**Universidad de las Fuerzas Armadas ESPE**

**Departamento: Ciencias de la Computación**

**Carrera : Ingeniería de Software**

**Taller académico Nª: 3**

**1. Información General**

* **Asignatura:** Análisis y Diseño
* **Apellidos y nombres de los estudiantes:** Pamela Chipe, Carlos Jaya, Elkin Pabón
* **NRC:** 23305
* **Fecha de realización:** 12**/**06**/**2025

**2. Objetivo del Taller y Desarrollo**

**Objetivo del Taller:**

Desarrollar en un entorno IDE de Java una aplicación que implemente un CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar datos de estudiantes (orden, nombre y edad), utilizando el patrón de arquitectura en 3 capas. Se busca demostrar cómo interactúan las capas entre sí y cómo se ejecuta la lógica del sistema desde la clase principal (Main.java).

**Desarrollo:**

**Requisitos Funcionales**

**• RF1:** La aplicación debe permitir agregar un nuevo estudiante con los datos: orden, nombre y edad.

**• RF2:** El sistema debe mostrar todos los estudiantes registrados.

**• RF3:** El usuario debe poder consultar un estudiante por su número de orden.

**• RF4:** El sistema debe permitir modificar los datos de un estudiante existente.

**• RF5: E**l sistema debe permitir eliminar un estudiante por su número de orden.

**• RF6:** La aplicación debe estar organizada en tres capas: modelo, repositorio, servicio y presentación.

**Construcción de la estructura:**

src/

|

└──ec.edu.espe/

│

├── datos/

│ ├── model/

│ │ └── Estudiante.java

│ │

│ └── repository/

│ └── EstudianteRepository.java

│

├── logica\_negocio/

│ └── EstudianteService.java

│

├── presentacion/

│ └── EstudianteUI.java

│

└── Main.java

***Main.java***

package ec.edu.espe;

import ec.edu.espe.presentacion.EstudianteUI;

/\*\*

\* Clase principal del programa.

\* Esta clase contiene el método main, punto de entrada de la aplicación.

\* Su función es crear y mostrar la interfaz gráfica de usuario para gestionar estudiantes.

\*/

public class Main {

/\*\*

\* Método principal (main) que se ejecuta al iniciar la aplicación.

\* Crea una instancia de la interfaz gráfica EstudianteUI y la hace visible.

\*

\* @param args Argumentos de línea de comandos (no utilizados en esta aplicación).

\*/

public static void main(String[] args) {

EstudianteUI estudianteUI = new EstudianteUI(); // Crear la interfaz de usuario

estudianteUI.setVisible(true); // Hacer la ventana visible

}

}

**Capa de datos**

Esta capa se encarga del acceso y la gestión de los datos. Aquí se definen los modelos de datos y el repositorio donde se guardan

**Capa de *modelo***

package ec.edu.espe.datos.model;

/\*\*

\* Clase que representa a un estudiante.

\* Contiene los atributos básicos de un estudiante como ID, nombre y edad.

\* Se proporcionan métodos para acceder y modificar estos atributos,

\* así como un método para mostrar la información del estudiante como cadena de texto.

\*/

public class Estudiante {

private int id; // Identificador único del estudiante

private String nombre; // Nombre del estudiante

private int edad; // Edad del estudiante

/\*\*

\* Constructor de la clase Estudiante.

\* Inicializa un nuevo estudiante con el ID, nombre y edad especificados.

\*

\* @param id Identificador 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 ID del estudiante

\*/

public int getId() {

return id;

}

/\*\*

\* Establece el ID del estudiante.

\*

\* @param id Nuevo ID del estudiante

\*/

public void setId(int id) {

this.id = id;

}

/\*\*

\* Obtiene el nombre del estudiante.

\*

\* @return Nombre del estudiante

\*/

public String getNombre() {

return nombre;

}

/\*\*

\* Establece el nombre del estudiante.

\*

\* @param nombre Nuevo nombre del estudiante

\*/

public void setNombre(String nombre) {

this.nombre = nombre;

}

/\*\*

\* Obtiene la edad del estudiante.

\*

\* @return Edad del estudiante

\*/

public int getEdad() {

return edad;

}

/\*\*

\* Establece la edad del estudiante.

\*

\* @param edad Nueva edad del estudiante

\*/

public void setEdad(int edad) {

this.edad = edad;

}

/\*\*

\* Retorna una representación en cadena del estudiante.

\*

\* @return Cadena con los datos del estudiante: ID, nombre y edad

\*/

@Override

public String toString() {

return "ID: " + id + ", Nombre: " + nombre + ", Edad: " + edad;

}

}

**Capa de *repositorio***

package ec.edu.espe.repository;

import ec.edu.espe.datos.model.Estudiante;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Clase que actúa como repositorio en memoria para objetos Estudiante.

\* Proporciona métodos para agregar, obtener, actualizar y eliminar estudiantes.

\* La lista interna simula una base de datos simple en memoria.

\*/

public class EstudianteRepository {

private List<Estudiante> estudiantes; // Lista que almacena los estudiantes

/\*\*

\* Constructor que inicializa la lista de estudiantes.

\*/

public EstudianteRepository() {

estudiantes = new ArrayList<>();

}

/\*\*

\* Agrega un nuevo estudiante a la lista.

\*

\* @param estudiante Objeto Estudiante a agregar

\*/

public void agregarEstudiante(Estudiante estudiante) {

estudiantes.add(estudiante);

}

/\*\*

\* Retorna la lista completa de estudiantes.

\*

\* @return Lista de estudiantes

\*/

public List<Estudiante> obtenerEstudiantes() {

return estudiantes;

}

/\*\*

\* Busca y retorna un estudiante según su ID.

\*

\* @param id Identificador del estudiante a buscar

\* @return Estudiante con el ID especificado, o null si no se encuentra

\*/

public Estudiante obtenerEstudiantePorId(int id) {

for (Estudiante estudiante : estudiantes) {

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

return estudiante;

}

}

return null; // Retorna null si el estudiante no se encuentra

}

/\*\*

\* Actualiza los datos de un estudiante existente en la lista.

\*

\* @param estudiante Objeto Estudiante con los nuevos datos (debe tener un ID existente)

\*/

public void actualizarEstudiante(Estudiante estudiante) {

Estudiante estudianteExistente = obtenerEstudiantePorId(estudiante.getId());

if (estudianteExistente != null) {

estudianteExistente.setNombre(estudiante.getNombre());

estudianteExistente.setEdad(estudiante.getEdad());

}

}

/\*\*

\* Elimina un estudiante de la lista según su ID.

\*

\* @param id Identificador del estudiante a eliminar

\*/

public void eliminarEstudiante(int id) {

Estudiante estudiante = obtenerEstudiantePorId(id);

if (estudiante != null) {

estudiantes.remove(estudiante);

}

}

}

**Capa de *servicio***

package ec.edu.espe.logica\_negocio;

import ec.edu.espe.datos.model.Estudiante;

import ec.edu.espe.repository.EstudianteRepository;

import java.util.List;

/\*\*

\* Clase que representa la capa de lógica de negocio para la gestión de estudiantes.

\* Esta clase actúa como intermediaria entre la interfaz de usuario y el repositorio,

\* encapsulando las reglas necesarias para manejar los datos de estudiantes.

\*/

public class EstudianteService {

private EstudianteRepository estudianteRepository; // Repositorio de estudiantes

/\*\*

\* Constructor que inicializa el repositorio de estudiantes.

\*/

public EstudianteService() {

estudianteRepository = new EstudianteRepository();

}

/\*\*

\* Agrega un nuevo estudiante con los datos proporcionados.

\*

\* @param id Identificador del estudiante

\* @param nombre Nombre del estudiante

\* @param edad Edad del estudiante

\*/

public void agregarEstudiante(int id, String nombre, int edad) {

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

estudianteRepository.agregarEstudiante(estudiante);

}

/\*\*

\* Retorna la lista completa de estudiantes registrados.

\*

\* @return Lista de objetos Estudiante

\*/

public List<Estudiante> obtenerEstudiantes() {

return estudianteRepository.obtenerEstudiantes();

}

/\*\*

\* Busca y retorna un estudiante según su ID.

\*

\* @param id Identificador del estudiante a buscar

\* @return Estudiante correspondiente al ID, o null si no existe

\*/

public Estudiante obtenerEstudiantePorId(int id) {

return estudianteRepository.obtenerEstudiantePorId(id);

}

/\*\*

\* Actualiza los datos de un estudiante existente.

\*

\* @param id Identificador del estudiante a actualizar

\* @param nombre Nuevo nombre del estudiante

\* @param edad Nueva edad del estudiante

\*/

public void actualizarEstudiante(int id, String nombre, int edad) {

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

estudianteRepository.actualizarEstudiante(estudiante);

}

/\*\*

\* Elimina un estudiante del sistema según su ID.

\*

\* @param id Identificador del estudiante a eliminar

\*/

public void eliminarEstudiante(int id) {

estudianteRepository.eliminarEstudiante(id);

}

}

**Capa de *presentación***

package ec.edu.espe.presentacion;

import ec.edu.espe.datos.model.Estudiante;

import ec.edu.espe.logica\_negocio.EstudianteService;

import javax.swing.\*;

import javax.swing.table.DefaultTableModel;

import java.awt.\*;

import java.util.List;

public class EstudianteUI extends JFrame {

private EstudianteService estudianteService;

private JTextField txtId, txtNombre, txtEdad;

private JButton btnCrear, btnVer, btnActualizar, btnBorrar;

private JTable tablaEstudiantes;

private DefaultTableModel tableModel;

// Constructor de la clase EstudianteUI que define la interfaz gráfica

public EstudianteUI() {

// Configuración de la ventana principal (JFrame)

setTitle("Gestión de Estudiantes");

setSize(600, 500); // Tamaño de la ventana

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLocationRelativeTo(null);

setLayout(new BorderLayout(10, 10)); // Añadimos espaciado entre los componentes

// Inicialización del servicio

estudianteService = new EstudianteService();

// Panel para los campos de texto (ID, Nombre, Edad)

JPanel panelCampos = new JPanel();

panelCampos.setLayout(new GridBagLayout());

GridBagConstraints gbc = new GridBagConstraints();

gbc.insets = new Insets(5, 5, 5, 5); // Espaciado entre componentes

// Componentes de la interfaz gráfica (campos de texto)

gbc.gridx = 0; gbc.gridy = 0;

panelCampos.add(new JLabel("ID:"), gbc);

gbc.gridx = 1; gbc.gridy = 0;

txtId = new JTextField(20);

panelCampos.add(txtId, gbc);

gbc.gridx = 0; gbc.gridy = 1;

panelCampos.add(new JLabel("Nombre:"), gbc);

gbc.gridx = 1; gbc.gridy = 1;

txtNombre = new JTextField(20);

panelCampos.add(txtNombre, gbc);

gbc.gridx = 0; gbc.gridy = 2;

panelCampos.add(new JLabel("Edad:"), gbc);

gbc.gridx = 1; gbc.gridy = 2;

txtEdad = new JTextField(20);

panelCampos.add(txtEdad, gbc);

// Panel para los botones, utilizando FlowLayout (para alinearlos horizontalmente)

JPanel panelBotones = new JPanel();

panelBotones.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10)); // Botones en una fila, centrados

// Botones de acción (sin color de fondo)

btnCrear = new JButton("Crear Estudiante");

btnVer = new JButton("Ver Todos");

btnActualizar = new JButton("Actualizar");

// Añadir los botones al panelBotones

panelBotones.add(btnCrear);

panelBotones.add(btnVer);

panelBotones.add(btnActualizar);

// Panel para los campos de texto y los botones

JPanel panelPrincipal = new JPanel();

panelPrincipal.setLayout(new BorderLayout());

panelPrincipal.add(panelCampos, BorderLayout.NORTH); // Los campos encima de los botones

panelPrincipal.add(panelBotones, BorderLayout.CENTER); // Los botones debajo de los campos

// Agregar el panel principal a la ventana

add(panelPrincipal, BorderLayout.NORTH);

// Crear la tabla para visualizar los estudiantes

String[] columnNames = {"ID", "Nombre", "Edad"};

tableModel = new DefaultTableModel(columnNames, 0);

tablaEstudiantes = new JTable(tableModel);

JScrollPane scrollPane = new JScrollPane(tablaEstudiantes);

add(scrollPane, BorderLayout.CENTER); // Mostrar la tabla en la parte inferior

// Eventos de los botones

btnCrear.addActionListener(e -> crearEstudiante());

btnVer.addActionListener(e -> verEstudiantes());

btnActualizar.addActionListener(e -> actualizarEstudiante());

}

// Método para crear un estudiante

private void crearEstudiante() {

try {

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

String nombre = txtNombre.getText();

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

// Verificar si el estudiante ya existe, si existe lo eliminamos antes de agregar

Estudiante estudianteExistente = estudianteService.obtenerEstudiantePorId(id);

if (estudianteExistente != null) {

// Eliminar al estudiante existente si tiene el mismo ID

estudianteService.eliminarEstudiante(id);

JOptionPane.showMessageDialog(this, "Estudiante existente eliminado.");

} else {

// Si el estudiante no existe, se muestra un mensaje indicando que no se puede eliminar

JOptionPane.showMessageDialog(this, "No existe un estudiante con ese ID para eliminar.");

}

// Crear el nuevo estudiante solo si el ID no está duplicado

estudianteService.agregarEstudiante(id, nombre, edad);

JOptionPane.showMessageDialog(this, "Estudiante creado con éxito");

limpiarCampos();

} catch (NumberFormatException ex) {

JOptionPane.showMessageDialog(this, "Por favor, ingresa datos válidos.");

}

}

// Método para ver todos los estudiantes en la tabla

private void verEstudiantes() {

List<Estudiante> estudiantes = estudianteService.obtenerEstudiantes();

// Limpiar la tabla antes de agregar los nuevos datos

tableModel.setRowCount(0);

// Agregar los estudiantes a la tabla

for (Estudiante estudiante : estudiantes) {

tableModel.addRow(new Object[]{estudiante.getId(), estudiante.getNombre(), estudiante.getEdad()});

}

}

// Método para actualizar los datos de un estudiante

private void actualizarEstudiante() {

try {

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

String nombre = txtNombre.getText();

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

estudianteService.actualizarEstudiante(id, nombre, edad);

JOptionPane.showMessageDialog(this, "Estudiante actualizado con éxito");

limpiarCampos();

} catch (NumberFormatException ex) {

JOptionPane.showMessageDialog(this, "Por favor, ingresa datos válidos.");

}

}

// Limpiar los campos de texto después de realizar una acción

private void limpiarCampos() {

txtId.setText("");

txtNombre.setText("");

txtEdad.setText("");

}

// Método principal para ejecutar la GUI

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

EstudianteUI ui = new EstudianteUI();

ui.setVisible(true); // Hacer la ventana visible

});

}

}

**Conclusiones:**

| Arquitectura en Capas |
| --- |
| 1. Divide el sistema en capas de presentación, lógica del negocio y acceso a los datos. |
| 1. Separa las capas de modo que cada capa tiene un propósito claro |
| 1. Esta separación hace que la mantenibilidad y las pruebas sean más fáciles de realizar, de modo que cualquier cambio sea más rápido de implementar |
| 1. Permite que la reutilización de la lógica del negocio sea mucho más sencilla, y que los datos puedan ser usados por diferentes interfaces |
| 1. Parece favorecer la escalabilidad |

**Conclusión:**

Gracias a la arquitectura, se puede separar las capas de forma que cada una tenga un propósito en específico, facilitando la mantenibilidad y el trabajo en equipo ya que cada integrante puede implementar una funcionalidad, y permite la reutilización y la escalabilidad