Informe: Aplicación CRUD con Arquitectura de 3 Capas y Patrón MVC

Caetano Flores
Jordan Guaman
Anthony Morales
Leonardo Narvaez

Noviembre 2025

1. Introducción

Este informe documenta el desarrollo de una aplicación CRUD (Create, Read, Update, Delete) para la gestión de estudiantes, implementando la arquitectura de 3 capas en conjunto con el patrón de diseño Modelo-Vista-Controlador (MVC). La aplicación permite administrar información básica de estudiantes (ID, nombres y edad) a través de una interfaz gráfica desarrollada en Java Swing.

2. Arquitectura del Sistema

La aplicación está estructurada siguiendo el patrón de arquitectura de 3 capas, que separa las responsabilidades en tres niveles distintos: la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Esta separación permite un código más mantenible, escalable y facilita las pruebas unitarias de cada componente de forma independiente.

El flujo de datos en la aplicación sigue una jerarquía clara: la interfaz de usuario (capa de presentación) interactúa con los servicios de negocio, los cuales a su vez se comunican con el repositorio de datos. Esta estructura garantiza que cada capa tenga una responsabilidad única y bien definida, cumpliendo con el principio de separación de responsabilidades.

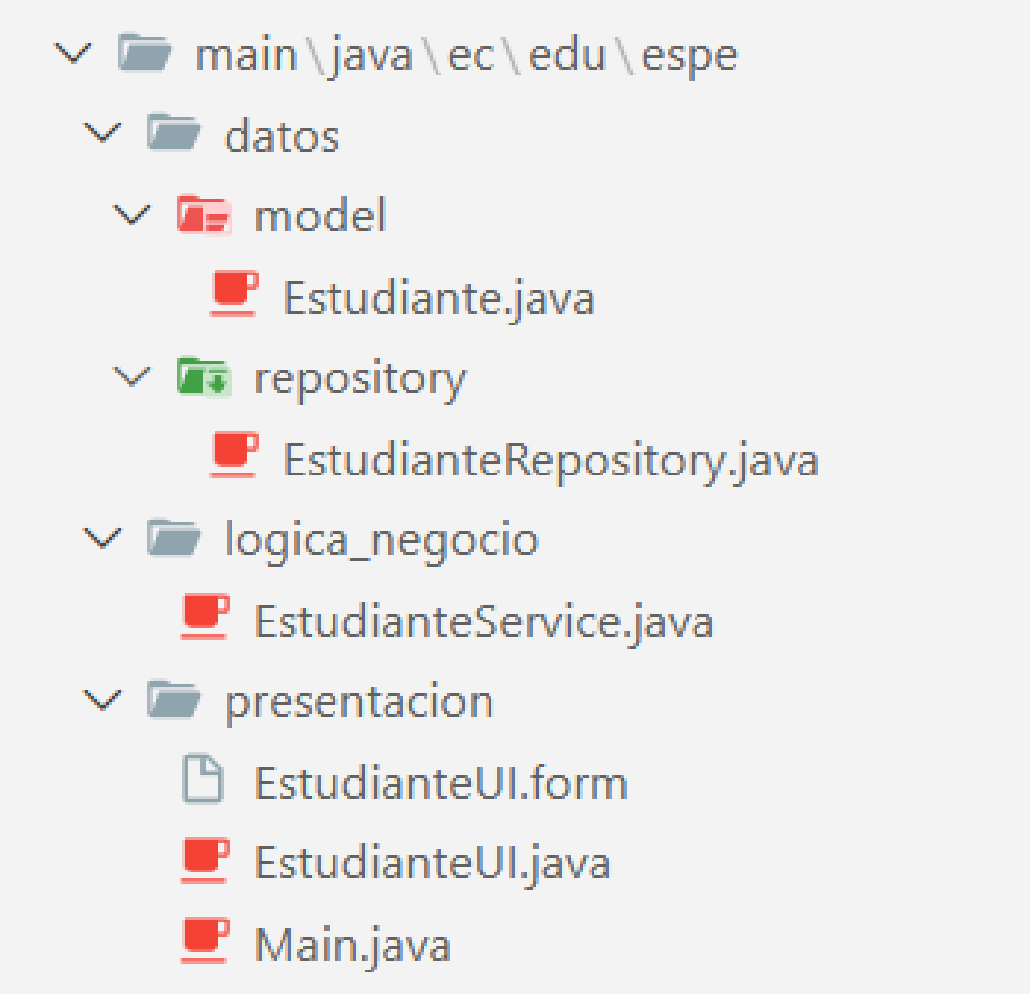


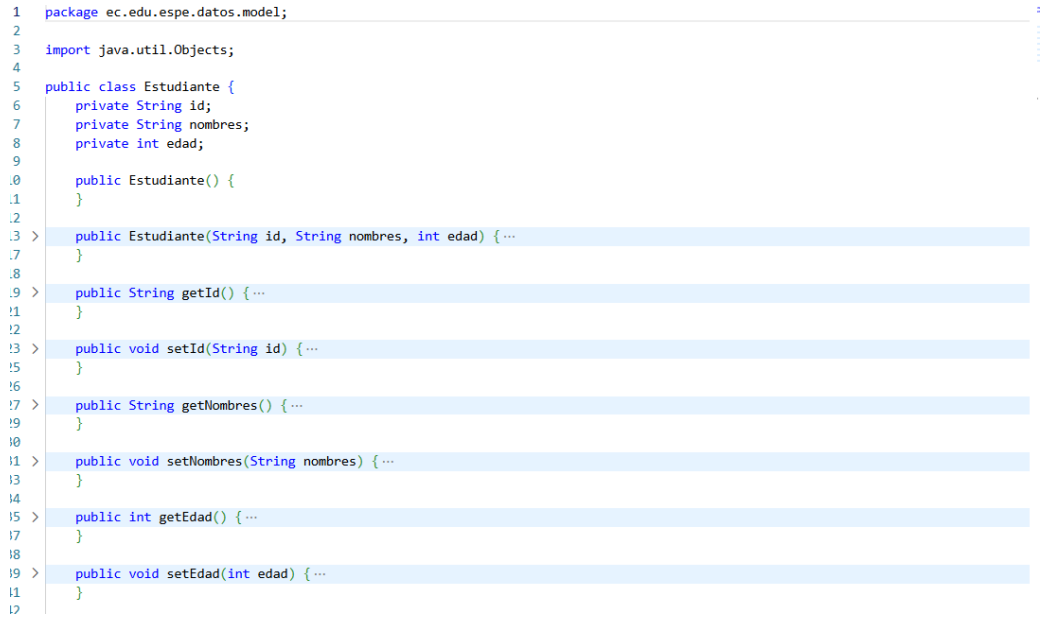
Figura 1: Diagrama de la arquitectura de 3 capas implementada

3. Descripción de las Clases

3.1. Capa Modelo - Clase Estudiante

La clase `Estudiante.java` representa la entidad de dominio del sistema. Esta clase encapsula los atributos principales de un estudiante: identificador único (ID), nombres completos y edad. Implementa el patrón JavaBean con un constructor por defecto, un constructor parametrizado, y métodos getters y setters para cada atributo.

Adicionalmente, la clase sobrescribe los métodos `equals()` y `hashCode()` utilizando el ID como criterio de igualdad, lo que permite comparar objetos de tipo Estudiante de manera efectiva. También implementa el método `toString()` para facilitar la representación textual del objeto durante procesos de depuración y logging.



```java
package ec.edu.espe.datos.model;

import java.util.Objects;

public class Estudiante {
    private String id;
    private String nombres;
    private int edad;

    public Estudiante() {
    }

    public Estudiante(String id, String nombres, int edad) {
    }

    public String getId() {
    }

    public void setId(String id) {
    }

    public String getNombres() {
    }

    public void setNombres(String nombres) {
    }

    public int getEdad() {
    }

    public void setEdad(int edad) {
    }
```

Figura 2: Código de la clase Estudiante (Capa Modelo)

3.2. Capa de Acceso a Datos - Clase EstudianteRepository

La clase `EstudianteRepository.java` constituye la capa de persistencia del sistema. Implementa el patrón Singleton para garantizar una única instancia del repositorio durante toda la ejecución de la aplicación, asegurando la consistencia de los datos en memoria.

Esta clase mantiene una colección interna de tipo `ArrayList` que almacena todos los estudiantes registrados. Proporciona los métodos CRUD fundamentales: `agregar()`, `editar()`, `eliminar()` y `listar()`. Además, incluye métodos auxiliares como `existsById()` y `getById()` que facilitan las validaciones y búsquedas. El método `listar()` retorna una lista inmutable para proteger la integridad de los datos internos.

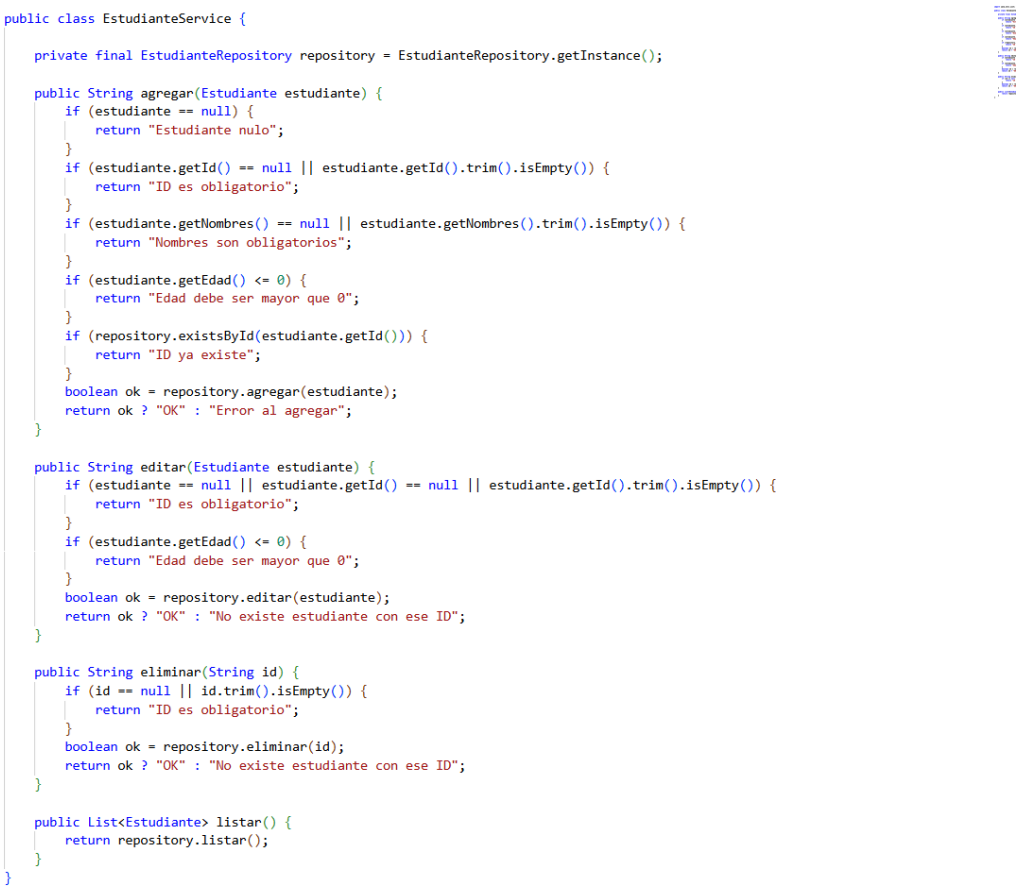
```java
package ec.edu.espe.datos.repository;

import ec.edu.espe.datos.model.Estudiante;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class EstudianteRepository {
    private final List<Estudiante> estudiantes = new ArrayList<>();

    private static final EstudianteRepository INSTANCE = new EstudianteRepository();

    private EstudianteRepository() {
    }

    public static EstudianteRepository getInstance() {
        return INSTANCE;
    }

    public boolean agregar(Estudiante estudiante) {
        if (existsById(estudiante.getId())) {
            return false;
        }
        return estudiantes.add(estudiante);
    }

    public boolean editar(Estudiante estudiante) {
        Estudiante existing = getById(estudiante.getId());
        if (existing == null) {
            return false;
        }
        existing.setNombres(estudiante.getNombres());
        existing.setEdad(estudiante.getEdad());
        return true;
    }

    public boolean eliminar(String id) {
        Estudiante existing = getById(id);
        if (existing == null) {
            return false;
        }
        return estudiantes.remove(existing);
    }

    public List<Estudiante> listar() {
        return Collections.unmodifiableList(new ArrayList<>(estudiantes));
    }
```

Figura 3: Código de la clase EstudianteRepository (Capa de Datos)

3.3. Capa de Lógica de Negocio - Clase EstudianteService

La clase `EstudianteService.java` implementa la lógica de negocio de la aplicación. Actúa como intermediario entre la capa de presentación y el repositorio de datos, aplicando todas las reglas de validación necesarias antes de realizar cualquier operación CRUD.

Esta clase valida que el ID y los nombres no sean nulos o vacíos, que la edad sea un valor positivo mayor a cero, y que no existan IDs duplicados al intentar agregar un nuevo estudiante. Cada método retorna un mensaje de tipo String que indica el resultado de la operación (.ºK.en caso de éxito o un mensaje descriptivo del error encontrado), permitiendo que la capa de presentación informe adecuadamente al usuario sobre el resultado de sus acciones.



```java
public class EstudianteService {

    private final EstudianteRepository repository = EstudianteRepository.getInstance();

    public String agregar(Estudiante estudiante) {
        if (estudiante == null) {
            return "Estudiante nulo";
        }
        if (estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
            return "ID es obligatorio";
        }
        if (estudiante.getNombres() == null || estudiante.getNombres().trim().isEmpty()) {
            return "Nombres son obligatorios";
        }
        if (estudiante.getEdad() <= 0) {
            return "Edad debe ser mayor que 0";
        }
        if (repository.existsById(estudiante.getId())) {
            return "ID ya existe";
        }
        boolean ok = repository.agregar(estudiante);
        return ok ? "OK" : "Error al agregar";
    }

    public String editar(Estudiante estudiante) {
        if (estudiante == null || estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
            return "ID es obligatorio";
        }
        if (estudiante.getEdad() <= 0) {
            return "Edad debe ser mayor que 0";
        }
        boolean ok = repository.editar(estudiante);
        return ok ? "OK" : "No existe estudiante con ese ID";
    }

    public String eliminar(String id) {
        if (id == null || id.trim().isEmpty()) {
            return "ID es obligatorio";
        }
        boolean ok = repository.eliminar(id);
        return ok ? "OK" : "No existe estudiante con ese ID";
    }

    public List<Estudiante> listar() {
        return repository.listar();
    }
}
```

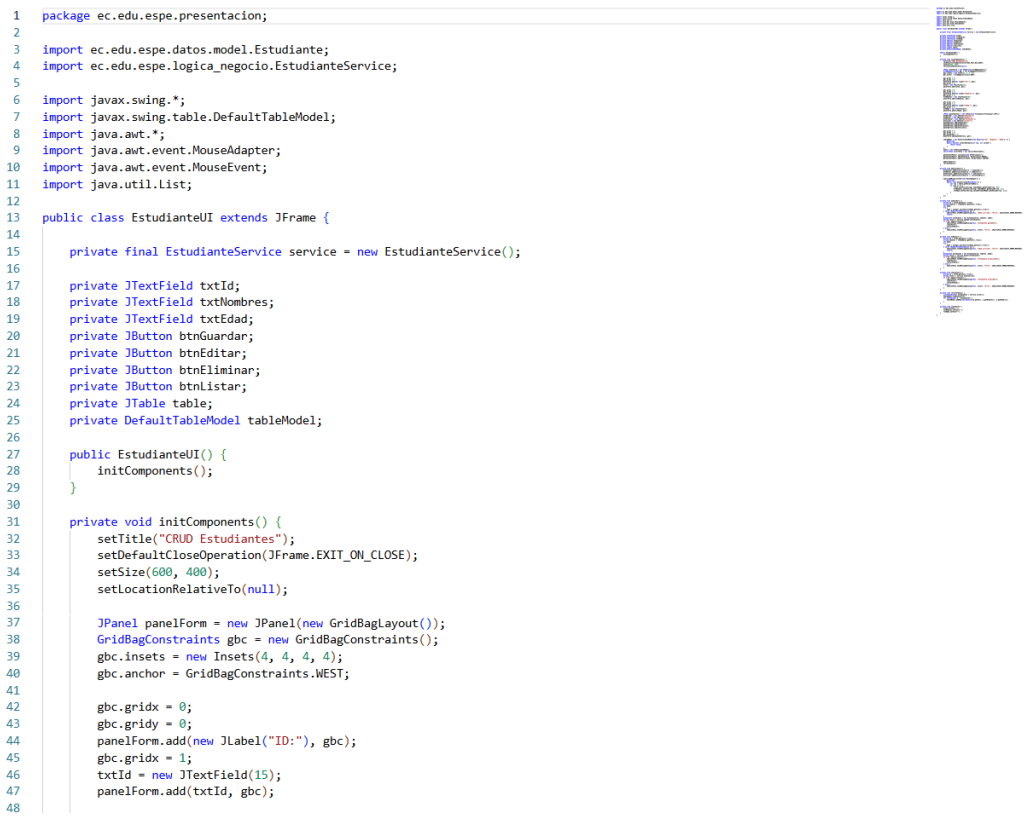
Figura 4: Código de la clase EstudianteService (Capa de Lógica de Negocio)

3.4. Capa de Presentación - Clase EstudianteUI

La clase `EstudianteUI.java` implementa la interfaz gráfica de usuario utilizando Java Swing. Extiende de `JFrame` y construye un formulario completo con campos de texto para capturar el ID, nombres y edad del estudiante, así como botones para ejecutar las operaciones de Guardar, Editar, Eliminar y Listar.

La interfaz incluye una `JTable` que muestra todos los estudiantes registrados en formato tabular. La clase utiliza un `DefaultTableModel` para gestionar los datos de la tabla, configurado como no editable para mantener la integridad de la información. Implementa listeners para los eventos de los botones y para la selección de filas en la tabla, permitiendo que al hacer clic sobre un estudiante, sus datos se carguen automáticamente en el formulario para facilitar operaciones de edición o eliminación.

Esta clase representa tanto la Vista como el Controlador del patrón MVC: maneja la presentación de la interfaz y también gestiona las interacciones del usuario, delegando la lógica de negocio al `EstudianteService`.



```java
package ec.edu.espe.presentacion;

import ec.edu.espe.datos.model.Estudiante;
import ec.edu.espe.logica_negocio.EstudianteService;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.List;

public class EstudianteUI extends JFrame {

    private final EstudianteService service = new EstudianteService();

    private JTextField txtId;
    private JTextField txtNombres;
    private JTextField txtEdad;
    private JButton btnGuardar;
    private JButton btnEditar;
    private JButton btnEliminar;
    private JButton btnListar;
    private JTable table;
    private DefaultTableModel tableModel;

    public EstudianteUI() {
        initComponents();
    }

    private void initComponents() {
        setTitle("CRUD Estudiantes");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(600, 400);
        setLocationRelativeTo(null);

        JPanel panelForm = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(4, 4, 4, 4);
        gbc.anchor = GridBagConstraints.WEST;

        gbc.gridx = 0;
        gbc.gridy = 0;
        panelForm.add(new JLabel("ID:"), gbc);
        gbc.gridx = 1;
        txtId = new JTextField(15);
        panelForm.add(txtId, gbc);

```

Figura 5: Código de la clase EstudianteUI (Capa de Presentación)

4. Ejecución del Programa

La aplicación se inicia ejecutando la clase `Main.java`, que crea una instancia de `EstudianteUI` y la hace visible. Una vez en ejecución, el usuario puede interactuar con la interfaz gráfica para realizar las siguientes operaciones:

- **Guardar**: Ingresar los datos de un nuevo estudiante (ID, nombres y edad) y hacer clic en "Guardar"para agregarlo al sistema.
- **Editar**: Seleccionar un estudiante de la tabla, modificar sus datos en el formulario y presionar ."Editar"para actualizar la información.
- **Eliminar**: Seleccionar un estudiante de la tabla y hacer clic en ."Eliminar"para removerlo del sistema.
- **Listar**: Actualizar la tabla para mostrar todos los estudiantes registrados actualmente en el sistema.

El sistema valida automáticamente los datos ingresados, mostrando mensajes de error cuando se detectan inconsistencias (como edad negativa, ID duplicado o campos vacíos). Al completar exitosamente una operación, se muestra un mensaje de confirmación y la tabla se actualiza automáticamente para reflejar los cambios.

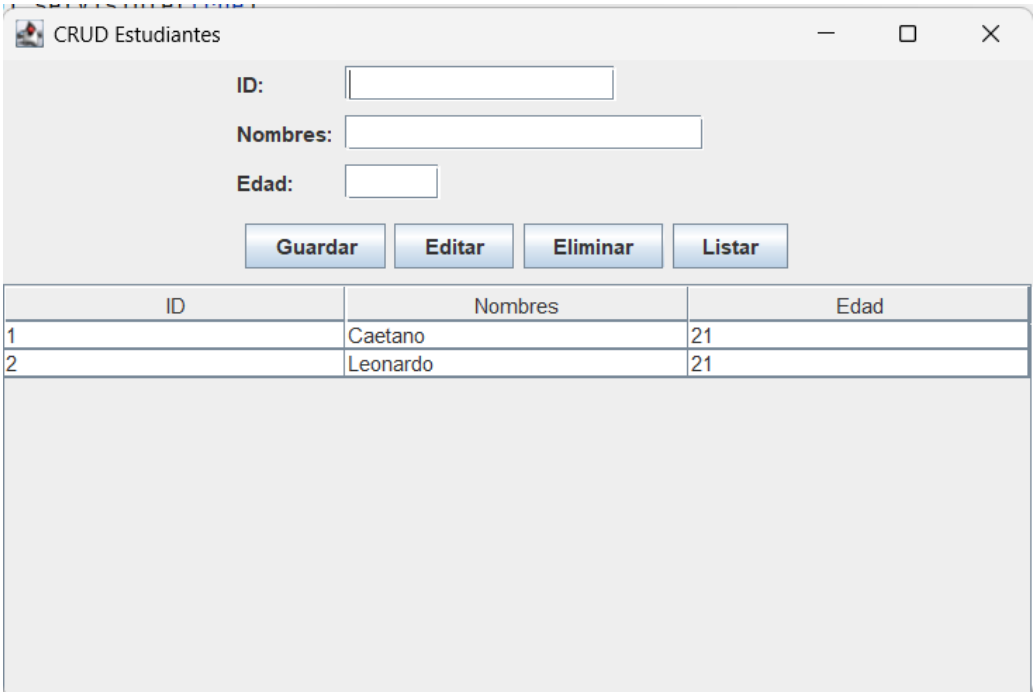


Figura 6: Interfaz gráfica de la aplicación en ejecución

4.1. Validación de ID

El sistema implementa una validación robusta para el identificador único de cada estudiante. Cuando el usuario intenta guardar un nuevo estudiante, la capa de servicio verifica que el ID no sea nulo, no esté vacío y que no exista previamente en el sistema. Si se detecta un ID duplicado, la aplicación rechaza la operación y muestra un mensaje de error al usuario indicando que ÏD ya existe", impidiendo así la creación de registros duplicados y manteniendo la integridad referencial de los datos.

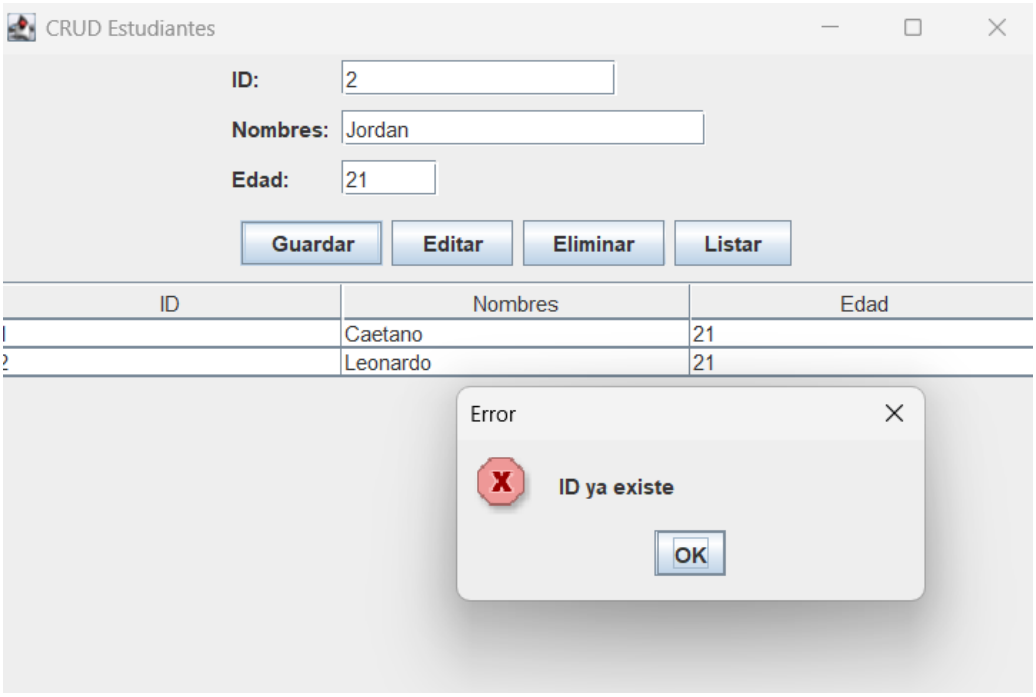


Figura 7: Mensaje de error al intentar registrar un ID duplicado

4.2. Validación de Edad

La validación de edad asegura que solo se ingresen valores numéricos positivos mayores a cero. En primer lugar, la interfaz gráfica valida que el texto ingresado sea convertible a un número entero; si no lo es, muestra un mensaje de .Edad inválida". Posteriormente, la capa de servicio verifica que el valor numérico sea mayor que cero, rechazando edades negativas o cero con el mensaje .Edad debe ser mayor que 0". Esta doble validación garantiza la coherencia de los datos almacenados.

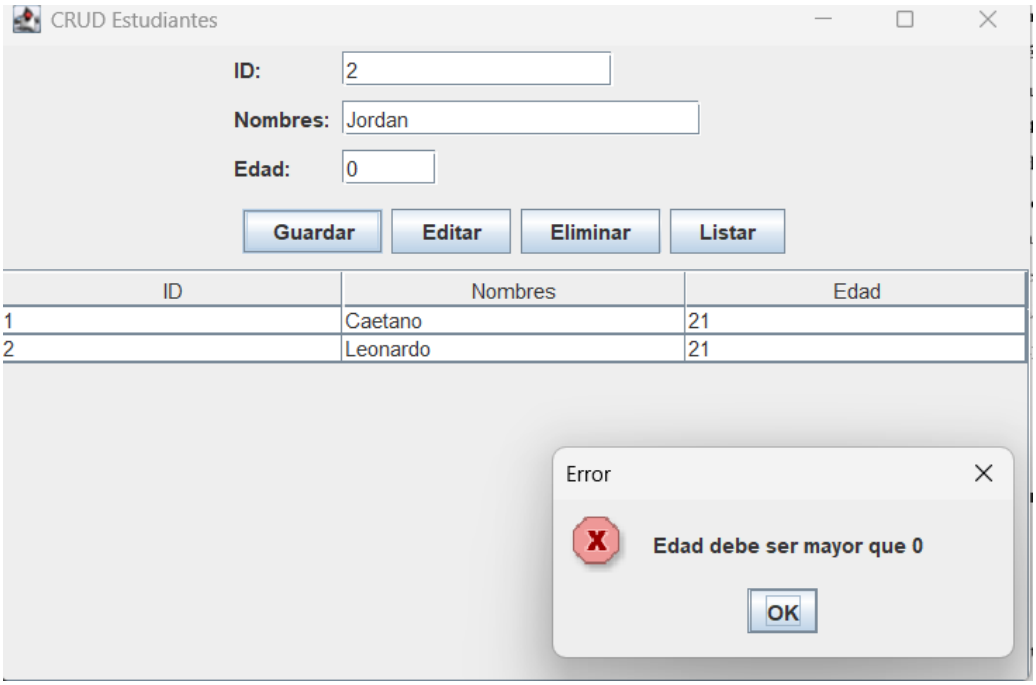


Figura 8: Mensaje de error al intentar ingresar una edad inválida

5. Conclusiones

1. La implementación de la arquitectura de 3 capas permitió una clara separación de responsabilidades, facilitando el mantenimiento y la escalabilidad del código al aislar la lógica de negocio, el acceso a datos y la presentación.

2. El patrón MVC en conjunto con la arquitectura de capas proporcionó una estructura organizada donde la interfaz de usuario no contiene lógica de negocio, cumpliendo con el principio de responsabilidad única y mejorando la testabilidad del sistema.

3. El uso del patrón Singleton en el repositorio garantizó la consistencia de los datos durante toda la ejecución de la aplicación, evitando problemas de sincronización y duplicación de información.

6. Recomendaciones

1. Se recomienda implementar persistencia de datos en archivos o base de datos para evitar la pérdida de información al cerrar la aplicación, reemplazando el almacenamiento en memoria actual.

2. Considerar la implementación de un sistema de logging para registrar las operaciones CRUD realizadas, lo que facilitaría la auditoría y el diagnóstico de problemas en producción.

3. Sería beneficioso agregar más validaciones en la capa de servicio, como la validación de formatos específicos para el ID, rangos de edad permitidos, y restricciones de caracteres en los nombres para mejorar la robustez del sistema.