



**Universidad de las Fuerzas Armadas**

**ESPE Departamento de Ciencias de la Computación**

**Análisis y Diseño**

**Integrantes:** Isaac Escobar, Eduardo Mortensen, Diego Ponce

**NRC:** 14571

## 1. OBJETIVO:

Examinar a fondo la implementación y aplicación del patrón de diseño Composite dentro del sistema de gestión de estudiantes, con el fin de comprender su utilidad estructural, los beneficios que aporta al algoritmo y su impacto positivo en la organización, extensibilidad y mantenimiento del código. Este análisis busca justificar por qué este patrón es adecuado para representar relaciones jerárquicas entre estudiantes y cómo contribuye a una arquitectura limpia, flexible y escalable.

## 2. ANÁLISIS DE LA ARQUITECTURA Y PATRONES DE DISEÑO:

El patrón de diseño Composite permite estructurar objetos en árboles jerárquicos, donde tanto los elementos individuales como las composiciones (conjuntos de elementos) son tratados de forma uniforme. Este patrón se aplica cuando se desea manipular colecciones de objetos con una interfaz común, permitiendo operaciones recursivas sin necesidad de distinguir entre objetos simples y compuestos. En el contexto del sistema de estudiantes, el uso de Composite facilita la agrupación de *EstudianteHoja* (individuos) y *GrupoEstudiante* (colecciones) bajo una interfaz común (*ComponenteEstudiante*), lo que permite realizar operaciones como mostrar, contar o procesar todos los elementos sin conocer la estructura interna. Esto mejora la extensibilidad del sistema, ya que se pueden agregar nuevas operaciones o estructuras jerárquicas sin alterar las clases existentes.

La ventaja principal del patrón Composite en este proyecto es que abstrae el manejo de jerarquías complejas con una interfaz unificada, promoviendo el principio de “cerrado para modificación, abierto para extensión”. Desde el punto de vista del algoritmo, el patrón reduce la necesidad de condicionales o verificaciones de tipo, reemplazándolas por polimorfismo. Además, permite realizar recorridos recursivos simples para operaciones como impresión, exportación o análisis de grupos de estudiantes. Su integración no solo mejora la legibilidad del código, sino que también incrementa la capacidad del sistema para representar relaciones reales como cursos, paralelos o niveles educativos, manteniendo la lógica de negocio intacta y limpia.

## 3. ESTRUCTURA DEL ALGORITMO:

### 3.1 ESTRUCTURA 3 CAPAS TRADICIONAL (original)

```
ProyectoEstudiantes/  
├── modelo/  
│   └── Estudiante.java  
├── repositorio/  
│   └── EstudianteRepositorio.java  
├── servicio/  
│   └── EstudianteServicio.java  
└── presentacion/  
    ├── EstudianteUI.java  
    └── Main.java
```

### 3.2 ESTRUCTURA EXTENDIDA CON PATRONES DE DISEÑO (Grupo 3)

```
ProyectoEstudiantes/  
├── modelo/  
│   ├── Estudiante.java  
│   ├── ComponenteEstudiante.java ← Interfaz Composite  
│   ├── EstudianteHoja.java ← Hoja  
│   └── GrupoEstudiante.java ← Nodo compuesto  
├── repositorio/  
│   └── EstudianteRepositorio.java  
├── servicio/  
│   └── EstudianteServicio.java  
├── presentacion/  
│   ├── EstudianteUI.java ← GUI Swing  
│   ├── Main.java  
│   ├── Comando.java ← Interfaz Interpreter  
│   ├── ComandoAgregar.java ← Comando concreto  
│   └── ComandoEliminar.java ← Comando concreto
```

### 3.3 EXPLICACIÓN DE LA ESTRUCTURA:

Claro, aquí tienes una versión más concisa pero aún técnica y clara del análisis por paquetes y clases:

#### **modelo/**

- **Estudiante.java**  
Clase POJO que representa la entidad estudiante con los atributos orden, nombre y edad. No contiene lógica de negocio. Su única responsabilidad es encapsular los datos y proporcionar acceso mediante getters y setters.
- **ComponenteEstudiante.java**  
Interfaz base del patrón Composite. Declara el método `mostrar()`, que permite tratar de forma uniforme tanto a estudiantes individuales como a grupos compuestos.
- **EstudianteHoja.java**  
Clase hoja del patrón Composite. Implementa `ComponenteEstudiante` y contiene un `Estudiante`. Su método `mostrar()` imprime la información del estudiante individual.
- **GrupoEstudiante.java**  
Clase compuesta del patrón Composite. Contiene una lista de `ComponenteEstudiante`, lo que le permite agrupar estudiantes y subgrupos. Su método `mostrar()` itera recursivamente sobre sus hijos.

### ¿Por qué Composite aquí?

Permite estructurar jerárquicamente estudiantes (por cursos, niveles, etc.) con una interfaz común. Facilita recorridos recursivos y mantiene el código flexible y extensible.

#### repositorio/

- EstudianteRepositorio.java  
Clase que simula un repositorio en memoria. Administra una lista de estudiantes y proporciona métodos CRUD (agregar, listar, buscar, actualizar, eliminar). No aplica patrones adicionales.

#### Propósito:

Aislar el almacenamiento de datos y facilitar su futura sustitución (por ejemplo, por una base de datos real).

#### servicio/

- EstudianteServicio.java  
Contiene la lógica de negocio. Conecta la UI con el repositorio. Implementa validaciones mínimas y delega operaciones CRUD al repositorio.

#### Beneficio:

Separa la lógica del sistema de la UI y del almacenamiento, facilitando el mantenimiento y la escalabilidad.

#### presentacion/

- EstudianteUI.java  
Interfaz gráfica implementada con Swing. Contiene campos de entrada, botones y tabla. Cada evento ejecuta una operación sobre EstudianteServicio.
- Main.java  
Clase principal que lanza la aplicación mediante `new EstudianteUI()`.

#### Uso del patrón Interpreter:

- Comando.java  
Interfaz que define el método `interpretar()`. Cada comando textual lo implementa para ejecutar una acción.
- ComandoAgregar.java  
Interpreta entradas tipo `agregar 1001 Juan 20` y llama al servicio para registrar el estudiante.

- ComandoEliminar.java  
Interpreta entradas tipo eliminar 1001 y solicita su eliminación al servicio.

### ¿Por qué Interpreter aquí?

Permite ejecutar comandos escritos en texto, ideal para una interfaz alternativa tipo consola. Facilita extensibilidad: para nuevos comandos basta implementar una nueva clase. Mejora la reutilización y permite automatizar acciones.

## 3.4 COMPARACIÓN Y BENEFICIOS

Aspecto	Arquitectura 3 capas	Arquitectura con patrones
Complejidad	Baja	Media (por introducción de patrones)
Mantenibilidad	Alta	Muy alta (alta cohesión)
Extensibilidad	Limitada	Alta (fácil agregar comandos, grupos)
Uso de Composite	No	Sí (estructura jerárquica)
Uso de Interpreter	No	Sí (procesamiento por texto)
Reutilización	Media	Alta
Pruebas automatizadas	Parciales	Mejores pruebas por comandos

## 4. IMPLEMENTACIÓN DEL CÓDIGO

Se desarrolló una aplicación en Java utilizando el patrón de arquitectura por capas (modelo-vista-servicio-repositorio), aplicando los patrones de diseño Composite e Interpreter para enriquecer la solución inicial simple.

- El patrón Composite permite estructurar estudiantes de manera jerárquica, agrupando múltiples estudiantes en una entidad lógica llamada GrupoEstudiante.
- El patrón Interpreter permite ejecutar comandos desde una entrada textual en la GUI, como agregar o eliminar estudiantes mediante frases tipo “agregar 1005 Maria 19”.

Ambos patrones se implementan dentro de la lógica de la aplicación para mejorar su extensibilidad, legibilidad y escalabilidad.

## 5. CÓDIGO

### **Estudiante.java**

```
package modelo;

public class Estudiante {
    private int orden;
    private String nombre;
    private int edad;

    public Estudiante(int orden, String nombre, int edad) {
        this.orden = orden;
        this.nombre = nombre;
        this.edad = edad;
    }

    public int getOrden() { return orden; }
    public void setOrden(int orden) { this.orden = orden; }

    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; }

    @Override
    public String toString() {
        return "Estudiante{" + "orden=" + orden + ", nombre=" + nombre + "\", edad=" +
        edad + "}";
    }
}
```

### **EstudianteRepositorio.java**

```
package repositorio;

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

public class EstudianteRepositorio {
    private final List<Estudiante> estudiantes = new ArrayList<>();

    public void agregar(Estudiante e) {
```

```

        estudiantes.add(e);
    }

    public List<Estudiante> listar() {
        return estudiantes;
    }

    public Estudiante buscarPorOrden(int orden) {
        return estudiantes.stream().filter(e -> e.getOrden() == orden).findFirst().orElse(null);
    }

    public boolean actualizar(Estudiante actualizado) {
        for (int i = 0; i < estudiantes.size(); i++) {
            if (estudiantes.get(i).getOrden() == actualizado.getOrden()) {
                estudiantes.set(i, actualizado);
                return true;
            }
        }
        return false;
    }

    public boolean eliminar(int orden) {
        return estudiantes.removeIf(e -> e.getOrden() == orden);
    }
}

```

### **EstudianteServicio.java**

```

package servicio;

import modelo.Estudiante;
import repositorio.EstudianteRepositorio;

import java.util.List;

public class EstudianteServicio {
    private final EstudianteRepositorio repo = new EstudianteRepositorio();

    public void agregarEstudiante(Estudiante e) {
        repo.agregar(e);
    }

    public List<Estudiante> obtenerEstudiantes() {
        return repo.listar();
    }

    public Estudiante obtenerPorOrden(int orden) {
        return repo.buscarPorOrden(orden);
    }
}

```

```

    public boolean modificarEstudiante(Estudiante e) {
        return repo.actualizar(e);
    }

    public boolean eliminarEstudiante(int orden) {
        return repo.eliminar(orden);
    }
}

```

### **Main.java**

```

package presentacion;

public class Main {
    public static void main(String[] args) {
        new EstudianteUI();
    }
}

```

### **EstudianteUI.java**

```

package presentacion;

import modelo.Estudiante;
import servicio.EstudianteServicio;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class EstudianteUI extends JFrame {
    private JTextField txtId, txtNombre, txtEdad;
    private JTable tabla;
    private DefaultTableModel modeloTabla;
    private EstudianteServicio servicio;

    public EstudianteUI() {
        servicio = new EstudianteServicio();

        setTitle("CRUD Estudiantes");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);
        setSize(600, 420);
        setLocationRelativeTo(null);

        JLabel lblId = new JLabel("ID:");
        lblId.setBounds(20, 20, 50, 25);
        add(lblId);
    }
}

```



```
txtId = new JTextField();  
txtId.setBounds(60, 20, 100, 25);  
add(txtId);
```

```
JLabel lblNombre = new JLabel("Nombre:");  
lblNombre.setBounds(180, 20, 70, 25);  
add(lblNombre);
```

```
txtNombre = new JTextField();  
txtNombre.setBounds(240, 20, 150, 25);  
add(txtNombre);
```

```
JLabel lblEdad = new JLabel("Edad:");  
lblEdad.setBounds(410, 20, 50, 25);  
add(lblEdad);
```

```
txtEdad = new JTextField();  
txtEdad.setBounds(460, 20, 50, 25);  
add(txtEdad);
```

```
JButton btnAgregar = new JButton("Agregar");  
btnAgregar.setBounds(20, 60, 100, 25);  
add(btnAgregar);
```

```
JButton btnActualizar = new JButton("Actualizar");  
btnActualizar.setBounds(130, 60, 100, 25);  
add(btnActualizar);
```

```
JButton btnEliminar = new JButton("Eliminar");  
btnEliminar.setBounds(240, 60, 100, 25);  
add(btnEliminar);
```

```
JButton btnMostrar = new JButton("Mostrar Todo");  
btnMostrar.setBounds(350, 60, 160, 25);  
add(btnMostrar);
```

```
JButton btnBuscar = new JButton("Buscar");  
btnBuscar.setBounds(240, 330, 100, 25);  
add(btnBuscar);
```

```
modeloTabla = new DefaultTableModel(new Object[]{"ID", "Nombre", "Edad"}, 0);  
tabla = new JTable(modeloTabla);  
JScrollPane scrollPane = new JScrollPane(tabla);  
scrollPane.setBounds(20, 100, 540, 220);  
add(scrollPane);
```

```
btnAgregar.addActionListener(e -> {
```

```

String idText = txtId.getText().trim();
String nombre = txtNombre.getText().trim();
String edadText = txtEdad.getText().trim();

if (idText.isEmpty() || nombre.isEmpty() || edadText.isEmpty()) {
    JOptionPane.showMessageDialog(this, "Por favor complete todos los campos.");
    return;
}

try {
    int id = Integer.parseInt(idText);
    int edad = Integer.parseInt(edadText);
    servicio.agregarEstudiante(new Estudiante(id, nombre, edad));
    limpiarCampos();
} catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(this, "ID y Edad deben ser números válidos.");
}
});

btnActualizar.addActionListener(e -> {
    String idText = txtId.getText().trim();
    String nombre = txtNombre.getText().trim();
    String edadText = txtEdad.getText().trim();

    if (idText.isEmpty() || nombre.isEmpty() || edadText.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Por favor complete todos los campos.");
        return;
    }

    try {
        int id = Integer.parseInt(idText);
        int edad = Integer.parseInt(edadText);
        boolean ok = servicio.modificarEstudiante(new Estudiante(id, nombre, edad));
        JOptionPane.showMessageDialog(this, ok ? "Actualizado" : "No encontrado");
        limpiarCampos();
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "ID y Edad deben ser números válidos.");
    }
});

btnEliminar.addActionListener(e -> {
    String idText = txtId.getText().trim();
    if (idText.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Por favor ingrese el ID a eliminar.");
        return;
    }

    try {

```

```

        int id = Integer.parseInt(idText);
        boolean ok = servicio.eliminarEstudiante(id);
        JOptionPane.showMessageDialog(this, ok ? "Eliminado" : "No encontrado");
        limpiarCampos();
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "ID debe ser un número válido.");
    }
});

btnMostrar.addActionListener(e -> {
    modeloTabla.setRowCount(0);
    for (Estudiante est : servicio.obtenerEstudiantes()) {
        modeloTabla.addRow(new Object[]{est.getOrden(), est.getNombre(),
est.getEdad()});
    }
});

btnBuscar.addActionListener(e -> {
    String idText = txtId.getText().trim();
    if (idText.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Por favor ingrese el ID a buscar.");
        return;
    }

    try {
        int id = Integer.parseInt(idText);
        Estudiante buscado = servicio.obtenerPorOrden(id);
        modeloTabla.setRowCount(0);
        if (buscado != null) {
            modeloTabla.addRow(new Object[]{buscado.getOrden(), buscado.getNombre(),
buscado.getEdad()});
        } else {
            JOptionPane.showMessageDialog(this, "Estudiante no encontrado");
        }
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "ID debe ser un número válido.");
    }
});

setVisible(true);
}

private void limpiarCampos() {
    txtId.setText("");
    txtNombre.setText("");
    txtEdad.setText("");
}
}

```

## 6. Requisitos Funcionales

Requisito	Descripción	¿Dónde se cumple en tu proyecto?
RF1	Agregar un nuevo estudiante (orden, nombre, edad).	En EstudianteUI.java, dentro del listener del botón <b>Agregar</b> (btnAgregar).
RF2	Mostrar todos los estudiantes registrados.	En EstudianteUI.java, botón <b>Mostrar Todo</b> (btnMostrar) muestra en JTable.
RF3	Consultar un estudiante por su número de orden.	En EstudianteUI.java, botón <b>Buscar</b> (btnBuscar) usa servicio.obtenerPorOrden(...).
RF4	Modificar los datos de un estudiante existente.	En EstudianteUI.java, botón <b>Actualizar</b> (btnActualizar).
RF5	Eliminar un estudiante por su número de orden.	En EstudianteUI.java, botón <b>Eliminar</b> (btnEliminar).
RF6	Aplicación organizada en tres capas: modelo, repositorio, servicio, presentación.	Cumplido completamente — ver detalle abajo.

## 7. ARQUITECTURA EN 3 CAPAS

Capa	Paquete / Clase	Responsabilidad principal
Modelo	modelo/Estudiante.java	Define la estructura del objeto Estudiante (atributos y métodos).
Repositorio	repositorio/EstudianteRepositorio.java	Contiene la lógica de acceso a datos en memoria (CRUD directo).
Servicio	servicio/EstudianteServicio.java	Encapsula la lógica de negocio, intermedia entre la GUI y el repositorio.
Presentación	presentacion/EstudianteUI.java	Contiene toda la lógica de la interfaz Swing (botones, tabla, inputs).
Principal	presentacion/Main.java	Solo inicia la UI (new EstudianteUI()).

## 8. Aplicación de los patrones Composite y Interpreter al CRUD de Estudiantes

### 8.1 Composite en el CRUD

Objetivo del patrón Composite: tratar objetos individuales y compuestos de la misma forma.

Aplicación: Agrupa estudiantes por categorías como si fueran "nodos" de una estructura jerárquica.

Implementación:

1. Crear una clase **GrupoEstudiante** que contenga estudiantes individuales o grupos de estudiantes.
2. Así puedes llamar **.mostrar()** a un grupo y se mostrarán todos sus miembros (estudiantes u otros grupos).

```
package modelo;
```

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

```
public interface ComponenteEstudiante {  
    void mostrar();  
}
```

```
// Hoja: estudiante individual
```

```
class EstudianteHoja implements ComponenteEstudiante {  
    private Estudiante estudiante;
```

```
    public EstudianteHoja(Estudiante estudiante) {  
        this.estudiante = estudiante;  
    }
```

```
    @Override  
    public void mostrar() {  
        System.out.println("Estudiante: " + estudiante);  
    }  
}
```

```
// Compuesto: grupo de estudiantes
```

```
class GrupoEstudiante implements ComponenteEstudiante {  
    private String nombreGrupo;  
    private List<ComponenteEstudiante> componentes = new ArrayList<>();
```

```
    public GrupoEstudiante(String nombreGrupo) {  
        this.nombreGrupo = nombreGrupo;  
    }
```

```
    public void agregar(ComponenteEstudiante c) {  
        componentes.add(c);  
    }
```

```
    @Override  
    public void mostrar() {
```

```

        System.out.println("Grupo: " + nombreGrupo);
        for (ComponenteEstudiante c : componentes) {
            c.mostrar();
        }
    }
}

```

## 8.2. Interpreter en el CRUD

**Objetivo del patrón Interpreter:** interpretar comandos simples como si fueran un lenguaje.

**Aplicación:** permitir ingresar comandos de texto en un **JTextField** como:

- **agregar 1001 Juan 21**
- **eliminar 1001**

**Implementación:**

```
package presentacion;
```

```
import modelo.Estudiante;
```

```
import servicio.EstudianteServicio;
```

```
interface Comando {
    void interpretar(String[] tokens, EstudianteServicio servicio);
}

```

```
class ComandoAgregar implements Comando {
    public void interpretar(String[] tokens, EstudianteServicio servicio) {
        int id = Integer.parseInt(tokens[1]);
        String nombre = tokens[2];
        int edad = Integer.parseInt(tokens[3]);
        servicio.agregarEstudiante(new Estudiante(id, nombre, edad));
    }
}

```

```
class ComandoEliminar implements Comando {
    public void interpretar(String[] tokens, EstudianteServicio servicio) {
        int id = Integer.parseInt(tokens[1]);
        servicio.eliminarEstudiante(id);
    }
}

```

## 9. CONCLUSIONES

- La implementación de los patrones de diseño Composite e Interpreter permite enriquecer la arquitectura del sistema original, otorgando mayor escalabilidad y reutilización del código.
- Composite permite trabajar con grupos de estudiantes como si fueran un solo objeto, útil para operaciones globales (mostrar todos, agrupar por nivel, etc).
- Interpreter introduce una interfaz flexible y extensible para ejecutar comandos, ideal para tareas automatizadas o interfaces tipo consola.

## **10. RECOMENDACIONES**

- Siempre que se requiera manejar estructuras que pueden contener elementos del mismo tipo (como carpetas, menús, grupos), se recomienda aplicar el patrón Composite para mantener una arquitectura extensible y coherente.
- Cuando se desee interpretar comandos personalizados, configuraciones o instrucciones en lenguaje textual dentro del sistema, es recomendable usar el patrón Interpreter, ya que separa la lógica de comandos del resto de la aplicación.
- Al implementar patrones de diseño, es importante asegurarse de que estén correctamente encapsulados en su propia capa o paquete, para mantener una arquitectura limpia y comprensible, y facilitar las pruebas y el mantenimiento.