DESARROLLO DE INTERFACES

TAREA I UD3

ALUMNO CESUR

25/26

Alejandro Muñoz de la Sierra

PROFESOR

Manuel Gómez Lora

I N T R O D U C C I O N

Crear una interfaz gráfica (GUI) implica más que colocar botones y texto en una ventana. Exige planificación para equilibrar la experiencia del usuario (UX) con un código limpio. Los desarrolladores necesitamos un mantenimiento sencillo. Planeé el proyecto con este enfoque y simulé un entorno laboral real. La prioridad fue la solidez de la aplicación.

Usé herramientas técnicas modernas para este proyecto. Elegí **Java JDK 25** (Early Access) y **JavaFX 21 (LTS).** Estas versiones permiten usar las optimizaciones más recientes de la máquina virtual. Esto presentó un reto. Tuve que configurar las dependencias y librerías de forma manual. El sistema no es modular y requiere más esfuerzo. El proceso se asemeja a la migración de sistemas antiguos.

La estructura sigue el patrón **Modelo-Vista-Controlador (MVC)**. Separé la lógica de negocio del diseño visual (.fxml) y evité mezclar responsabilidades. Integré pruebas unitarias con **JUnit 5**. Esto confirma la calidad del trabajo. Valido el formato del email y el aforo con precisión. Todo funciona antes de abrir la interfaz.

Las siguientes páginas describen mis decisiones paso a paso. Muestro desde el boceto inicial hasta el montaje en **Scene Builder**. Explico la solución a los problemas técnicos. Estos fallos surgieron al empaquetar y desplegar la aplicación en Windows.

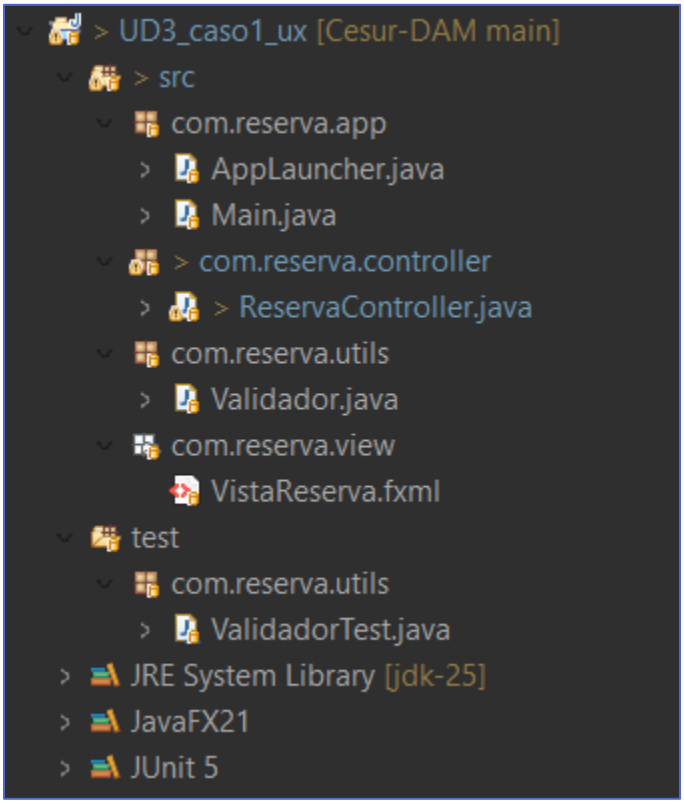
01
PREPARANDO EL TERRENO: EL ENTORNO DE DESARROLLO

Lenguaje: Elegimos Java **JDK 25**. Es una versión reciente y quisimos usar las mejoras de la JVM.

Librería Gráfica: Usamos **OpenJFX 21 (LTS)**. Preferimos una versión con soporte a largo plazo. La estabilidad evita problemas en los componentes visuales.

IDE: El proyecto usa **Eclipse IDE y Scene Builder** para la parte visual. Scene Builder acelera el diseño.

Tuvimos problemas al integrar JavaFX con Java 25. El JDK no incluye estas librerías por defecto. Configuramos una **User Library** en el **Classpath** de Eclipse para solucionar esto. Evitamos el sistema de módulos y sus errores de configuración. Es un método directo para compilar en esta fase.



0 2
F A S E D E D I S E Ñ O Y M O C K U P

Analizamos el enunciado antes de programar para esquematizar la ventana. Priorizamos la usabilidad. El usuario debe entender la pantalla de inmediato.

Creamos un boceto (wireframe) y dividimos la ventana en dos columnas. Las etiquetas están a la izquierda y los campos de texto a la derecha.

El diseño restringe los errores como muestra el esquema:

Usamos un Spinner (0-10) para el número de asistentes. Esto impide la entrada de números negativos o letras.

Incluimos un placeholder en el campo del email. Esto indica el formato esperado.

0 3
ARQUITECTURA DEL PROYECTO (PATRÓN MVC)

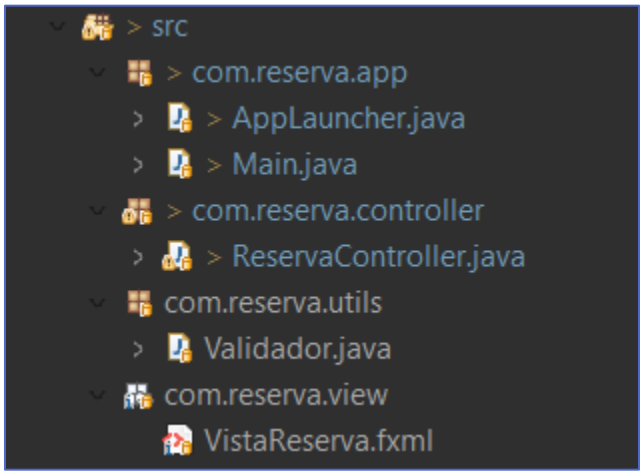
Usamos el patrón **Modelo-Vista-Controlador (MVC)** separando los datos y la lógica, de la interfaz, y de la conexión entre ambas partes, para así evitar mezclar código en un solo archivo. Organizamos el proyecto en **paquetes** según su función:

com.reserva.view: Contiene solo el archivo **VistaReserva.fxml**. Define la interfaz sin código Java.

com.reserva.controller: Contiene la clase **ReservaController**. Escucha las acciones del usuario y conecta la vista con la lógica.

com.reserva.utils: Movimos la lógica de **validación** a una clase auxiliar estática llamada Validador. Esto facilitó las pruebas posteriores.

com.reserva.app: Gestiona el **arranque**. Creamos una clase **AppLauncher** para evitar problemas de carga con las librerías de JavaFX. Estos problemas son comunes en versiones modernas.



0 4
I M P L E M E N T A C I O N D E L A I N T E R F A Z

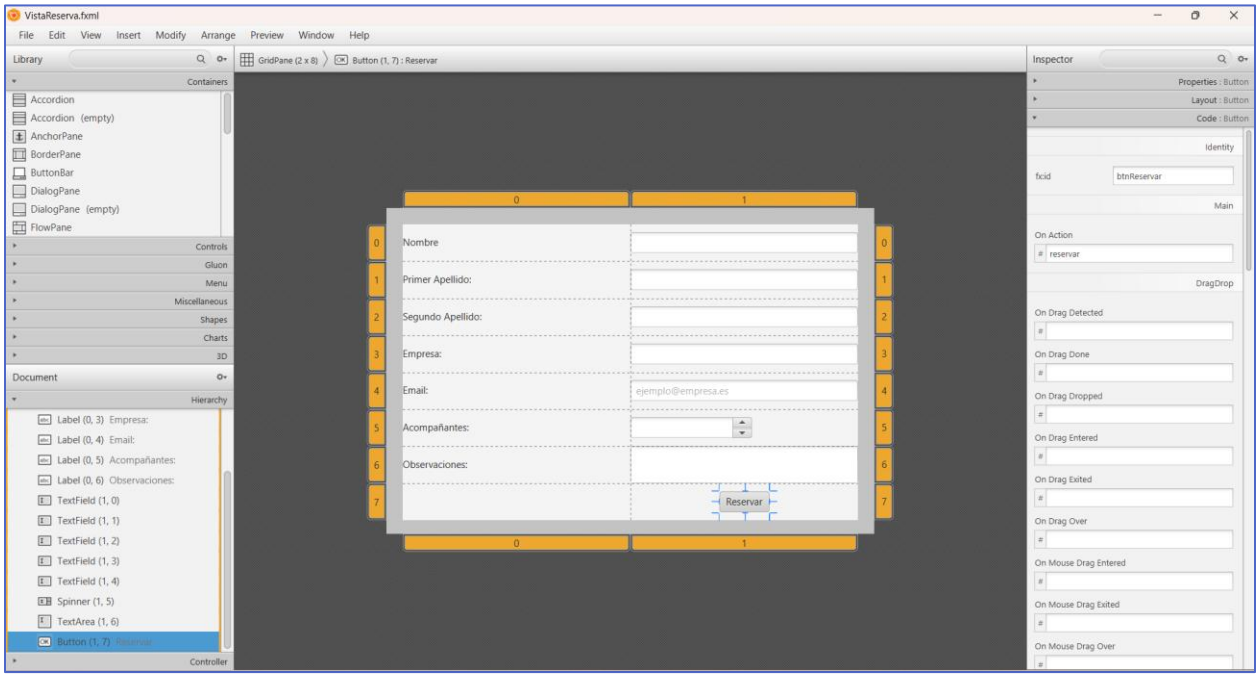
Abrimos **Scenebuilder** manualmente, creamos el archivo **VistaReserva.fxml** manualmente con la cabecera correcta y lo abrimos desde Scenebuilder, desde el que empezamos a trabajar.

4.1. Construcción de la Vista

Usamos un **GridPane** de 8 filas y 2 columnas para maquetar la ventana. A Este Grid le damos un margen (**padding**) de 20 alrededor para que quede un poco más estético.

Probamos otros contenedores. El GridPane alineó mejor las etiquetas y los campos de texto sin ajustes manuales.

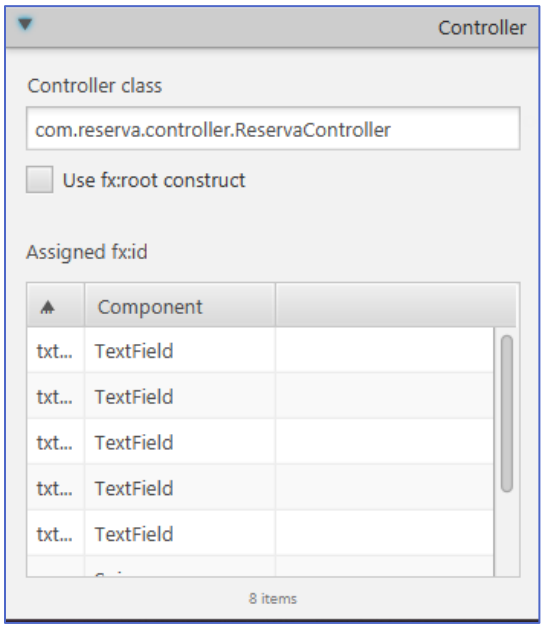
Añadimos 7 **Labels** a la izquierda y 5 **Textfields**, 1 **Spinner**, 1 **Textarea** y 1 **Button** a la derecha.

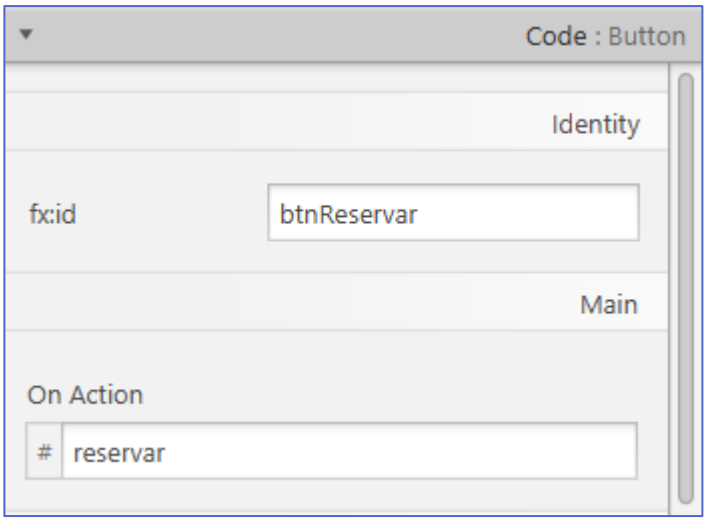


4.2. Lógica y gestión de eventos

Conectamos la interfaz a la clase **ReservaController** que llevará toda la lógica.

El funcionamiento principal reside en el método **reservar()**. El **botón** activa un filtro antes de guardar los datos.





05
IMPLEMENTACIÓN DE LAS CLASES Y ANALISIS

Explicamos la estructura de cada una de las clases en este apartado. También detallaré las razones detrás de nuestras decisiones técnicas.

5.1. Configuración del proyecto (**No-Modular**)

Esta decisión fue importante. El sistema de módulos añade complejidad desde Java 9 debido a la visibilidad y reflexión. Usamos **JavaFX 21 y JDK 25**. El uso de módulos exigía declarar las aperturas y exportaciones de cada paquete. Esto suele causar errores y consumir tiempo de configuración. Decidimos gestionar las dependencias mediante el **Classpath y User Libraries** en Eclipse. Esto nos dio flexibilidad y evitó conflictos entre el runtime de Java y JavaFX.

5.2. Explicación de las Clases

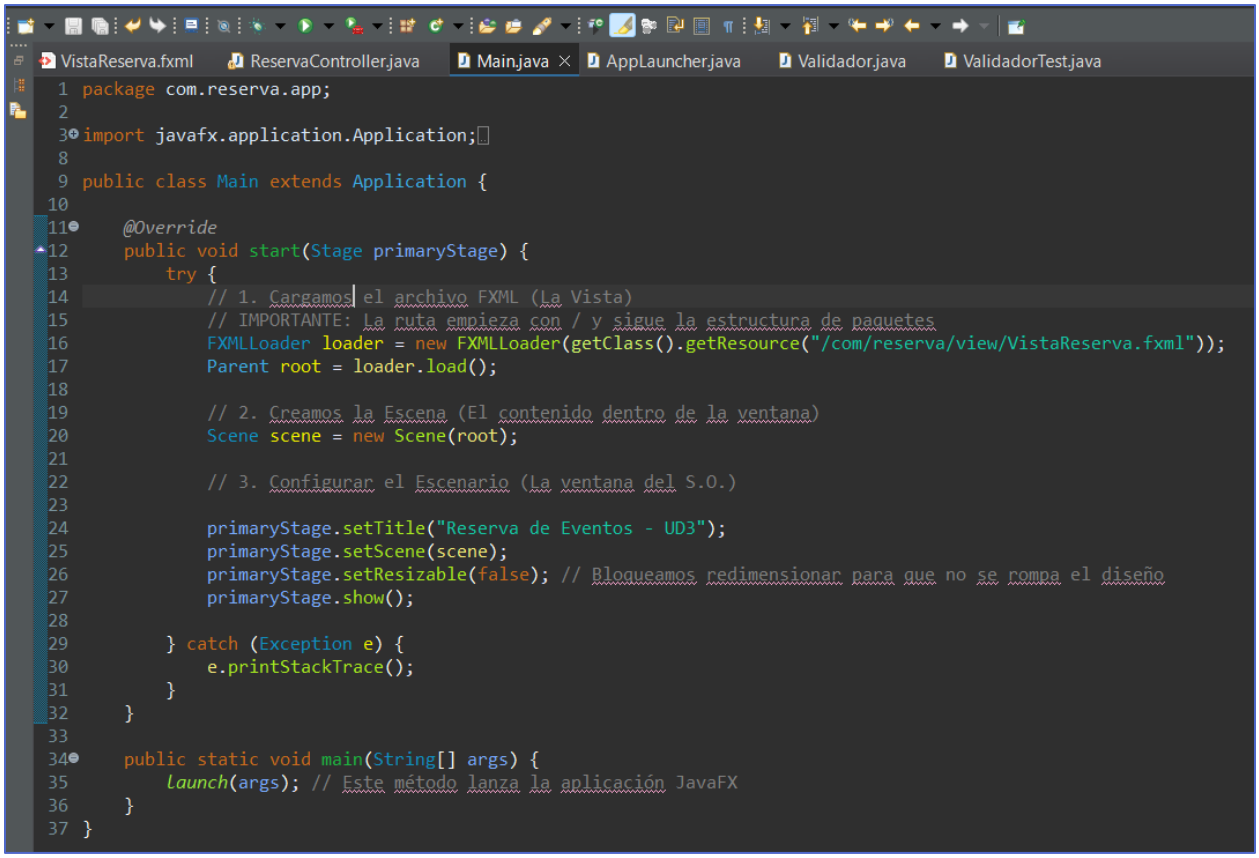
El código se divide en cuatro partes fundamentales:

A. AppLauncher.java (El "truco" para arrancar)

Esta clase es pequeña pero necesaria. La JVM a veces intenta cargar los gráficos antes que las librerías si inicias una app JavaFX desde una clase heredada de Application. Esto provoca fallos de componentes faltantes. Creamos esta clase sin herencia. Solo tiene un método main que **llama al main real**. Esta solución asegura que todo cargue en orden.

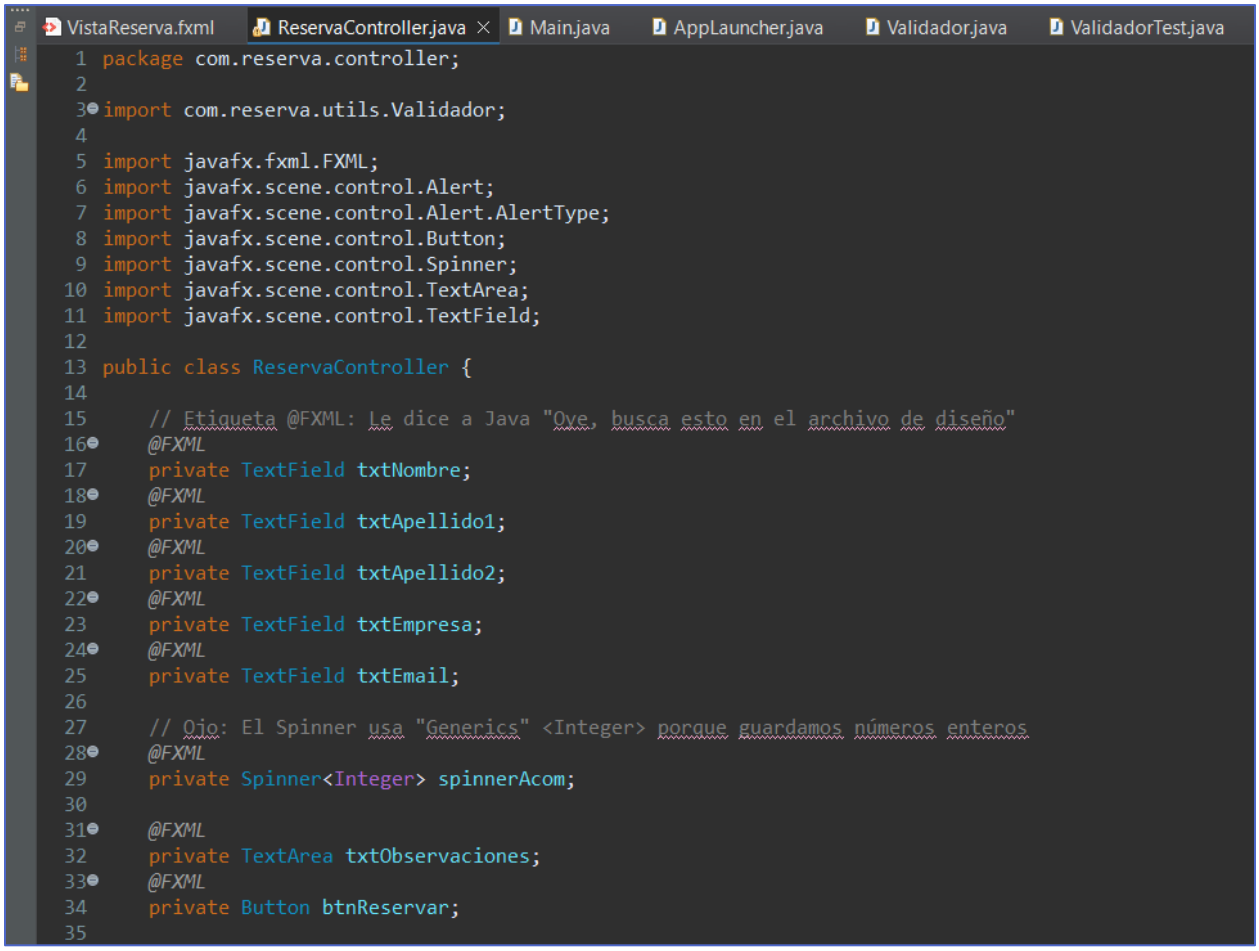
B. Main.java

Esta clase hereda de javafx.application.Application. **Carga el FXML**, lo introduce en una **Scene** y configura el. **Stage** También usamos **setResizable(false).** El usuario no podrá deformar el diseño al estirar la ventana.



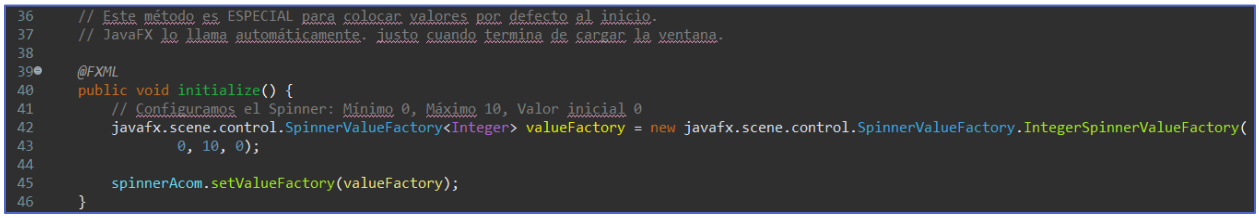
C. ReservaController.java

Este es el controlador. Usamos anotaciones **@FXML** para **vincular** botones y campos de texto con el **código**. Tiene dos métodos clave:



```java
package com.reserva.controller;

import com.reserva.utils.Validador;

import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.Spinner;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;

public class ReservaController {

    // Etiqueta @FXML: Le dice a Java "Oye, busca esto en el archivo de diseño"
    @FXML
    private TextField txtNombre;
    @FXML
    private TextField txtApellido1;
    @FXML
    private TextField txtApellido2;
    @FXML
    private TextField txtEmpresa;
    @FXML
    private TextField txtEmail;

    // Ojo: El Spinner usa "Generics" <Integer> porque guardamos números enteros
    @FXML
    private Spinner<Integer> spinnerAcom;

    @FXML
    private TextArea txtObservaciones;
    @FXML
    private Button btnReservar;

```

initialize(): Se ejecuta al inicio. Configuramos el Spinner entre 0 y 10 mediante código para mayor seguridad.



```java
    // Este método es ESPECIAL para colocar valores por defecto al inicio.
    // JavaFX lo llama automáticamente, justo cuando termina de cargar la ventana.

    @FXML
    public void initialize() {
        // Configuramos el Spinner: Mínimo 0, Máximo 10, Valor inicial 0
        javafx.scene.control.SpinnerValueFactory<Integer> valueFactory = new javafx.scene.control.SpinnerValueFactory.IntegerSpinnerValueFactory(
                0, 10, 0);

        spinnerAcom.setValueFactory(valueFactory);
    }
```

reservar(): Recoge los datos con .getText(). **Llama al Validador** y decide si muestra un mensaje de éxito o error.

```java
    @FXML
    public void reservar() {
        // 1. Recolección de datos
        String nombre = txtNombre.getText();
        String ape1 = txtApellido1.getText();
        String ape2 = txtApellido2.getText();
        String empresa = txtEmpresa.getText();
        String email = txtEmail.getText();
        String observaciones = txtObservaciones.getText();

        // El spinner devuelve un valor Integer directamente gracias a <Integer>
        Integer asistentes = spinnerAcom.getValue();

        // 2. Acumulador de errores
        // Usamos StringBuilder porque es más eficiente que sumar Strings con +
        StringBuilder errores = new StringBuilder();

        // 3. Validaciones paso a paso
        if (Validador.esCampoVacio(nombre))
            errores.append("- El nombre es obligatorio.\n");
        if (Validador.esCampoVacio(ape1))
            errores.append("- El primer apellido es obligatorio.\n");
        if (Validador.esCampoVacio(ape2))
            errores.append("- El segundo apellido es obligatorio.\n");
        if (Validador.esCampoVacio(empresa))
            errores.append("- La empresa es obligatoria.\n");

        // Validación especial del Email
        if (Validador.esCampoVacio(email)) {
            errores.append("- El email es obligatorio.\n");
        } else if (!Validador.esEmailValido(email)) {
            errores.append("- El email debe tener formato texto@dominio.es\n");
        }

        // 4. Decisión final
        if (errores.length() > 0) {
            // SI HAY ERRORES: Mostramos alerta de Error
            Alert alert = new Alert(AlertType.ERROR);
            alert.setTitle("Error en la reserva");
            alert.setHeaderText("Por favor, corrige los siguientes campos:");
            alert.setContentText(errores.toString());
            alert.showAndWait();
        } else {
            // SI TODO ESTÁ BIEN: Mostramos éxito
            Alert alert = new Alert(AlertType.INFORMATION);
            alert.setTitle("Reserva Completada");
            alert.setHeaderText(null);
            alert.setContentText("¡Reserva realizada con éxito para " + nombre + "!");
            alert.showAndWait();
        }
    }

    // Método auxiliar para mostrar ventanitas de mensaje
    private void mostrarAlerta(String titulo, String mensaje) {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle(titulo);
        alert.setHeaderText(null);
        alert.setContentText(mensaje);
        alert.showAndWait();
    }
}
```

D. Validador.java

Se encuentra en el paquete de utilidades. Solo contiene **métodos estáticos**. Esto cumple con el principio de responsabilidad única. El controlador maneja la vista y el validador realiza los cálculos.

VistaReserva.fxml | ReservaController.java | Main.java | AppLauncher.java | Validador.java

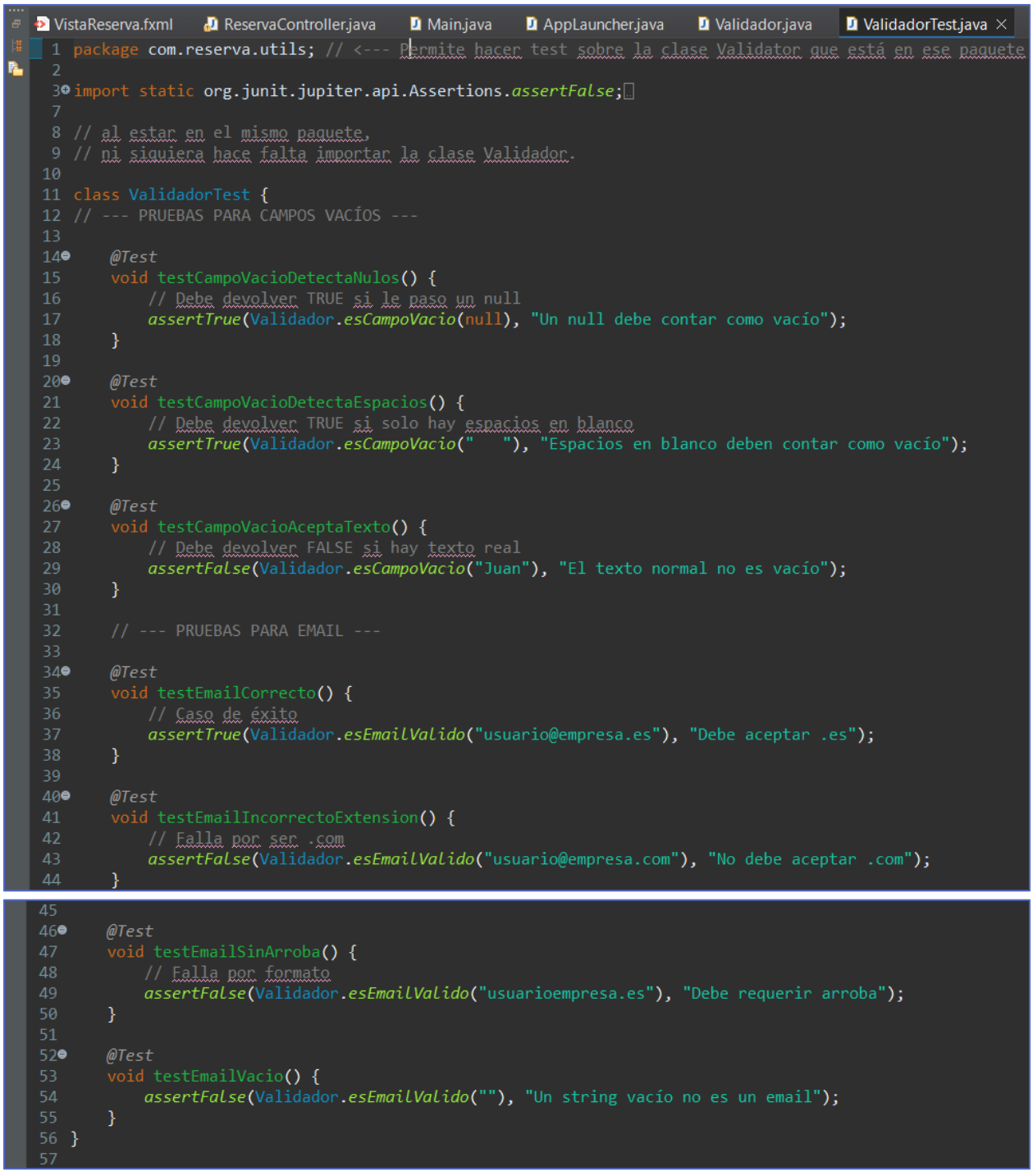
```java
package com.reserva.utils;

public class Validador {

    // Método estático: No necesita instanciar la clase para usarse
    // Método privado para validar el email con Regex
    // El enunciado pide estructura: XXXX@XXXX.es

    public static boolean esEmailValido(String email) {
        return email != null && email.matches("^[\\w-\\.]+@[\\w-]+\\.es$");

        // Explicación del Regex:
        // ^ -> Empieza aquí
        // [\\w-\\.]+ -> Cualquier letra, número, guión o punto (antes de la @)
        // @ -> Una arroba obligatoria
        // [\\w-]+ -> Cualquier letra, número o guión (el dominio)
        // \\.es -> Un punto seguido de "es" OBLIGATORIO
        // $ -> Termina aquí
    }

    // Método para comprobar si un campo está vacío o solo tiene espacios

    public static boolean esCampoVacio(String texto) {
        return texto == null || texto.trim().isEmpty();
    }
}
```

Usamos una validación estricta con **Expresiones Regulares** (Regex) para el requisito del email (XXXX@XXXX.es):

Esto obliga al dominio a terminar en ".es". Guardamos los mensajes de error en un StringBuilder y mostramos una alerta de tipo ERROR. Una ventana de tipo INFORMATION confirma si todo es correcto.

0 6
T E S T S (J U N I T) Y A N A L I S I S

No queríamos probar la aplicación solo con clics. Creamos **ValidadorTest.java**. La estrategia abordó los puntos donde suele fallar el usuario:



```java
package com.reserva.utils; // <--- Permite hacer test sobre la clase Validator que está en ese paquete

import static org.junit.jupiter.api.Assertions.assertFalse;

// al estar en el mismo paquete,
// ni siquiera hace falta importar la clase Validador.

class ValidadorTest {
// --- PRUEBAS PARA CAMPOS VACÍOS ---

    @Test
    void testCampoVacioDetectaNulos() {
        // Debe devolver TRUE si le paso un null
        assertTrue(Validador.esCampoVacio(null), "Un null debe contar como vacío");
    }

    @Test
    void testCampoVacioDetectaEspacios() {
        // Debe devolver TRUE si solo hay espacios en blanco
        assertTrue(Validador.esCampoVacio("   "), "Espacios en blanco deben contar como vacío");
    }

    @Test
    void testCampoVacioAceptaTexto() {
        // Debe devolver FALSE si hay texto real
        assertFalse(Validador.esCampoVacio("Juan"), "El texto normal no es vacío");
    }

    // --- PRUEBAS PARA EMAIL ---

    @Test
    void testEmailCorrecto() {
        // Caso de éxito
        assertTrue(Validador.esEmailValido("usuario@empresa.es"), "Debe aceptar .es");
    }

    @Test
    void testEmailIncorrectoExtension() {
        // Falla por ser .com
        assertFalse(Validador.esEmailValido("usuario@empresa.com"), "No debe aceptar .com");
    }

    @Test
    void testEmailSinArroba() {
        // Falla por formato
        assertFalse(Validador.esEmailValido("usuarioempresa.es"), "Debe requerir arroba");
    }

    @Test
    void testEmailVacio() {
        assertFalse(Validador.esEmailValido(""), "Un string vacío no es un email");
    }
}
```

-Campos vacíos o con espacios:

Si el usuario mete solo espacios, tiene que dar error

-Validación del Email: Este punto era crítico. Probamos tres cosas:

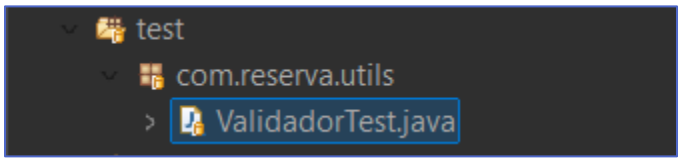
Formato correcto: usuario@empresa.es es válido.

Extensión incorrecta: usuario@empresa.com debe devolver FALSO por el requisito estricto del .es.

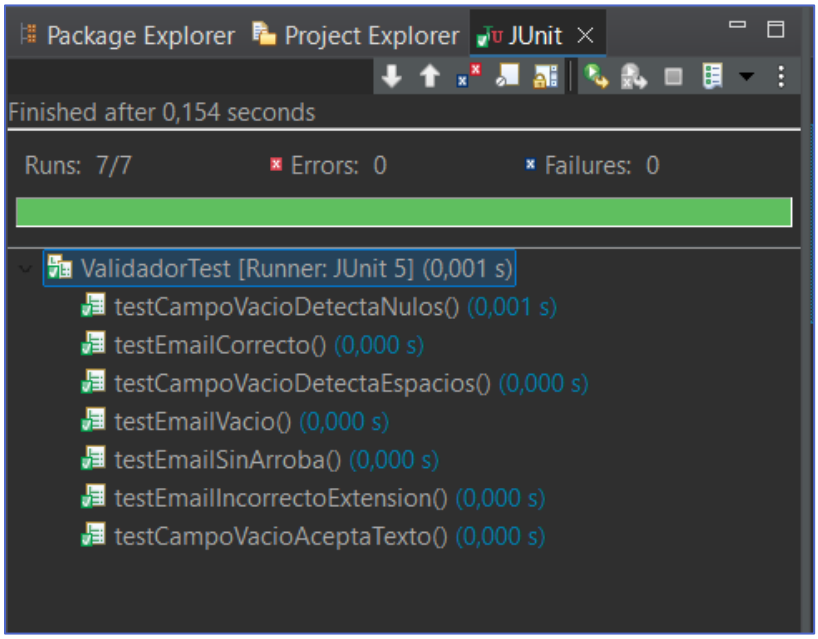
Formato roto: Correos sin arroba.

-Pasamos la batería de siete pruebas con éxito. Esto confirma que la lógica interna funciona antes de trabajar con la interfaz gráfica.

Vamos a nuestra clase con los test y la ejecutamos como JUNIT test



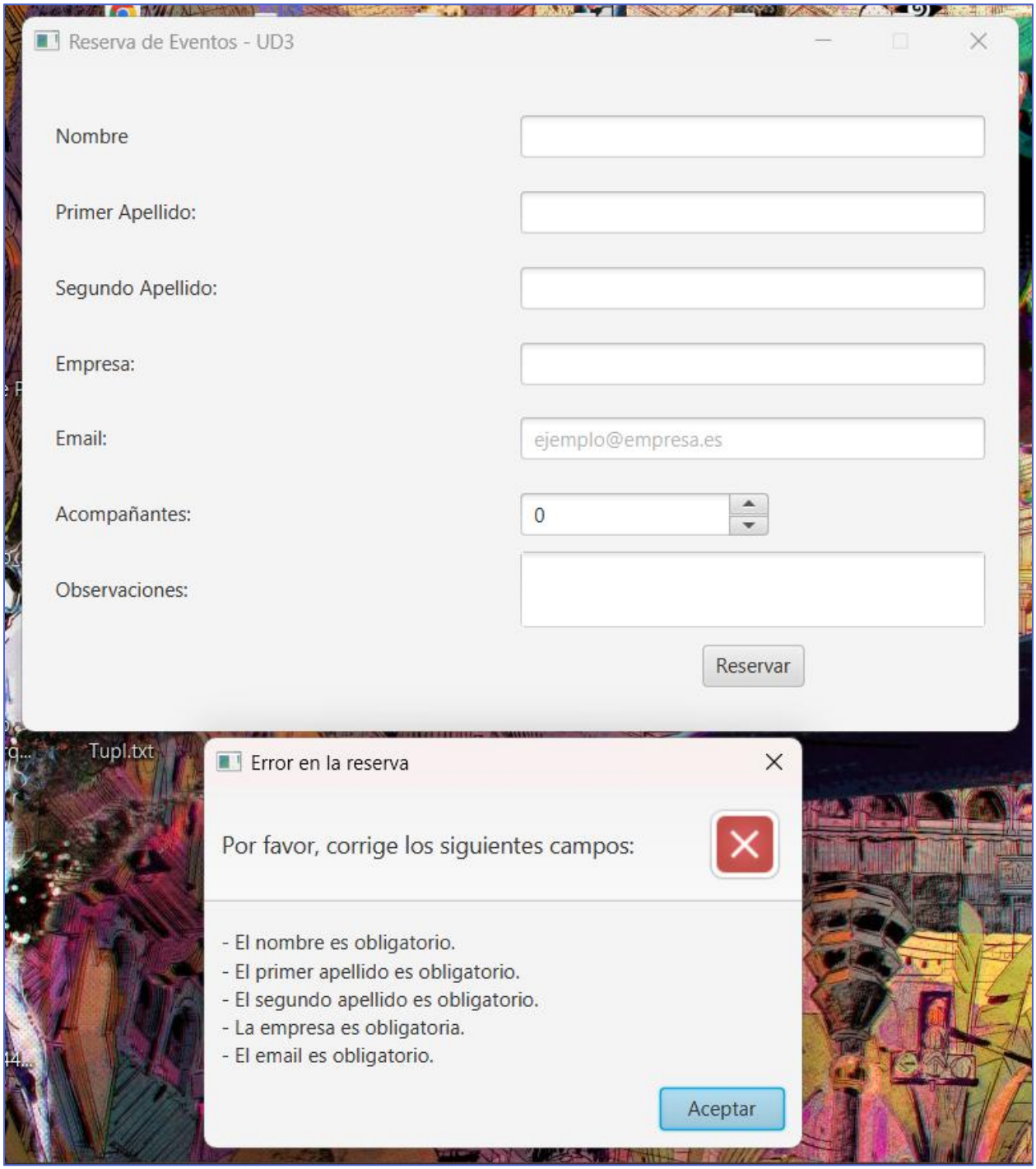
Observamos los resultados. Todos los tests son correctos:



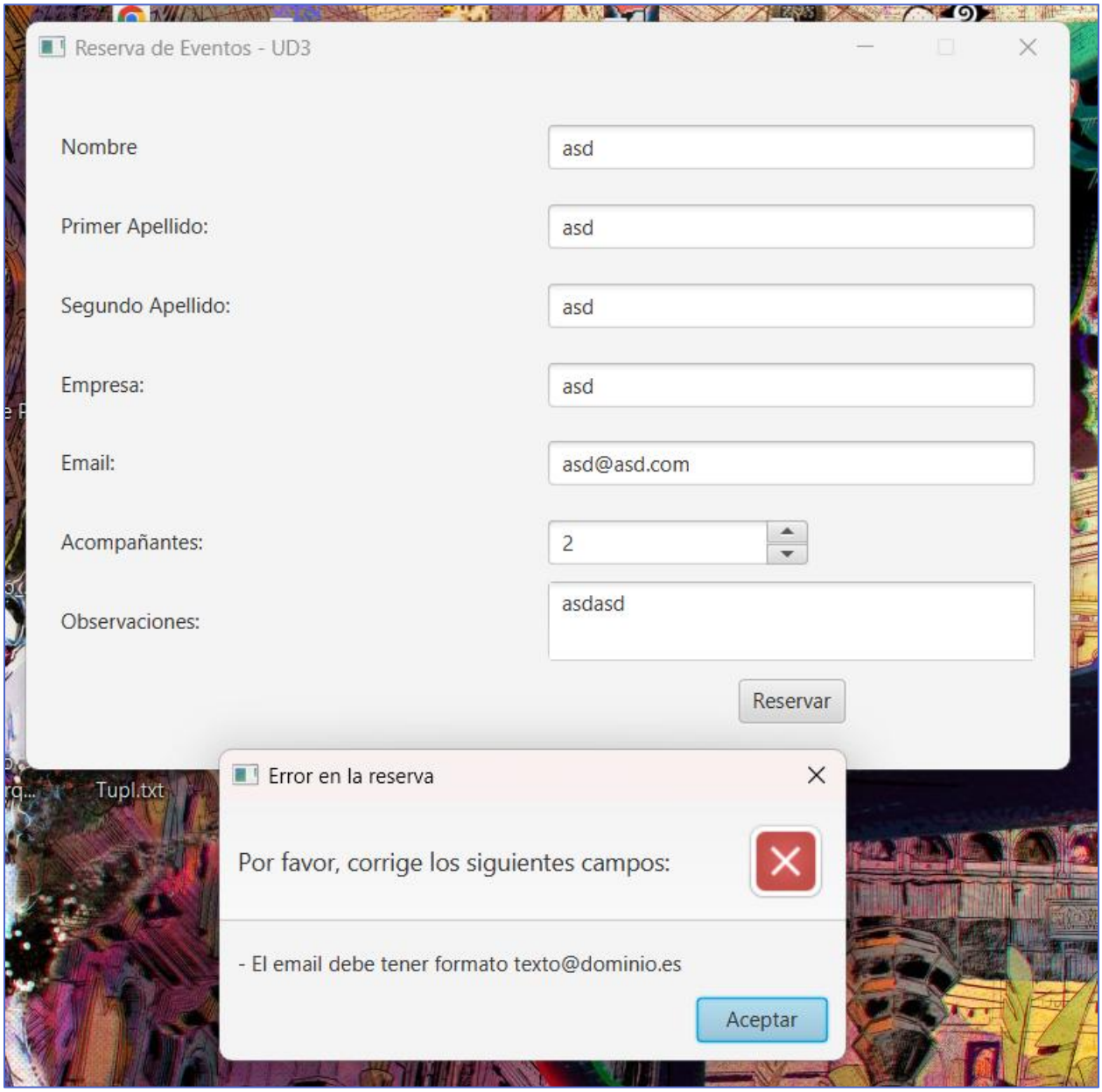
0 7
E J E C U C I Ó N Y P R U E B A S

Hacemos varias pruebas forzando errores, y otra con todos los elementos correctos para comprobar el funcionamiento.

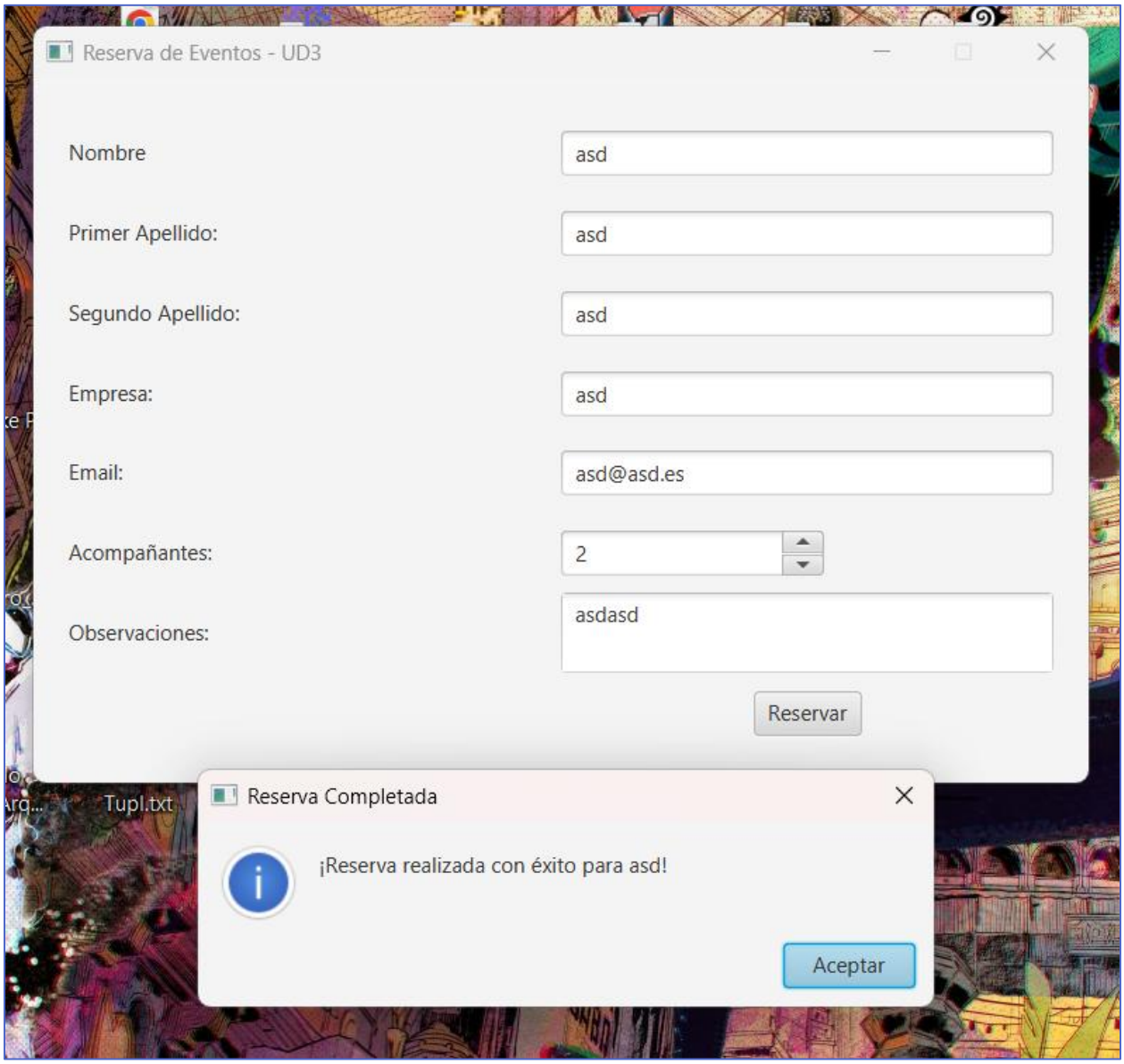
Datos incorrectos o campos faltantes



Formato de Email incorrecto

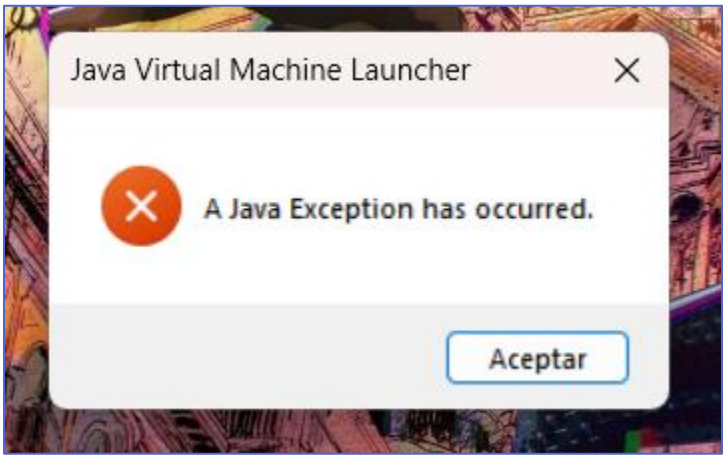


Datos correctos, sin campos faltantes



08
DESPLIEGUE (GENERACIÓN DEL JAR)

Generamos un **Runnable JAR** para la entrega final. Tuvimos multitud de problemas al exportar el proyecto JavaFX. El JAR **no abría** porque faltaban dependencias. Configuramos la exportación para empaquetar las librerías necesarias dentro del archivo. También usamos nuestra clase **AppLauncher** como punto de entrada. El archivo **ReservaEvento.jar siguió sin funcionar**, daba igual que opción de empaquetado usaramos o desde donde lo ejecutaramos.



Llegamos a la conclusión con investigación que JDK25 es demasiado restrictivo con la seguridad para ejecutar JAVAFX desde un Jar.

Entregamos el proyecto exportado con las aclaraciones pertinentes.

0 9
C O N C L U S I O N E S

Creo que asenté las bases de las interfaces gráficas en Java. Elegí un stack moderno con **JavaFX 21 y JDK 25**. El trabajo no consistió solo en maquetar ventanas. Resolví problemas reales de arquitectura y configuración. No esperaba estos retos al principio.

Destaco tres puntos útiles del proyecto:

El **patrón MVC** es importante. Crear tantos paquetes y clases parece difícil. Separar la vista (.fxml), la lógica (utils) y el control ayudó mucho. Mantiene el código limpio. Pasar las pruebas fue más sencillo. Pude validar las matemáticas sin arrancar la ventana.

Valoro la robustez. Implementé validaciones con Expresiones Regulares. Protegí el código con **JUnit 5**. Esto confirma la fiabilidad de la aplicación. Una interfaz intuitiva es inútil si el programa falla con datos extraños.

La gestión del entorno y el despliegue fue el mayor reto. Exportar el JAR ejecutable costó trabajo. El JDK 25 tiene restricciones de seguridad y acceso nativo. Aprendí mediante ensayo y error. Entendí el funcionamiento del **classpath**. Leer la documentación técnica es necesario fuera de Eclipse.

El objetivo está cumplido. La aplicación funciona y es escalable. Cumple con los requisitos. Me llevo el aprendizaje técnico. Enfrenté las incompatibilidades de las versiones recientes de Java.

10 REFERENCIAS

https://openjfx.io/

https://gluonhq.com/products/scene-builder/

https://www.geeksforgeeks.org/java/javafx-tutorial/

https://www.geeksforgeeks.org/java/javafx-tutorial/

https://docs.junit.org/6.0.1/overview.html

https://docs.oracle.com/javase/tutorial/essential/regex/

https://regex101.com/

https://www.youtube.com/watch?v=9XJicRt_FaI

https://www.youtube.com/@pildorasinformaticas/search?query=javafx

https://www.youtube.com/watch?v=cEidY4DNqgQ

https://www.youtube.com/watch?v=Y1bosCt5eJI

https://www.youtube.com/watch?v=LELiSbEFkuk