TALLER 4 - U2

1. DATOS INFORMATIVOS

Carrera: Ingeniería de Software

Asignatura: Análisis y Diseño de Software

Tema del taller: Taller de Implementación de patrones de diseño a MVC

Docente: Mgt. Jenny Alexandra Ruiz Robalino

Integrantes:

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

Fecha: 27/11/2025 Paralelo: 27835

2. DESARROLLO

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

Parte A: Arquitectura con Singleton + Factory

Objetivo: Implementar un CRUD aplicando el patrón MVC combinado con Singleton y Factory para garantizar una arquitectura organizada, una única instancia del repositorio y una creación estandarizada de objetos.

Instrucciones:

1. **Revisar la clase EstudianteRepository transformada en Singleton y su integración con Factory.**
   El repositorio EstudianteRepository está implementado como Singleton para asegurar que exista una sola instancia en toda la aplicación. Esto se logra usando un constructor privado, una variable estática y el método getInstance(). Gracias a esto, todos los controladores, servicios y vistas comparten la misma lista de estudiantes.
   Por otro lado, el patrón Factory se integra encargándose de crear objetos Estudiante validados antes de ser enviados al servicio y luego al repositorio. Así, el repositorio Singleton solo recibe objetos construidos correctamente por el Factory, lo que evita datos defectuosos.
   - Método en el repository

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

/**
 * Obtiene la instancia única del repositorio (Singleton)
 * @return la instancia única de EstudianteRepository
 */
public static EstudianteRepository getInstance() {  1 usage
    if (instance == null) {
        synchronized (EstudianteRepository.class) {
            if (instance == null) {
                instance = new EstudianteRepository();
            }
        }
    }
    return instance;
}
```

- Método en el model

```java
public class EstudianteFactory {  5 usages

    /** Crea un estudiante con validaciones centralizadas ...*/
    public static Estudiante crearEstudiante(String id, String nombres, int edad) {  3 usages
        // Validación de ID
        if (id == null || id.trim().isEmpty()) {
            throw new IllegalArgumentException("El ID no puede estar vacío");
        }
        if (id.trim().length() < 2) {
            throw new IllegalArgumentException("El ID debe tener al menos 2 caracteres");
        }

        // Validación de nombres
        if (nombres == null || nombres.trim().isEmpty()) {
            throw new IllegalArgumentException("Los nombres no pueden estar vacíos");
        }
        if (nombres.trim().length() < 3) {
            throw new IllegalArgumentException("Los nombres deben tener al menos 3 caracteres");
        }
```

2. **Ejecutar el CRUD asegurando una única lista compartida y estudiantes creados mediante Factory.**

   En esta implementación, todas las operaciones del CRUD trabajan sobre una única lista porque el repositorio utiliza el patrón Singleton, esto significa que, sin importar cuántos controladores o servicios existan, todos apuntan a la misma instancia del repositorio y comparten exactamente los mismos datos en memoria. Además, cada estudiante es creado mediante el Factory, lo que asegura que antes de ingresar al CRUD los datos pasen por una validación centralizada y que todos los objetos se construyan de manera uniforme.

   - Desde el Service (demuestra Singleton):

```java
public class EstudianteService {  6 usages
    private final EstudianteRepository repo;  7 usages

    public EstudianteService() {  2 usages
        this.repo = EstudianteRepository.getInstance();
    }
```

- Desde el Controlador (demuestra Factory):

```
public boolean accionCrear(String id, String nombres, int edad) {  1 usage
    try {
        // Usar Factory en lugar de new Estudiante()
        Estudiante estudiante = EstudianteFactory.crearEstudiante(id, nombres, edad);
        boolean resultado = service.agregarEstudiante(estudiante);
        System.out.println("[Controlador " + identificador + "] CREAR: " + estudiante + " - Éxito");
        return resultado;
    } catch (IllegalArgumentException e) {
        System.err.println("[Controlador " + identificador + "] CREAR: Error - " + e.getMessage());
        throw e;
    }
}
```

- Ejecución

```
Run    Main
C:\Users\Reishel\.jdks\ms-21.0.9\bin\java.exe "-javaagent:C:\Program F
--- Cargando datos de ejemplo desde archivo ---
[Singleton] Nueva instancia de EstudianteRepository creada.
  ✓ Cargado: Estudiante{id='ADS1', nombres='Kevin Amaguaña', edad=23}
  ✓ Cargado: Estudiante{id='ADS2', nombres='Jairo Bonilla', edad=22}
  ✓ Cargado: Estudiante{id='ADS3', nombres='Daniel Guaman', edad=22}
  ✓ Cargado: Estudiante{id='ADS4', nombres='Reishel Tipan', edad=22}
✓ 4 estudiantes cargados exitosamente
--- Datos cargados ---

[Controlador UI-Controlador] Creado y conectado al servicio.
[Controlador UI-Controlador] LISTAR: 4 estudiantes encontrados
```

La salida demuestra que el repositorio Singleton se creó una sola vez y que toda la aplicación trabaja sobre la misma lista compartida. Los estudiantes cargados desde archivo aparecen inmediatamente cuando el controlador los lista, confirmando que las operaciones CRUD usan la misma instancia del repositorio, lo que garantiza consistencia en los datos.

3. **Comparar los resultados del CRUD con MVC sin Singleton y sin Factory.**
   En un CRUD normal sin Singleton, cada controlador o servicio tendría su propio repositorio, por lo que los datos no serían compartidos y cada ventana o proceso tendría listas diferentes. Además, sin Factory, los objetos Estudiante se crearían manualmente con new, repitiendo validaciones o incluso permitiendo la creación de datos incorrectos.
   En cambio, en el CRUD implementado con Singleton y Factory, la lista es única y compartida, lo que garantiza coherencia entre vistas. Y el Factory centraliza la creación de estudiantes, evitando errores y manteniendo la validación en un solo lugar.

4. **Explicar el impacto en la persistencia de datos y en la creación de objetos.**
   El impacto principal es que el sistema se vuelve más consistente y seguro ya que gracias al Singleton, todos los cambios realizados durante la ejecución afectan a una sola lista, lo que

garantiza que no existan duplicados o listas paralelas con datos distintos, por ende, esto mejora la persistencia temporal en memoria y asegura integridad.

Por otra parte, el uso del Factory mejora la creación de objetos, ya que cada Estudiante se construye con validaciones previas, asegurando que no ingresen datos incompletos o incorrectos al sistema. La combinación de ambos patrones genera un CRUD más limpio, ordenado y fácil de mantener.

Parte B: Arquitectura con Singleton + Observer

Objetivo: Integrar el patrón Singleton con Observer en el CRUD para asegurar una única fuente de datos y notificaciones automáticas a las vistas cuando ocurren cambios en el repositorio.

Instrucciones:

1. **Revisar la clase EstudianteRepository transformada en Singleton e integrada con Observer.**

   La clase EstudianteRepository mantiene la implementación de Singleton para asegurar que solo exista una instancia global donde se almacena la lista de estudiantes. Esto garantiza que todas las ventanas del sistema y los controladores trabajen sobre la misma fuente de datos.

   El Observer no se implementa directamente en el repositorio, sino en el Servicio, lo cual es correcto porque la capa de negocio debe coordinar las notificaciones. Sin embargo, la integración es completa porque el repositorio se usa dentro del Servicio Singleton, y cualquier cambio realizado aquí es propagado automáticamente a todas las vistas a través del patrón Observer.

```
/**
 * Notifica a todos los observadores sobre un cambio
 * @param tipoEvento Tipo de evento ocurrido
 * @param estudiante Estudiante afectado
 */
private void notificarObservadores(TipoEvento tipoEvento, Estudiante estudiante) {  3 usages
    for (EstudianteObserver observador : observadores) {
        observador.actualizar(tipoEvento, estudiante);
    }
}
```

2. **Revisar la clase EstudianteService transformada en Singleton para mantener consistencia con la UI**

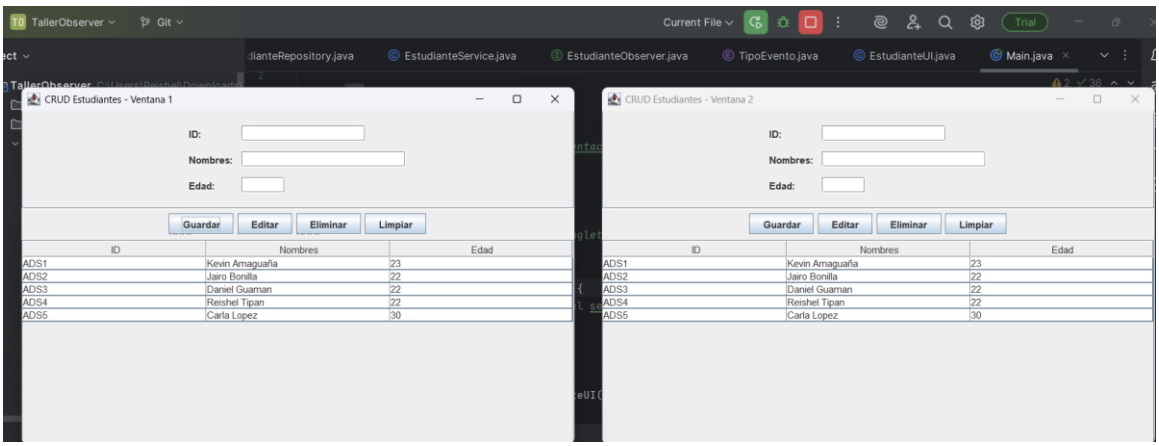
   La clase EstudianteService mantiene la implementación de Singleton para asegurar que la UI mantenga persistencia de cambios entre múltiples ventanas abiertas simultáneamente. Al utilizar el patrón Singleton, todas las interfaces gráficas comparten la misma instancia del servicio y, por lo tanto, la misma lista de observadores registrados. Esto garantiza que cualquier operación CRUD (crear, actualizar o eliminar) realizada en una ventana notifique automáticamente a todas las demás ventanas a través del patrón Observer implementado, manteniendo así la sincronización de datos en tiempo real.. El patrón Singleton evita la creación de múltiples instancias del servicio que tendrían listas de observadores independientes y, por ende, no podrían comunicarse entre sí. De esta manera, se asegura la consistencia de la información mostrada en todas las interfaces activas del sistema.

```java
/**
 * Obtiene la instancia única del servicio (Singleton)
 * @return la instancia única de EstudianteService
 */
public static EstudianteService getInstance() {  3 usages  AlexDaniel593
    if (instance == null) {
        synchronized (EstudianteService.class) {
            if (instance == null) {
                instance = new EstudianteService();
            }
        }
    }
    return instance;
}
```

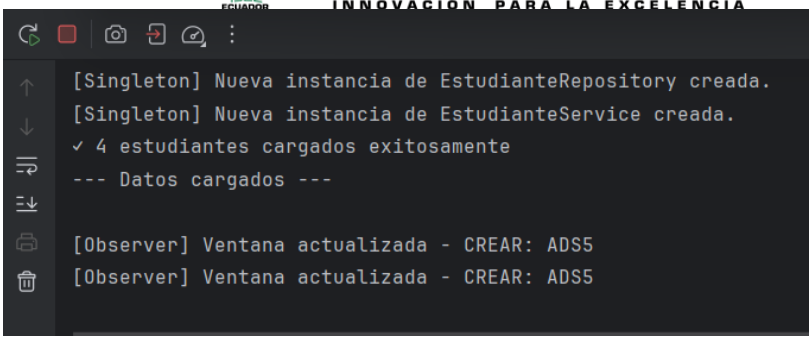
3. **Ejecutar el CRUD garantizando una única lista compartida y notificaciones automáticas a las vistas.**

Cada operación CRUD —crear, editar y eliminar— se ejecuta sobre la misma lista compartida debido al Singleton en EstudianteRepository y también en EstudianteService.
Cuando un estudiante cambia, el servicio notifica a todas las ventanas registradas, gracias al patrón Observer. Como resultado, si existen varias interfaces abiertas, todas se actualizan simultáneamente sin necesidad de recargar la tabla manualmente.
Proceso:

- Dos ventanas abiertas simultáneamente
- Se creó un estudiante en la ventana 1
- La otra ventana se actualiza automáticamente



- Consola mientras ocurre esta creación donde se observa la lista compartida, notificaciones automáticas y el observer funcionando



4. **Comparar resultados del CRUD con MVC sin Singleton/Observer vs Singleton+Observer.**
   En una arquitectura MVC sin Singleton, cada ventana tendría su propio repositorio interno, generando múltiples listas independientes y datos inconsistentes. Además, sin Observer, cada interfaz debe actualizarse manualmente después de una operación CRUD, lo que provoca desincronización visual y pobre experiencia de usuario.
   En cambio, con Singleton+Observer:

   - Toda la aplicación usa la misma lista de estudiantes
   - Si un controlador modifica un dato, todas las ventanas se actualizan automáticamente
   - La información se mantiene consistente, sincronizada y centralizada
   - El Service controla la lógica y notificaciones, lo que mantiene limpia la interfaz

5. **Explicar el impacto en la persistencia de datos y comunicación entre componentes.**

   - Impacto en persistencia:
     - Solo existe una lista en memoria para estudiantes.
     - Todos los controladores y ventanas trabajan sobre la misma instancia.
     - Los datos permanecen consistentes mientras la aplicación está activa.
     - Evita duplicación de estudiantes, listas paralelas y estados corruptos.
   - Impacto en comunicación entre componentes:
     - Las vistas ya no consultan manualmente los cambios: reciben notificaciones automáticas.
     - El servicio actúa como mediador central, informando a todas las interfaces cuando ocurre un CRUD.
     - Se reduce el acoplamiento: la UI no depende del repositorio ni del resto de ventanas.
     - La aplicación soporta múltiples vistas sincronizadas sin esfuerzo adicional.

Actividad Integrada

Realiza un cuadro comparativo entre MVC + Singleton con Factory y Observer:

Aspecto	MVC + Singleton + Factory	MVC + Singleton + Observer
¿Qué problema resuelve?	Resuelve la creación inconsistente de objetos. Centraliza y valida la	Resuelve la desincronización entre vistas. Permite que múltiples interfaces se actualicen

	construcción de Estudiante, evitando errores, duplicación de validaciones y objetos mal formados.	automáticamente cuando cambian los datos, manteniendo coherencia global.
¿En qué capa se utiliza?	Se usa en la capa de Modelo/Factory y se invoca desde el Controlador antes de llamar al Servicio. Asegura que los objetos lleguen correctamente formados.	Se implementa en la capa de Servicio, donde se registra, notifica y gestiona a los observadores (vistas). Las vistas implementan Observer.
¿Cómo influye en el mantenimiento?	Facilita el mantenimiento porque todas las reglas de creación están en un solo punto. Si cambia el formato del ID, nombres o edad, solo se actualiza el Factory, no todo el código.	Mejora el mantenimiento porque elimina actualizaciones manuales entre ventanas. Permite añadir nuevas vistas sin modificar la lógica: solo se registran como observadores.
¿Cómo evita fallas de diseño?	Evita objetos inválidos, duplicación de código, validaciones repetidas y lógica de creación distribuida. El patrón previene inconsistencias en el modelo.	Evita duplicación de estados en distintas ventanas, inconsistencias visuales, refrescos manuales y acoplamiento directo entre vistas. Garantiza sincronización automática.

3. CONCLUSIONES

- La integración de Factory y Observer dentro de la arquitectura MVC permitió evidenciar cómo los patrones complementan responsabilidades específicas: Factory se encargó de estandarizar la creación de objetos y Observer de mantener sincronizadas múltiples vistas sin acoplarlas entre sí. Esto demuestra cómo el diseño modular incrementa la estabilidad de la aplicación.
- El uso del patrón Singleton resultó clave para asegurar que tanto el repositorio como el servicio manejen un único estado compartido. Gracias a ello, todas las vistas y controladores acceden a la misma información, evitando inconsistencias y permitiendo que el Observer funcione correctamente en tiempo real.
- Implementar ambos patrones en el mismo proyecto permitió apreciar la diferencia entre resolver problemas de creación de objetos (Factory) y problemas de comunicación entre componentes (Observer). Esto aclara que cada patrón tiene un propósito bien definido y que su combinación genera una arquitectura más mantenible y flexible.

4. RECOMENDACIONES

- Es recomendable documentar claramente los flujos de notificación entre el Servicio y las Vistas, ya que conforme el proyecto crezca, será importante identificar qué eventos existen y qué vistas reaccionan ante cada cambio. Esto ayudará a evitar comportamientos inesperados o notificaciones duplicadas.
- Para próximas mejoras, sería beneficioso extender el Factory para manejar diferentes tipos de estudiantes (por ejemplo, regulares, becarios, extranjeros), lo cual permitiría probar la escalabilidad real del patrón sin alterar la lógica del Controlador o del Servicio.
- Se recomienda considerar pruebas unitarias para las validaciones del Factory y para las notificaciones del Observer, con el fin de asegurar que los cambios futuros no rompan la creación estandarizada de objetos ni la sincronización entre ventanas. Esto incrementará la confiabilidad del diseño.

5. REFERENCIAS

Carvajal, L. (2019). Arquitecturas de software: principios, patrones y prácticas. Editorial Alfaomega.

Freeman, E., & Freeman, E. (2016). Head First Diseño de Patrones. O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2020). Patrones de diseño: Elementos de software orientado a objetos reutilizable. Addison-Wesley Professional.