DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Ingeniería de Software

Análisis de Desarrollo de Software

Nombre:

Tipán Oswaldo

Fernando Sandoval

Simone Medina

Zaith Mandangón

NRC: 27835

Date: 25/11/2025

1. OBJETIVOS

Objetivo General

Aplicar la arquitectura de 3 capas (Modelo, Repositorio y Servicio) junto con el patrón MVC/NVC, para construir un sistema CRUD de Estudiantes implementado en JavaScript, siguiendo buenas prácticas de diseño y organización de software.

Objetivos Específicos

- Implementar el CRUD completo respetando la separación por capas.
- Explicar claramente la responsabilidad de cada capa (Modelo, Repositorio, Servicio, Vista).
- Construir un diagrama simplificado de la arquitectura MVC/NVC.
- Ejecutar y documentar las operaciones del CRUD: crear, listar, actualizar y eliminar.
- Implementar el patrón Singleton en el repositorio y analizar su impacto.
- Comparar MVC y Singleton, identificando cómo resuelven problemas distintos.
- Evaluar el mantenimiento, organización, ventajas, limitaciones y riesgos de ambas arquitecturas.

2. DESARROLLO

2.1 ARQUITECTURA DE 3 CAPAS (Modelo, Repositorio, Servicio)

Estructura de 3 capas:

```
src/
└── datos/
    ├── model/
    │   └── Estudiante.js
    └── repository/
        └── EstudianteRepository.js
└── logica_negocio/
    └── EstudianteService.js
└── presentacion/
    ├── EstudianteUI.js
    └── Main.js
```

Estructura desarrollada usando Javascript:

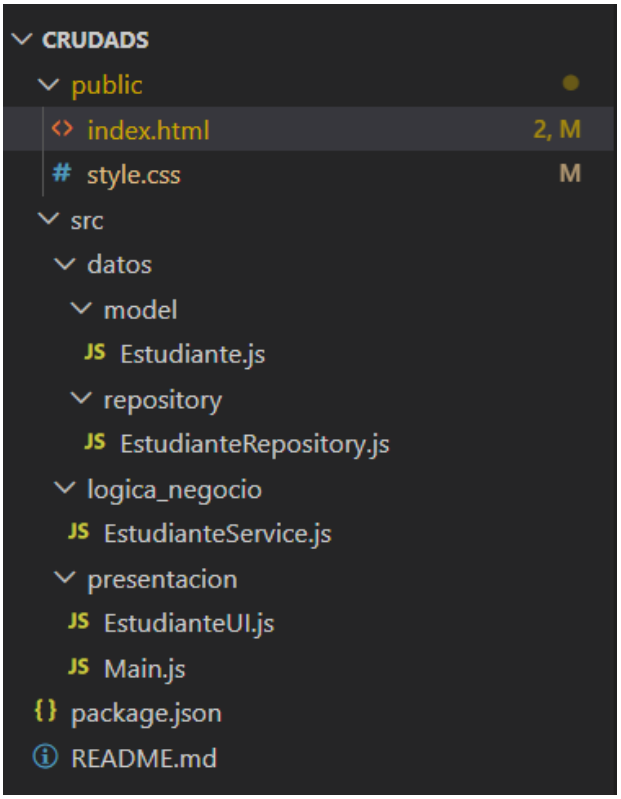


Figura 1. Estructura NVC

Descripción de capa capa:

a) Capa Modelo (Model)

Archivo: ***Estudiante.js***

Responsabilidad:

- Representa la entidad del negocio y asegura datos válidos.
- Define atributos: id, nombres, edad.
- Valida datos mediante el método static crear().
- Garantiza que solo existan estudiantes válidos.

b) Capa Repositorio (Datos)

Archivo: ***EstudianteRepository.js***

Responsabilidad:

- Manejar el almacenamiento interno de estudiantes.
- Ofrecer métodos CRUD: agregar, editar, eliminar, listar, buscarPorId.
- No contiene reglas de negocio, solo operaciones de persistencia.

c) Capa Servicio (Lógica de Negocio)

Archivo: ***EstudianteService.js***

Responsabilidad:

- Gestión de reglas de negocio.
- Validar duplicados.
- Validar existencia antes de editar/eliminar.
- Intermediario entre la vista y los datos.

d) Capa Vista / Controlador (UI / Presentación)

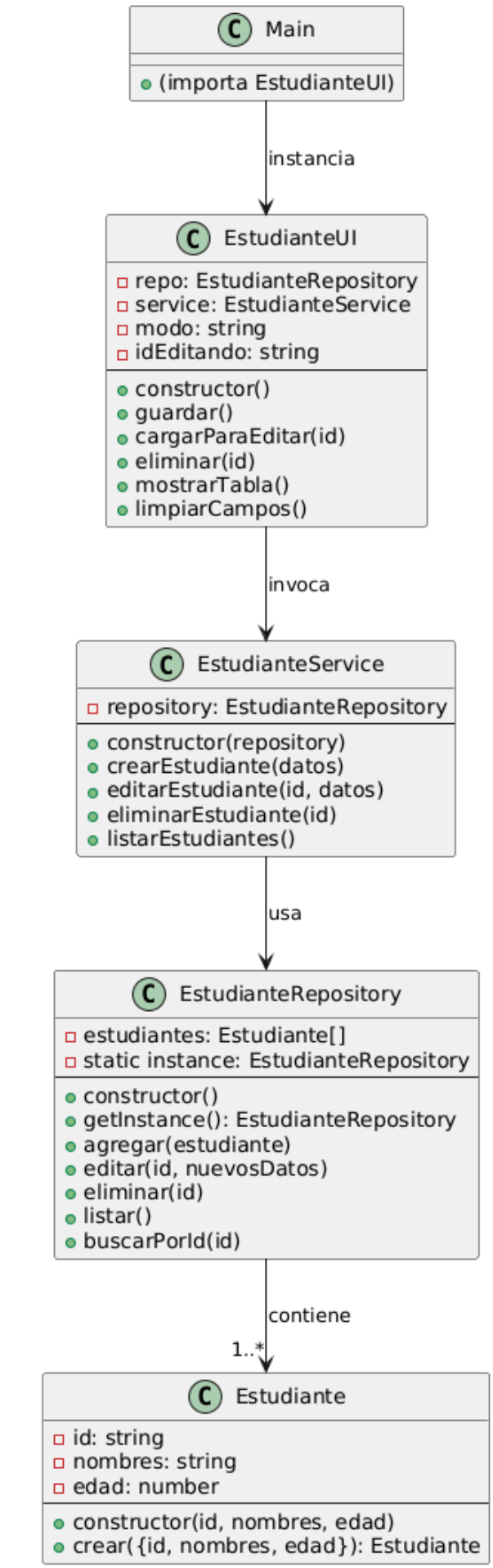
Archivo: ***EstudianteUI.js***

Responsabilidad:

- Recibir acciones del usuario.
- Enviar solicitudes al servicio.
- Mostrar resultados en HTML (tabla).
- Manejar modo “crear” y “editar”.

2.2 Diagrama del modelo MVC/NVC

UML - CRUD Estudiantes (MVC + Singleton)



2.3 Ejecución del CRUD

1. Crear Estudiante

- El usuario ingresa ID, nombres y edad.
- La UI llama a service.crearEstudiante()
- Si es válido, se agrega al repositorio.
- Se actualiza la tabla.

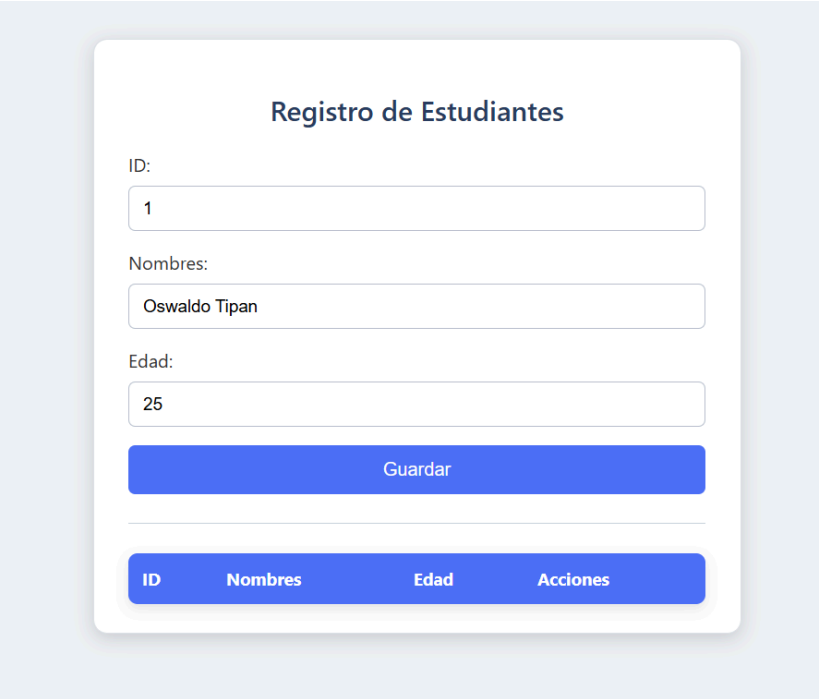


Figura 2. Crud Estudiante

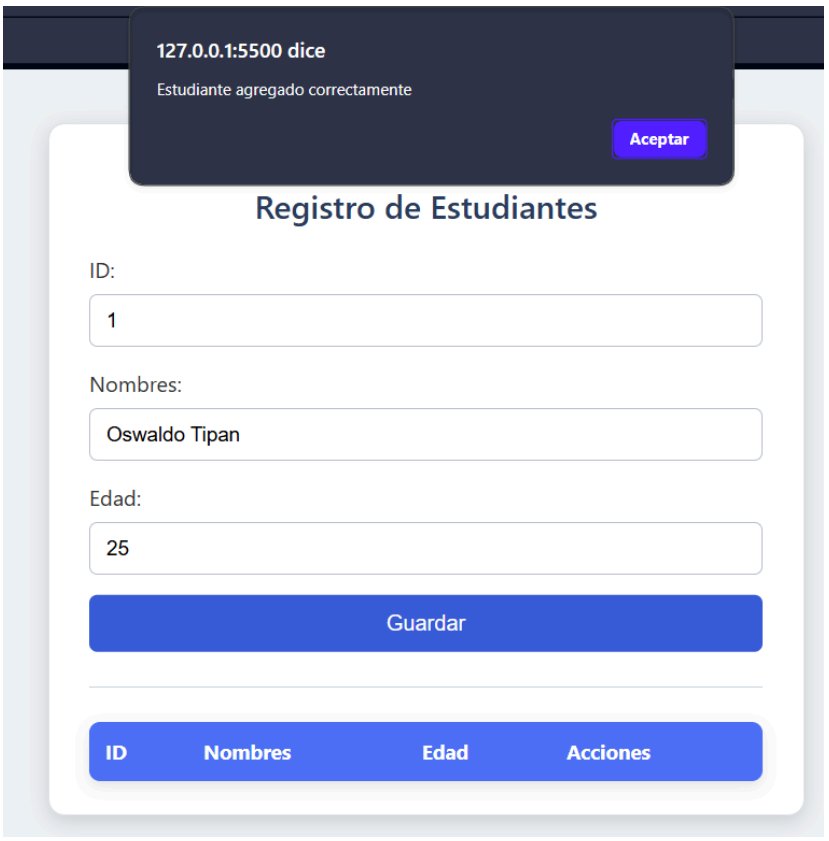


Figura 3. Estudiante Agregado

2. Listar Estudiantes

- La UI llama a service.listarEstudiantes()
- Se reconstruye la tabla HTML.

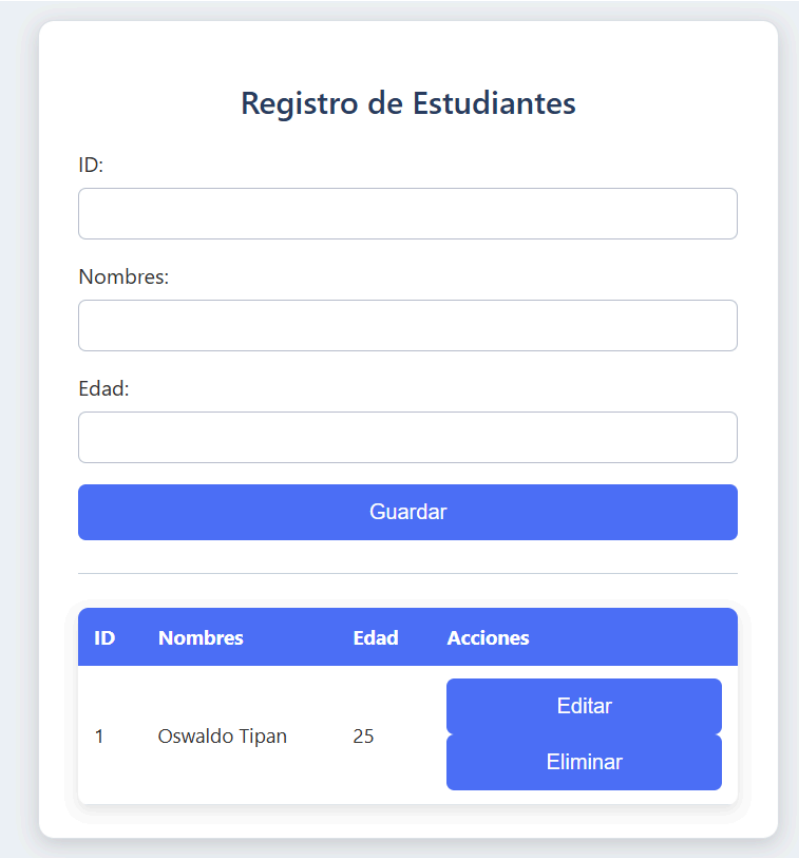


Figura 4. Estudiantes Listados

3. Editar Estudiante

- Al pulsar “Editar”, la UI carga datos en los inputs.
- Al guardar, se llama a service.editarEstudiante().
- El ID se bloquea (no editable).

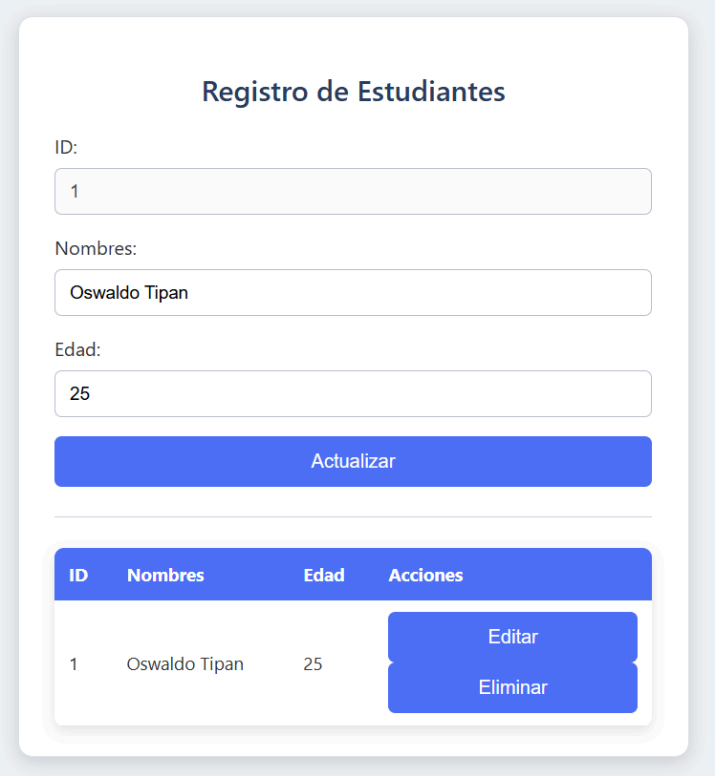


Figura 5. Editar Estudiante

Figura 6. Estudiante Editado

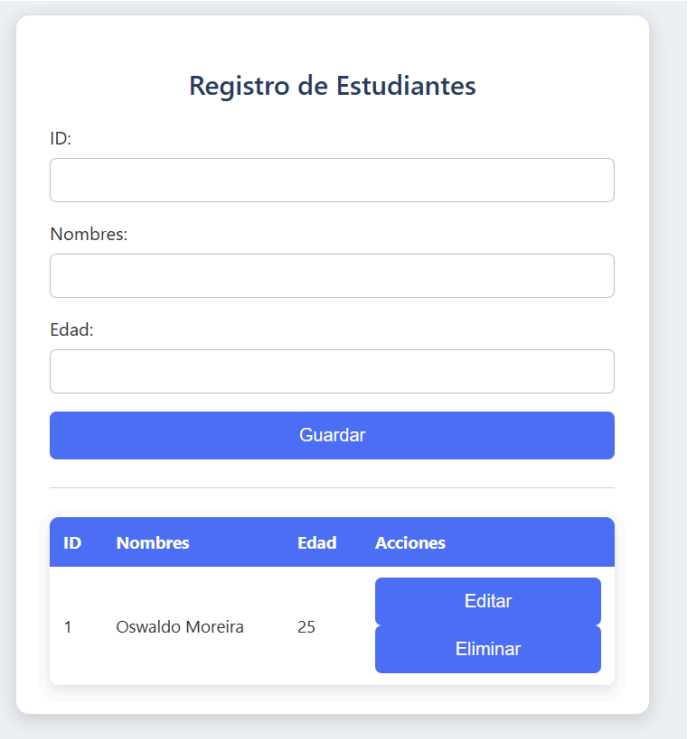


Figura 7. Resultado Estudiante Editado

4. Eliminar

- El botón “Eliminar” llama a service.eliminarEstudiante(id).
- Si existe, se borra y se actualiza la tabla.

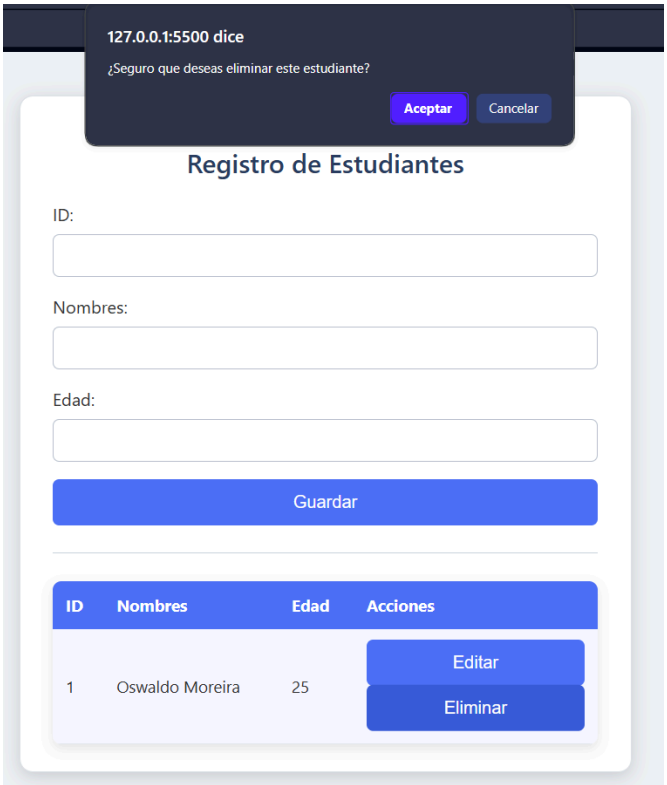


Figura 8. Eliminar Estudiante

Figura 9. Estudiante Eliminado

Figura 10. Resultado Estudiante Eliminado

2.4 ¿CÓMO MVC FACILITA EL MANTENIMIENTO?

- Separa responsabilidades, por lo que un cambio en la UI no afecta la lógica.
- Facilita pruebas unitarias sobre el servicio sin tocar el HTML.
- Permite cambiar la vista por React, Angular o consola sin modificar el backend.
- Reduce el acoplamiento y mejora la reutilización.

3. PATRÓN SINGLETON + NVC

3.1 Implementación del Singleton (Descripción Teórica)

EstudianteRepository.js

```
export class EstudianteRepository {
  constructor() {

    if (EstudianteRepository.instance) {
      return EstudianteRepository.instance;
    }

    this.estudiantes = [];
    EstudianteRepository.instance = this;
  }
  static getInstance() {
    if (!EstudianteRepository.instance) {
      EstudianteRepository.instance = new EstudianteRepository();
    }
    return EstudianteRepository.instance;
  }
  agregar(estudiante) {
    this.estudiantes.push(estudiante);
  }

  editar(id, nuevosDatos) {
    const index = this.estudiantes.findIndex(e => e.id === id);
    if (index === -1) return false;
    this.estudiantes[index] = { ...this.estudiantes[index], ...nuevosDatos };
    return true;
  }

  eliminar(id) {
      const originalLength = this.estudiantes.length;
      this.estudiantes = this.estudiantes.filter(e => e.id !== id);
    return this.estudiantes.length < originalLength;
  }

  listar() {
    return this.estudiantes;
  }

  buscarPorId(id) {
```

```
    return this.estudiantes.find(e => e.id === id);
  }
}
```

Descripción:

El patrón Singleton garantiza:

- Una sola instancia en toda la aplicación.
- Que cada vez que hagas new EstudianteRepository() se devuelva la misma instancia.
- Que la lista estudiantes sea compartida entre todos los controladores, vistas o servicios

Se comprueba si ya existe una instancia:

```
if (EstudianteRepository.instance) {
  return EstudianteRepository.instance;
}
```

Crea la instancia una sola vez:

```
this.estudiantes = [];
EstudianteRepository.instance = this;
```

Este método garantiza que todos los módulos usen la misma instancia:

```
static getInstance() {
  if (!EstudianteRepository.instance) {
    EstudianteRepository.instance = new EstudianteRepository();
  }
  return EstudianteRepository.instance;
}
```

3.2 ¿Cómo Singleton complementa NVC?

Singleton garantiza una única instancia del repositorio, evitando que la Vista, el Control y la Lógica de Negocio creen listas duplicadas. Gracias a esto, NVC mantiene su separación de responsabilidades sin perder consistencia en los datos.

- Mientras MVC separa vista, lógica de negocio y modelo,
- Singleton garantiza consistencia en la capa de datos.

Beneficios combinados:

- Datos centralizados
- Menos errores de duplicación

Persistencia en memoria controlada

3.3 Ejemplo de Persistencia Compartida

Dentro del Singleton:

```
if (EstudianteRepository.instance) {
  return EstudianteRepository.instance;
}
```

y

```
static getInstance() {
  if (!EstudianteRepository.instance) {
    EstudianteRepository.instance = new EstudianteRepository();
  }
  return EstudianteRepository.instance;
}
```

hacen que aunque crees varios controladores UI, todos compartirán la misma lista. Aunque en HTML se cree:

```
window.ui1 = new EstudianteUI();
window.ui2 = new EstudianteUI();
```

no existe problema dado que lo que se hace es:

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

4. COMPARACIÓN MVC VS SINGLETON

Criterio / Pregunta	Arquitectura MVC (Modelo-Vista-Controlador)	Patrón Singleton

¿Qué problema resuelve?	Resuelve el problema del código espagueti y el alto acoplamiento. Evita mezclar la lógica de negocio con la interfaz gráfica, permitiendo que la aplicación sea ordenada y escalable.	Resuelve el problema de la persistencia en memoria y la instanciación múltiple innecesaria. Evita que se creen múltiples listas de datos vacías cada vez que se llama al repositorio, garantizando una única fuente de verdad.
¿En qué capa se utiliza?	Es una estructura global que organiza todo el proyecto. Se manifiesta dividiendo el código en paquetes: model, view (presentación) y controller (o lógica de negocio).	Se utiliza específicamente en la Capa de Datos (Repository). Su función está restringida a la gestión del acceso a los datos almacenados.
¿Cómo influye en el mantenimiento?	Facilita enormemente el mantenimiento modular. Si necesitas cambiar el diseño (HTML/CSS), no rompes la lógica (JS). Permite que varios desarrolladores trabajen en capas distintas simultáneamente sin estorbarse.	Facilita el mantenimiento del acceso a recursos compartidos. Si necesitas cambiar la forma en que se inicializa la base de datos o la lista, solo modificas el método getInstance() y el cambio se refleja en toda la app automáticamente.
¿Cómo evita fallas de diseño?	Evita la creación de "Clases Dios" (clases gigantes que hacen todo). Previene que la interfaz gráfica manipule datos directamente sin pasar por validaciones de negocio.	Evita la inconsistencia de datos. Previene el error lógico donde una parte de la aplicación ve una lista de estudiantes y otra parte ve una lista diferente o vacía por haber instanciado un nuevo objeto.

5. CONCLUSIONES

- La arquitectura de 3 capas aplicada correctamente mejora la organización del sistema y facilita la evolución futura del CRUD.
- MVC/NVC permite separar completamente la vista del manejo de datos y reglas de negocio.

- El patrón Singleton garantiza una única instancia del repositorio, evitando pérdida de datos durante la ejecución.
- La integración de MVC + Singleton es adecuada para aplicaciones pequeñas o medianas que requieren mantener datos consistentes en memoria.
- El CRUD implementado cumple con los requerimientos del taller, mostrando estructura clara, operaciones funcionales y código mantenible.

6. RECOMENDACIONES

- Implementar pruebas unitarias para el servicio y el repositorio.
- Sustituir la persistencia en memoria por localStorage o una base de datos real.
- Migrar la interfaz a un framework moderno (React, Vue, Svelte) para escalar el sistema.
- Aplicar manejo de errores más descriptivo en la capa UI.
- Incorporar el patrón Observer para actualizar la vista automáticamente.