Universidad de las Fuerzas Armadas - ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

Análisis y Diseño de Software - NRC:22426

Trabajo:

U2T3
Patrón de diseño estructural: Decorator
Patrón de diseño de comportamiento: Iterator

Grupo: 4

Integrantes:

Diego Casignia

Anthony Villarreal

Javier Ramos

Profesora: Ing. Jenny Ruiz

Objetivo

Desarrollar una aplicación de escritorio para la gestión de estudiantes, utilizando una arquitectura organizada que permita:

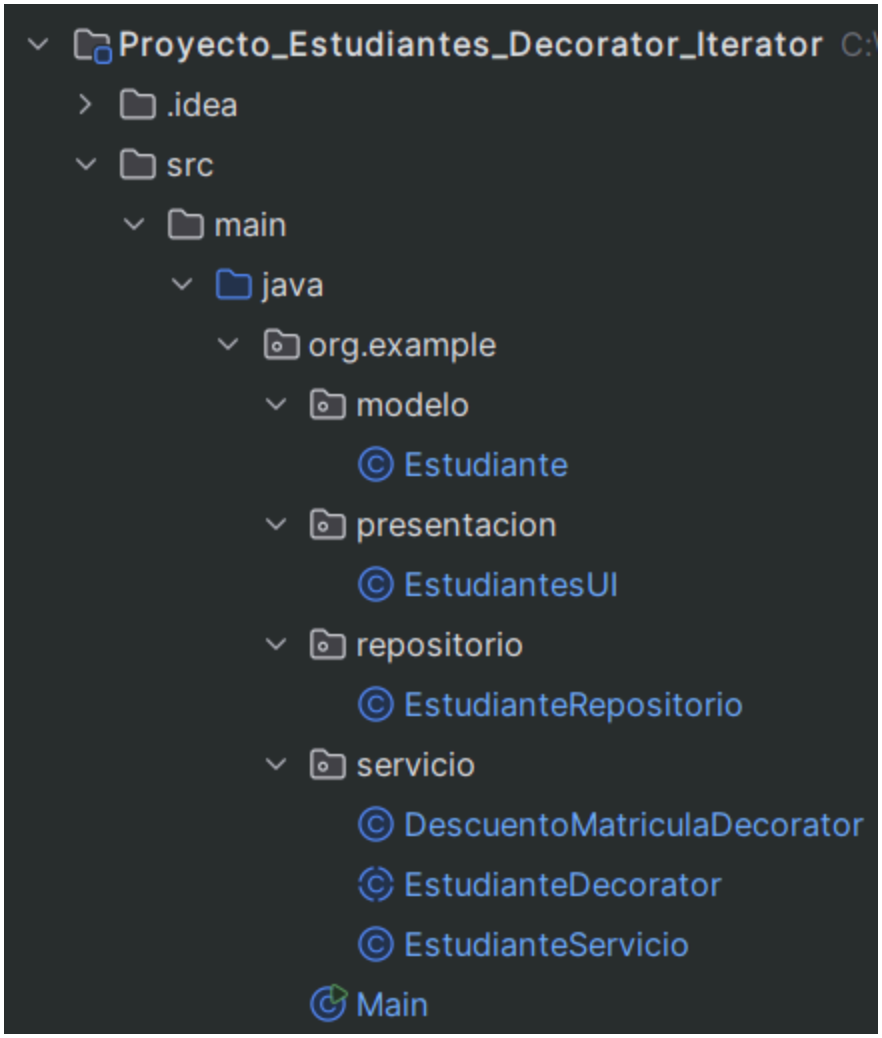
- Agregar, actualizar, eliminar y visualizar estudiantes.
- Aplicar un descuento decorativo a la matrícula de estudiantes menores de 20 años.
- Separar claramente responsabilidades para facilitar el mantenimiento y escalabilidad del sistema.

Paso por paso de cómo se realizó

- Definir la entidad principal (modelo):
    - Crear la clase Estudiante que representa los datos básicos.
- Diseñar la capa de repositorio:
    - Crear EstudianteRepositorio para manejar la lista de estudiantes (simulando persistencia en memoria).
- Crear la capa de servicio:
    - Desarrollar EstudianteServicio para aplicar lógica de negocio y actuar como puente entre presentación y repositorio.
- Implementar el patrón Decorator:
    - Crear EstudianteDecorator y DescuentoMatriculaDecorator para añadir dinámicamente un descuento.
- Construir la interfaz gráfica (presentación):
    - Desarrollar EstudiantesUI con Swing, conectada al servicio para manejar eventos de usuario.
- Crear clase Main:
    - Lanza la interfaz dentro del hilo principal.

Nombre del proyecto: ProyectoEstudiantesDecoratorIterator

Estructura.



Definición de cada capa

Modelo (Entidad) Representa la estructura de datos del dominio.

Repositorio (Datos) Encargada de almacenar, actualizar y eliminar los datos.

Servicio (Negocio) Contiene la lógica de negocio. Aplica reglas, validaciones y maneja decoradores.

Presentación (UI) Interfaz visual. Interactúa con el usuario final y conecta con el servicio.

Función de cada clase

- **DescuentoMatriculaDecorator.java**

```
package org.example.servicio;

import org.example.modelo.Estudiante;
```

```
/**
 * Clase que implementa el patrón Decorator para agregar un descuento a un estudiante.
 * Extiende EstudianteDecorator para heredar la funcionalidad base y añade información
 * sobre un porcentaje de descuento en la matrícula.
 */
public class DescuentoMatriculaDecorator extends EstudianteDecorator {
    // Porcentaje de descuento aplicado al estudiante
    private final double porcentaje;

    /**
     * Constructor que inicializa el decorador con un estudiante base y un porcentaje de descuento.
     * @param estudiante El estudiante al que se le aplica el descuento.
     * @param porcentaje El porcentaje de descuento a aplicar.
     */
    public DescuentoMatriculaDecorator(Estudiante estudiante, double porcentaje) {
        super(estudiante);
        this.porcentaje = porcentaje;
    }

    /**
     * Sobrescribe el método getInfo para incluir información del estudiante base
     * y el porcentaje de descuento aplicado.
     * @return Una cadena con la información del estudiante y el descuento.
     */
    @Override
    public String getInfo() {
        return estudianteBase.getInfo() + String.format(" | Descuento: %.0f%%", porcentaje);
    }
}
```

Explicación: La clase DescuentoMatriculaDecorator implementa el patrón Decorator, que permite añadir funcionalidades adicionales a un objeto de forma dinámica sin modificar su código base. Esta clase extiende EstudianteDecorator y envuelve un objeto Estudiante para añadir un porcentaje de descuento (por ejemplo, 15%) a su información. En tiempo de ejecución, cuando se crea un estudiante con edad menor a 20 en EstudiantesUI, se envuelve con esta clase, y su método getInfo() agrega el texto del descuento al resultado del estudiante base. El patrón Decorator se utiliza aquí para extender la funcionalidad de Estudiante de manera flexible, permitiendo que otros decoradores puedan añadirse en el futuro sin alterar la lógica existente.

- **Estudiante.java**

```
package org.example.modelo;

/**
 * Clase base que representa a un estudiante con atributos básicos.
 * Proporciona métodos para acceder y modificar los datos del estudiante.
 */
public class Estudiante {
    // Identificador único del estudiante
    protected int id;
    // Nombre del estudiante
    protected String nombre;
    // Edad del estudiante
    protected int edad;

    /**
     * Constructor que inicializa un estudiante con su ID, nombre y edad.
     * @param id Identificador único del estudiante.
     * @param nombre Nombre del estudiante.
     * @param edad Edad del estudiante.
     */
    public Estudiante(int id, String nombre, int edad) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }

    /**
     * Obtiene el ID del estudiante.
     * @return El ID del estudiante.
     */
    public int getId() { return id; }

    /**
     * Obtiene el nombre del estudiante.
     * @return El nombre del estudiante.
     */
    public String getNombre() { return nombre; }

    /**
```

```
     * Obtiene la edad del estudiante.
     * @return La edad del estudiante.
     */
    public int getEdad() { return edad; }

    /**
     * Establece el nombre del estudiante.
     * @param nombre El nuevo nombre del estudiante.
     */
    public void setNombre(String nombre) { this.nombre = nombre; }

    /**
     * Establece la edad del estudiante.
     * @param edad La nueva edad del estudiante.
     */
    public void setEdad(int edad) { this.edad = edad; }

    /**
     * Devuelve una representación en cadena de la información del estudiante.
     * @return Una cadena con el ID, nombre y edad del estudiante.
     */
    public String getInfo() {
        return String.format("ID: %d | Nombre: %s | Edad: %d", id, nombre, edad);
    }
}
```

Explicación: La clase Estudiante es una clase modelo que representa la entidad base de un estudiante con atributos como id, nombre y edad. Es la base para el patrón Decorator, ya que otras clases como DescuentoMatriculaDecorator la envuelven para añadir funcionalidades. No implementa un patrón de diseño en sí misma, pero su diseño simple y genérico permite que sea extensible. En ejecución, los objetos de esta clase se crean en EstudiantesUI al agregar un estudiante y se almacenan en EstudianteRepositorio. Su método getInfo() proporciona una representación básica que puede ser extendida por decoradores, lo que la hace ideal para el patrón Decorator.

- **EstudianteServicio.java**

```
package org.example.servicio;

import org.example.modelo.Estudiante;
import org.example.repositorio.EstudianteRepositorio;
```

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

/**
 * Clase que actúa como capa de servicio para gestionar operaciones sobre estudiantes.
 * Implementa el patrón de diseño Facade, proporcionando una interfaz simplificada
 * para interactuar con el repositorio de estudiantes.
 */
public class EstudianteServicio {
    // Repositorio que maneja el almacenamiento de estudiantes
    private final EstudianteRepositorio repositorio;

    /**
     * Constructor que inicializa el servicio con un repositorio.
     * @param repositorio El repositorio de estudiantes a utilizar.
     */
    public EstudianteServicio(EstudianteRepositorio repositorio) {
        this.repositorio = repositorio;
    }

    /**
     * Agrega un estudiante al repositorio.
     * @param e El estudiante a agregar.
     */
    public void agregarEstudiante(Estudiante e) {
        repositorio.agregar(e);
    }

    /**
     * Elimina un estudiante del repositorio por su ID.
     * @param id El ID del estudiante a eliminar.
     */
    public void eliminarEstudiante(int id) {
        repositorio.eliminar(id);
    }

    /**
     * Actualiza la información de un estudiante en el repositorio.
     * @param id El ID del estudiante a actualizar.
```

```
     * @param nombre El nuevo nombre del estudiante.
     * @param edad La nueva edad del estudiante.
     */
    public void actualizarEstudiante(int id, String nombre, int edad) {
        repositorio.actualizar(id, nombre, edad);
    }

    /**
     * Obtiene todos los estudiantes almacenados en el repositorio.
     * @return Una lista de todos los estudiantes.
     */
    public List<Estudiante> obtenerTodos() {
        return repositorio.obtenerTodos();
    }

    /**
     * Obtiene los estudiantes que tienen un decorador aplicado (como descuento).
     * @return Una lista de estudiantes con decoradores.
     */
    public List<Estudiante> obtenerConDescuento() {
        List<Estudiante> filtrados = new ArrayList<>();
        for (Estudiante e : repositorio) {
            if (e instanceof EstudianteDecorator) {
                filtrados.add(e);
            }
        }
        return filtrados;
    }
}
```

Explicación: La clase EstudianteServicio implementa el patrón Facade, proporcionando una interfaz simplificada para interactuar con el repositorio de estudiantes (EstudianteRepositorio). Su propósito es encapsular las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y filtrar estudiantes con descuentos. En ejecución, esta clase es utilizada por EstudiantesUI para manejar las acciones del usuario, cómo agregar, actualizar o eliminar estudiantes, delegando las operaciones al repositorio. El método obtenerConDescuento utiliza el polimorfismo para identificar estudiantes decorados, lo que demuestra la integración con el patrón Decorator. Este diseño mejora la modularidad y reduce el acoplamiento entre la interfaz de usuario y el almacenamiento.

- **EstudiantesUI.java**

```
package org.example.presentacion;

import org.example.modelo.Estudiante;
import org.example.repositorio.EstudianteRepositorio;
import org.example.servicio.EstudianteServicio;
import org.example.servicio.DescuentoMatriculaDecorator;

import javax.swing.*;
import java.awt.*;

/**
 * Clase que representa la interfaz gráfica de usuario (GUI) para la gestión de estudiantes.
 * Utiliza Swing para crear un formulario interactivo y aplica el patrón Decorator
 * para estudiantes con descuento.
 */
public class EstudiantesUI extends JFrame {
    // Campos de texto para ingresar datos del estudiante
    private JTextField txtId, txtNombre, txtEdad;
    // Área de texto para mostrar la lista de estudiantes
    private JTextArea areaTexto;
    // Servicio que maneja la lógica de negocio
    private final EstudianteServicio servicio;

    /**
     * Constructor que inicializa la ventana y sus componentes.
     */
    public EstudiantesUI() {
        super("Gestión de Estudiantes");
        setSize(550, 400);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        // Inicializa el servicio con un repositorio
        servicio = new EstudianteServicio(new EstudianteRepositorio());

        // Panel para los campos de entrada
        JPanel panel = new JPanel(new GridLayout(4, 2));
        txtId = new JTextField();
```

```
txtNombre = new JTextField();
txtEdad = new JTextField();

panel.add(new JLabel("ID:"));
panel.add(txtId);
panel.add(new JLabel("Nombre:"));
panel.add(txtNombre);
panel.add(new JLabel("Edad:"));
panel.add(txtEdad);

// Botón para agregar un estudiante
JButton btnAgregar = new JButton("Agregar");
btnAgregar.addActionListener(e -> {
    int id = Integer.parseInt(txtId.getText());
    String nombre = txtNombre.getText();
    int edad = Integer.parseInt(txtEdad.getText());

    Estudiante estudiante = new Estudiante(id, nombre, edad);

    // Aplica el decorador si el estudiante tiene menos de 20 años
    if (edad < 20) {
        estudiante = new DescuentoMatriculaDecorator(estudiante, 15);
    }

    servicio.agregarEstudiante(estudiante);
    mostrarEstudiantes();
});

// Botón para actualizar un estudiante
JButton btnActualizar = new JButton("Actualizar");
btnActualizar.addActionListener(e -> {
    servicio.actualizarEstudiante(
            Integer.parseInt(txtId.getText()),
            txtNombre.getText(),
            Integer.parseInt(txtEdad.getText())
    );
    mostrarEstudiantes();
});

// Botón para eliminar un estudiante
```

```
    JButton btnEliminar = new JButton("Eliminar");
    btnEliminar.addActionListener(e -> {
        servicio.eliminarEstudiante(Integer.parseInt(txtId.getText()));
        mostrarEstudiantes();
    });

    // Botón para mostrar estudiantes con descuento
    JButton btnDescuentos = new JButton("Mostrar con Descuento");
    btnDescuentos.addActionListener(e -> {
        StringBuilder sb = new StringBuilder("Estudiantes con Descuento:\n\n");
        for (Estudiante est : servicio.obtenerConDescuento()) {
            sb.append(est.getInfo()).append("\n");
        }
        areaTexto.setText(sb.toString());
    });

    // Panel para los botones
    JPanel panelBotones = new JPanel();
    panelBotones.add(btnAgregar);
    panelBotones.add(btnActualizar);
    panelBotones.add(btnEliminar);
    panelBotones.add(btnDescuentos);

    // Área de texto para mostrar resultados
    areaTexto = new JTextArea();
    areaTexto.setEditable(false);

    // Agrega los componentes a la ventana
    add(panel, BorderLayout.NORTH);
    add(panelBotones, BorderLayout.CENTER);
    add(new JScrollPane(areaTexto), BorderLayout.SOUTH);
}

/**
 * Actualiza el área de texto con la lista de estudiantes registrados.
 */
private void mostrarEstudiantes() {
    StringBuilder sb = new StringBuilder("Estudiantes registrados:\n\n");
    for (Estudiante e : servicio.obtenerTodos()) {
        sb.append(e.getInfo()).append("\n");
```

```
        }
        areaTexto.setText(sb.toString());
    }
}
```

Explicación: La clase EstudiantesUI es la interfaz gráfica de usuario (GUI) construida con Swing, que permite al usuario interactuar con el sistema mediante un formulario para agregar, actualizar, eliminar y listar estudiantes. Utiliza el patrón Decorator al envolver estudiantes con DescuentoMatriculaDecorator si tienen menos de 20 años, integrándose con el patrón Facade a través de EstudianteServicio. En ejecución, la interfaz responde a eventos de botones, delegando las operaciones al servicio y actualizando el área de texto con la información de los estudiantes. Aunque no implementa un patrón de diseño específico en sí misma, actúa como el componente de presentación en una arquitectura en capas, integrando los patrones Decorator y Facade para ofrecer una experiencia de usuario cohesiva.

- **EstudianteRepositorio.java**

```
package org.example.repositorio;

import org.example.modelo.Estudiante;

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

/**
 * Clase que actúa como repositorio para almacenar y gestionar estudiantes.
 * Implementa el patrón Repository y la interfaz Iterable para permitir
 * la iteración sobre la lista de estudiantes.
 */
public class EstudianteRepositorio implements Iterable<Estudiante> {
    // Lista que almacena los estudiantes
    private final List<Estudiante> estudiantes = new ArrayList<>();

    /**
     * Agrega un estudiante a la lista.
     * @param estudiante El estudiante a agregar.
     */
    public void agregar(Estudiante estudiante) {
        estudiantes.add(estudiante);
    }
```

```
/**
 * Elimina un estudiante por su ID.
 * @param id El ID del estudiante a eliminar.
 */
public void eliminar(int id) {
    estudiantes.removeIf(e -> e.getId() == id);
}

/**
 * Actualiza la información de un estudiante identificado por su ID.
 * @param id El ID del estudiante a actualizar.
 * @param nombre El nuevo nombre del estudiante.
 * @param edad La nueva edad del estudiante.
 */
public void actualizar(int id, String nombre, int edad) {
    for (Estudiante e : estudiantes) {
        if (e.getId() == id) {
            e.setNombre(nombre);
            e.setEdad(edad);
            break;
        }
    }
}

/**
 * Obtiene todos los estudiantes almacenados.
 * @return Una lista de estudiantes.
 */
public List<Estudiante> obtenerTodos() {
    return estudiantes;
}

/**
 * Proporciona un iterador para recorrer la lista de estudiantes.
 * @return Un iterador de estudiantes.
 */
@Override
public Iterator<Estudiante> iterator() {
    return estudiantes.iterator();
```

```
    }
}
```

Explicación: La clase EstudianteRepositorio implementa el patrón Repository, que encapsula la lógica de almacenamiento y recuperación de datos, en este caso, una lista en memoria de objetos Estudiante. También implementa la interfaz Iterable para permitir la iteración sobre los estudiantes, lo que facilita su uso en bucles for-each en EstudianteServicio. En ejecución, esta clase gestiona las operaciones CRUD, almacenando estudiantes (incluidos los decorados) y proporcionando métodos para agregar, eliminar, actualizar y recuperar datos. El patrón Repository asegura que la lógica de almacenamiento esté separada de la lógica de negocio, promoviendo un diseño limpio y mantenible.

- **Main.java**

```
package org.example;
import org.example.presentacion.EstudiantesUI;

/**
 * Clase principal que inicia la aplicación.
 * Utiliza SwingUtilities para ejecutar la interfaz gráfica en el hilo de despacho de eventos.
 */
public class Main {
    /**
     * Método principal que lanza la aplicación.
     * @param args Argumentos de la línea de comandos (no utilizados).
     */
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(() -> new EstudiantesUI().setVisible(true));
    }
}
```

Explicación: La clase Main es el punto de entrada de la aplicación, encargada de iniciar la interfaz gráfica EstudiantesUI. Utiliza SwingUtilities.invokeLater para garantizar que la GUI se ejecute en el hilo de despacho de eventos de Swing, siguiendo las mejores prácticas para aplicaciones gráficas en Java. No implementa un patrón de diseño específico, pero su simplicidad permite que el resto del sistema, que utiliza patrones como Decorator, Facade y Repository, funcione de manera coordinada. En ejecución, esta clase crea y muestra la ventana principal, iniciando la interacción del usuario con el sistema.

- **EstudianteDecorator.java**

```
package org.example.servicio;

import org.example.modelo.Estudiante;

/**
 * Clase abstracta que sirve como base para los decoradores de estudiantes.
 * Implementa el patrón Decorator, permitiendo extender la funcionalidad de un estudiante
 * sin modificar su clase base.
 */
public abstract class EstudianteDecorator extends Estudiante {
    // Referencia al estudiante base que se va a decorar
    protected Estudiante estudianteBase;

    /**
     * Constructor que inicializa el decorador con un estudiante base.
     * @param estudianteBase El estudiante a decorar.
     */
    public EstudianteDecorator(Estudiante estudianteBase) {
        super(estudianteBase.getId(), estudianteBase.getNombre(), estudianteBase.getEdad());
        this.estudianteBase = estudianteBase;
    }

    /**
     * Método abstracto que debe ser implementado por los decoradores concretos.
     * Por defecto, delega la llamada al estudiante base.
     * @return La información del estudiante base.
     */
    @Override
    public String getInfo() {
        return estudianteBase.getInfo();
    }
}
```

Explicación: La clase EstudianteDecorator es una clase abstracta que define la estructura base para el patrón Decorator. Extiende Estudiante y mantiene una referencia al objeto Estudiante que decora, permitiendo que clases como DescuentoMatriculaDecorator añadan funcionalidades adicionales. En ejecución, esta clase actúa como un puente entre el estudiante base y los decoradores concretos, asegurando que los métodos como getInfo() puedan ser extendidos sin

modificar la clase Estudiante. El patrón Decorator permite una extensión flexible y dinámica, como añadir descuentos u otras características, sin alterar el código existente de los estudiantes.

Arquitectura	
Separación de responsabilidades	Cada capa tiene una tarea clara y definida.
Mantenibilidad	Puedes modificar una capa sin afectar las demás.
Reutilización	Las clases del modelo o repositorio pueden usarse en otros contextos.
Pruebas unitarias	Es más fácil probar cada capa de forma aislada.
Escalabilidad	Permite añadir más funcionalidades (como base de datos real) sin rehacer todo.

Conclusiones.

La arquitectura en capas permite organizar el código dividiendo responsabilidades. Cada capa cumple un rol específico: la presentación muestra la interfaz, el servicio gestiona la lógica del negocio, el repositorio maneja los datos, y el modelo representa las entidades.

Gracias a esta estructura, el sistema es más fácil de mantener, escalar y modificar, ya que los cambios en una capa no afectan directamente a las demás. Además, mejora la claridad del código, facilita las pruebas y permite integrar patrones de diseño como el Decorator sin complicaciones.

En resumen, usar una arquitectura en capas ayuda a construir aplicaciones más limpias, flexibles y preparadas para crecer o adaptarse a futuro.