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

2.1. Arquitectura MVC con 3 Capas

La aplicación está estructurada siguiendo el patrón de arquitectura de 3 capas en conjunto con el patrón Modelo-Vista-Controlador (MVC), que separa las responsabilidades en tres niveles distintos: la capa de presentación (Vista), la capa de lógica de negocio (Controlador/Servicio) y la capa de acceso a datos (Modelo + Repositorio). Esta separación permite un código más mantenible, escalable y facilita las pruebas unitarias de cada componente de forma independiente.

2.1.1. Capa 1: Datos (Modelo + Repositorio)

Esta capa se encuentra en el paquete `ec.edu.espe.datos` y tiene dos componentes principales:

Modelo (`Estudiante.java`): Define la estructura de datos del estudiante con sus atributos (ID, Nombres, Edad). Representa la entidad de negocio con sus constructores, métodos getters y setters, `equals()`, `hashCode()` y `toString()`.

Repositorio (`EstudianteRepository.java`): Gestiona el almacenamiento y recuperación de datos en memoria mediante un `ArrayList`. Implementa el patrón Singleton para garantizar una única instancia y persistencia compartida durante toda la ejecución de la aplicación. Proporciona los métodos CRUD: `agregar()`, `editar()`, `eliminar()`, `listar()`, `getById()` y `existsById()`.

2.1.2. Capa 2: Lógica de Negocio (Servicio)

Esta capa se encuentra en el paquete `ec.edu.espe.logica_negocio` y contiene la clase `EstudianteService.java`, que implementa las reglas de negocio y validaciones. Las validaciones incluyen: datos no nulos, ID obligatorio y único, nombres obligatorios, edad mayor a 0, y genera mensajes de error descriptivos. Esta capa coordina entre la presentación y los datos, aplicando todas las reglas de negocio antes de persistir la información.

2.1.3. Capa 3: Presentación (Vista)

Esta capa se encuentra en el paquete `ec.edu.espe.presentacion` y contiene dos clases principales:

Vista (`EstudianteUI.java`): Implementa la interfaz gráfica utilizando Java Swing, con formulario para captura de datos (ID, Nombres, Edad), botones para operaciones CRUD (Guardar, Editar, Eliminar, Listar), tabla para visualizar estudiantes, y manejo de eventos y mensajes.

Main (`Main.java`): Punto de entrada de la aplicación que inicializa la interfaz gráfica.

El flujo de datos en la aplicación sigue una jerarquía clara: Usuario → Vista (UI) → Servicio (Negocio) → Repositorio (Datos) → Modelo. Esta estructura garantiza que cada capa tenga una responsabilidad única y bien definida, cumpliendo con el principio de separación de responsabilidades.

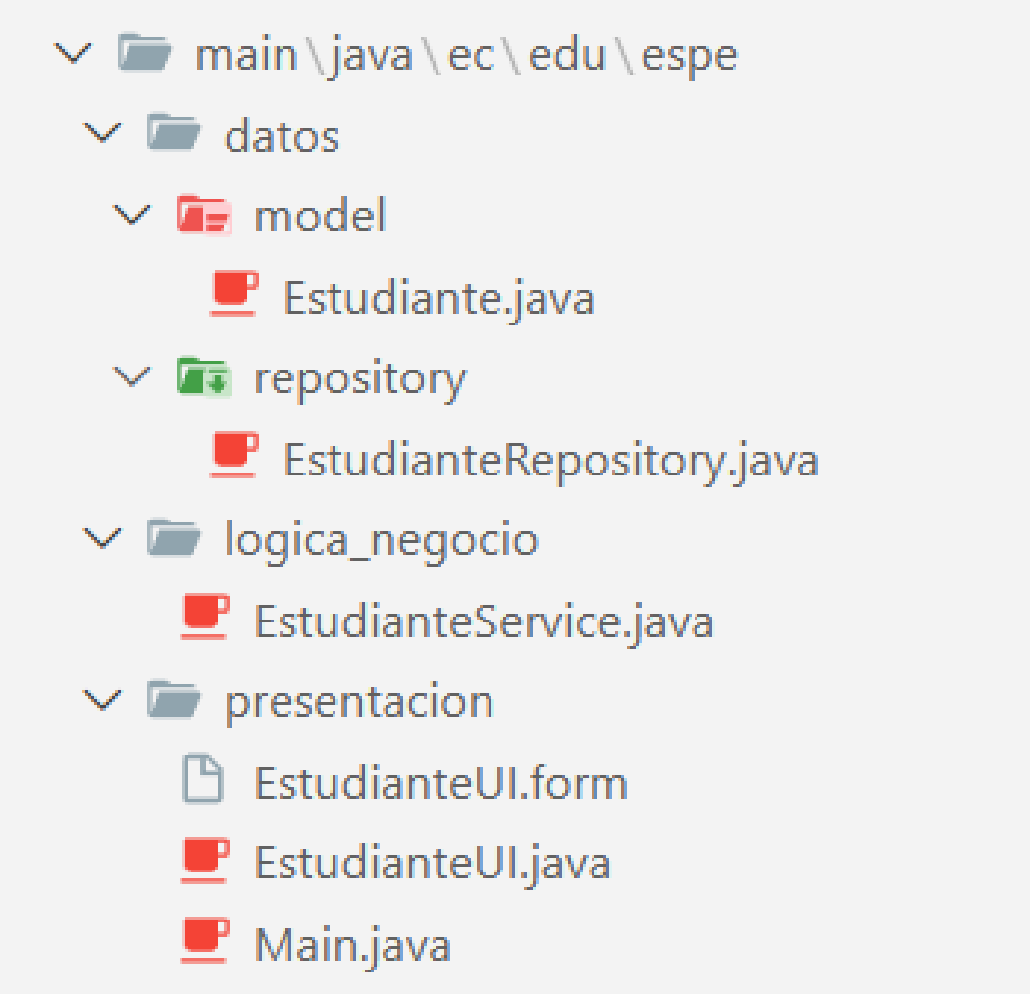


Figura 1: Diagrama de la arquitectura de 3 capas con patrón MVC implementada

3. Descripción de las Clases

3.1. Patrón Singleton en EstudianteRepository

Antes de describir las clases individuales, es importante destacar que el repositorio implementa el patrón de diseño Singleton. Este patrón garantiza que solo exista una instancia del repositorio durante toda la ejecución de la aplicación, lo cual es crucial para mantener la consistencia de los datos en memoria.

La implementación del Singleton se realiza mediante:

- Una instancia estática final: `private static final EstudianteRepository INSTANCE`
- Un constructor privado que previene la creación directa de instancias
- Un método estático `getInstance()` que retorna siempre la misma instancia

Este patrón resuelve el problema de persistencia compartida: sin Singleton, cada vez que se crea un nuevo servicio podría crear un nuevo repositorio, perdiendo los datos anteriores. Con Singleton, todos los servicios comparten la misma instancia y los datos persisten durante toda la ejecución.

3.2. 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)

3.3. 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.

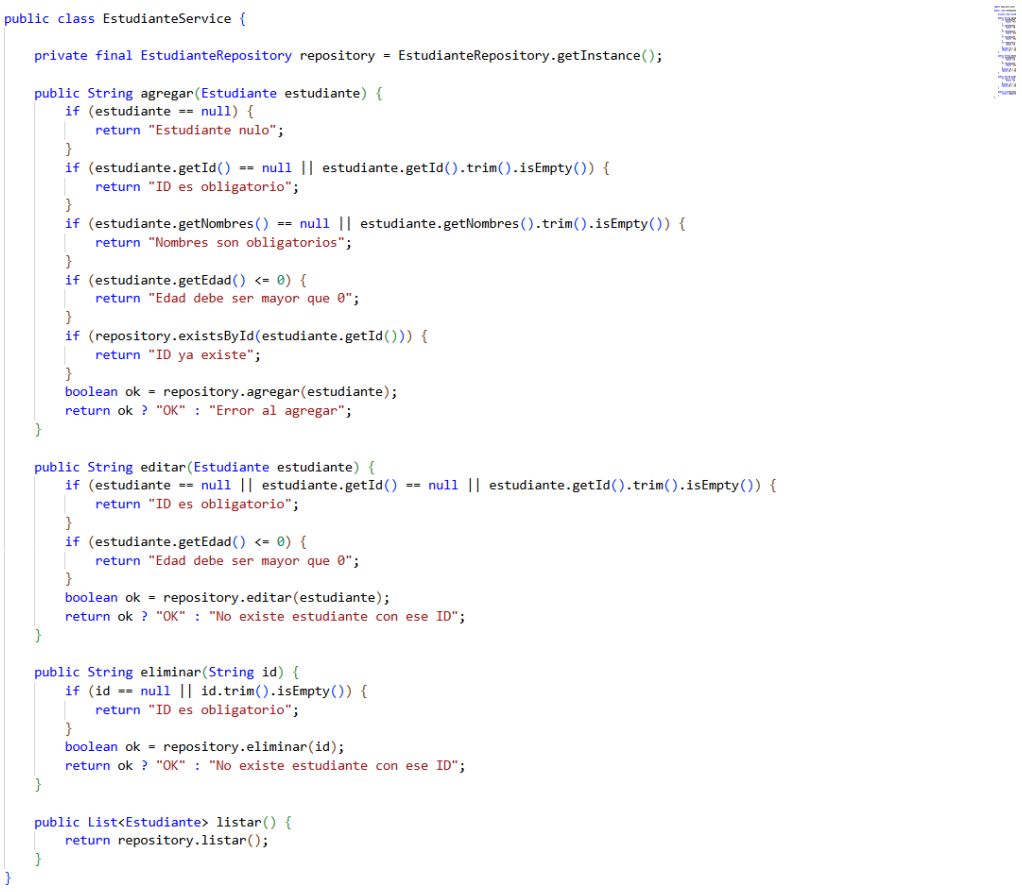
```java
package ec.edu.espe.datos.repository;

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)

3.4. 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();
    }
}
```

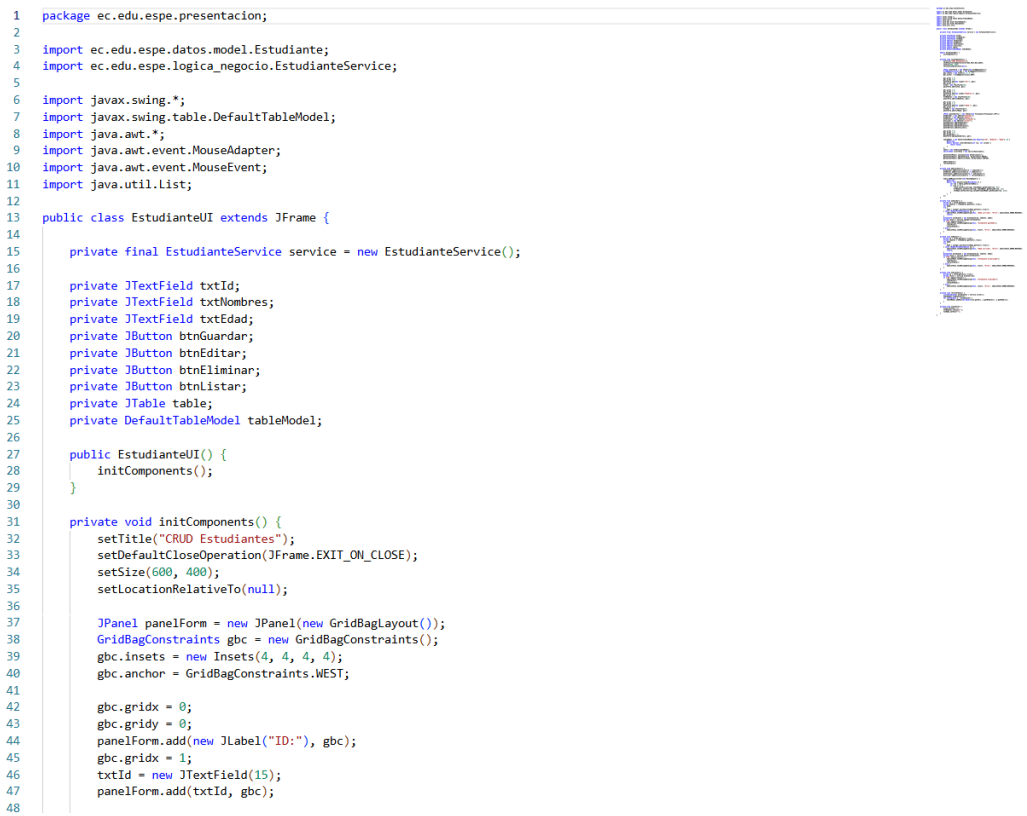
Figura 4: Código de la clase EstudianteService (Capa de Lógica de Negocio)

3.5. 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` en el Event Dispatch Thread de Swing 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 o cero, ID duplicado, campos vacíos o edad con formato inválido). Al completar exitosamente una operación, se muestra un mensaje de confirmación y la tabla se actualiza automáticamente para reflejar los cambios.

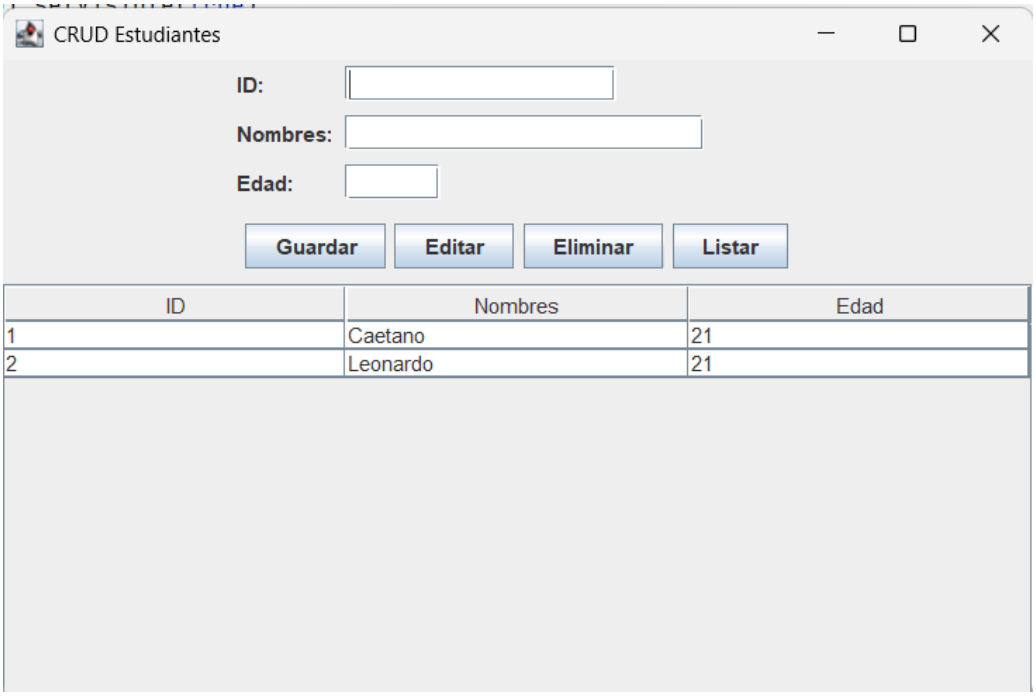


Figura 6: Interfaz gráfica de la aplicación en ejecución

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.

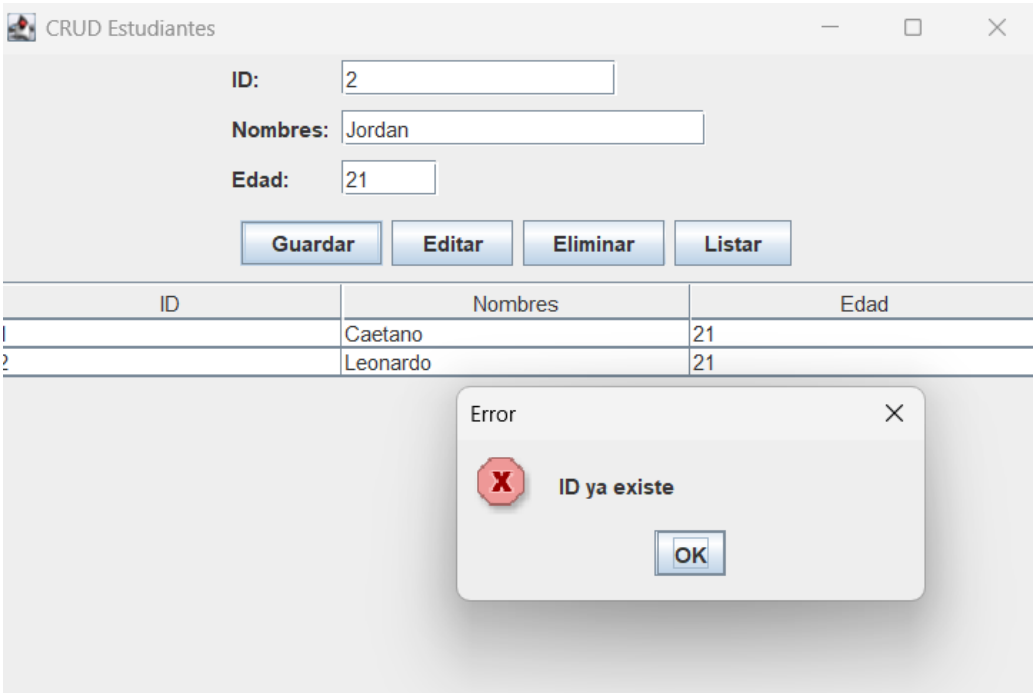


Figura 7: Mensaje de error al intentar registrar un ID duplicado

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.

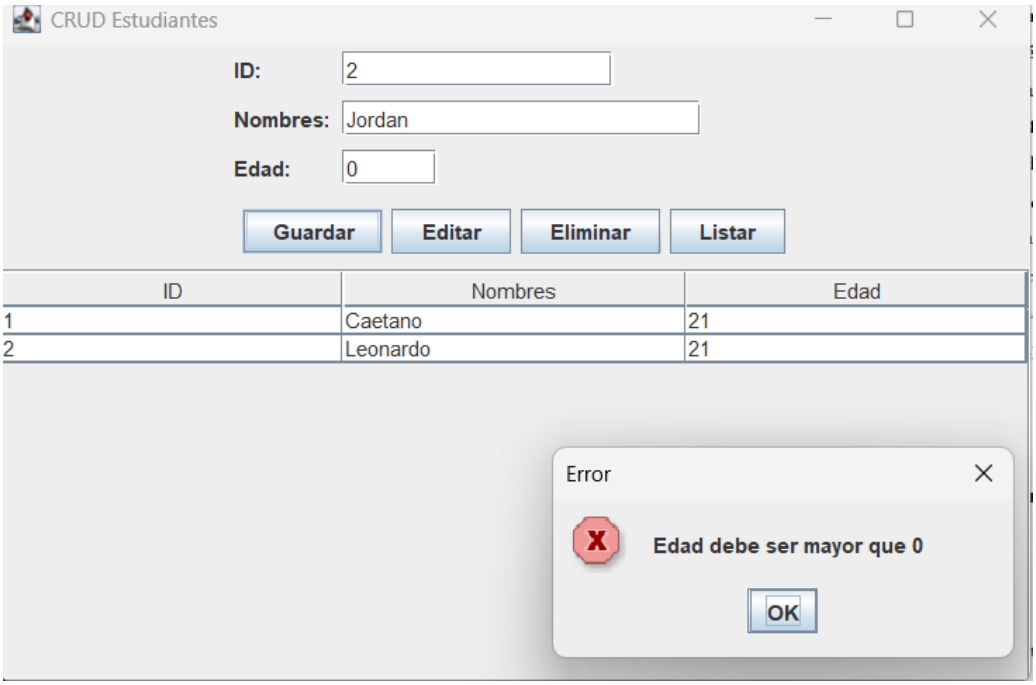


Figura 8: Mensaje de error al intentar ingresar una edad inválida

5. Análisis Comparativo de Patrones

5.1. Patrón MVC vs Patrón Singleton

El proyecto implementa dos patrones fundamentales que trabajan de manera complementaria. El patrón MVC (Modelo-Vista-Controlador) es un patrón arquitectónico que organiza toda la estructura del sistema, mientras que el patrón Singleton es un patrón de diseño creacional que controla cómo se crean e instancian ciertos objetos específicos.

5.1.1. Problemas que Resuelve Cada Patrón

MVC: Resuelve el problema de separación de responsabilidades entre la presentación, la lógica de negocio y los datos. Evita el código espagueti donde todo está mezclado y facilita enormemente el mantenimiento y la escalabilidad del sistema.

Singleton: Garantiza que una clase tenga una única instancia en toda la aplicación. En nuestro caso, evita la creación de múltiples repositorios con listas diferentes de estudiantes, lo que causaría pérdida de datos y comportamiento inconsistente.

5.1.2. Capas de Utilización

MVC: Se aplica en toda la arquitectura, organizando las tres capas completas del sistema: Modelo (entidad Estudiante), Vista (EstudianteUI) y Controlador (EstudianteService que coordina entre ambos).

Singleton: Se utiliza principalmente en la capa de Datos, específicamente en `EstudianteRepository` aunque podría aplicarse en otras capas según las necesidades (por ejemplo, en servicios de configuración o logging).

5.1.3. Influencia en el Mantenimiento

Ambos patrones influyen positivamente en el mantenimiento del código. **MVC** permite modificar la interfaz de usuario sin tocar la lógica de negocio, facilita agregar nuevas funcionalidades de forma modular y permite que múltiples desarrolladores trabajen en paralelo en diferentes capas sin interferencias. **Singleton** simplifica el acceso a recursos compartidos al tener un único punto de acceso, reduce bugs relacionados con instancias duplicadas y centraliza la gestión de datos, facilitando el debugging y el mantenimiento.

5.1.4. Prevención de Fallas de Diseño

MVC previene el acoplamiento fuerte entre las capas del sistema, evita dependencias circulares que complican el código, facilita el testing unitario al permitir probar cada capa de forma independiente y promueve código reutilizable y modular.

Singleton previene problemas de sincronización de datos al garantizar una única fuente de verdad, evita inconsistencias causadas por múltiples instancias con datos diferentes y garantiza que el estado sea único y compartido correctamente entre todos los componentes que lo necesitan.

5.2. Complementariedad de los Patrones

Los patrones MVC y Singleton no son excluyentes, sino complementarios. En el proyecto, MVC define la estructura general del sistema organizando las responsabilidades en tres capas claramente separadas, mientras que Singleton define cómo se crea e instancia específicamente el repositorio dentro de esa estructura arquitectónica. Así, el servicio (capa de negocio del MVC) utiliza `EstudianteRepository.getInstance()` para acceder siempre a la misma instancia del repositorio, garantizando persistencia de datos.

5.3. Reflexión sobre Ventajas, Limitaciones y Riesgos

5.3.1. Ventajas del MVC

- Código organizado, predecible y fácil de entender para nuevos desarrolladores
- Permite cambiar tecnologías de presentación (de Swing a Web o Mobile) sin afectar la lógica
- Testing más sencillo al poder probar cada capa de forma independiente
- Facilita el trabajo colaborativo en equipos

5.3.2. Limitaciones del MVC

- Overhead inicial en proyectos muy pequeños o triviales
- Curva de aprendizaje para desarrolladores principiantes
- Puede resultar excesivo para aplicaciones de una sola funcionalidad simple

5.3.3. Riesgos del MVC

- Si las capas no están bien definidas, se puede romper la separación de responsabilidades
- Los controladores pueden crecer demasiado y convertirse en "God Objects"

5.3.4. Ventajas del Singleton

- Acceso global simplificado a la instancia desde cualquier parte del código
- Ahorro de memoria al tener una sola instancia en lugar de múltiples objetos duplicados
- Persistencia garantizada durante todo el ciclo de vida de la aplicación
- Implementación técnica simple y directa

5.3.5. Limitaciones del Singleton

- Dificulta el testing unitario debido al estado global compartido
- Puede ocultar dependencias entre clases
- La implementación básica no es thread-safe en ambientes concurrentes
- Complica el uso de técnicas modernas como inyección de dependencias

5.3.6. Riesgos del Singleton

- El sobreuso puede crear acoplamiento fuerte entre componentes
- Dificulta el escalado a sistemas distribuidos o multi-instancia
- Puede convertirse en una variable global disfrazada
- En aplicaciones multi-hilo puede causar condiciones de carrera si no se maneja correctamente

5.4. Recomendaciones de Uso de Patrones

Para el patrón MVC, se recomienda su uso en aplicaciones de tamaño mediano a grande, cuando se prevén cambios en la interfaz de usuario, en proyectos con múltiples desarrolladores trabajando simultáneamente, y cuando se requiere testing exhaustivo de cada componente.

Para el patrón Singleton, se recomienda su uso para recursos compartidos como conexiones a bases de datos, caches o configuraciones globales, cuando se necesita un punto de acceso centralizado, y para evitar instancias duplicadas de clases que consumen muchos recursos. Sin embargo, debe usarse con precaución en aplicaciones multi-hilo y no debe abusarse de él para evitar crear puntos de acoplamiento global.

En el contexto de este CRUD de Estudiantes, la combinación de MVC + Singleton es ideal porque MVC organiza el código en capas lógicas bien definidas, mientras que Singleton garantiza que todos los componentes utilicen el mismo repositorio, asegurando que los datos persistan de manera consistente durante toda la ejecución de la aplicación.

6. Tecnologías y Herramientas Utilizadas

El proyecto ha sido desarrollado utilizando las siguientes tecnologías y herramientas:

6.1. Lenguaje de Programación

Java: Lenguaje orientado a objetos que permite la implementación de patrones de diseño y arquitecturas robustas. La aplicación está desarrollada en Java 8 o superior.

6.2. Framework de Interfaz Gráfica

Java Swing: Framework estándar de Java para desarrollo de interfaces gráficas de usuario (GUI). Se utilizaron componentes como `JFrame`, `JTable`, `JTextField`, `JButton` y `JOptionPane` para construir la interfaz de usuario completa.

6.3. Gestión de Dependencias y Compilación

Apache Maven: Herramienta de gestión y construcción de proyectos Java. El archivo `pom.xml` define la estructura del proyecto, dependencias y configuración de compilación.

6.4. Estructura de Paquetes

La organización del código sigue la convención de nombres de paquetes Java:

- `ec.edu.espe.datos.model` - Entidades del dominio
- `ec.edu.espe.datos.repository` - Capa de acceso a datos
- `ec.edu.espe.logica_negocio` - Servicios y lógica de negocio
- `ec.edu.espe.presentacion` - Interfaces de usuario y punto de entrada

6.5. Almacenamiento de Datos

Colecciones en Memoria: Se utiliza `ArrayList<Estudiante>` para almacenar los datos durante la ejecución. Los datos no persisten entre sesiones, permaneciendo en memoria solo mientras la aplicación está en ejecución.

6.6. Patrones de Diseño Implementados

- **MVC (Modelo-Vista-Controlador):** Patrón arquitectónico para organizar el código
- **Singleton:** Patrón creacional para garantizar instancia única del repositorio
- **JavaBean:** Patrón para la clase Estudiante con getters/setters

6.7. Ejecución del Proyecto

La aplicación puede ejecutarse mediante:

- **Maven:** Comando `mvn exec:java -Dexec.mainClass=.ec.edu.espe.presentacion.Main"`
- **IDE:** Ejecutando directamente la clase `Main.java` desde cualquier entorno de desarrollo integrado (IntelliJ IDEA, Eclipse, NetBeans)

7. Casos de Prueba y Validaciones

El sistema implementa múltiples casos de prueba para verificar el correcto funcionamiento de todas las operaciones CRUD:

7.1. Pruebas de CREATE (Crear)

- Creación exitosa de estudiante con datos válidos
- Validación de ID duplicado - debe rechazar con mensaje de error
- Validación de edad inválida (cero o negativa) - debe rechazar
- Validación de nombres vacíos - debe rechazar
- Validación de formato de edad (debe ser numérico)

7.2. Pruebas de READ (Leer)

- Listar todos los estudiantes en la tabla
- Visualización correcta de columnas (ID, Nombres, Edad)
- Selección de estudiantes desde la tabla
- Carga automática de datos en el formulario al seleccionar

7.3. Pruebas de UPDATE (Actualizar)

- Actualización exitosa de estudiante existente
- Validación de ID no existente - debe rechazar con mensaje
- Preservación del ID durante la edición
- Actualización inmediata en la tabla

7.4. Pruebas de DELETE (Eliminar)

- Eliminación exitosa de estudiante existente
- Validación de ID no existente - debe rechazar con mensaje
- Remoción inmediata de la tabla
- Confirmación mediante mensajes al usuario

7.5. Pruebas de Persistencia con Singleton

- Verificación de que los datos permanecen entre operaciones
- Comprobación de que múltiples accesos usan la misma instancia
- Validación de consistencia de datos durante toda la sesión

8. Conclusiones

1. La implementación de la arquitectura de 3 capas en conjunto con el patrón MVC permitió una clara separación de responsabilidades, facilitando el mantenimiento y la escalabilidad del código al aislar la lógica de negocio (EstudianteService), el acceso a datos (EstudianteRepository) y la presentación (EstudianteUI).

2. El patrón MVC proporcionó una estructura organizada donde la interfaz de usuario no contiene lógica de negocio, cumpliendo con el principio de responsabilidad única. Esto mejora significativamente la testabilidad del sistema al permitir probar cada capa de forma independiente y facilita cambios futuros en la tecnología de presentación sin afectar la lógica central.

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. La implementación mediante constructor privado y método `getInstance()` asegura que todos los servicios compartan la misma instancia del repositorio.

4. La validación multicapa (tanto en la interfaz como en la capa de servicio) proporciona una robustez superior al sistema, asegurando la integridad de los datos antes de ser persistidos. Las validaciones incluyen verificación de ID único, nombres obligatorios y edad mayor a cero, con mensajes descriptivos para el usuario.

5. La combinación complementaria de MVC (patrón arquitectónico global) y Singleton (patrón de diseño creacional específico) demuestra cómo diferentes tipos de patrones pueden trabajar juntos para resolver problemas en distintos niveles de abstracción del sistema.

6. El análisis comparativo de los patrones MVC y Singleton revela que, aunque tienen propósitos diferentes, ambos contribuyen a la calidad del software: MVC organiza la estructura general promoviendo la modularidad, mientras que Singleton garantiza la unicidad y consistencia de recursos críticos como el repositorio de datos.

9. Recomendaciones

1. Se recomienda implementar persistencia de datos en archivos JSON, XML o base de datos relacional (como SQLite o MySQL) para evitar la pérdida de información al cerrar la aplicación, reemplazando el almacenamiento en memoria actual del `ArrayList`.

2. Considerar la implementación de un sistema de logging utilizando frameworks como Log4j o SLF4J para registrar las operaciones CRUD realizadas, incluyendo timestamps, usuario y detalles de la operación. Esto 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 (por ejemplo, patrón EST[0-9]3), rangos de edad permitidos (por ejemplo, entre 15 y 100 años), y restricciones de caracteres en los nombres para evitar caracteres especiales o números.

4. Para mejorar la arquitectura, se podría separar el Controlador de la Vista creando una clase `EstudianteController` independiente que maneje la coordinación entre `EstudianteUI` y `EstudianteService`, haciendo el patrón MVC más explícito y puro.

5. Implementar testing unitario utilizando JUnit para cada capa del sistema: tests para las validaciones del `EstudianteService`, tests para las operaciones CRUD del `EstudianteRepository`, y tests para verificar el comportamiento del patrón Singleton.

6. Para aplicaciones concurrentes o multi-hilo, considerar hacer thread-safe la implementación del Singleton utilizando sincronización o el patrón "Bill Pugh Singletonçon clase interna estática, evitando así posibles condiciones de carrera.

7. Agregar manejo de excepciones personalizado creando clases como `EstudianteNotFoundExceptio`n o `DuplicateIdException` en lugar de retornar mensajes de tipo String, lo que facilitaría el manejo de errores y haría el código más robusto y profesional.

10. Referencias y Recursos Adicionales

10.1. Repositorio del Código Fuente

El código completo del proyecto está disponible en el repositorio de GitHub:
`https://github.com/FloresCaetano/27837_G6_ADS/tree/main/U2/Talleres/TALLER1/CRUD`

10.2. Documentación Complementaria

En el directorio `DOCUMENTACION/` del proyecto se encuentran los siguientes archivos de referencia:

- `Explicacion_Arquitectura.md` - Descripción detallada de las 3 capas y patrones
- `Diagrama_Arquitectura.txt` - Diagramas visuales de arquitectura y flujos
- `Cuadro_Comparativo_MVC_vs_Singleton.md` - Análisis crítico comparativo
- `Guia_Pruebas.md` - Casos de prueba detallados del CRUD
- `RESUMEN_ENTREGA.md` - Resumen completo de lo entregado
- `INDICE_GENERAL.md` - Índice de navegación del proyecto

10.3. Bibliografía y Patrones de Diseño

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Oracle Corporation. (2024). *Java Platform, Standard Edition Documentation*.
- Reenskaug, T. (1979). *The Model-View-Controller (MVC) - Its Past and Present*.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

10.4. Información del Proyecto

- **Institución:** Escuela Politécnica del Ejército (ESPE)
- **Asignatura:** Arquitectura de Software
- **Talleres:** U2T1, U2T2, U2T3
- **Grupo:** G6
- **Integrantes:** Caetano Flores, Jordan Guaman, Anthony Morales, Leonardo Narvaez
- **Fecha:** Noviembre 2025