TALLER 1,2,3 - U2

1. DATOS INFORMATIVOS

Carrera: Ingeniería de Software

Asignatura: Análisis y Diseño de Software

Tema del taller: Taller del Patrón Singleton con Patrón NVC

Docente: Mgt. Jenny Alexandra Ruiz Robalino

Integrantes:

- Amaguaña Casa Kevin Fernando
- Bonilla Hidalgo Jairo Smith
- Guamán Pulupa Alexander Daniel
- Tipán Ávila Reishel Dayelin

Fecha: 24/11/2025 Paralelo: 27835

2. DESARROLLO

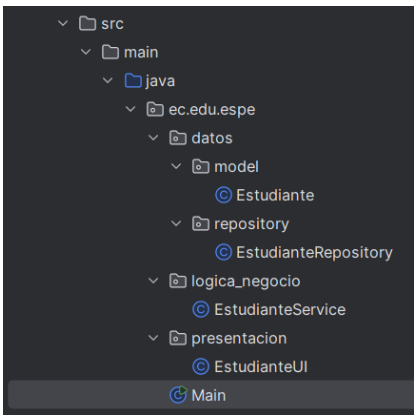
U2T1 – Taller de Arquitectura con GEMA (MVC – 3 Capas)

Objetivo del Taller

Comprender y aplicar la arquitectura de 3 capas (Modelo, Repositorio y Servicio), junto con el patron NVC para desarrollar un-CRUD de Estudiante (ID, Nombres, Edad), basado en el documento de Arquitectura con GEMA

1. **Implementa el CRUD del Estudiante respetando la separación por capas.**
   El proyecto desarrollado implementa un CRUD completo de Estudiantes siguiendo la arquitectura de 3 capas:
   - Modelo (Model)
   - Repositorio (Repository)
   - Servicio (Service – Lógica de Negocio)

   Y la arquitectura MVC mediante una interfaz GUI en Java Swing que actúa como la Vista y cuyo controlador actúa mediante eventos en la interfaz.
   El CRUD permite:
   - Crear estudiantes
   - Listar estudiantes
   - Editar estudiantes



- Eliminar estudiantes

Cada operación respeta el flujo: *UI → Service → Repository → Modelo.*

2. **Describe la responsabilidad de cada capa con tus propias palabras.**
   - **Capa Modelo (Model)**
     La capa Modelo se encarga de representar la información que maneja el sistema, es decir, las entidades del mundo real convertidas en clases. En mi caso, la clase Estudiante es la encargada de almacenar los datos esenciales como el ID, los nombres y la edad. Esta capa no contiene lógica complicada ni reglas; su único propósito es mantener los datos organizados y disponibles para que las demás capas puedan trabajar con ellos de manera confiable.

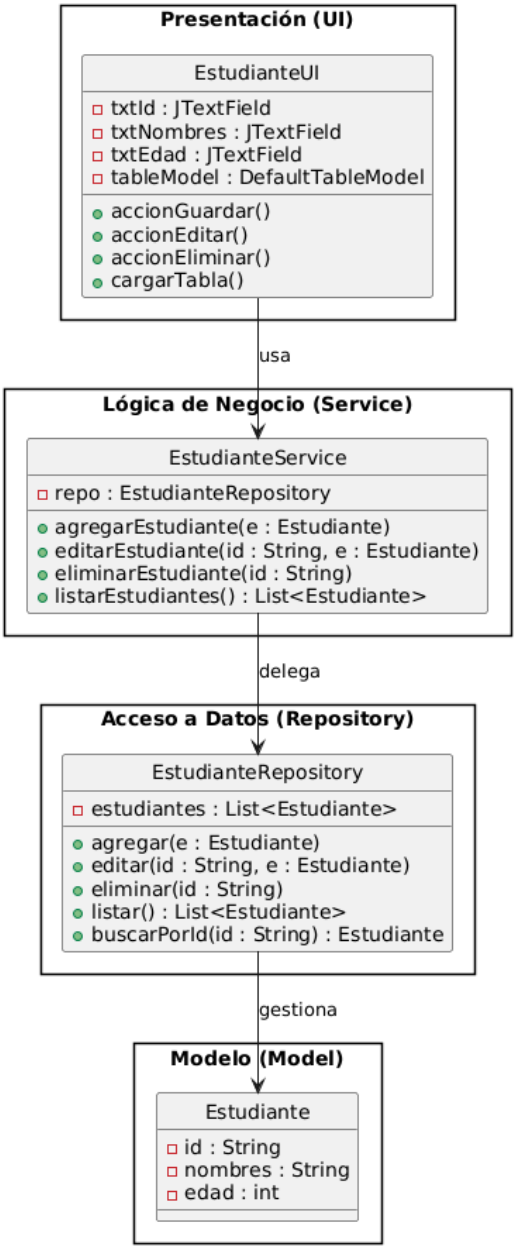
   - **Capa Repositorio (Repository)**
     La capa Repositorio es la responsable directa del manejo de los datos, sin importar de dónde provengan. En este proyecto, los datos se almacenan en una lista en memoria, pero lo importante es que esta capa define cómo se guardan, buscan, actualizan y eliminan los estudiantes. Esta capa actúa como si fuera una "mini base de datos" interna y oculta los detalles de almacenamiento para que otras capas no tengan que preocuparse por eso.

   - **Capa Servicio (Service – Lógica de Negocio)**
     La capa de Servicio funciona como el "cerebro" del sistema, ya que aquí se concentran las reglas que determinan cómo debe comportarse la aplicación. Antes de que cualquier información llegue al repositorio, el servicio se encarga de verificar que los datos cumplan las condiciones necesarias, como evitar IDs duplicados o edades incorrectas. Esta capa evita errores y garantiza que las operaciones del CRUD se realicen de forma correcta. Además, sirve como intermediaria entre la interfaz gráfica y el repositorio, manteniendo una separación clara entre lo que el usuario ve y lo que el sistema hace internamente.

   - **Capa Presentación (Vista – UI)**
     La capa de Presentación es el medio a través del cual el usuario interactúa con el sistema. En este proyecto, diseñé una interfaz gráfica en Java Swing donde se encuentran los formularios, botones y la tabla que muestran los estudiantes. Esta capa no contiene lógica de validación ni manejo de datos; su función principal es capturar la información del usuario, mostrar resultados y comunicar las acciones al Servicio. Es la parte más visual y la que conecta a las personas con el sistema.

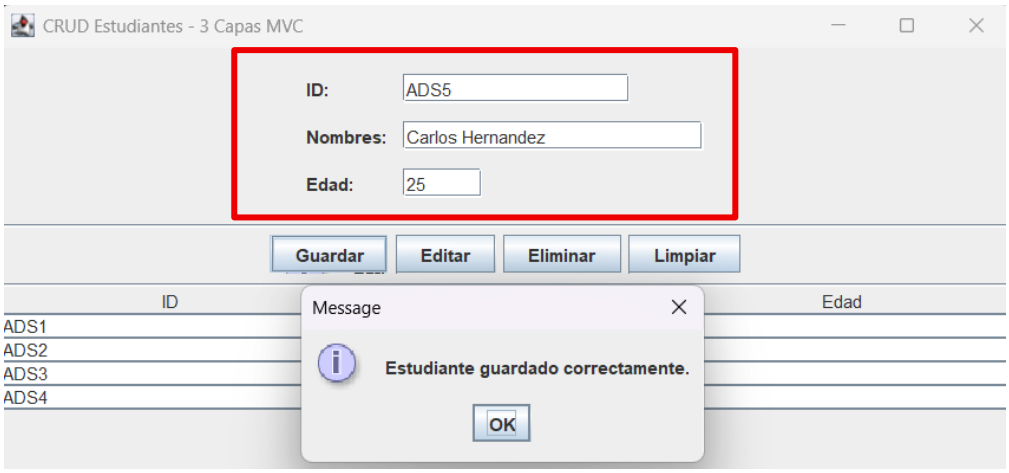
3. **Dibuja un diagrama simplificado de la arquitectura (a mano o UML).**

- La arquitectura está organizada en cuatro capas: Presentación, Servicio, Repositorio y Modelo.
- La UI captura las acciones del usuario y las envía al Servicio, que aplica las reglas de negocio.
- El Servicio delega el acceso y manipulación de datos al Repositorio, que administra la lista de estudiantes.
- Finalmente, el Modelo define la estructura de los datos que se intercambian entre todas las capas.

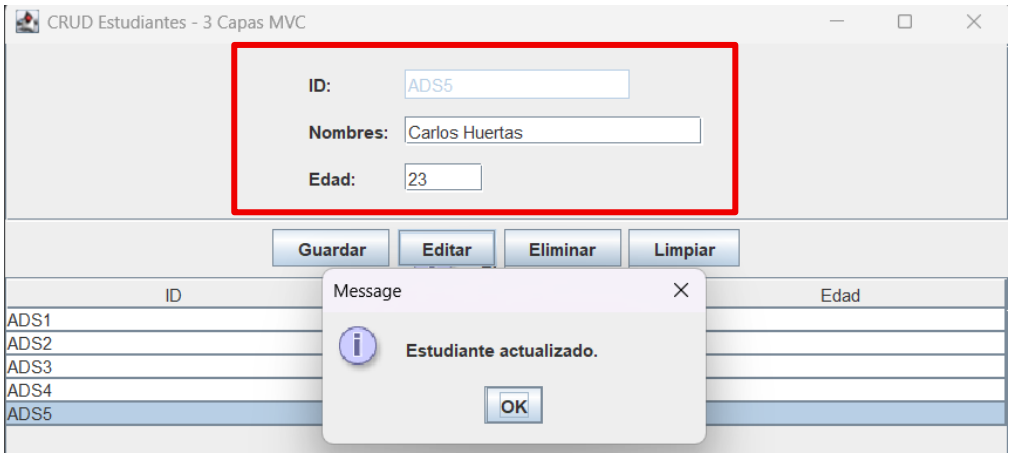


4. Ejecuta un ejemplo completo del CRUD: crear, actualizar, listar y eliminar.

- Crear



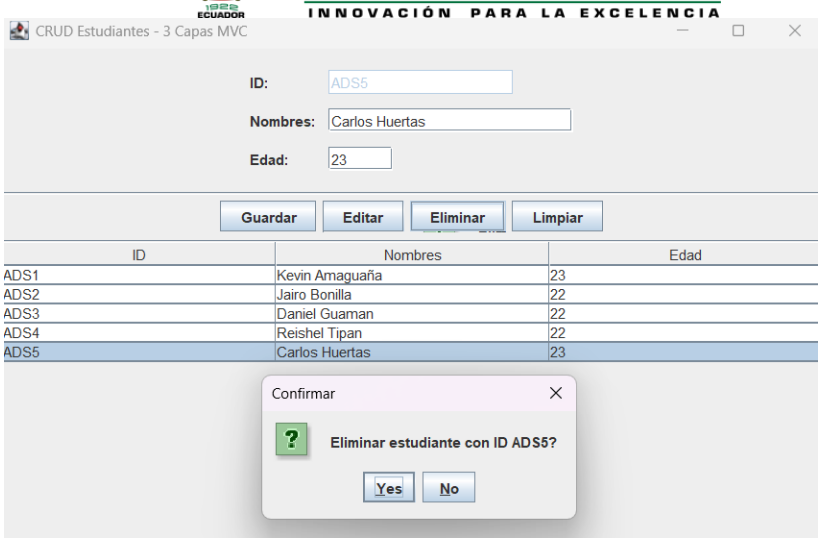
- Actualizar

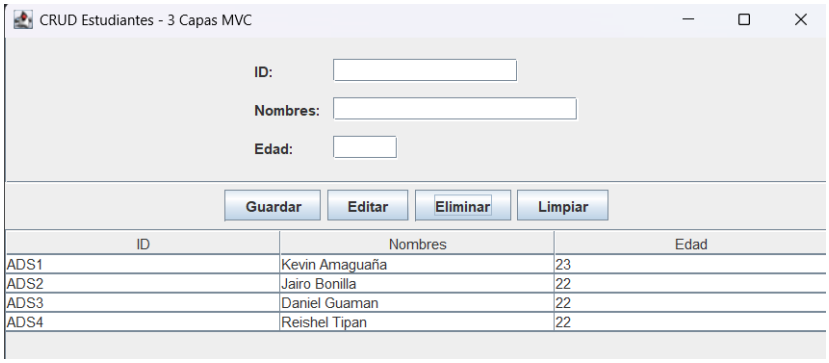


- Listar

ID	Nombres	Edad
ADS1	Kevin Amaguaña	23
ADS2	Jairo Bonilla	22
ADS3	Daniel Guaman	22
ADS4	Reishel Tipan	22
ADS5	Carlos Huertas	23

- Eliminar





5. **Explica cómo la arquitectura MVC facilita el mantenimiento del código.**

   La arquitectura MVC facilita el mantenimiento porque separa de forma clara las responsabilidades dentro del sistema ya que la interfaz gráfica solo se encarga de interactuar con el usuario, mientras que la lógica del negocio reside completamente en el Servicio, y los datos permanecen aislados en el Repositorio. Esto hace que modificar una parte no afecte a las otras; por ejemplo, si mañana quiero cambiar la forma de almacenar los estudiantes, solo debo modificar el repositorio sin tocar la interfaz. Esta separación modular ayuda a reducir errores, permite realizar pruebas más fácilmente y hace que el proyecto sea escalable, limpio y sencillo de comprender para otros desarrolladores.

U2T2 – Taller del Patrón Singleton con Patrón NVC

Objetivo del Taller

Comprender, aplicar e integrar el Patrón Singleton dentro de la capa de Datos y utilizar el patrón NVC (Negocio–Vista–Control) para estructurar correctamente una aplicación CRUD de Estudiante.

Instrucciones del Taller

1. **Implementa el Singleton en la clase EstudianteRepository asegurando una única instancia.**

Código:

```java
/**
 * Obtiene la instancia única del repositorio (Singleton)
 * @return la instancia única de EstudianteRepository
 */
public static EstudianteRepository getInstance() {
    if (instance == null) {
        synchronized (EstudianteRepository.class) {
            if (instance == null) {
                instance = new EstudianteRepository();
            }
        }
    }
    return instance;
}
```

Se realiza uso de getInstace() en la capa de servicio (EstudianteService)

```java
public class EstudianteService {
    private final EstudianteRepository repo;

    public EstudianteService() {
        this.repo = EstudianteRepository.getInstance();
    }
```

Uso de EstudianteService en el controlador

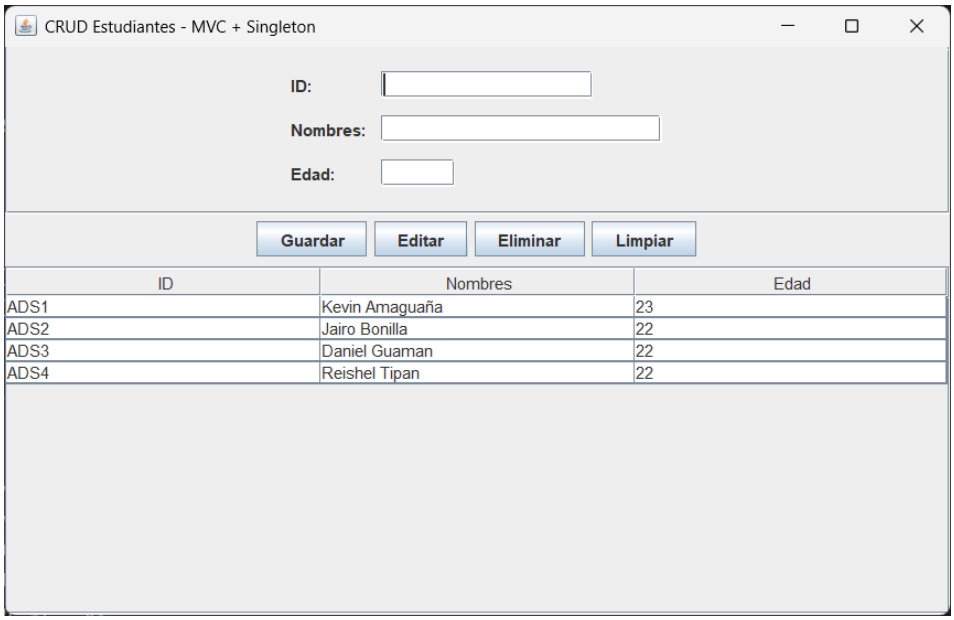
```java
public class EstudianteControlador {
    private final EstudianteService service;
    private final String identificador;

    public EstudianteControlador(String identificador) {
        this.service = new EstudianteService();
        this.identificador = identificador;
        System.out.println("[Controlador " + identificador + "] Creado y conectado al servicio.");
    }

```

2. Realiza pruebas demostrando que la persistencia en memoria es compartida gracias al Singleton.

Creando múltiples controladores y verifica que comparten la misma lista

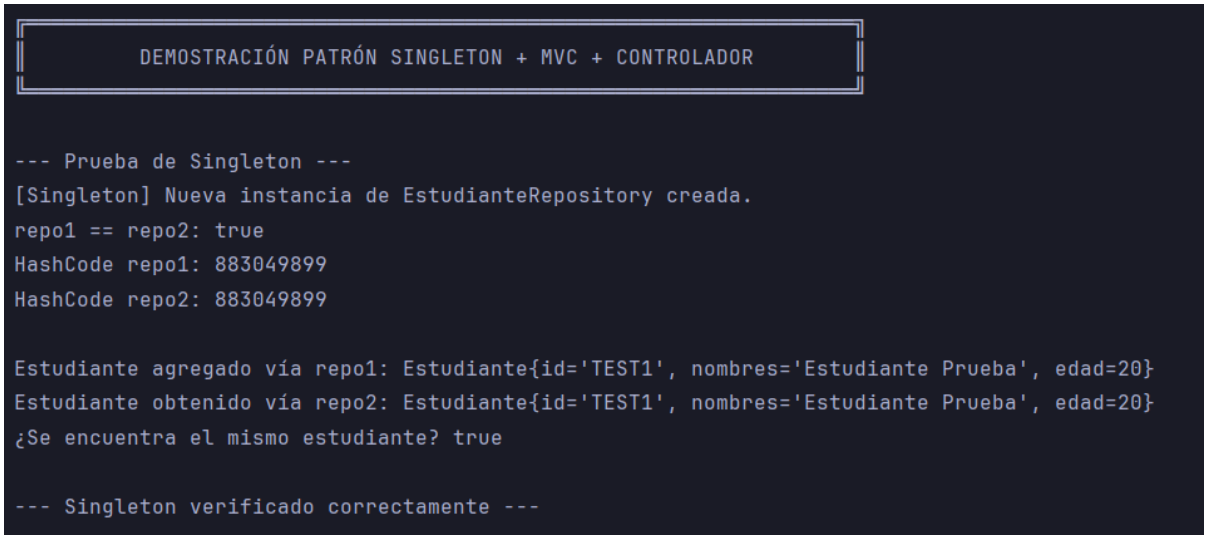
```
--- Prueba de Múltiples Controladores ---
[Controlador 1] Creado y conectado al servicio.
[Controlador 2] Creado y conectado al servicio.
[Controlador 1] CREAR: Estudiante{id='DEMO1', nombres='Estudiante Demo', edad=25} - Éxito
[Controlador 2] BUSCAR: Encontrado - Estudiante{id='DEMO1', nombres='Estudiante Demo', edad=25}
¿Controlador 2 ve estudiante de Controlador 1? ✓ SÍ
[Controlador 1] ELIMINAR: ID=DEMO1 - Éxito
--- Controladores verificados correctamente ---
```

Ejecución

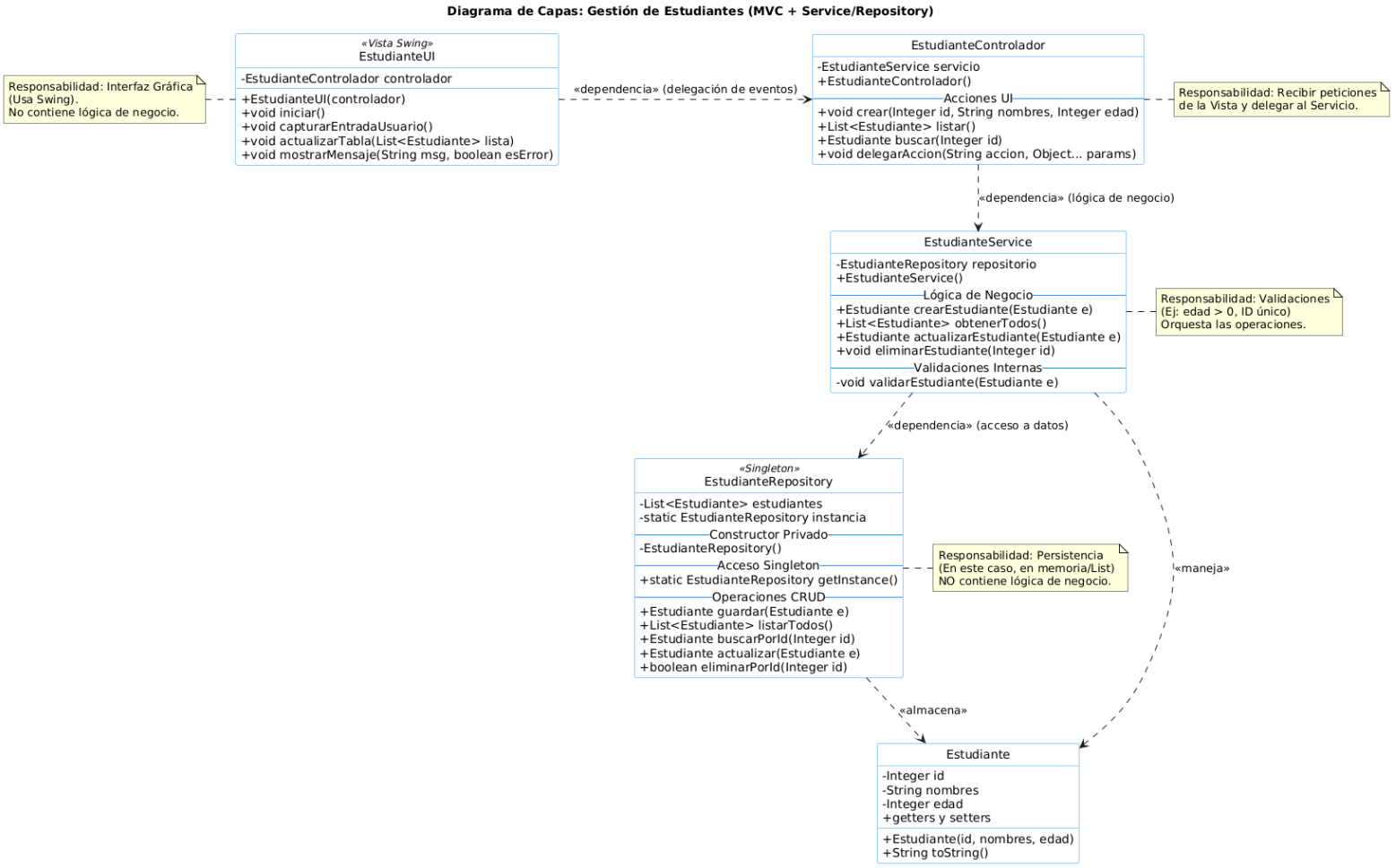


3. **Explica cómo Singleton complementa a NVC y evita pérdida de datos en el repositorio.**
   El patrón Singleton garantiza que existe una única instancia del EstudianteRepository en toda la aplicación, evitando la pérdida de datos en memoria. Cuando múltiples componentes (EstudianteService, EstudianteUI, u otras vistas) llaman a EstudianteRepository.getInstance(), todos obtienen la misma referencia al repositorio, compartiendo así el mismo HashMap<String, Estudiante> donde se almacenan los datos. Sin Singleton, cada new EstudianteRepository() crearía una instancia separada con su propia colección de datos, provocando que los estudiantes agregados en un servicio no sean visibles en otro, fragmentando la información, lo cual es fundamental en donde múltiples controladores y vistas interactúan con el mismo modelo de datos.

4. **Evidencia de persistencia compartida.**



5. Diagrama NVC mostrando separación de responsabilidades.



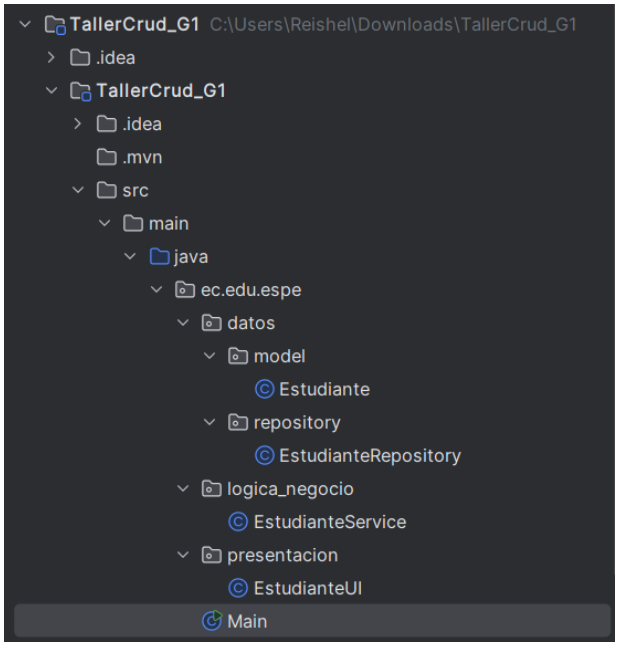
U2T3 – Taller Parte A y B: Arquitectura MVC y Patrón Singleton

Parte A: Arquitectura con MVC

Objetivo: Aplicar la arquitectura Modelo–Vista–Controlador en un CRUD de estudiante (ID, Nombres, Edad) basado en los documentos U2T1 y U2T2.

Instrucciones:

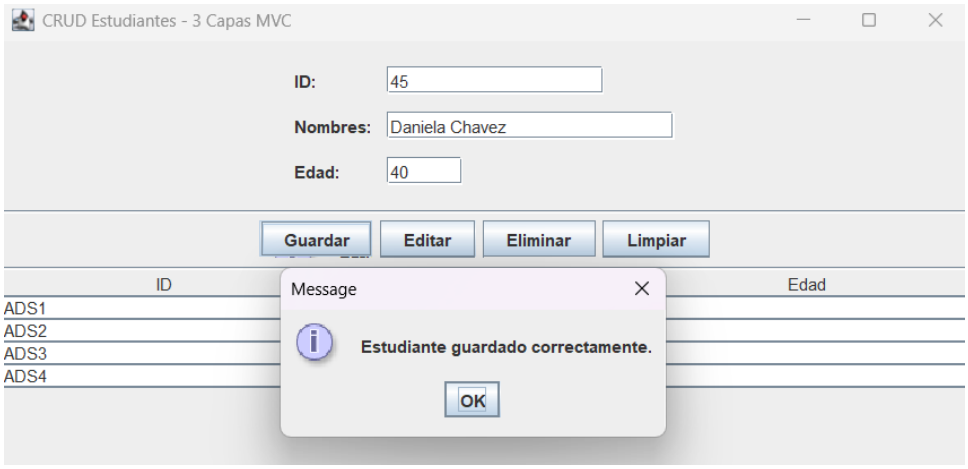
1. **Analiza la estructura de 3 capas: Modelo, Repositorio y Servicio.**
   La estructura de 3 capas en el CRUD desarrollado permite separar claramente el modelo, la lógica de negocio y la interfaz visual. La capa Modelo está representada por la clase Estudiante, que contiene los atributos y métodos básicos de la entidad. La capa Servicio se encarga de aplicar las reglas de negocio, validar los datos y coordinar las operaciones del CRUD. Finalmente, la Vista (UI con Swing) gestiona la interacción con el usuario, capturando entradas y mostrando resultados. Esta división hace que el sistema sea más ordenado, fácil de entender y mantenible.



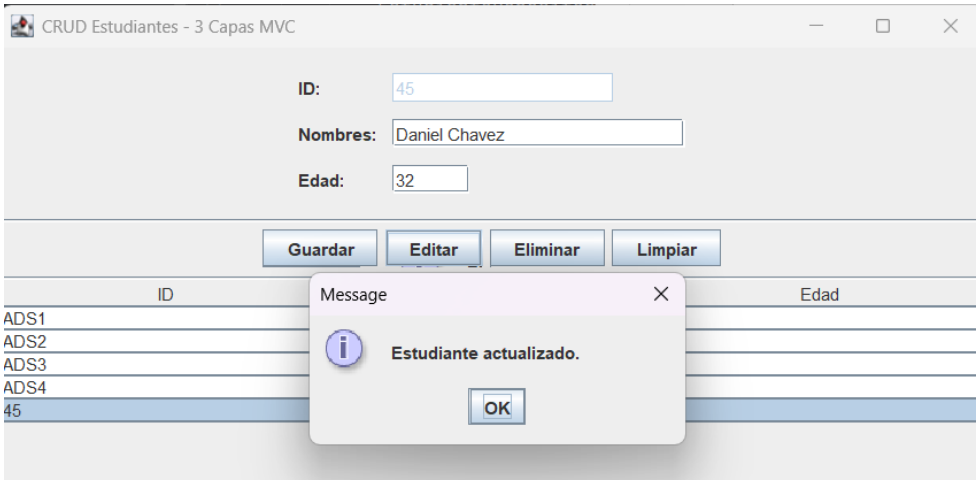
2. **Identifica responsabilidades de cada capa.**
   Cada capa del MVC cumple un rol específico: el Modelo almacena la información del estudiante sin lógica adicional; el Servicio procesa solicitudes, controla la coherencia de los datos y comunica acciones al repositorio; el Controlador actúa como puente entre la Vista y la lógica, mientras que la Vista únicamente presenta información al usuario. Gracias a esta separación, el sistema permite modificar la forma en que se muestra la interfaz sin afectar la lógica ni los datos internos.

3. Ejecuta un-CRUD estudiantil con NVC.

- Crear

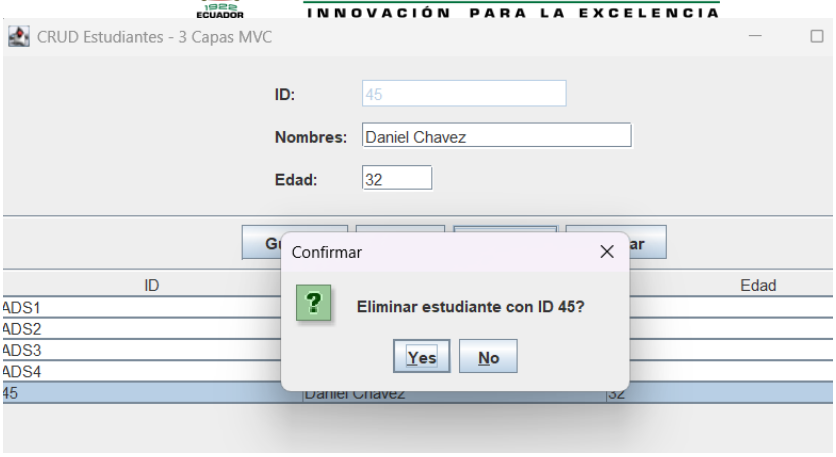


- Actualizar



- Listar

- Eliminar



4. **Explica cómo NVC favorece separación de responsabilidades.**
   El patrón NVC facilita la separación de responsabilidades porque divide el sistema en partes independientes que trabajan en conjunto. La Vista no contiene lógica ni reglas del negocio, simplemente muestra datos y captura eventos; el Controlador recibe esas acciones y las envía al Servicio, evitando que la Vista dependa directamente de la lógica. El Servicio valida y gestiona las operaciones sin preocuparse por cómo se muestran los resultados. Esta estructura modular permite reemplazar la UI, agregar nuevas reglas o cambiar el repositorio sin afectar a las demás capas, lo cual mejora el mantenimiento y reduce errores.

Parte B: Arquitectura con Singleton

Objetivo: Implementar Singleton en la capa de datos para evitar múltiples instancias del repositorio, basado en el documento U2T2.

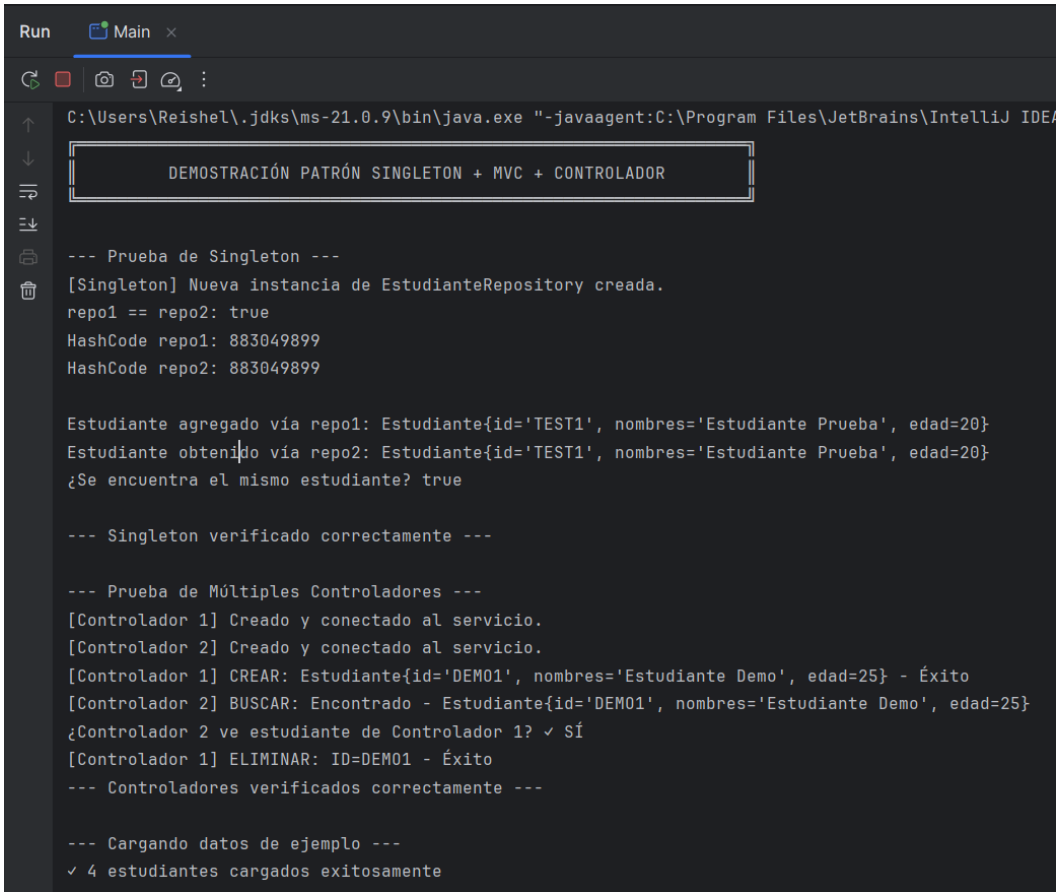
Instrucciones:

1. **Revisar la clase EstudianteRepository transformada en Singleton.**

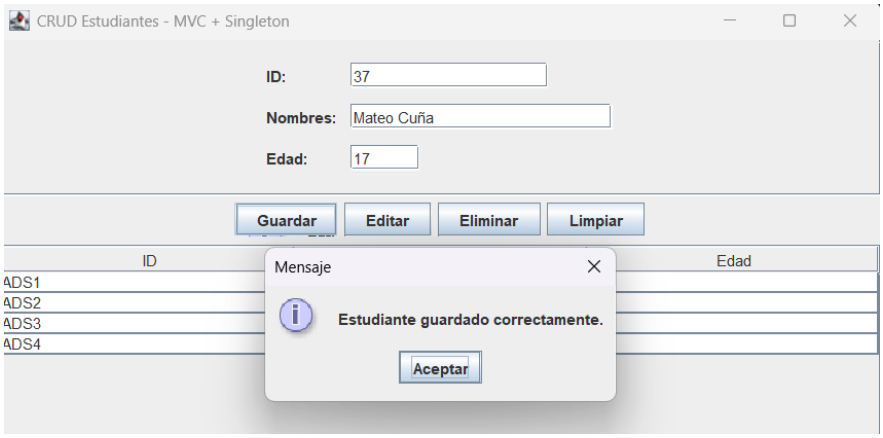
```
private EstudianteRepository() {  1 usage
    System.out.println("[Singleton] Nueva instancia de EstudianteRepository creada.");
}

public static EstudianteRepository getInstance() {  3 usages
    if (instance == null) {
        synchronized (EstudianteRepository.class) {
            if (instance == null) {
                instance = new EstudianteRepository();
            }
        }
    }
    return instance;
}
```

La clase *EstudianteRepository* fue transformada en un Singleton para garantizar que exista una sola instancia del repositorio durante toda la ejecución. Esto permite que diferentes controladores, servicios o interfaces compartan exactamente la misma lista de estudiantes, evitando duplicación de datos y manteniendo consistencia en el CRUD. La implementación incluye doble verificación con *synchronized* para evitar problemas en escenarios concurrentes.

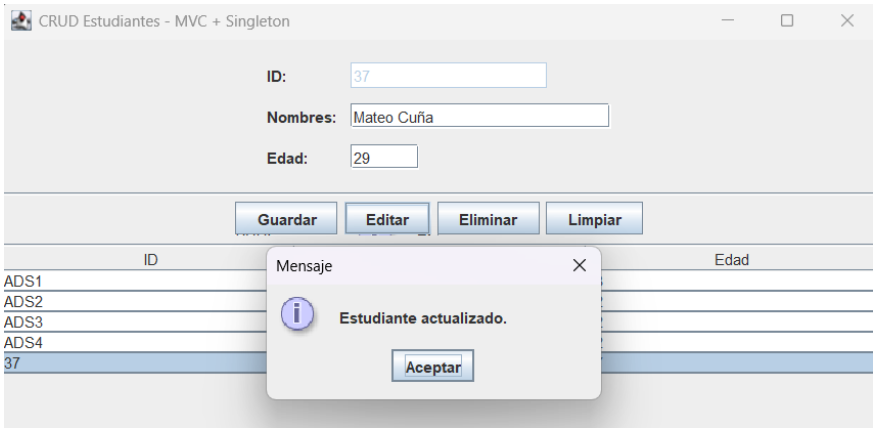
2. **Ejecutar el CRUD garantizando una única lista compartida.**
    - Al ejecutar el CRUD con el repositorio Singleton se observa que todas las operaciones CRUD afectan a la misma colección interna. Esto se demuestra creando dos referencias distintas al repositorio y verificando que ambas comparten la misma instancia y lista de estudiantes. Cualquier estudiante agregado desde repo1 es visible inmediatamente en repo2, confirmando la persistencia compartida.



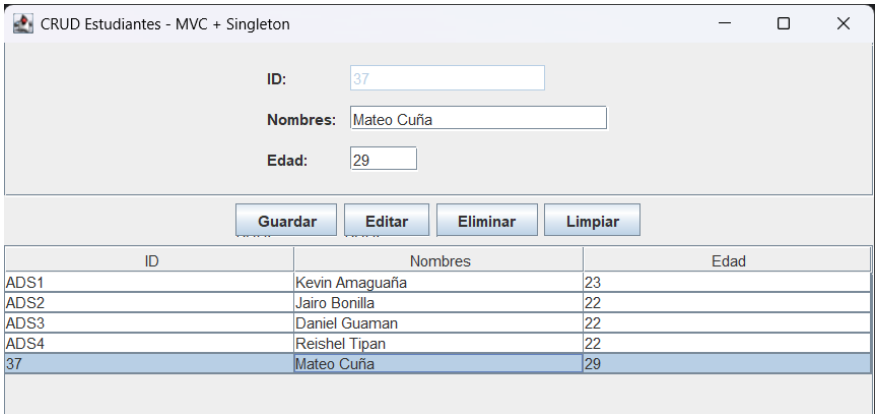
- **Crear**



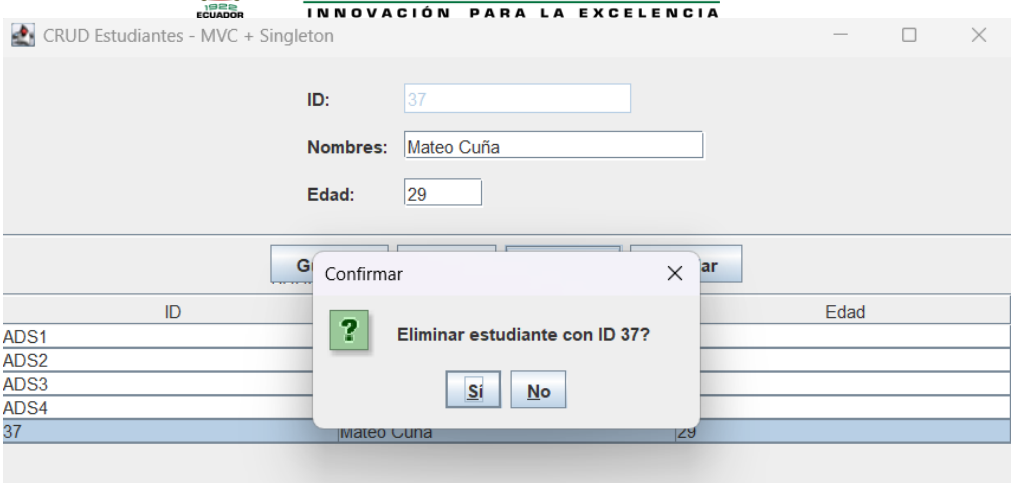
- **Actualizar**



- **Listar**



- **Eliminar**



3. **Comparar resultados con NVC sin Singleton.**
   En la versión sin Singleton, cada vez que se creaba un Service o un Controlador nuevo, se generaba un repositorio distinto con listas distintas. Esto provocaba inconsistencia, ya que un controlador podía crear un estudiante que no se reflejara en otra interfaz o controlador. Con Singleton, la fuente de datos es única, evitando fragmentación y pérdida de registros.

4. **Explicar el impacto en la persistencia de datos.**
   El uso del Singleton tiene un impacto directo en la persistencia en memoria, ya que garantiza que toda la aplicación maneje datos de un solo repositorio compartido. Esto evita duplicaciones, mantiene la coherencia entre controladores y permite que la UI acceda a información centralizada. Aunque la persistencia es volátil (se pierde al cerrar la aplicación), el patrón es fundamental para aplicaciones educativas o prototipos basados en memoria.

Actividad Integrada

Realiza un cuadro comparativo entre MVC y Singleton:

Aspecto	MVC (Modelo–Vista–Controlador)	Singleton
¿Qué problema resuelve?	Organiza el sistema separando datos, vista y lógica; evita mezclas de responsabilidades.	Evita múltiples instancias del repositorio; garantiza un solo punto de acceso a los datos.
¿En qué capa se utiliza?	Vista, Controlador y Servicio (flujo completo de la aplicación).	Capa de Datos (Repositorio).

¿Cómo influye en el mantenimiento?	Facilita cambios en UI, lógica o datos sin afectar las demás capas.	Reduce errores por duplicación de datos; asegura consistencia.
¿Cómo evita fallas de diseño?	Evita que la UI contenga lógica o validaciones.	Evita listas duplicadas, pérdida de datos y múltiples fuentes de verdad.
Ejemplo en el proyecto	EstudianteUI, EstudianteControlador, EstudianteService.	EstudianteRepository.getInstance() con instancia única.

3. CONCLUSIONES

- La aplicación de la arquitectura MVC permitió una separación clara de responsabilidades, lo cual facilitó el desarrollo de los tres talleres. Cada capa cumplió un rol específico: el Modelo gestionó los datos, el Servicio procesó la lógica de negocio y la Vista manejó la interacción con el usuario, logrando una estructura limpia, ordenada y fácil de mantener.
- El uso del patrón Singleton fortaleció la persistencia en memoria, ya que garantizó que toda la aplicación compartiera una única instancia del repositorio. Esto evitó problemas de fragmentación de datos, inconsistencias entre controladores y pérdida de información cuando se manejaban múltiples componentes simultáneamente.
- La integración de ambos patrones, MVC y Singleton, resultó efectiva y coherente, demostrando que combinar una arquitectura modular con un mecanismo de instancia única permite desarrollar un CRUD robusto, escalable y con buen flujo de datos entre capas. Además, se comprobó mediante pruebas que los controladores comparten correctamente el mismo estado del repositorio.
- Los tres talleres contribuyeron a una comprensión profunda del diseño de software, ya que permitieron aplicar teoría en escenarios prácticos y progresivos. Desde la estructura base en 3 capas, pasando por la incorporación del controlador, hasta la implementación de persistencia compartida, cada etapa reforzó principios clave: desacoplamiento, modularidad, reutilización y claridad arquitectónica.

4. RECOMENDACIONES

- Migrar el repositorio en memoria hacia una base de datos real, utilizando tecnologías como MySQL, PostgreSQL o MongoDB, manteniendo el patrón Singleton únicamente para servicios o configuraciones globales. Esto permitiría persistencia permanente entre ejecuciones y una mayor escalabilidad.
- Implementar pruebas unitarias e integración para cada capa, especialmente en el Servicio y el Repositorio. Esto garantiza que las reglas de negocio, validaciones y operaciones

CRUD mantengan su consistencia incluso cuando se agreguen nuevas funcionalidades o se realicen refactorizaciones en el sistema.

5. REFERENCIAS

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley Professional.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Desig*n. Prentice Hall.