

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CARRERA: INGENIERÍA DE SOFTWARE

NRC: 27837

ASIGNATURA: ANÁLISIS Y DISEÑO DE SOFTWARE

TEMA: Arquitectura 3 Capas

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.

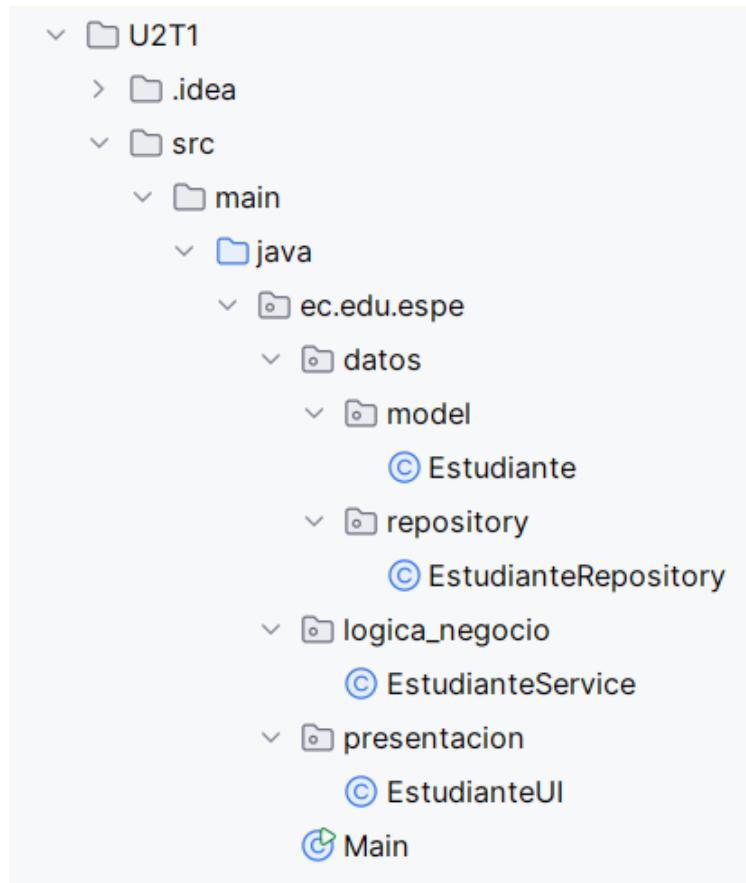
Los objetivos cumplidos en esta taller son:

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()`

```

package ec.edu.espe.datos.model;

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

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

    /**
     * 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) {
        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
    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

```

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;
    }
}

```

```

    /**
     * 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

```

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";
        }
    }
}

```

```
/**  
 * 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 vacio";
        }

        // 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

```

package ec.edu.espe.presentacion;

import ec.edu.espe.datos.modelo.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.

```

/**
 * 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. Pruebas y Evidencia de Funcionamiento

ID	Nombres	Edad

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

Gestión de Estudiantes - CRUD

Datos del Estudiante

ID:	1724122427
Nombres:	Marcelo Acuña
Edad:	23

Listado de Estudiantes

ID	Nombres	Edad
1724122427	Marcelo Acuña	23

Éxito

Estudiante editado exitosamente

OK

[Nuevo](#) [Guardar](#) [Editar](#) [Eliminar](#)

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

Gestión de Estudiantes - CRUD

Datos del Estudiante

ID:	<input type="text"/>
Nombres:	<input type="text"/>
Edad:	<input type="text"/>

Listado de Estudiantes

ID	Nombres	Edad
1724122427	Marcelo Acuña	23
1712726601	Christian Bonifaz	23

[Nuevo](#) [Guardar](#) [Editar](#) [Eliminar](#)

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

Gestión de Estudiantes - CRUD

Datos del Estudiante

ID:	1724122427
Nombres:	Marcelo Acuña
Edad:	24

Lista de Estudiantes

ID	Nombres	Edad
1724122427	Marcelo Acuña	23
1712726601	Christian Bonifaz	23

Éxito X

i Estudiante editado exitosamente

OK

Nuevo Guardar Editar Eliminar

Gestión de Estudiantes - CRUD

Datos del Estudiante

ID:	<input type="text"/>
Nombres:	<input type="text"/>
Edad:	<input type="text"/>

Lista de Estudiantes

ID	Nombres	Edad
1724122427	Marcelo Acuña	24
1712726601	Christian Bonifaz	23

Nuevo Guardar Editar Eliminar

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?

Gestión de Estudiantes - CRUD

Datos del Estudiante

ID:	<input type="text"/>
Nombres:	<input type="text"/>
Edad:	<input type="text"/>

Lista de Estudiantes

ID	Nombres	Edad
1712726601	Christian Bonifaz	23

5. 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.

6. 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.

7. 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.