DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CARRERA: INGENIERÍA DE SOFTWARE

NRC: 27837

ASIGNATURA: ANÁLISIS Y DISEÑO DE SOFTWARE

TEMA: Arquitectura 3 Capas e implementación de 3 patrones de diseño

Nombre:

- Marcelo Acuña
- Abner Arboleda
- Christian Bonifaz

DOCENTE: PhD. Jenny Ruiz

FECHA: 25 de Noviembre del 2025

1. Introducción y Propósito

En el presente informe se desarrolló una aplicación CRUD de estudiantes integrando la arquitectura de 3 Capas, el patrón arquitectónico MVC (Modelo-Vista-Controlador) y el patrón de diseño Singleton.

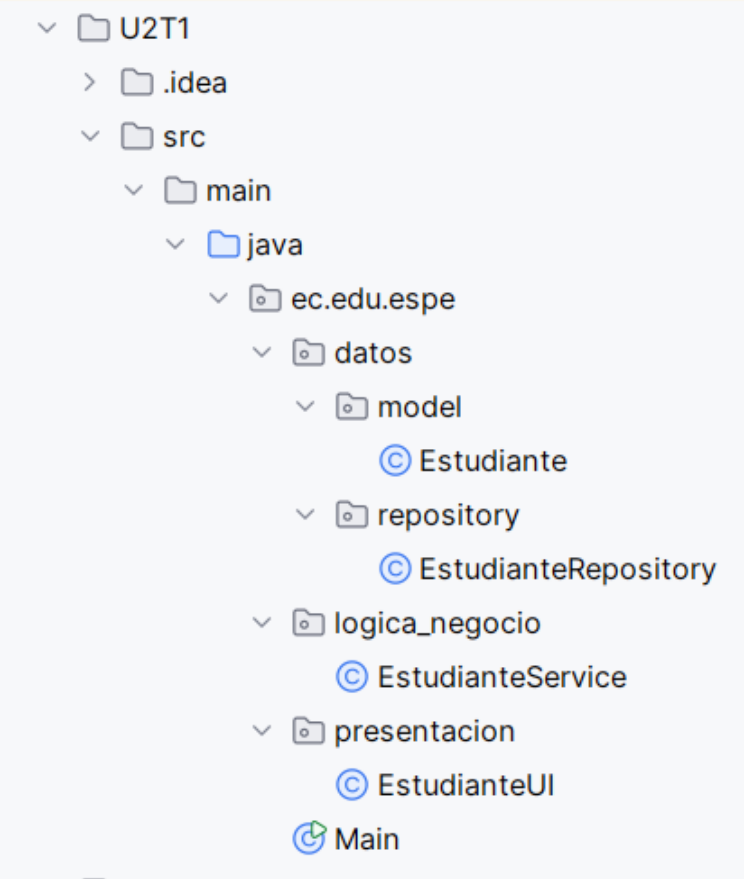
Los objetivos cumplidos en esta taller son:

1. Aplicar la arquitectura de 3 capas (Modelo, Repositorio, Servicio) para separar responsabilidades.
2. Implementar el patrón Singleton en la capa de datos para garantizar una única instancia del repositorio y la persistencia de datos en memoria.
3. Analizar las diferencias, ventajas e impactos en el mantenimiento entre MVC puro y la integración con Singleton.

2. Arquitectura

La aplicación se organizó 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
```



El MVC es un patrón de arquitectura de software que divide una aplicación en tres partes principales para separar responsabilidades y su objetivo es que la lógica del negocio no esté mezclada con la interfaz gráfica.

Para mantener el orden y la escalabilidad, la solución se dividió en cuatro paquetes o módulos, siguiendo la arquitectura GEMA/MVC.

- **El Modelo (datos.model):** Constituye la representación básica de un "Estudiante". Esta clase define únicamente los atributos: ID, Nombres y Edad , sin incluir lógica compleja ni acceso a datos.

  Capa Modelo (Estudiante.java)

  - Atributos: ID, nombres, edad
  - Constructor completo y vacío
  - Getters y setters
  - Método toString()

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

/**
 * Clase Estudiante - Modelo de dominio
 * Representa un estudiante con sus atributos básicos
 */
public class Estudiante {  adarboleda
    private String id;
    private String nombres;
    private int edad;

    /**
     * Constructor vacío
     */
    public Estudiante() {  adarboleda
    }

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

    // Getters y Setters
    public String getId() { return id; }

    public void setId(String id) { this.id = id; }

    public String getNombres() { return nombres; }

    public void setNombres(String nombres) { this.nombres = nombres; }

    public int getEdad() { return edad; }

    public void setEdad(int edad) { this.edad = edad; }

    @Override  adarboleda
    public String toString() {
        return "Estudiante{" +
                "id='" + id + '\'' +
                ", nombres='" + nombres + '\'' +
                ", edad=" + edad +
                '}';
    }
}
```

- **El Repositorio (datos.repository):** Actúa como el almacén de datos, este es el componente encargado de guardar, buscar y eliminar la información dentro de la memoria del sistema.

Capa Repository (EstudianteRepository.java)

- Patrón Singleton

- CRUD completo: agregar, editar, eliminar, listar
- Métodos auxiliares: buscarPorId, existePorId
- Persistencia en ArrayList

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

import ...

/**
 * EstudianteRepository - Capa de Acceso a Datos
 * Gestiona las operaciones CRUD utilizando una colección interna (ArrayList)
 * Implementado como Singleton para garantizar una única instancia
 */
public class EstudianteRepository {  6 usages  adarboleda
    private static EstudianteRepository instance;  3 usages
    private List<Estudiante> estudiantes;  8 usages

    /**
     * Constructor privado para patrón Singleton
     */
    private EstudianteRepository() { this.estudiantes = new ArrayList<>(); }

    /**
     * Obtiene la instancia única del repositorio
     * @return Instancia de EstudianteRepository
     */
    public static EstudianteRepository getInstance() {  1 usage  adarboleda
        if (instance == null) {
            instance = new EstudianteRepository();
        }
        return instance;
    }

    /**
     * Agrega un nuevo estudiante al repositorio
     * @param estudiante Estudiante a agregar
     * @return true si se agregó correctamente, false en caso contrario
     */
    public boolean agregar(Estudiante estudiante) {  1 usage  adarboleda
        if (estudiante == null || buscarPorId(estudiante.getId()).isPresent()) {
            return false;
        }
        return estudiantes.add(estudiante);
    }

    /**
     * Edita un estudiante existente
     * @param estudiante Estudiante con los datos actualizados
     * @return true si se editó correctamente, false si no existe
     */
    public boolean editar(Estudiante estudiante) {  1 usage  adarboleda
        if (estudiante == null) {
            return false;
        }

        for (int i = 0; i < estudiantes.size(); i++) {
            if (estudiantes.get(i).getId().equals(estudiante.getId())) {
                estudiantes.set(i, estudiante);
                return true;
            }
        }
        return false;
    }
```

```java
    /**
     * Elimina un estudiante por su ID
     * @param id Identificador del estudiante a eliminar
     * @return true si se eliminó correctamente, false si no existe
     */
    public boolean eliminar(String id) { return estudiantes.removeIf( Estudiante e -> e.getId().equals(id)); }

    /**
     * Lista todos los estudiantes
     * @return Lista de todos los estudiantes
     */
    public List<Estudiante> listar() { return new ArrayList<>(estudiantes); }

    /**
     * Busca un estudiante por su ID
     * @param id Identificador del estudiante
     * @return Optional con el estudiante si existe, vacío en caso contrario
     */
    public Optional<Estudiante> buscarPorId(String id) {  3 usages   adarboleda
        return estudiantes.stream()
                .filter( Estudiante e -> e.getId().equals(id))
                .findFirst();
    }


    /**
     * Verifica si existe un estudiante con el ID especificado
     * @param id Identificador a verificar
     * @return true si existe, false en caso contrario
     */
    public boolean existePorId(String id) { return buscarPorId(id).isPresent(); }
}
```

- **El Servicio o Lógica de Negocio (logica_negocio):** Funciona como un intermediario o controlador de reglas y antes de permitir el almacenamiento de un estudiante, este componente verifica la validez de los datos, asegurando, por ejemplo, que la edad no sea negativa o que el nombre no esté vacío.

  Capa Service (EstudianteService.java)

  - Validaciones de negocio:

    ID no vacío y no repetido

    Nombres no vacíos

    Edad > 0 y <= 120

  - Mensajes descriptivos de error
  - Delegación al repositorio

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

import ...

/**
 * EstudianteService - Capa de Lógica de Negocio
 * Aplica reglas de negocio y validaciones antes de delegar al repositorio
 */
public class EstudianteService {  3 usages  adarboleda
    private EstudianteRepository repository;  9 usages

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

    /**
     * Agrega un nuevo estudiante aplicando validaciones de negocio
     * @param estudiante Estudiante a agregar
     * @return Mensaje con el resultado de la operación
     */
    public String agregarEstudiante(Estudiante estudiante) {  1 usage  adarboleda
        // Validar que el estudiante no sea nulo
        if (estudiante == null) {
            return "Error: El estudiante no puede ser nulo";
        }

        // Validar ID
        if (estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
            return "Error: El ID no puede estar vacío";
        }

        // Validar que el ID no esté repetido
        if (repository.existePorId(estudiante.getId())) {
            return "Error: Ya existe un estudiante con el ID: " + estudiante.getId();
        }

        // Validar nombres
        if (estudiante.getNombres() == null || estudiante.getNombres().trim().isEmpty()) {
            return "Error: Los nombres no pueden estar vacíos";
        }

        // Validar edad
        if (estudiante.getEdad() <= 0) {
            return "Error: La edad debe ser mayor a 0";
        }

        if (estudiante.getEdad() > 120) {
            return "Error: La edad no puede ser mayor a 120 años";
        }

        // Si todas las validaciones pasan, agregar al repositorio
        boolean resultado = repository.agregar(estudiante);
        if (resultado) {
            return "Estudiante agregado exitosamente";
        } else {
            return "Error al agregar el estudiante";
        }
    }
```

```java
/**
 * Edita un estudiante existente aplicando validaciones
 * @param estudiante Estudiante con los datos actualizados
 * @return Mensaje con el resultado de la operación
 */
public String editarEstudiante(Estudiante estudiante) {  1 usage   adarboleda
    // Validar que el estudiante no sea nulo
    if (estudiante == null) {
        return "Error: El estudiante no puede ser nulo";
    }

    // Validar ID
    if (estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
        return "Error: El ID no puede estar vacío";
    }

    // Validar que el estudiante exista
    if (!repository.existePorId(estudiante.getId())) {
        return "Error: No existe un estudiante con el ID: " + estudiante.getId();
    }

    // Validar nombres
    if (estudiante.getNombres() == null || estudiante.getNombres().trim().isEmpty()) {
        return "Error: Los nombres no pueden estar vacíos";
    }

    // Validar edad
    if (estudiante.getEdad() <= 0) {
        return "Error: La edad debe ser mayor a 0";
    }
```

```
        if (estudiante.getEdad() > 120) {
            return "Error: La edad no puede ser mayor a 120 años";
        }

        // Si todas las validaciones pasan, editar en el repositorio
        boolean resultado = repository.editar(estudiante);
        if (resultado) {
            return "Estudiante editado exitosamente";
        } else {
            return "Error al editar el estudiante";
        }
    }

    /**
     * Elimina un estudiante por su ID
     * @param id Identificador del estudiante a eliminar
     * @return Mensaje con el resultado de la operación
     */
    public String eliminarEstudiante(String id) {  1 usage   adarboleda
        // Validar ID
        if (id == null || id.trim().isEmpty()) {
            return "Error: El ID no puede estar vacío";
        }

        // Validar que el estudiante exista
        if (!repository.existePorId(id)) {
            return "Error: No existe un estudiante con el ID: " + id;
        }

        // Eliminar del repositorio
        boolean resultado = repository.eliminar(id);
        if (resultado) {
            return "Estudiante eliminado exitosamente";
        } else {
            return "Error al eliminar el estudiante";
        }
    }

    /**
     * Lista todos los estudiantes
     * @return Lista de estudiantes
     */
    public List<Estudiante> listarEstudiantes() { return repository.listar(); }

    /**
     * Busca un estudiante por su ID
     * @param id Identificador del estudiante
     * @return Optional con el estudiante si existe
     */
    public Optional<Estudiante> buscarEstudiantePorId(String id) {  no usages   adarboleda
        if (id == null || id.trim().isEmpty()) {
            return Optional.empty();
        }
        return repository.buscarPorId(id);
    }
}
```

- **La Vista (presentacion):** Corresponde a la interfaz gráfica (ventanas y botones), donde su función exclusiva es capturar las interacciones del usuario y visualizar la información en una tabla.

Capa Presentación (EstudianteUI.java)

- Formulario con campos: ID, Nombres, Edad
- Botones: Nuevo, Guardar, Editar, Eliminar
- JTable para visualizar estudiantes
- Selección de fila para editar/eliminar
- Validaciones en la interfaz
- Confirmación de eliminación

```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.util.List;

/**
* EstudianteUI - Capa de Presentación (Vista y Controlador)
* Interfaz gráfica para la gestión de estudiantes
*/
public class EstudianteUI extends JFrame {
   private EstudianteService service;

   // Componentes del formulario
   private JTextField txtId;
   private JTextField txtNombres;
   private JTextField txtEdad;

   // Botones CRUD
   private JButton btnGuardar;
   private JButton btnEditar;
   private JButton btnEliminar;
   private JButton btnNuevo;

   // Tabla para mostrar estudiantes
   private JTable tableEstudiantes;
   private DefaultTableModel tableModel;

   /**
    * Constructor que inicializa la interfaz
    */
   public EstudianteUI() {
       this.service = new EstudianteService();
       initComponents();
```

```
        actualizarTabla();
    }

    /**
     * Inicializa y configura los componentes de la interfaz
     */
    private void initComponents() {
        setTitle("Gestión de Estudiantes - CRUD");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(800, 600);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout(10, 10));

        // Panel superior con el formulario
        JPanel panelFormulario = crearPanelFormulario();
        add(panelFormulario, BorderLayout.NORTH);

        // Panel central con la tabla
        JPanel panelTabla = crearPanelTabla();
        add(panelTabla, BorderLayout.CENTER);

        // Panel inferior con los botones
        JPanel panelBotones = crearPanelBotones();
        add(panelBotones, BorderLayout.SOUTH);
    }

    /**
     * Crea el panel del formulario con los campos de entrada
     */
    private JPanel crearPanelFormulario() {
        JPanel panel = new JPanel();
        panel.setBorder(BorderFactory.createTitledBorder("Datos del
Estudiante"));
        panel.setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);
        gbc.fill = GridBagConstraints.HORIZONTAL;

        // Campo ID
        gbc.gridx = 0;
        gbc.gridy = 0;
        panel.add(new JLabel("ID:"), gbc);

        gbc.gridx = 1;
        txtId = new JTextField(20);
        panel.add(txtId, gbc);
```

```
        // Campo Nombres
        gbc.gridx = 0;
        gbc.gridy = 1;
        panel.add(new JLabel("Nombres:"), gbc);

        gbc.gridx = 1;
        txtNombres = new JTextField(20);
        panel.add(txtNombres, gbc);

        // Campo Edad
        gbc.gridx = 0;
        gbc.gridy = 2;
        panel.add(new JLabel("Edad:"), gbc);

        gbc.gridx = 1;
        txtEdad = new JTextField(20);
        panel.add(txtEdad, gbc);

        return panel;
    }

    /**
     * Crea el panel con la tabla de estudiantes
     */
    private JPanel crearPanelTabla() {
        JPanel panel = new JPanel(new BorderLayout());
        panel.setBorder(BorderFactory.createTitledBorder("Lista de
Estudiantes"));

        // Crear modelo de tabla
        String[] columnas = {"ID", "Nombres", "Edad"};
        tableModel = new DefaultTableModel(columnas, 0) {
            @Override
            public boolean isCellEditable(int row, int column) {
                return false; // Hacer la tabla no editable
            }
        };

        // Crear tabla
        tableEstudiantes = new JTable(tableModel);
tableEstudiantes.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Agregar listener para selección de filas
        tableEstudiantes.getSelectionModel().addListSelectionListener(e -> {
            if (!e.getValueIsAdjusting()) {
                cargarEstudianteSeleccionado();
```

```
        }
    });

    JScrollPane scrollPane = new JScrollPane(tableEstudiantes);
    panel.add(scrollPane, BorderLayout.CENTER);

    return panel;
}

/**
 * Crea el panel con los botones de acción
 */
private JPanel crearPanelBotones() {
    JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));

    // Botón Nuevo
    btnNuevo = new JButton("Nuevo");
    btnNuevo.addActionListener(e -> limpiarFormulario());
    panel.add(btnNuevo);

    // Botón Guardar
    btnGuardar = new JButton("Guardar");
    btnGuardar.addActionListener(e -> guardarEstudiante());
    panel.add(btnGuardar);

    // Botón Editar
    btnEditar = new JButton("Editar");
    btnEditar.addActionListener(e -> editarEstudiante());
    panel.add(btnEditar);

    // Botón Eliminar
    btnEliminar = new JButton("Eliminar");
    btnEliminar.addActionListener(e -> eliminarEstudiante());
    panel.add(btnEliminar);

    return panel;
}

/**
 * Guarda un nuevo estudiante
 */
private void guardarEstudiante() {
    try {
        // Obtener datos del formulario
        String id = txtId.getText().trim();
        String nombres = txtNombres.getText().trim();
```

```
            int edad = Integer.parseInt(txtEdad.getText().trim());

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

            // Llamar al servicio
            String resultado = service.agregarEstudiante(estudiante);

            // Mostrar resultado
            if (resultado.contains("exitosamente")) {
                JOptionPane.showMessageDialog(this, resultado, "Éxito",
JOptionPane.INFORMATION_MESSAGE);
                limpiarFormulario();
                actualizarTabla();
            } else {
                JOptionPane.showMessageDialog(this, resultado, "Error",
JOptionPane.ERROR_MESSAGE);
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Error: La edad debe ser un
número válido", "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Edita un estudiante existente
     */
    private void editarEstudiante() {
        try {
            // Obtener datos del formulario
            String id = txtId.getText().trim();
            String nombres = txtNombres.getText().trim();
            int edad = Integer.parseInt(txtEdad.getText().trim());

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

            // Llamar al servicio
            String resultado = service.editarEstudiante(estudiante);

            // Mostrar resultado
            if (resultado.contains("exitosamente")) {
                JOptionPane.showMessageDialog(this, resultado, "Éxito",
JOptionPane.INFORMATION_MESSAGE);
                limpiarFormulario();
                actualizarTabla();
            } else {
```

```
                JOptionPane.showMessageDialog(this, resultado, "Error",
JOptionPane.ERROR_MESSAGE);
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Error: La edad debe ser un
número válido", "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Elimina un estudiante
     */
    private void eliminarEstudiante() {
        String id = txtId.getText().trim();

        if (id.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Debe seleccionar un
estudiante para eliminar", "Advertencia", JOptionPane.WARNING_MESSAGE);
            return;
        }

        // Confirmar eliminación
        int confirmacion = JOptionPane.showConfirmDialog(this,
                "¿Está seguro de eliminar el estudiante con ID: " + id + "?",
                "Confirmar eliminación",
                JOptionPane.YES_NO_OPTION);

        if (confirmacion == JOptionPane.YES_OPTION) {
            String resultado = service.eliminarEstudiante(id);

            if (resultado.contains("exitosamente")) {
                JOptionPane.showMessageDialog(this, resultado, "Éxito",
JOptionPane.INFORMATION_MESSAGE);
                limpiarFormulario();
                actualizarTabla();
            } else {
                JOptionPane.showMessageDialog(this, resultado, "Error",
JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    /**
     * Actualiza la tabla con los datos del servicio
     */
    private void actualizarTabla() {
        // Limpiar tabla
        tableModel.setRowCount(0);
```

```
        // Obtener estudiantes del servicio
        List<Estudiante> estudiantes = service.listarEstudiantes();

        // Agregar estudiantes a la tabla
        for (Estudiante estudiante : estudiantes) {
            Object[] fila = {
                estudiante.getId(),
                estudiante.getNombres(),
                estudiante.getEdad()
            };
            tableModel.addRow(fila);
        }
    }

    /**
     * Carga los datos del estudiante seleccionado en el formulario
     */
    private void cargarEstudianteSeleccionado() {
        int filaSeleccionada = tableEstudiantes.getSelectedRow();

        if (filaSeleccionada >= 0) {
            txtId.setText(tableModel.getValueAt(filaSeleccionada,
0).toString());
            txtNombres.setText(tableModel.getValueAt(filaSeleccionada,
1).toString());
            txtEdad.setText(tableModel.getValueAt(filaSeleccionada,
2).toString());

            // Deshabilitar el campo ID al cargar un estudiante
            txtId.setEnabled(false);
        }
    }

    /**
     * Limpia el formulario
     */
    private void limpiarFormulario() {
        txtId.setText("");
        txtNombres.setText("");
        txtEdad.setText("");
        txtId.setEnabled(true);
        tableEstudiantes.clearSelection();
    }
}
```

Flujo de la Información: La interacción se inicia en la Vista, la cual comunica los datos al Servicio; este último valida la información y, finalmente, delega el almacenamiento al Repositorio.

3. Descripción de Cambios e Implementación del Patrón Singleton

Se identificó el riesgo de que, al abrir múltiples ventanas o instancias del servicio, se crearan múltiples listas de estudiantes independientes, lo que ocasionaría la pérdida o desincronización de los datos.

Por esto para mitigar este problema, se aplicó el patrón de diseño *Singleton* en la clase EstudianteRepository, lo que fuerza al sistema a utilizar siempre la misma instancia de la lista.

Cambios Realizados en EstudianteRepository.java

1. Modificación del Constructor:

Se cambió el constructor de public a private para evitar la instanciación directa mediante el operador new, lo que garantiza que no se puedan crear múltiples instancias del repositorio desde otras clases.

```
/**
 * Constructor privado para patrón Singleton
 */
private EstudianteRepository() { this.estudiantes = new ArrayList<>(); }
```

2. Adición de Variable Estática:

Se agregó el atributo estático private static EstudianteRepository instance para almacenar la única instancia de la clase, y al ser estática, esta variable es compartida por todas las referencias a la clase, no pertenece a ninguna instancia particular.

```
private static EstudianteRepository instance;  3 usages
```

3. Implementación del Método getInstance():

Se creó el método público estático getInstance() como único punto de acceso a la instancia del repositorio, el cual solo crea la instancia la primera vez que se solicita y en llamadas posteriores, retorna siempre la misma instancia previamente creada.

```java
/**
 * Obtiene la instancia única del repositorio
 * @return Instancia de EstudianteRepository
 */
public static EstudianteRepository getInstance() {  1 usage   adarboleda
    if (instance == null) {
        instance = new EstudianteRepository();
    }
    return instance;
}
```

Resultado: Independientemente de cuántas veces se invoque al servicio, todas las operaciones de lectura y escritura se realizan sobre el mismo espacio de memoria, garantizando la persistencia temporal de los datos.

4. Descripción de Cambios e Implementación del Patrón Strategy

Se identificó que el código de validación de estudiantes estaba hardcodeado directamente en el método agregarEstudiante() de la clase EstudianteService, creando un bloque monolítico de validaciones difícil de mantener, extender y reutilizar. Cada vez que se requería agregar, modificar o eliminar una validación, era necesario modificar directamente el método, violando el Principio Open/Closed (abierto para extensión, cerrado para modificación).

Por esto, para resolver este problema de diseño, se aplicó el patrón de diseño Strategy, que permite encapsular cada algoritmo de validación en clases independientes e intercambiables, facilitando la extensibilidad y mantenibilidad del código sin necesidad de modificar la lógica principal del servicio.

Cambios Realizados en la Estructura del Proyecto

1. Creación de la Interfaz Base ValidationStrategy.java:

Se creó la interfaz ValidationStrategy que define el contrato que todas las estrategias de validación deben cumplir. Esta interfaz declara el método validate(Estudiante estudiante) que retorna un String con el mensaje de error si la validación falla, o null si la validación es exitosa. Al ser una interfaz, permite que múltiples clases implementen diferentes tipos de validaciones manteniendo una estructura común.


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

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

/**
 * ValidationStrategy - Patrón Strategy para validaciones
 * Define el contrato para las diferentes estrategias de validación
 */
public interface ValidationStrategy {
    /**
     * Valida un estudiante según la estrategia específica
     * @param estudiante Estudiante a validar
     * @return null si la validación es exitosa, mensaje de error en caso contrario
     */
    String validate(Estudiante estudiante);
}
```

2. Implementación de Estrategias Concretas:

Se crearon tres clases concretas que implementan la interfaz ValidationStrategy, cada una responsable de validar un aspecto específico del estudiante:

- **IdValidationStrategy.java:** Valida que el ID del estudiante no sea nulo, no esté vacío y no contenga sólo espacios en blanco.

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

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

/**
 * IdValidationStrategy - Validación del ID del estudiante
 * Implementación del patrón Strategy para validar el identificador
 */
public class IdValidationStrategy implements ValidationStrategy {

    @Override
    public String validate(Estudiante estudiante) {
        if (estudiante == null) {
            return "Error: El estudiante no puede ser nulo";
        }

        if (estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
            return "Error: El ID no puede estar vacío";
        }

        // Validar formato de ID (opcional)
        if (estudiante.getId().length() < 3) {
            return "Error: El ID debe tener al menos 3 caracteres";
        }

        return null; // Validación exitosa
    }
}
```

- **NombresValidationStrategy.java:** Valida que el campo nombres del estudiante no sea nulo, no esté vacío y cumpla con las reglas de negocio establecidas para los nombres.

```java
public class NombresValidationStrategy implements ValidationStrategy {

    @Override
    public String validate(Estudiante estudiante) {
        if (estudiante == null) {
            return "Error: El estudiante no puede ser nulo";
        }

        if (estudiante.getNombres() == null || estudiante.getNombres().trim().isEmpty()) {
            return "Error: Los nombres no pueden estar vacíos";
        }

        // Validar que tenga al menos dos caracteres
        if (estudiante.getNombres().trim().length() < 2) {
            return "Error: Los nombres deben tener al menos 2 caracteres";
        }

        // Validar que solo contenga letras y espacios
        if (!estudiante.getNombres().matches("^[a-zA-ZáéíóúÁÉÍÓÚñÑ\\s]+$")) {
            return "Error: Los nombres solo pueden contener letras y espacios";
        }

        return null; // Validación exitosa
    }
}
```

- **EdadValidationStrategy.java:** Valida que la edad del estudiante sea un valor positivo y se encuentre dentro de rangos lógicos aceptables (mayor a 0 y menor o igual a 120 años).

```java
public class EdadValidationStrategy implements ValidationStrategy {

    private static final int EDAD_MINIMA = 1;
    private static final int EDAD_MAXIMA = 120;

    @Override
    public String validate(Estudiante estudiante) {
        if (estudiante == null) {
            return "Error: El estudiante no puede ser nulo";
        }

        if (estudiante.getEdad() <= 0) {
            return "Error: La edad debe ser mayor a 0";
        }

        if (estudiante.getEdad() < EDAD_MINIMA) {
            return "Error: La edad mínima permitida es " + EDAD_MINIMA;
        }

        if (estudiante.getEdad() > EDAD_MAXIMA) {
            return "Error: La edad no puede ser mayor a " + EDAD_MAXIMA + " años";
        }

        return null; // Validación exitosa
    }
}
```

Cada estrategia encapsula su propia lógica de validación de forma independiente y cohesiva, permitiendo que sean probadas, modificadas o reemplazadas sin afectar a las demás.

3. Creación de la Clase Contexto ValidationContext.java:

Se implementó la clase ValidationContext que actúa como el ejecutor de estrategias. Esta clase mantiene una lista de estrategias de validación y proporciona el método validate(Estudiante estudiante) que itera sobre todas las estrategias registradas. Si alguna estrategia detecta un error, el proceso se detiene inmediatamente y retorna el mensaje de error; si todas las validaciones son exitosas, retorna null. Además, incluye el método addStrategy(ValidationStrategy strategy) para agregar nuevas estrategias de forma dinámica.

```java
public class ValidationContext {
    private List<ValidationStrategy> strategies;

    public ValidationContext() { this.strategies = new ArrayList<>(); }

    /**
     * Agrega una estrategia de validación
     * @param strategy Estrategia a agregar
     */
    public void addStrategy(ValidationStrategy strategy) { strategies.add(strategy); }

    /**
     * Valida un estudiante usando todas las estrategias configuradas
     * @param estudiante Estudiante a validar
     * @return null si todas las validaciones pasan, mensaje de error del primer fallo
     */
    public String validate(Estudiante estudiante) {
        for (ValidationStrategy strategy : strategies) {
            String resultado = strategy.validate(estudiante);
            if (resultado != null) {
                return resultado; // Retorna el primer error encontrado
            }
        }
        return null; // Todas las validaciones pasaron
    }

    /**
     * Limpia todas las estrategias
     */
    public void clearStrategies() { strategies.clear(); }
}
```

4. Refactorización de EstudianteService.java:

Se realizaron modificaciones significativas en la clase EstudianteService para integrar el patrón Strategy:

ANTES: El método agregarEstudiante() contenía múltiples bloques if anidados con validaciones hardcodeadas directamente en el código, mezclando la lógica de validación con la lógica de negocio principal.

```java
public String agregarEstudiante(Estudiante estudiante) {
    // Validaciones hardcodeadas
    if (estudiante == null) {
        return "Error: El estudiante no puede ser nulo";
    }
    if (estudiante.getId() == null || estudiante.getId().trim().isEmpty()) {
        return "Error: El ID no puede estar vacío";
    }
    if (estudiante.getNombres() == null || estudiante.getNombres().trim().isEmpty()) {
        return "Error: Los nombres no pueden estar vacíos";
    }
    if (estudiante.getEdad() <= 0) {
        return "Error: La edad debe ser mayor a 0";
    }
    // ... más validaciones
}
```

DESPUÉS:

- Se agregó el atributo privado ValidationContext validationContext para gestionar todas las validaciones.
- Se modificó el constructor para inicializar el contexto de validación y llamar al método configurarValidaciones().
- Se creó el método privado configurarValidaciones() que registra todas las estrategias de validación necesarias mediante addStrategy().
- El método agregarEstudiante() ahora delega toda la validación a una única llamada: validationContext.validate(estudiante), simplificando drásticamente el código y separando responsabilidades.

```java
private ValidationContext validationContext;

public EstudianteService() {
    this.repository = EstudianteRepository.getInstance();
    this.validationContext = new ValidationContext();
    configurarValidaciones();
}

private void configurarValidaciones() {
    validationContext.addStrategy(new IdValidationStrategy());
    validationContext.addStrategy(new NombresValidationStrategy());
    validationContext.addStrategy(new EdadValidationStrategy());
}

public String agregarEstudiante(Estudiante estudiante) {
    // Aplicar todas las estrategias de validación
    String errorValidacion = validationContext.validate(estudiante);
    if (errorValidacion != null) {
        return errorValidacion;
    }
    // ... resto del código
}
```

Resultado

El código se volvió altamente extensible y mantenible. Para agregar una nueva validación (por ejemplo, validación de email o teléfono), solo es necesario:

1. Crear una nueva clase que implemente ValidationStrategy
2. Agregar la instancia de esta nueva estrategia en el método configurarValidaciones()

No se requiere modificar la lógica existente en agregarEstudiante() ni en las demás estrategias, cumpliendo así con el Principio Open/Closed y los principios SOLID. Las validaciones ahora son modulares, reutilizables y fáciles de probar unitariamente de forma aislada. El patrón Strategy transformó un código rígido y monolítico en una solución flexible y profesional.

5. Descripción de Cambios e Implementación del Patrón Observer

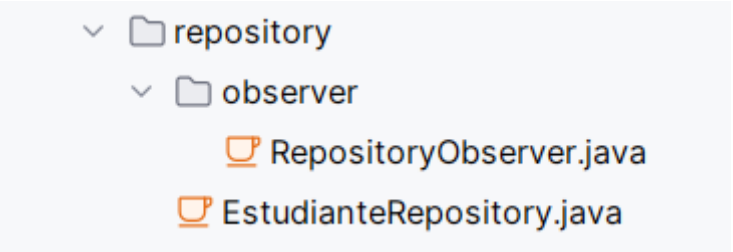
Se identificó que la actualización de la interfaz gráfica estaba fuertemente acoplada a la lógica de negocio, requiriendo llamadas manuales al método actualizarTabla() después de cada operación CRUD (crear, leer, actualizar, eliminar). Esto generaba código repetitivo en múltiples métodos de la clase EstudianteUI, aumentando el riesgo de olvidar actualizar la tabla en alguna operación y causando inconsistencias entre los datos del repositorio y lo que se mostraba al usuario. Además, este acoplamiento directo violaba el Principio de Responsabilidad Única, ya que la UI debía conocer cuándo y cómo actualizarse en lugar de reaccionar automáticamente a los cambios.

Por esto, para resolver este problema de acoplamiento y sincronización, se aplicó el patrón de diseño Observer, que establece una relación de dependencia uno-a-muchos entre objetos, permitiendo que cuando el objeto observable (el repositorio) cambia su estado, todos sus observadores (las interfaces de usuario) sean notificados y actualizados automáticamente sin necesidad de llamadas manuales explícitas.

Cambios Realizados en la Estructura del Proyecto

1. Creación de la Interfaz RepositoryObserver.java:

Se creó la interfaz RepositoryObserver dentro del paquete datos/repository/observer/ que define el contrato que todos los observadores deben cumplir. Esta interfaz declara el método onDataChanged() que será invocado automáticamente cuando el repositorio notifique cambios en los datos. Cualquier clase que desee ser notificada de cambios en el repositorio debe implementar esta interfaz, garantizando un mecanismo estándar de comunicación entre el sujeto observable y sus observadores.




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

/**
 * RepositoryObserver - Patrón Observer
 * Define el contrato para los observadores del repositorio
 * Los objetos que implementen esta interfaz serán notificados cuando cambien los datos
 */
public interface RepositoryObserver {
    /**
     * Método llamado cuando los datos del repositorio cambian
     */
    void onDataChanged();
}
```

2. Modificación de EstudianteRepository.java (Sujeto Observable):

Se realizaron cambios significativos en la clase EstudianteRepository para convertirla en un sujeto observable:

a) Adición de la Lista de Observadores: Se agregó el atributo privado List<RepositoryObserver> observers para mantener un registro de todos los objetos que están observando el repositorio y desean ser notificados cuando ocurran cambios en los datos.

```java
private List<RepositoryObserver> observers;
```

b) Inicialización en el Constructor: Se modificó el constructor privado para inicializar la lista de observadores como un ArrayList vacío, preparando el repositorio para gestionar múltiples observadores desde su creación.

```java
/**
 * Constructor privado para patrón Singleton
 */
private EstudianteRepository() {
    this.estudiantes = new ArrayList<>();
    this.observers = new ArrayList<>();
}
```

c) Método addObserver(RepositoryObserver observer): Se implementó este método público que permite registrar nuevos observadores en el repositorio. Incluye validaciones para evitar agregar observadores nulos o duplicados, garantizando la integridad de la lista de observadores.

```
/**
 * Agrega un observador al repositorio
 * @param observer Observador a agregar
 */
public void addObserver(RepositoryObserver observer) {
    if (observer != null && !observers.contains(observer)) {
        observers.add(observer);
    }
}
```

d) Método removeObserver(RepositoryObserver observer): Se creó este método público para permitir que los observadores se desregistren del repositorio cuando ya no necesiten recibir notificaciones, facilitando la gestión del ciclo de vida de los observadores.

```
/**
 * Elimina un observador del repositorio
 * @param observer Observador a eliminar
 */
public void removeObserver(RepositoryObserver observer) { observers.remove(observer); }
```

e) Método notifyObservers(): Se implementó este método privado que itera sobre todos los observadores registrados e invoca su método onDataChanged(), propagando la notificación de cambios a todos los interesados de forma automática.

```
/**
 * Notifica a todos los observadores sobre cambios en los datos
 */
private void notifyObservers() {
    for (RepositoryObserver observer : observers) {
        observer.onDataChanged();
    }
}
```

f) Integración en Métodos CRUD: Se modificaron los métodos agregar(), editar() y eliminar() para que, después de realizar exitosamente la operación correspondiente, invoquen al método notifyObservers(), asegurando que

cualquier cambio en los datos sea comunicado inmediatamente a todos los observadores registrados.

```java
// Lista de observadores
private List<RepositoryObserver> observers;

// Constructor inicializa la lista
private EstudianteRepository() {
    this.estudiantes = new ArrayList<>();
    this.observers = new ArrayList<>();  // ← NUEVO
}

// Métodos para gestionar observadores
public void addObserver(RepositoryObserver observer) {
    if (observer != null && !observers.contains(observer)) {
        observers.add(observer);
    }
}

public void removeObserver(RepositoryObserver observer) {
    observers.remove(observer);
}

private void notifyObservers() {
    for (RepositoryObserver observer : observers) {
        observer.onDataChanged();
    }
}

// Modificación en métodos CRUD
public boolean agregar(Estudiante estudiante) {
    // ... código existente
    boolean resultado = estudiantes.add(estudiante);
    if (resultado) {
        notifyObservers();  // ← NUEVO: Notificar cambios
    }
    return resultado;
}

// Similar en editar() y eliminar()
```

3. Refactorización de EstudianteUI.java (Observador Concreto):

Se realizaron modificaciones sustanciales en la clase de interfaz gráfica para implementar el patrón Observer:

ANTES:

- La clase EstudianteUI no implementaba ninguna interfaz relacionada con observación.
- Cada método que realizaba operaciones CRUD (guardarEstudiante(), editarEstudiante(), eliminarEstudiante()) contenía una llamada manual a actualizarTabla() al final del método.
- Esto generaba código duplicado y dependencia directa entre las operaciones de datos y la actualización de la UI.

- Era fácil olvidar agregar la llamada a actualizarTabla() en nuevas operaciones, causando bugs de sincronización.

```java
public class EstudianteUI extends JFrame {
    // ... campos

    private void guardarEstudiante() {
        // ... guardar estudiante
        if (resultado.contains("exitosamente")) {
            JOptionPane.showMessageDialog(this, resultado, "Éxito", JOptionPane.INFORMATION_MESSAGE);
            limpiarFormulario();
            actualizarTabla();  // ← Llamada manual
        }
    }
}
```

DESPUÉS:

a) Implementación de la Interfaz: Se modificó la declaración de la clase para que implemente RepositoryObserver: public class EstudianteUI extends JFrame implements RepositoryObserver, estableciendo el contrato de observador.

```
public class EstudianteUI extends JFrame implements RepositoryObserver {
```

b) Registro como Observador en el Constructor: Se agregó en el constructor la línea EstudianteRepository.getInstance().addObserver(this) que registra la instancia de la UI como observador del repositorio, estableciendo la relación de observación desde el momento en que se crea la ventana.

```
// Registrar esta UI como observador del repositorio
EstudianteRepository.getInstance().addObserver(this);
```

c) Implementación del Método onDataChanged(): Se implementó el método obligatorio de la interfaz RepositoryObserver. Este método utiliza SwingUtilities.invokeLater() para ejecutar actualizarTabla() en el hilo de eventos de Swing (Event Dispatch Thread), garantizando que las actualizaciones de la interfaz gráfica sean thread-safe y evitando problemas de concurrencia en componentes visuales.

```java
/**
 * Implementación del patrón Observer
 * Este método es llamado automáticamente cuando cambian los datos en el repositorio
 */
@Override
public void onDataChanged() {
    // Actualizar la tabla automáticamente cuando hay cambios
    SwingUtilities.invokeLater(() -> actualizarTabla());
}
```

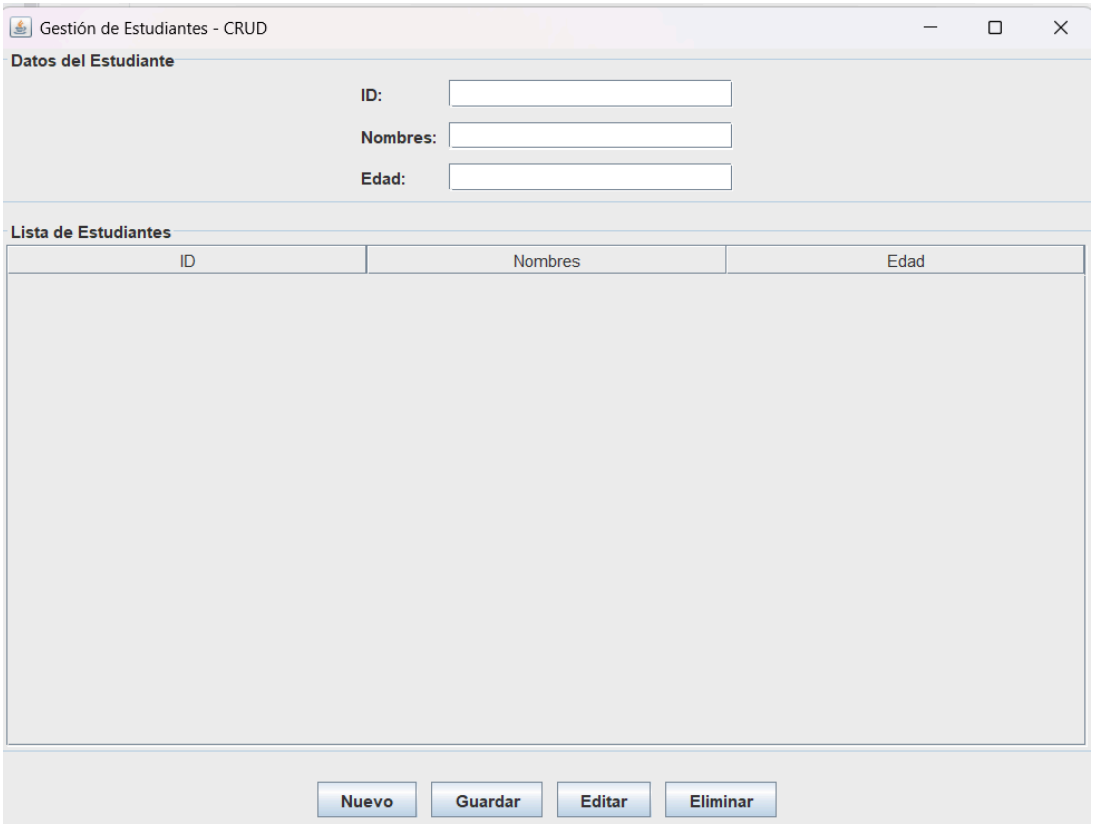
d) Eliminación de Llamadas Manuales: Se removieron todas las llamadas manuales a actualizarTabla() de los métodos guardarEstudiante(), editarEstudiante() y eliminarEstudiante(). Ahora, cuando estas operaciones se completan exitosamente y el repositorio modifica los datos, la notificación automática a través de onDataChanged() se encarga de actualizar la tabla sin intervención explícita.

```java
public class EstudianteUI extends JFrame implements RepositoryObserver {
    // ... campos

    public EstudianteUI() {
        this.service = new EstudianteService();

        // Registrar como observador
        EstudianteRepository.getInstance().addObserver(this);  // ← NUEVO

        initComponents();
        actualizarTabla();
    }

    // Implementación del patrón Observer
    @Override
    public void onDataChanged() {
        // Actualización automática cuando hay cambios
        SwingUtilities.invokeLater(() -> actualizarTabla());  // ← NUEVO
    }

    private void guardarEstudiante() {
        // ... guardar estudiante
        if (resultado.contains("exitosamente")) {
            JOptionPane.showMessageDialog(this, resultado, "Éxito", JOptionPane.INFORMATION_MESSAGE);
            limpiarFormulario();
            // Ya NO se llama actualizarTabla() - el Observer lo hace automáticamente
        }
    }
}
```

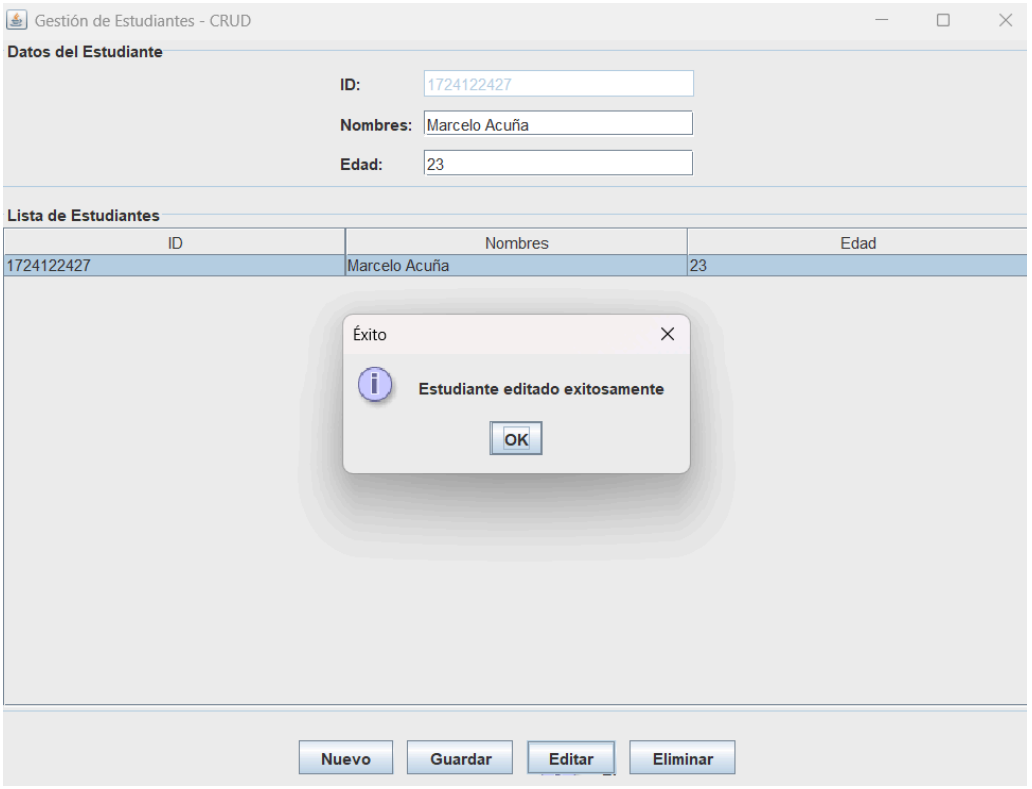
Resultado

El sistema logró un desacoplamiento efectivo entre la capa de datos y la capa de presentación. La interfaz gráfica ahora reacciona automáticamente a cualquier cambio en el repositorio sin necesidad de código repetitivo. Esta arquitectura es altamente escalable, permitiendo agregar múltiples interfaces de usuario (dashboards, reportes, paneles de administración) que observen el mismo repositorio y se actualicen sincronizadamente sin modificar el código existente.

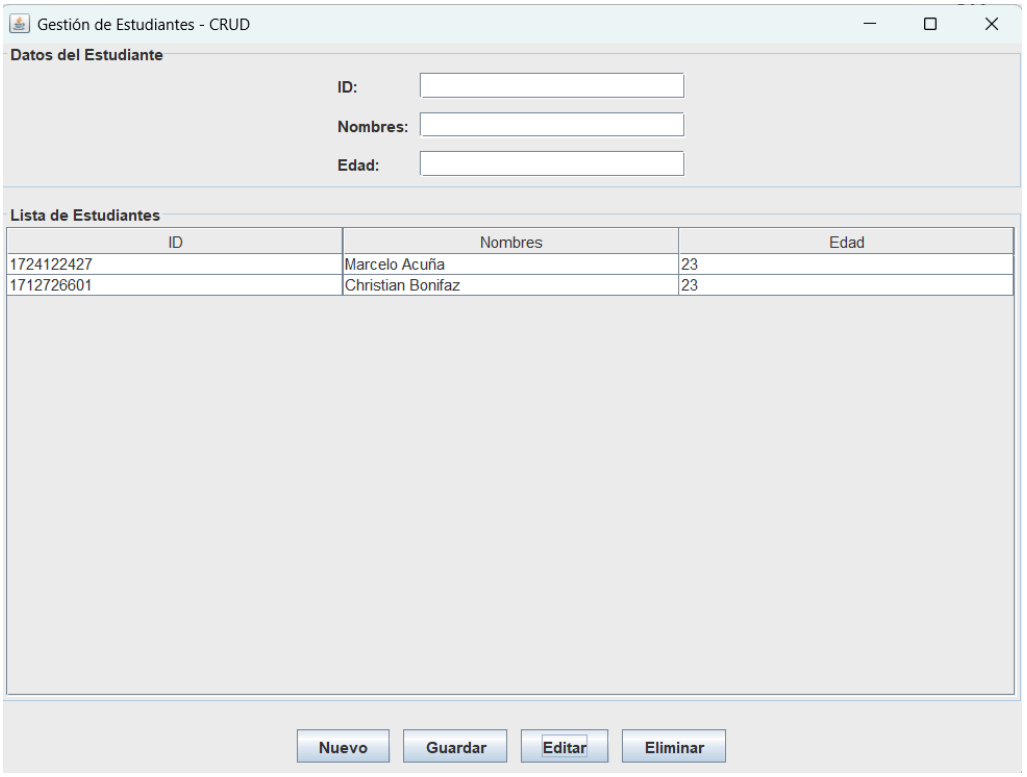
6. Pruebas y Evidencia de Funcionamiento



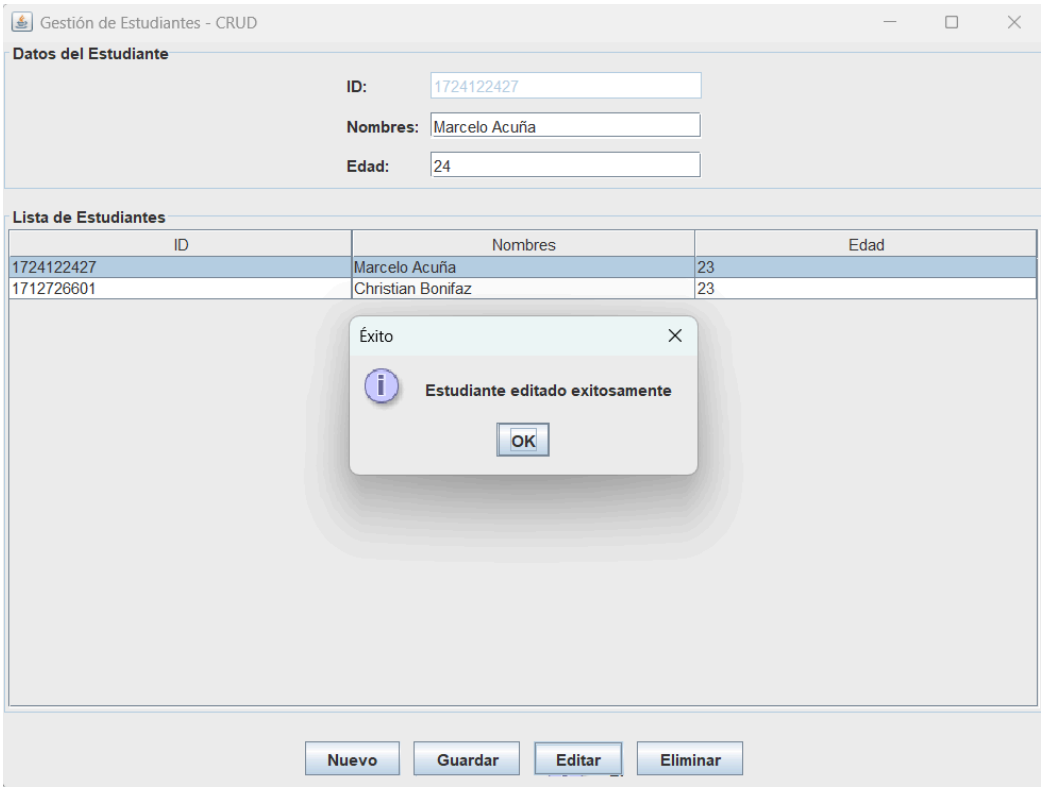
Registro: Se ingresó un estudiante (ej. "Marcelo Acuña", ID "1724122427"). El sistema validó los datos y permitió el guardado.

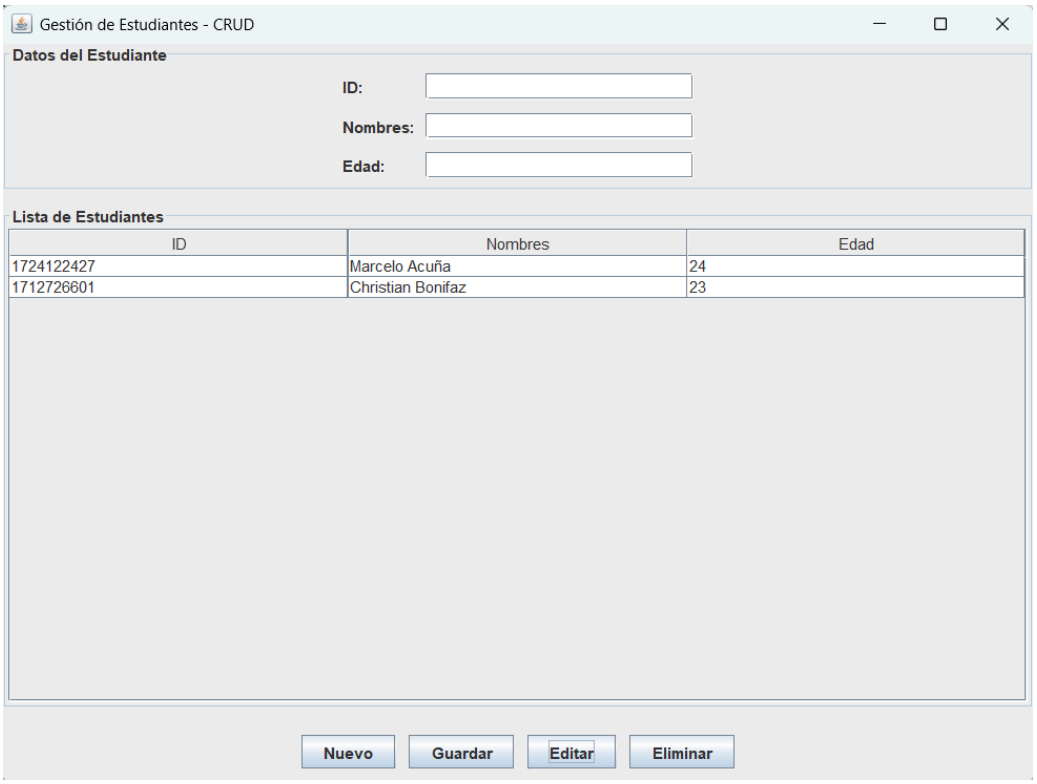


Visualización: El registro apareció de inmediato en la tabla de la interfaz.

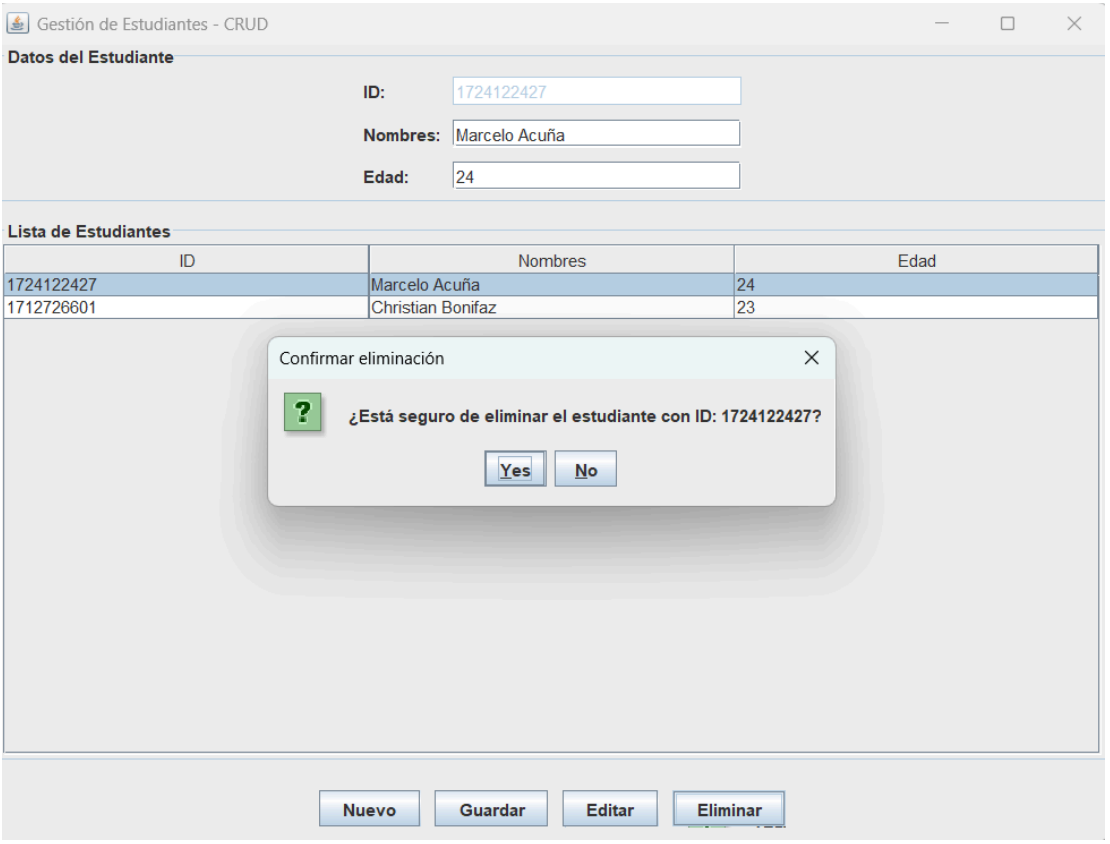


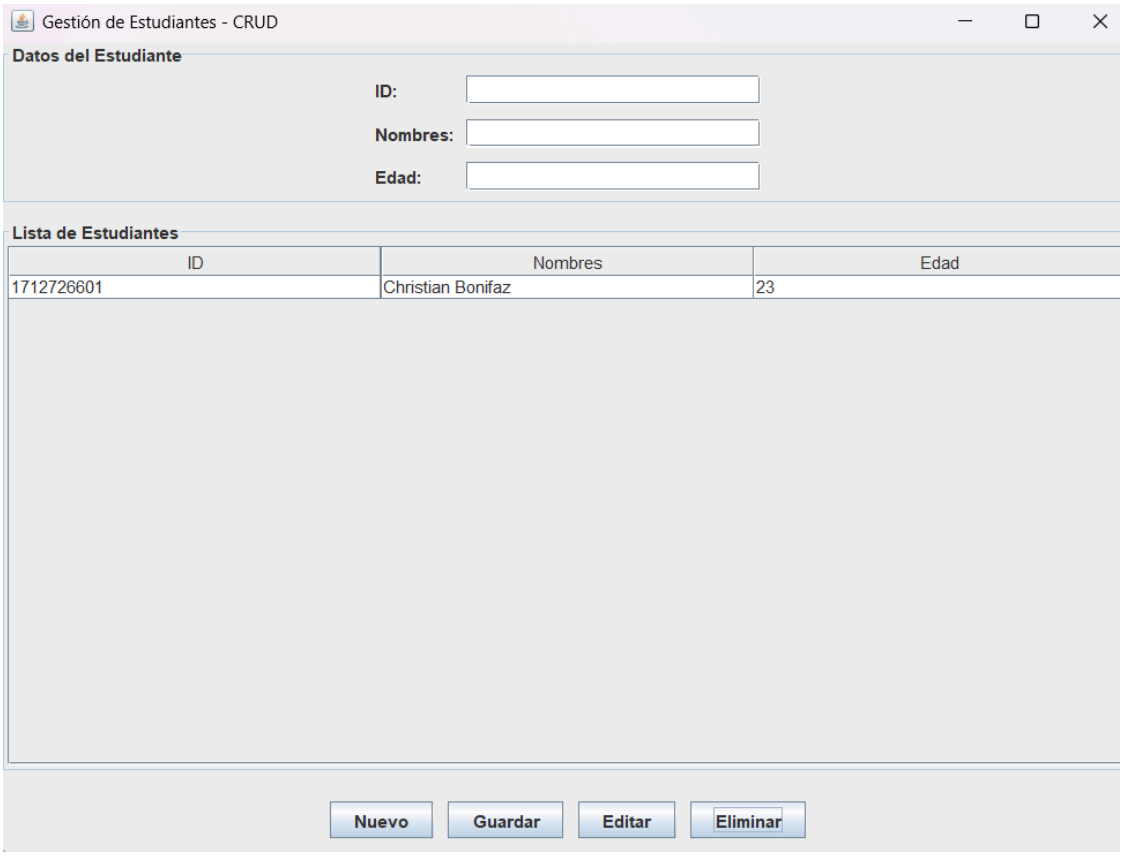
Edición: Al seleccionar el registro y modificar un campo, el sistema actualizó la información mostrando el mensaje de éxito correspondiente.





Eliminación: Se comprobó que el sistema solicita confirmación antes de borrar un registro, evitando acciones accidentales.


Gestión de Estudiantes - CRUD
Datos del Estudiante
ID:
1724122427
Nombres:
Marcelo Acuña
Edad:
24
Lista de Estudiantes
ID
Nombres
Edad
1724122427
Marcelo Acuña
24
1712726601
Christian Bonifaz
23
Confirmar eliminación
¿Está seguro de eliminar el estudiante con ID: 1724122427?
Yes
No
Nuevo
Guardar
Editar
Eliminar


Gestión de Estudiantes - CRUD
Datos del Estudiante
ID:
Nombres:
Edad:
Lista de Estudiantes
ID
Nombres
Edad
1712726601
Christian Bonifaz
23
Nuevo
Guardar
Editar
Eliminar

7. Análisis Comparativo

Comparación

Característica	Arquitectura MVC Estándar (Sin Singleton)	Arquitectura MVC + Singleton
Consistencia de Datos	Existe el riesgo de generar listas vacías independientes, perdiendo la información previa.	Se garantiza una lista única compartida, pues los datos se mantienen accesibles desde cualquier parte de la aplicación.
Eficiencia de Memoria	Mayor consumo al instanciar múltiples objetos repositorio innecesarios.	Alta eficiencia, ya que solo se mantiene un objeto repositorio en memoria.
Mantenimiento	Cambiar el método de almacenamiento requeriría ajustes en múltiples puntos.	Centralizado, ya que al existir un único punto de acceso, los cambios en la persistencia no afectan al resto del sistema.

Actividad Integrada

Criterio / Pregunta	Arquitectura MVC (Modelo-Vista-Controlador)	Patrón Singleton
¿Qué problema resuelve?	Resuelve la mezcla de responsabilidades y evita el "código espagueti" donde la lógica de negocio, los datos y la interfaz gráfica están revueltos en una sola clase, lo que hace el sistema difícil de entender.	Resuelve la inconsistencia de datos y la instanciación múltiple, pues evita que existan múltiples copias de una misma lista o recurso, asegurando que todos los

		componentes usen la misma información.
¿En qué capa se utiliza?	Es un patrón arquitectónico global que estructura toda la aplicación y se manifiesta dividiendo el proyecto en paquetes: presentacion (Vista), logica_negocio (Controlador/Servicio) y datos (Modelo).	Se utiliza específicamente en la Capa de Datos (Repository) y se aplica a la clase encargada de la persistencia (EstudianteRepository) para controlar su creación.
¿Cómo influye en el mantenimiento?	Facilita la escalabilidad, ya que permite modificar la interfaz gráfica (Vista) sin romper las reglas de negocio, o cambiar la base de datos sin afectar la pantalla, ya que las capas están desacopladas.	Centraliza el acceso, ya que si en el futuro se decide cambiar la forma de guardar datos, por ejemplo de memoria a SQL, solo se necesita modificar la clase Singleton que está en el Repository, y el resto del sistema seguirá funcionando igual mediante getInstance().
¿Cómo evita fallas de diseño?	Evita el acoplamiento fuerte, pues impide que la interfaz gráfica manipule directamente los datos sin pasar por las validaciones del negocio, reduciendo errores lógicos y bugs visuales.	Evita la desincronización de estado, ya que previene la falla común en aplicaciones de escritorio donde abrir una nueva ventana reinicia los datos; el Singleton garantiza que el estado sea persistente y compartido entre ventanas.

Comparación de los 3 patrones

Criterio	Singleton	Strategy	Observer
Categoría	Creacional	Comportamiento	Comportamiento
Propósito	Controlar instancias	Algoritmos intercambiables	Notificación de cambios
Problema que resuelve	Múltiples instancias inconsistentes	Código monolítico con lógica repetitiva	Acoplamiento entre componentes
Aplicado en	EstudianteRepository	Validaciones en EstudianteService	Comunicación Repository-UI
Clases principales	EstudianteRepository	ValidationStrategy + implementaciones (IdValidationStrategy, NombresValidationStrategy, EdadValidationStrategy), ValidationContext	RepositoryObserver, EstudianteRepository, EstudianteUI
Número de clases nuevas	0 (modificación)	5 nuevas clases	1 nueva clase + modificaciones
Complejidad	Baja	Media	Media
Flexibilidad	Baja (instancia única fija)	Alta (estrategias intercambiables)	Alta (múltiples observadores)
Reutilización	Media	Alta	Alta
Impacto en código existente	Bajo	Medio (refactorización)	Medio (refactorización)

Ventaja principal	Consistencia de datos	Extensibilidad sin modificar código	Actualizaciones automáticas
Cuándo usarlo	Recursos compartidos (DB, Config)	Múltiples variantes de algoritmos	Sincronización de estados
Escalabilidad	No escalable (1 instancia)	Muy escalable	Muy escalable
Facilita testing	Difícil (estado global)	Fácil (estrategias aisladas)	Medio

8. Conclusión

- La implementación de la arquitectura MVC (Modelo-Vista-Controlador) permitió lograr una separación efectiva de responsabilidades en la aplicación CRUD de estudiantes, donde cada capa cumple un rol específico y bien definido: el Modelo representa los datos, el Servicio valida la lógica de negocio, y la Vista gestiona la interfaz gráfica. Esta separación resultó en un código más organizado, legible y fácil de mantener, eliminando el problema del "código espagueti" donde todas las responsabilidades están mezcladas en una sola clase.

- La integración del patrón Singleton en la capa de Repositorio fue fundamental para garantizar la consistencia de datos en memoria. Al asegurar que existe una única instancia de EstudianteRepository, se eliminó el riesgo de desincronización de datos que ocurre cuando múltiples instancias manejan listas independientes, lo que mejoró significativamente la eficiencia de memoria y proporcionó un punto centralizado de acceso a los datos, facilitando futuras modificaciones en el sistema de persistencia.

- La combinación de la arquitectura MVC con el patrón Singleton demostró ser sinérgica y complementaria, pues mientras MVC estructura la aplicación en capas independientes, Singleton asegura la integridad de los datos compartidos. Esta

integración resultó en una aplicación robusta que maneja correctamente las operaciones CRUD, implementa validaciones efectivas y mantiene la persistencia temporal de datos sin errores de sincronización.

- La implementación del patrón Strategy transformó un método monolítico de validación en un sistema modular y organizado, donde cada tipo de validación tiene su propia responsabilidad claramente definida, facilitando enormemente el mantenimiento y la comprensión del código.

- La implementación del patrón Observer logró separar completamente la capa de datos (repositorio) de la capa de presentación (UI), eliminando dependencias directas y permitiendo que cada componente evolucione de forma independiente.

9. Recomendaciones

- Se sugiere aplicar el patrón Singleton no solo al Repositorio sino también a servicios compartidos como gestión de configuración, logging o validadores globales. Por ejemplo, crear un ConfigurationManager Singleton que centralice todas las configuraciones de la aplicación (rutas de archivos, parámetros de conexión, constantes de validación) evitaría duplicación de código y aseguraría que todos los componentes accedan a los mismos valores. Esto mantendría la consistencia en toda la aplicación y facilitaría cambios globales de configuración desde un único punto de acceso.