**UNIVERSIDAD DE LAS FUERZAS ARMADAS**
**ESPE**


**NOMBRES:** MIDEROS SAMIR, MIRANDA ALISON, MORÁN DAVID, VIVANCO GABRIEL

**NRC: 27837**

**FECHA:** 19/11/2025

Tema:

“Taller CRUD estudiantes”

**Análisis y diseño del software**

**PERIODO 2025-2026**

# Taller: Creación de un CRUD (ID, Nombres, Edad) aplicando Arquitectura de 3 Capas y Patrón Modelo–Vista–Controlador (MVC)

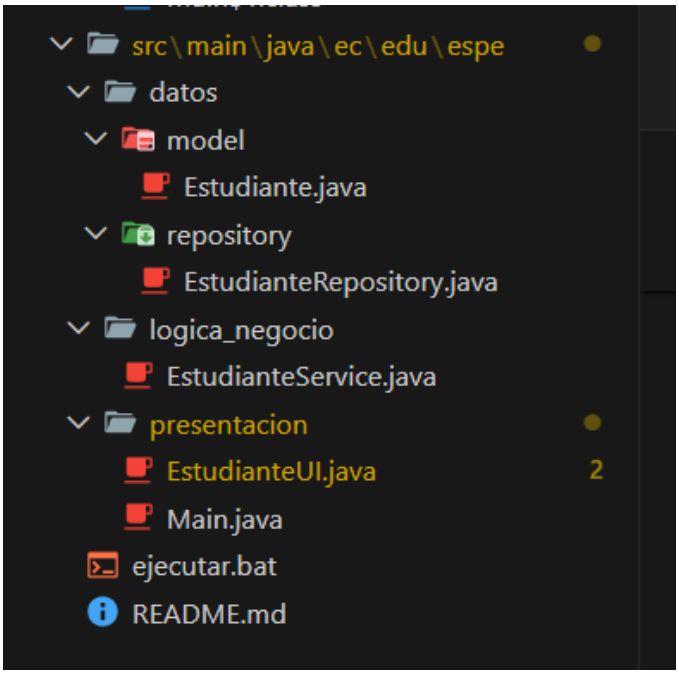
## 1. Objetivo del Taller

Construir una aplicación CRUD de Estudiante (ID, nombres, edad) utilizando la arquitectura de 3 capas y el patrón de diseño Modelo–Vista–Controlador (MVC) con una interfaz gráfica en Java Swing.

## 2. Estructura del Proyecto

La aplicación deberá organizarse en la siguiente estructura de paquetes:

```
src/main/java/ec/edu/espe/
├── datos/
│   ├── model/
│   │   └── Estudiante.java
│   └── repository/
│       └── EstudianteRepository.java
├── logica_negocio/
│   └── EstudianteService.java
└── presentacion/
    ├── EstudianteUI.java
    └── Main.java
```



## 3. Descripción de las Capas

### 3.1 Capa Modelo (Model)

Contiene la clase Estudiante.java, que representa el dominio con sus atributos: ID, nombres y edad. No contiene lógica de negocio ni acceso a datos.

```java
 * Clase modelo que representa un estudiante
 * Contiene los atributos básicos: ID, nombres y edad
 */
public class Estudiante {
    private int id;
    private String nombres;
    private int edad;

    /**
     * Constructor por defecto
     */
    public Estudiante() {
    }

    /**
     * Constructor con parámetros
     * @param id Identificador único del estudiante
     * @param nombres Nombres completos del estudiante
     * @param edad Edad del estudiante
     */
    public Estudiante(int id, String nombres, int edad) {
        this.id = id;
        this.nombres = nombres;
        this.edad = edad;
    }
```

```java
    // Getters
    public int getId() {
        return id;
    }

    public String getNombres() {
        return nombres;
    }

    public int getEdad() {
        return edad;
    }
```

**3.2 Capa de Acceso a Datos (Repository)**

EstudianteRepository.java administra las operaciones CRUD utilizando una colección interna (ArrayList) o almacenamiento simple en archivo. Opcionalmente puede implementarse como Singleton.

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

import ec.edu.espe.datos.model.Estudiante;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

/**
 * Repositorio para gestionar las operaciones CRUD de estudiantes
 * Utiliza ArrayList para almacenamiento interno
 * Implementa patrón Singleton para una única instancia
 */
public class EstudianteRepository {

    // Instancia única del repositorio (Singleton)
    private static EstudianteRepository instance;

    // Lista interna para almacenar estudiantes
    private List<Estudiante> estudiantes;

    /**
     * Constructor privado para implementar Singleton
     */
    private EstudianteRepository() {
        this.estudiantes = new ArrayList<>();
        inicializarDatosPrueba(); // Datos iniciales para pruebas
    }

    /**
     * Método para obtener la única instancia del repositorio
     */
```

```
    public static EstudianteRepository getInstance() {
        if (instance == null) {
            instance = new EstudianteRepository();
        }
        return instance;
    }

    /**
     * Agregar un nuevo estudiante al repositorio
     * @param estudiante El estudiante a agregar
     * @return true si se agregó exitosamente, false si ya existe el
ID
     */
    public boolean agregar(Estudiante estudiante) {
        // Verificar que no exista un estudiante con el mismo ID
        if (buscarPorId(estudiante.getId()).isPresent()) {
            return false; // Ya existe un estudiante con ese ID
        }
        return estudiantes.add(estudiante);
    }
```

**3.3 Capa Lógica de Negocio (Service)**

EstudianteService.java aplica reglas de negocio como validaciones (ej., edad > 0, ID no repetido) y delega las operaciones CRUD al repositorio.

```
package ec.edu.espe.logica_negocio;

import ec.edu.espe.datos.model.Estudiante;
import ec.edu.espe.datos.repository.EstudianteRepository;
import java.util.List;
import java.util.Optional;

/**
 * Servicio de lógica de negocio para estudiantes
 * Contiene las validaciones y reglas de negocio
 * Delega las operaciones CRUD al repositorio
 */
```

```java
public class EstudianteService {

    private EstudianteRepository estudianteRepository;

    /**
     * Constructor que inicializa el repositorio
     */
    public EstudianteService() {
        this.estudianteRepository =
EstudianteRepository.getInstance();
    }

    /**
     * Agregar un nuevo estudiante con validaciones de negocio
     * @param estudiante El estudiante a agregar
     * @return Resultado de la operación con mensaje
     */
    public ResultadoOperacion agregarEstudiante(Estudiante
estudiante) {
        // Validar datos del estudiante
        ResultadoOperacion validacion =
validarEstudiante(estudiante);
        if (!validacion.isExito()) {
            return validacion;
        }

        // Verificar que el ID no esté duplicado
        if (estudianteRepository.existeId(estudiante.getId())) {
            return new ResultadoOperacion(false, "Error: Ya existe
un estudiante con el ID " + estudiante.getId());
        }

        // Intentar agregar al repositorio
        boolean agregado = estudianteRepository.agregar(estudiante);
        if (agregado) {
```

```
            return new ResultadoOperacion(true, "Estudiante agregado
exitosamente");
        } else {
            return new ResultadoOperacion(false, "Error: No se pudo
agregar el estudiante");
        }
    }
```

### 3.4 Capa Presentación – Swing (Vista y Controlador)

EstudianteUI.java representa la interfaz gráfica del usuario. Incluye:

- Campos de texto para ID, nombres y edad.

```
public class EstudianteUI extends JFrame {

    // Componentes de la interfaz
    private JTextField txtId;           ← CAMPOS DE TEXTO
    private JTextField txtNombres;
    private JTextField txtEdad;
    private JButton btnGuardar;         ← BOTONES CRUD
    private JButton btnEditar;
    private JButton btnEliminar;
    private JButton btnLimpiar;
    private JButton btnNuevo;
    private JTable tblEstudiantes;      ← JTABLE COMO GRID
    private DefaultTableModel modeloTabla;

    // Servicio de lógica de negocio
    private EstudianteService estudianteService;  ← ENVÍA SOLICITUDES AL SERVICE
```

- Botones para las acciones CRUD

```
// ID
panel.add(new JLabel("ID:"), gbc);
txtId = new JTextField(10);             ← CAMPO ID

// Nombres
panel.add(new JLabel("Nombres:"), gbc);
txtNombres = new JTextField(20);        ← CAMPO NOMBRES

// Edad
panel.add(new JLabel("Edad:"), gbc);
txtEdad = new JTextField(10);           ← CAMPO EDAD

// Panel de botones
btnNuevo = new JButton("Nuevo");        ← BOTONES CRUD
btnGuardar = new JButton("Guardar");
btnEditar = new JButton("Editar");
btnEliminar = new JButton("Eliminar");
```

- Una JTable que actúa como 'grid' para mostrar los estudiantes.

```
// Crear modelo de tabla
String[] columnas = {"ID", "Nombres", "Edad"};     ← GRID DE ESTUDIANTES
modeloTabla = new DefaultTableModel(columnas, 0);

// Crear tabla
tblEstudiantes = new JTable(modeloTabla);          ← JTABLE COMO GRID
```

- La UI envía las solicitudes al Service y actualiza la vista según resultados.

```
// Llamar al servicio según el modo                ← ENVÍA AL SERVICE
ResultadoOperacion resultado;
if (modoEdicion) {
    resultado = estudianteService.editarEstudiante(estudiante);
} else {
    resultado = estudianteService.agregarEstudiante(estudiante);
}

// Mostrar resultado                               ← ACTUALIZA VISTA
if (resultado.isExito()) {
    cargarDatos();                                 ← ACTUALIZA SEGÚN RESULTADOS
    limpiarFormulario();
}
```

**4. Flujo de Interacción MVC + 3 Capas**

- El usuario interactúa con EstudianteUI.

```
// Evento del botón Guardar
btnGuardar.addActionListener(e -> guardarEstudiante());  ← 1. USUARIO INTERACTÚA

// Evento del botón Editar
btnEditar.addActionListener(e -> {
    if (tblEstudiantes.getSelectedRow() != -1) {
        modoEdicion = true;
        establecerEstadoEdicion();
    }
});
```

- La UI llama a EstudianteService.

```
// Crear objeto estudiante
Estudiante estudiante = new Estudiante(id, nombres, edad);

// Llamar al servicio según el modo                    ← 2. UI LLAMA AL SERVICE
ResultadoOperacion resultado;
if (modoEdicion) {
    resultado = estudianteService.editarEstudiante(estudiante);
} else {
    resultado = estudianteService.agregarEstudiante(estudiante);
}
```

- El Service valida datos y llama a EstudianteRepository.

```
public ResultadoOperacion agregarEstudiante(Estudiante estudiante) {
    // Validar datos del estudiante                     ← 3. SERVICE VALIDA DATOS
    ResultadoOperacion validacion = validarEstudiante(estudiante);
    if (!validacion.isExito()) {
        return validacion;
    }

    // Verificar que el ID no esté duplicado
    if (estudianteRepository.existeId(estudiante.getId())) {
        return new ResultadoOperacion(false, "Error: Ya existe...");
    }

    // Intentar agregar al repositorio                  ← 3. SERVICE LLAMA AL REP
    boolean agregado = estudianteRepository.agregar(estudiante);
}
```

- El Repository ejecuta las operaciones CRUD y devuelve resultados.

```
public boolean agregar(Estudiante estudiante) {          ← 4. REPOSITORY EJECUTA CI
    // Verificar que no exista un estudiante con el mismo ID
    if (buscarPorId(estudiante.getId()).isPresent()) {
        return false; // Ya existe un estudiante con ese ID
    }
    return estudiantes.add(estudiante);                  ← 4. DEVUELVE RESULTADOS
}
```

- La UI actualiza el formulario y el grid.

```
// Mostrar resultado
if (resultado.isExito()) {
    JOptionPane.showMessageDialog(this,
        resultado.getMensaje(),
        "Éxito",
        JOptionPane.INFORMATION_MESSAGE);

    cargarDatos();                                    ← 5. UI ACTUALIZA EL GRI
    limpiarFormulario();                              ← 5. UI ACTUALIZA EL FOR
    modoEdicion = false;
    establecerEstadoInicial();
}
```

# IMPLEMENTACIÓN DEL PATRÓN SINGLETON EN LAS 3 CAPAS

**Sistema CRUD de Estudiantes - Arquitectura de 3 Capas**

**1. ESTADO INICIAL (ANTES)**

**PROBLEMAS IDENTIFICADOS:**

- Solo la Capa de Datos tenía Singleton implementado
- Capa de Lógica de Negocio: Múltiples instancias posibles
- Capa de Presentación: Múltiples ventanas posibles
- Falta de consistencia en el patrón entre capas
- Posible duplicación de recursos y estados

**2. IMPLEMENTACIÓN REALIZADA (DESPUÉS)**

## CAPA 1: DATOS - EstudianteRepository (YA TENÍA SINGLETON)

En mi clase EstudianteRepository sí estaba implementado el patrón Singleton desde el inicio. Esto se puede demostrar identificando tres elementos fundamentales que forman parte de este patrón y que mi código ya tenía correctamente:

1. **Una instancia estática privada de la propia clase**

**private static EstudianteRepository instance;**

Esta línea indica que existe una única variable estática que almacenará la única instancia permitida del repositorio en toda la aplicación. Esto es uno de los requisitos esenciales del Singleton, porque evita que existan múltiples objetos diferentes de la misma clase.

2. **Un constructor privado**

**private EstudianteRepository() { ... }**

El constructor privado impide que la clase pueda ser instanciada desde fuera con new EstudianteRepository().

Esto garantiza que la única forma de crear o acceder al repositorio sea a través del método getInstance().

Este es otro componente indispensable del patrón Singleton, porque evita la creación de instancias duplicadas.

3. **Un método público estático que crea y devuelve la única instancia**

```
    /**
     * Método para obtener la única instancia del repositorio
     */
    public static EstudianteRepository getInstance() {
        if (instance == null) {
            instance = new EstudianteRepository();
        }
        return instance;
    }
```

Este método controla el acceso global a la instancia. Verifica si la instancia ya existe y, si no existe, la crea. Si ya está creada, simplemente la devuelve. Este comportamiento es exactamente la esencia del Singleton: asegurar que solo exista un objeto de la clase y que todos los módulos de la aplicación utilicen exactamente el mismo repositorio.

En resumen, mi clase EstudianteRepository cumple completamente las reglas del patrón Singleton porque:

- Tiene una variable estática privada que guarda la instancia única.
- Tiene un constructor privado que evita que se creen objetos por fuera.
- Tiene un método estático getInstance() que garantiza que la instancia se cree solo una vez y sea accesible globalmente.

Por estas razones, puedo afirmar que mi código ya tenía correctamente implementado el patrón Singleton en la capa de datos.

## CAPA 2: LÓGICA DE NEGOCIO - EstudianteService (IMPLEMENTADO)

**ANTES:**

La clase EstudianteService no implementa el patrón Singleton porque no cumple con los elementos obligatorios del patrón.

Primero, su constructor es público:

```
     */
    public EstudianteService() {
        this.estudianteRepository = EstudianteRepository.getInstance();
    }
```

Esto permite crear todas las instancias que quiera:
new EstudianteService().

Segundo, no tiene una instancia estática de sí misma, como debería tener un Singleton. Por ejemplo, en la clase no existe algo así:

**private static EstudianteService instance;**

Y tercero, no tiene un método estático tipo getInstance() que devuelva siempre la misma instancia. Es decir, falta algo como:

**public static EstudianteService getInstance() { ... }**
Por esas razones, EstudianteService no controla su creación y por lo tanto no es Singleton.

## CAMBIOS REALIZADOS DESPUÉS CON SINGELTON

**EXPLICACIÓN DE LOS 3 COMPONENTES DEL SINGLETON:**

**1. INSTANCIA ESTÁTICA**

```
private static EstudianteService instance;
```

**¿Qué hace?** Guarda la única instancia de la clase
**¿Por qué es Singleton? S**olo puede existir UNA variable de este tipo para toda la aplicación

**2. CONSTRUCTOR PRIVADO**

```
private EstudianteService() {
    this.estudianteRepository = EstudianteRepository.getInstance();
}
```

**¿Qué hace?** Impide que se cree con new EstudianteService()
**¿Por qué es Singleton?** Nadie puede crear instancias desde fuera de la clase

**3. MÉTODO GETINSTANCE()**

```
public static synchronized EstudianteService getInstance() {
    if (instance == null) {
        instance = new EstudianteService();
    }
    return instance;
}
```

**¿Qué hace?** Es la ÚNICA manera de obtener la instancia
**¿Por qué es Singleton?** Controla que solo se cree UNA instancia

**CAPA 3: PRESENTACIÓN - EstudianteUI (IMPLEMENTADO)**
**ANTES:**

```
public class EstudianteUI extends JFrame {
```

```
    // Constructor PÚBLICO - Cualquiera puede hacer "new
EstudianteUI()"
    public EstudianteUI() {
        this.estudianteService = EstudianteService.getInstance();
        // ...
    }
}
```

**¿Por qué NO cumple Singleton?**
Constructor PÚBLICO:
**¿Qué permite?** Cualquiera puede crear múltiples ventanas con new EstudianteUI()
**¿Por qué NO es Singleton?** Permite crear MÚLTIPLES instancias de la ventana

**Sin instancia estática:**

**¿Qué falta?** private static EstudianteUI instance;
**¿Por qué NO es Singleton?** No hay control sobre cuántas instancias existen
Sin método getInstance():

**¿Qué falta?** Un método que controle la creación
**¿Por qué NO es Singleton?** No hay manera de garantizar UNA sola instancia

**IMPLEMENTANDO SINGLETON EN EstudianteUI (CAPA DE PRESENTACIÓN):**

**1. INSTANCIA ESTÁTICA:**

**private static EstudianteUI instance;**

**¿Qué hace?** Guarda LA ÚNICA instancia de la ventana
**¿Por qué es Singleton?** Solo puede existir UNA variable para toda la aplicación
**2. CONSTRUCTOR PRIVADO:**

```
private EstudianteUI() { /* Constructor privado */ }
```

**¿Qué hace?** Impide que alguien haga new EstudianteUI()
**¿Por qué es Singleton?** Solo la misma clase puede crear la ventana

**3. MÉTODO getInstance():**

```
public static EstudianteUI getInstance() {
    if (instance == null) {
```

```
        instance = new EstudianteUI();
    } else {
        instance.toFront();      // Trae ventana al frente
        instance.requestFocus(); // Da foco a la ventana
    }
    return instance;
}
```

**¿Qué hace?** Es la ÚNICA manera de obtener la ventana
**¿Por qué es Singleton?** Controla que solo exista UNA ventana, si ya existe la trae al frente

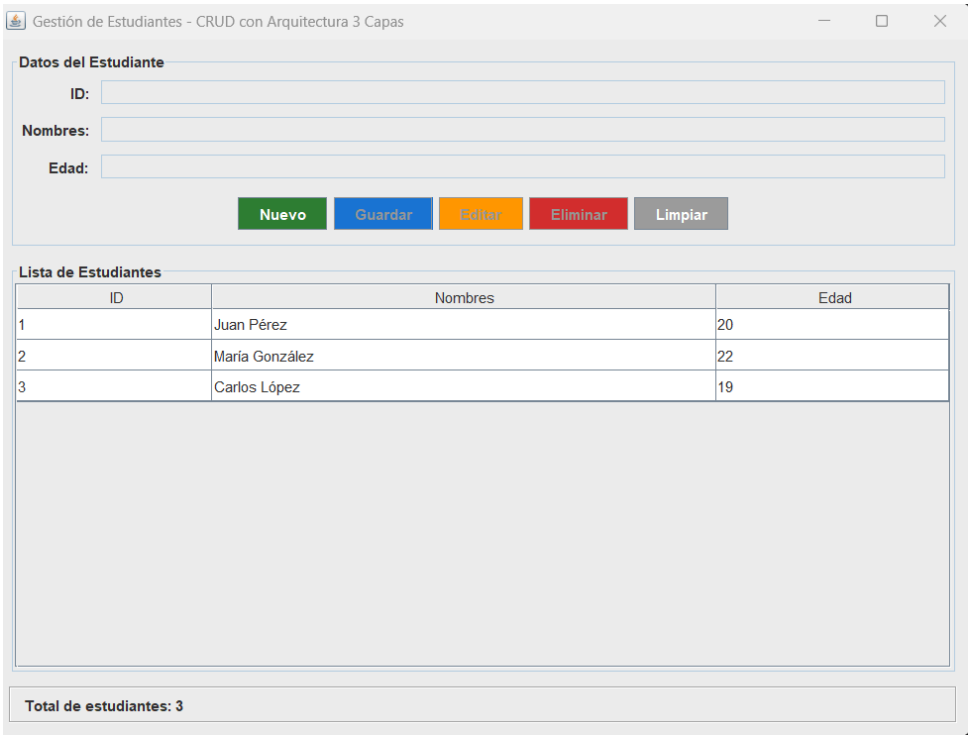
**BENEFICIO ADICIONAL EN UI:**
**ANTES**: Podrían abrirse múltiples ventanas confundiendo al usuario
**DESPUÉS**: Solo una ventana, si intentas abrir otra, trae la existente al frente

## CUADRO COMPARATIVO: SISTEMA SIN SINGLETON vs CON SINGLETON

| Aspecto | Sin Singleton (Antes) | Con Singleton (Después) | Beneficio directo |
|---|---|---|---|
| **Control de Instancias** | Cada clase podía crearse varias veces con `new`, generando múltiples objetos en memoria. | Solo se permite una única instancia por clase gracias al método `getInstance()`. | Evita duplicación innecesaria de objetos. Reduce el consumo de memoria. |
| **Consistencia de Datos** | Cada instancia tenía su propio estado → podían existir datos distintos entre objetos. | Todas las capas usan la misma instancia compartida. | Los datos siempre se mantienen sincronizados y consistentes. |
| **Acceso Global Controlado** | El acceso dependía de crear nuevas instancias cada vez. | La instancia se obtiene desde cualquier parte a través de `getInstance()`. | Facilita el acceso y reduce acoplamientos innecesarios. |
| **Manejo de Recursos** | Podía haber múltiples repositorios, servicios o ventanas en memoria. | Solo existe una instancia por capa: Datos, Servicio y UI. | Optimiza el rendimiento y evita conflictos por objetos duplicados. |

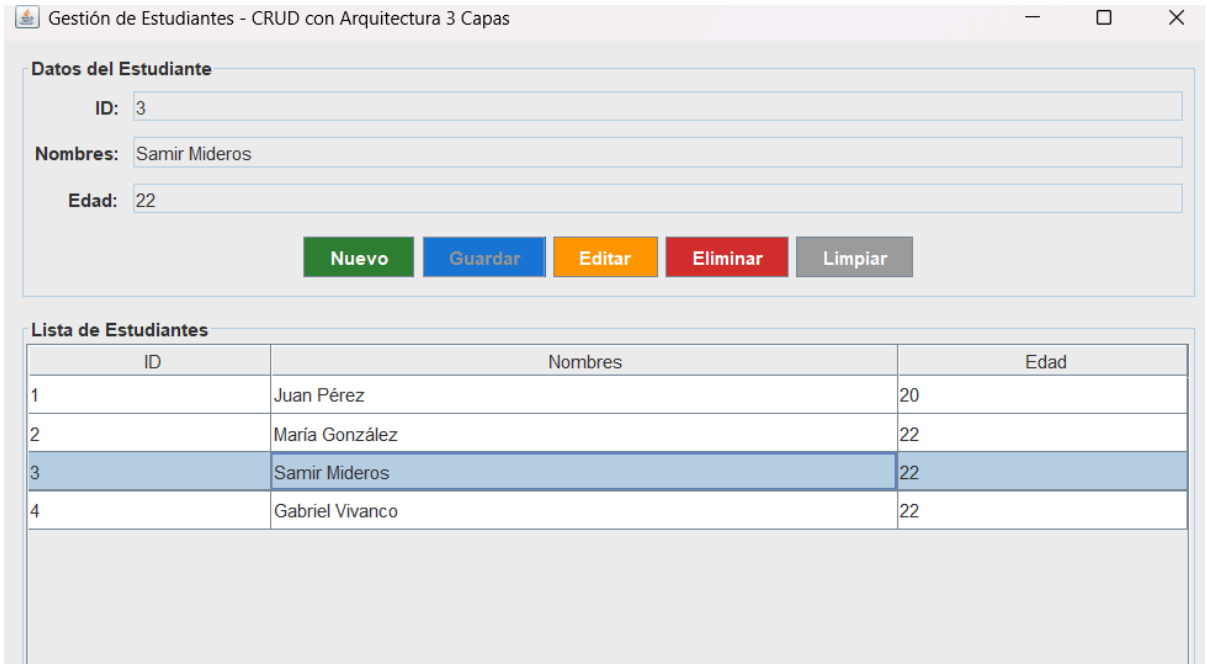
| | | | |
|---|---|---|---|
| **Capa de Presentación (UI)** | Se podían abrir varias ventanas duplicadas de EstudianteUI. | Siempre se usa la misma ventana, si existe se trae al frente. | Mejora la experiencia del usuario y evita caos visual. |
| **Capa de Lógica de Negocio (Service)** | Podía haber varios servicios aislados con reglas repetidas. | Todas las validaciones y reglas se ejecutan desde un único servicio centralizado. | Centralización de la lógica y mayor mantenibilidad. |
| **Capa de Datos (Repository)** | Riesgo de tener múltiples repositorios y listas diferentes. | Toda la aplicación usa un solo repositorio compartido. | Se evita corrupción o pérdida de datos. |
| **Complejidad** | Más código repetido, más control manual de instancias. | Código más limpio y uniforme gracias al patrón. | Mayor simplicidad y escalabilidad. |
| **Errores Comunes** | Datos duplicados, ventanas duplicadas, inconsistencias. | Estado único, ventanas únicas, acceso controlado. | Reduce fallos y mejora la estabilidad del sistema. |

**CAPTURA DE FUNCIONAMIENTO DEL PROGRAMA**

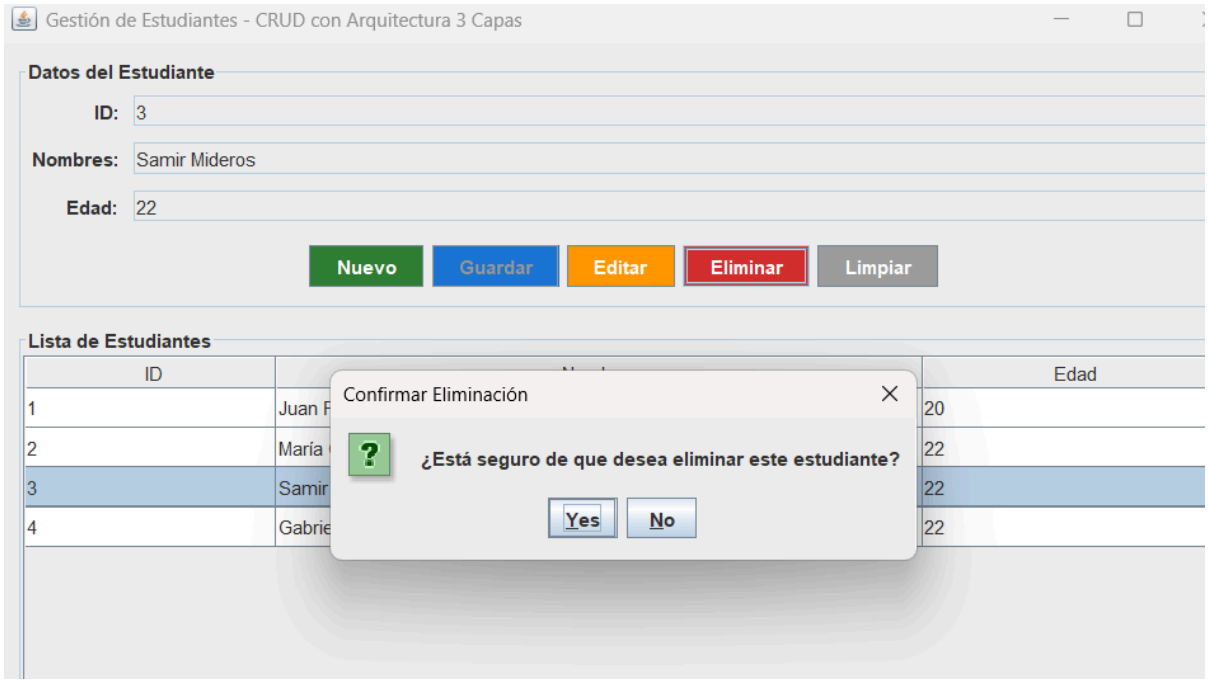


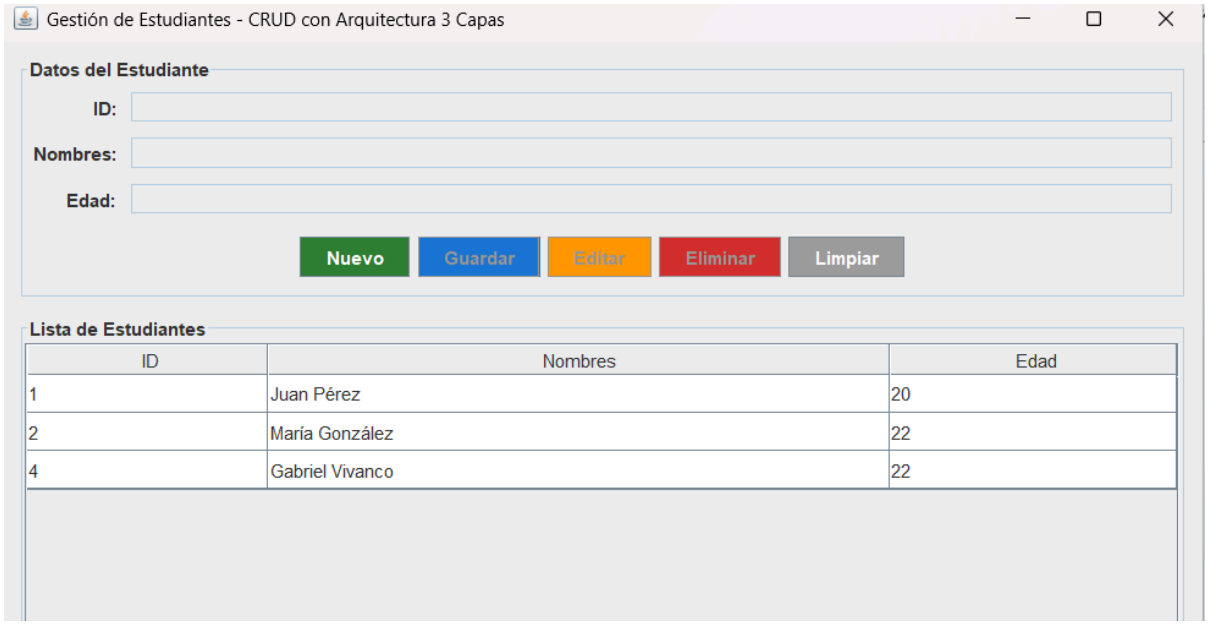
**Agregar un nuevo estudiante:**


**Editar un estudiante:**




**Eliminar un estudiante:**




## - Rúbrica de Evaluación (20 puntos)

| Criterio | Descripción | Puntaje |
|---|---|---|
| Estructura del proyecto | Implementa correctamente la arquitectura de 3 capas. | 0–5 |

| | | |
|---|---|---|
| Modelo, Servicio y Repositorio | Clases correctamente implementadas con separación de responsabilidades. | 0–5 |
| Interfaz gráfica Swing | Formulario funcional y tabla desplegando datos. | 0–5 |

| | | |
|---|---|---|
| Aplicación del patrón MVC | La vista no contiene lógica de negocio ni acceso a datos. | 0–5 |