Universidad de las Fuerzas Armadas - ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

CRUD ESTUDIANTES EN JAVA CON PATRONES DE DISEÑO

Análisis y Diseño de Software - NRC:23305

Taller 3

Grupo: 6

Integrantes:

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

Docente:
Ing. Jenny Ruiz

Fecha:
16/06/202

CRUD de Estudiantes con Arquitectura en Capas

Este informe presenta el desarrollo de una aplicación de escritorio en Java para la gestión de estudiantes mediante operaciones CRUD (Crear, Leer, Actualizar y Eliminar), implementada con arquitectura en tres capas: presentación, lógica de negocio y datos. La interfaz gráfica fue desarrollada con Swing, ofreciendo una visualización clara y dinámica mediante una tabla interactiva.

Además, se integraron dos patrones de diseño: Flyweight, para optimizar el uso de memoria compartiendo datos repetidos, y Memento, para permitir la reversión de operaciones mediante la restauración de estados anteriores. La aplicación está estructurada para ser escalable, mantenible y fácil de extender en futuras versiones.

La aplicación implementa completamente las operaciones básicas de gestión de estudiantes, conocidas como CRUD (Create, Read, Update, Delete). Cada una se detalla a continuación:

- Agregar estudiante:
  A través de campos de entrada de texto (JTextField), el usuario introduce un ID, un nombre y una edad. Al presionar el botón “Agregar”, se crea un objeto Estudiante, el cual es almacenado en la capa de repositorio y simultáneamente se actualiza la tabla (JTable) para reflejar el nuevo registro.

- Buscar por ID:
  Mediante el botón “Buscar”, el usuario puede ingresar un ID existente. El sistema busca en el repositorio y, si encuentra un estudiante con ese ID, carga su información en los campos de entrada para facilitar su edición o verificación.

- Actualizar estudiante:
  Si un estudiante ya existe (según el ID), el usuario puede modificar sus datos y pulsar el botón “Actualizar”. Esto reemplaza el registro correspondiente en el repositorio y actualiza automáticamente la visualización en la tabla.

- Eliminar estudiante:
  Al presionar “Eliminar” con un ID válido, el sistema elimina el registro correspondiente. La tabla también se refresca al instante.

- Visualización en tabla:
  En lugar de mostrar resultados en una consola o campo de texto, la aplicación utiliza un componente JTable que se actualiza automáticamente después de cada operación. La tabla presenta las columnas ID, Nombre y Edad, y refleja en tiempo real los datos actuales del repositorio.

Integración del Patrón Flyweight

Objetivo: El patrón estructural Flyweight busca reducir el uso innecesario de memoria compartiendo instancias inmutables de objetos repetidos. En este caso, se aplica para optimizar el almacenamiento de nombres de estudiantes que son iguales.

Implementación concreta:

- Se definió una clase FlyweightFactory que mantiene un mapa (Map<String, String>) donde se almacenan las cadenas de texto utilizadas como nombres.

- Cada vez que se crea un estudiante, su atributo nombre se obtiene a través de la fábrica:
  this.nombre = FlyweightFactory.getNombre(nombre);

- Si un nombre ya ha sido usado, el mismo objeto String es retornado, evitando crear múltiples copias de la misma cadena de texto.

- Esta implementación reduce el consumo de memoria en escenarios donde muchos estudiantes comparten nombres iguales (caso común en instituciones educativas).

Integración del Patrón Memento

Objetivo: El patrón de comportamiento Memento permite restaurar el estado anterior del sistema. En este contexto, permite al usuario revertir la última operación ejecutada sobre la lista de estudiantes.

Componentes implementados:

- Memento:
  Clase que encapsula una copia profunda de la lista de estudiantes al momento de guardar un estado.

- Originator:
  Clase responsable de crear un Memento a partir del estado actual del repositorio, así como de restaurarlo a partir de un Memento previamente guardado.

- Caretaker:
  Clase que mantiene una pila (Stack) de Mementos. Cada vez que se realiza una operación que modifica la lista (agregar, eliminar, actualizar), se guarda un nuevo estado previo. Al hacer “deshacer”, se recupera el último y se restaura.

Flujo de funcionamiento:

- Antes de ejecutar cualquier operación que modifique el repositorio, se invoca el método respaldarEstado(), que guarda una copia del estado actual en la pila del Caretaker.

- Al presionar el botón “Deshacer” en la interfaz, se invoca servicio.deshacer(), que extrae el último estado guardado y reemplaza completamente la lista de estudiantes con esa copia.

- Se actualiza automáticamente la tabla de la interfaz para mostrar los datos restaurados.

Validación y pruebas:

- Se realizaron pruebas interactivas agregando varios estudiantes y utilizando el botón “Deshacer” para observar cómo los datos volvían a su estado anterior.

- Se confirmó que es posible deshacer múltiples acciones consecutivas, ya que los Mementos se almacenan en una pila.

- Si el usuario intenta deshacer más veces de las permitidas (es decir, cuando la pila está vacía), se muestra un mensaje informativo indicando que no hay más acciones por revertir.

Interfaz gráfica (Swing) Componentes clave:

- JTextField: se utilizan para capturar los valores del ID, nombre y edad del estudiante.

- JButton: cada operación del sistema tiene un botón asignado: Agregar, Buscar, Actualizar, Eliminar, Deshacer y Verificar Flyweight.

- JTable: tabla interactiva para mostrar todos los estudiantes. Se actualiza automáticamente cada vez que se modifica el repositorio.

- JScrollPane: contiene la JTable para facilitar el scroll en caso de muchos registros.

Extras implementados:

- Diseño funcional y claro, adecuado tanto para pruebas como para usuarios finales.

- Se definió correctamente la clase principal (Main.java) como punto de entrada en el archivo MANIFEST.MF.

- El archivo .jar fue generado mediante Build → Build Artifacts → Build, y se encuentra dentro de la carpeta out/artifacts.

- El programa puede ejecutarse desde consola con el siguiente comando:
  java -jar NombreDelJar.jar
- Uso de JOptionPane para mostrar mensajes informativos, errores de validación y confirmaciones visuales.

Estructura general del sistema

ProyectoEstudiantes/

src/

└── ec/edu/espe/

├── presentacion/ → Interfaz gráfica (EstudianteUI.java)

├── logica_negocio/ → Servicios y lógica de negocio

│ ├── EstudianteService.java

│ └── memento/ → Implementación del patrón Memento

│ ├── Memento.java

│ ├── Originator.java

│ └── Caretaker.java

├── datos/

│ ├── model/ → Modelo de dominio (Estudiante)

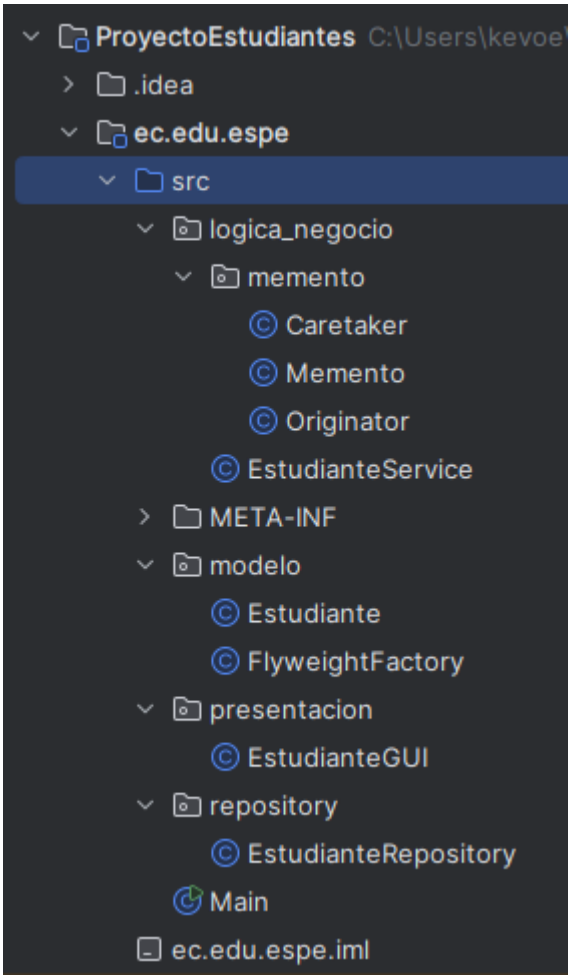
│ │ ├── Estudiante.java

│ │ └── FlyweightFactory.java

│ └── repository/

│ └── EstudianteRepository.java

└── Main.java



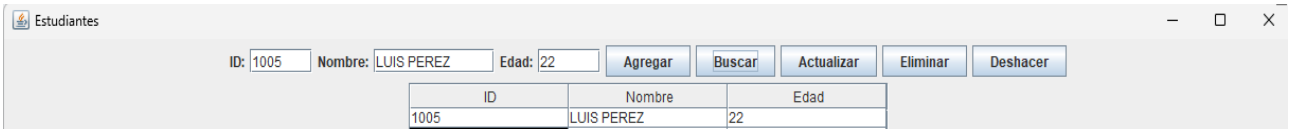
Pruebas funcionales realizadas

Durante la fase de validación se aplicaron pruebas interactivas desde la interfaz gráfica, incluyendo:

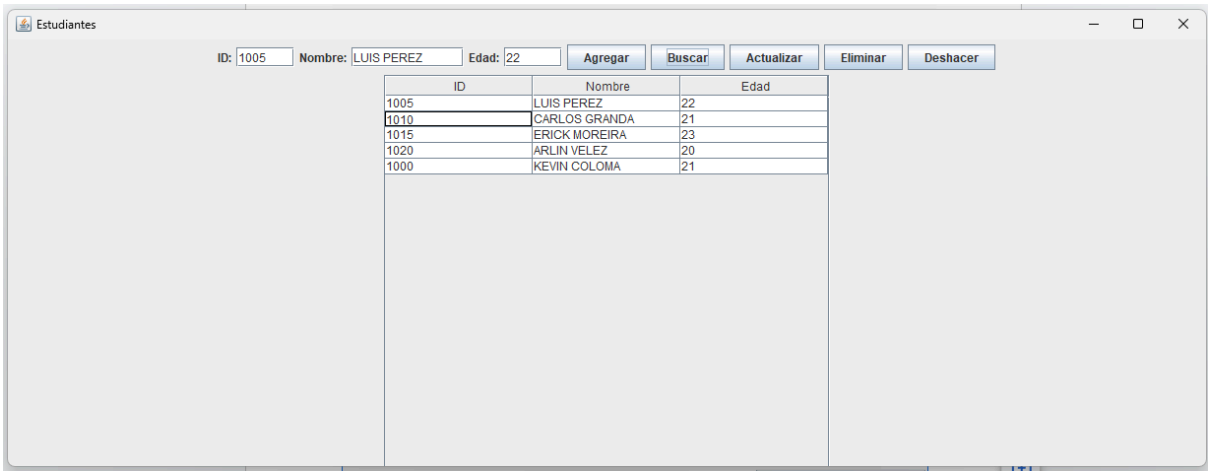
- Actualización y eliminación de registros → reversión verificada mediante “Deshacer”.
- Múltiples operaciones seguidas → deshacer múltiples veces sin errores.
- Verificación de consistencia en JTable → sincronización confirmada después de cada cambio.

Ejecución:

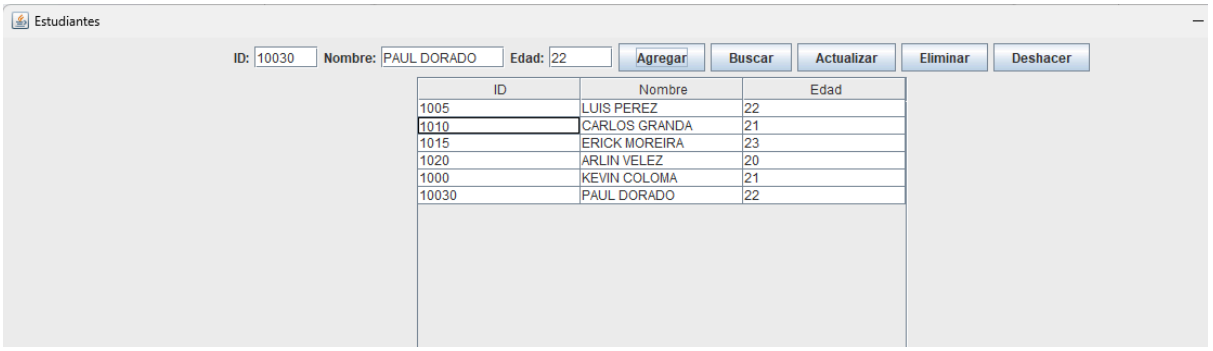
Agregar estudiante:



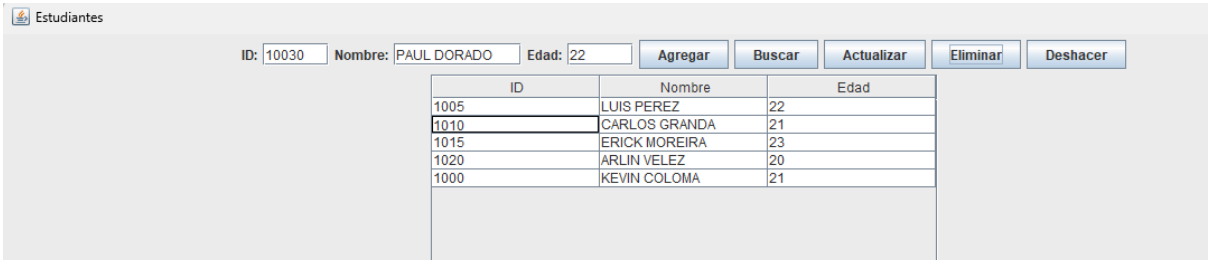
Listar estudiantes:



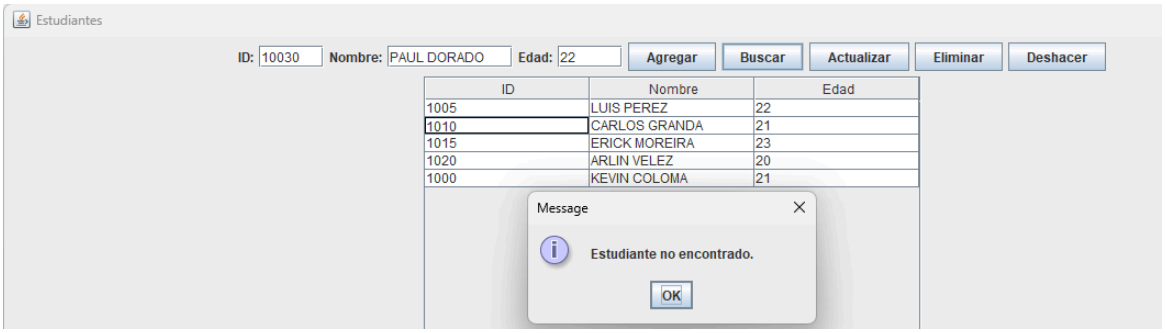
Eliminar estudiantes:



Verificar Eliminación estudiantes:



Buscar estudiante eliminado:



Escalabilidad y Aislamiento de Responsabilidades

El sistema ha sido construido sobre una arquitectura en 3 capas que promueve la escalabilidad de manera natural. Esto significa que es posible agregar nuevas funcionalidades —como búsqueda por filtros, exportación de datos o integración con bases de datos— sin alterar la lógica central. La separación entre presentación, lógica de negocio y datos permite que cada componente se expanda de forma independiente. Por ejemplo, si se desea reemplazar la interfaz Swing por una interfaz web (como Spring Boot + Thymeleaf o JavaFX), este cambio se puede realizar sin afectar el funcionamiento de la lógica o el modelo. Asimismo, la estructura permite introducir nuevas tecnologías de persistencia (como SQLite, MySQL o MongoDB) con un impacto mínimo, al mantener los detalles de almacenamiento encapsulados en la capa de repositorio.

Además, el proyecto aplica correctamente el principio de responsabilidad única (SRP), lo que facilita la mantenibilidad y el trabajo colaborativo. Cada clase y paquete cumple una función específica: la interfaz gráfica se encarga exclusivamente de interactuar con el usuario, la lógica de negocio gestiona las operaciones del sistema y los patrones de diseño, y la capa de datos administra el almacenamiento. Los patrones Flyweight y Memento están implementados en componentes separados (FlyweightFactory y memento package), lo que permite extenderlos o reutilizarlos sin interferir en otras partes del sistema. Esta clara separación de responsabilidades reduce los acoplamientos innecesarios, mejora la legibilidad del código y sienta una base sólida para el crecimiento del sistema en nuevas versiones.

Recomendaciones

Una mejora importante para el sistema es incorporar persistencia permanente, ya sea mediante bases de datos relacionales (como SQLite o MySQL) o mediante archivos serializados. Esto permitiría que los datos no se pierdan al cerrar la aplicación, convirtiéndola en una herramienta útil para entornos reales. También se recomienda extender la funcionalidad del patrón Memento implementando un historial “rehacer” (Redo), complementando así la función actual de deshacer y mejorando la experiencia de usuario. Junto con esto, el sistema podría beneficiarse de validaciones más robustas en los formularios (por ejemplo, restricción de edad, detección de campos vacíos, o uso de expresiones regulares en nombres), lo cual aseguraría integridad en los datos ingresados.

Desde el punto de vista de desarrollo, se aconseja aplicar pruebas unitarias con JUnit para verificar el correcto comportamiento de los métodos de servicio y repositorio ante distintos escenarios. También sería útil implementar un generador automático de IDs para evitar duplicidades manuales, e incluso considerar la internacionalización (i18n) del sistema para adaptarlo a múltiples idiomas. A nivel de arquitectura, si el sistema sigue creciendo, sería beneficioso incorporar el patrón Observer para desacoplar la interfaz de los eventos internos. Finalmente, se sugiere mantener el proyecto bajo control de versiones con Git, documentar los métodos usando Javadoc y aplicar principios SOLID para sostener la calidad a largo plazo. Estas recomendaciones fortalecerán la escalabilidad, la confiabilidad y el profesionalismo del sistema.

Conclusión general

El sistema de gestión de estudiantes desarrollado cumple con los requerimientos funcionales propuestos, utilizando una arquitectura en 3 capas bien definida. Además, la integración de los patrones de diseño Flyweight y Memento aporta importantes beneficios:

- Flyweight permite reducir el uso de memoria al compartir datos repetidos de forma eficiente.
- Memento introduce la capacidad de restaurar estados anteriores, ofreciendo control y seguridad ante operaciones erróneas.

Ambos patrones fueron implementados de forma adecuada y validada mediante pruebas desde la GUI, sin interferir con la lógica principal de la aplicación.

Anexos

Github:

https://github.com/KevinColoma/23305_G6_ADS/tree/951e4b37311e14ffdc1f59b7f97d247e77652c1d/Talleres/UNIDAD%202/U2T3%20TallerCrudEstudiante2.0

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<>();
```

```java
//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);
    }

    public void reemplazarTodo(List<Estudiante> nuevosEstudiantes) {
        estudiantes.clear();
        estudiantes.addAll(nuevosEstudiantes);
    }

}
```

EstudianteGUI.java

```
package presentacion;


import modelo.Estudiante;
import logica_negocio.EstudianteService;


import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.ActionEvent;


public class EstudianteGUI extends JFrame {
    private final EstudianteService servicio = new EstudianteService();
     private final DefaultTableModel modeloTabla = new DefaultTableModel(new
String[]{"ID", "Nombre", "Edad"}, 0);
    private final JTable tablaEstudiantes = new JTable(modeloTabla);
```

```
public EstudianteGUI() {
    setTitle("Estudiantes");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new FlowLayout());

    JTextField campoId = new JTextField(5);
    JTextField campoNombre = new JTextField(10);
    JTextField campoEdad = new JTextField(5);

    JButton btnAgregar = new JButton("Agregar");
    JButton btnBuscar = new JButton("Buscar");
    JButton btnActualizar = new JButton("Actualizar");
    JButton btnEliminar = new JButton("Eliminar");



    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);
    JButton btnDeshacer = new JButton("Deshacer");
    btnDeshacer.addActionListener(e -> {
        servicio.deshacer();
        actualizarTabla();
    });
    add(btnDeshacer);

    add(new JScrollPane(tablaEstudiantes));

    btnAgregar.addActionListener(e -> {
```

```
            int id = Integer.parseInt(campoId.getText());
            String nombre = campoNombre.getText();
            int edad = Integer.parseInt(campoEdad.getText());
            servicio.agregarEstudiante(new Estudiante(id, nombre, edad));
            actualizarTabla();
        });

        btnBuscar.addActionListener(e -> {
            int id = Integer.parseInt(campoId.getText());
            Estudiante est = servicio.buscarEstudiante(id);
            if (est != null) {
                campoNombre.setText(est.getNombre());
                campoEdad.setText(String.valueOf(est.getEdad()));
            } else {
                        JOptionPane.showMessageDialog(this, "Estudiante no
encontrado.");
            }
        });

        btnActualizar.addActionListener(e -> {
            int id = Integer.parseInt(campoId.getText());
            String nombre = campoNombre.getText();
            int edad = Integer.parseInt(campoEdad.getText());
                 if (servicio.actualizarEstudiante(new Estudiante(id, nombre,
edad))) {
                actualizarTabla();
            } else {
                        JOptionPane.showMessageDialog(this, "Estudiante no
encontrado.");
            }
        });
```

```
        btnEliminar.addActionListener(e -> {
            int id = Integer.parseInt(campoId.getText());
            if (servicio.eliminarEstudiante(id)) {
                actualizarTabla();
            } else {
                            JOptionPane.showMessageDialog(this, "Estudiante no
encontrado.");
            }
        });



        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    private void actualizarTabla() {
        modeloTabla.setRowCount(0);
        for (Estudiante est : servicio.obtenerEstudiantes()) {
                modeloTabla.addRow(new Object[]{est.getId(), est.getNombre(),
est.getEdad()});
        }
    }
}
```

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 = FlyweightFactory.getNombre(nombre); // Aplicación del
patrón
        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 logica_negocio.memento.Caretaker;
import logica_negocio.memento.Memento;
import logica_negocio.memento.Originator;
import modelo.Estudiante;
import repository.EstudianteRepository;


import java.util.ArrayList;
import java.util.List;


public class EstudianteService {
    private final Originator originator = new Originator();
    private final Caretaker caretaker = new Caretaker();
            private final EstudianteRepository repositorio = new
EstudianteRepository();


    private void respaldarEstado() {
        originator.setEstado(new ArrayList<>(repositorio.obtenerTodos()));
        caretaker.guardar(originator.guardar());
    }


    //Llamada al metodo agregar estudiante
    public void agregarEstudiante(Estudiante estudiante) {
        respaldarEstado(); // Guardar antes de modificar
        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) {
    respaldarEstado();
    return repositorio.eliminar(id);
}

public boolean deshacer() {
    Memento memento = caretaker.deshacer();
    if (memento != null) {
        List<Estudiante> restaurado = originator.restaurar(memento);
        repositorio.reemplazarTodo(restaurado);
        return true;
    }
```

```
        return false;
    }
}
```

FlyweightFactory.java

```
package modelo;

import java.util.HashMap;
import java.util.Map;


public class FlyweightFactory {
    private static final Map<String, String> nombres = new HashMap<>();


    public static String getNombre(String nombre) {
        return nombres.computeIfAbsent(nombre, k -> k);
    }
}
```

Originator.java

```
package logica_negocio.memento;

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



public class Originator {
    private List<Estudiante> estado;


    public void setEstado(List<Estudiante> estado) {
        this.estado = estado;
    }
```

```
    public Memento guardar() {
        return new Memento(estado);
    }

    public List<Estudiante> restaurar(Memento memento) {
        return memento.getEstado();
    }
}
```

Memento.java

```
package logica_negocio.memento;

import modelo.Estudiante;

import java.util.ArrayList;

import java.util.List;


public class Memento {

    private final List<Estudiante> estado;


    public Memento(List<Estudiante> estado) {

        // Copia profunda

        this.estado = new ArrayList<>();

        for (Estudiante e : estado) {

                    this.estado.add(new  Estudiante(e.getId(),  e.getNombre(),
e.getEdad()));

        }

    }


    public List<Estudiante> getEstado() {

        return estado;
```

```
    }
}
```

Caretaker.java

```
package logica_negocio.memento;

import java.util.Stack;



public class Caretaker {

    private final Stack<Memento> historial = new Stack<>();


    public void guardar(Memento memento) {

        historial.push(memento);

    }


    public Memento deshacer() {

        return historial.isEmpty() ? null : historial.pop();

    }

}
```