TALLER N. 3

DATOS INFORMATIVOS

Carrera: Ingeniería de Software

Asignatura: Analisis y Diseño de Software

Tema del taller: Arquitectura 3 capas

Docente: Ing. Jenny Ruiz

Integrantes: Stefany Díaz, Moisés Benalcázar, Mateo Medranda

Fecha: 24 de nov. de 25 **NRC**: 27837

1. Introducción

En el presente trabajo se desarrolló una aplicación CRUD para la gestión de estudiantes, aplicando la arquitectura de tres capas junto con el patrón Modelo–Vista–Controlador (MVC) bajo el enfoque GEMA, e integrando el patrón de diseño Singleton en la capa de datos.

El objetivo principal fue estructurar el sistema de forma modular, clara y mantenible, asegurando una adecuada separación de responsabilidades entre las capas de modelo, lógica de negocio y presentación.

Además, se buscó demostrar cómo la aplicación del patrón Singleton permite garantizar la existencia de una única instancia del repositorio de datos, evitando problemas de inconsistencias y pérdida de información durante la ejecución del sistema.

2. Desarrollo

2.1. ¿Qué es GEMA?

El enfoque GEMA representa una forma estructurada de aplicar el patrón MVC junto con la arquitectura de tres capas. Su objetivo es organizar el sistema de manera que cada componente tenga una responsabilidad clara, evitando el acoplamiento innecesario entre módulos.

En este proyecto, GEMA permitió definir una arquitectura donde:

- El modelo se encarga únicamente de representar los datos.
- El repositorio gestiona el almacenamiento.
- El servicio aplica la lógica de negocio.
- La interfaz gráfica actúa como vista y controlador.

Esta estructura facilita el mantenimiento, escalabilidad y comprensión del código.

2.2. Arquitectura de Tres Capas

La aplicación se estructuró en las siguientes capas:

Tabla 1. Arquitectura de 3 capas

Capa	Clase	Función
Modelo	Estudiante	Representa los datos
Datos	EstudianteRepository	Gestiona almacenamiento
Negocio	EstudianteService	Aplica reglas
Vista / Control	EstudianteUI	Interfaz y eventos

- **Modelo**: Representación de la entidad estudiante.
- **Repositorio**: Encargado del acceso a datos en memoria.
- **Servicio**: Aplica reglas de negocio y validaciones.

Además, se implementó una capa de presentación usando Java Swing.

Esta separación garantiza que cada capa tenga una única responsabilidad, facilitando cambios futuros sin afectar toda la estructura del sistema.

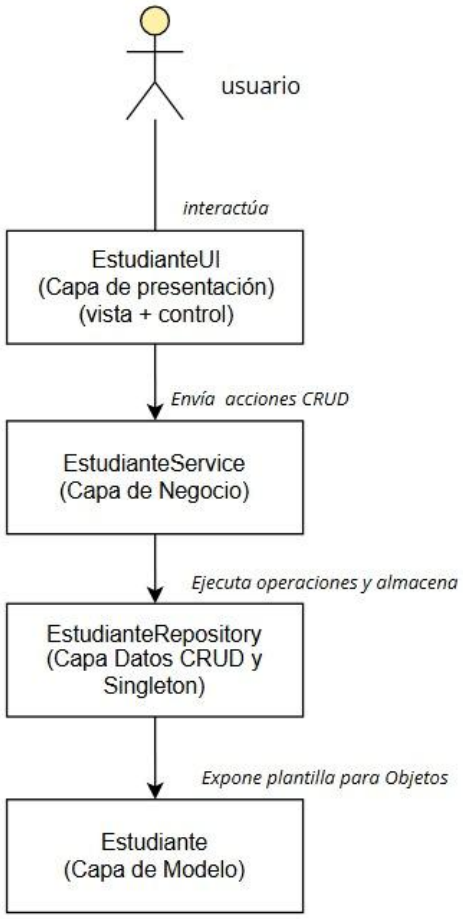

Figura 1. Arquitectura GEMA – MVC – NVC – Singleton CRUD

3. Implementación del CRUD

3.1. Modelo

```
package main.java.ec.edu.espe.datos.model;


public class Estudiante {

    private String id;

    private String nombre;

    private int edad;


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

        this.id = id;

        this.nombre = nombre;

        this.edad = edad;

    }


    public String getId() {

        return id;

    }


    public void setId(String 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;
    }

}
```

3.2. Repository (antes de Singleton)

```
package main.java.ec.edu.espe.datos.repository;
import main.java.ec.edu.espe.datos.model.Estudiante;
import java.util.ArrayList;

public class EstudianteRepository {
    private ArrayList<Estudiante> listaEstudiantes = new ArrayList<>();
    public EstudianteRepository() {
    }

    public String agregar(Estudiante estudiante) {
        try {
            listaEstudiantes.add(estudiante);
        } catch (Exception e) {
            return "Hubo un error al insertar un estudiante: " + e.toString();
        }
        return "El estudiante se agregó con éxito";
    }

    public ArrayList<Estudiante> listar() {
```

```
        return listaEstudiantes;
    }

    public Estudiante buscarPorId(String id) {
        for (Estudiante estudiante : listaEstudiantes) {
            if (estudiante.getId().equals(id)) {
                return estudiante;
            }
        }
        return null;
    }

    public Estudiante editar(String id, Estudiante estudianteEditado) {
        for (int i = 0; i < listaEstudiantes.size(); i++) {
            if (listaEstudiantes.get(i).getId().equals(id)) {
                listaEstudiantes.set(i, estudianteEditado);
                return estudianteEditado;
            }
        }
        return null;
    }

    public String eliminar(String id) {
        try {
            listaEstudiantes.removeIf(estudiante ->
estudiante.getId().equals(id));
        } catch (Exception e) {
            return "Hubo un error al eliminar un estudiante: " + e.toString();
        }
        return "El estudiante se eliminó con éxito";
    }
```

}

Inicialmente el repositorio gestionaba una lista interna, pero cada nueva instancia creaba una lista diferente, lo cual generaba inconsistencia de datos.

4. Implementación del patrón Singleton

4.1. Definición del patrón Singleton

El patrón Singleton es un patrón de diseño creacional que garantiza que una clase tenga una única instancia durante toda la ejecución del programa, proporcionando un punto global de acceso a dicha instancia.

En este proyecto se aplicó Singleton al repositorio `EstudianteRepository` para asegurar que todos los componentes del sistema utilicen una sola lista de estudiantes compartida.

4.2. Código del Singleton

```
package main.java.ec.edu.espe.datos.repository;

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

import java.util.ArrayList;


public class EstudianteRepository {

    // Variable estática que almacenará la única instancia de la clase

    // Al ser static, pertenece a la clase y no a un objeto específico

    private static EstudianteRepository instance;


    // Lista donde se almacenarán todos los estudiantes en memoria

    private ArrayList<Estudiante> listaEstudiantes = new ArrayList<>();


    // Constructor PRIVADO para evitar que se creen objetos desde fuera

    // Con esto obligamos a que solo se use getInstance()

    private EstudianteRepository() {

        // Inicializamos la lista

        listaEstudiantes = new ArrayList<>();

    }
```

```
// Método público y estático que permite obtener la ÚNICA instancia de la clase
public static EstudianteRepository getInstance() {
    // Si aún no existe ninguna instancia...
    if (instance == null) {
        // Se crea la única instancia
        instance = new EstudianteRepository();
    }
    // Se devuelve la misma instancia siempre
    return instance;
}

public String agregar(Estudiante estudiante) {
    try {
        listaEstudiantes.add(estudiante);
    } catch (Exception e) {
        return "Hubo un error al insertar un estudiante: " + e.toString();
    }
    return "El estudiante se agregó con éxito";
}

public ArrayList<Estudiante> listar() {
    return listaEstudiantes;
}

public Estudiante buscarPorId(String id) {
    for (Estudiante estudiante : listaEstudiantes) {
        if (estudiante.getId().equals(id)) {
            return estudiante;
        }
    }
    return null;
```

```
    }


    public Estudiante editar(String id, Estudiante estudianteEditado) {
        for (int i = 0; i < listaEstudiantes.size(); i++) {
            if (listaEstudiantes.get(i).getId().equals(id)) {
                listaEstudiantes.set(i, estudianteEditado);
                return estudianteEditado;
            }
        }
        return null;
    }


    public String eliminar(String id) {
        try {
            listaEstudiantes.removeIf(estudiante                                    ->
estudiante.getId().equals(id));
        } catch (Exception e) {
            return "Hubo un error al eliminar un estudiante: " + e.toString();
        }
        return "El estudiante se eliminó con éxito";
    }
}
```

Con este patrón se garantiza que toda la aplicación utilice una sola fuente de datos compartida.

4.3. Modificación del service

```
repo = EstudianteRepository.getInstance();
```

Esta instrucción permite acceder a la instancia única del repositorio, implementando el patrón Singleton. Esto asegura que todas las operaciones del sistema trabajen sobre una misma fuente de datos compartida, evitando inconsistencias causadas por la creación de múltiples instancias.

5. Integración con NVC

El sistema también sigue el patrón NVC (Negocio–Vista–Control), donde:

Tabla 2. Componentes y roles de NVC

Componente	Rol
EstudianteService	Negocio
EstudianteUI	Vista
Eventos en botones	Control

Esta estructura permite que la vista no contenga lógica de negocio, sino que se limite únicamente a interactuar con el usuario y delegar las acciones al servicio, fortaleciendo así la separación de responsabilidades.

6. Integración MVC + Singleton

La integración del patrón Singleton dentro de la arquitectura MVC permite mejorar la estabilidad y la consistencia del sistema.

- Mientras MVC se encarga de distribuir responsabilidades, el patrón Singleton asegura que el manejo de datos sea centralizado y coherente.
- Esta combinación evita errores comunes como la duplicación de información o la creación innecesaria de objetos, optimizando el uso de recursos del sistema.

7. Análisis y discusión

Tabla 3. Cuadro comparativo: MVC vs Singleton

Criterio	MVC	Singleton
¿Qué problema resuelve?	Resuelve el problema de mezclar la lógica con la interfaz, separando responsabilidades entre Modelo, Vista y Controlador para organizar mejor el sistema.	Resuelve el problema de crear múltiples instancias de una clase que debería ser única, evitando inconsistencias en el manejo de datos.
¿En qué capa se utiliza?	Se aplica en toda la arquitectura: Vista (UI), Controlador (eventos/acciones) y Modelo (datos). Principalmente organiza la estructura del sistema.	Se utiliza principalmente en la capa de datos, en clases como el repositorio (EstudianteRepository).

¿Cómo influye en el mantenimiento?	Facilita el mantenimiento porque permite modificar una parte (por ejemplo, la interfaz) sin afectar la lógica ni los datos.	Mejora el mantenimiento al centralizar el acceso a un recurso único, evitando duplicaciones y comportamientos inconsistentes.
¿Cómo evita fallas de diseño?	Evita código desordenado, acoplamiento excesivo y errores causados por mezclar lógica con interfaz gráfica.	Evita fallas relacionadas con la creación de múltiples objetos que deberían ser únicos, como inconsistencias o pérdida de datos.

El patrón MVC permite una correcta separación de responsabilidades, evitando que la lógica de negocio se mezcle con la interfaz gráfica, lo que mejora notablemente la mantenibilidad y escalabilidad del sistema. Por otro lado, el patrón Singleton garantiza que exista una sola instancia del repositorio, evitando problemas de duplicación de datos y errores de consistencia.

Ambas soluciones contribuyen a reducir fallos de diseño, aunque actúan en niveles diferentes: MVC a nivel estructural y Singleton a nivel de control de instancias.

8. Evidencias de funcionamiento

El correcto funcionamiento del sistema se evidencia mediante capturas de pantalla donde se muestra:

- **Interfaz principal.**

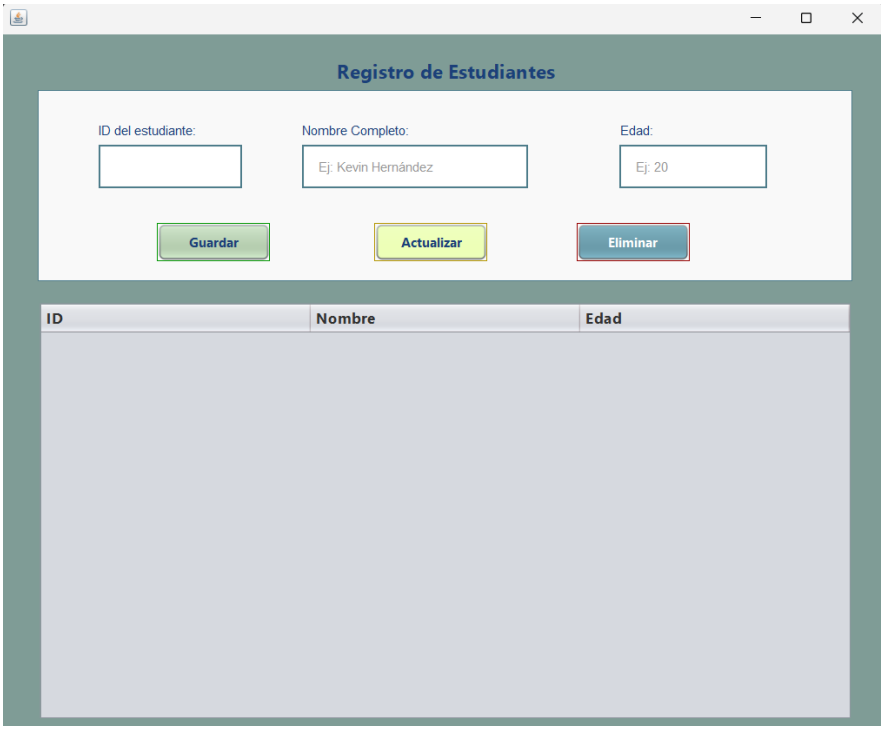

Figura 2. Captura de pantalla de interfaz de la aplicación

- **Registro de estudiantes.**

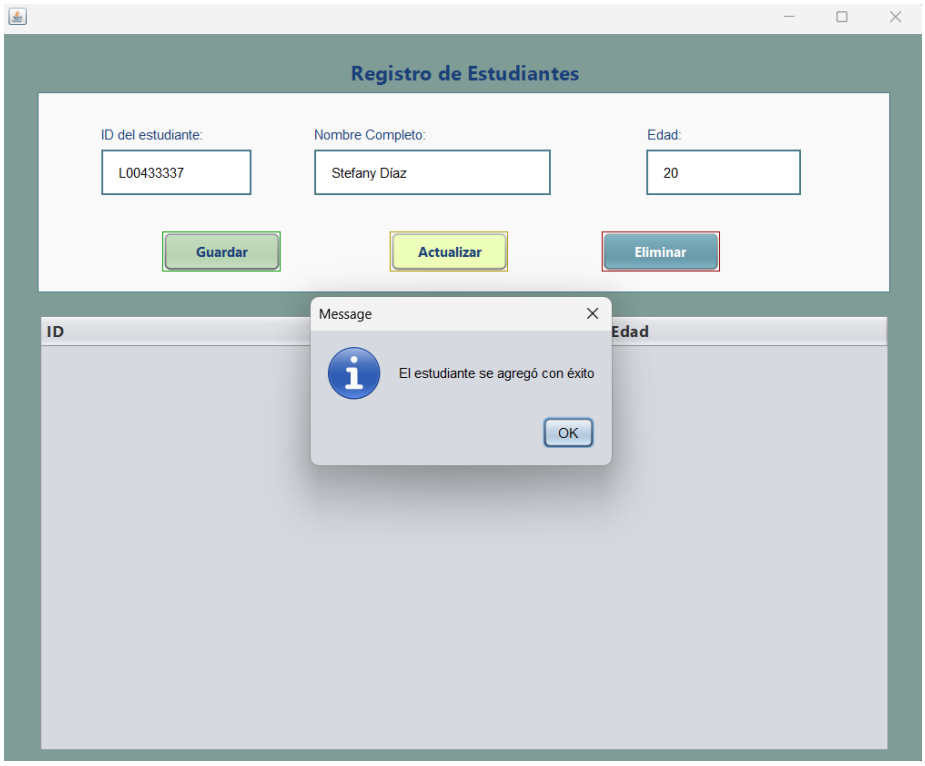

Figura 3. Captura de pantalla de mensaje de estudiante agregado exitosamente.

- **Modificación de registros.**

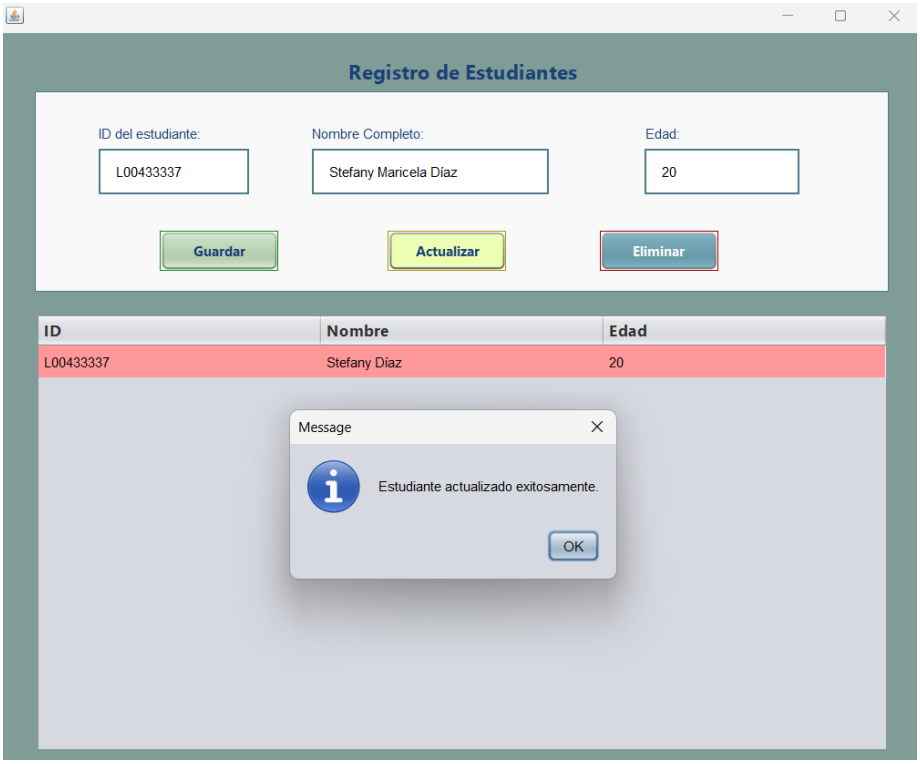

Figura 4. Captura de pantalla de mensaje de estudiante modificado exitosamente.

- **Eliminación de registro.**

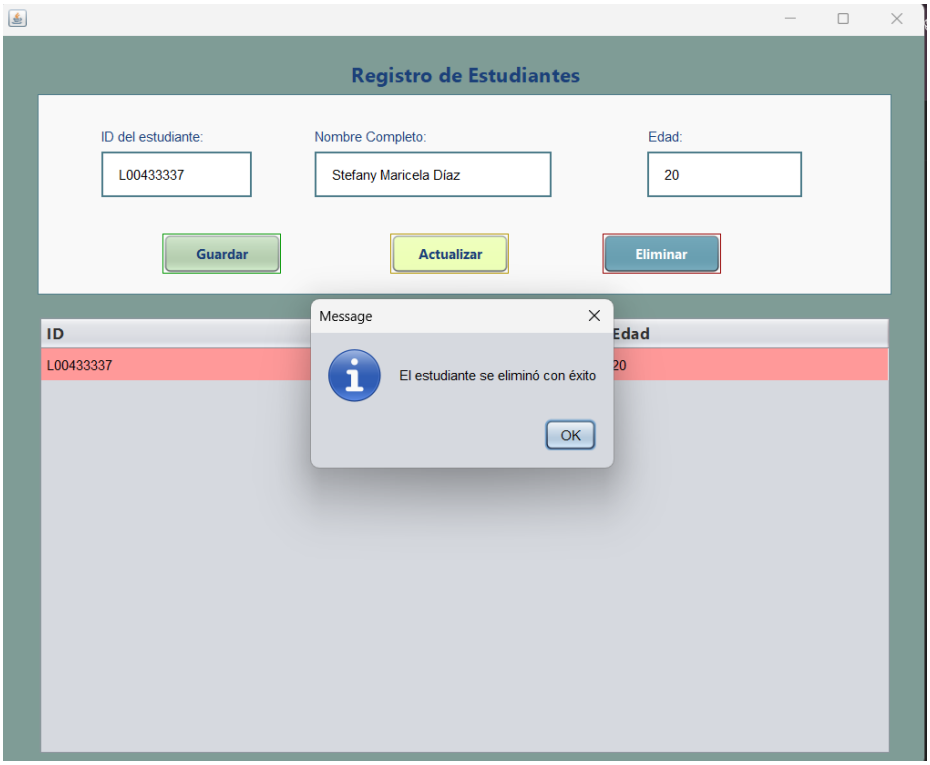

Figura 5. Captura de pantalla de mensaje de estudiante eliminado exitosamente.

- **Persistencia de datos utilizando Singleton.**

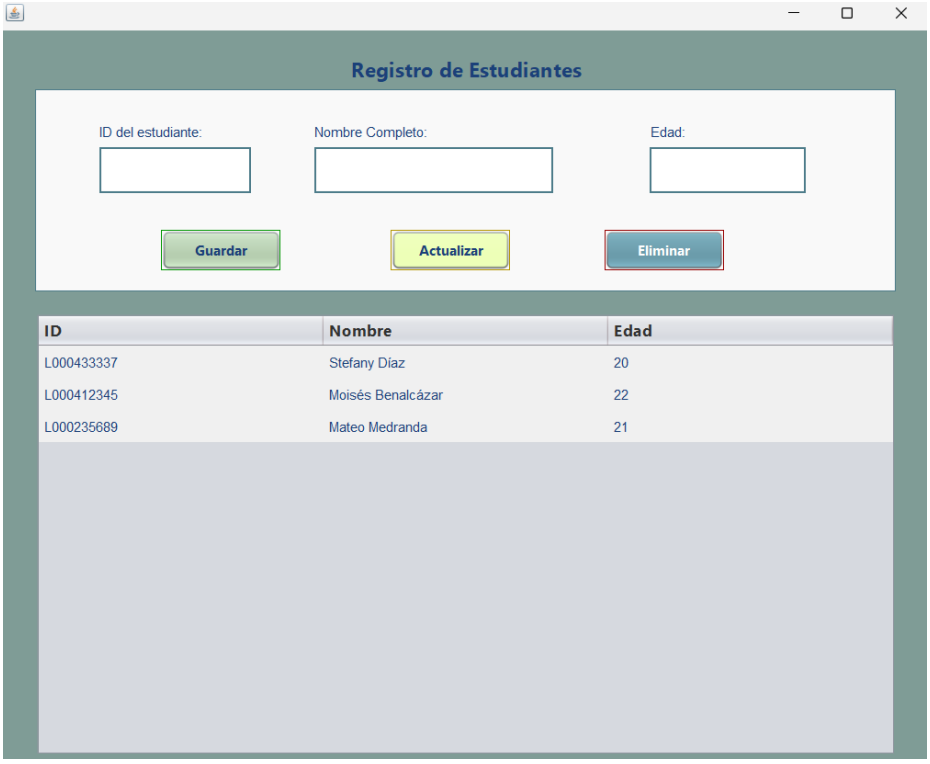

Figura 6. Captura de pantalla de datos con persistencia de Singleton.

Estas evidencias demuestran que tanto el CRUD como la arquitectura solicitada funcionan correctamente.

9. Conclusiones

1. La implementación de la arquitectura GEMA permitió estructurar el sistema en capas bien definidas, separando correctamente la presentación, la lógica de negocio y el acceso a datos. Esta organización facilitó el desarrollo del CRUD, haciendo que cada componente tenga una responsabilidad clara y evitando el acoplamiento innecesario entre módulos.
2. La integración del patrón Singleton en el repositorio de datos garantizó la existencia de una única fuente de información, evitando inconsistencias y pérdida de datos causadas por múltiples instancias. Esto permitió que todas las capas trabajen con los mismos datos, mejorando la confiabilidad y estabilidad del sistema.
3. El uso combinado de los patrones MVC, NVC y Singleton fortaleció la calidad del diseño del software, facilitando su mantenimiento, escalabilidad y comprensión. Este trabajo permitió evidenciar que una buena arquitectura no solo mejora el orden del código, sino que también reduce errores y optimiza el proceso de desarrollo en aplicaciones reales.

10. Referencias

Alexander, S. (2019). Sumérgete en los patrones de diseño.

Geeksforgeeks. (16 de Abril de 2016). Singleton method design pattern. Obtenido de GeeksforGeeks: https://www.geeksforgeeks.org/system-design/singleton-design-pattern/

MDN. (13 de Noviembre de 2023). MVC - Glosario de MDN Web Docs. Obtenido de MDN Web Docs: https://developer.mozilla.org/es/docs/Glossary/MVC

Smith, S. (06 de Noviembre de 2024). Información general de ASP.NET Core MVC. Obtenido de Microsoft: https://learn.microsoft.com/es-es/aspnet/core/mvc/overview?view=aspnetcore-8.0

Toxboe, A. (12 de Mayo de 2025). Obtenido de UI-Patterns: https://ui-patterns.com/patterns