes adecuados para cada aplicación que queramos hacer → volver a personalizar

2 – PERSONALIZACIÓN

✓ Podemos personalizar componentes, pero no en el sentido de su utilización (apariencia y funciones) ya que esto lo hemos hecho.

✓ La idea es modificar su **comportamiento interno desde SceneBuilder, sin tener que tocar el código.** Para ello, elegimos un componente, heredamos de su clase y lo personalizamos con atributos propios según el comportamiento deseado. También habrá que crear los eventos que utilicen dichos atributos. Obtendremos variables personalizables desde Scene Builder.

```
 *
 * @author Molina
 */
public class MiBoton extends Button {

    // Propiedad para el textoDebug
    private final StringProperty textoDebug = new SimpleStringProperty();

    // Propiedad para el valorNum
    private final DoubleProperty valorNum = new SimpleDoubleProperty();

    // Constructor
    public MiBoton() {
        super(); // Llama al constructor de la clase base (Button)

    }
```



Luego vemos este ejemplo en detalle.

2 – PERSONALIZACIÓN

✓ **¿Cómo se personaliza?** En JavaFX, hay una serie de propiedades (**PROPERTY**) que son **OBSERVABLES** sirven para representar variables modificables desde SceneBuilder (y también para usar bindings).

– Podemos verlos como “variables/atributos” de clase muy versátiles.

– Algunos de los tipos de propiedades más comunes en JavaFX son:

- **SimpleStringProperty:**

```
SimpleStringProperty stringProperty = new
SimpleStringProperty("Hola Mundo!!");
```

2 – PERSONALIZACIÓN

- **SimpleIntegerProperty:**

```
SimpleIntegerProperty integerProperty = new
SimpleIntegerProperty(23);
```

- **SimpleDoubleProperty:**

```
SimpleDoubleProperty doubleProperty = new
SimpleDoubleProperty(3.1415);
```

- **SimpleBooleanProperty:**

```
SimpleBooleanProperty booleanProperty = new
SimpleBooleanProperty(true);
```

2 – PERSONALIZACIÓN

Utilizado en el proyecto básico:
TableViewFoto.
Lo usaremos en otro ejemplo

- **SimpleObjectProperty:**

```
SimpleObjectProperty<AlgunaClase> objectProperty = new
SimpleObjectProperty<>(new AlgunaClase());
```

- **SimpleListProperty:**

```
SimpleListProperty<String> listProperty = new
SimpleListProperty<>(FXCollections.observableArrayList("Elemento1", "Elemento2"));
```

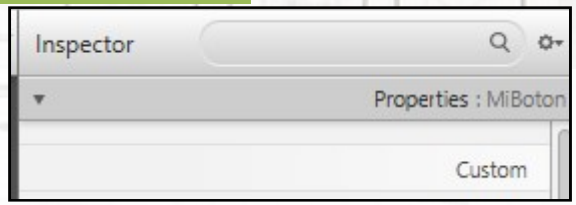
- **SimpleMapProperty:**

```
SimpleMapProperty<String, Integer> mapProperty = new
SimpleMapProperty<>(FXCollections.observableHashMap());
```

- **SimpleSetProperty:**

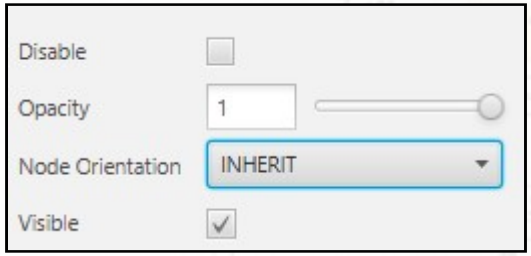
```
SimpleSetProperty<String> setProperty = new
SimpleSetProperty<>(FXCollections.observableSet("Elemento1","Elemento2"));
```

2 – PERSONALIZACIÓN



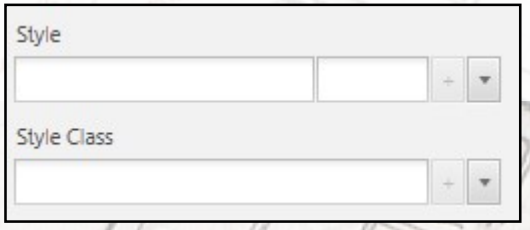
✓ **Cómo se traduce esto a SceneBuilder??:**

Podremos crear campos personalizados como los que ya existen y utilizan los nodos para su funcionamiento interno. Ejemplo de sección Node donde vemos propiedades que YA hemos usado:



- Visible y Disable: propiedad Booleana
- Opacity: propiedad Double
- Node Orientation: propiedad List

✓ En este ejemplo vemos otros atributos tipo String de la sección Node:



2 – PERSONALIZACIÓN

Abrimos el proyecto LoginPasword

✓ **Desarrollo**

– **A)** Creamos atributos personalizados y su comportamiento, por ejemplo, queremos tener 2 variables asociadas a un botón, una será utilizada para mostrar un texto para Debug y otra un valor numérico utilizado para tareas generales. En este caso se crea un string (StringProperty) y un valor real (DoubleProperty), además de sus getters y setters.

```
public class MiBoton extends Button {

    // Propiedad para el textoDebug
    private final StringProperty textoDebug = new SimpleStringProperty();

    // Propiedad para el valorNum
    private final DoubleProperty valorNum = new SimpleDoubleProperty();

    // Constructor
    public MiBoton() {
        super(); // Llama al constructor de la clase base (Button)
        inicializar();
    }

    // Constructor con texto
    public MiBoton(String text) {
        super(string:text);
        inicializar();
    }
```

```
    // Métodos getter y setter para textoDebug
    public String getTextoDebug() {
        return textoDebug.get();
    }
    public void setTextoDebug(String textoDebug) {
        this.textoDebug.set(t: textoDebug);
    }

    // Métodos getter y setter para valorNum
    public double getValorNum() {
        return valorNum.get();
    }

    public void setValorNum(double valorNum) {
        this.valorNum.set(d: valorNum);
    }

     private void inicializar() {
        // Aquí se podrían inicializar variables
    }
```

2 – PERSONALIZACIÓN

✓ Desarrollo

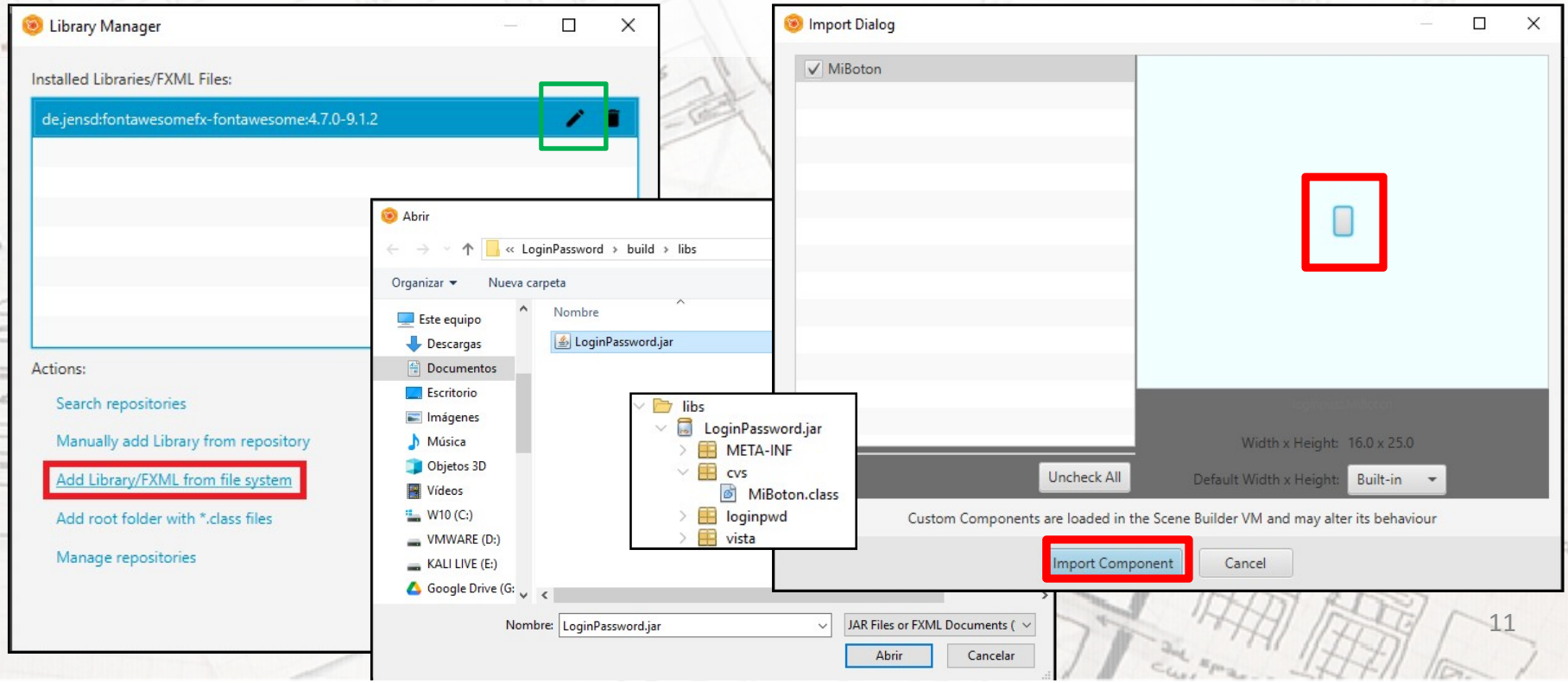
– **B)** Ahora, en el controlador, utilizamos las propiedades que hemos creado. En este caso, se usa un texto por defecto para el Debug y un número de intentos para hacer login:

```
public void enviarButton() {
    if (intentos > 1) {
        String usuario = usuarioTextField.getText();
        String password = passwordTextField.getText();

        System.out.println("Usuario: " + usuario);
        System.out.println("Password: " + password);

        System.out.println("Texto Debug: " + miBoton.getTextoDebug());
        intentos--;
        System.out.println("Intentos: " + String.valueOf(i: intentos));
        //Aquí habría que hacer la comprobación de user/password correcto
    } else {
        System.out.println(x: "ERROR! Número de intentos superado");
    }
}

@Override
public void initialize(URL url, ResourceBundle rb) {
    intentos = (int) miBoton.getValorNum();
```

– **C)** Generemos al JAR como ya sabemos (builds/libs). Importante compilar con la JDK 21 como máximo (si compilamos con la 23, no importará correctamente la clase a SB).

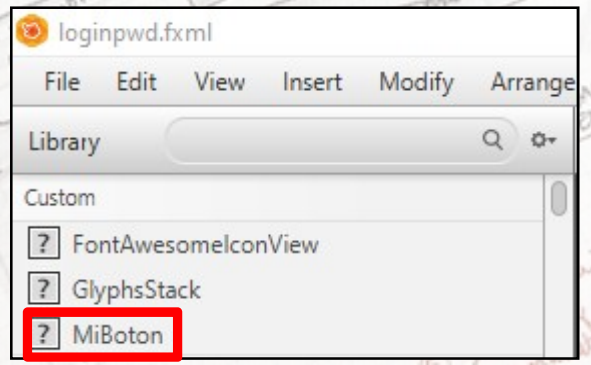
2 – PERSONALIZACIÓN

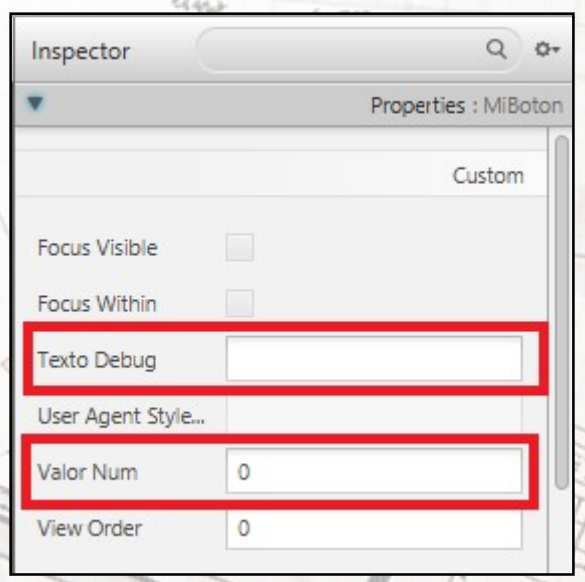
– **D)** En SceneBuilder: pinchamos en la opción *Add Library/FXML from file system* y buscamos el jar generado en el punto anterior. Ahora le damos a abrir, buscará todos los componentes que contenga. Seleccionamos el/los componente/s y finalmente pinchamos en *Import Component*.



2 – PERSONALIZACIÓN

– **E)** En este caso encuentra la clase MiBoton y la agrega a los componentes custom. Si usamos el componente en un fxml y miramos los atributos del nuevo componente encontrará los atributos personalizados que ya se definieron en el nuevo componente.





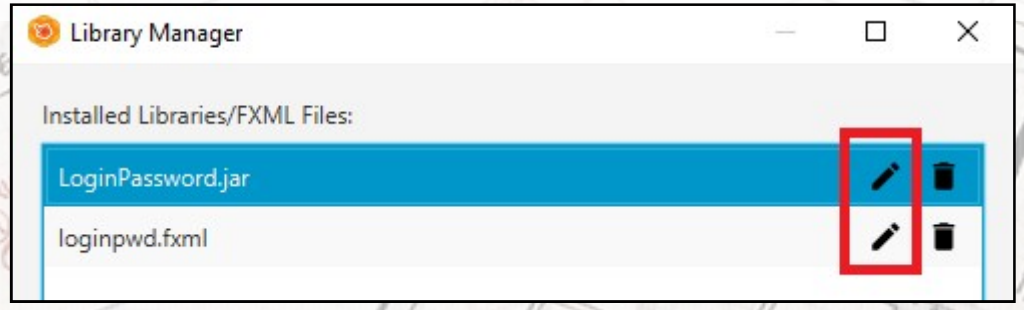
2 – PERSONALIZACIÓN

✓ **Ejemplo:**

– Contiene la clase MiBoton: se usa para importar un nuevo componente.

– Una vez importado, usamos MiBoton en Pantalla inicial de LOGIN/PASSWORD

– Si hacemos algún cambio en el componente (.jar), pinchamos en el lápiz para que relea el componente)

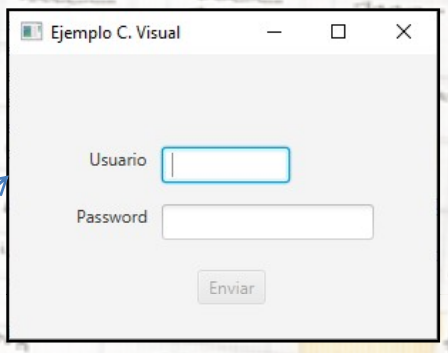


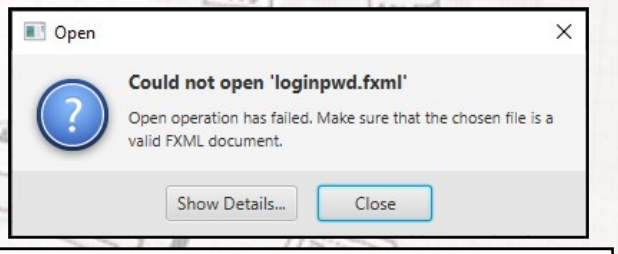
3 – REUTILIZACIÓN

✓ Podemos crear pequeñas interfaces que a su vez sean reutilizables, tales como pequeños formularios compuestos a su vez por varios componentes. Ej: loginpwd

– **A)** Lo primero es abrir el proyecto LoginPassword y compilarlo para ver que funciona sin problemas. Este proyecto tiene un FXML que será el que usemos para crear un nuevo componente en Scene Builder.

– Si intentamos abrir el FXML de este proyecto sin antes haber "registrado" miBoton entonces saldrá un error:





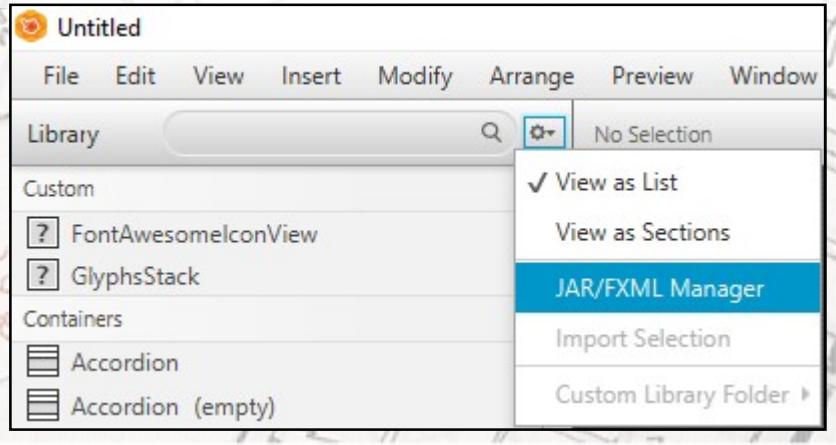

```
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.HBox?>
<?import cvs.MiBoton?>

<AnchorPane prefHeight="200.0" prefWidth="300.0"
```

3 – REUTILIZACIÓN

– Si abrimos el FXML habiendo "importado" correctamente MiBoton ya sí podremos verla, pero NO es esto lo que queremos.

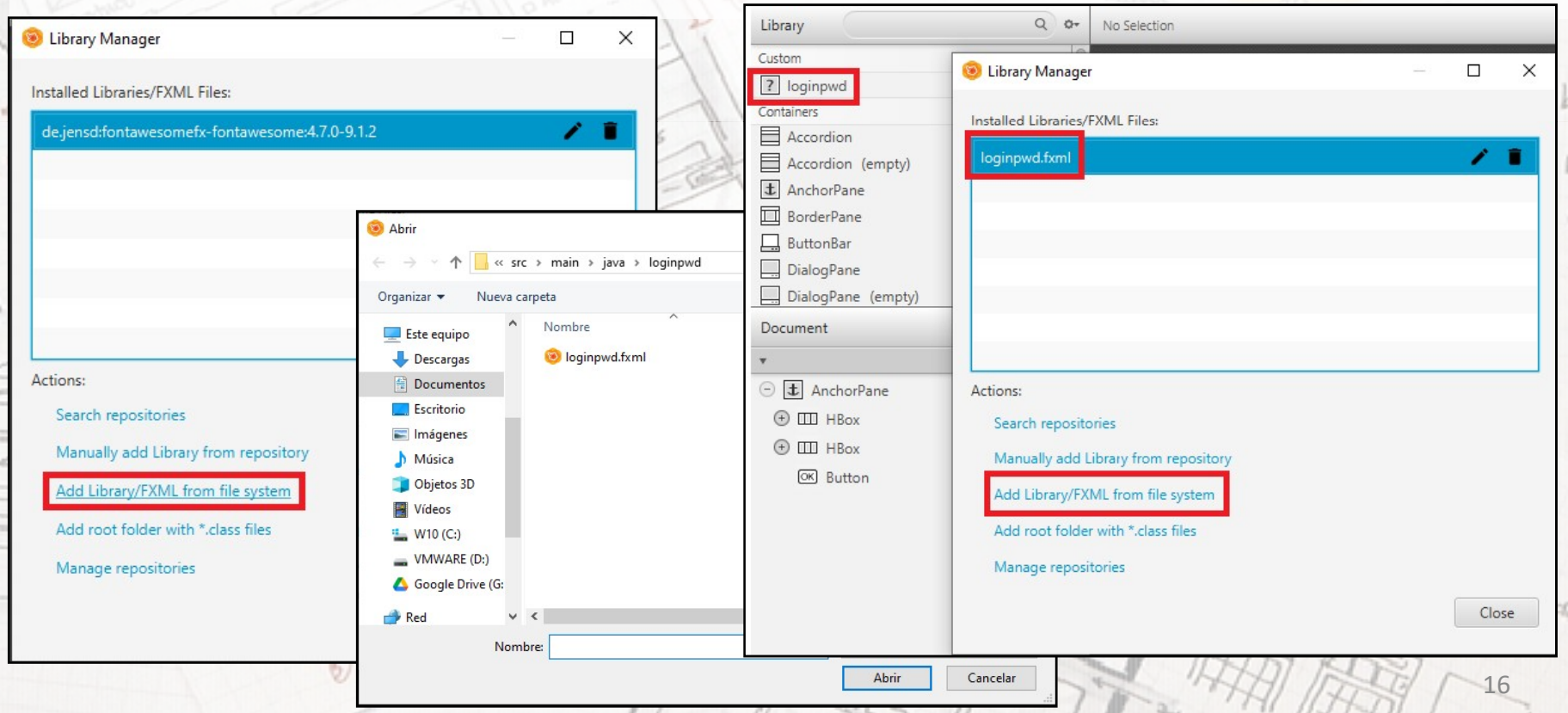
– **B)** La forma más sencilla de reutilización es mediante la **importación de un FXML a la librería**. De esta forma, podríamos utilizarlo en varios proyectos. Hay que tener en cuenta que seguirá buscando su controlador en el mismo package que se indicó en el FXML original (a no ser que se cambie).

– **C)** Para esto nos vamos a Scene Builder, lo abrimos **SIN CARGAR NINGÚN FXML (EMPTY Project)**, luego en la sección Library pinchamos en la rueda de la derecha. Abrimos la opción JAR/FXML Manager.



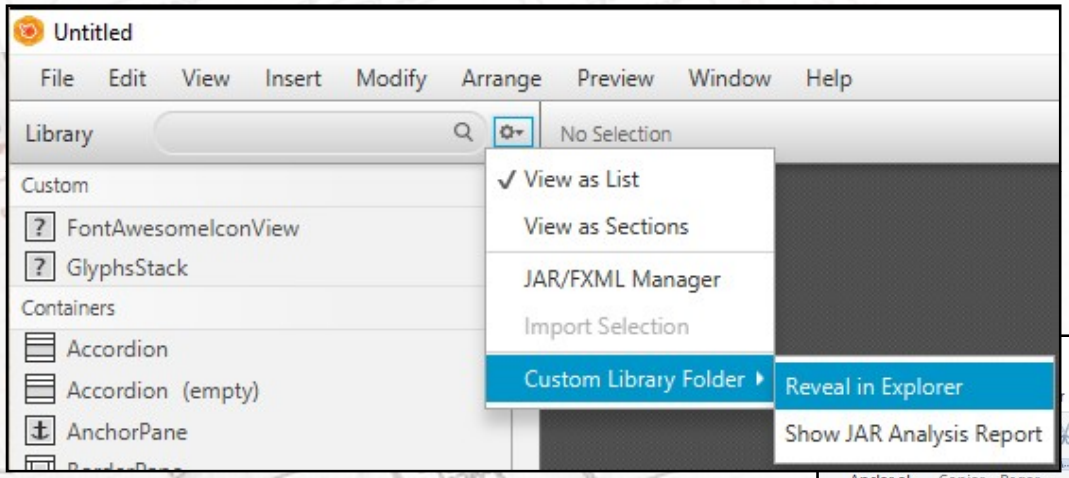
3 – REUTILIZACIÓN

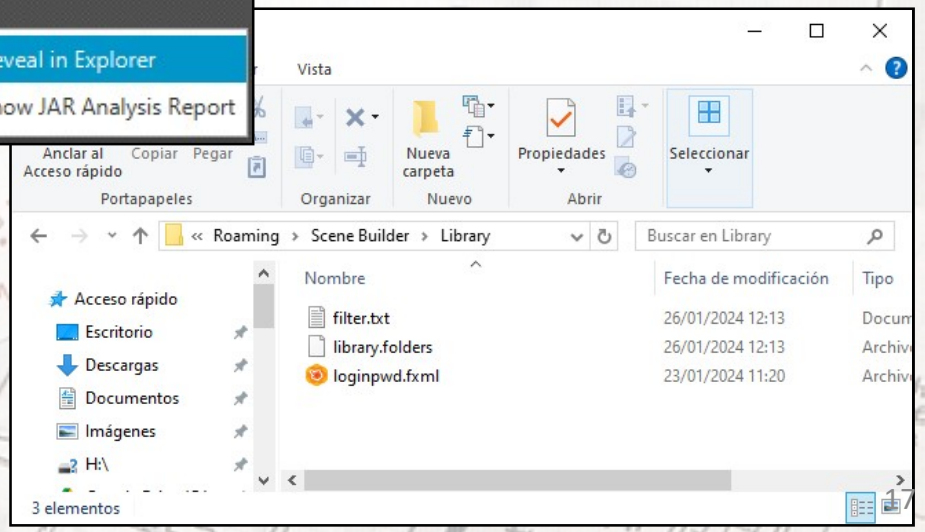
– **D)** Pinchamos en la opción Add Library/FXML from file system y buscamos el FXML comentado en el punto A) y le damos a abrir, buscará todos los componentes que contenga:



3 – REUTILIZACIÓN

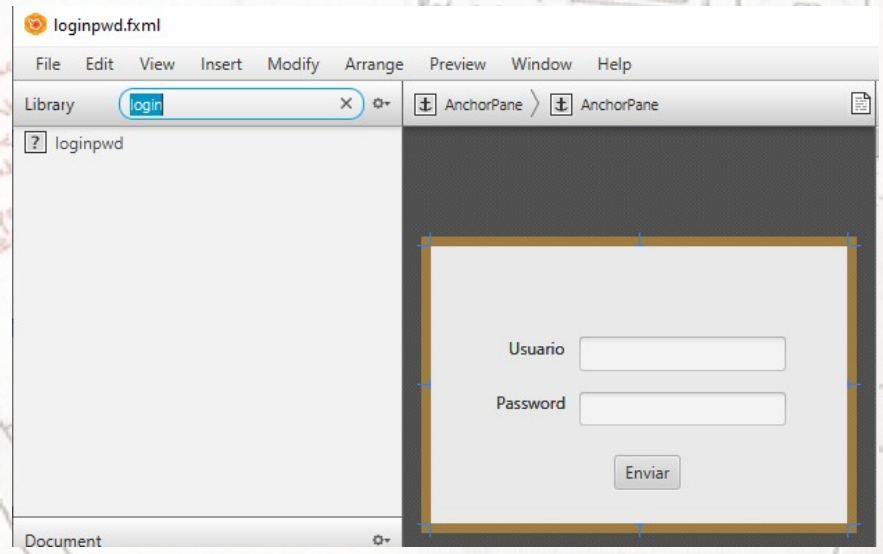
– **D)** Otra opción es pinchar en *Reveal in Explorer* y copiar ahí el FXML del punto A)



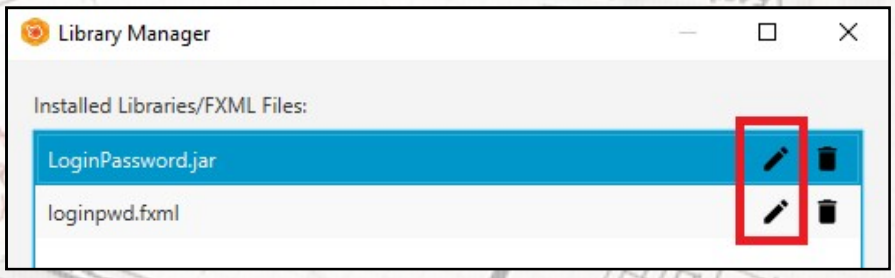


3 – REUTILIZACIÓN

– **E)** Al hacer doble click sobre el componente Custom, estaríamos pegando el FXML que hemos cargado previamente.



– Si hacemos algún cambio en el .fxml, pinchamos en el lápiz para que relea el componente)



4 – BINDINGS

- ✓ **¿Qué hace el siguiente código?**

```
checkCombo.selectedProperty().addListener((observable, valorAnt, valorAct) -> {
    mititulo.setDisable(valorAct);
});
```

```
toggleGroup.selectedToggleProperty().addListener((observable, valorAnt, valorAct) -> {
    if (valorAct != null) {
        System.out.println(observable);
        System.out.println("Pasamos de" + ((RadioButton) valorAnt).getText()
                + " a: " + ((RadioButton) valorAct).getText());
    }

});
```

```
lvSeries.getSelectionModel().selectedItemProperty().addListener((observable, viejoValor, nuevoValor) -> {
    // Manejar el evento de selección aquí
    if (nuevoValor != null) {
        System.out.println("Elemento seleccionado: " + nuevoValor);
    }
});
```

4 – BINDINGS

✓ **¿Cómo podríamos hacer?** para que si no se ha escrito en campo Nombre, que no active botón de enviar. (se puede hacer de forma "normal" con un Listener o de forma más simple con un binding).

✓ Bindings: funcionalidad que permite enlazar propiedades de los **nodos** de la GUI de forma que si cambia una → cambia la otra. Es otra forma de **personalizar** para **reutilizar** y **simplificar el código!!**

– Si el valor de la propiedad fuente cambia, automáticamente se actualiza el valor de la propiedad de destino (internamente implementa un ChangeListener).

– Desventaja→código limitado

4 – BINDINGS

✓Tipos de Bindings

– **Unidireccionales**: son los utilizados por defecto mediante la función .bind(). Una propiedad afecta a la enlazada **en una única dirección**

– **Bidireccionales**: una propiedad **afecta a la enlazada y viceversa**. Se debe utilizar la función .bindBidirectional()

4 – BINDINGS

✓ Al hacer un bind usamos un parámetro que ha de devolver un valor válido (Binding del tipo que necesite la propiedad que estamos enlazando). Ej: si boton tiene propiedad booleana (disabled) necesito devolver un BooleanBinding.

✓ Existe una clase Binding que permite hacer operaciones comunes y devolver el valor correcto.

✓ Bindings Booleanos:

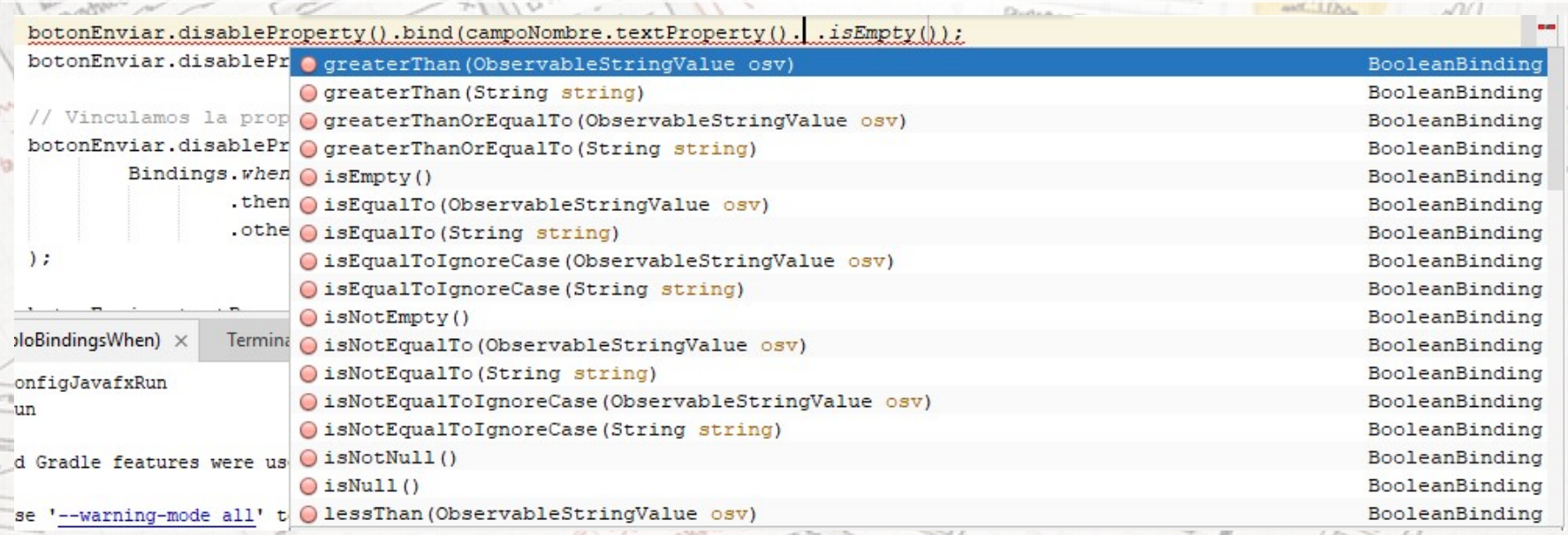
✓ **Comparaciones:** Bindings.greaterThan(…), lessThan, equal, notEqual

✓ **Condiciones:** and, or, not

✓ **Propiedades booleanas:** isEmpty

4 – BINDINGS

✓ Ejemplo, la propiedad disableProperty() se puede enlazar con otro valor, siempre y cuando el valor que se devuelva sea un BooleanBinding. En el ejemplo se ve todas las posibles funciones que podríamos utilizar:



4 – BINDINGS

– Ejemplo: si no se ha escrito en *campoNombre*, que no active *botonEnviar*. (se puede hacer de forma "normal" con un Listener como ya sabemos o de forma más simple con un binding).

```
TextField campoNombre = new TextField();
Button botonEnviar = new Button("Enviar");

// Deshabilitamos como ya sabemos
campoNombre.textProperty().addListener((observable, oldValue, newValue) -> {
    botonEnviar.setDisable(newValue.isEmpty());
});

//2 formas de simplificar el código mediante Bindings,
//Ambas líneas devuelven un valor BooleanBinding
botonEnviar.disableProperty().bind(campoNombre.textProperty().isEmpty());
botonEnviar.disableProperty().bind(Bindings.isEmpty(campoNombre.textProperty()));
```

– Tiene una pequeña desventaja y es que si queremos hacer más de una acción en el binding (bloque de código) sería más sencillo optar por programar el Listener como ya sabemos.

4 – BINDINGS

✓ Bindings **PERSONALIZADOS**:

– **Bindings.createXXXBinding(...):** Facilita la creación de vinculaciones personalizadas mediante expresiones lambda o propiedades específicas. Algunos tipos: createBooleanBinding, createIntegerBinding, createStringBinding, createDoubleBindings etc.) depende del tipo de PROPERTY que estemos manipulando.

– Ejemplo de una expresión lambda donde el tamaño del texto define el ancho de un TextField y de propiedades específicas de unos nodos que afectan a otros:

```
//Ejemplos de BINDINGS
//Binding entre propiedades de un mismo nodo
usuarioTextField.prefWidthProperty().bind(ov: Bindings.createDoubleBinding( //Valor que es devuelto: double
        () -> usuarioTextField.getText().length()*2+90 < 150 ?
                (usuarioTextField.getText().length()*2.0 + 90) : 150, //Valor que cambia (variable+mínimo)
        os: usuarioTextField.textProperty()
//En qué momento realiza el bind (cada vez que cambia la propiedad textProperty)
));

//Binding entre propiedades de varios nodos (boton y cajas de texto). Activa solo si hay datos en ambas cajas
miBoton.disableProperty().bind(ov: usuarioTextField.textProperty().isEmpty());
miBoton.disableProperty().bind(ov: passwordTextField.textProperty().isEmpty());
```

4 – BINDINGS

✓ Bindings de Cadenas:

– Bindings.concat(...), Bindings.format(...): Permiten la concatenación y el formateo de cadenas, respectivamente.

✓ Bindings Condicionales:

– Bindings.when(...).then(...).otherwise(...): Posibilita la creación de vinculaciones condicionales.

4 – BINDINGS

✓ Bindings de Selección:

– Bindings.select(...): facilita la creación de enlaces a propiedades anidadas, lo que simplifica el desarrollo de aplicaciones JavaFX . Ej: Persona->dirección

✓ Bindings de Colecciones:

– Bindings.selectList(...), Bindings.selectMap(...), Bindings.selectSet(...): Utilizados para crear vinculaciones en colecciones.

4 – BINDINGS

- ✓ Ideas para Bindings?
  - – Enlaces TableView→formulario
  - – Enlaces ListView→ formulario
  - – Control de paso formulario→botones,
  - – Subopciones dependientes de otras en formulario
  - – Etc, etc..

4 – ÚLTIMOS PASOS

✓ La dificultad de este tema está en:

– Personalizar la amplia variedad de componentes existentes y hacer esa personalización mediante la amplia variedad de *Properties* existentes.

– Hacer esa personalización mediante la variedad de *Bindings* que existen.

– Podemos consultar todas las propiedades desde la ayuda oficial, dentro de cada control nos vamos a la sección Property Summary, aparecen las propias y las heredadas. Por ejemplo, para button:

https://openjfx.io/javadoc/23/javafx.controls/javafx/scene/control/Button.html#property-summary

Property Summary

Properties

Type	Property
final BooleanProperty	cancelButton
final BooleanProperty	defaultButton

Properties declared in class javafx.scene.control.ButtonBase

armed, onAction

Properties declared in class javafx.scene.control.Labeled

alignment, contentDisplay, ellipsisString, font, graphic, graphicTextGap, textFill, textOverrun, text, textTruncated, underline, wrapText

4 – ÚLTIMOS PASOS

✓Finalmente, una vez creado el componente faltaría hacer:

– Pruebas unitarias/funcionales (ya sabemos cómo)

– Documentación de cada componente (ya sabemos cómo, usaremos un excel para esto)

– Empaquetado (en Jar para su distribución y uso con SceneBuilder, que también sabemos cómo hacerlo). **Vamos a hacer un almacén común de componentes** (enlace en Moodle)