



**ESPE**  
UNIVERSIDAD DE LAS FUERZAS ARMADAS  
INNOVACIÓN PARA LA EXCELENCIA

## ***Universidad de las Fuerzas Armadas - ESPE***

***Departamento de Ciencias de la Computación***

***Carrera de Ingeniería de Software***

### **CRUD de Estudiantes con Arquitectura en Capas**

***Análisis y Diseño de Software - NRC:23305***

## ***Taller 3***

***Grupo: 6***

***Integrantes:***

- *Erick Moreira*
- *Carlos Granda*
- *Kevin Coloma*

***Docente:***

***Ing. Jenny Ruiz***

***Fecha:***

***12/06/202***

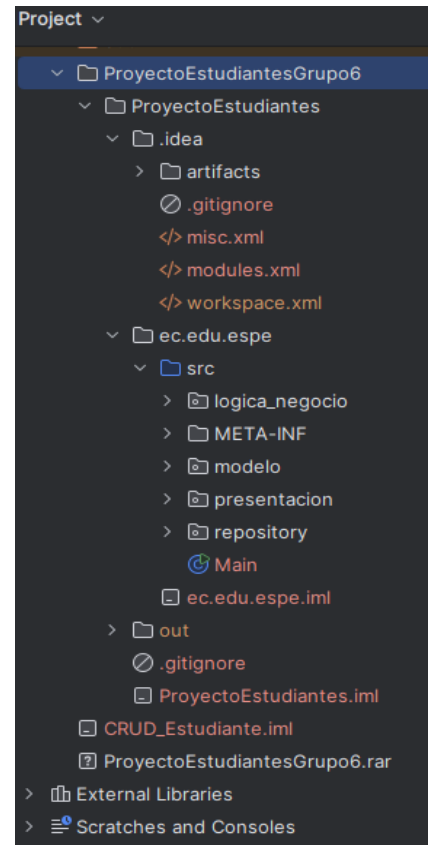
## CRUD de Estudiantes con Arquitectura en Capas

El programa desarrollado implementa una funcionalidad básica de gestión de estudiantes mediante operaciones CRUD (crear, leer, actualizar y eliminar), utilizando una arquitectura estructurada en **capas jerárquicas**. Esta separación en capas permite dividir las responsabilidades de forma clara, lo que contribuye a una mejor organización, facilidad de mantenimiento y posibilidades de escalamiento.

### Estructura general del sistema

ProyectoEstudiantes/

```
|— src/
|   |— ec.edu.espe/
|       |— Main.java          ->Clase Principal
|       |— logica_negocio/
|           |— EstudianteService.java    -> Logica de Negocio
|           |— modelo/
|               |— Estudiante.java        -> Modelo de Datos( Clase entad)
|               |— presentacion/
|                   |— EstudianteGUI.java  -> Interfaz de Usuario Grafica
|                   |— repository/
|                       |— EstudianteRepository.java    -> Acceso a datos
|                       |— ec.edu.espe.iml
|                       |— ProyectoEstudiantes.iml
|                       |— .gitignore
```



Cada una de estas capas se comunica únicamente con la capa inmediatamente inferior. La presentación conoce la lógica de negocio, y esta, a su vez, conoce el repositorio. Ninguna capa accede directamente a detalles de otras capas no vecinas, lo que favorece el desacoplamiento.

### Capa de Presentación (**presentacion**)

Contiene la clase **EstudianteGUI**, que es la interfaz gráfica de usuario (GUI) construida con Swing. Su responsabilidad es mostrar los datos, capturar entradas del usuario y notificar eventos (clics, inputs) al servicio correspondiente. No contiene lógica de negocio ni lógica de almacenamiento.

- **Colabora directamente con:** la clase **EstudianteService** (capa lógica).
- **Ejemplo:** Al pulsar “Agregar”, recoge los datos de los campos y los pasa al servicio de negocio.

## Capa de Lógica de Negocio (**logica\_negocio**)

Representada por **EstudianteService**, se encarga de coordinar las operaciones solicitadas por la capa de presentación. Aplica reglas del negocio si las hubiera (aunque en este caso son mínimas). Esta clase sirve de puente entre la interfaz y el repositorio.

- **Colabora con:** la clase **EstudianteRepository**.
- **Ejemplo:** Cuando **EstudianteGUI** llama a **agregarEstudiante**, **EstudianteService** delega esa llamada en el repositorio.

## Capa de Acceso a Datos (**repository**)

La clase **EstudianteRepository** actúa como el repositorio, almacenando los objetos **Estudiante** en una colección en memoria (**List<Estudiante>**). Esta capa encapsula toda la lógica para CRUD (agregar, obtener, actualizar, eliminar).

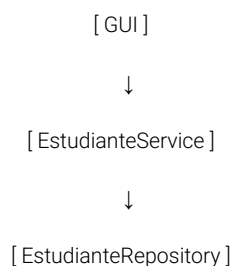
- **No conoce nada de la GUI ni de la lógica de negocio.**
- **Ejemplo:** La lógica para encontrar estudiantes por ID está encapsulada aquí.

## Modelo de Dominio (**modelo**)

La clase **Estudiante** es el modelo de datos compartido entre todas las capas. Define los atributos **id**, **nombre**, y **edad**, con sus respectivos getters y setters. Todas las capas utilizan este objeto para representar a los estudiantes.

## 2. Relación entre Capas

La comunicación es unidireccional y respeta los principios de dependencia:



La capa superior conoce a la inferior, pero no al revés. Esto asegura bajo acoplamiento y permite intercambiar implementaciones si fuera necesario (por ejemplo, cambiar **EstudianteRepository** para usar una base de datos real).

### También se aplicó Patrón Repository

La clase **EstudianteRepository** implementa claramente el patrón Repository, el cual permite abstraer el acceso a los datos y encapsular la lógica de almacenamiento. Aunque en este caso se usa una estructura de datos en memoria (una lista), este patrón facilita el reemplazo posterior del mecanismo de persistencia por una base de datos u otra fuente de datos sin modificar el resto del sistema. Este patrón resulta especialmente útil para separar la lógica de negocio de los detalles concretos del almacenamiento.

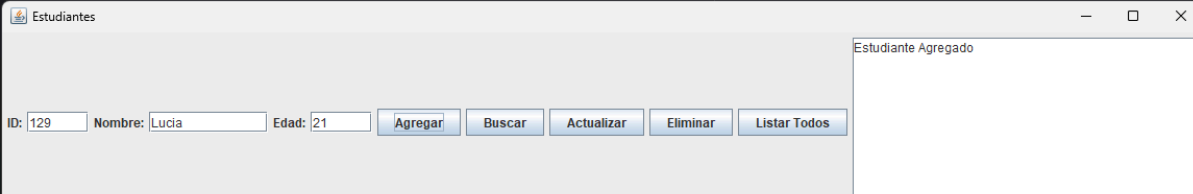
## Principios de diseño SOLID

El sistema cumple de forma general con varios de los principios SOLID:

- **Responsabilidad Única (SRP):** Cada clase tiene una función bien definida. La interfaz gráfica solo gestiona la interacción con el usuario, la clase de servicio coordina la lógica del programa y el repositorio administra el acceso a los datos.
- **Abierto/Cerrado (OCP):** El diseño permite extender funcionalidades sin modificar las clases existentes. Por ejemplo, se podría heredar o sustituir el repositorio para usar una base de datos, o extender la interfaz con nuevas opciones, sin romper el código actual.
- **Sustitución de Liskov (LSP) y Segregación de Interfaces (ISP)** aún no son relevantes debido a la ausencia de jerarquías de clases o interfaces específicas, pero podrían aplicarse si se decide introducir abstracciones más generales, como interfaces para el repositorio.
- **Inversión de Dependencias (DIP)** no se aplica completamente, ya que las dependencias entre capas están fuertemente acopladas mediante instanciación directa (**new**). Para mejorar en este aspecto, se recomienda emplear inyección de dependencias, lo que permitiría desacoplar aún más las clases y facilitar pruebas o sustituciones de componentes.

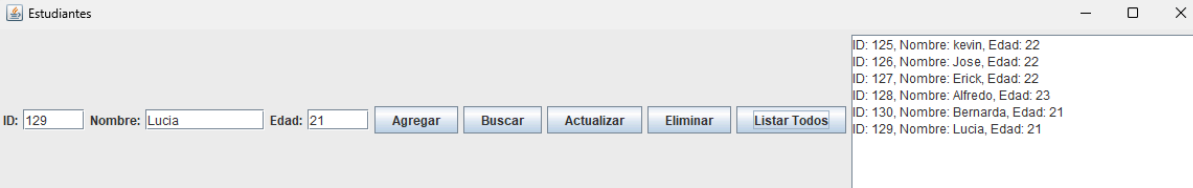
Ejecucion:

Agregar estudiante:



The screenshot shows a window titled 'Estudiantes'. On the left, there are input fields for 'ID:' (containing 129), 'Nombre:' (containing Lucia), and 'Edad:' (containing 21). Below these fields are five buttons: 'Agregar', 'Buscar', 'Actualizar', 'Eliminar', and 'Listar Todos'. On the right side of the window, there is a panel titled 'Estudiante Agregado' which is currently empty.

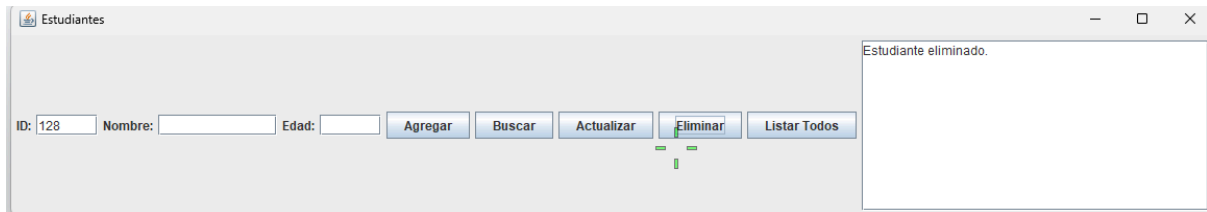
Listar estudiantes:



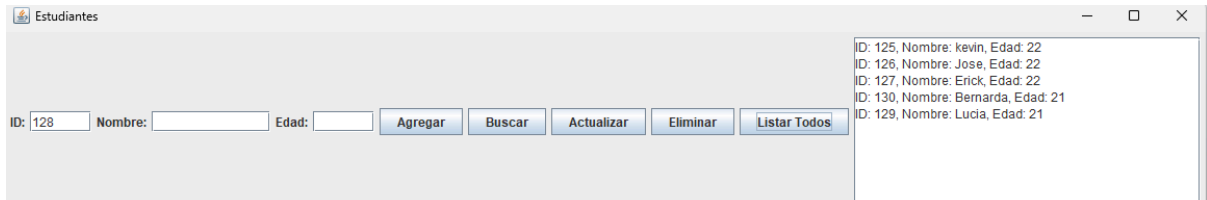
The screenshot shows the same 'Estudiantes' window. The 'ID:', 'Nombre:', and 'Edad:' fields still contain 129, Lucia, and 21 respectively. The 'Listar Todos' button is highlighted. The 'Estudiante Agregado' panel on the right now displays a list of student records:

- ID: 125, Nombre: kevin, Edad: 22
- ID: 126, Nombre: Jose, Edad: 22
- ID: 127, Nombre: Erick, Edad: 22
- ID: 128, Nombre: Alfredo, Edad: 23
- ID: 130, Nombre: Bernarda, Edad: 21
- ID: 129, Nombre: Lucia, Edad: 21

Eliminar estudiantes:



Verificar Eliminacion estudiantes:



Buscar estudiante:



## Escalabilidad y aislamiento de responsabilidades

Gracias a su diseño en capas y a la aplicación de patrones adecuados, el programa es fácilmente escalable. Se pueden incorporar nuevas funciones (como validación de datos, persistencia externa o servicios remotos) sin modificar profundamente las clases existentes. Además, las responsabilidades están bien distribuidas y aisladas, permitiendo modificar o extender una capa sin afectar directamente las demás.

Por ejemplo, si se quisiera implementar persistencia con una base de datos real, solo sería necesario modificar la clase `EstudianteRepository` (o crear una nueva implementación de una interfaz común), sin tocar la interfaz gráfica ni la lógica de negocio. Del mismo modo, si se desea reemplazar la interfaz gráfica por una API REST o una aplicación web, la lógica del programa se mantendría intacta.

## Conclusión

Como grupo, consideramos que el programa implementa de forma adecuada una arquitectura en capas, respetando principios de diseño y aplicando el patrón Repository para aislar la lógica de persistencia. Aunque se trata de una aplicación sencilla orientada al manejo de estudiantes, la estructura modular y jerarquizada representa una base sólida para la evolución del sistema hacia mayor complejidad y funcionalidad.

Cada capa cumple claramente su rol:

- La capa de presentación se encarga exclusivamente de la interacción con el usuario.
- La lógica de negocio coordina las operaciones y sirve de intermediaria entre la interfaz y el almacenamiento.

- El repositorio encapsula la persistencia de los datos, manteniéndose completamente aislado del resto de la aplicación.

Esta separación facilita no solo el desarrollo individual de cada parte, sino también la prueba, el mantenimiento y la evolución del sistema.

Para futuras versiones, proponemos como mejoras la introducción de interfaces para desacoplar aún más la lógica de negocio del repositorio, aplicar inyección de dependencias para mejorar la flexibilidad, y enriquecer la capa de lógica de negocio con validaciones, restricciones o reglas específicas del dominio.

Usar una arquitectura en capas no solo impone orden y claridad al diseño del software, sino que reduce el acoplamiento, mejora la legibilidad, y facilita la escalabilidad. No estructurar el sistema de esta manera lleva fácilmente a soluciones frágiles, difíciles de probar y costosas de mantener, especialmente cuando el sistema crece o el equipo de trabajo se amplía. En nuestro caso, este enfoque ha permitido construir una aplicación limpia, comprensible y lista para evolucionar.

Anexo Código del programa :

Main.java

```
import presentacion.EstudianteGUI;

public class Main {

    public static void main(String[] args) {

        new EstudianteGUI();

    }

}
```

EstudianteRepository.java

```
package repository;

import modelo.Estudiante;
import java.util.ArrayList;
import java.util.List;

public class EstudianteRepository {

    private final List<Estudiante> estudiantes = new ArrayList<>();

    //Metodo para agregar un objeto de tipo estudiante
```

```
public void agregar(Estudiante estudiante) {

    estudiantes.add(estudiante);

}

//Metodo para mostrar todos los estudiantes registrados

public List<Estudiante> obtenerTodos() {

    return estudiantes;

}

//Metodo para consultar estudiante por id

public Estudiante buscarPorId(int id) {

    for (Estudiante estudiante : estudiantes) {

        if (estudiante.getId() == id) {

            return estudiante;

        }

    }

    return null;

}

//Metodo para modificar un estudiante existente

public boolean actualizar(Estudiante estudiante) {

    for (int i = 0; i < estudiantes.size(); i++) {

        if (estudiantes.get(i).getId() == estudiante.getId()) {

            estudiantes.set(i, estudiante);

            return true;

        }

    }

    return false;

}

//Metodo para eliminar un estudiante

public boolean eliminar(int id) {

    return estudiantes.removeIf(estudiante -> estudiante.getId() == id);

}
```

```
}
```

## EstudianteGUI.java

```
package presentacion;

import modelo.Estudiante;

import logica_negocio.EstudianteService;

import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

public class EstudianteGUI extends JFrame {

    private final EstudianteService servicio = new EstudianteService();

    private final JTextArea areaSalida = new JTextArea(10, 30);

    public EstudianteGUI() {

        setTitle("Estudiantes");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new FlowLayout());

        //Los inputs

        JTextField campoId = new JTextField(5);

        JTextField campoNombre = new JTextField(10);

        JTextField campoEdad = new JTextField(5);

        //Botones

        JButton btnAgregar = new JButton("Agregar");

        JButton btnBuscar = new JButton("Buscar");

        JButton btnActualizar = new JButton("Actualizar");

        JButton btnEliminar = new JButton("Eliminar");

        JButton btnListar = new JButton("Listar Todos");

        //Label para los datos a ingresar

        add(new JLabel("ID:")); add(campoId);

        add(new JLabel("Nombre:")); add(campoNombre);

        add(new JLabel("Edad:")); add(campoEdad);

    }

}
```



```
        add(btnAgregar); add(btnBuscar); add(btnActualizar); add(btnEliminar);
add(btnListar);

        add(new JScrollPane(areaSalida));

        // Eventos

        btnAgregar.addActionListener((ActionEvent e) -> {

            int id = Integer.parseInt(campoId.getText());

            String nombre = campoNombre.getText();

            int edad = Integer.parseInt(campoEdad.getText());

            servicio.agregarEstudiante(new Estudiante(id, nombre, edad));

            areaSalida.setText("Estudiante Agregado");

        });

        btnBuscar.addActionListener(e -> {

            int id = Integer.parseInt(campoId.getText());

            Estudiante est = servicio.buscarEstudiante(id);

            if (est != null)

                areaSalida.setText("Encontrado: " + est.getNombre() + ", Edad: " + est.getEdad());

            else

                areaSalida.setText("Estudiante no encontrado.");

        });

        btnActualizar.addActionListener(e -> {

            int id = Integer.parseInt(campoId.getText());

            String nombre = campoNombre.getText();

            int edad = Integer.parseInt(campoEdad.getText());

            boolean ok = servicio.actualizarEstudiante(new Estudiante(id, nombre, edad));

            areaSalida.setText(ok ? "Estudiante actualizado." : "No se encontró estudiante.");

        });

        btnEliminar.addActionListener(e -> {

            int id = Integer.parseInt(campoId.getText());

            boolean ok = servicio.eliminarEstudiante(id);
```

```

        areaSalida.setText(ok ? "Estudiante eliminado." : "No se encontró
estudiante.");

    });

    btnListar.addActionListener(e -> {

        StringBuilder sb = new StringBuilder();

        for (Estudiante est : servicio.obtenerEstudiantes()) {

            sb.append("ID: ").append(est.getId()).append(", Nombre: ")

                .append(est.getNombre()).append(", Edad: ")

                .append(est.getEdad()).append("\n");

        }

        areaSalida.setText(sb.toString());

    });

    pack();

    setLocationRelativeTo(null);

    setVisible(true);

}

}

```

## Estudiante.java

```

package modelo;

public class Estudiante {

    private int id;

    private String nombre;

    private int edad;

    public Estudiante(int id, String nombre, int edad) {

        this.id = id;

        this.nombre = nombre;

        this.edad = edad;

    }

    // Getters y Setters

```

```

public int getId() {

    return id;

}

public void setId(int id) {

    this.id = id;

}

public String getNombre() {

    return nombre;

}

public void setNombre(String nombre) {

    this.nombre = nombre;

}

public int getEdad() {

    return edad;

}

public void setEdad(int edad) {

    this.edad = edad;

}

}

```

#### EstudianteService.java

```

package logica_negocio;

import modelo.Estudiante;

import repository.EstudianteRepository;

import java.util.List;

public class EstudianteService {

    private final EstudianteRepository repositorio = new
EstudianteRepository();

    //Llamada al metodo agregar estudiante

    public void agregarEstudiante(Estudiante estudiante) {

        repositorio.agregar(estudiante);

    }

}

```

```
}

//Llamada al metodo obtener estudiante

public List<Estudiante> obtenerEstudiantes() {

    return repositorio.obtenerTodos();

}

//Llamada al metodo buscar estudiante por id

public Estudiante buscarEstudiante(int id) {

    return repositorio.buscarPorId(id);

}

//Llamada al metodo actualizar estudiante

public boolean actualizarEstudiante(Estudiante estudiante) {

    return repositorio.actualizar(estudiante);

}

//Llamada al metodo eliminar estudiante

public boolean eliminarEstudiante(int id) {

    return repositorio.eliminar(id);

}
```